A Framework for the Definition of a Generative Design Pattern

David Wilson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Computing and Engineering
The University of Huddersfield

Supervisor
Dr Gary Allen
Dr Adrian Jackson

June 2008

To Helen xxx

Helen, thy beauty is to me
Like those Nicean barks of yore,
That gently, o'er a perfumed sea,
The weary, way-worn wanderer bore
To his own native shore.

On desperate seas long wont to roam
Thy hyacinth hair, thy classic face,
Thy Naiad airs have brought me home
To the glory that was Greece,
And the grandeur that was Rome.

Lo! in yon brilliant window-niche
How statue-like I see thee stand,
The agate lamp within thy hand!
Ah, Psyche, from the regions which
Are Holy-Land!

'Edgar Allen Poe'

DECLARATION

In presenting this dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the University of Huddersfield, I declare that this work has not been submitted for a degree at this or any other University, and that unless otherwise stated, it is entirely my own work.

David Wilson

June 2008

ACKNOWLEDGMENTS

First and foremost I would like to thank my supervisor, Dr. Gary Allen for his advice, support and feedback throughout this project.

I would also like to thank Professor Lee McCluskey and Dr. Christopher Newman for their support, advice and occasional feedback.

I am grateful to the University of Huddersfield and the academic staff in the Department of Computing and Engineering for their support and occasional advice, and to the technical department for providing the resources on which to develop my research work.

Much of my time was spent in the main researchers office in the Computing and Mathematics building, and I would like to extend my appreciation to my fellow researchers for making my stay there a pleasant one.

Most of all, I would like to thank Dr Adrian Jackson, who provided me with the opportunity and funding to study for this project and the opportunity it gave me to pursue a career in academia.

My deepest appreciation goes to my wife Helen, who's support throughout my degree and this research project has been invaluable. I would like to thank her for her time, her financial support and for her unwavering commitment when times were hard.

And finally, I would like to thank my children Kelly, Skye and Ailsa just for being there when I needed them.

ABSTRACT

Conventional design patterns found in many pattern catalogues are *static* components of reusable design knowledge. They are fully descriptive of the problems they will solve, but the descriptive knowledge and design they provide does not describe how they can work with other patterns in a design and development process. Therefore, the contention of this thesis is that the knowledge contained within static design patterns is inadequate for the purpose of applying the patterns to generate a software architecture with the intention of developing software systems.

The focus of this research has been the investigation of Design Patterns and their potential contribution to a *generative* development pattern language. Generative design patterns are active and dynamic: they describe how to create something and can be observed in the resulting systems they help to create.

To this end, a framework is presented that identifies the notational qualities that can be applied to a design pattern for the benefit of implementing architectural design. The impracticality of static design patterns for architectural design is addressed by revising the standard design pattern with a notation that describes the pattern as a generative component. The notation required for this revision is abstracted in part from the rich set of design notations and knowledge contained within:

(a) the quality driven processes contained in development methods that contributed to the now standard Unified Modelling Language (UML),

(b) the descriptive content of two distinct pattern classifications

i Design Patterns: Elements of Reusable Object-Oriented Software[45],

ii A Catalogue of General-Purpose Software Design Patterns[104] and

(c) a known study of relationships between design patterns

i Relationships Between Design Patterns[119].

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Generative programming is a concept familiar to software engineers and is an ideological goal for software development. Generative programming has attracted a considerable amount of research and development over the years culminating in a number of sophisticated CASE tools. Czarnecki[37] defines generative programming as:

> *A software engineering paradigm based on modelling software system families such that given a particular requirements specification, a highly customized and optimized end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.*

Czarnecki goes on to define a *Generative Domain Model* that consists of a problem space, a solution space and configuration knowledge, which maps the two together.

- The solution space consists of the implementation components with all possible combinations. The implementation components maximize compatibility, maximize reuse and minimize redundancy.
- The problem space consists of the application-oriented concepts and features that are required to fulfil a specification.
- The configuration knowledge specifies default combinations, illegal combinations, development rules, dependencies and optimizations[37].

The concept of generative programming maps adequately to the concept of generative design patterns, which have a problem / solution pair held together by the context in which the pair can be applied (the configuration knowledge).

The configuration knowledge is particularly useful in that there may be default combinations, illegal combinations and specific rules that need to be applied to any given combination of patterns.

However, in the scheme of life-cycle development, programming is edging towards the output phase of development. Prior to development comes analysis and design, yet generative design has attracted less research and development than generative programming. The amount of research into generative design through design patterns is extremely limited by comparison to design patterns in general.

The designs pattern found in software engineering (An example design pattern can be seen in Appendix I) are analogous to those described in architecture by Christopher Alexander[2, 3, 4]. Patterns, like those defined by Alexander, are described as being generative, mainly because they will generate structures. That is, a collection of patterns can be brought together to create a new structure. Software design patterns, like those of Alexander, are a development principle that contains the knowledge of experts who have used recurring design constructs in development projects. These experts have recorded necessary information about these patterns for others to use in their own development projects. However, although these design patterns are abstracted from application design, they are not described in such a way that they can be used to design applications. Expert knowledge describes them as being constructs that can be slotted into an application, but the knowledge to do that is missing from the patterns. The fact is that design patterns cannot be used to generate the architectures from which they are abstracted. They do not describe how pattern A will collaborate with pattern B. They do not describe how separate patterns may share resources or design components. There is a need to introduce additional knowledge into design patterns to empower them with the ability to generate systems. The majority of design patterns used in software development are static, they describe a problem that exists, but do not describe much beyond their own environment — they will mention a relationship to other patterns but little else. In other words, they are not adequate for generating new environments. Appleton[6] provides a simple account of generative and non-generative patterns:

> *Generative patterns are active and dynamic: they tell us how to create something and can be observed in the resulting system architectures they helped shape. Non-generative patterns are static and passive: they describe recurring phenomena without necessarily saying how to reproduce them. We should strive to document generative patterns because they not only show us the characteristics of good systems, they teach us* how *to build them*!

As such, the use of static design patterns as a means of developing systems is problematic, in that the design pattern is static and needs to be generative. Therefore, the aim of this thesis is to contribute to the methodology of software systems development by introducing into design patterns design knowledge that facilitates communication between separate design patterns.

Fundamental to this aim is the assertion that:

Generative design patterns will assist in improving software design when compared to using static design patterns.

In support of this assertion, both generative and static design patterns are compared and assessed in a number of application development case-studies. From the case-studies a further assertion is made that:

Generative design patterns provide a more efficient software solution to that of static design patterns when communication between separate design patterns is required.

This assertion is supported by the metrics calculated in comparative studies on generative and static design patterns. The metrics confirm that there is an overall improvement in systems design and software efficiency for the generative patterns examined.

The main contribution to this thesis, and to generative design patterns, is a notation that has been abstracted from a range of pattern styles. A means of pattern classification has been included in the generative pattern description to identify their contribution to systems functionality. Problem solving notation has been added to the patterns to identify appropriate development contexts in which the patterns can be applied. And finally, relational notation has been added to the generative pattern to identify how separate patterns will communicate.

The re-engineered notation has been applied to four different design patterns as examples of how to use the notation, and are included within the thesis in Chapter Six and Appendices A, B, C and D. An example of a static design pattern from the Design Patterns[45] catalogue can be seen in Appendix I.

This thesis is organised as follows:

Chapter Two considers software development methods, which have been studied as a means of finding qualities that could be applied to the notation of a generative design pattern. Although there are large numbers of different development methods, they all have a common goal and that is a quality product. Many use similar techniques to achieve their goal whilst others use alternative techniques or are specific to a particular phase in the development life-cycle.

Chapter Three looks at pattern notation with a view to understanding where the notation comes from and how it is applied in specific types and styles of patterns. Because software design patterns are not defined as generative, there are no guidelines on how to document them as generative. Therefore, in order to find a suitable document notation for generative design patterns, multiple pattern notations are explored for clues to a quality driven design process.

Chapter Four considers the functionality and the relationships between patterns. Different types of patterns have different functions; therefore some property of the pattern needs to describe the relationship that exists between different functional types. Some of these patterns form the body of a system whilst others perform some operational requirement of the system. Whatever type of pattern they may be, all different types need to define how they collaborate.

Chapter Five looks at the modelling notation that is used within patterns. Many software patterns use only a class diagram in the notation of a pattern although there are many modelling notations in the Unified Modelling Language that could be used to help describe the usability of a design pattern. Quite often, pattern writers who reproduce the patterns originally described in the Design Patterns[45] catalogue will modify the design notation in their interpretation of the pattern to meet the needs of their example. However, what is evident from some of these interpretations is that the example code they provide does not match the design notation that they use.

Chapter Six integrates the work described in Chapters Two to Five. The work from the previous chapters is summarised and a generative pattern is defined from the desired notation and the requirements of the defined relationships. Three separate examples of generative design are provided with different configurations of design. Further examples are provided in the Appendices.

Chapter Seven evaluates the approach taken in defining a generative pattern, reflecting on the construction process used for the framework and the examples that were produced to support the generative pattern concept.

Chapter Eight considers the work still to be done in the area of communication between specific classifications of patterns. A customised Computer Aided Software Engineering tool is proposed as well as some alternative research that can be conducted in relation to generative design patterns.

Chapter Nine concludes the work undertaken in defining a generative pattern.

Appendix A provides an example of a Composite combines Command design pattern.

Appendix B provides an example of a Composite combines Builder design pattern.

Appendix C provides an example of a Builder combines Command combines Composite design pattern. This pattern also combines with the Composite pattern and is also an example of a Creational, Behavioural and Structural pattern working together.

Appendix D provides an example of a Builder uses Command design pattern.

Appendix E describes the Relationship Trees from the hierarchy of classifications. The hierarchy includes only the patterns defined in the Design Patterns[45] catalogue.

Appendix F contains the source code for the generative patterns described in Chapter Six.

Appendix G describes the metrics that are available for assessing software quality.

Appendix H contains three sets of paired patterns that have been used in the evaluation process of generative design patterns.

Appendix I contains an example of a static design pattern from the Design Patterns[45] catalogue.

Chapter 2

EXPLORING THE LINK BETWEEN SOFTWARE DEVELOPMENT METHODS AND PATTERNS

2.1 Introduction

In this chapter an overview of software development methods is presented with the purpose of exploring the expert knowledge and quality driven aspects contained within the methods. Consideration is given to development methods as a means of determining if expert knowledge contained within methods can be appended to the expert knowledge contained within design patterns. From this study it has been determined that there are similarities between methods and patterns. It is found that there are qualities in some methods, mainly design aspects and coding principles, which can be used to enhance the quality of a design pattern.

In Section 2.2 a representative selection of early Object-Oriented development methods is listed, offering an historical insight into the evolving practice of quality driven software development. Several of the methods listed represent the driving force behind the Unified Modelling Language[15, 54] (UML), elements of which feature in design pattern notation. The authors of the original methods that contributed to the UML are seen as experts in the field of software engineering and the methods they devised have been a significant influence in the development of modern methods. The same experts that devised the UML made a significant contribution to IBM's[1] Rational Unified Process[89] (RUP), a contemporary development method, which is explored further in Section 2.3.

Section 2.3 examines several contemporary methods: RUP, Extreme Programming[9, 10] (XP) and Scrum[12] as well as the Object Management Group's[53] (OMG) Model Driven Architecture[55] (MDA).

2.1.1 Pattern / Method Analogy

The principle of a software development method is to impose discipline, predictability and efficiency on a project. In most cases, the methods that are used to develop a software system often follow some life-cycle process, which in many cases will expand upon the subjects of Analysis, Design and

[1] International Business Machines

Development. By following these methods, which are often defined by experts in their field, who have tried, tested and refined the processes that facilitate the effectiveness of the method, the likelihood of project failure is often reduced. In this respect, there is a simple comparison that can be applied between a method and a design pattern:

- Expert knowledge: design patterns are the documentation of expert knowledge.
- Failure reduction: design patterns are tried and tested examples of quality design.

The concept of software design patterns and the expert knowledge that they contain are described in greater detail in Chapter Three.

By examining design pattern catalogues and the patterns contained within, it is not obvious to the reader that the patterns contain simple methodical principles of software development. The simple reason for this is that patterns are not methods and no evidence has been obtained to suggest that they were ever intended to be methods. However, the analogy is there. For example:

- Many patterns contain the sections Problem and Forces, which analyse the situation in which a pattern can be applied – (Analysis).
- In many software design patterns there is often some form of design using a class, sequence and/or other diagrams – (Design).
- And, quite often there will be implementation details in the form of sample code – (Development)

Working on this principle, it can be seen that methods and patterns share some common ground and in taking advantage of this common ground it is conceivable that methods and patterns could work effectively as a unified subject in the field of software development.

2.2 Object-Oriented Software Development Methods

With the emergence of the object-oriented software paradigm came many object-oriented development methods. In the period 1988–1995 at least 19 object-oriented methods had been proposed in book form and many more were proposed in conference and journal papers[115]. To abstract good practice for the assimilation of methodical processes into design patterns a thorough review of all such methods

could be applied in order to obtain the best and most appropriate aspects of these methods. However, this line of research would be extensive and would detract from the main purpose of defining a generative pattern. In addition, during this period 1988–1995, many studies were conducted into the state of object-oriented development methods and comparisons made between them — A Comparison of Object-Oriented Development Methodologies[13] by Berard of The Object Agency[1] lists seven such studies whilst being a study in its own right. Brighton University[107] documents significantly more comparative studies, which itself includes a list of development methods. Therefore, conducting yet another comparative study is unlikely to provide any new and usable information that would be of beneficial use within the current research program. However, by examining previous studies, an insight into some of the more common and popular development methods has been established.

The list below has been constructed primarily from the list presented by Berard[13], although not in its entirety, and represents some of the early, more common object oriented methods — ascertained through their repeated inclusion in comparative studies.

- Object Modelling Technique[97] (OMT). OMT was originally created as a method for developing object-oriented systems. It uses many of the design techniques that became part of the UML.

- Object-Oriented Software Engineering[63] (OOSE). OOSE is very similar to OMT and employs Use-Cases to drive design. OOSE Became one of the key components of the Unified Modelling Language.

- Object-Oriented Analysis and Design[14] (OOAD). OOAD concentrates on the analysis and design phases but exemplifies the processes with existing applications. OOAD represents a good example of applying expert knowledge — a common theme in design patterns. OOADS is another design-oriented method that evolved into the Unified Modelling Language.

- Berard Object-Oriented Method[100] (BOOM). BOOM is a set of integrated methodologies such as OOSE and OOAD among others.

- Business Object Notation[113] (BON). BON was designed to work seamlessly with the programming language Eiffel[80] and has been used successfully with other programming languages.

- Object-Oriented Analysis, Object-Oriented Design, Object-Oriented Programming [23, 24, 25] (OOA, OOD, OOP). OOA / OOD / OOP covers the principles of object-oriented technology through basic life-cycle processes of Analysis, Design and Programming.

- Shlaer-Mellor[101]. Shlaer and Mellor devised an OOA / OOD method to compensate for the perceived deficiencies in the structured analysis and structured design techniques that were being used in the late 1980s, such as SSADM[47] and YSM[117, 118].

- Wirfs-Brock[116]. Wirfs-Brocks method is a design process that can be applied to both object-oriented and non object-oriented development.

- Fusion[28]. Fusion is an object-oriented analysis and design method that integrates features from existing methods such as OMT and OOAD.

Several of these methods focus on design and therefore have a strong design content, which is a prime feature of design patterns. Three of the methods listed above have between them received over 2200 known citations[86] and were the forerunners of the Unified Modelling Language[15, 54] (UML) namely, OOAD, OOSE and OMT. It is design elements from these traditional methods that have been applied to design patterns. However, only limited elements have been utilised in a design pattern, namely, class and interaction diagrams.

As a result of continuous development and revision some of the above listed methods have evolved into contemporary working practices that can be found in methods such as the RUP. Agile methods such as XP can also be considered as contemporary but these too have long rooted histories[26]. Although not based on Object-Oriented methods such as those above, Agile methods were a reaction to rigid, heavyweight methods of the day[44], which often adapted the lifecycle framework of the Waterfall model devised Royce[96]. The RUP and other contemporary methods are discussed in section 2.3 below.

2.3 Contemporary Software Development Methods

2.3.1 Rational Unified Process

About the Rational Unified Process

The Rational Unified Process is a life-cycle process that provides a disciplined approach to assigning tasks and responsibilities within a development team. Its aim is to ensure the development of quality driven software that meets the requirements of end-users[62, 67]. The RUP provides every team member with access to a knowledge base. By having all team members access the same knowledge base, irrespective of whether a team member is working with requirements, design, testing, project management, or configuration management, the process ensures that all team members share a common view of how to develop software. Rather than focusing on the production of documentation, the

RUP emphasizes the development and maintenance of models. That is, the Rational Unified Process is a guide on how to use the Unified Modelling Language, which was developed by the same team that created the RUP. Like design patterns, which will be discussed in Chapter Three, the RUP is the result of expert knowledge and, as will be shown in the following text, defines a number of similarities between the two concepts and contains modelling elements that can contribute to an appropriate notation for generative design patterns,

Four Phases of the Process

The Rational Unified Process attempts to capture what is considered to be best practices in modern software development within a four phase strategy:

- Inception phase
- Elaboration phase
- Construction phase
- Transition phase

Inception

During the inception phase a business case for the system is proposed and the scope of the project is identified. In this, all external entities with which the system will interact are established (actors). The interaction with external entities involves identifying prominent use cases and describing those that will have a significant impact on the system[62, 67]. The outcome of the inception phase is, among other things:

- A document of the project's requirements, key features, and main constraints. (*Analysis*)
- An initial use-case model. (*Design*)
- An optional domain model. (*Design*)
- One or several prototypes. (*Implementation*)
- A number of project plans and business related models.

Elaboration

The purpose of the elaboration phase is to analyze the problem domain, establish an architecture, develop the project plan, and eliminate high risk elements of the project. Architectural decisions have to be made with an understanding of the whole system: its scope, major functionality and non-functional requirements such as performance requirements.

At the end of this phase, the analysis and design aspects are considered to be complete and decisions are made on whether or not to commit to the construction and transition phases. While the process must always accommodate changes, the elaboration phase ensures that the architecture, requirements and plans are stable, and risks have been assessed.

In the elaboration phase, an executable prototype is built in one or more iterations, depending on the scale of the project, which at minimum should address the critical use-cases identified in the inception phase. Whilst a prototype of a production-quality component is always the goal, one or more throw-away prototypes may be produced as a means of testing design and requirements trade-offs.

The outcome of the elaboration phase is:

- A use-case model where all use cases and actors have been identified, and most use-case descriptions have been developed.
- Identification of supplementary requirements that are not associated with specific use-cases.
- A software Architecture.
- An executable prototype.
- A revised risk list and business case.
- A development plan.
- An optional user manual.

Construction

During the construction phase, all remaining components and application features are developed and integrated into the product, and all features are thoroughly tested. The construction phase is a process where emphasis is placed on managing resources and controlling operations to optimize costs, and quality.

Often, projects are large enough that concurrent construction plans can be implemented. These parallel activities can hasten the availability of deployable releases; however, they can also increase the complexity of resource management and workflow synchronization. This is one reason why the balanced development of the architecture and the plan is stressed during the elaboration phase.

The outcome of the construction phase is a product ready to put in the hands of its end-users. At minimum, it consists of:

- A software product configured for desired platforms.
- User manuals.
- A description of current releases.

Transition

The transition phase is concerned with placing the software into the hands of the users. Once the product has been given to the end user, issues usually arise that require the team to develop new releases, correct problems, or complete any features that were postponed.

The transition phase is entered when a product is sufficiently robust that it can be deployed in the end-user domain. This typically requires that a prototype of the system has been completed to an acceptable level of quality and that user documentation is available. This includes:

- Testing; to validate the new system against user expectations.
- Parallel operation with a legacy system that it may be replacing.
- Conversion of operational data stores.
- Training of users and those involved with maintenance.
- Roll-out the product to the marketing, distribution, and sales teams.

Typically, this phase may include several iterations, including beta releases, general availability releases, maintenance releases and enhancement releases. At this point, effort will be put into developing user documentation, user training, user support in product use, and reacting to user feedback. User feedback is usually confined to product tuning, configuring, installation, and usability issues.

The primary objectives of the transition phase include:

- Achieving self-support from the user.
- Achieving stakeholder agreement that deployment requirements are complete and consistent with the evaluation criteria.
- Achieving a final product as rapidly and cost effectively as practical.

This phase can range from being simple to complex, depending on the product. For example, a new release of an existing desktop product may be very simple, whereas developing a nation's medical records system would be very complex.

Conclusion

According to DeMarco[38], "*Analysis is the study of a problem, prior to taking some action*" - a familiar concept in terms of design pattern notation in that a problem is identified and a solution provided. According to Coad[24] Analysis is a process of extracting system requirements from the major stakeholders in the system under development. Therefore, the main concern of analysis is to determine what is required in order to develop the system that is being commissioned. On investigating the Rational Unified Process it can be seen from the authoritative texts that the process is heavily weighted towards analysis with emphasis on analysing the business processes involved in systems development. However, the RUP does not dwell heavily on problems but concentrates significantly on a generic solution to quality driven systems development. In this respect there are few similarities between the RUP and design patterns as there are in many of the early Object-Oriented methods such as OOAD, OMT and OOSE, which contributed towards the UML. However, the concept of Solution, which is a significant aspect of RUP, is also a significant aspect of design patterns, which presents in one aspect a similarity between this particular method and design patterns.

What can be seen in the RUP, which stands out significantly against other aspects of the process is the use-case. The originators of the RUP have put great emphasis on the use-case, which is used throughout the early stages of process in most aspects of the analysis and design lifecycle. This is backed up by the authors of the RUP who describe the Unified Process as being "Use-Case Driven, Architecture-Centric, Iterative and Incremental[62]. In this respect, the authors of the method are putting architecture and how they realise that architecture at the forefront of the RUP. One of the key aspects of the generative pattern is architecture, in that collaborating patterns can be used to define the architecture of a software system. From this, one can consider that there is a correlation between design

goals in RUP and the design goals of generative design patterns. This notion is supported by the Unified Software Development Process[62] (USDP), an early version of RUP, that looks at architecture from various viewpoints. This aspect is similar to one of the quality aspects contained within the Pattern Oriented Software Architecture[20] (POSA)catalogue of design patterns in that alternative views of a solution are considered. Given this correlation between these quality aspects of RUP and the POSA design patterns it is conceivable that multiple views of generative design should be incorporated into generative design patterns. The dynamic aspects of the POSA design patterns and how these aspects can be incorporated into a generative design pattern will be discussed further in Chapters Three and Five.

2.3.2 Agile Methods

About Agile Methods

A criticism of early object-oriented methods described them as being too bureaucratic. As a reaction to this criticism a number of new methods appeared. These new methods are referred to as lightweight or agile methods[44]. The new agile methods are a concession between no process and too much process, providing just enough procedure to gain a reasonable compromise. The result is that agile methods have some significant changes in emphasis from lifecycle methods. One of the core aspects of agile software development is the use of light but sufficient rules of project behaviour and the use of human and communication-oriented rules[26]. One of the most visible aspects of this is that they are less document-oriented, usually emphasizing a smaller amount of documentation for a given task. According to Fowler[44], they are rather code-oriented: following a route that says the key part of documentation is source code. whilst some methods can be quite rigid, agile methods encourage flexibility in their procedures. What may be suitable for one project may not be the right process for every project or situation. Therefore, the agile team is encouraged to refine and reflect as it goes along, constantly improving its practices in its local circumstances.

Extreme Programming (XP)

About XP

In the early days of XP the method was defined with the distinct section headings of Problem, Solution and Implementation — common notation used in many design patterns. The method looked at the problems of software development, proposed some solutions to those problems and described how to implement the method. There were four values, fifteen basic principles, and twelve practices[9]. In the contemporary version of XP the distinct aspects of Problem, Solution and Implementation are removed

in name whilst the core aspects of values, principles and practices that underpin the method have been redefined. The very basic XP paradigm of adaptation and change has been applied to XP itself[10]. There are now five values, fourteen principles, thirteen primary practices and eleven consequential practices. Of the twelve original practices, two have been abandoned, which gives the revised method fourteen new practices with which to apply the method[10]. In fact, whilst the newer version of XP retains its original values, the whole method has been refined in terms of its principles and its practices. Just as it is expected that patterns will evolve, and as will be shown they can evolve into generative patterns, XP has evolved and may continue to evolve, which demonstrates a similarity that patterns have with this particular method. The original concept of XP was divided into three founding sections:

1. **Problem:** where the values and principles of XP are explained and activities defined.

2. **Solution:** where good practices are applied following the guiding values and principles.

3. **Implementation:** how the strategies discussed in the solution can be put into practice.

Although these concepts are removed in name, the underlying essence of the method is still evident in that the basic content of the method that underpinned these three sections is still evident. References are still made to problems but the emphasis of where the problem lies has been redefined. When once the problem was defined in terms of where weaknesses may be evident in the development process, the problem is now defined in terms of a developer's inability to cope with change[10], and the solution is XP itself. The solution in regards of XP begins by first understanding the core concepts of the method which are represented by values, principles and practices.

Five Values

The basic root elements of XP are five core values that are deemed to be strategically important for the successful development of software. These core values are guidelines for XP as a method and a focal point for development itself. The first four values are retained from the original XP, and respect is added as an additional value. The values for the contemporary version of XP are:

1. **Communication:** Most problems and errors are caused by lack of communication.

2. **Simplicity:** The main guideline is to keep the system simple and do not plan too far ahead. New features, when needed, can be added to a simple system with greater ease.

3. **Feedback:** Feedback is seen as being an important component of communication in that when you communicate you are in a position to gain feedback. Feedback also contributes to simplicity in that the simpler a system, the easier it is to get feedback about it.

4. **Courage:** If fear is expressed about a project, then the burden of tackling the project becomes much bigger. However courage alone is not enough to tackle a project and should be backed up by communication, simplicity and feedback.

5. **Respect:** If members of a team do not care about each other and their work, the chances of development failure are much greater.

The five values that support XP as a method do not give specific advice on how to manage a project, or how to write software. To this end, what are required are practices. However, bridging the gap between values and practices are principles[10].

Fourteen Principles

1. **Humanity:** Software is developed by people for people, so human factors are taken into consideration in attempting to deliver quality software.

2. **Economics:** Ensure that what is being developed has business value, meets business goals and meets business needs. Someone has to pay and they want value for money.

3. **Mutual benefit:** All activities should benefit both developers and clients alike, both in the present and in the future

4. **Self-Similarity:** Try copying the structure of one solution into a new context, even at different scales.

5. **Improvement:** XP asks for excellence in software development through continuous improvement.

6. **Diversity:** Teams should include a variety of skills attitudes and viewpoints in order to identify problems and provide solutions.

7. **Reflection:** An effective team should ask themselves how they are working, and why they are working in that way. They need to analyze the reasons behind success or failure without hiding their mistakes and learn from them.

8. **Flow:** The practices of XP assume a continuous flow of software by engaging in all activities simultaneously, rather than a sequence of discrete phases.

9. **Opportunity:** Problems must be seen as an opportunity for learning and improvement.

10. **Redundancy:** Critical and difficult problems should be solved in several different ways. Thus, if one solution fails, another solution may prevent a disaster.

11. **Failure:** Failure should not be viewed as failure but an opportunity for learning. Failure is not a waste if it imparts knowledge.

12. **Quality:** Sacrificing quality is not an effective means of control. Projects do not go faster by accepting lower standards. In addition, team members need to do work they are proud of.

13. **Baby Steps:** One of the reasons behind baby steps is that a small step in the wrong direction is easier to recover. A big step that fails can damage a project. It is more prudent to proceed iteratively in baby steps. Baby steps do not mean proceeding slowly. A team proceeding in baby steps can take a lot of them in a short period of time.

14. **Accepted Responsibility:** Accepted Responsibility is about being responsible. Responsibility should only be taken if you are confident enough to accept it.

Principles are a means of providing a better understanding of practices and to improvise complementary practices when a practice cannot be found for a given purpose. They also give a better idea of what the practice is intended to accomplish[10].

Twenty Four Practices

The updated version of XP defines thirteen primary practices, and eleven corollary (consequential) practices. The primary practices must be applied first, and each of them may add to an improvement in the software development process. Consequential practices require expertise in primary practices, and may be difficult to apply without first having considered the primary ones. All twenty four practices are an integral part of the method, and should be fully applied in order to obtain the maximum benefit of XP.

Thirteen Primary practices

1. **Sit Together:** The working environment should be an open space that is able to host the whole team.

2. **Whole Team:** A team should be composed of members that have all the skills necessary for the project to succeed.

3. **Informative Workspace:** The workspace should be supplied with information on the status of the project and the tasks to be performed.

4. **Energized Work:** The team must respect a work – life balance, so that they can focus on their job and be productive.

5. **Pair Programming:** Code should be written by two team members at one workstation.

6. **Stories:** The system should be described using short descriptions of functionalities that are accessible to the customer.

7. **Weekly Cycle:** At the beginning of the week a meeting should take place where the functionalities (Stories) to develop in the week are chosen by the customer.

8. **Quarterly Cycle:** Development is planned on a lager time scale. This considers feedback on the team, the project and what progress is being made.

9. **Slack:** Avoid making promises that cannot be fulfilled. Consider tasks that can be dropped if the plan falls behind schedule.

10. **Ten-Minute Build:** The build and testing of a system should only take minutes.

11. **Continuous Integration:** Teams should be integrating changes regularly.

12. **Test-First Programming:** Before updating or adding code, tests should be written in order to verify the code.

13. **Incremental Design:** XP is opposed to producing a complete design prior to development and suggests that design should be done incrementally during coding.

Eleven Corollary Practices

1. **Real Customer Involvement:** Stakeholders who are affected by the system must became a part of the team. They can contribute to quarterly and weekly planning.

2. **Incremental Deployment:** When replacing a system, start by replacing some of the functionality and gradually replace all the system.

3. **Team Continuity:** Development teams should remain intact throughout several projects.

4. **Shrinking Teams:** As a team becomes more productive, gradually reduce its size, sending free members to form new teams.

5. **Root-Cause Analysis:** When a defect is detected, find the causes of the defect and eliminate them.

6. **Shared Code:** Any member of the development team must be able to change any part of development at any time.

7. **Code and Tests:** Code and tests are permanent artefacts and have to be preserved.

8. **Single Code Base:** There should be only one version of the system. Temporary systems can be created but must not be preserved.

9. **Daily Deployment:** New software should be put into production every night. A gap between what is on a programmer's desk and what is in production is a risk.

10. **Negotiated Scope Contract:** Contracts should be written for software development that have fixed time, costs and quality, but call for an ongoing negotiation of the scope of the system.

11. **Pay-Per-Use:** The customer usually pays for each release of the software.

According to Beck[10], the primary and corollary practices are not everything that is needed to successfully develop software. They are however, core elements of excellence in software development. If a problem arises that is not covered by one of the practices then one should look back at the values and principles to come up with a solution.

Conclusion

What is evident with XP and particularly the updated version is that there is a strong emphasis applied to the management of the method. The method does not go into detail about analysis or design and how to conduct these aspects of software development. Application of knowledge and how to apply such aspects as analysis and design, as XP relates, is in the hands of the experts who are following the method.

One thing that stands out in the original version of Extreme Programming[9] is the emphasis that was put on defining the method in terms of Problem and Solution. In this respect, XP had very obvious similarities with design patterns, in that Problem and Solution is representative of the expert knowledge contained within them.

Another significant aspect of XP in both new and older versions is that coding is seen as a key activity of the method. This, from the point of view of the authors of the method, is one of the strengths of XP. What will be shown in Chapter Six is that the generative design pattern will utilise this strength to demonstrate coded examples of generative design. The concept of coded examples is supported by the concluding comments on RUP in Section 2.3.1, where it is noted that multiple views can be applied to a generative pattern. In this respect, multiple views refer to multiple coded examples.

Scrum

About Scrum

Scrum is an agile process that can be used to manage and control product development using iterative and incremental practices. The method is capable of producing a set of functioning artefacts at the end of every iteration. Scrum facilitates the development of the best possible software from the available resources with acceptable quality within required release dates. Product functionality is delivered at the end of what is known as a sprint, which may last between fifteen to thirty days, depending on the size of the project. As requirements and design are evolving so the product will evolve. The name Scrum refers to the scrum in rugby – a tight formation of forwards who bind together in specific positions when a scrum down is called.

Roles

There are three primary roles in the Scrum development process:

- **The Scrum team:** The team normally consists of 5-9 people. The team members decide how the work is arranged and how assignments are distributed. There are no set project roles, everyone should be able to swap tasks with another member. The team is self-organized and the members have a joint responsibility for the results.

- **Product owner:** The product owner represents the customer and ensures that the Scrum Team is working effectively from a business perspective. The Product Owner administers a Product Backlog, a to-do list, where all the specifications for a product are listed and prioritised. Before each Sprint, the highest prioritized goals are transferred to a Sprint Backlog. The Product Backlog is visible to the whole organization so that everyone is aware of what to expect in future releases

of the product.

- **Scrum master:** The Scrum Master meets with the team every day in brief meetings known as daily scrums. When someone from outside the project has an issue to discuss with the team, the Scrum Master ensures that the team are disturbed as little as possible in their work. After each Sprint, the Scrum Master holds an evaluation meeting with the Scrum team, during which experiences and conclusions are reviewed. The purpose of the evaluation meeting is to raise the teams level of knowledge and strengthen motivation prior to the next Sprint.

The Scrum Process

- **Creating a backlog:** The Product Owner compiles requests and specifications that are the basis of the product, such as any new functionality or bug fixes. After goals have been defined, the specification is broken down into chunks of work achievable in a sprint. The Product Owner makes a to-do list arranged according to how market demands and customer requests may change over time and decides in what order any changes should be made and delivered. Each sprint should create in-part a working sub-section of the product. When it is time to start a new Sprint, the Product Owner freezes the leading items on the to-do list and summons the Scrum Team to a meeting.

- **The sprint phase:** Of the Sprints 15 to 30 days period, the first one or two days are set aside to create a Sprint Backlog. When the tasks have been determined, the Product Owner releases work to the development team. From that point, the team works under its own responsibility. If the team has been properly composed, the work should be self organising.

- **Daily Scrum:** Every day, usually at the same time in the morning, the Scrum Master and the Scrum Team have a brief meeting. The purpose is to try and eliminate any restrictions that may have developed within the group. Each of the participants should in some way answer three questions:

  1. What have you done since the last meeting?
  2. What will you do between now and the next meeting?
  3. Is there anything preventing you from doing what you have planned?

  The first two questions give the participants an insight into how the project is progressing. The third question provides a basis for problem solving that may range from damaged resources to

organizational changes at the company. Anyone may attend and listen at the meeting, but only the Scrum Master and the team members may have some input.

- **Demonstration and evaluation:** Each Sprint finishes with a demonstration of functioning software. Attending at the demonstration will be the Product Owner, users and possibly representatives of corporate management. This is in effect an evaluation meeting and the starting point for the next Sprint.

Conclusion

Like XP, the focus of Scrum is significantly directed towards facilitation of the method and the activities of the team, whilst leaving the processes of analysis, design and development in the hands of the experts that are using the method. This practice can be seen in most of the agile methods. Because this and other similar methods are more concerned with their own processes they have little to offer in terms of expert content that can be appended to a design pattern.

2.3.3 Model Driven Architecture (MDA)

About MDA

The Object Management Group's Model Driven Architecture is a standards driven process to build systems from models using model transformations. A complete MDA specification consists of a platform-independent model (PIM), one or more platform-specific models (PSM) and a set of interface definitions, each describing how the PIM is implemented on a different platform. MDA development looks at the functionality and behaviour of a system, independent of the platform or platforms on which it will be implemented. Thus, it is not necessary to repeat the process of defining a system's functionality and behaviour when new platforms or technologies are developed. With MDA, functionality and behaviour are modelled only once[55].

The whole ethos of the MDA is to design an architecture and generate an application or system from that architecture. Therefore, MDA models must be extremely detailed: the application will be generated from it, and will include only those functional components that are explicitly represented in the model. MDA works by separating the business logic of an application (the code that implements its functionality) from the infrastructure in which it is deployed. Once captured, the business logic can be reused in other ways with other applications, as long as they adhere to the standards. The MDA approach captures business logic in reusable models that are written in a standard modelling language, such as UML. These models form the metadata describing the structure and characteristics of a system.

The metadata is then used by the MDA tools to generate and deploy the application.

MDA Development Life Cycle

The MDA development life cycle is not much different from the traditional life cycle of many development methods. Requirements are gathered and analysed, a design is created, code is written and the system is tested and deployed. The major difference lies in the nature of the components that are created during the development process. The components are formal models that can be understood by computers[66]. Figure 2.1 below illustrates the MDA development lifecycle.

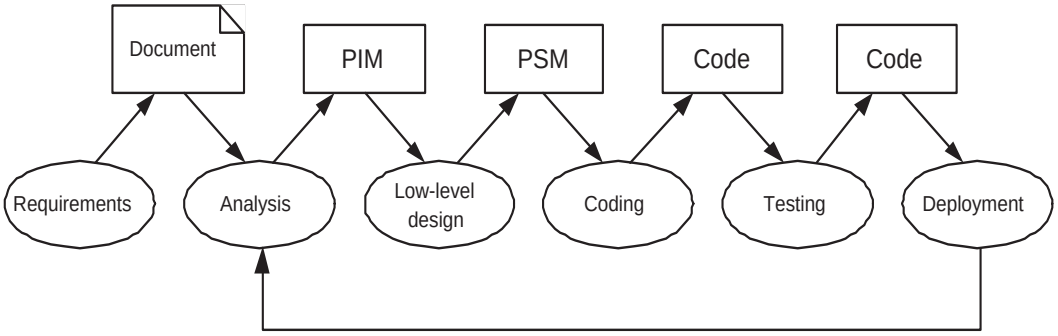

Figure 2.1: MDA Development Lifecycle[66]

The formal models of the MDA are:

- PIM - describes a software system that supports some business.
- PSM - for each specific technology platform a separate PSM is generated.
- Code - each PSM is transformed into code that fits the platform technology.

The PIM, PSM and Code are shown in Figure 2.1 as artefacts of different steps in the development lifecycle and represent different abstraction levels in the system specification.

Models

The UML contains both static and dynamic modelling notation and can be used to provide static and dynamic views a software system. However, MDA makes no distinction between static and dynamic models. MDA regards different diagrams in UML as being a view of the same model, if they are all written in the same language. That is, the MDA will make no distinction between a rectangle that represents a static class in a class diagram and a rectangle that represents an object instance of a class in an interaction diagram. Models in MDA are not restricted to UML, for example, a Petrinet or ER

model could be used to describe a system[66]. If a particular modelling language is not capable of defining a specific aspect of a system then more than one model will have to be used to define the system.

Transformations

The MDA process as described in Figure 2.1 is very similar to traditional development where transformations from model to model or model to code are done by hand. With MDA the transformations are done by tools. Transferring a PSM to code is nothing new, there are several very sophisticated, and not so sophisticated, tools on the market that will do this (Together[16], Visual Paradigm[85], Rational Rose[61]). What is new is transferring PIMs to PSMs. Figure 2.2 below shows the three major steps in the MDA transformation process.

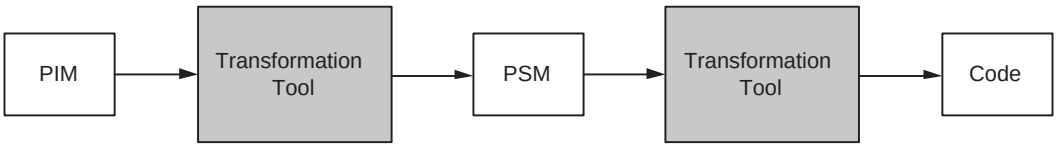

Figure 2.2: MDA Transformation Process[66]

A transformation tool takes as input a PIM and returns as output a PSM. A second transformation tool, or the same tool depending on the level of sophistication, transforms the PSM to code. Within the tool(s) there is a transformation definition that describes how the model should be transformed.

Conclusion

What stands out about MDA is that it is driven by design, but more significantly it uses tools to transform designs and generate code. MDA is not so much a method, but a process to be used in generating systems and any well-written modelling language can be used to model a system. A meta-modelling language is used to transform models to models and models to code. So long as the meta-modelling language is well-written in the same language as the model, a model can be transformed by the transformation tool. MDA itself makes no distinction about how to analyse or put together a model of a system, it leaves that up to the expert. However, MDA can be used with the RUP or other agile methods such as XP. Indeed, because changing a model means changing the software, the MDA approach helps support agile software development[66].

MDA is a step closer to the utopian goal of generating systems from reusable components but is working mostly at the generative programming level as discussed by Czarnecki[37], and not at a level of generative design, which is proposed in this thesis. For MDA to work, someone has to create an initial

design for a proposed system, which is then fed into a transformation tool. A tool designed to build architectures from generative design patterns could be used to create the models that are fed into a transformation tool.

2.4 Summary

In the study on methods it is shown that a method is a way of using an ordered set of instructions to select and apply a number of techniques and tools to identify and analyse a problem and construct a solution to that problem. It is also shown that there are numerous development methods from which to choose. Many of the methods viewed represent what their authors see as being good software development practice. The study provided a summary of those methods that gave birth to the standard design notation, the UML. What is evident in many of the older Object-Oriented methods is that the design notations from these methods, and the UML, are not used to their fullest extent within design patterns, particularly state, object interaction and use-case diagrams. The summary on object-oriented methods found that there are significant similarities between patterns and methods. Both Object-Oriented methods and patterns have analysis, design and implementation details contained in their documentation. With modern agile methods the similarities between patterns and methods is not as strong as it is with the older methods. Where a pattern has analysis, design and implementation detail contained in its documentation, which follows the lifecycle detail of some older methods, modern methods do not concern themselves with how to analyse or design a system.

However, modern methods do have something to offer in providing quality aspects for a generative pattern. The following points represent practices from these contemporary methods that can be used to document generative patterns:

- Section 2.3.2 comments on developers who found older methods too bureaucratic. As a result, many of the modern development methods are code-oriented rather than document-oriented. This aspect can be applied to a generative design pattern in that discussions about the Problem and Solution aspects of a pattern could be kept to a minimum. More emphasis can be put into design and implementation, rather than analysis. In this respect, generative patterns could move towards a more graphical notation than textual notation – however, this aspect is an issue for further investigation.

- One aspect that stands out with agile methods is source code. Again, as commented upon in Section 2.3.2, many of the practices of these agile methods concentrate on practices that support

coding. Borrowing from this, more emphasis could be put into using source code to demonstrate the usability of a generative pattern.

- Providing more examples of source code supports use of the quality aspects of USDP/RUP discussed in Section 2.3.1 in that different views of an architecture can be used to demonstrate the generative concept of a pattern. In this respect, several different examples of source-code could be used to explain how several patterns can work together.

- Design is a key aspect of design patterns and is used to emphasise the structure of a pattern, yet design is used sparingly in many design patterns. The Rational Unified Process makes considerable use of design techniques, particularly the use-case diagram. From this it can be considered that the use-case is a useful modelling aspect that can be used in a design pattern. The use-case can be used to illustrate a business aspect that is being demonstrated in a source code example of collaborating design patterns.

One of the functional aspects of agile methods is flexibility, in that methods can be adapted to meet the needs of different project situations. Based on the idea of thinking in terms of different project situations and adapting to those, which accounts for some of the flexibility of agile methods, it is conceivable that the same generative design patterns can be redefined with alternative examples to cover a specific software domain. For example, patterns that are aimed at desktop applications can be defined with a different set of examples to patterns that are aimed at, for instance, client / server applications – again, this aspect is an issue for further investigation.

Chapter 3

UNDERSTANDING DESIGN PATTERN NOTATION

3.1 Introduction

Within the design pattern community there have come several stylistic forms of patterns — the most common being the style used in the Design Patterns[45] catalogue. This format is often referred to as the GoF Format (Gang of Four), which is a reference to the four authors of the Design Patterns[45] catalogue. Another format is referred to as Alexandrian Form — the style of pattern written by the Architect Christopher Alexander[2], whose pattern language has inspired much of the growth in writing design patterns. Yet another form, and one that is often used in non-software patterns and patterns that are only discussed in brief, is the Portland Form, which is purely narrative.

How patterns are written is only one factor in understanding the nature of the pattern itself. This chapter is an exploration of design patterns, explaining their origins, their purpose and their distinctions. In attempting to understand the nature of design patterns, four different pattern concepts are discussed: Idioms, Design Pattern Catalogues, Pattern Systems and Pattern Languages. Also within this chapter, the rationale behind pattern notation and how that notation should reflect the context in which the pattern is described is discussed. Knowing why a pattern is described in a given way is important for describing new pattern types or refactoring existing patterns.

This chapter continues with a look at how elements of good practice from within a diverse range of design pattern types and styles can be abstracted for the benefit of defining a generative pattern. A selection of pattern writers from different software disciplines and pattern concepts is selected for study. From this study the most fundamental notation is determined and selected as being the type of notation that can be used in a generative pattern without cluttering the pattern with unnecessary detail.

3.2 Patterns in Object-Oriented Software

3.2.1 The Pattern Concept

The current use of the term 'pattern' within the software community is popularised from the writings of the architect Christopher Alexander[2, 3, 4] who wrote several books on the topic of patterns in urban

planning. Although these books are ostensibly about architecture and urban planning, many of the concepts captured therein are applicable to many other disciplines, including software development[6]. Alexander proposed that urban development should be based on a collection of reusable patterns. In the software domain, collections of patterns can be categorized by their structure and intent. Based on structure and intent, a Pattern System is different to a Pattern Catalogue or Pattern Language, which are defined by their specified relationships expressed within the pattern collections.

Each pattern described by Alexander represents a single element in a hierarchy known as a pattern language. Alexander's notion of a pattern is that a pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice[3]. This indicates that a pattern is not a fixed entity and will provide, if required, a unique solution. What this implies is that the patterns can be modified to suit individual needs without losing the essence that is central to the pattern.

3.2.2 Idioms

Whilst design patterns describe general structural problems, idioms are less portable when viewed at the level of a programming language. Idioms are the lowest level of abstraction in a pattern classification. Because idioms are at a low level of abstraction they are specific to a programming language. They describe how to implement particular components, their functionality, and their relationships to other components in the language itself. They may also depend upon, or represent, features that are not present in other programming languages. For example, the pointer mechanism in C++ that has no corresponding feature in the Java programming language. Because idioms are at the lowest level of abstraction and deal with source code, they represent a link between design and implementation.

3.2.3 Pattern Catalogues – (Design Patterns)

A pattern catalogue is typically a collection of related patterns. It subdivides the patterns into separate categories and may include some amount of cross-referencing between them[6]. *Design Patterns: Elements of Reusable Object-Oriented Software*[45] is a benchmark example of a pattern catalogue and typifies the concept of the Design Pattern in software.

The motivation for design patterns and/or the pattern catalogue is the concept of software reuse. A software design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating reusable object-oriented systems[45]. In a pattern language, the patterns

are organised by the relationships between the patterns, whilst in a pattern catalogue the patterns are organised by some classification scheme[84]. The patterns in the Gamma[45] catalogue are divided into Creational, Structural and Behavioural. These are subdivided by scope as being Class or Object. The design pattern identifies participating classes and instances, their roles and collaborations and the distribution of responsibilities. The notation of the pattern describes when it applies, whether it can be applied in view of other design constraints and the consequences and trade-offs of its use[45]. The pattern provides graphical solutions using abstract modelling and exemplifies solutions with code fragments (which might be thought of as being equivalent to recommending the type of bricks and mortar to use in an Alexandrian solution). Unlike Alexander's pattern language, the Design Patterns catalogue was not without precedent. It follows Alexander's principles on patterns but adapts the genre for the software domain.

Gamma's pattern catalogue consists of 23 patterns, which conform to a thirteen-point structure:

Rule	**Description**
Name	A name by which the pattern is known
Intent	The purpose of the pattern
Also Known As	A pattern of a similar nature but with a different name
Motivation	A scenario that illustrates the design problem
Applicability	The situations in which the pattern can be applied
Structure	A standard modelling notation, e.g. UML
Participants	The different classes and objects involved in the design
Collaborations	How the participants collaborate
Consequences	The way in which the pattern supports its objectives
Implementation	Pros, cons, hints, techniques, language specific issues
Sample Code	An illustration of how the pattern may be implemented
Know Uses	Where the pattern has been applied in the real world
Related Patterns	Other patterns that can be used in combination with this one

Table 3.1: Design Patterns' Notation[45]

3.2.4 Pattern Systems

A pattern system (system architecture) is an extended concept of the pattern catalogue, but is one step removed from the completeness of the pattern language. Some of the patterns defined in a pattern system link together to form sequences, similar to those found in a pattern language, whilst other patterns within the system have no direct relationship to any other pattern. Therefore, those patterns that have no relationship with other patterns within a pattern system represent an individual solution to a problem within the confines of that architectural concept.

Although pattern languages are thought to be complete, they are not created complete; they evolve over time from pattern systems. Likewise, a pattern system may evolve over time from a pattern catalogue[6], indicating that some element of refactoring may take place within the patterns of a catalogue.

The concept of the pattern system is a cohesive set of related patterns that are organized into groups and subgroups. A system describes the inter-relationships between patterns and groups of patterns and how they may be combined to solve more complex problems. The patterns in a pattern system need to cover a sufficiently broad base of problems and solutions to enable significant portions of complete architectures to be built[6].

A pattern system is significantly similar to a pattern language in terms of the relationships between patterns. However, a pattern language requires that its constituent patterns cover every aspect of its given domain. For example, in some given software domain a pattern language for that domain is computationally complete: at least one pattern must be available for every aspect of the construction and implementation within that software domain — that is, there must be no gaps or blanks[20]. Whereas, in a pattern system the patterns described may only cover certain aspects of the given domain — that is, in some given software domain, that domain will not be computationally complete.

The pattern system described by Buschmann[20] separates patterns into two categories: those that will create a system architecture and those that stand alone as design patterns. These patterns are then sub-classified by their intent. Buschmann follows both Alexander's and Gamma's principles on patterns, adapting the genre for the system architecture domain.

Buschmann's pattern system conforms to the following structure:

Rule	**Description**
Name	A name by which the pattern is known
Example	An example of where the pattern is used
Context	A situations to which the pattern applies
Problem	A description of the problem
Solution	A brief description of how the solution is achieved
Structure	A complete description of the components used, and any models that may aid in describing components
Dynamics	A number of scenarios that illustrate behaviour
Implementation	Guidelines for implementing the pattern. May be supplemented with abstract or concrete code examples.
Example Resolved	A discussion of the implementation
Variants	Similar situations where the pattern can be used
Known Uses	Where the pattern has been applied in the real world
Consequences	The way in which the pattern supports its objectives
See Also	References to related patterns

Table 3.2: Buschmann's Pattern Notation[20]

In Buschmann's early writings on patterns[19] he had some alternatively named categories (See Table 3.3 on the following page), thereby indicating an evolutionary process in how patterns are written and described.

This evolutionary process is indicative of the aim set out in this research program in that standard design patterns will be provided with the additional structure, which will allow the patterns to evolve into generative design patterns.

Rule	Description
Rationale	The motivation for developing the pattern
Applicability	When to use the pattern
Classification	A pattern is classified according to its properties
Description	Participants and collaborators in the pattern and the responsibilities and relationships to other patterns
Diagram	A graphical representation of the pattern's structure
Methodology	The steps for constructing the pattern
Discussion	A discussion of the constraints in applying the pattern

Table 3.3: Buschmann's alternative categories of Notation[19]

3.2.5 A Pattern Language

A pattern language can be described as being more than just a collection of patterns. The pattern language written by Alexander explains how patterns should be applied to a greater problem than the problem solved by a single pattern. Alexander's book, A Pattern Language, also says that no pattern should be an isolated entity[3]. Each pattern can exist in the world only to the extent that it is supported by other patterns i.e. the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded within it[3]. A popular cliché may suggest that the pattern language is greater than the sum of its parts. For instance, any small sequence of patterns from this language is itself a language for a smaller part of the environment, i.e. a subset of a higher order of the language. This small list of patterns is then capable of generating, as Alexander says, a million different elements of that environment[3]. When patterns are put together in this way they can create an infinite variety of combinations and, therefore, an infinite variety of solutions.

Alexander also says that each pattern is a three-part rule[3], which expresses a relation between a certain context, a problem and a solution. With this three-part rule we can look at the patterns from a language in two ways:

1. As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occur repeatedly in that context, and some process that allow these forces to resolve themselves. The forces are the goals that are desired when applying the pattern. For example, the study of algorithms in computer science, where the main force to be resolved is efficiency or time complexity[70].

2. As an element of a language, a pattern is an instruction that shows how this configuration can be used over and over again, to resolve the given system of forces, wherever the context makes it relevant[2].

Alexander's pattern language consists of 253 patterns, all conforming to the following seven point structure:

Rule	**Description**
Name	A short meaningful name which may be an indication of the solution
Picture	An archetypical example of the solution
Problem	A set of forces that occur in a given context
Context	Recurring situations to which the pattern applies
Solution	Rules applied to resolve the given forces
Diagram	The solution in the form of a diagram
Related patterns	Higher/Lower order patterns which connect to the given pattern

Table 3.4: Alexander's Pattern Notation[3]

The structure of patterns, the methods and the processes surrounding them are not exclusive to architectural design. The interrelationship that exists between context, problem, forces, and solutions, makes Alexander's framework an ideal basis for capturing other kinds of design knowledge.

In Coplien's "A Generative Development-Process Pattern Language"[29], a pattern language that can be used to shape a new organization and its development processes, is also defined by seven rules. However, there is no specific rule for graphics, although graphics may appear within the occasional pattern. Coplien separates the forces that define the problem from the problem itself. He also introduces a context that results from the pattern after it has been applied, and a set of reasons for using the pattern, described as a rationale (See Table 3.5 on the following page).

Rule	Description
Name	A short meaningful name
Problem	The problem in brief
Context	Recurring situations to which the pattern applies
Forces	A set of forces that apply to the problem
Solution	Rules applied to resolve the problem
Resulting Context	The result of applying the pattern
Design Rationale	Reasons for using the pattern

Table 3.5: Coplien's Pattern Notation for a Generative Development-Process[29]

The patterns introduced by Coplien are inspired by Alexander's language and principles. Indeed, some of the patterns in Coplien's pattern language are refinements of Alexander's communication and organizational patterns. For example:

- The philosophy of establishing stable communication paths across the industry has strong analogies with the Alexandrian patterns that establish transportation webs in a city (Web of Public Transportation[3]). Here, the concern for Coplien is the transportation of information between individuals and groups.

- Many of the organization patterns are refinements of Alexander's circulation patterns which define the higher-order pattern (Circulation Realms[3]). This inspired the pattern "Shaping Circulation Realms", which acts as a building block for other patterns in Coplien's language.

Pattern languages are generative in nature in that the patterns that a given language contains generate systems or parts of systems, or will shape the system architecture in which they are used[30]. Coplien uses the English language as an analogy in which he says the English language can generate all possible papers in conference proceedings, so a pattern language can generate all sentences in a given domain[30]. That is, the letters of an alphabet work together to form words, a collection of words form sentences; sentences form paragraphs and so forth. Viewed in this way, the pattern language works in the same way as natural language.

Non-generative patterns, such as those from a pattern catalogue, are static and passive. They may make references to other patterns or may be related in some way but they are not dependent on other patterns, they do not generate architectures, and they only provide a solution to a problem in a given area.

3.2.6 Design Pattern Structure

Because design patterns are primarily a communication tool, written within the confines of a specific concept, it is important to have a more or less standard way of describing them[64]. However, many stylistic variants of Alexander's pattern description are possible. Some are written in a literary style like Alexander's, whilst others favour a more detailed approach used in Design Patterns[45]. Other patterns may adopt a totally different structure. The attribute shared by all these pattern structures is just that - structure[109]. The most popular format is that used in the Design Patterns catalogue[45] illustrated in Table 3.1.

The pattern forms that exist in software differ by the kind of template used to emphasize their message, although most forms contain the basic categories: *name, problem, statement, context, description of forces, solution and related patterns*[52], interspersed with elements specific to the pattern form. However, a comprehensive structure for a pattern format should provide: *a description of best practices, appropriate generality, evidence that the pattern recurs, scope, constructiveness, completeness, utility, examples, appropriate level of abstraction, lack of originality, appropriate name and clarity*[70]. These elements are not necessarily headings to be included within a pattern template, but represent elements that contain the overall communication criteria for a well-defined pattern.

Quite often, the differences in pattern types, such as Architectural patterns, Design Patterns or Idioms are in their corresponding levels of abstraction[6]. That is, the need to describe the level of detail required for a certain pattern type. For example, a higher level pattern such as those for software architecture require more detail than lower level patterns such as idioms because an idiom is already specific to a given area and only needs to describe its appropriateness to that area.

A pattern needs to convey a message relating to its context in the real world and an important step in defining an appropriate pattern structure for any given pattern concept is the identification of a Target Audience[79]. Once an audience has been identified, patterns can be written for that audience with an appropriate pattern structure. For example, patterns written for the target audience of catalogues such as Pattern Oriented Software Architecture[20, 99], (POSA) or Analysis Patterns[42] could have a slightly different structure to patterns written for the target audience of the Design Patterns[45] catalogue.

Meszaros and Doble[79] have written a number of patterns to assist in writing effective patterns. In this they define a number of issues, described in Table 3.6, relating to the content of a pattern or pattern type.

Pattern	Force
Mandatory Elements Present	Not all patterns require the same kinds of information to be effectively communicated. Capturing all elements regardless of need only clutters many patterns. For a pattern to be truly useful, it must have a minimum set of essential information. These information elements are required to allow patterns to be found when required and to be applied when applicable. If the necessary elements are missing, it becomes much harder to determine whether the pattern solves the reader's problem in an acceptable way. There is no single correct style or template for patterns; trying to impose one could stifle creativity and get in the way of effective communication Readers expect certain information to be present in a pattern. This is what differentiates a pattern from a mere problem/solution description.
Optional Elements when Helpful	All patterns do not require the same kind of information to be effectively communicated. Capturing all elements regardless of need only clutters many patterns.

Table 3.6: Meszaros' Criteria on Pattern Structure[79]

The forces within the pattern "Optional Elements when Helpful" reiterate the first item of the forces within "Mandatory Elements Present", indicating that a pattern should convey a finite amount of information but can be extended with elements acceptable within that pattern's domain, when required to convey additional information.

Patterns in the form adopted by Gamma, Buschmann, and other pattern writers are much longer than Alexander's so, although they share a common literary style, they provide a more detailed selection of concrete information. Table 3.4 represents the pattern structure proposed within architecture for describing towns and buildings, whilst Table 3.5 modifies that structure to describe patterns relating to a development process. For software and architectural-software patterns, whose structures are described in Table 3.1 and 3.2, the template (the structure of the pattern) is redefined to include implementation details. The design pattern template used by Gamma serves to be more descriptive than generative. However, design patterns of Gamma and other pattern writers could be redefined to make them generative[11].

A pattern needs to balance between providing sufficient and insufficient understanding. If room is left for interpretation, then different readers may interpret the same pattern in different ways, or may see the pattern as being part of some other language. The implication for the structure of patterns suggests that the structure or template for the pattern is subject to the intended use of the pattern[92].

Therefore, in order to create generative patterns from standard design patterns a template appropriate to the intended use of the pattern is required, which should reflect the generative process of the pattern.

3.2.7 Narrative Form (Portland)

All pattern forms are a narrative; they are a written description of the knowledge and experience of experts in the field. Several forms of pattern writing have already been mentioned above whilst discussing different types of patterns. The Design Patterns[45] catalogue and the POSA[20, 99] catalogues by Buschmann use what has become known as the GoF Format and which is used extensively in one variation or another. Organizational Patterns by James Coplien[29], discussed above, use the Alexandrian form.

One pattern form that has not been discussed is Portland Form[35, 56], named as such because the originators of the form come from Portland, Oregon, USA. Portland Form, unlike other forms, is a pure narrative and is often referred to as a Narrative Form. Also, unlike other formats, the Portland Form does not use a full-featured pre-defined template with specified headings to discuss the knowledge contained in a design pattern. The writer of a pattern that is written in a Narrative Form may choose to write a pattern in a set layout, and all patterns written by that person may follow that layout. However, a different pattern writer may choose to write with a totally different format in the narrative style. Some patterns of the narrative style are written as a step-wise account of utilizing the knowledge within[60], whilst others, such as the Checks Pattern Language[34, 31] and the Caterpillars Fate Pattern Language[65, 31] are written in a few paragraphs describing how to go about some task. Within patterns of this type, there may or may not be one or more headings that relate to a popular definition of a pattern - the headings being Problem, Solution, Context as well as several other headings. Woolf has used such headings in his Smalltalk ENVY/Developer[110] pattern language.

What can be seen in patterns of this style is a lack of fine detail. What is often being written about patterns can, in most cases, only be described as an overview or abstract of what could be contained in the pattern. However, many of the patterns written in this form are patterns for defining some form of process. The Caterpillar's Fate Pattern Language by Kerth[31] is a pattern language for making the transformation from Analysis to Design. In patterns of this type there will often be some element related to a software process but there is no real software involved.

Whilst patterns of this style provide good reading material for defining a software process, there are several catalogues that have applied the style to software components. Software catalogues that are written in the Narrative Form retain the singular discussion, but will often exemplify the discussion

with diagrams and in some cases with code. The discussion of each pattern is based on the motivating factors of the pattern itself and any sub-headings that may be written into a pattern are relative to that pattern only. For example, in the J2EE Design Patterns catalogue by Crawford and Kaplan[33] the pattern *Service to Worker in J2EE* has the following sub-headings:

- Models and Views
- Actions
- The Dispatcher
- The Front Controller

In the same catalogue, the pattern *Composite View* has a completely different set of sub-headings:

- The Composite View Pattern
- Implementing Composite Views
- Reusing the Front Controller and Dispatcher
- Building the Custom Tags
- Using Templates

The patterns from this catalogue contain code, tips and one or more diagrams. A notable feature of this catalogue is that the patterns often contain more code than discussion. However, in most cases, the discussion does cover vital aspects of why the pattern is useful and what it will achieve, although how it will achieve its goal is not discussed in fine detail, particularly in respect of other patterns, which is consistent with the discussion in most patterns.

The EJB Design Patterns catalogue by Marinescu[74] also embraces the Narrative Form with mostly written discussion, small snippets of code and the occasional diagram. Although there are no set categories of discussion as there is in the Alexandrian or GoF formats, Marinescu's discussion of patterns does have some structure.

Most of the patterns by Marinescu conform to the following structure:

- Name
- Identify a need
- Promote a question
- Discuss the problem relative to the question
- Bullet point issues raised from the discussion
- Promote a solution
- Discuss the solution
- Bullet point the benefits
- Close the discussion

Not all the patterns have this exact structure; some contain more of the structure than others, and in varying levels of detail. For example the pattern *Stored Procedures for Auto generated Keys* has a lengthy discussion of the problem but does not bullet point the issues raised from the discussion. The *Universally Unique Identifier for Enterprise JavaBeans* (UUID for EJB) pattern has a short discussion of the problem with no issues bulleted, a lengthy discussion of the solution and no closing comment. The basis of this structuring is the identification of a problem and the formulation of a solution, which represents the constituent parts of a popular definition of a pattern (Solution, Problem, Context, Forces). Although the patterns in this catalogue provide a reasonable discussion with some useful information contained within the content of that discussion, the detail in this catalogue does not match up to the discussion provided in other catalogues such as Core J2EE Patterns[5], which may be a result of the limited way in which patterns of the Narrative Form are discussed.

The catalogue *Server Component Patterns* by Völter[112] presents patterns in a very similar format to that of Marinescu. However, Völter has no specific points relating to issues raised by the problem or the solution. However, one interesting feature that Völter presents in his patterns is a cartoon drawing that summarises the pattern. It would be easy to dismiss this feature as irrelevant, but the cliché does maintain that 'a picture paints a thousand words' and the drawings do add some weight to the limited discussion of the patterns. Although the drawings add an interesting feature to the patterns it is not

always straightforward in making the connection between the drawing and the purpose of the pattern. When this is the case then the drawing is not adding significantly to the content of the pattern. If this type of feature were to be added to the definition of a pattern then it would have to be free of ambiguity, which would be difficult to maintain given the varying perceptions that people may hold on usefulness of design patterns.

3.3 Defining a Template

So far in this chapter a number of different pattern formats, styles and templates have been discussed, the purpose being to form an understanding of the pattern concept. From this understanding it is envisaged that a template for a generative design pattern can be proposed, the template being the different notational sections included for discussion of the generative pattern itself. As can be seen from this chapter, different people have different ideas for what they include in the patterns they are discussing. Each of these individuals and groups of individuals has their own justification for what they discuss in a pattern. This chapter has introduced only a small proportion of the stylistic variants that are available in the discussion of design patterns. For example, there are a number of Hypermedia[94, 95] design patterns that have a template similar to the GoF format as do many HCI[60, 105], User Interface[106, 114] and Multimedia[36] patterns.

However, as can be seen from the discussion above, the different templates used by various pattern writers incorporate many of the same named categories, which carry a similar discussion or have categories of a similar discussion, but are introduced under an alternative name. For example, the GoF template has the heading Structure whilst the Alexandrian template uses Diagram. Both headings have the same intent in that they produce a graphical representation of the solution. Völter's Server Component Patterns, although without named heading, uses a cartoon, which is equivalent to Alexander's Drawing component in his template.

Table 3.7 lists a selection of pattern writers and the notation that they use in describing patterns:

	Writers						
Notation	HCI [60, 105]	Hypermedia [46, 72, 94]	Multimedia [36]	Software [20, 45, 48, 102]	User I'face [27, 106, 114]	Web App [5, 50, 81, 103]	Other [3, 29]
Also Known As	-	-	-	[20, 45]	-	-	-
Applicability	-	[46, 72]	-	[45, 102]	-	-	-
Applications	-	[46]	-	-	-	-	-
Background	[60]	-	-	-	-	-	-
Bad Example	[60]	-	-	-	-	-	-
Benefit/Drawbacks	-	-	-	[102]	-	-	-
Collaborations	-	[46, 94]	-	[45]	-	-	-
Comments	-	-	[36]	-	-	-	-
Consequences	-	[46, 94]	-	[20, 45, 48]	[27]	[5, 50, 103]	-
Consider ...	[60]	-	-	-	-	-	-
Context	[105]	-	-	[20, 48]	[27]	[50, 81]	[3, 29]
Description	-	-	-	[102]	-	-	-
Design Rationale	-	-	-	-	-	-	[29]
Diagram	-	[46]	-	-	-	-	[3]
Dynamics	-	-	-	[20]	-	-	-
Example	-	-	-	[102]	-	-	-
Example (Graphic)	[60, 105]	[72]	[36]	[20]	[106, 114]	-	-
Example Resolved	-	-	-	[20]	-	-	-
Forces	[105]	-	-	[48]	[27]	[5, 50, 81, 103]	[29]
How	-	-	-	-	[106]	-	-
Implementation	-	[72, 94]	-	[20, 45, 48, 102]	-	[50, 81, 103]	-

Table 3.7: Varying Uses of Notation

Continued on next page.

Varying Uses of Notation continued:

	Writers						
Notation	HCI [60, 105]	Hypermedia [46, 72, 94]	Multimedia [36]	Software [20, 45, 48, 102]	User I'face [27, 106, 114]	Web App [5, 50, 81, 103]	Other [3, 29]
Introduction	-	-	-	[102]	-	-	-
Intent	-	[46, 72, 94]	-	[45]	-	[103]	-
Known Uses	-	[46, 94]	-	[20, 45]	[27]	[50]	-
Motivation	-	[46, 72, 94]	-	[45]	-	-	-
Notes	[105]	-	-	-	-	-	-
Participants	-	[46, 72, 94]	-	[45]	-	[5, 103]	-
Picture	-	-	-	-	-	-	[3]
Post-condition	-	-	[36]	-	-	-	-
Pre-condition	-	-	[36]	-	-	-	-
Problem	[60, 105]	[94]	[36]	[20]	[27, 114]	[5, 81, 103]	[3, 29]
Properties	-	-	-	[102]	-	-	-
Purpose	-	-	-	[102]	-	-	-
Rationale	-	[46]	-	-	-	-	-
Related Patterns	-	[72]	-	[45, 48, 102]	[27]	[5, 50, 81, 103]	[3]
Resulting Context	[105]	-	-	-	-	[81]	[29]
Sample Code	-	-	-	[45, 48]	-	[5, 50, 103]	-
See Also	-	-	-	[20]	-	-	-
Solution	[60, 105]	[72, 94]	[36]	[20, 48]	[27, 114]	[5, 50, 81]	[3, 29]
Structure	-	[46, 94]	[36]	[20, 45]	-	[5, 103]	-
Synopsis	-	-	-	[48]	-	[50]	-
Thumbnail	-	-	-	-	[27]	-	-

Table 3.7 Varying Uses of Notation - Continued

Continued on next page.

Varying Uses of Notation continued:

	Writers						
Notation	HCI [60, 105]	Hypermedia [46, 72, 94]	Multimedia [36]	Software [20, 45, 48, 102]	User I'face [27, 106, 114]	Web App [5, 50, 81, 103]	Other [3, 29]
Variants	-	-	-	[20, 102]	-	-	-
When to Use	-	-	[36]	-	[106, 114]	-	-
What	-	-	-	-	[106]	-	-
Why	-	-	-	-	[106, 114]	-	-

Table 3.7 Varying Uses of Notation - Continued

Although the list of notations described in Table 3.7 could be extended further by including notations used by all known pattern writers and known catalogues, it is felt that extending the table would not add to the goal of finding an acceptable list of usable notations for a generative pattern. Some of the notations that are displayed in the above table represent similar information but have been discussed under an alternative name by different pattern writers. For example, Problem, which is used by the majority of writers listed and Introduction, used by Stelting[102] are essentially the same — they discuss a recurring problem. Synopsis by Grand[48, 50] and Background by Hong[60] also share similar intent in their detail in that they provide an overview of the patterns in which they occur. Although different pattern writers use different headings in their notation, they are not providing dissimilar information in what they write under those specific headings. Therefore including all notations would only add to the task of filtering out related and lesser-used notations.

A unilateral decision could be made on what notation to include in a generative design pattern, however it would be useful to find out what different notations have to offer in the way of describing a design pattern. From this understanding of what can be described as good practice in describing patterns a decision can be made on what to include in the notation of a generative pattern. Some notations displayed in the above table are obscure and only used by a single pattern writer, whilst only a small minority of writers may use some notations. Other notations are used by a majority of writers and are therefore perceived as being an important factor in describing specific types of pattern. In the case of Solution and Problem, these are indicative of a common definition of a pattern — that it is a Solution to a Problem in a Context. Others such as Sample Code are indicative of patterns that can provide this type of information.

Two criteria can be used for filtering out related and obscure notation:

1. Amalgamate related information.

2. Remove obscure and seldom used notation.

From this position a portfolio of notation can be acquired that will serve as an appropriate source of information for describing a generative pattern. The notation that is left in the list will have been devised by experts, be used by multiple experts and used in a range of different pattern styles.

Firstly, related information can be amalgamated into a single notation. The reason for amalgamating notation first is that some amalgamated notation may still come under the criteria of being obscure and seldom used. For example, "See Also" and "Consider..." both represent similar information about the patterns they describe. "Consider..." is in a HCI pattern and "See Also" is in a Software pattern. Although both of the pattern types relate to software through design and development, they don't add to the general information of an individual design pattern — they are only directing the reader to some other pattern. Although they have been amalgamated, they can still come under the category of obscure and seldom used, as there is still only a small minority of writers who have used this type of notation.

For the purpose of clarity, any amalgamated notations are given the name of one of the notations from the group that is amalgamated. In all cases the most popular name has been chosen to represent the group of amalgamated notations. For example, in the group 'Consequences', Consequences is used by nine of the pattern writers, whilst different writers use the other notations in the group sporadically.

Table 3.8 represents the related notations that can be amalgamated into single notations.

Notation	Original Notation	About the Notation	Notes
Related Patterns	Consider ...[60]	Provides a list of patterns that complement the existing pattern.	The notation refers to other patterns that provide a similar solution. In the case of Consider... the notation provides a list of other patterns but does not indicate a collaboration or variation. Both Buschmann [20] and Stelting[102] have dual entries. It is felt that a variation of a pattern is a pattern in its own right that holds some connection to the existing pattern.
	Related Patterns [3, 5, 27, 45, 48] [50, 72, 81, 102] [103]	Names other patterns, which can be used with the existing pattern. More often names patterns that perform a similar function to the existing pattern	
	See Also[20]	A reference to patterns that solve similar problems, and to patterns that refine the existing pattern	
	Variants[20, 102]	Other patterns that are associated with or are variations of the existing pattern	
Intent	Background[60]	Some rudimentary information describing the need for the pattern.	The notations that come under Intent, as they are used by most writers, represent some form of introduction to the pattern. The brief descriptions that are used under this notation are providing an overview of the pattern relating to its purpose and what can be achieved through its use. Some writers have included information about the problem and several have briefly described a solution. Whilst most pattern writers provide just a few lines of introduction, Stelting[102], provides a deeper insight into the problem and solution
	Intent[45, 46, 72] [94, 103]	A short statement relating to design issues or problems that the pattern may address.	
	Introduction[102]	A description of the problem where the pattern might be used as a solution.	
	Purpose[102]	A short statement relating to design issues or problems that the pattern may address.	
	Synopsis[48, 50]	A brief description of the pattern that conveys the essence of the solution.	
	Thumbnail[27]	Brief notes on the problem and solution	
	What[106]	A short statement of design issues	

Table 3.8: Amalgamating Notation

Continued on next page.

Amalgamating Notation continued:

Notation	Original Notation	About the Notation	Notes
Example (Graphic)	Diagram[3, 46] Example[20, 114] [36, 60, 72, 106, 105] Picture[3]	A picture or sketch that shows the possible output from using the pattern. An example in the form of a graphic. Can be a sketch or a picture. A snapshot of a known example of where the pattern has been used.	Buschmann[20] has a written explanation of the example but always provides a sketch to back up his argument. Alexander[3] shows how the pattern can be applied and how it has been applied. Some writers provide an image but do not put it under a heading.
Comments	Comments[36] Example Resolved [20] Notes[105]	Additional comments about the pattern that have no place in the main notation Aspects about the pattern that are not covered in the main notation. A discussion about the environment in which the pattern can be applied	Tidwell's[105] Notes are more anecdotal whilst Cybulski's[36] Comments and Buschmann's[20] Example Resolved are more additional to the existing notation. Microsoft[81], uses Resulting Context to discuss benefits and liabilities.
Forces	Applicability[45] Forces[105, 48, 27] [5, 50, 81, 103, 29] When to use[36]	Situations in which the pattern can be applied. Considerations that lead towards a solution, often written as a number of one line statements. Situations in which the pattern is applicable.	In most cases, Forces can be seen as pre-conditions for the solution. An alternative view could see Forces as being a list of requirements. Therefore Forces could be seen as being a short form of requirements analysis. Analysis is a life-cycle feature that is mentioned in **Chapter Two** and included as part of the generative pattern in **Chapter Six**

Table 3.8 Amalgamating Notation - Continued

Continued on next page.

Amalgamating Notation continued:

Notation	Original Notation	About the Notation	Notes
Conse--quences	Benefits/ Drawbacks[102] Consequences[46] [45, 94, 20, 48, 27] [5, 50, 103] Rational[46] Why[106, 114] Resulting Context [105, 81, 29]	The consequences of using the pattern and issues that may arise from its use. Trade-offs, results and issues of using the pattern, including any benefits and drawbacks. Benefits of using the pattern. The benefits the pattern will bring. The expected results of using the pattern.	The various notations under Consequences are discussing the benefits and drawbacks of using the pattern. An alternative way of looking at this element of notation is to see it as a conclusion to the descriptive nature of the pattern. However, if Consequences is viewed this way then it should appear at the end of the pattern description. Germán[46] uses both Consequences and Rational.
Problem	Context[20, 48, 105] [27, 50, 81, 3, 29]	The situations in which a problem may exist, or a description of a problem addressed by the pattern.	There are different interpretations on what is defined as being Context and Problem. Grand[48] states directly that Context "*describes the problem that the pattern addresses*". Buschmann[20] uses one line of text to describe the context in which the pattern applies, and follows up with a long passage on the Problem. Grand[48] on the other hand uses Context to describe the whole problem and follows up with a significant discussion of Forces. Coldewey [27] and Microsoft[81] use a short passage for Context, which describe a number of problems, then describes the Problem as a question. Tidwell[105] also asks a question of the Problem but only has one or two sentences relating to the Context.
	Description[102]	A discussion of the pattern and/or the problem.	
	Motivation[46, 72] [94, 45]	A discussion of the problem and the situations in which the problem might occur.	
	Problem[60, 105] [94, 36, 20, 27], [29] [114, 5, 81, 103, 3]	A general discussion of a problem that may apply in a development process	

Table 3.8 Amalgamating Notation - Continued

Continued on next page.

Amalgamating Notation continued:

Notation	Original Notation	About the Notation	Notes
Participants	Collaborations[45] [46, 94] Participants[45] [46, 72, 94, 5, 103]	How the various participants in the pattern collaborate. Classes and Objects that participate in the pattern	Most pattern writers that use this notation give a brief description of the participants. Gamma[45] Rossi[94] and Germán[46] follow this up with a few sentences on how the participants collaborate. It is interesting to note that Stelting uses the notation, 'Implementation' to discuss participants.
Implementation	Implementation [81, 46, 72, 94, 103] [20, 45, 48, 102] Sample Code[45] [5, 50, 48, 103] Example[102] Dynamics[20]	Issues that may arise from implementing this pattern. May contain sample code to illustrate the issue. A coded example, often accompanied by implementation issues. A coded example, with instructions. Three typical examples of the pattern in use	Several pattern writers have separate sections for Implementation and Source Code. However, these notations are complementary as they both relate to the implementation of the pattern. It can also be observed that writers are often discussing the consequences of implementing the pattern. Stelting's[102] version of 'Implementation' is to discuss participants of the pattern. Buschmann uses Dynamics to discuss implementation details.
Known Uses	Applications[46] Known Uses[46] [20, 45, 27, 50, 94]	Some known applications of the pattern Some known applications of the pattern	Germán[46] uses Applications and Known Uses interchangeably.

Table 3.8 Amalgamating Notation - Continued

From the notations that have been collated, as used by the referenced collection of pattern writers, thirty-seven of the original forty-six notations, described in Table 3.7, have been amalgamated into a list of ten distinct notations. Each of the ten notations is derived from notations with similar intent and functionality and therefore can be stated as an individual component in a design pattern. It is felt that "Forces" and "Problem" are inexorably linked and as such, Forces could be stated or described as part of the discussion of the problem. Alexander[3], who is cited in most discussions on patterns, does not use the notation Forces in his patterns, although he does refer to the term forces on several occasions. The use of the notation Forces stems from early pattern writers who have discussed Alexander, and made

strong references to the term Forces in their description of Alexander's notation Problem. Lea[69], for example, states of Alexander's Problem notation, "*A description of the relevant forces and constraints, and how they interact*". Hence, several pattern writers have taken it upon themselves to separate out Forces from Problem, including Coplien[29] — Table 3.5. In an attempt to simplify patterns and rewrite them to be more accessible to novice users, Forces and Problem will be unified under the notation Problem.

From the original collection of pattern notations, as listed in Table 3.7, there are still nine notations to consider. Each of these nine notations is individual in nature and will not fit neatly into any of the notations of the amalgamated list. For example, Pre-Condition and Post-Condition could fit in with Comments, but Pre-Condition could also fit into Intent, Forces or Problem. Therefore it seems fitting to consider any remaining notation on its own merit. Consideration for the remaining notations is based on their frequency of use by the pattern writers. The criterion for retaining any given notation is that it is used by the majority of writers. Also taken into consideration at this point is the amalgamated notation. As stated earlier, there are two criteria for filtering out notation: similarity and obscurity. Some of the amalgamated notation is still obscure and therefore will be filtered out. There are exceptions to the rule for filtering out obscure notation. For example, Dynamics[20], although an obscure notation, is a quality driven concept of a pattern. For a pattern to be accepted as a pattern it has to have three known uses and Dynamics, as used by Buschmann[20], provides three examples of using the pattern.

Table 3.9 below lists the notation that is not considered for use in the future definition of a generative pattern:

Notation	About the Notation	Notes
Also Known As[20] [45]	Other names for the pattern.	Only used in the catalogues of Gamma[45] and Buschmann[20], two early pattern writers. Adds nothing to the pattern in question and attracts few entries in their catalogues, particularly that of Buschmann

Table 3.9: Rejected Notation

Continued on next page

Rejected Notation continued:

Notation	About the Notation	Notes
Bad Example[105]	Places where the pattern has been applied but is not user friendly	An obscure notation that is only used by one pattern writer. May be appropriate for HCI patterns, but not for software patterns. However, for software patterns this type of notation could be useful in an Anti Pattern
Design Rational[29]	Reasons for using the pattern.	Used by Coplien[29], Table 3.5, in his Development Process pattern language. Not strictly a software based notation. Could have been amalgamated with Comments, Intent or Forces as it provides some background information.
How[106]	A description of how the pattern could be used.	Used by only one pattern writer. It could be amalgamated with Implementation as it contains, among other information, instructions on how to apply the pattern.
Post Condition[36]	What the artefact to which the pattern has been applied should look like after application of the pattern.	Post Condition represents something of a conclusion, which all good documentation should have. However, this notation is only used by one pattern writer, and can be stated as part of the Implementation details.
Pre Condition[36]	Conditions that should exist before the pattern is applied.	Pre Condition describes what needs to be in place before the pattern can be applied. It is difficult to ascertain as to whether this notation should fit in with Intent, Problem, Forces or none of these notations.
Properties[102]	The purpose/classification of the pattern.	Used by only one pattern writer. See below

Table 3.9 Rejected Notation - Continued

Table 3.10 below lists the amalgamated notation that is not considered for use in the future definition of a generative pattern:

Amalgamated Notation (Rejected)	
Example (Graphic)	Used by almost half of the selected pattern writers but not significantly by writers where software code is being considered. Buschmann[20] uses a graphic to emphasise a point and Germán[46] uses a diagram to show participants, which are not, in most cases, class based components. Other pattern writers, are for the most part, showing a desired or finished product as a result of applying the pattern. It is therefore felt that an example graphic would not add significantly to a software based pattern.
Comments	Used by only a small number of the selected pattern writers. Comments are often only used to provide some additional background information. Any vital information about a pattern can be written into the main content of the pattern itself. In Buschmann's[20] case, where he uses Resulting Context, this can be seen as something of a Conclusion and can be built into the implementation details of a pattern or can be written into a conclusion to the pattern document. The notation Consequences is seen as something of a conclusion and would be an ideal place to fit any meaningful comments.
Known Uses	Known Uses is only used by a small number of the selected pattern writers. This, in the first instance, seems to be a problem in the description of some patterns in that this notation is not consistently used throughout pattern writing. Known Uses is one of the accepted principles of a pattern in that it should have as a minimum three known uses to be considered a pattern. However, some pattern writers use examples (graphics) to show where a pattern is being implemented — but not always three examples. As just indicated, Known Uses is a principle concept of a pattern and needs to be retained in some form. However, there are no known uses of generative patterns as they are presented in this thesis therefore, this notation is removed in name. However, the concept of three known uses will be presented under the name of Dynamics. That is, Dynamics, as will be used in a Generative pattern, will represent three examples of generative patterns rather than three known uses of generative patterns.

Table 3.10: Rejected Amalgamated Notation

From the original list of used notations described in Table 3.7, seven notations have been removed from consideration for use in a Generative pattern on the grounds that they are obscure. Of the nine notations that were created through amalgamation, three have been discarded on the grounds that the amalgamated notation is still obscure. However, there are some obscure notations including those not yet considered (Dynamics and Structure) that deserve special consideration.

- Dynamics[20], although an obscure notation, is a quality driven aspect of Buschmann's[20] design patterns. For a pattern to be accepted as a pattern it has to have three known uses and Dynamics, as used by Buschmann[20], provides three examples of using the pattern. It would be a fault in the notation to specify generative patterns in a dynamic way and not consider using Dynamics as a heading within that notation. Dynamics can be amalgamated with Implementation as indicated in Table 3.8 therefore, Dynamics will be used as the main notation for providing three different scenarios of how two or more patterns will work together, whilst Implementation will be the heading for the included source code.

- Structure, like Dynamics, comes under the category of obscure. However, Structure in software design patterns represents a design notation, often written using a modelling notation such as UML. One of the key aspects of a design pattern is design and although Structure is not used by the majority of pattern writers, a software design pattern is not a practical solution to a design issue without the design notation to demonstrate how the problem is being solved. Therefore Structure has to be included as an element of notation in a design pattern.

- Participants also comes under the category of obscure. However, it would be extremely difficult to discuss a software pattern and not discuss its participants. Therefore, the notation Participants needs to be included in a pattern in some form. In this respect, Participants is to be included in a pattern as a sub-heading within the Implementation aspects of the pattern - See Chapter Six.

- In the case of Properties[102], Properties is used by only one of the listed pattern writers, and is considered to be an overlooked aspect of a software pattern. The notation 'Properties' refers to the classification and scope of a pattern. Gamma[45] defines each of his patterns by classification and scope but does not discuss this under a notation. Although the use of Properties as a notation is extremely limited, classifying patterns is an important factor of defining good quality software design patterns. Again, although Properties will be filtered out by name, for a Generative pattern, the properties of a pattern will be revealed by the relationship that any given pattern has with another pattern - See Chapter Four.

The only notation not yet considered from the original forty-six notations listed in Table 3.7 is Solution. Solution is the most popular of the documented notations, used by fourteen of the nineteen listed pattern writers. Of the four software pattern writers, it is only used by two of them. It is interesting to note that Solution is not used in the Design Patterns[45] catalogue despite the authors indicating that a pattern has four essential elements, Solution being one of them. As the most popular of the notations

listed, Solution will be used in the definition of a generative design pattern. This links in with the Problem notation in that a pattern is often defined as being a solution to a problem – it is fitting therefore, that both these notations are to be used.

From the original list of forty-six there are a total of nine notations that represent a list of headings that can be adequately used to describe a generative design pattern. The list represents the most significant notations that are used by the majority of pattern writers in describing a design pattern and those notations that are not widely used.

Table 3.11 below represents the list of nine notations that will be used in defining a Generative pattern. The order of the list at this point is not representative of the format for a generative pattern. Formatting of the pattern will be considered in Chapter Six.

Notation	**Rational**
Consequences	A conclusion to the pattern document, highlighting any advantages or disadvantages of using the pattern. Certain issues may arise from using this pattern therefore they should be discussed at this point.
Dynamics	Three examples of generative patterns in use, with instructions on how to implement the pattern. Can be linked to Implementation to give a complete overview of how patterns are collaborating.
Implementation	Examples of source code showing how patterns work together. Elements of Structure (design) can be included to illustrate how the Participants of the pattern collaborate.
Intent	Intent represents an introduction to the pattern. It can describe the origins of the pattern or the need for the pattern itself. It may outline different Problem scenarios or a Solution that will come from using the pattern. A design pattern is a document of good practice and all good quality documents will or should come with an introduction. Intent can fill the place of an introduction in a design pattern.

Table 3.11: Accepted Notation

Continued on next page

Accepted Notation continued:

Notation	**Rational**
Participants	The individual components that participate in the defined structure of the pattern. Participants can be discussed within a pattern as a sub-heading of the notation Implementation. Within the notation of Participants, how each component collaborates can also be discussed.
Problem	A problem that can be addressed by the pattern. Amalgamated with Context and Forces through interdependency. A Solution that is provided by the pattern is derived from a specific Problem assignation. A Problem is defined by the Context in which it arises and the Forces that drive the Problem. Any change in Context or Forces reveals a new Problem or leads to an alternative Solution. Each example in a generative pattern will require its own Problem specification.
Related Patterns	In the context of a Generative pattern, Related Patterns refers to patterns that will combine with the existing pattern, and not to patterns that represent a similar solution. See Chapter Four.
Solution	The Solution is a resolution of the defined problem. A Solution is unique to a problem, therefore, each example in a Generative pattern will require its own problem specification. Although the Solution given in each example may be similar, each will be different based on the defined problem.
Structure	Structure represents the design considerations for the specific Solution to the defined Problem. If a pattern is defined with three examples, then each example should be supported by the appropriate design structure. In order to convey alternative design structures for the pattern, alternative Problem/Solution pairings should be presented.

Table 3.10: Accepted Notation Continued

The main issue in defining a suitable notation for design patterns is interpretation. As has been seen in this discussion, different pattern writers have placed their own interpretation on how to describe a pattern. Some of the interpretation used has a common theme, whilst others are unique to an individual, or pattern style or type. The defining factor in describing a pattern is conveying relevant information about the knowledge represented in a pattern. Meszaros and Doble[79], see Table 3.6, describe the issues relating to defining a pattern, and their main issue is not cluttering the pattern with unnecessary detail. However, as they convey, specific information needs to be present within the pattern or the reader will not be able to determine if the pattern will solve their problem. The list of headings in Table 3.11 are drawn from a selection of experts in their field and together provide the relevant information about a pattern without cluttering the pattern with unnecessary detail. How these headings and relevant information are applied to a pattern is discussed in Chapter Six.

3.4 Summary

The study of design patterns has looked into pattern origins and the different pattern styles and abstractions that have emanated from the concept. The patterns that have now been defined range from code specific patterns (Idioms), to pattern languages written for a given context (Architecture, Process, Software). An integral part of the study of design patterns sought to gain an understanding of the rationale behind a pattern notation. The output from this study lists four different pattern abstractions (Idioms, Catalogues, Systems and Languages) and three different pattern styles (GoF, Portland and Alexandrian).

All the different patterns considered, whether they be type abstractions or particular styles, are all text based patterns. However, there are some patterns that have a high degree of graphical notation. The user interface patterns by Hong and Tidwell use screenshots to emphasise how the pattern can be applied or where they should be applied. Larman also uses a high degree of design graphics (UML) in demonstrating Analysis Patterns. As a notation, graphics can be applied if it will serve the purpose of demonstrating the uses of the pattern. Cooper, like Hong and Tidwell uses screenshots to demonstrate the output of using a pattern. It is not inconceivable that a design pattern could be purely graphical, in that there is a minimum amount of text in the pattern and the majority of the pattern is graphics and design notation, however, no evidence has been found to suggest patterns of such nature exist.

It is also determined within this study that design patterns should be structured to the context in which they are being written and the audience at whom they are aimed. Therefore, specific types of patterns (HCI, Process, Software) have alternative notations. However, different types of patterns have many similarities and it is these similarities that define the overall communication criteria for a well-defined pattern. In completing this study a pattern notation has been abstracted that represents the definitive notation used by experts in pattern writing and it is this notation that will be used for defining a generative design pattern.

Chapter 4

RELATIONSHIPS BETWEEN PATTERNS

4.1 Introduction

There are potentially hundreds of patterns that could be redefined as generative patterns, which within themselves could define many hundreds of dynamic relationships to the other patterns. It would, therefore, be appropriate to produce a standardised way of defining the relationships between collections of patterns — the objective being that a pattern defined or redefined as generative by any given individual, can be read and understood by any other pattern reader, writer or developer, including novice developers.

The focus of many research papers has been on the patterns described within the Design Patterns[45] catalogue. These patterns are accepted by many in the software industry as being benchmark patterns, and have been the influence for many other pattern writers. There are hundreds of software patterns published in many different texts but the patterns defined by Gamma and colleagues are probably the best known and therefore make an ideal representative selection of patterns from which to discuss new techniques of pattern definition and writing. In modifying static design patterns to be used as generative design patterns it is necessary to establish how pattern X is related to pattern Y. In the following discussion on the definition of relationships between patterns, one catalogue and two individual areas of research are considered:

- The demarcation of patterns based on the type classification in the Design Patterns[45] catalogue.
- The demarcation of patterns based on the problem solving classification of Tichy[104].
- The relationship between patterns based on the classification of relationships by Zimmer[119].

Both of the individual areas of research by Zimmer and Tichy identified above have used the Design Patterns[45] catalogue as their primary source of material for their own elements of discussion. The discussion within this thesis retains the Design Patterns[45] catalogue as the patterns chosen for study.

Within this chapter the relational requirements of generative design patterns are discussed. This is

considered from the point of view of how static design pattern notation can be redefined to realize the component's potential as a generative component.

4.2 Classification of Design Patterns

4.2.1 High Level Classification

There are many different types of patterns with different roles and functionality. In order to define generative patterns whereby different patterns will work together to produce systems, it is necessary to establish which patterns will collaborate. Patterns as they are currently defined do not provide sufficient information as to which other patterns they will readily communicate with. In order to define a communication protocol for design patterns, and thereby use them to build systems, it is necessary that the purpose of the design pattern is established. If we can determine what a pattern does, then we can establish some rules on whether it can be applied in collaboration with another pattern. Rules are required as it may be possible for two patterns to work together although there is no practical reason for them to do so. However, determining practicality can be established through experimentation and explicitly written into the details of a pattern.

The catalogue of Design Patterns[45] referred to in Chapter Three by Gamma and colleagues are classified by two criteria; Purpose and Scope. Scope specifies whether a pattern is applicable to a class or an object.

		Purpose		
		Creational	**Structural**	**Behavioural**
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Stratergy Visitor

Table 4.1: Design Pattern Classification[45]

However, the main interest for this body of research, in classifying design patterns, reflects what the pattern does in terms of the purpose of the pattern, which is the criteria being sought. Table 4.1,

presented by Gamma and colleagues, sets out the classification of the patterns in the Design Patterns catalogue.

As can be seen from the table there are three different types of pattern defined by the Design Patterns[45] catalogue; Creational, Structural and Behavioural. Although the catalogue and the patterns themselves may describe why a pattern is Creational, Structural and Behavioural the immediate information provided by the pattern is not sufficient to describe a relationship to another pattern. If the relationship between pattern types is defined at this level of abstraction then it could be argued that all relationships between individual patterns are the same and that all patterns irrespective of their type or functionality will collaborate. It could be possible that all patterns, irrespective of their type or functionality, will collaborate for the simple reason that the collaboration could be forced. However, the question arises as to why one would do this if it is not practical to do so. For example, it may not be practical for certain Creational patterns to work with certain Behavioural patterns. The type abstraction within a pattern at this level can be seen as a generic classification where a relationship could be applied to any pattern within that type or to any pattern of a different type. At a lower level of abstraction a pattern may define a relationship that only exists between patterns that solve a particular type of problem. However, from the Design Patterns catalogue it is known that there are relationships between patterns that have been defined at this level of abstraction[45, 119].

Each pattern in the Design Patterns catalogue comes under one of these categories – *Structural*, *Creational* and *Behavioural*. Figure 4.1 shows the Creational classification of the patterns within the catalogue.

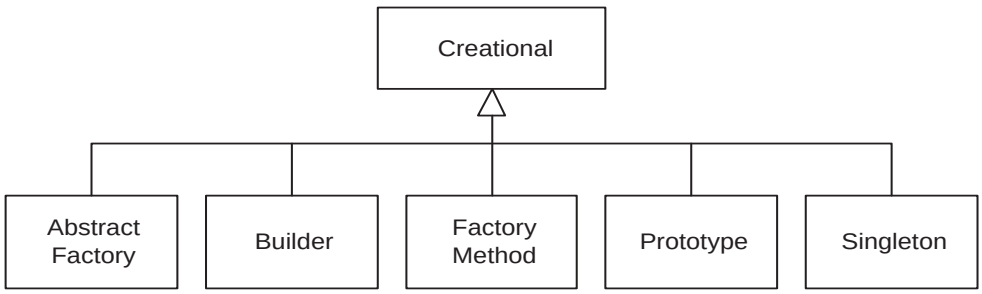

Figure 4.1: Relationships Between Design Patterns

In defining the relationships between different classifications of pattern, a given pattern from any classification has three high-level relational options. For example, a structural pattern may be related to another structural pattern, a creational pattern or a behavioural pattern. Some patterns may be required to define a relationship to three different classifications whilst other patterns may only need to describe one or two relationships. For example, the structural pattern *Composite* defines a relationship

to structural, creational and behavioural patterns whilst the creational pattern *Abstract Factory* only defines relationships to other creational patterns[45, 119].

Writing this in terms of the notation of a pattern, we can add this information to the pattern description. For the Composite pattern, which is classified in the Design Patterns[45] catalogue as Object Structural, under the heading Related Patterns we can add information about related patterns and their classification. Table 4.2 below represents the proposed descriptive information for a generative design pattern.

Related Patterns
Classification types
Structural (Flyweight, Decorator)
Commentary.....
Creational (Builder)
Commentary.....
Behavioural (Visitor, Interpreter, Iterator, Command, Chain of Responsibility)
Commentary.....

Table 4.2: Logical Information for a Generative Design Pattern - Iteration 1

4.2.2 Low Level Classification

The catalogue of patterns presented by Gamma represents a definitive collection of available patterns. Tichy[104], on the other hand, defines a catalogue of over 100 general-purpose patterns, although the patterns are not defined in detail as in the case of Gamma. Although Tichy does not define relationships between patterns he does classify patterns by the problems that they solve. Whilst Gamma defines three families of purposeful patterns, Tichy's classification contains nine separate problem solving categories[104]:

1. **Decoupling**: Dividing a software system into independent parts in such a way that the parts can be built, changed, replaced, and reused independently.

2. **Variant Management**: Treating different objects uniformly by factoring out their commonality.

3. **State Handling**: Generic manipulation of object state.

4. **Control**: Control of execution and method selection.

5. **Virtual Machines**: Simulated processors.

6. **Convenience Patterns**: Simplified coding.

7. **Compound Patterns**: Patterns composed from others, with the original patterns visible.

8. **Concurrency**: Controlling parallel and concurrent execution.

9. **Distribution**: Problems relevant to distributed systems.

In the case of Tichy, there is a deeper level of refinement in the classification. In describing the relationship between different types of patterns, it could be useful as a descriptive element to discuss how different classes of pattern interact based on the type of problem they solve, particularly where sub-systems or non-functional elements are concerned. This can be achieved by introducing problem solving properties into a pattern based on Tichy's classification. With the implied detail that is contained within the problem type classification, Tichy's classification can be defined at a lower level than the classification presented by Gamma. By lower level it is implied that Tichy's classification types are a subclass of the Gamma classification types. Tichy's classification relates to specific problems or areas of concern whereas Gamma's classification is generalised within three areas. For example, a Structural definition, as defined by Gamma, indicates that the pattern will represent some aspect of the basic framework of a system, whereas the concept of Decoupling, as defined by Tichy, provides a level of detail that describes how certain aspects of the base framework can be separated out for ease of development and maintenance.

In attempting to determine which patterns are best suited to communicate in a generative pattern language, information about the patterns and their purpose is going to play an important role. Figure 4.1 shows that certain patterns, as classified by Gamma, are defined as Creational patterns. We can now add information to those patterns based on the classification provided by Tichy. Figure 4.2 on the following page uses the same Creational patterns as is contained in Figure 4.1 but which are now extended with an additional level of information. The information as it is presented is analogous to an inheritance hierarchy where the top level of the tree contains base information. Lower levels of the hierarchy contain more detailed information that is specific to the components at that level. To this end, the tree represents a classification hierarchy of information.

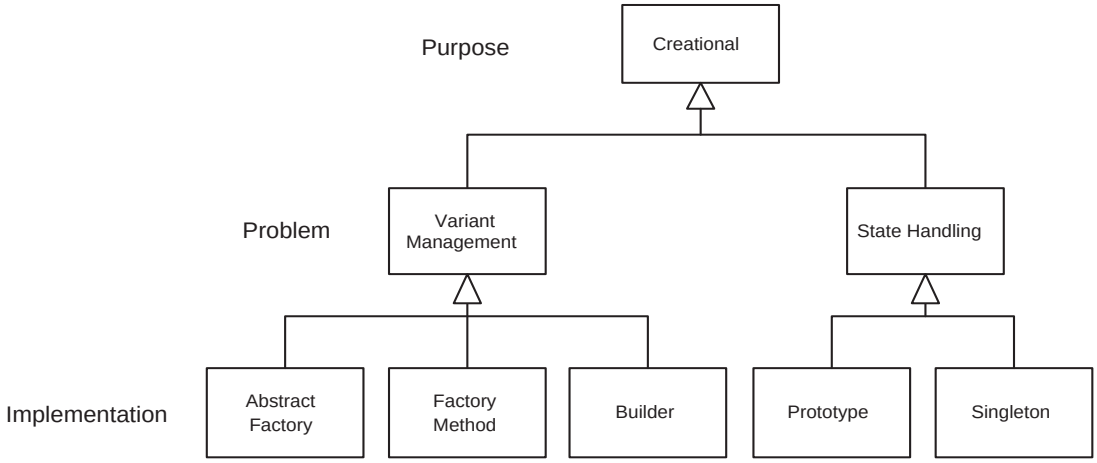

Figure 4.2: Creational Pattern Information Hierarchy

Although the patterns classified by Gamma are present within the catalogue of Tichy, the other patterns in Tichy's catalogue have not been classified in terms of structural, creational or behavioural. Therefore only those patterns classified by Gamma can be worked into the hierarchy tree. Adding Gamma's classification types to all the patterns in Tichy's catalogue is an element of future work and will be discussed further in Chapter Nine, Future Work. The tree describes three layers in the hierarchy where each descending layer is described at a lower level of abstraction. The highest layer describes the general purpose of the patterns. The intermediate layer describes the problem area that is best suited to the patterns, whilst the lowest layer describes the finer detail of the patterns.

Figure 4.3 on the following page illustrates the information tree based on behavioural patterns at the root of the tree. It is important to separate information trees to indicate that specific low-level patterns do not inherit context from multiple parents. Tichy indicates that the pattern classifications that he proposed are mutually exclusive; therefore a pattern cannot belong to more than one category. For example, the Decorator pattern cannot be a Decoupling pattern and a Variant Management pattern at the same time. The complete set of relational trees based on structural, creational and behavioural patterns is illustrated in Appendix E.

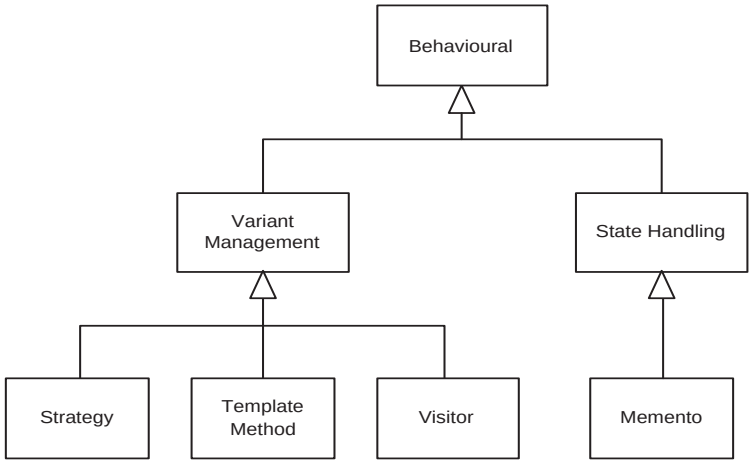

Figure 4.3: Behavioural Pattern Information Hierarchy

We can now add this additional criterion to the existing pattern description that was defined earlier. Again, concentrating on the Composite pattern, we can add the problem solving types. Table 4.3 below represents the proposed descriptive information for a generative design pattern.

Related Patterns

***Classification type* (Structural)**

Commentary on Structural relationship between patterns.....

***Related Pattern* (Flyweight)**

Commentary on Flyweight pattern.....

***Problem Solving Type* (State Handling)**

Commentary on State Handling relationship between patterns in a Structural context.....

***Related Pattern* (Decorator)**

Commentary on Decorator pattern.....

***Problem Solving Type* (Decoupling)**

Commentary on Decoupling relationship between patterns in a Structural context.....

Table 4.3: Logical Information for a Generative Design Pattern - Iteration 2

Continued on next page.

Classification type **(Creational)**

Commentary on Creational relationship between patterns.....

Related Pattern **(Builder)**

Commentary on Builder pattern.....

Problem Solving Type **(Decoupling)**

Commentary on Decoupling relationship between patterns in a Creational context.....

Classification type **(Behavioural)**

Commentary on Behavioural relationship between patterns.....

Related Pattern **(Visitor)**

Commentary on Visitor pattern.....

Problem Solving Type **(Variant Management)**

Commentary on Variant Management relationship between patterns in a Behavioural context.....

Related Pattern **(Iterator)**

Commentary on Iterator pattern.....

Problem Solving Type **(Decoupling)**

Commentary on Decoupling relationship between patterns in a Behavioural context.....

Related Pattern **(Interpreter)**

Commentary on Interpreter pattern.....

Problem Solving Type **(Virtual Machines)**

Commentary on Virtual Machines relationship between patterns in a Behavioural context.....

Related Pattern **(Command)**

Commentary on Command pattern.....

Related Pattern **(Chain of Responsibility)**

Commentary on Chain of Responsibility pattern.....

Continued on next page.

***Problem Solving Type* (Control)**

Commentary on Control relationship between patterns in a Behavioural context.....

4.3 Individual Relationships

A significant area of interest in identifying how and which patterns should communicate in terms of a generative pattern is Zimmer's *Relationships between Design Patterns*[119] criteria. Zimmer's classification explores the relationships between existing design patterns and uses the Design Patterns[45] catalogue as the role models on which to define the classification. In this, three relationship classifications are discussed:

- Pattern X uses Pattern Y in its solution.
- Pattern X can be combined with Pattern Y.
- Pattern X is similar to Pattern Y.

The third classification, described above, refers to patterns that have a similar problem/solution pairing. That is, patterns that give details of an alternative solution. This relationship classification is not taken into consideration because it defines an alternative solution to a given pattern and not how solutions are related, which represents the work in progress.

The pattern map defined by Zimmer is illustrated in Figure 4.4 above showing the *Pattern X uses Pattern Y in its solution* classification and the *Pattern X can be combined with Pattern Y* classification. The third relationship defined by Zimmer, Pattern X is similar to Pattern Y, has been left out of the illustration to improve the clarity of the two elements that are useful as an additional element of notation for a generative design pattern.

As can be seen from Zimmer's classification, there are two different types of relationship. However, the map indicates that a pattern only has one type of relationship to any one related pattern. If a given pattern is related to more than one other pattern then the definition for each relationship needs to reflect the type of relationship between each of the two related patterns.

It is an argument within this thesis that the description of any relationship between patterns should reflect how patterns are used by other patterns. The argument also maintains that the relationship between the classification types to which the patterns belong be defined. This argument is founded

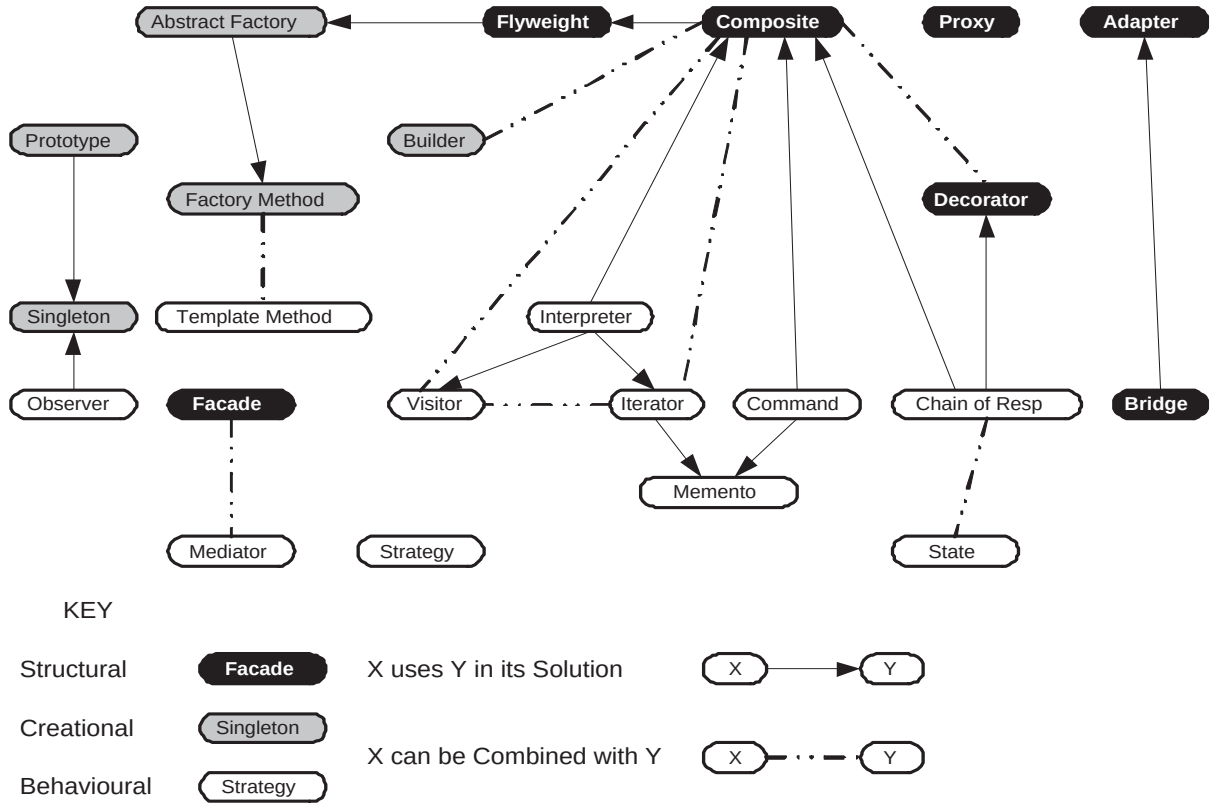

Figure 4.4: Relationships Between Design Patterns (Based on Zimmer[119])

on the principle that the relationship between, for example, two structural patterns may be different to the relationship between a structural pattern and a creational or behavioural pattern. Any given description within a pattern would first describe how particular types of patterns communicate and secondly how the individual patterns collaborate to form a bond between the two.

When defining the relationships between patterns any given pattern requires one description of a relationship for each pattern to which it is related. Given the case that a structural pattern could be related to more than one structural pattern or more than one behavioural pattern the content of the description of the relationship between pattern classifications would be repeated – which is an unnecessary duplication of effort. In this situation therefore, the consideration described above detailing the relationships between classifications of patterns would remove the duplication of descriptive passages.

Within the classification map in Figure 4.4 there are two defined relationships: *Pattern X uses Pattern Y in its solution* and *Pattern X can be combined with Pattern Y*. However, there is a third relationship within the map that is not discussed. The reason for not extending the relationship as a classification is because it is the same relationship as Pattern X uses Pattern Y in its solution but described in a different way; namely **Pattern X is used by Pattern Y in its solution**. The 'used by' relationship is

defined by Meszaros[79] and further discussed by Noble[83], who describes it as the inverse of the 'uses' relationship. On the basis of two individual communicating patterns these are identical relationships.

It is shown[20, 83] that not only can a pattern use another pattern but also a pattern can be used by another pattern. In each definition of the relationship the focus of attention is on the pattern being defined. In the relationship *X uses Y*, X is the defined pattern. In the case of *X is used by Y*, X is still the defined pattern. Therefore, X is the dominant partner in the two definitions of the relationship so the relationship becomes X uses / X is used by, illustrated in Figure 4.5 below.

Figure 4.5: Pattern X uses, is used by

The 'used by' relationship, although only the inverse of the 'uses' relationship, provides information of a known relationship between two patterns. It is not enough when defining a generative pattern only to discuss which pattern a given pattern uses — the generative pattern has to describe known related patterns in order to be generative, including those patterns it is used by.

By including the 'used by' relationship, relationships can be defined not only by how a pattern is related to another pattern but how another pattern is related to the pattern in question. In this way it is possible to define how architectures are built from patterns by defining matching join-points between patterns. That is, a relationship can be defined within the pattern on how it uses other patterns and how it is used by other patterns.

We can now add this criterion to the existing pattern description that was defined earlier. Again, concentrating on the Composite pattern, we can add the association types. Table 4.4 on the following page represents the proposed descriptive information for a generative design pattern.

Related Patterns

***Classification type* (Structural)**

Commentary on Structural relationship between patterns.....

***Related Pattern* (Uses Flyweight)**

Commentary on how Composite uses the Flyweight pattern.....

***Problem Solving Type* (State Handling)**

Commentary on State Handling relationship between patterns in a Structural context.....

***Related Pattern* (Combines Decorator)**

Commentary on how Composite combines with the Decorator pattern.....

***Problem Solving Type* (Decoupling)**

Commentary on Decoupling relationship between patterns in a Structural context.....

***Classification type* (Creational)**

Commentary on Creational relationship between patterns.....

***Related Pattern* (Combines Builder)**

Commentary on how Composite combines with the Builder pattern.....

***Problem Solving Type* (Decoupling)**

Commentary on Decoupling relationship between patterns in a Creational context.....

***Classification type* (Behavioural)**

Commentary on Behavioural relationship between patterns.....

***Related Pattern* (Combines Visitor)**

Commentary on how Composite combines with the Visitor pattern.....

Table 4.4: Logical Information for a Generative Design Pattern - Iteration 3

Continued on next page.

***Problem Solving Type* (Variant Management)**

Commentary on Variant Management relationship between patterns in a Behavioural context.....

***Related Pattern* (Combines Iterator)**

Commentary on how Composite combines with the Iterator pattern.....

***Problem Solving Type* (Decoupling)**

Commentary on Decoupling relationship between patterns in a Behavioural context.....

***Related Pattern* (Used By Interpreter)**

Commentary on how Composite is used by Interpreter pattern.....

***Problem Solving Type* (Virtual Machines)**

Commentary on Virtual Machines relationship between patterns in a Behavioural context.....

***Related Pattern* (Used By Command)**

Commentary on how Composite is used by the Command pattern.....

***Related Pattern* (Used By Chain of Responsibility)**

Commentary on how Composite is used by the Chain of Responsibility pattern.....

***Problem Solving Type* (Control)**

Commentary on Control relationship between patterns in a Behavioural context.....

4.4 Pattern Map

As can be seen in Section 4.3, the composite pattern alone takes up a full page of text just to highlight the known patterns with which it will function. This itself is not a problem but, when descriptive commentary has been added the reader may have difficulty in finding and visualising the related patterns through the jumble of text. A quick and easy way to visualise the related patterns is to include at this point a relational map, which expands on the models described in Figures 4.2 and 4.3.

The Pattern map for the Composite pattern, described in Figure 4.6 provides a visual representation of related patterns and the categories into which they fall.

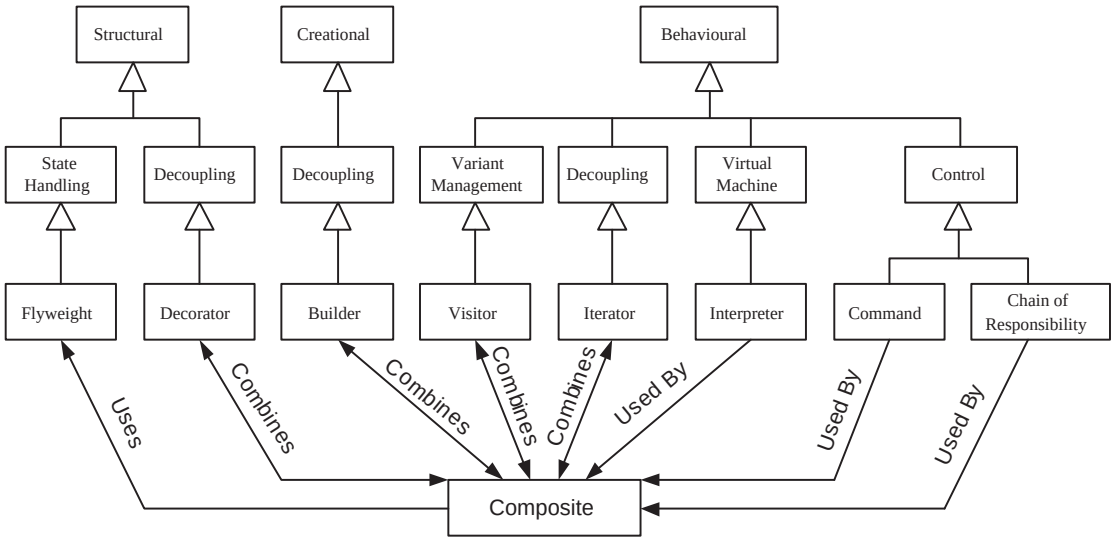

Figure 4.6: Patterns Related to Composite

The map itself in this current form is not over complex but could be if patterns from outside the Design Patterns[45] catalogue were also defined as part of the Composite pattern. Also the map does not give the full picture of relationships — it only describes at this point a relationship to other patterns. The map could also include attributes of a pattern, for example the participating classes of a particular pattern.

4.5 Describing Relationships

In defining a meaningful description of the relationship to other patterns, three levels of description have been identified.

- A description of the classifications of pattern.
- A description of the problem solving classification of patterns.
- A description of the relationship between patterns.

As discussed in section 4.2.1 *High Level Classification* each pattern could describe a relationship to three different classification types, assuming that a classification of pattern could be related to a pattern of its

own class. To alleviate any possible confusion and for reasons of practicality, only those classifications attributed to related patterns should be discussed in the text of a pattern. For example, if a Creational Pattern is not related to any Behavioural patterns then do not discuss the relationship between Creational and Behavioural classification types. To discuss non-essential relationships will waste time and effort of the pattern reader and will ultimately serve no purpose other than to add confusion.

4.5.1 Classification

In defining a relationship or relationships, the pattern should define its own classification and the classification types to which it is related, and what makes that relationship. The relationship between classification types can be defined through an assessment of the intent of the pattern, and the intent of the pattern to which a relationship is proposed. For example, structural patterns define how classes and objects can be composed to form larger structures. Structural patterns often use inheritance to compose interfaces. Through multiple inheritances, two or more interfaces can be combined to form a composite pattern (that is, a union of two or more patterns and not The Composite Pattern). Therefore, structural patterns often form a 'combines' relationship with other structural patterns. However, structural patterns also use other structural patterns, which must be made explicit when defining the relationship to other patterns.

Therefore, in the meaningful description, the relationship between classification types should make explicit that the relationship is a *Combines* relationship or a *Uses* relationship or a *Used By* relationship, together with the intent of that relationship. Because a Structural pattern, such as Composite, may also be related to Creational and Behavioural patterns as well as to other Structural, defining how the patterns cooperate (*Combines, Uses, Used By*) needs to be done at a lower level than the classification relationship level.

The related pattern information in Table 4.5, on the following page, is an example of the type of commentary that could describe the relationship between Structural patterns within the section on related patterns for a generative design pattern.

Related Patterns ***Relational Classification type* (Structural)** The connection of two or more Structural Patterns serves to form a larger structure. For example, multiple inheritances will mix specific participants from a pattern into one participating class. The result is a single participating class inherited from two or more patterns that combines the properties of its parent classes.

Table 4.5: Concrete Information for a Generative Design Pattern - Iteration 1

4.5.2 Problem Solving

The second level of description is the problem type that a pattern may solve. Again the pattern should be clear about the problem that it solves and its relationship to other problem solving types. Types are mutually exclusive so it is unlikely that a pattern will solve more than one problem, although, according to Tichy, there are a few exceptions[104]. Within this section only those problems that are recognised as being an attribute of the patterns in the Design Patterns[45] catalogue are discussed.

Decoupling

A large proportion of patterns tend to deal with Decoupling which helps to divide a system into independent units. A system composed of decoupled parts can easily be extended or adapted by adding or modifying parts[104]. Decoupling patterns are often structural or behavioural and mostly use or work in combination with other structural or behavioural patterns.

For a Decoupling pattern the pattern could include:

- *Why the decoupling takes place.*
- *How the decoupling takes place.*
- *What the decoupling will add to a system*, or
- *What the decoupling will modify.*

Variant Management

Variant Management patterns treat different objects with a common purpose in a consistent manner by factoring out their commonality. However, Variant Management patterns are often dependent on

the features of a programming language[104]. Patterns that solve Variant Management problems come from all classification types and are usually an alternative to another pattern. Variant Management patterns usually have a Combines relationships to other Variant Management patterns, whilst a few patterns have a Used and a Used By relationship.

For Variant Management patterns the description could describe:

- *What objects are being manipulated.*
- *Why they are being manipulated.*
- *What objects will be manipulated through a Combines relationship and how the combination will affect the object.*
- *How the pattern will use other patterns to manipulate an object*, or
- *How the pattern will be used by other patterns to manipulate an object.*

State Handling

State Handling patterns manipulate the state of objects generically. This means that these patterns work on the state of any object, independent of their actual purpose. Like Variant Management patterns, patterns that solve State Handling problems come from all classification types. State Handling patterns are most often used by patterns within their own classification type but rarely use or combine with other patterns.

The pattern description for State Handling could include:

- *The state of an object prior to a change of state and after a change of state.*
- *How the change of state is affected by the pattern that is using the current pattern*, and
- *How the current pattern might affect the state of the objects manipulated by a pattern that is being used.*

Control

Control patterns deal with the control of execution and the selection of appropriate methods. Control patterns are mostly Behavioural patterns and use other patterns in their manipulation of system functionality. Although Control patterns are mostly behavioural, they do for the most part use Structural patterns. Occasionally, Control patterns will combine to manipulate aspects of functionality.

In describing aspects of Control the description could define:

- *For what aspects of functionality it is responsible.*
- *How it uses and/or controls the functionality of other patterns*, and
- *How it will combine with other patterns to enhance functionality.*

Virtual Machines

A Virtual Machine problem is derived from system processes. It is mostly an element of functionality that interprets a program written in a specific language. Like Control problems, Virtual Machine problems are Behavioural. They also use other Behavioural and Structural patterns, but are rarely used or combine with other patterns.

A Virtual Machine problem could describe:

- *How its internal functionality is executed*, and
- *How it will use the functionality provided by a related pattern.*

Section 4.5.2 gives a brief description of the problem type under each category of problem solving relationships, as described by Tichy[104]. This however, is inadequate to describe the relationship to other patterns. Each brief description above provides details of what should be described in the relationship to other patterns for a given problem solving type. However, the How, What and Why of the relationship will depend on the individual pattern. How Pattern 'X' uses Decoupling with Pattern 'Y' may be different from how Pattern 'X' uses Decoupling with Pattern 'Z'. Therefore, this finer detail of How, What and Why should be described in the Relational Association Type section.

The basic detail can now be added to the description of the problem solving relationship under the heading Related Patterns. The detail in Table 4.6, on the following page, is repeated from previous sections to show how the definition is being built up.

Related Patterns

***Relational Classification type* (Structural)**

The connection of two or more Structural Patterns serves to form a larger structure. For example, multiple inheritances will mix specific participants from a pattern into one participating class. The result is a single participating class inherited from two or more patterns that combines the properties of its parent classes.

***Problem Solving Type* (Decoupling)**

Decoupling helps to divide a system into independent units. A system that includes decoupled elements can easily be extended or adapted by adding or modifying those elements[104].

Table 4.6: Concrete Information for a Generative Design Pattern - Iteration 2

4.5.3 Association Type

The final element in defining a relationship between related patterns is the type of association and the individual knowledge of how the participating patterns communicate. This description should reflect the individuality of the pattern and its relations. The description should encapsulate the How, What and Why of the relationship as well as describing what element of Pattern X *Uses*, is *Used By* or *Combines* with Pattern Y. This could be a general description or could be described at the coding level with an example.

Again, we can now add this type detail to the definition of the relationship. The detail in Table 4.7 is repeated from previous sections to show how the definition is being built up.

Related Patterns

***Relational Classification type* (Structural)**

The connection of two or more Structural Patterns serves to form a larger structure. For example, multiple inheritances will mix specific participants from a pattern into one participating class. The result is a single participating class inherited from two or more patterns that combines the properties of its parent classes.

Table 4.7: Concrete Information for a Generative Design Pattern - Iteration 3

Continued on next page.

***Problem Solving Type* (Decoupling)**

Decoupling helps to divide a system into independent units. A system that includes decoupled elements can easily be extended or adapted by adding or modifying those elements[104].

***Association Type* (Combines (Decorator))**

Why the decoupling takes place.

How the decoupling takes place.

What the decoupling will add to a system, or

What the decoupling will modify.

These descriptions can be supported by the generative modelling, discussed in Chapters Two and Five, or more specific models attached to an example. Further to this, more specific Pattern Maps can support the whole section on related Structural patterns. Figure 4.6 on page 68 shows a complete pattern map for Composite as a whole. This can be used at the beginning of the section in its current form. To show greater detail in the map it can be broken down for each individual pattern for which Composite has a relation — as is shown below in Figure 4.7.

Relational Map

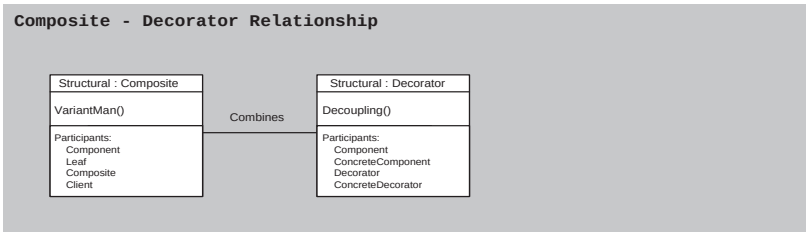

Figure 4.7: Relationship between Composite and Decorator

The section on Related Patterns as it is defined above for the purpose of a generative pattern, is far more detailed than the Related Patterns section as it is defined in the Design Patterns catalogue, which is reprinted below.

Related Patterns[45]

"*Decorator is often used with Composite. When Decorators and Composites are used together, they will usually have a common parent class. So Decorators will have to support the Component interface with operations like Add, Remove, and GetChild*"[45].

This commentary does provide some information, but it is limited in detail. The corresponding text in the Decorator pattern from the Design Patterns catalogue[45] has even less detail.

4.6 Summary

The relationship between patterns has been discussed on three levels: firstly at a high level based on the classification introduced by Gamma which divides patterns into three types based on their purpose; secondly at an intermediary level based on the problem solving classification introduced by Tichy; and finally on the individual relationships between patterns as defined by Zimmer. These relationships are central to the generative design pattern as they describe how different patterns interact based on the type of interaction defined by the pattern.

The relationship trees that have been defined break the patterns down into specific categories, allowing a pattern to be traced back to its roots in the relational hierarchy. The trees provide the rationale for the conditions that have been defined in determining the relations of the current pattern. The conditions are supported by a pattern map, which itself is based on the relational tree.

As a result of bringing together a classification from a popular book and two individual pieces of research a way of defining relationships between individual patterns has been defined.

Chapter 5

PATTERN MODELLING

5.1 Introduction

Chapter Five is an exploration of the modelling notation used in existing software design patterns, particularly the Class and Sequence diagrams. An important factor in a design pattern is the design structure of the pattern itself. In existing design patterns such as those produced by Gamma et al[45] and many other pattern writers (for example the Pattern Oriented Software Architecture[20, 99] catalogues and the Pattern Language of Program Design series[31, 41, 76, 110, 111]) the class diagram is a significant feature of the pattern. However, it can be seen from these and other catalogues, that other important models such as the sequence diagram are not used as consistently as the class diagram. Indeed, the wide range of models available in numerous software development methods and the Unified Modelling Language[54], a standard modelling notation, are hardly given any consideration at all.

It is shown in Chapter Five that the class diagram is used extensively throughout software design patterns but is not used consistently throughout pattern catalogues. In some design patterns a class diagram is drawn but lacks viable discussion[45], whilst diagrams within other design patterns are given considerable thought, and are often discussed with the inclusion of sample code such as those by Grand[48, 49] and Stelting[102]. A similar situation can be seen with sequence diagrams in that there is limited discussion in some design patterns whilst in others they are given a more important role to play in the discussion of the pattern. For example, Buschmann[20, 99] makes significant use of the sequence diagram.

From looking at existing modelling notations that are used within design patterns, evidence is presented in Section 5.3 to show that not only are class diagrams used inconsistently but what is being modelled by pattern writers is not consistent with what is being provided as an example. In most cases, the model being used and described does not match the sample code that is being provided by the pattern writer (See Figures 5.10 and 5.11). To this end, it is recommended that what is being defined in a model should be developed in the sample code.

5.2 Sequence Diagrams

Sequence diagrams may or may not be used in different pattern catalogues or in individual patterns. When they are used in pattern catalogues, they may be used in some patterns but not in others. In the Design Patterns[45] catalogue, the sequence diagram is rarely used. There is an example of a sequence diagram in the Design Patterns[45] catalogue of the following patterns: Builder, Chain of Responsibility, Command, Mediator, Memento, Observer and Visitor. The diagram is presented mostly in the Collaborations section of the design pattern notation and occasionally in the Motivation section. Six of the seven examples are defined within Behavioural patterns with the exception being Builder, which is a Creational pattern. What this represents is the inconsistency of use of a valuable modelling technique in this particular catalogue of patterns. However, in the POSA[20, 99] catalogues the sequence diagram is used far more consistently. Almost all the patterns present a sequence diagram and it is always contained within the Dynamics section of the pattern, which would appear to be an appropriate place to display the diagram as the sequence diagram is a dynamic modelling artefact[54]. The sequence diagram that is presented with the Broker pattern from the POSA[20] catalogue is shown in Figure 5.1 below. As well as providing the sequence diagram, the dynamics of the interaction are also discussed with a step-by-step account of what is tacking place in the diagram.

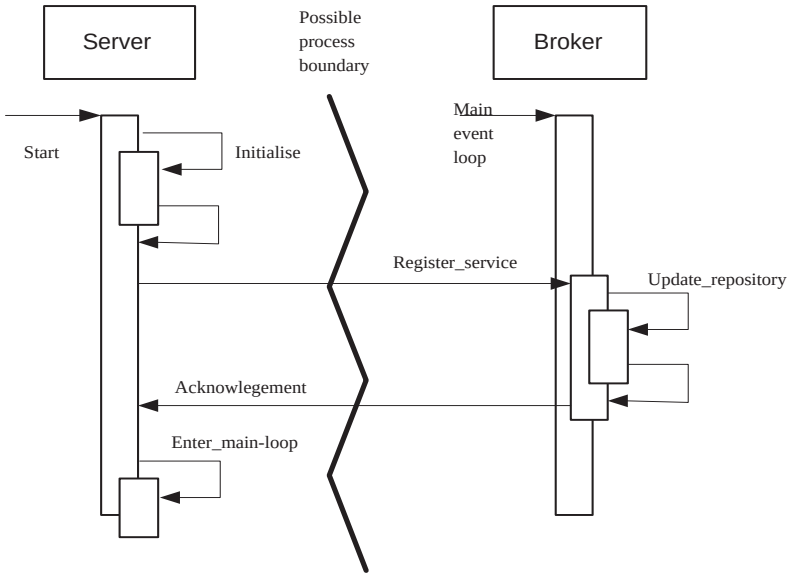

Figure 5.1: Sequence Diagram for the Broker Pattern

Additionally, most patterns from this catalogue will provide more than one example of the dynamics of the pattern, which is felt to be good practice and which is in keeping with the principle of a minimum of three known uses before being accepted as a pattern. For the Broker pattern, the Dynamics section

of the pattern presents three examples. Example number three from the Broker pattern, as defined in the POSA[20] catalogue is shown in Figure 5.2 below.

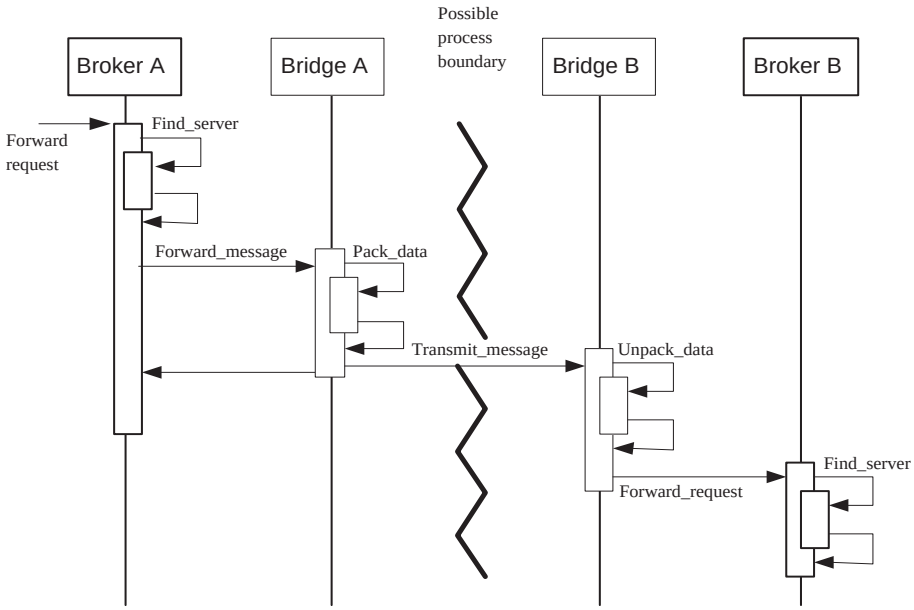

Figure 5.2: Sequence Diagram for the Broker Pattern

The two diagrams in Figures 5.1 and 5.2 represent different scenarios on the use of the Broker pattern. By presenting alternative dynamics for a pattern the reader of the pattern can explore different approaches to solving problems or indeed, developing systems. For example, if scenario one does not meet the needs of the reader then there are other scenarios that may solve the problem or aid development.

A similar situation is evident in the J2EE[5] patterns catalogue where the sequence diagram is accompanied by a number of coded examples. In the J2EE examples the pattern in question is regularly shown interacting with components from related patterns, although it has to be said, the separate patterns often share the same individual components. Where the POSA[20] catalogue defines sequence diagrams under the Dynamics heading of a pattern, in the J2EE[5] catalogue sequence diagrams are presented under the heading Participants and Responsibilities. Along with the diagram each component of the diagram is accompanied by a short description. Using the sequence diagram as part of a coded example can make the pattern easier to understand, as the diagram itself is providing an objective view that the class diagram cannot portray. This type of diagram is a positive aspect for generative design patterns as generative patterns are intended to be dynamic in nature in that they will create systems.

The sequence diagram for the J2EE pattern shown in Figure 5.3 below shows the range of objects that could be used in this particular pattern.

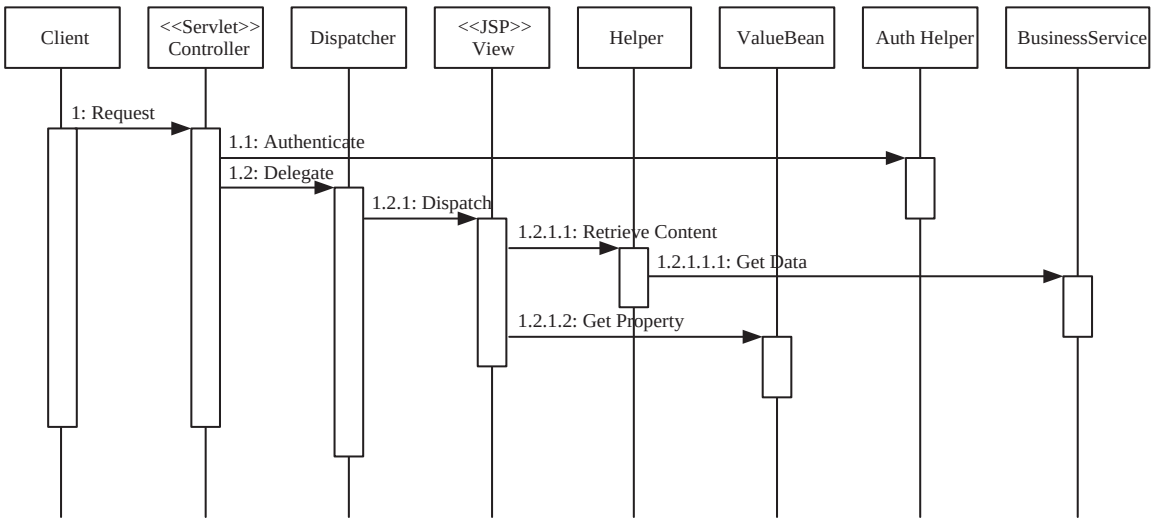

Figure 5.3: Sequence Diagram for the Dispatcher View Pattern

It is unlikely that all the components described in the diagram above will be utilized at any one time in an implementation of the pattern, as is shown by Figures 5.4 and 5.5. The sequence diagram in Figure 5.4 below is an example from the Dispatcher View pattern and represents the first of two different strategies for its use.

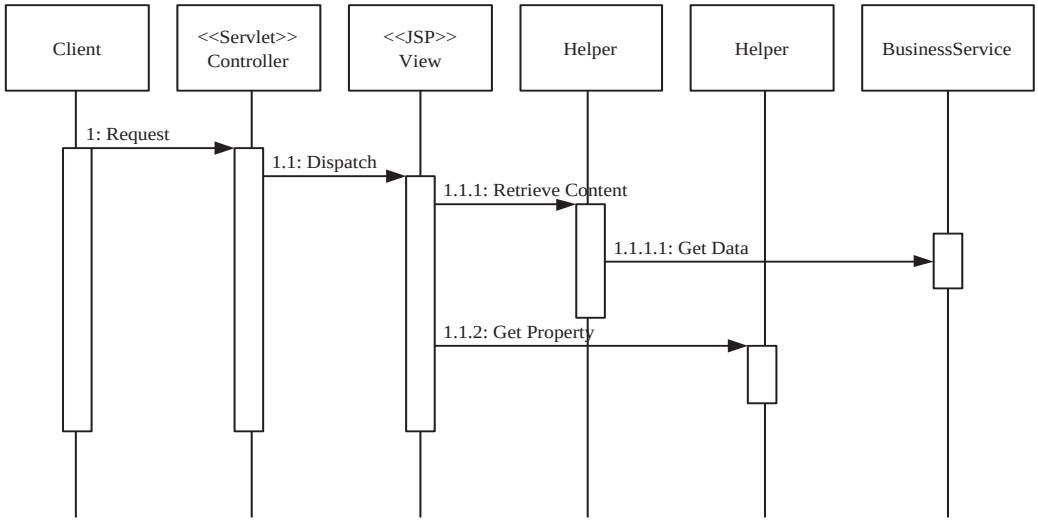

Figure 5.4: Dispatcher in Controller Strategy

Figure 5.5 below represents an alternative strategy for using the Dispatcher View pattern.

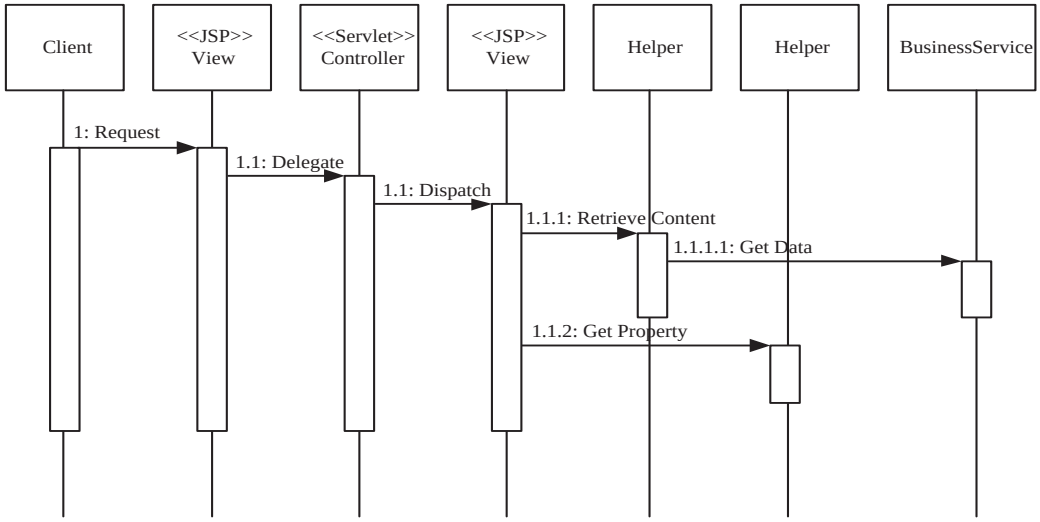

Figure 5.5: Dispatcher in View Strategy

Within the section on Participants and Responsibilities each component of the sequence diagram shown in Figure 5.3 is discussed, although in brief. However, unlike the POSA patterns the J2EE patterns do not provide a step-by-step account of what is taking place in the diagram, although each scenario is discussed in general. Also, unlike the POSA patterns the sample code appears in a section of the pattern disconnected from the sequence diagram, although there is some discussion of the code that does relate to the diagram. However, the sample code and discussion is confusing as it appears to make references to both strategies as if they were the same implementation. Although the use of sequence diagrams representing alternative scenarios is a step in the right direction, its usefulness as applied in the J2EE[5] catalogue is still not sufficient for providing adequate understanding for the reader, particularly when the discussion in the Sample Code section is confusing.

There are many other catalogues on design patterns: some are specific to Java[33, 48, 49, 50, 74, 102, 112], whilst others are specific to .NET[51, 81, 103], others such as the PLoPD[31, 41, 76, 110, 111] catalogues have no allegiance to any specific language. What is consistent among the many catalogues is inconsistency, not just in the use of sequence diagrams but in the way design patterns are described in general. Many of the catalogues do not use standard UML diagrams whilst others will use UML diagrams to emphasise a particular point.

What can be seen in the POSA[20, 99] and J2EE[5] catalogues can be considered as good practice in providing knowledge and understanding for a pattern. Whilst some pattern catalogues provide only a written description of a pattern or a description with a limited number of design considerations, the POSA[20, 99] and J2EE catalogues provide good additional information that should be exploited in

defining a generative pattern. Each utilization of the sequence diagram that has been discussed has its good and bad points. The good points from these separate configurations can be combined to make a pattern easier to understand and use.

5.3 Class Diagrams

The class diagram is a significantly important UML component for modelling the attributes, operations and relationships between separate components of a software artefact. Any person involved in the development phase of a software system will use the models of design, particularly the class diagram, as a blueprint for writing the software code. Similarly, the class diagram is a significantly important component of a design pattern; it models the components of what is being described in the pattern itself, and the relationships between those components. Without the class diagram the design pattern would lack credibility as a component of expert knowledge because there would be no key component to hold the discussion together. That is, software is built from models of design and without that design the end product is open to interpretation.

Class diagrams represent an element of consistency throughout most pattern catalogues. In any pattern catalogue there will be very few software patterns that do not contain a class diagram in their notation. One such exception is the pattern 'Identity Map' in the Patterns of Enterprise Application Architecture[43] catalogue by Fowler. Indeed, even in individual patterns that are often presented at conferences and published in proceedings there will be few that do not contain a class diagram.

In the Design Patterns[45] catalogue the notation of the pattern contains an element 'Structure' where a high-level model of the pattern's components are displayed. Structure as defined in the Design Patterns[45] catalogue is often nothing more than a single class diagram. Whereas most pattern catalogues will include a discussion together with the model being displayed, or use a class diagram to emphasise a discussion, in the Design Patterns[45] catalogue, the model is not connected to any discussion of the design or indeed the processes or collaborations between any of the model's components. However, the heading 'Structure' where the model is displayed is followed up by the heading 'Participants', which names each component in the diagram and gives a brief summary of the role each participant plays. In defence of the Design Patterns[45] catalogue, they do use additional class diagrams to further discuss elements of notation such as 'Motivation', where scenarios of problems to be solved are exemplified. However, the diagrams used in this area are often there to show how a particular system structure is being inefficiently utilised. Figure 5.6 below is based on the structural model of the Composite pattern as defined in the Design Patterns catalogue.

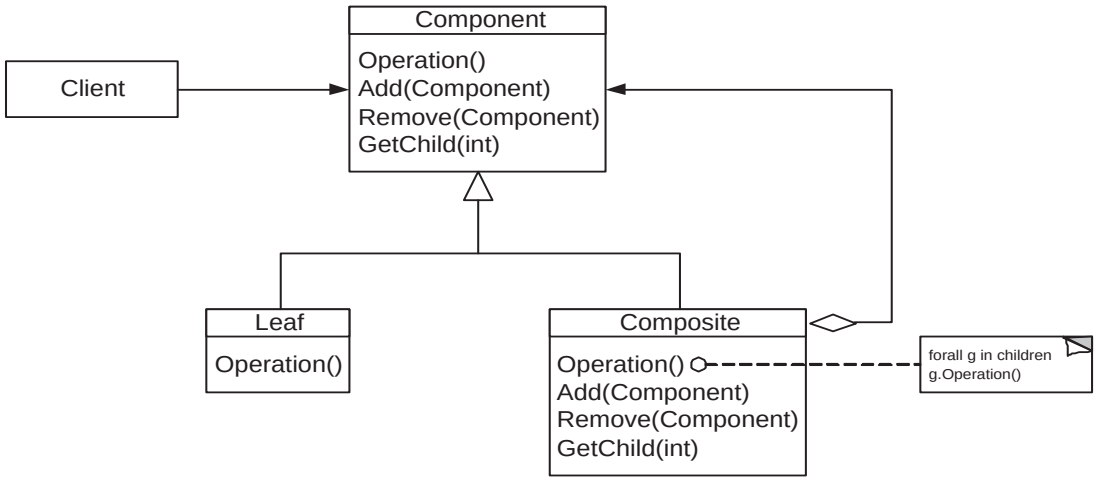

Figure 5.6: Composite Class Diagram

The class diagrams used in the Design Patterns[45] catalogue are often used as benchmark designs by other catalogues or people who contribute to the discussion of design patterns through education or other means. However, it can be shown that the model being used to discuss the pattern does not always match the model that is created from reverse engineering the supplied code. In other words, people are interpreting the design put forward in the Design Patterns[45] catalogue, but not altering the design to match the true implementation, which adds significant confusion to the understanding of design patterns, particularly for novice developers.

From the book Patterns in Java: a Catalogue of Reusable Design Patterns Illustrated with UML[48], the class diagram described in the Forces and Solution sections of the Composite pattern are a simulation of the class diagram described in the Design Patterns[45] catalogue. The only difference in the diagram described by Grand[48] and that described by Gamma et al[45] is that the Composite pattern by Grand uses a Composition association and not an Aggregation association. This is understandable as the example being described is a document, which requires a Composition association. An illustration of Grand's Composite class diagram from the Solution section of the design pattern can be seen in Graphic A of Figure 5.7. on the following page.

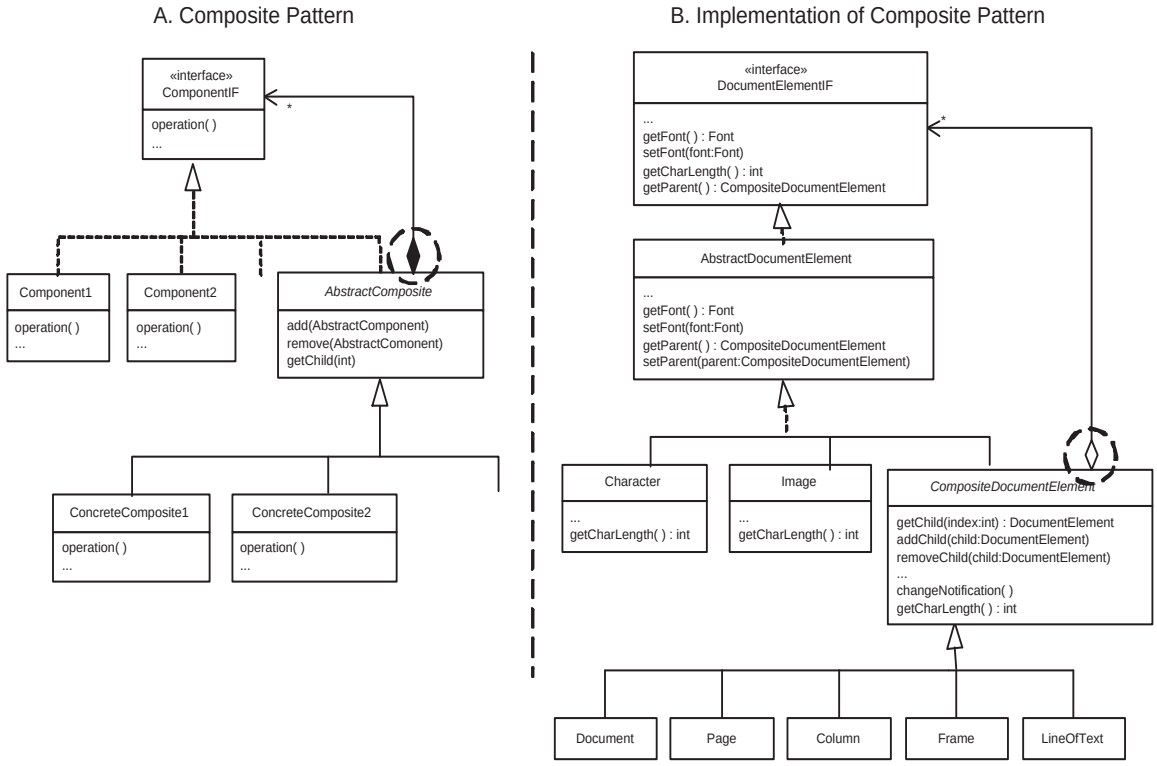

Figure 5.7: Grand's Composite Class Diagram[48]

The minor difference in the class diagrams is not a cause for concern, the reader of the pattern, be it novice or professional, should accept that what is being modelled in the class diagram is the structural composition of the pattern being described. However, further reading of the Composite design pattern as described by Grand[48] reveals a different class structure described in the Code Example section to that defined in the Solution section of the pattern. The class diagram shown in Graphic B of Figure 5.7 has acquired an AbstractDocumentElement component that implements the interface component. Hence, the Leaf components and the Composite component extend the newly acquired Abstract class. Also in the Code Example in Graphic B, the Composition association has changed back to an Aggregation association, as can be found in the standard design of the Composite pattern. Also noticeable in both examples of the design is the lack of a collection object in the details of the Composite class, despite the model indicating that one exists.

Further to the change in diagram structure to that described in the Solution section of the pattern, the code examples are supplied to the reader from a Web site that acts as a companion to the book[48]. On reverse engineering the code example of the Composite pattern, a different class structure is revealed to that described by Grand in Graphic B of Figure 5.7. Although the supplied code does match, in part, the code written in the catalogue, the engineered diagram does not match the diagram that is modelled in the Code Example section of the pattern. What is revealed in the engineered diagram, shown in

Figure 5.8, is a structure similar to that in Graphic A of Figure 5.7. Although these differences are minor in this particular example of a design pattern, there are inconsistencies and these inconsistencies can be confusing to a novice developer. Further examples are shown in the next few pages that can be confusing to both novice and experienced developer.

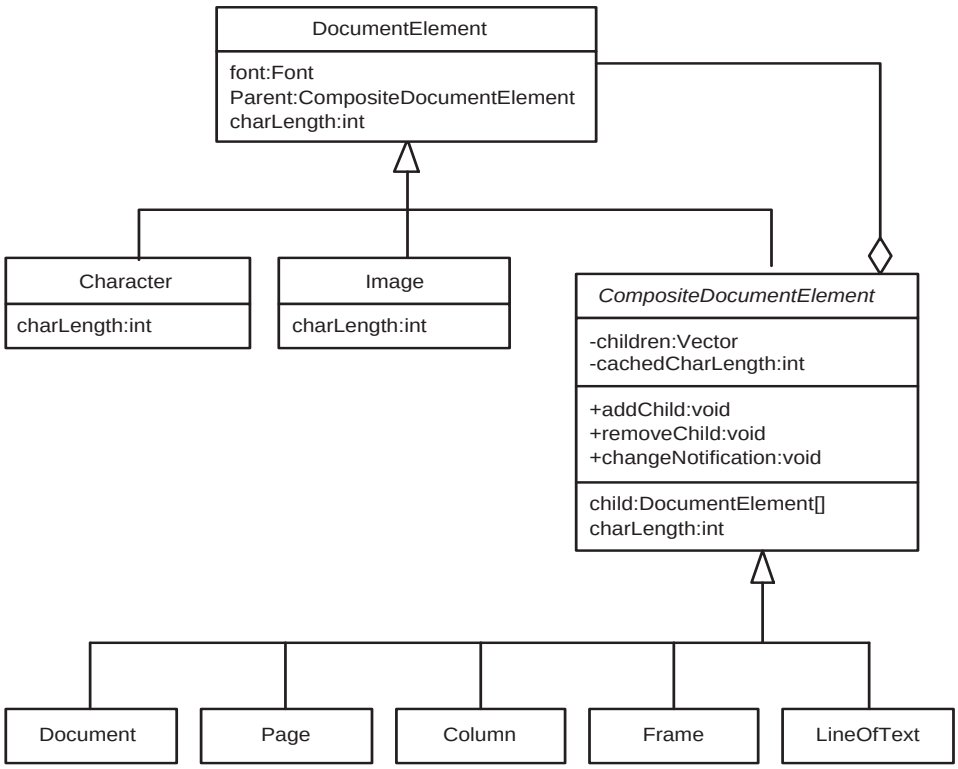

Figure 5.8: Reverse Engineered Composite Class Diagram[48]

It is true to say that the class diagrams of the Design Patterns[45] that are duplicated in the book by Grand[48] are replicas with, in some cases, minor modifications. It is also true to say that with few exceptions, the physical class diagram abstracted from the code through reverse engineering does not match the class diagram described in the book. However, Grand is not alone in his interpretation of modelling design patterns. Stelting and Massen[102] also interpret the Design Patterns[45] in the same way. In this catalogue of patterns, the classic model of the Design Patterns[45] is described in the book but the resulting model that is abstracted from the supplied code is significantly different. It is plain to see that in many of the patterns, several helper classes have been added to the application to embellish the example. However it is also plain to see that the models provided in the patterns themselves are not describing the real interpretation of the coded example.

This reworking of the defined structure is endemic throughout pattern catalogues and Internet related educational and informational discussion pages on design patterns. Consult the majority of web-sites

that discuss patterns, and a structure that replicates the structure defined in the Design Patterns[45] catalogue will be found. Yet, the resulting code, if code is provided, will paint a different picture. For example, the implementation of the Composite pattern on the Rice University[108] web-site reveals the standard modelling structure replicated from the Design Patterns[45] catalogue, but the resulting model abstracted through reverse engineering the source code presents a distorted view of the model. A further example[82] directly uses the discussion and structure from the Design Patterns[45] catalogue, but the resulting structure of the supplied code is different from the model.

Figure 5.9 below is a model of what pattern writers are describing in their discussion of the Composite pattern. This however, is not what is being created in their examples. Quite often in examples of the Composite pattern the aggregation association from the component to the composite will be missing or will just be a common association. Furthermore, the client, if there is one, will be calling the composite and leaf classes directly. To follow the design that is being modelled, the client should be interacting with the Component element of the pattern; declaration of Components within the Client class should be global; the Leaf and Composite classes should extend the Component class; and the association between the Component class and the Composite class should be an aggregation. The aggregation could be in the form of a static array or a dynamic vector.

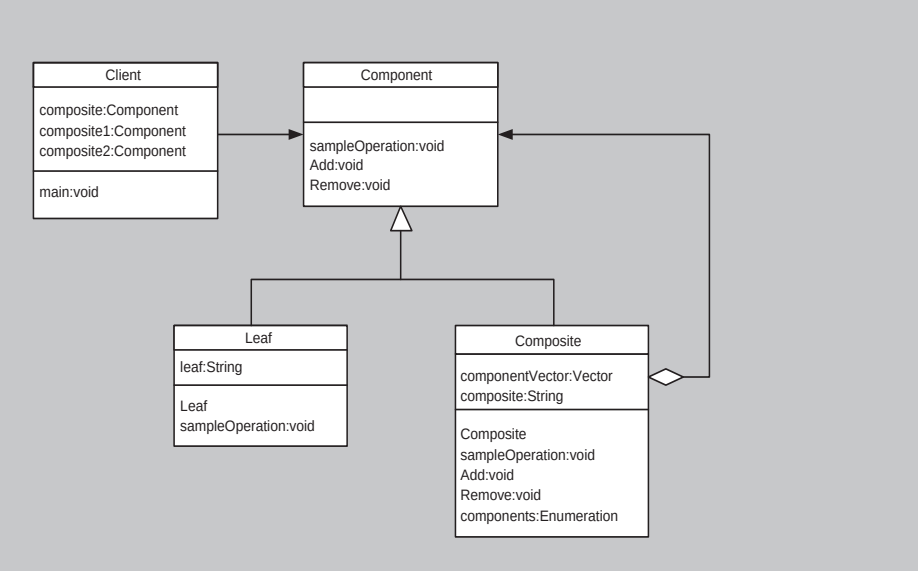

Figure 5.9: Composite Pattern

To this point, only the Structural pattern Composite has been discussed, but this situation where a pattern is modelled in a particular fashion, replicated from a popular book, is also evident in other structural patterns. Grand[48] again displays the structure defined in the Design Patterns[45] book for the Decorator pattern, but does not implement that same structure in the coded example.

Figure 5.10 provides a comparison of the model for the Decorator pattern as defined by Grand[48] and the model abstracted from his coded example.

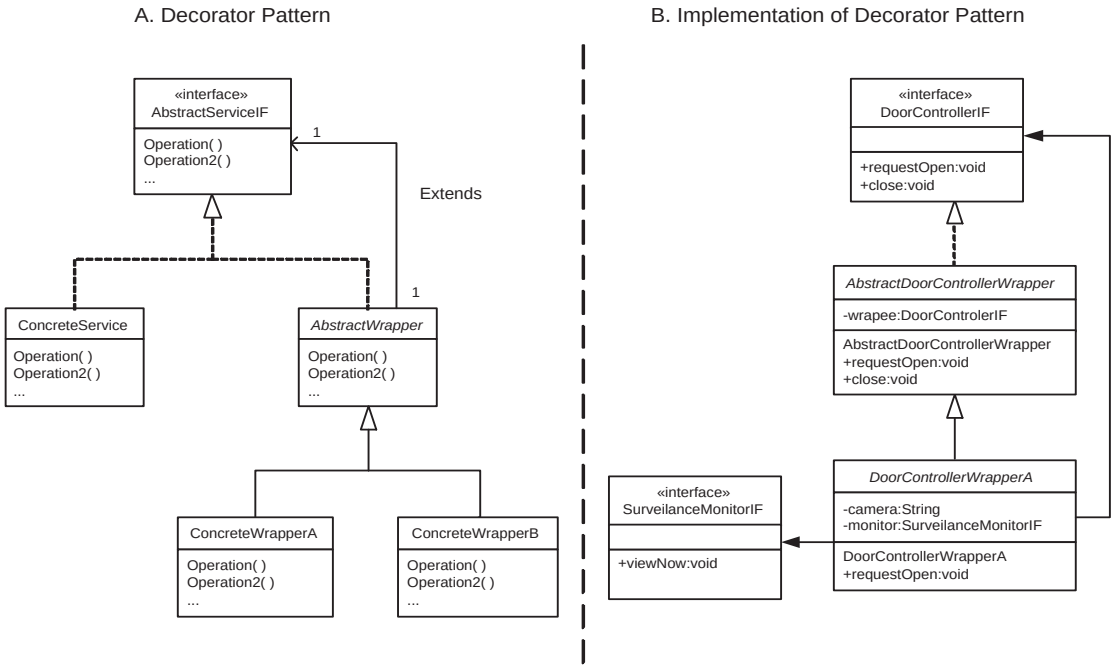

Figure 5.10: Grand's Decorator Class Diagrams

Changes to how a particular pattern is modelled and to how the supplied examples are modelled filters through to other classifications of models. For example, the Creational pattern, Builder, which is defined in the Design Patterns[45] catalogue, also receives an implementation makeover at the hands of numerous authors. This particular pattern, as can be seen in Figure 5.11 below, receives such a change to that described in the book from which it was extracted that it is unrecognizable as a Builder pattern. Similarly, the implemented structure of Behavioural patterns is afforded the same implementation shift from that of the original description. It would appear that the whole ethos of describing known patterns is to ignore the design written in the structure of the pattern.

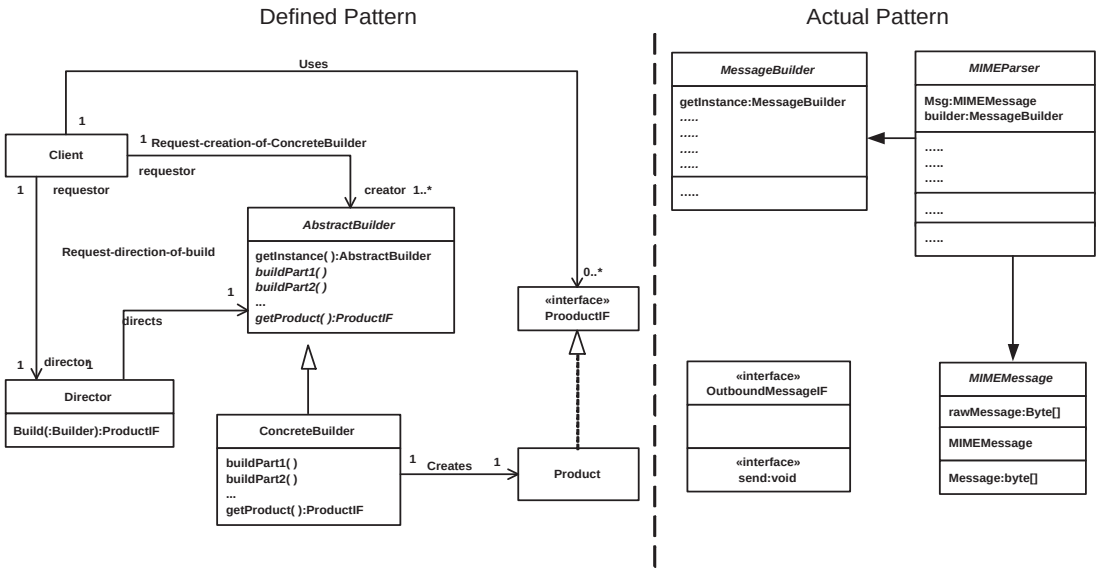

Figure 5.11: Grand's Builder Class Diagrams

The differences that are evident between the published model of a design pattern and that revealed in the supplied code examples is not restricted to Java based design patterns. The same discrepancy in models also applies to examples and patterns described in C# and other languages. The Composite patterns described on various community web-sites[87, 88, 32] also construct the model from the pattern as a near replica of the Composite class model described in the Design Patterns catalogue, yet the resulting models abstracted from the supplied code are different.

5.4 Summary

The discussion in Chapter Five began with a review of how sequence diagrams are used and presented in a range of design pattern catalogues. It was soon established that while they are used in only a subset of catalogues observed, how they are used and presented is an asset to the intended knowledge conveyed in a design pattern. Whilst some design patterns only pay a passing interest in extending knowledge with the aid of these diagrams, some pattern writers link the diagrams to different scenarios associated with the pattern and to alternative coded examples of the pattern in its use. From the point of view of defining a design pattern, the use of sequence diagrams linked to multiple scenarios and coded examples is seen as good practice and should be co-opted into the standard for defining a generative pattern.

Where class diagrams are concerned, it is easy to observe, through reading multiple catalogues, that diagrams of this type are used in almost all descriptions of a software design pattern. However, what

is not obvious is that what pattern writers are describing with a class diagram is not what they are producing in a coded example. It can be seen that most examples of well-known design patterns, first produced in the Design Patterns catalogue, are recreating the class diagrams used in that catalogue. What cannot be seen, until the coded examples have been investigated and re-engineered, is that writers are not producing an example based on the classic design they have described. They are creating an example modified from that design but not matching the code to the modified design. What can be seen in the description of many patterns from a diverse range of sources is that the writer is describing one thing and constructing another, which suggests pattern writers are being led by a traditional acceptance of the class structure first adopted in the Design Patterns[45] catalogue. Writers therefore are not challenging the traditional view of design notation, whilst they are interpreting the concept through example. Discussing patterns without acknowledging this interpretation is not conducive to extending the knowledge and understanding that is the raison d'etre of design patterns. One of the principles of the design pattern is that they are easy for novices to use. However, because there are often discrepancies between the supplied code and the design contained in many design patterns, the novice can easily be confused and turned off the concept of the design pattern. Furthermore, by being inconsistent, one could call into question the usefulness of design patterns in aiding software development. With this in mind, it is intended that the coded examples produced for the generative design patterns described in this work will match the design.

Chapter 6

A GENERATIVE DESIGN PATTERN

6.1 Introduction

In Section Two of this chapter, the output from the four previous chapters is brought together to serve as a reminder of the qualities that have been abstracted in order to define a generative pattern. Methodical qualities were discussed in Chapter 2 and a summary of the similarities that exist between software development methods and design patterns is given. In Chapter Three a range of pattern styles and catalogues were discussed and from these styles and catalogues a desired format for a pattern notation has been abstracted. This desired notation is again summarised prior to applying the notation in Section 6.4. It was shown in Chapter Four that different types of classification are attributed to design patterns and that these classifications can be used to describe the relationship between two different generative patterns. Finally, a summary of Chapter Five is presented where modelling notation is discussed.

In Section 6.3 and 6.4 the main contribution to this thesis is described in the format of a generative pattern. (The format of the generative pattern can be compared against the static pattern example in Appendix I). What distinguishes the generative pattern from a static pattern is the additional knowledge presented in the generative pattern. To present this extra knowledge, the generative pattern expands upon the principle of pattern classification by introducing a Problem Solving classification, as discussed in Chapter Four. Also from Chapter Four the knowledge of how collaborating patterns can combine or be used by other patterns is introduced. An additional contribution to the generative pattern comes from showing, in the coded examples, how patter 'X' will collaborate with pattern 'Y'.

Having gathered the desired notation and the additional knowledge for a generative pattern, a generative pattern is described. Within several of the previous chapters, several references are made towards the Composite design pattern. For this reason, the Composite pattern is used here as an example of a generative pattern. The Composite pattern, as described by several authors, is related to the Decorator pattern. Not only is it related in terms of its appearance as a structure, it can be combined with the Decorator pattern to form a larger structure. The generative design pattern seen in Section 6.4 is an example of two individual patterns, the Composite and Decorator combining to form a larger structure.

6.2 Generative Process

Software design patterns come in many different styles and each style has its own way of describing a pattern. Some styles use what is considered to be common forms of notation, common in that the notation is used consistently in different styles by different pattern writers. Other styles use less common notations but make passing reference to notations that are used in other patterns. Some pattern catalogues define a pattern under a specific set of headings, while other catalogues defining the same patterns will use some of the same headings but present them in a different order. Other catalogues use a different set of headings altogether. The defining issue is that most catalogues will use what is considered a popular design notation as well as obscure notations and it is these popular notations that have been exploited in defining a generative design pattern.

6.2.1 Summary of Chapter Two

In Chapter Two, design patterns were described as having a lot in common with development methods — the common factor being the elements of a life-cycle. It was found through analysis of methods that certain aspects of methods will map to aspects of design patterns and this has served to re-enforce the qualities obtained and discussed in Chapter Three. Some of the design models that are used in software development methods are appropriate notation for a generative design pattern.

6.2.2 Summary of Chapter Three

Chapter Three sought to provide an understanding of pattern notation and showed how specific pattern styles used different types of notation to format a pattern. Many of the notations are not used by all pattern writers but the frequency of use throughout different pattern notations suggests that the theme is an element of good practice and should be retained in a refactored pattern notation. The elements of common notation that are attributed to a range of design pattern types are to be used in the definition of a generative design pattern.

6.2.3 Summary of Chapter Four

In Chapter Four, two classification protocols and a relationship protocol were explored and a suitable communication diagram was defined, where a relationship between different classifications of patterns was established. It was also established how patterns can be classified by the type of problem they solve. This type of knowledge is used in deciding if a pattern is suitable to work with another pattern.

It also provides some idea of the function of the pattern itself. Individual relationships between patterns were discussed. From this, three relational types were established.

6.2.4 Summary of Chapter Five

Finally, Chapter Five took a closer look at modelling elements contained within patterns with a view to obtaining what may be considered as best practice. An element of good practice that was described in Chapter Five was the use of different scenarios within a pattern. This, it is felt, is in keeping with one of the principles of the design pattern in that there should be three known uses of the pattern. Therefore, it is proposed that a generative pattern should describe, as a minimum, three scenarios of the pattern in use.

6.3 Generative Pattern Format

The format of the generative pattern is based on what is already contained within existing static design patterns - that of a simple methodical process. Although design patterns do not present themselves as methodical components, the analogy is there. There are analytical elements, design components and implementation details encapsulated in what is the notation of a pattern. Many of these encapsulated details have been confirmed by examining the details of a range of development methods.

A collection of generative design patterns, if put to proper use, have the potential to produce a software system or subsystem. Therefore, it is desirable that some methodical process is engaged in the development of that system or subsystem. Whilst generative patterns are not intended to represent a methodical process, the format of the generative pattern is written in such a way that it mirrors the life-cycle aspects of a methodical process in terms of analysis, design and development.

The findings from Chapters Two to Five and summarised in Section 6.2 above can now be integrated into the profile of the generative pattern. As such, the profile and the desired notation form the current version of the structure of a generative design pattern. As an addition to current research, it is envisaged that through a process of refinements to the generative pattern profile a concrete and final profile will be established. From this, it is expected that the profile itself will undergo mathematical scrutiny from the process of Formal Specification — see Chapter Nine, Future Work.

Structure of a Generative Design Pattern

Name

Classification Type

Problem Solving Type

Analysis

- Intent (Introduction)
- Problem
- Solution

Design

- Structure - Class diagram – (Classic View)

Implementation

- Participants
- Code example

Related Patterns (Dynamics - Three Examples of Generative Design)

Scenarios 1, 2, 3

- (Analysis)
  - Details of scenario
- (Design)
  - Use-Case Diagram – (if applicable)
  - Activity Diagram – (if applicable)
  - Class Diagram – (Required as a model of the applied code)
  - Sequence Diagram – (if applicable)
- (Implementation)
  - Details of implementation
  - Participants
  - Coded Example
  - Consequences

6.4 Composite as a Generative Design Pattern

The following highlighted sections represent the notation of a generative design pattern. Each section is a specific aspect of the pattern as described above in Section 6.3. The source code for each of the scenarios can be seen in Appendix F.

The highlighted sections on pages 94 - 98 represent the profile of the named pattern, which is split into four areas. The four areas represent pattern classification, analysis, design and implementation – those aspects that define the pattern itself. The highlighted sections on pages 99 - 100 represent the relationship that Composite has with the Decorator pattern. The final sections, pages 100 - 104, highlight the Dynamics of the generative pattern as applied in Scenario 1.

Name – Composite

Classification type Structural.

Structural patterns are an arrangement of classes that together form a larger structure.

Solves Problem Type Variant Management.

Variant Management patterns treat different objects in a uniform manner by factoring out their common properties.

Analysis

Intent

Compose objects into a collection or collections of objects.

Composite describes an object that is composed of Composite objects. The Composite object is built as a collection of objects and can be defined by a Collection, ArrayList, Vector or simply an Array among other artefacts that define a collection of components. The Composite can be described as being a tree-like structure.

Continued on next page.

Problem

Often, developers require complex objects that can be manipulated in a dynamic fashion by users during the execution of a program. For example, in a drawing package drawn items come in different shapes and sizes and can be added to a drawing, removed from a drawing or resized, reshaped or repositioned. Components of a drawing can be grouped together or ungrouped; multiple grouped items can be collected into a single group. A document object may be composed of text, images and drawings, which can be grouped into Chapters, Sections and Subsections. The problem is to store these items in a uniform manner as individual items in a collection of items. Each of these examples represent complex objects that need to be manipulated.

Solution

Provide a Composite object to store individual or composite objects. Clients can build and access the composite through an interface component that is implemented by all components in the Composite structure. Collections such as ArrayList and Vector make ideal components to store composite objects.

Design

Structure

The Structure represents the classic view of Composite as described in the Design Patterns[45] catalogue.

Continued on next page.

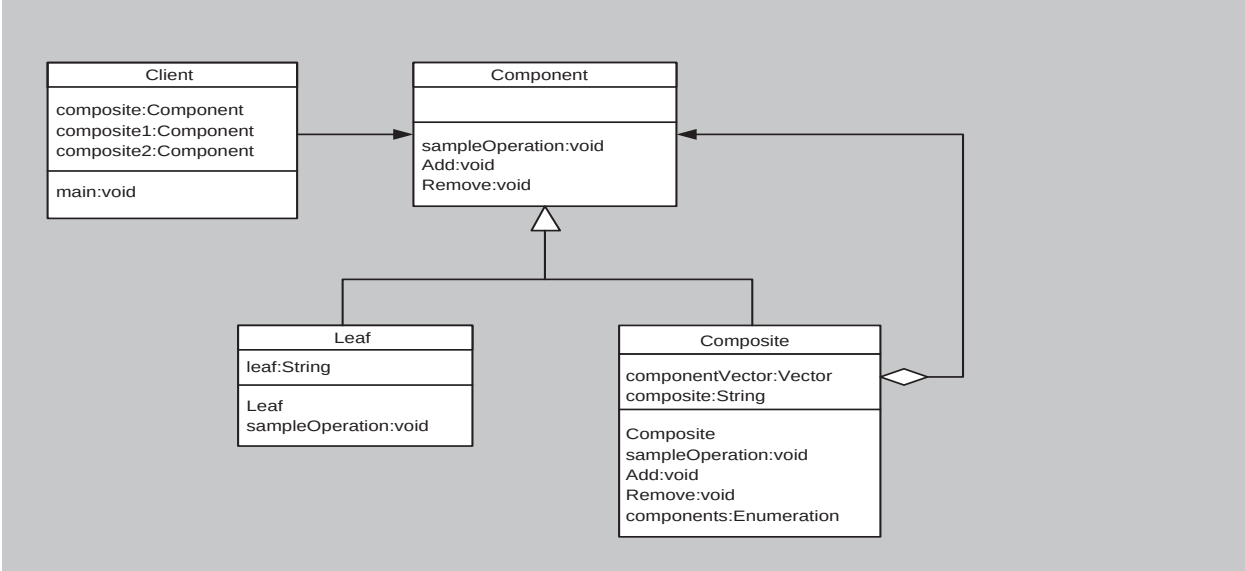

Figure 6.1: Structure of the Composite Pattern

Participants

Client

A client can create or manipulate the composed collection through the interface component.

Component

Represents an interface for objects in the composed collection.

Defines common methods for child components

Composite

Defines a composite object that has children

Stores the collection object.

Leaf

Represents an individual or leaf object that has no children.

Continued on next page.

Implementation

Sample Code

The following code, in Java, is a demonstration of how the implementation follows the design:

Example 6.1 **Client.java**

```
public class Client{
   protected static Component composite = new Composite("Composite 0");
   protected static Component composite1 = new Composite("Composite 1");
   protected static Component composite2 = new Composite("Composite 2");
   public static void main( String arg[] ){
      try{
         Leaf leaf1 = new Leaf("Leaf 1");
         Leaf leaf2 = new Leaf("Leaf 2");
         composite1.add(leaf1);
         composite1.add(leaf2);
         Leaf leaf3 = new Leaf("Leaf 3");
         Leaf leaf4 = new Leaf("Leaf 4");
         composite2.add(leaf3);
         composite2.add(leaf4);
         composite.add(composite1);
         composite.add(composite2);
         composite.sampleOperation();
      }catch( Exception e ){e.printStackTrace();}
   }
}
```

Continued on next page.

Example 6.2 **Component.java**

```
public class Component{
   void add(Component component);
   void sampleOperation(){System.out.println("Component Operation");}
}
```

Example 6.3 **Leaf.java**

```
public class Leaf extends Component{
   private String leaf;
   public Leaf(String leaf){this.leaf = leaf;}
   public void sampleOperation(){System.out.println(leaf);}
}
```

Example 6.4 **Composite.java**

```
import java.util.Vector;
import java.util.Enumeration;
public class Composite extends Component{
   private Vector componentVector = new Vector();
   private String composite;
   public Composite(String composite) {this.composite = composite;}
   public void sampleOperation(){
      System.out.println(composite);
      Enumeration components = components();
      while (components.hasMoreElements()){((Component)components.nextElement()).sampleOperation();}
   }
   public void add(Component component){componentVector.addElement(component);}
   public Enumeration components(){return componentVector.elements();}
}
```

Related Patterns

Decorator (See Decorator Pattern)

Provide a way of adding functionality or decoration to an object.

Classification type **(Structural)**

The connection of two or more Structural Patterns serves to form a larger structure.

Problem Solving Type **(Decoupling)**

Why the decoupling takes place.

Decoupling helps to divide a system into independent units. A system that includes decoupled elements can easily be extended or adapted by adding or modifying those elements.[104].

What the decoupling will add to a system

The Decorator provides decorative or functional embellishment of objects that were created separately. Decoration is applied to the object rather than being part of the object.

By providing a specified functional construct, decoration of the object can change without affecting the object. That is, the decoration is decoupled from the object.

How the decoupling takes place.

An object can be created that is defined with specific functionality. The decorative component can be included in that functionality. However if the object is modified then the decorative functionality may have to be modified also. For this reason the decorative component is decoupled from the objects construction by providing alternative classes to handle the decoration.

Association Type **(Combines (Decorator))**

The interface components of both Composite and Decorator combine to form a single interface. The Composite element of the pattern supplies the collection object for the combined patterns whilst the Decorator element acts as an interface to the ConcreteDecorator components. The Leaf elements of Composite retain their original purpose and functionality.

Continued on next page.

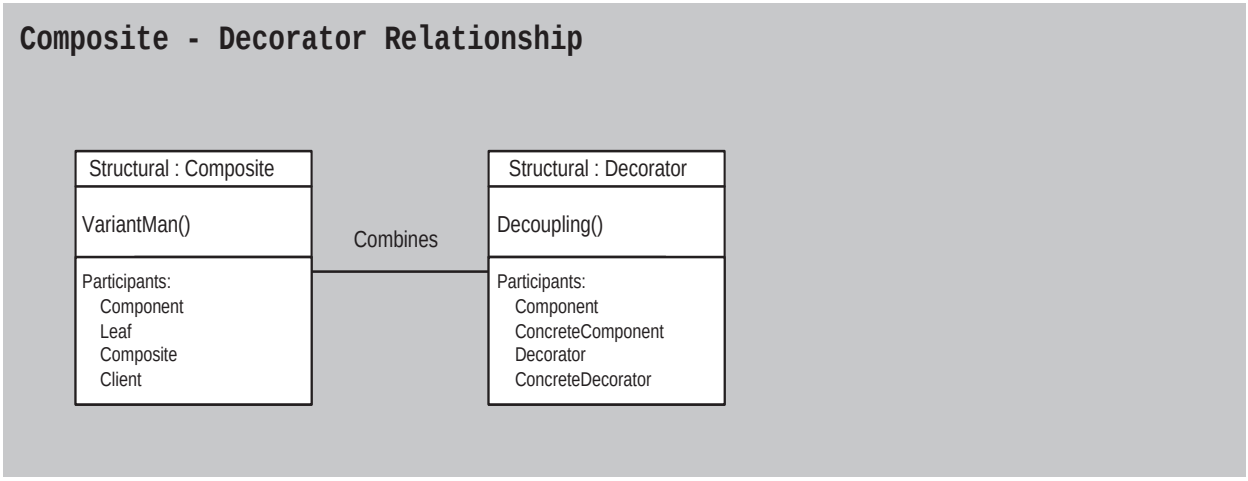

Figure 6.2: Relationship between Composite and Decorator

Dynamics – Examples of Generative Design

Scenario 1

Analysis

Scenario 1 illustrates a simple drawing package where lines, squares and circles can be drawn within a frame. Each drawing item can be individually decorated or a group of drawing items can be decorated. Each individual item and or groups of items can be collected into a composite object.

Design

Use-Case Diagram

The use-case diagram represents a business process that defines the activities that can be applied to the drawing scenario. In this case, drawing components can be created, decorated and displayed.

Continued on next page.

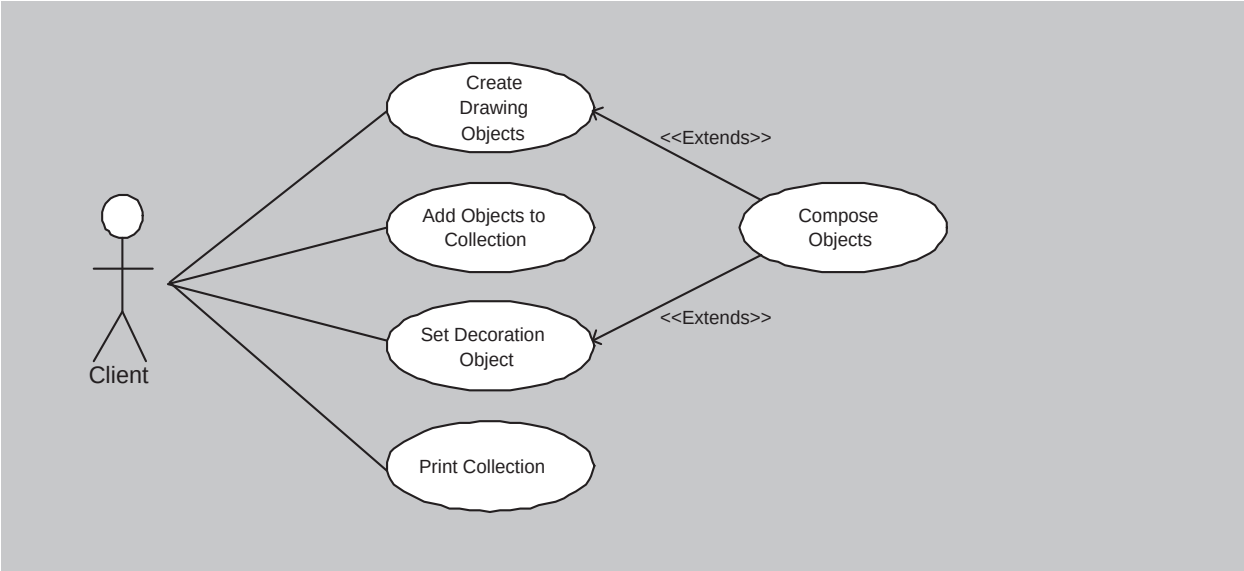

Figure 6.3: Use-Case Diagram - Composite combines Decorator

The diagram on the following page shows the class components that collaborate to form the structure of the Composite – Decorator drawing scenario. Three different drawing components can be created and can be decorated with colour and or line weighting can be applied (thickness of lines).

Class Diagram

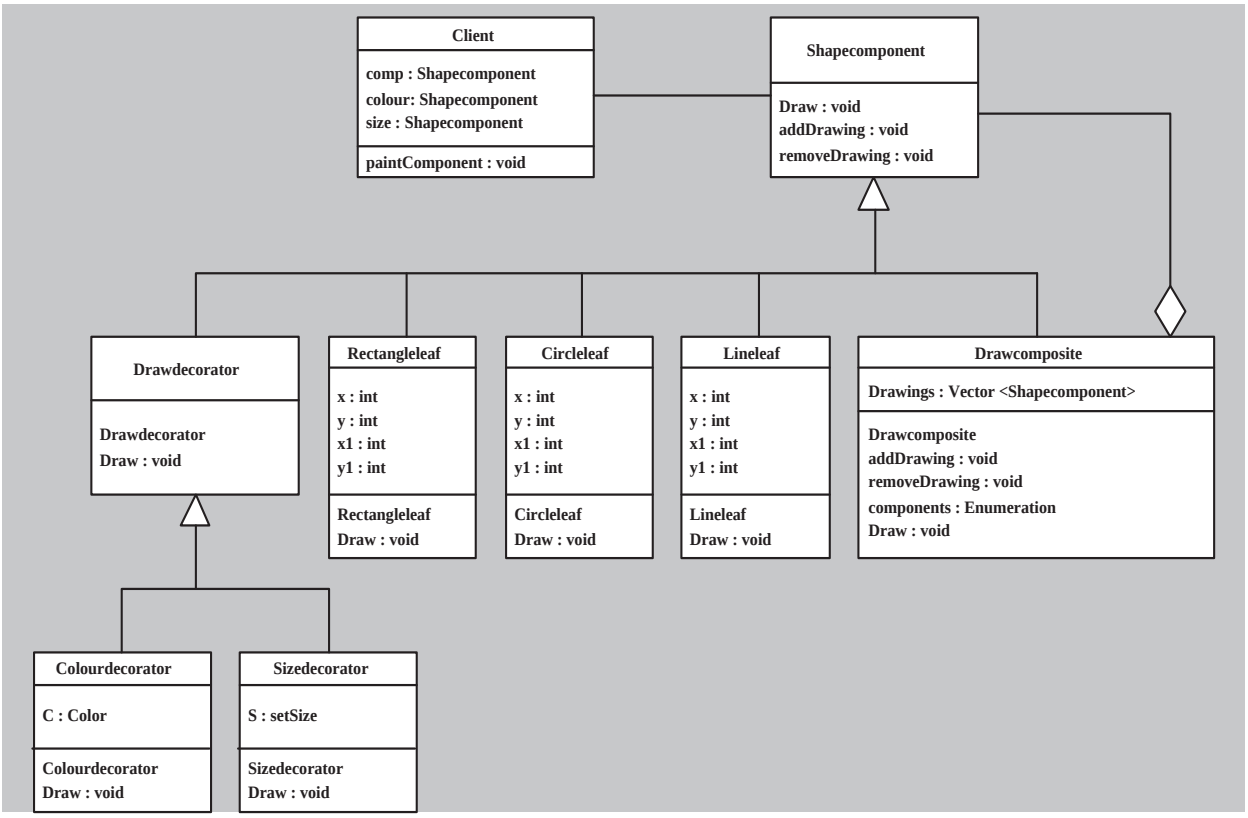

Figure 6.4: Class Diagram - Composite combines Decorator

Implementation

This example uses the Composite and Decorator patterns to demonstrate a simple drawing package. For the purpose of demonstration the components are hard coded into the client but in a live application the components would be created dynamically.

Continued on next page.

Participants

Client

The Client component is a simple driver used to create the drawn components. The client creates the component and decoration objects and adds them to the collection object.

Shapecomponent

The Shapecomponent class specifies an Abstract interface to the main components of the Composite and Decorator. Shapecomponent defines three methods that can be implemented by all sub-classes. The addDrawing(Shapecomponent draw)and removeDrawing(Shapecomponent draw) methods are implemented in the Composite class and the Draw(Graphics g) method is implemented in all sub-classes.

Drawcomposite

Drawcomposite has two functions; one is to add or remove items from the Collection object (the Vector) - private Vector<Shapecomponent> drawings; and the other is to call back the items from the collection (print to the frame) - ((Shapecomponent)components.nextElement()).Draw(g). In this example items are only added to the collection - addDrawing(Shapecomponent draw){drawings. addElement(draw)}.

Lineleaf, Circleleaf, Rectangleleaf

Leaf components represent the drawing objects that are added to the composite collection. Each component defines its own type of drawing object, which is called in the Draw method - g.drawLine(x, y, x1, y1).

Drawdecorator

Drawdecorator specifies an Abstract interface whose Draw method is implemented in the Colour and Size sub-classes. Like the Shapecomponent class Drawdecorator is something of a Facade in that access to sub-classes is only made through the Facade.

Continued on next page

Colourdecorator, Linestyledecorator

The decoration objects that are used to set the decoration for the drawn components. Each component defines its own type of decoration, which is called in the Draw method - g.setColor(colour). Because they are Shapecomponents decoration is added to the composite collection as an object.

Consequences

The main benefit of linking Composite and Decorator together is the separation of functionality into specific class components. This itself will bring easier maintenance to the system in that leaf items can be added or removed without affecting the decoration and decoration can be changed without affecting leaf items. The drawback to this is the level of functionality required within the client to manipulate the drawing objects within the collection.

The source code for this scenario and the remaining two scenarios can be seen in Appendix F.

6.5 Conclusion

Redefining design patterns as generative has not been as simple a process as deciding to add some detail to a pattern and calling it generative. What has been conducted is a systematic study of design patterns to find out how to define them appropriately, consistently and how to implement the relationships between the collaborating patterns.

There are a number of obvious differences between static design patterns and the generative pattern examples presented in Chapter Six and the Appendices. Notably:

- The generative pattern identifies a life-cycle process (a mini methodical process) that is not readily recognisable in a static pattern.
- Some static patterns are quite complex and it is difficult to identify within these patterns what aspects of the notation are the Problem and the Solution. In the generative pattern the Problem and Solution have a prominent position in the notation of the pattern.
- Most patterns offer only one coded example of a working pattern whilst the generative pattern has three such examples.

- Studies have shown that design patterns do work well together[22, 77, 91, 98], but do not describe openly a method of pattern integration. Therefore, whilst static patterns focus directly on the named pattern, the generative pattern identifies other patterns with which it will communicate and the components of the collaborating patterns that facilitate that communication.

- Although static patterns describe some problem in their narrative, the pattern only describes what the problem is in relation to the static nature of the named pattern. The generative pattern introduces a problem type which offers an idea of the functionality of other collaborating patterns, thereby facilitating a possible solution in conjunction with another pattern.

- In static design patterns the section on Related Patterns is extremely brief. The generative design pattern by its very nature treats Related Patterns as being the significant contribution to the pattern itself.

The above points highlight the differences between the static design pattern and the generative design pattern. An evaluation of the generative design pattern and the example experiments that were carried out on collaborating patterns is discussed in Chapter Seven, Evaluation.

6.6 Summary

The concluding aspect for this Chapter is the definition of a generative design pattern. To this end a pattern has been written that shows the definition in use. The Composite pattern has been defined in the created format for the generative pattern and is shown as an example of generative design, collaborating with the Decorator pattern. In composing the Composite pattern in this generative format a significant observation can be made. The reader will notice that there is a lack of detail in the pattern compared to other definitions of software design patterns contained in prominent pattern catalogues. There are two reasons for this; firstly, the generative design pattern described in this Chapter is not about the knowledge or the content of the pattern, it is about the framework of the pattern and how it is defined in a generative format; secondly, the content of the notation as it is written is not intended to be read in a book-based catalogue, therefore fine detail has been omitted. That is, if detail is required for distribution in catalogue form, then detail can be added. However, there is sufficient detail to recognise the process of generative design.

Chapter 7

EVALUATION

7.1 Introduction

Studies have shown that design patterns do work well together[22, 77, 91, 98], but do not describe openly a method of pattern integration. Although there have been a few attempts to take standard design patterns and define them as generative, these attempts are related to tool development that will generate code from design patterns[17, 18, 40, 73]. The term generative in this context relates only to the fact that patterns are used to generate code.

This chapter concludes the work undertaken in defining a generative pattern for the purpose of generating systems. The groundwork has been laid for the re-engineering of design patterns with a view to using the defined notation as a template for design and development. Currently, static design patterns are used as a solution to an individual problem in the development of a system, but with generative design patterns multiple problems can be brought together to simplify the development of a system.

However, redefining design patterns as generative is not as simple a process as deciding to add some detail to a pattern and calling it generative. What has been conducted is a systematic study of design patterns to find out how patterns do or do not work well together, how to define them consistently and how to implement the relationships between collaborating patterns.

In Chapter Seven, the patterns that have been considered in the main text and the appendices of this study are evaluated for their appropriate quality and usefulness as a development artefact. This quality and usefulness is determined by comparing the use of static and then generative design patterns in several simple and more complex case-studies. Firstly, the composite and decorator design patterns are evaluated in a simple desktop based scenario. Secondly, the use of three and then four design patterns are evaluated using the same simple desktop based scenario. Finally, a more complex application is developed and compared using the same collection of static and generative design patterns. Three additional paired pattern case-studies are evaluated in Appendix H, which contribute to the overall results of the evaluation. The static and generative scenarios are evaluated and compared using metrics to establish the quality of the developed applications in terms of the basic software principles of coupling, cohesion and complexity.

7.1.1 Evaluation Strategy

In an ideal situation the generative design patterns would be independently tested by teams of software developers. Team A would develop test applications using generative patterns and team B develop test applications using static patterns. However, this is not a realistic proposition. It is unlikely that a software company would allocate teams of developers to an evaluation process for an academic study without payment.

However, there are alternatives to securing the assistance of professional developers. As a second resort the evaluation could be put in the hands of computing students who could act as study groups for the testing of software development projects. However, this itself has its own set of problems:

- The students may not be willing to participate.
- They may lack the necessary education in design patterns.
- Separate control groups may be unbalanced.
- A single group will develop prior learning from the static or generative pattern development process, which will enhance the students' ability with the second example study, thereby creating an unbalanced comparison.

As a necessary alternative to evaluation by independently conducted experiments, dependant experimentation can be conducted on the case-studies. In this instance, the author of the generative design patterns can develop the case-studies, conduct the experiments and evaluate the results. A problem with this is the author could deliberately or unconsciously put a bias on ensuring the results of any experiments were in favour of the generative patterns. To counter this problem the code and test results can be independently checked and / or the code and results made available for public scrutiny.

The pragmatics of case-studies and the evaluation process can be flawed for many reasons and attempting to overcome those flaws to ensure unbiased or independent results can be a major task in itself. Securing the assistance of independent developers, whether industrial or academic, is initially dependant on the willingness or availability of the persons approached to do the task. Time is the real problem in securing independent evaluation.

Self evaluation, although not ideal, is from a practical point of view the best that can be achieved given the limited time available. Given more time a more independent approach could have been taken.

However, this approach was not available. Therefore, a self evaluation has been conducted on the generative design patterns.

7.2 Metrics

Metrics as the principle evaluation criteria have been used to determine the usefulness of generative design patterns. The Borland Together[16] modelling tool that was used for modelling the generative design patterns lists eighty eight different metrics tests. The full list of these metrics can be seen in Appendix G. Rosenberg and Hyatt[93] propose a range of both traditional and object-oriented metrics for testing object oriented systems. The metrics they use are useful in a wide range of models and evaluate the following attributes:

- Efficiency
- Complexity
- Understanding
- Reuse
- Testing
- Maintenance

The metrics used for the evaluation of the generative design patterns defined in this thesis are a subset of the metrics proposed by Rosenberg and Hyatt. Although the metrics used here are borrowed from Rosenberg and Hyatt based on what they consider to be appropriate metrics in object-oriented environments, the metrics used are commonly discussed throughout many papers[8, 21, 90] and texts[57, 68, 71].

The following metrics are used in this study to evaluate the generative design patterns:

1. **Coupling Between Objects (CBO)** CBO represents the number of other classes to which a class is coupled. It counts the number of reference types that are used in attribute declarations, formal parameters, return types, throws declarations, local variables, and types from which attribute and method selections are made. Excessive coupling between objects is detrimental to

modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. The larger the CBO figure for a class, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object coupling, the more rigorous the testing needs to be. CBO evaluates efficiency and reusability. The Together modelling tool recommends an upper limit of 30 for this metric, where a higher number represents a higher degree of required testing.

2. **Cyclomatic Complexity (CC)** Cyclomatic Complexity[78] is used to evaluate the complexity of methods within a class rather than the class itself for reasons of inheritance. Ideally, a low number should be returned, preferably below ten. However, there is a slight drawback to CC in that a low figure could be returned because decisions are deferred through message passing, not because the method is not complex[93].

3. **Lack Of Cohesion Of Methods (LCOM)** LCOM[57] measures the degree of similarity between methods in a class. A low value indicates good class subdivision, implying simplicity and high reusability. A high lack of cohesion increases complexity, thereby increasing the likelihood of errors during the development process[16]. Cohesion can be measured by calculating the percentage of methods that use a data field. Average the percentages, then subtract from 100. Lower percentages indicate greater data and method cohesion within the class. High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during development. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. This metric evaluates efficiency and reusability[71, 93]. The Together modelling tool recommends an upper limit of 101 and a lower limit of 30 for this metric, where a higher number represents lower cohesion.

4. **Lines Of Code (LOC)** Lines Of Code is the number of lines of code in a class, including comments and empty lines. A large class may pose a higher risk to understandability, reusability, and maintainability[71, 93]. There are no recommended figures for how many lines of code there should be in a class, although the Together[16] modelling tool defaults to a 1000 line upper limit.

5. **Response For Class (RFC)** The size of the response for the class includes methods in the class's inheritance hierarchy and methods that can be invoked on other objects. A class, which provides a larger response, is considered to be more complex and require more effort in testing than one with a smaller response figure[16]. If a large number of methods can be invoked in response to a message, testing and debugging the class requires a greater understanding on the

part of the tester. This metric evaluates understandability, maintainability, and testability[71, 93]. The Together modelling tool recommends an upper limit of 50 for this metric, where a higher number represents a higher degree of required testing.

6. **Weighted Methods Per Class (WMPC)** WMPC is the sum of the complexity of all methods for a class, where each method is weighted by its cyclomatic complexity. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class[16]. A class with a large numbers of methods is likely to be specific to an application, which will limit its possibility of reuse[93]. This metric measures understandability, reusability, and maintainability[71, 93]. The Together modelling tool recommends an upper limit of 30 for this metric, where a higher number represents greater complexity.

7. **Number of Classes** The NOC metric is very simple, it counts the number of class and interface components in the application.

8. **Executable Size** The size of the binary files in Kilobytes. Although this is not a specific metric it is used by Arnout[7] in her comparison of systems that do and do not use componentised patterns. In her thesis she presents a set of patterns that have been componentised into a set of library classes. A system is then compared with and without these classes and the size of the executable application is measured.

7.3 Static vs. Generative Patterns

7.3.1 Introduction

The following evaluation provides statistics and a discussion of each case-study / experiment conducted on the generative patterns using metrics. In each case-study, the same generative patterns are compared against the same static patterns in the same scenario. The scenario is a representation of a coffee shop where a drink can be purchased. When the drink is purchased a description of the drink and the cost are displayed on the output window of the application.

All the patterns used in these case-studies were first discussed in Chapter Six and Appendices A to D. The first case-study in this chapter and the three case-studies in Appendix H consist of pairs of patterns. The second and third case-studies contain three and four patterns respectively. The final case-study uses all four patterns but in a larger application.

For each case-study, a comparative model of the generative and static design patterns is provided.

The first table in each case-study provides the general statistics for the compared applications. The Together modelling tool that provides the metrics uses the highest value obtained from the individual class components as the benchmark figure for the application as a whole. Therefore the figures in the first table represent the overall project statistics.

The second table provides the statistics for two types of individual class components:

1. Those that have a like-for-like component in the comparative application but show different statistical results.

2. Those that have no corresponding component in the comparative applications, where a comparison cannot be made.

Both the first and second table of each case-study indicates whether there is a positive or negative difference between the comparative examples or whether there is no difference at all.

Table Key

- + Represents a positive result in favour of the generative pattern.
- − Represents a negative result against the generative pattern.
- / Indicates that there is no difference between the comparative examples.
- ***** Indicates that there is no corresponding component with which to compare.
- A blank space indicates that the metrics compiler did not return a value.
- **GP** A component from a generative pattern.
- **SP** A component from a static pattern.

7.3.2 A Simple Case Study using Composite and Decorator

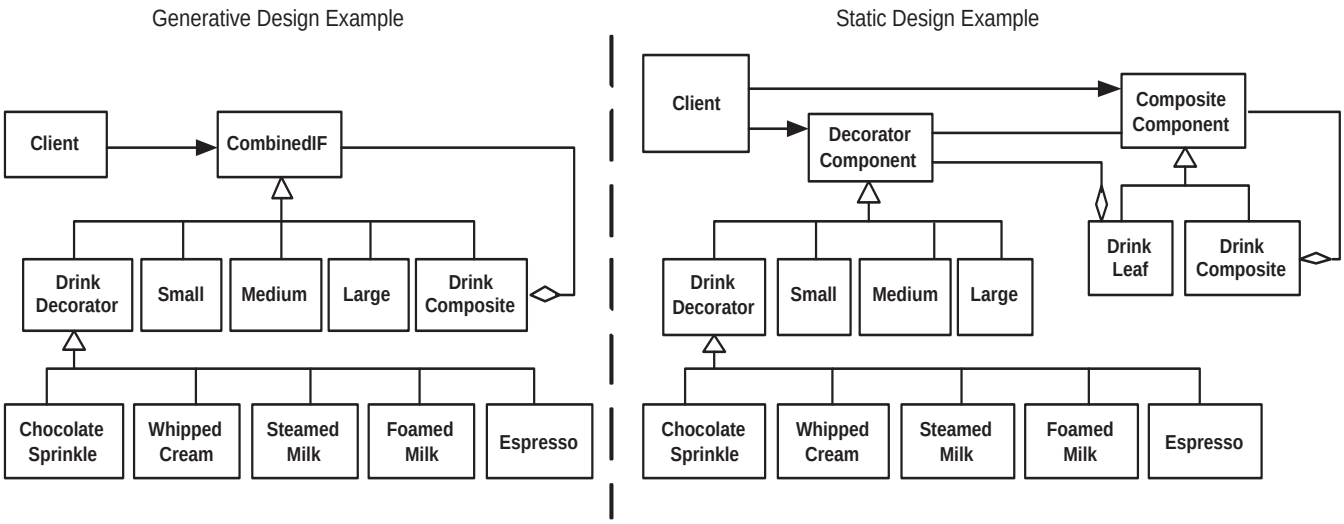

Figure 7.1: Generative vs. Static – Composite and Decorator

Figure 7.1 above provides a class diagram for the comparative examples of the composite and decorator patterns used in a generative and static pattern environment. As can be seen from the diagram, the generative example on the left has an interface that is combined from the two interface components that are used in the static example on the right. The four sub-components of the DecoratorComponent from the static pattern example are now leaf components to the DrinkComposite class in the generative pattern example.

In order for the two patterns to work together in the static environment, a decorator object is created and added to a collection object in the DrinkLeaf component of the composite pattern. As such, multiple decorator objects can be added to one or more DrinkLeaf components and one or more DrinkLeaf components can be added to a DrinkComposite component. DrinkComposite components can be added to other DrinkComposite components as is intended with a composite pattern.

In the generative example, because any decorator object that is created is now a leaf component to the DrinkComposite component, it can be added directly to a DrinkComposite object.

Table 7.1 shows the overall results of the metrics that were produced from the generative and static examples of the composite and decorator patterns described above.

Metric	Generative Patterns	Static Patterns	Difference (%)
CBO	11	13	+
CC	2	2	/
LCOM	100	100	/
LOC	242	278	+12.9%
RFC	31	33	+
WMPC	5	6	+
NOC	12	14	+
EXE SIZE	10.0	12.1	+17.4%

Table 7.1: General statistics for the Generative and Static versions of Composite and Decorator

What the statistics in Table 7.1 indicate, which is confirmed by the CBO and RFC metrics, is that the generative pattern example will require slightly less testing than the static pattern example. As can be seen in Table 7.2 the higher values for the CBO and RFC metrics comes from the client of the static example, which is having to communicate with two interface components instead of just one interface component in the generative example.

Whilst a like-for-like comparison cannot be made between components that have no counterpart, the additional components in the static example have something of an overhead in terms of complexity. The overall value of the WMPC metric suggests that the generative example is slightly less complex. The higher complexity value in the static example comes from the CompositeComponent, which defines two sets of methods (add(Component drink) and remove(Component drink) in the composite, and add(CoffeeProduct drink) and remove(CoffeeProduct drink) in the leaf) for the collection components defined in each subclass of the interface. Significantly, the joint values for the interface components in the static example are double that of the single interface in the generative example.

Where the two separate interfaces are concerned (DecoratorComponent and CompositeComponent) in the static example, the statistics in Table 7.2 show that the overall testing figures for the CBO and RFC values are double that of the CombinedIF interface component of the generative example. So, whilst the client in the static example will require more testing than that of the generative example, the static example, yet again, will require more testing in the individual interface components plus some additional testing for the DrinkLeaf component.

Three other significant points in favour of the generative example are the reduction in the number of lines of code, the reduction in the number of classes and the size of the executable file. As can be seen

in this example, the executable file for the static patterns is over 17% higher, whilst it has almost 13% more code and two extra classes.

Class	**CBO**		**CC**		**LCOM**		**LOC**		**RFC**		**WMPC**	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Client	11	13	1	1	86	86	56	56	31	33	5	5
	+		/		/		/		+		/	
Class	**CBO**		**CC**		**LCOM**		**LOC**		**RFC**		**WMPC**	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
CombinedIF	0	*	1	*		*	7	*	4	*	4	*
	*		*		*		*		*		*	
Composite	*	1	*	1	*		*	10	*	6	*	6
Component	*		*		*		*		*		*	
Decorator	*	0	*	1	*		*	5	*	2	*	2
Component	*		*		*		*		*		*	
Drink	*	4	*	2	*	0	*	28	*	8	*	5
Leaf	*		*		*		*		*		*	

Table 7.2: Individual statistics for the Generative and Static versions of Composite and Decorator

The individual class statistics for the like-for-like components in the examples are identical throughout all metric categories, therefore they are not included in Table 7.2. In this example this equates to the components that make up the decorator and the composite class elements of the application. The reason for this is modularity, in that each corresponding component provides identical functionality. The only exception in like-for-like components is the client. For the client there is a minor difference in that it communicates with two separate interfaces.

Three additional paired pattern case-studies can be seen in Appendix H.

7.3.3 A Simple Case Study using Composite, Command and Builder

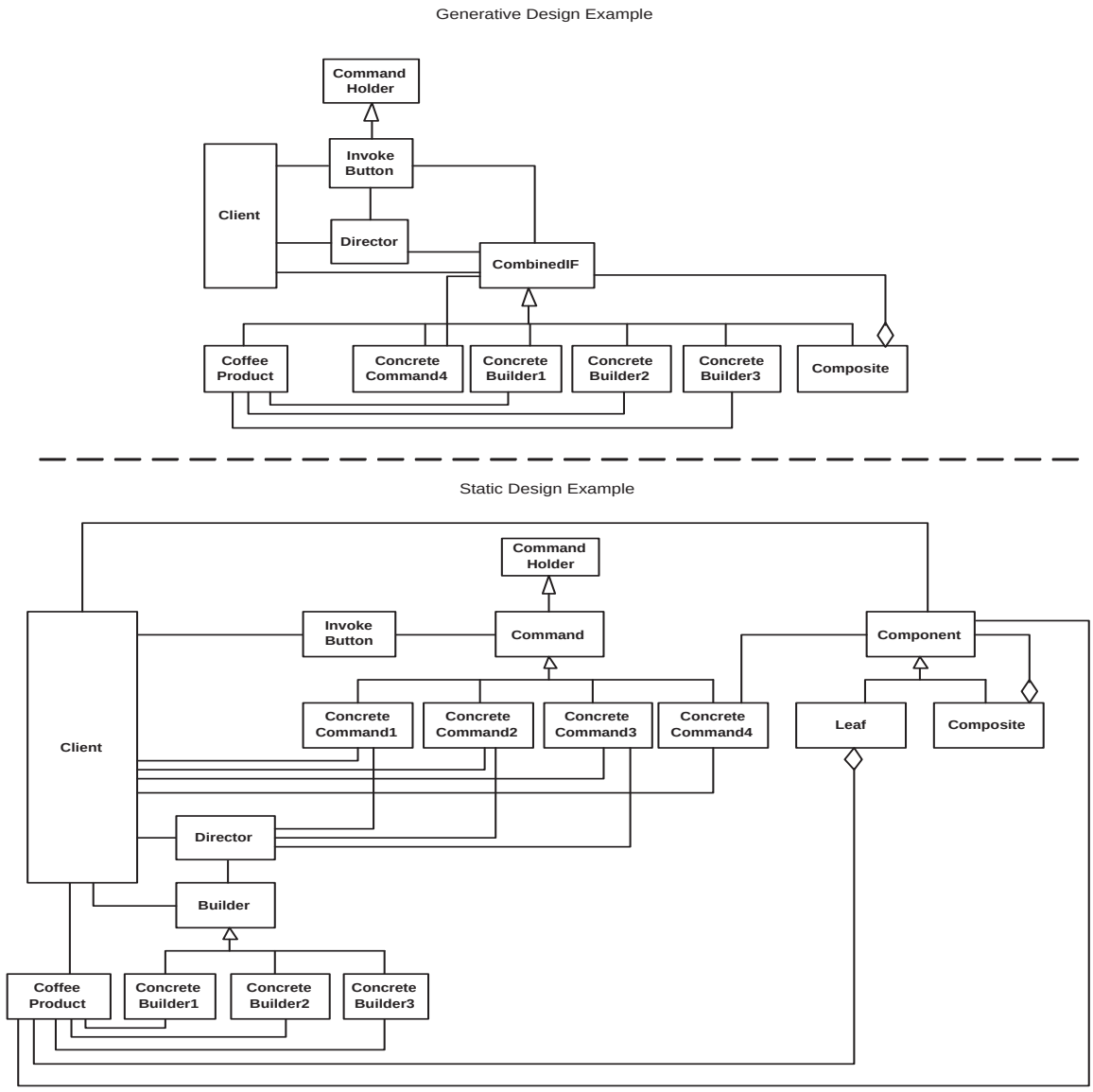

Figure 7.2: Generative vs. Static – Command + Composite + Builder

Figure 7.2 above provides a class diagram for the comparative examples of the composite, command and builder patterns used in a generative and static pattern environment. This example of the three patterns is very similar to the command and builder examples seen in Figure H.2 of Appendix H. However, other than the inclusion of the composite pattern, there are some slight differences.

In the generative example, both the command and builder patterns are being combined with the composite pattern, but the builder pattern is still only using the InvokeButton from the command pattern. Additionally, although ConcreteCommand4 from the command pattern shares an interface with the composite pattern, ConcreteCommand4 is not strictly a leaf component of the composite as it

is not practical to add ConcreteCommand4 to the composite object. ConcreteCommand4 in this instance is using the composite to extract information for later use.

The builder and command patterns are collaborating as they do in the command and builder example seen earlier, except in this example the CoffeeProduct object that is being created is now being stored in the composite object.

In order for the patterns to work together in the static environment, a product object is built and created when a ConcreteCommand is issued through an InvokeButton command. The ConcreteCommand instructs the ConcreteBuilder to build the product; the product is then added to a collection object in the Leaf component of the composite pattern. As such, multiple product objects can be added to one or more Leaf components and one or more Leaf components can be added to a Composite component. Composite components can be added to other Composite components as is intended with a composite pattern.

In the generative example, because any product object that is created is now a leaf component to the Composite component, it can be added directly to a Composite object.

Table 7.3 shows the overall results of the metrics that were produced from the generative and static examples of the composite, command and builder patterns described above.

Metric	**Generative Patterns**	**Static Patterns**	**Difference (%)**
CBO	20	26	+
CC	17	17	/
LCOM	88	88	/
LOC	428	478	+10.5%
RFC	14	11	–
WMPC	14	11	–
NOC	11	17	+
EXE SIZE	18	20.7	+12.1%

Table 7.3: Code statistics for the Generative and Static versions of Command + Composite + Builder

The statistics in Table 7.3 show that the static pattern will require more testing in respect of the CBO metric than that of the generative pattern. With three patterns in these examples, the client in the static example has to communicate with two more interface components than the client in the generative example. As such the CBO metric in the static example is thirty percent higher than that

in the generative pattern example. This represents a higher degree of coupling in the static example and a higher degree of testing of the client.

However, as can be seen from Table 7.3, the RFC metric of the generative pattern is thirty percent higher than the static pattern. This higher value comes from the CombinedIF component of the generative pattern, which is listed in Table 7.4. This is a result of the CombinedIF component defining methods for three different types of subcomponent. As a result, there will be a need for more testing of the CombinedIF component. Additionally, the WMPC metric shows that the complexity of the generative example is higher than the static example. Again the higher value for this metric comes from the CombinedIF component which is acting as a communication hub between the client and the remaining class components in the example.

A significant negative aspect of the generative example is shown in the statistics of the InvokeButton, which can be seen in Table 7.4. The InvokeButton as it is used in the generative example is providing two sets of methods that are used to set the commands for the ConcreteCommand4 component and the ConcreteBuilder components:

```
public void setCommand(CoffeeDirector comd);

public CoffeeDirector getCommand();

public void setPriceCommand(CombinedIF comd);

public CombinedIF getPriceCommand();
```

In a regular command pattern there would only be one pair of set and get methods. As such, the LCOM value is significantly higher for the generative example - indicating that the InvokeButton of the generative example is less cohesive than the static example. Likewise, the CommandHolder interface which provides the implementation details for the InvokeButton also exhibits the negative values for its metrics.

Whilst the three separate interfaces (Builder, Command and Component) in the static example still have collective values lower than the CombinedIF interface of the generative example, the static example will require some additional testing and maintenance for the ConcreteCommand and Leaf components. Although the generative pattern returns a higher complexity value, the additional components in the static example have some degree of complexity, which has to be taken into account.

Whilst there are some negative aspects relating to the InvokeButton in the generative example, the generative example exhibits better collective values in its metric than the static pattern. This takes into account the reduction in the number of lines of code, the reduction in the number of classes and

the size of the executable file, which are in favour of the generative example.

Class	CBO		CC		LCOM		LOC		RFC		WMPC	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Client	20	26	17	17			130	130	2	2	2	2
	+		/				/		/		/	
Button	7	9	17	17			49	45	14	14	17	17
Handler	+		/				–		/		/	
Command	2	1	1	1			7	5	4	2	4	2
Holder	–		/				–		–		–	
Invoke	2	1	1	1	66	0	26	17	4	2	5	3
Button	–		/		–		–		–		–	
Concrete	1	2	1	1			14	14	3	3	2	2
Command4	+		/				/		/		/	

Class	CBO		CC		LCOM		LOC		RFC		WMPC	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
CombinedIF	1	*	1	*		*	21	*	14	*	14	*
Component	*	1	*	1	*		*	12	*	5	*	5
Command	*	0	*	1	*		*	4	*	1	*	1
Builder	*	1	*	1	*		*	10	*	6	*	6
Leaf	*	4	*	2	*	0	*	29	*	8	*	5
Concrete	*	1	*	1	*		*	9	*	2	*	2
Commands	*		*		*		*		*		*	

Table 7.4: Individual statistics for the Generative and Static versions of Command, Composite and Builder

Like in the previous example, the individual class statistics for the like-for-like components in the examples are identical throughout all metric categories, therefore they are not included in Table 7.4. In this example this equates to the ConcreteBuilder components, the CoffeeProduct, the Director and the Composite class. Again, like the previous examples the reason for this is modularity, in that each corresponding component provides identical functionality.

7.3.4 A Case Study using Composite, Command, Decorator and Builder

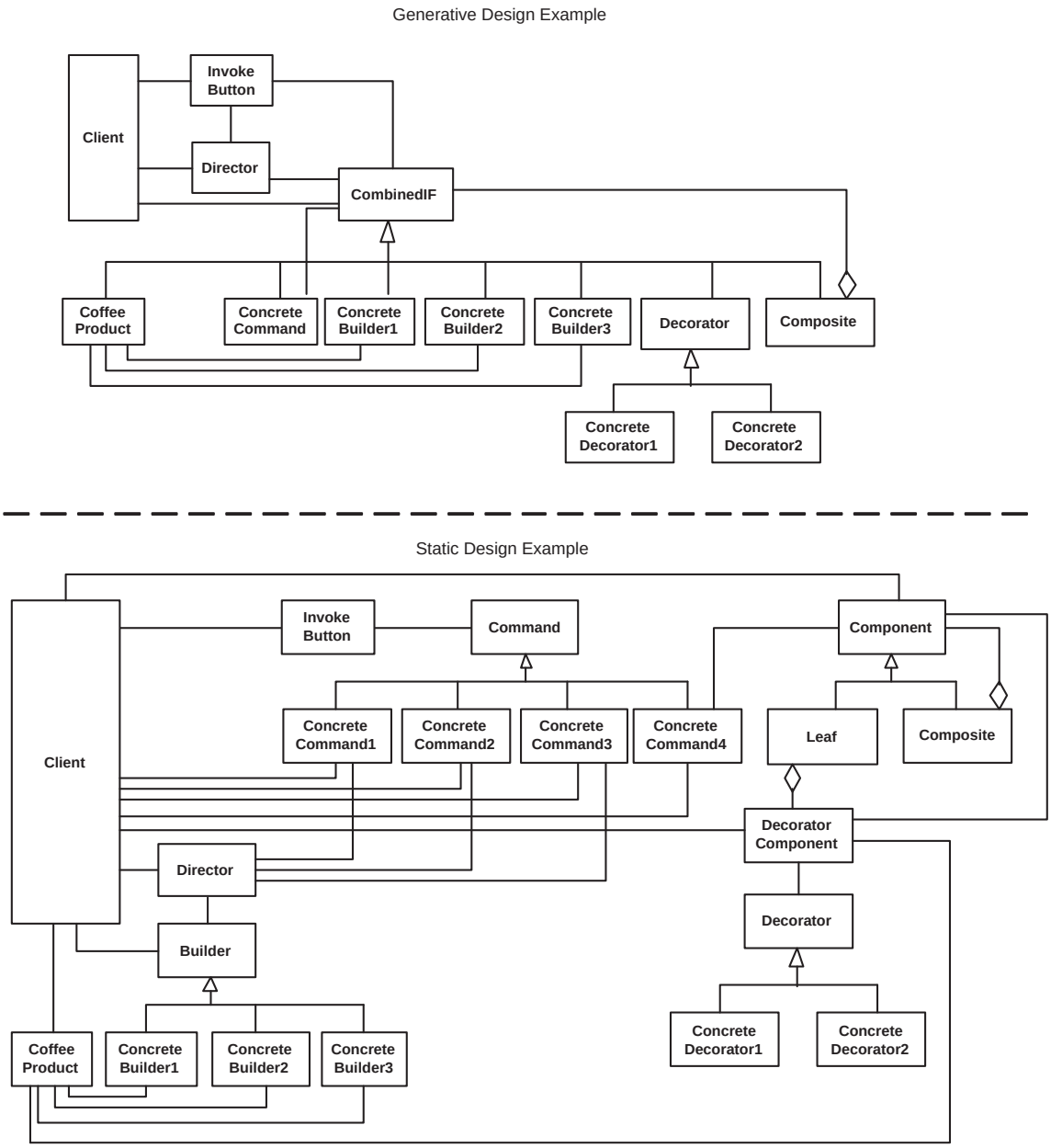

Figure 7.3: Generative vs. Static – Command + Composite + Builder + Decorator

Figure 7.3 above provides a class diagram for the comparative examples of the composite, command, decorator and builder patterns used in a generative and static pattern environment. This example of the four patterns is very similar to the composite, command and builder examples seen in Figure 7.2. However, with the inclusion of the decorator pattern, there are some minor changes to how the examples operate.

Like the previous example on the generative side of the diagram, both the command and builder

patterns are being combined with the composite pattern, but the builder pattern is still only using the InvokeButton from the command pattern. Again, similar to the previous example, ConcreteCommand4 from the command pattern shares an interface with the composite pattern, but is not strictly a leaf component of the composite as it is not practical to add ConcreteCommand4 to the composite object.

The builder and command patterns are collaborating as they do in the previous example, except in this example it is not the CoffeeProduct object that is being stored in the composite object. In this example it is a decorator object that is being stored in the composite object. With this difference, the decorator component is taking as a parameter a CoffeeProduct object. This in effect means that the CoffeeProduct is a leaf in the decorator pattern, whilst the Decorator is a leaf in the composite pattern. However, the CoffeeProduct is still a leaf in the composite pattern as it could be added to the composite object without having decoration applied to it.

The situation described in the previous paragraph is difficult to achieve in the static example as the leaf component has been defined to take decorator components only in its collection object.

In order for the patterns to work together in the static environment, the user has to decide whether the sale of a coffee is to be a drink indoors or a drink out option. Depending on the decision, one of the decorator objects is created. Following this, a product object is built and created when a ConcreteCommand is issued through an InvokeButton command. The ConcreteCommand instructs the ConcreteBuilder to build the product; the product is then added as a parameter to the previously created decorator object. The decorator object is added to a collection object in the Leaf component of the composite pattern. As such, multiple decorator objects can be added to one or more Leaf components and one or more Leaf components can be added to a Composite component. Composite components can be added to other Composite components as is intended with a composite pattern.

Table 7.5 shows the overall results of the metrics that were produced from the generative and static examples of the composite, command, decorator and builder patterns described above.

Metric	Generative Patterns	Static Patterns	Difference (%)
CBO	21	28	+
CC	23	23	/
LCOM	88	88	/
LOC	487	543	+10.1%
RFC	14	11	–
WMPC	14	11	–
NOC	13	20	+
EXE SIZE	20.5	23.8	+13.9%

Table 7.5: Code statistics for the Generative and Static versions of Command, Composite, Decorator and Builder

The statistics in Table 7.5 show that the static pattern will require more testing in respect of the CBO metric than that of the generative pattern. With four patterns in these examples, the client in the static example is communicating with four interface components whereas the client in the generative example only communicates with one. As such the CBO metric in the static example is considerably higher than that in the generative pattern example. This represents a higher degree of coupling in the static example and a higher degree of testing and maintenance of the client.

Like the previous example, the RFC metric of the generative pattern is higher than the static pattern. This higher RFC value comes from the CombinedIF component of the generative pattern, which is listed in Table 7.6. This is a result of the CombinedIF component defining methods for four different types of subcomponent. As a result, there will be a need for more testing of the CombinedIF component. Again, like in previous examples, the WMPC metric shows that the complexity of the generative example is higher than the static example. The higher value for this metric again comes from the CombinedIF component which is acting as a communication point between the client and the remaining class components in the example.

The significant negative aspect in this generative example comes from the InvokeButton which, as is shown in Table 7.6, is quite high in terms of RFC and WMPC metrics. This is a result of it providing two sets of methods to set the commands for the ConcreteCommand4 component and the ConcreteBuilder components, as described in the previous example.

Whilst the four separate interfaces (Builder, Command, Component and DecoratorComponent) in the static example still have collective values lower than the CombinedIF interface of the generative example, the difference is marginal. When taking all the additional components of the static example into

consideration, the static example will require some additional testing and maintenance. Although the generative pattern returns a higher complexity value, the additional components in the static example have some degree of complexity, which has to be taken into account.

Whilst there are some negative aspects relating to the InvokeButton in the generative example, the generative example exhibits better collective values in its metric than the static pattern. This takes into account the reduction in the number of lines of code and the size of the executable file, which are in favour of the generative example.

Because the general metrics in this example are so similar to the previous example, it appears that the addition of the decorator pattern has not added to the overall value of the general metrics, other than the client. However, the addition of extra components has increased the collective values of the metrics, which increases additional work that may need to be applied in terms of the attributes listed in Section 7.2.

Class	**CBO**		**CC**		**LCOM**		**LOC**		**RFC**		**WMPC**	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Client	21	28	23	23			151	151	2	2	2	2
	+		/				/		/		/	
Button	9	12	23	23			59	55	16	15	23	23
Handler	+		/				–		–		/	
Command	2	1	1	1			7	5	4	2	4	2
Holder	–		/				–		–		–	
Invoke	2	1	1	1	66	0	26	17	4	2	5	3
Button	–		/		–		–		–		–	
Concrete	1	2	1	1			14	14	3	3	2	2
Command4	+		/				/		/		/	
Concrete	0	1	1	1			14	14	2	2	2	2
Decorators	–		/				/		/		/	

Continued on next page.

Class	CBO		CC		LCOM		LOC		RFC		WMPC	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
CombinedIF	1	*	1	*		*	21	*	14	*	14	*
Component	*	1	*	1	*		*	12	*	5	*	5
Command	*	0	*	1	*		*	4	*	1	*	1
Builder	*	1	*	1	*		*	10	*	6	*	6
Decorator	*	1	*	1	*		*	5	*	1	*	1
Component	*		*		*		*		*		*	
Leaf	*	4	*	2	*	0	*	29	*	8	*	5
Concrete	*	1	*	1	*		*	9	*	2	*	2
Commands	*		*		*		*		*		*	

Table 7.6: Individual statistics for the Generative and Static versions of Command, Composite, Decorator and Builder

Like in previous examples, the individual class statistics for the like-for-like components in the examples are identical throughout all metric categories, therefore they are not included in Table 7.6 above. In this example this equates to the ConcreteBuilder components, the CoffeeProduct, the Director and the Composite class. Again, like the previous examples the reason for this is modularity, in that each corresponding component provides identical functionality.

7.3.5 An Alternative Case Study using Composite, Command, Decorator and Builder

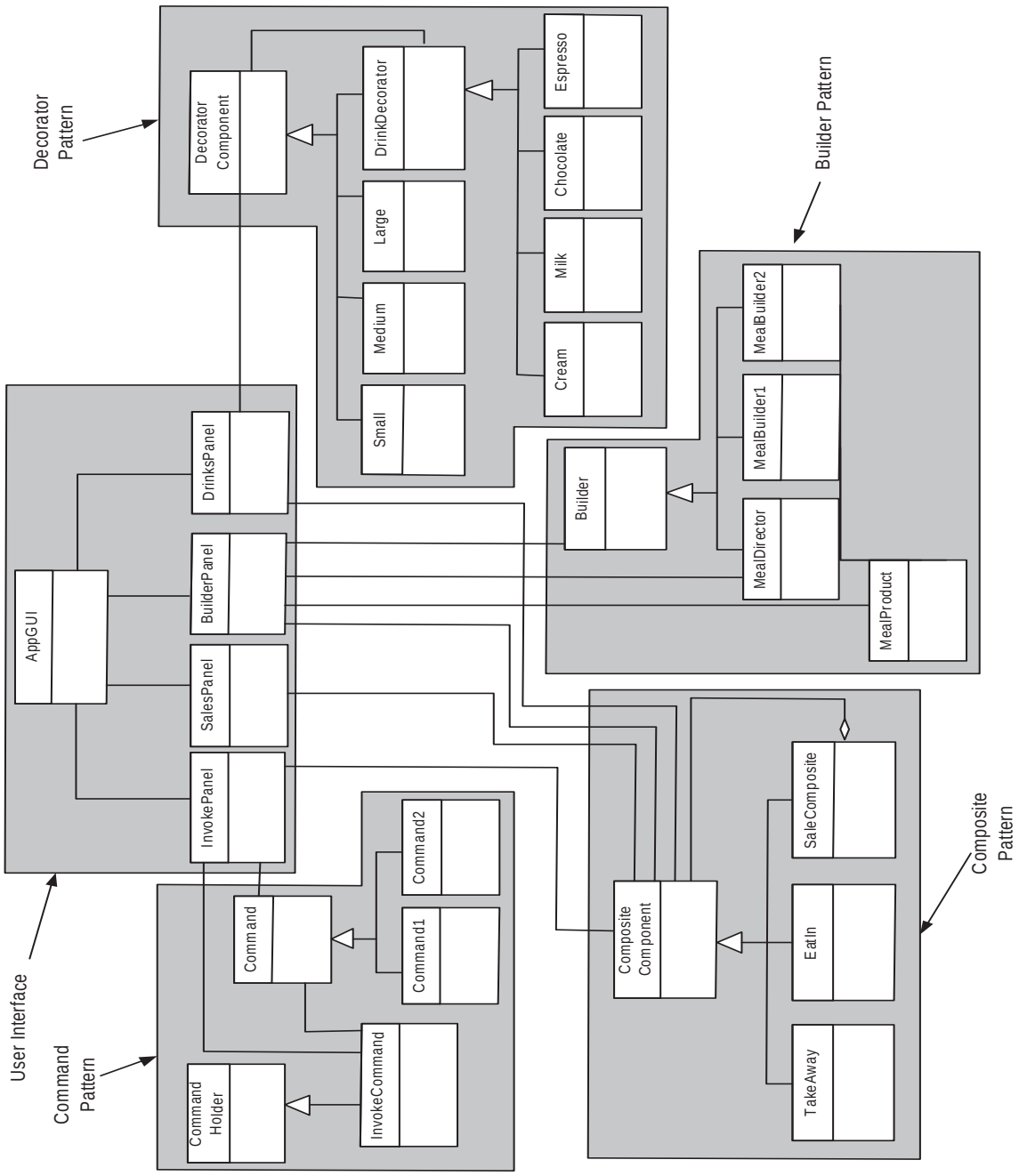

Figure 7.4: Example Application using Static Design Patterns

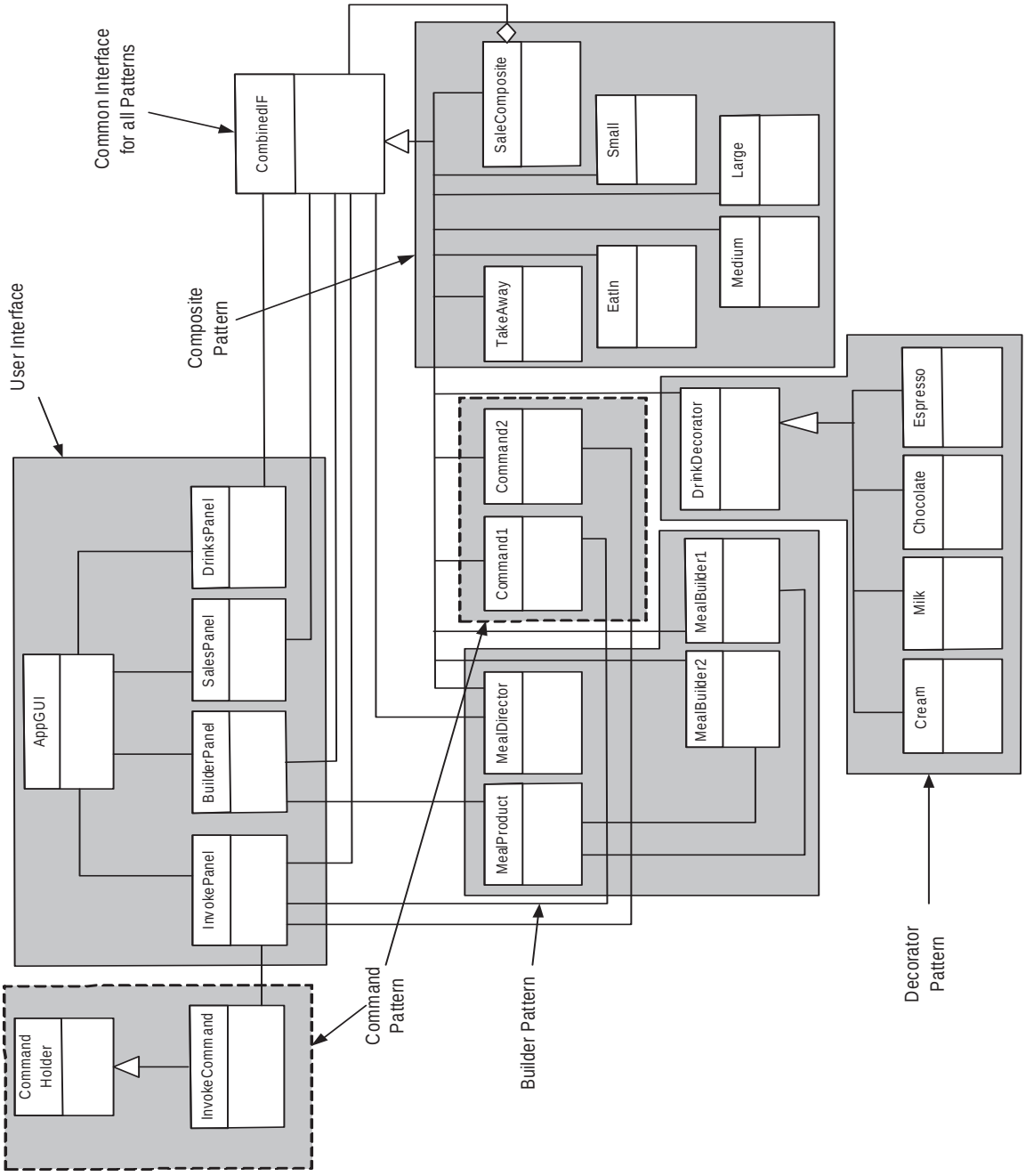

Figure 7.5: Example Application using Dynamic Design Patterns

The examples illustrated by the designs in Figure 7.4 and Figure 7.5 uses the same four patterns as in the previous examples. However, where the design patterns in the previous examples supported a single functional aspect of an application, the patterns in these examples support several independent functions within the context of a touch screen cash register.

Other than the composite pattern that is used to store data relating to different types of sales, the decorator, command and builder pattern each represents a different sales type:

- The decorator pattern creates a small, medium or large drink of coffee that can be decorated with a range of different additives.

  drink = new Espresso(new SteamedMilk(new ChocolateSprinkle(new WhippedCream(new Small()))));

- The command pattern represents the sale of a single item.

  InvokeCommand commOne = new InvokeCommand();

  Command eggs = new EggCommand();

  commOne.setCommand(eggs);

- The builder pattern represents the sale of several different types of meal that can be built up from a range of different items.

  BuildEggs(){mealProduct.setBuildEggs("One Egg");}

  BuildSausage(){mealProduct.setBuildSausage("One Sausage");}

  BuildBeans(){mealProduct.setBuildBeans("Small Beans");}

  BuildChips(){mealProduct.setBuildChips("Small Chips");}

  BuildToast(){mealProduct.setBuildToast("One Toast");}

  BuildPrice(){mealProduct.setBuildPrice(3.50);}

Although the three patterns that are mentioned in the list above work together for the application as a whole, they do not work together to support a single functional aspect of the application. This approach to using the patterns was deliberate in an attempt to demonstrate how patterns can still collaborate whilst supporting different functionality, which is different to how the previous examples have been applied. There is an exception to this in that two different implementations of a ConcreteCommand have been used. One implementation supports its own functionality, whilst the second implementation creates a decorator object.

The generative example shown in Figure 7.5 provides a single interface component to all four of the patterns as it did in the previous example. However this interface has four different clients, one for each pattern. In this respect, the TillComponentIF component is acting like a facade[45] pattern, providing access to sub-components of the application.

In the static pattern example, each object that is created from each different pattern has two methods that return a cost and a description. A get method is used to extract the cost and description from each object and is passed into an instance of a leaf component of the composite pattern, which can then be added to the composite object.

Table 7.7 below shows the metric values that were returned for each of the generative and static examples of the application.

Metric	Generative Patterns	Static Patterns	Difference (%)
CBO	30	30	/
CC	4	8	+
LCOM	100	100	/
LOC	1322	1312	-0.76%
RFC	47	47	/
WMPC	31	25	–
NOC	25	28	+
EXE SIZE	56.1	58.4	+3.9%

Table 7.7: Code statistics for the Generative and Static versions of a touch screen cash register

The statistics in Table 7.7 show that the testing requirements in respect of the CBO metric are exactly the same. In previous examples the value of the CBO metric in the general statistics table has come from the client, which in all cases has been higher in the static example. However, the examples in Figures 7.4 and 7.5 have a slight difference in the client component, particularly with the static example, in that the client for each of the different patterns is a different JPanel. Where the generative example is concerned, each JPanel client matches the functionality that is provided by the patterns that are used in the static example. Therefore, both examples have been developed with the same functionality in relation to the patterns.

The CBO value in each of the examples comes from the CashTillGUI component where the graphical user interface is built. Because both CashTillGUI components are built in exactly the same way, leaving the JPanel extended components to maintain functionality, the CBO values are the same. The value of the

RFC metric in the general statistics also comes from the CashTillGUI component and is of equal value for the same reasons as the CBO metric mentioned above. Like in previous examples, the WMPC metric shows that the complexity of the generative example is higher than the static example. The higher value for this metric again comes from the CombinedIF component which is acting as a communication point between the client and the remaining class components in the example. This metric, for the generative pattern, is above the upper value recommended by the modelling tool where the metrics were taken. This suggests that it may be better to separate some of the functionality into separate interface components.

In the generative example, the patterns do not share any components other than the CombinedIF interface because each of the patterns in the generative example is supporting separate functionality. Therefore, the subcomponents in the static example match the subcomponents in the generative example. However, in the generative example there are three extra ConcreteCommand components, which execute the creation of the decorator pattern objects. So, whilst there is a reduction in the number of interface components in the generative example there is an increase in subcomponents. For this reason, there is virtually the same number of lines of code in each of the examples.

The four separate interfaces (Builder, Command, CompositeComponent and DecoratorComponent) in the static example have a collective value of thirty for both the WMPC and RFC metric. Although this value is lower than in the generative example, the difference is marginal. As such, the coupling and complexity aspects of the two examples are very similar. Additionally, the testing and maintenance aspects are also very similar. There are some minor differences in some of the components, which can be seen in Table 7.8. These differences are brought about by how the patterns handle the functionality. In the generative pattern functionality is accessed or provided through the CombinedIF component. In the static example, functionality is handled through one or more different interface components.

Class	**CBO**		**CC**		**LCOM**		**LOC**		**RFC**		**WMPC**	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Builder	13	12	2	2			130	123	12	11	2	2
Panel	–		/				–		–		/	
Button	7	6	2	2			20	15	12	9	2	2
Handler	–		/				–		–		/	
Drinks	11	10	4	8	100	100	132	178	15	12	3	3
Panel	–		+		/		+		–		/	
Button	8	15	4	8			23	50	7	20	4	8
Handler	+		+				+		+		+	
Invoke	10	11	2	2	100	100	115	115	14	14	3	3
Panel	+		/		/		/		/		/	
Button	7	8	2	2			12	12	8	8	2	2
Handler	+		/				/		/		/	
Bacon	1	2	1	1			18	18	5	5	2	2
Command	+		/		/		/		/		/	
Egg	1	2	1	1			18	18	5	5	2	2
Command	+		/		/		/		/		/	

Continued on next page.

Class	CBO		CC		LCOM		LOC		RFC		WMPC	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
CombinedIF	2	*	1	*		*	43	*	31	*	31	*
Concrete	3	*	1	*		*	15	*	8	*	2	*
Commands	*		*		*		*		*		*	
Composite	*	1	*	1	*		*	22	*	13	*	13
Component	*		*		*		*		*		*	
Command	*	1	*	1	*		*	4	*	1	*	1
Builder	*	1	*	1	*		*	28	*	13	*	13
Decorator	*	0	*	1	*		*	5	*	2	*	2
Component	*		*		*		*		*		*	

Table 7.8: Individual statistics for the Generative and Static versions of Command, Composite, Decorator and Builder

Like in previous examples, the individual class statistics for many of the like-for-like components in the examples are identical throughout all metric categories, therefore they are not included in Table 7.8 above. In this example this equates to the ConcreteBuilder components, MealProduct and MealDirector, the Composite and Leaf components, the Decorator and ConcreteDecorator components, the SalesPanel, the CommandHolder and the CashTillGUI. Again, like the previous examples the reason for this is modularity, in that each corresponding component provides identical functionality.

7.4 Conclusion

From the evidence presented in the evaluation a number of observations can be made. On the whole there is a positive set of results, as indicated by the metrics, in favour of the generative patterns when compared to the metrics for the static patterns. Where different patterns have been used there are minor differences in metric values and in some cases the metrics favour the static patterns, particularly where individual components are concerned. However, taking the metrics as a whole the generative patterns have more plus points than negative points. Where multiple patterns have been used, the gap between the positive and negative aspects of the metrics for the collaborating patterns comes down. It would be evident at this point to suggest that using pairs of patterns is more efficient than using multiple patterns. To conclude that as a fact for the majority of patterns would require considerably more testing with pairs and multiples of patterns. However, from the patterns used for this thesis and the way they have been used, they are better used in pairs.

Where the four patterns have been used in a single application there is a difference in results between the generative example that supports a single function and the generative example that supports multiple functions. In the final case-study the four patterns used each supported a different function. What was found was that there was little difference between the static and generative example. Whereas in the previous case-study using four patterns where the patterns supported a single function, it was found that there was a favourable result for the generative example. Again, a factual conclusion cannot be drawn from a single case-study, but for the patterns used in this thesis, generative patterns that support a single function give better results than generative patterns that support multiple functions.

In any development process, there is always a balancing act between obtaining good coupling, cohesion and complexity in a system. Coupling problems could be eliminated by putting all aspects of functionality in one class, but this would not be good programming practice and would adversely affect testing and maintenance of the application. On the opposite side of this there could be multiple classes each with a small piece of functionality. Again, this would not be good practice and would serve to increase the overall complexity of the application. What is evident from the generative patterns is that there are a reduced number of classes without any significant loss of integrity in the aspects of coupling, cohesion and complexity. There are some minor losses in certain components, mainly the combined interface, but overall, the generative patterns provide a better option than static patterns when multiple static patterns are used to support a single function.

The final result to these experiments is to conclude, that for the patterns used:

they are better used in small numbers to support a single function,

there is an overall improvement in the coupling, cohesion and complexity measures.

A positive aspect that has come from the case-studies and the evaluation is that conducting multiple experiments has provided information that can be used in providing knowledge that can be supplied with the generative patterns. For example, it was found that when the command pattern is used with the builder pattern it is easier to use the Director component of the builder pattern as a subcomponent of the combined interface. However, when builder is combined with the composite pattern it is better to use the Product component of the builder pattern as a subcomponent of the combined interface. The reason for this is that the Product is a leaf of the composite component and can be added directly to the composite object.

Therefore, there are certain pros and cons that can be discussed in patterns as a result of conducting multiple experiments. For example, the case-study with the four patterns in Figure 7.3 showed that the InvokeCommand had grown in complexity because it was supporting two different types of

ConcreteCommand. Therefore this can be discussed as a con, with a recommendation to provide one InvokeCommand for each type of ConcreteCommand.

7.5 Summary

Defining generative patterns can only be done when it is known how two particular patterns work together. The combinations described above are only a small proportion of all possibilities for relationships, but are a start. These pairings and multiple combinations are described with the design knowledge of how they combine, therefore it is possible to extrapolate a descriptive relationship from that design. In the Design Patterns catalogue there are many references to combinations of patterns working together, such as the Iterator pattern being used to traverse the Composite pattern, but the design knowledge for this is not provided. Some pattern users might say that the design knowledge contained in a design pattern is the Structure (Class diagram). Others might say that the design pattern is more than a class diagram and some implementation knowledge. If the principle of the pattern is to be maintained then the pattern should convey more knowledge than the design. This however, is one of the problems with current patterns, in that they do not convey sufficient knowledge[59]. The experiments conducted above are an attempt to provide some of the extra knowledge that can be written into a design pattern. Although the experiments conducted have been a comparison of two different development styles, the experiments have shown that combining patterns as a generative component can provide some additional knowledge for pattern users.

Chapter 8

CONCLUSION

The principle contributions of this thesis in defining patterns as a generative development component are:

- *The notation required for a generative design pattern.* This is the contribution of the work in progress. The notation contained in a standard design pattern defines a static structure of a reusable component. To provide patterns with a dynamic structure, quality driven processes have been examined, and elements of these processes have been integrated into the generative design patterns. Independent elements of pattern classifications have also been examined and included in the generative pattern description.

- *The relationships between collaborating design patterns.* There are three primary definitions of relationship between design patterns, Combines, Uses and Used By. Of these three relationships the Combines relationship has been applied in the experiments on implementing a relationship between patterns. From this it was determined that a Uses relationship was more appropriate in one of the experiments.

- *The application of generative design patterns.* The re-engineered pattern notation has been applied to four separate patterns in seven examples of how to use the notation. The generative patterns that have been written using the generative notation are included within this thesis in Chapter Six and Appendices A to D.

Therefore, this chapter concludes the work undertaken in defining a generative pattern. The introduction in Chapter One made the point that the goal of generative programming is the selection of reusable components from a coded library for the automation of application development. However, the point was also made that in the scheme of application development, the precursor to development is design. In order to facilitate the concept of generative programming the precursor to this ought to be generative design.

The aim of this thesis has been met through the refactoring of a static design pattern notation to produce a dynamic pattern notation for the purpose of generative design.

A generative design pattern framework has been constructed through identifying aspects of common notation and has been applied in the definition of a number of generative design patterns. The generative design patterns published in this thesis represent a small number of experiments that have been conducted into finding how these static design patterns will work together in a generative format. By continuing with experimentation on design patterns rules can be established that will allow pattern writers to stipulate criteria for their own patterns to be considered as a generative design pattern.

As a subtext to the framework, a significant contribution to the output of the work conducted in this thesis is the implementation details required to support the dynamic aspects of the pattern. Relational qualities have been applied that supports the concept of a relationship that Combines patterns through collaboration between individual or multiple components of the collaborating patterns.

However, it was found that contrary to published material, in a Uses relationship, certain patterns do not use other patterns in their entirety, they may only use an individual component from another pattern in their collaboration with that pattern. In addition it was found that the Uses relationship may be better suited to certain patterns when attempting to form a collaboration between those patterns. It is not essential when combining patterns that all aspects of the collaborating patterns have to utilized in the collaboration. In this respect the use of an individual component from a pattern will be sufficient to form the collaboration.

Chapter 9

FUTURE WORK

9.1 Introduction

A thesis can be looked upon as being an apprenticeship where skills are first developed. After the apprenticeship has been completed, these skills are expanded upon and refined until the practitioner becomes the expert they aspire to be. This thesis represents the apprenticeship and should be seen as a beginning on which refinements can be made. Although design patterns have continued to be published over the years since Design Patterns[45] was first released in 1995, the patterns in the book have not changed and very little has been done to change them. Henney's[58] view is that refinements did not happen with Design Patterns[45].

Although a generative pattern has been defined there is still much work to be done. The template for the notation can change and there is every possibility that as refinements are made to generative patterns that the template will change. To date, only a small number of experiments have been conducted on a small number of patterns from one catalogue of patterns. There is the potential for years of future work experimenting with combinations of patterns. Given the seven structural patterns in the Gamma catalogue, there are twenty one possible combinations of connectivity.

Calculating the number of potential experiments on patterns uses the formula:

*((n * (n - 1)) / 2)* e.g. 7 x 6 / 2 = 21. Where n = number of patterns.

In the Gamma catalogue as a whole, there are twenty three different patterns. If we were to experiment with combining every pattern with every other pattern there would be 253 experiments. Maintaining the principle of three examples for every possible combination there are 759 possible experiments. The Gamma catalogue mentions the MVC pattern, add that into the figures and there are 828 possible experiments. The list of patterns presented by Tichy alone has the potential for almost five thousand experiments on paired combinations of patterns. As more experiments are conducted so a clearer understanding of which patterns will collaborate and which should not will be developed and an understanding of best practice will be developed. This will possibly lead to a revision of the template for the generative pattern.

The summary of future work in pursuit of update and revision is as follows:

- *To label patterns by their Classification type, Problem type and Association type.*
- *A definitive standard or formula for combining or excluding combinations of patterns.*
  - *Obtained through developing coded examples from related patterns.*
- *Develop a Computer Aided Software Engineering tool for the process of architectural design with generative design patterns.*

The following aspects of future work represent projects separate from the above that could be undertaken in respect of the development of generative design patterns.

- *A generative pattern development method.*
- *A formal mathematical specification of generative design patterns.*

9.2 To label patterns by their Classification, Problem and Association type

9.2.1 Problem type

For many patterns a Problem type classification is already known — Tichy, who is mentioned in this thesis, has classified approximately one hundred patterns. However, Tichy published his work in 1998 and since that time many other patterns have been published but have not been subjected to the same scrutiny as patterns were from that period. Therefore, known software patterns that have not been subject to problem solving scrutiny need to be examined and classified. To do this, publications that relate to appropriate software patterns can be analysed for their content and problem solving intent.

9.2.2 Classification type

In looking at the problem solving intent of patterns the classification type of a pattern can also be determined. The patterns defined by Gamma[45] are already classified and are the inspiration for classifying patterns by this type of labelling. However, most patterns, including those classified by Tichy, are not identified by the Gamma classification. Although some patterns, other than the Gamma patterns, have been given a classification, analysing published material on patterns will, along with determining the

Problem type classification, provide valuable information towards the knowledge contained within a generative pattern and as such, work in this area should continue.

9.2.3 Relational type

Although Zimmer has not instigated an investigation into finding common attributes between patterns he has, through his investigation of the Design Patterns catalogue, detailed a quantity of known relationships. However it is evident, through studying the literature on patterns, that there are some relationships between patterns that are not described by Zimmer. For example the Composite / Command combination in Appendix A has not been identified by Zimmer and is not mentioned in the Design Patterns catalogue.

Of the nine possible combinations of unidirectional cooperation (See Table 4.2), only six are revealed by Zimmer's classification of relationships. The relationships suggest, as revealed by Zimmer, that structural patterns do not use behavioural patterns, whilst behavioural patterns do use structural patterns. However, this does not mean that other uses relationships do not exist and further experiments with patterns will assist in revealing acceptable relationships between patterns that are not currently defined.

The relationships between patterns defined by Zimmer are only a small proportion of the relationships that could exist between all known software design patterns. Zimmer concentrated only on those relationships that were mentioned in the Design Patterns catalogue, neglecting other possible relationships. For example, it is reasonable to expect that there might be some form of relationship between the Facade pattern and the Singleton pattern. The fact that this relationship is not documented is not an indicator that the relationship does not exist. Therefore, further experiments between patterns will reveal as yet undefined relationships that may exist between patterns – not only the patterns from the design patterns catalogue, but patterns from other catalogues and proceedings.

There is also additional work to be considered in this area. The experiment with Builder combines Command revealed that the Builder pattern does not combine with the Command pattern, it only Uses the Invoke object from the Command pattern. Further analysis may reveal that other Uses relationships between patterns may not be as they have been described.

9.3 A definitive standard or formula for combining or excluding combinations of patterns

To overcome the potential volume of work in defining relationships between every pattern, some common ground could be found that defines the relationship between specific pattern types. In finding common ground it can be shown that two previously unrelated patterns can be defined as related, because the two patterns meet on the common ground. The objective therefore is not to set out and define the relationship between every possible combination of patterns, but to find the common attributes of patterns and map them to the attributes of other patterns. However, this task requires experimenting with combinations of patterns to determine what, if any, relationship exists between any given patterns, and what attributes can be abstracted from the relationships that are common to other patterns and their relationships.

It is not enough to say that there is, or should be, a relationship between patterns. The proposed relationship has to be applied and documented, primarily to be accepted as a generative pattern. Defining a universal relationship between patterns requires rules in order to meet the needs of the individual pattern(s) to which the relationship applies. It could be the case that the 'uses' relationship between Pattern X and Pattern Y is different to the 'uses' relationship between Pattern Y and Pattern X. In this situation the rules may define default or illegal combinations. However, defining relationships to other patterns as a universal property of the patterns classification will not account for any unique circumstances in a relationship or exceptions to the rule.

Zimmer reveals in his work that the 'Uses' relationship can have two separate meanings. The standard meaning is Pattern X *uses* Pattern Y in its solution. However, the relationship could be: Pattern X *must use* Pattern Y in its solution, or Pattern X *might use* Pattern Y in its solution. This, as Zimmer relates, indicates the strength of the relationship. Noble[83], reveals twelve different types of relationship between patterns: three primary relationships and nine secondary. Through observation it can be concluded that most of the relationships are refinements of a Uses or Combines classification, whilst others indicate that two patterns are similar.

It could be said that there is only one type of relationship between patterns and that is a Combines relationship. This is stated on the grounds that: if two or more patterns work together to form a solution, then the patterns 'Combine' their resources to solve the problem. For a generative pattern this train of thought could make the description of the pattern easier to define.

It may be the case that Variant Management patterns of the type Behavioural should not work in combination. Until significant testing has been done on these types of patterns a definitive answer is not known and rules that will exclude them in combination cannot be defined.

The intended future work should bring forth rules for defining relationships between specific patterns. By applying the rules to the patterns it should make it a simple task to identify which patterns will work together to build an architecture. However, the rules will not reveal themselves. A continuous search of new and existing literature can be conducted to find examples of patterns working together. Hands on experimentation can be conducted by developing examples of applications constructed from related patterns. This type of work will also provide sample code for inclusion within a generative pattern.

This therefore represents the body of future work. By defining examples, common attributes can be abstracted from the designs of related patterns, and utilized in the definition of generative patterns.

9.4 Develop a case tool for design using generative patterns

There are many CASE tools on the market that can be used to design software. Some are sophisticated Integrated Development Environments that can be purchased at a significant price, whilst others are free or open source with an emphasis on simplicity. However, design tools of whatever standard have one thing in common. They cannot be used to develop architectures by combining design patterns. Several tools do have support for design patterns, such as Together[16] but there is no means of generating an architecture from those patterns. The patterns when applied to a design have to be modified in order to work together.

The ultimate output from the work conducted in this thesis could be a tool that can be used to apply generative pattern design. Although generative design patterns are as yet a proposition under continued investigation, a tool can be instigated that will apply generative patterns to a design. As more generative patterns are defined, so the tool can be expanded to include additional design components. One interesting factor in this process would be to design the tool from generative design patterns.

9.5 Formal Mathematical Specification of generative patterns

One aspect of quality assurance for generative design patterns would be to provide a formal mathematical specification for generative design patterns. Again, like a generative pattern development method this is an offshoot to the current project but is dependent on developing a significant number of generative design patterns.

9.6 Consideration of design patterns for definition and usability

Many relationships between the patterns in the Design Patterns catalogue have not been defined — not because they don't exist, but because the patterns themselves may be lacking in the quality required to define the relationships. Henney[59] gave a tutorial on the patterns from the Design Patterns catalogue in which he:

> *Reflects on them, deconstructs them and re-evaluates them from a practitioner's perspective.*

His discussion was aimed at, in his own words:

> *Why patterns such as Abstract Factory, Builder, Flyweight, Command and others are missing vital ingredients to be proper parts of an architectural vocabulary.*

He discusses:

> *Why Singleton decreases a system's flexibility and testability.*
>
> *Why Iterator is not always the best solution for traversing aggregates.*
>
> *Why State is not the only state pattern.*
>
> *Why some patterns, such as Bridge, are more than one pattern.*

Henney concludes that Design Patterns[45] was a start to the design pattern culture and not the end result; that improvements in design knowledge has lead to a greater understanding of design patterns, and that the Design Patterns catalogue is dated[58].

In Appendix A the Composite pattern is combined with the Command pattern. However, according to Gamma[45] and Zimmer[119] the Command pattern can use the Composite pattern in its implementation. In reality what the Command pattern uses, if it does use the Composite in its implementation, is the Composite object and not the Composite pattern. This emphasizes the point made earlier by Henney[59] that the Design Patterns[45] catalogue is not a comprehensive reference guide to design patterns. In fact there is one visible anomaly presented by Gamma in their description of the Command pattern. The sequence diagram shows an interaction between the Client and the Invoke object, which does not appear in the class structure.

To this end, the re-engineering of standard design patterns should not only look at how patterns can be defined as generative but should also look at the pattern itself.

BIBLIOGRAPHY

[1] The Object Agency. http://www.toa.com, 2004 [Accessed August 2007].

[2] C. Alexander. *The Timeless Way of Building.* OUP, 1979.

[3] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and A. Shlomo. *A Pattern Language.* OUP, 1977.

[4] C. Alexander, S. Sanford, S. Ishikawa, C. Coffin, and A. Shlomo. *Houses Generated by Patterns.* Berkley (Calif.), Center for Environmental Structure, 1970. ISBN x3360704.

[5] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns.* Prentice Hall, 2001.

[6] B. Appleton. Patterns and software: Essential concepts and terminology. http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html, [Last Accessed June 2008].

[7] K. Arnout. *From Patterns to Components.* PhD thesis, Swiss Federal Institute of Technology Zurich, 2004.

[8] J. Avotins and C. Mingins. Metrics for object-oriented design. In *TOOLS (12/9)*, 1993.

[9] K. Beck. *Extreme Programming Explained.* Addison Wesley, 2000.

[10] K. Beck and C. Andres. *Extreme Programming Explained 2nd Ed.* Addison Wesley, 2004.

[11] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings of ECOOP 94.* Springer Verlag, 1994.

[12] M. Beedle and K. Schwaber. *Agile Software Development with Scrum.* Prentice Hall, 2008.

[13] E. Berard. A comparison of object-oriented development methodologies. http://www.ipipan.gda.pl/marek/objects/TOA/OOMethod/mcr.html, 1995 [Last Accessed June 2008].

[14] G. Booch. *Object Oriented Design With Applications 2nd ed.* Benjamin Cummings, 1994.

[15] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, 1999.

[16] Borland. Together architect. http://www.borland.com/us/products/together/index.html/, 1994 - 2008 [Last Accessed November 2008].

[17] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming, Vol. 11, No. 2*, 1998.

[18] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. *IBM Systems Journal. Vol. 35, No. 2*, 1996.

[19] F. Buschmann and R. Meunier. A system of patterns. In *Proceedings of Pattern Languages of Programming*, 1994.

[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture: A System of Patterns.* Wiley, 1996.

[21] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering.*, June 1994.

[22] M. Cline. The pros and cons of adopting and applying design patterns in the real world. *Communications of the ACM, Vol. 39, No. 10*, 1996.

[23] P. Coad and J. Nichola. *Object-Oriented Programming.* Prentice Hall, 1993.

[24] P. Coad and E. Yourdon. *Object-Oriented Analysis, 2nd Ed.* Prentice Hall, 1991.

[25] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice Hall, 1991.

[26] A. Cockburn. *Agile Software Development.* Addison Wesley, 2002.

[27] J. Coldewey. User interface software. In *Proceedings of the Conference on Pattern Languages of Programming*, 1998.

[28] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.

[29] J. Coplien. A generative development-process pattern language. In *Pattern Languages of Program Design Vol. 1.*, 1995.

[30] J. Coplien. Software design patterns: Common questions and answers. *The Patterns Handbook: Techniques, Strategies, and Applications*, 1998.

[31] J. Coplien and D. Schmidt. *Pattern Languages of Program Design.* Addison Wesley, 1995.

[32] C# Corner. Composite patterns in c#. http://www.c-sharpcorner.com/Language/CompositPattersnInCSRVS.asp, 1999 - 2008 [Last Accessed June 2008].

[33] W. Crawford and J. Kaplan. *J2EE Design Patterns.* O'Reilly, 2003.

[34] W. Cunningham. The checks pattern language of information integrity. In *Pattern Languages of Program Design Vol. 1.*, 1995.

[35] W. Cunningham. Portland pattern repository. http://c2.com/ppr/index.html, [Last Accessed June 2008].

[36] J. Cybulski and T. Lynden. Composing multimedia artefacts for reuse. In *Pattern Languages of Programming*, Allerton Park, Illinois, USA, 1998.

[37] K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison Wesley, 2000.

[38] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press - Prentice Hall, 1978.

[39] B. Eckel. *Thinking in Patterns - Electronic Book.* Bruce Eckle, MindView Inc, 2003.

[40] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *European Conference on Object-Oriented Programming, Vol. 1241 of LNCS, Springer*, 1997.

[41] B. Foote, N. Harrison, and H. Rohnert. *Pattern Languages of Program Design 4.* Addison Wesley, 1999.

[42] M. Fowler. *Analysis Patterns 2nd Ed.* Addison Wesley, 1997.

[43] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison Wesley, 2003.

[44] M. Fowler. The new methodology. http://www.martinfowler.com/articles/newMethodology.html, 2005 [Last Accessed November 2008].

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software.* Addison Wesley, 1995.

[46] DM. Germán and DD. Cowan. Hypermedia design patterns. In *7th. Mini Conference on Decision Support Systems, Groupware, Multimedia and Electronic Commerce*, Brugge, Belgium, 1997.

[47] M. Goodland and C. Slater. *Structured Systems Analysis and Design Method: A Practical Approach.* McGraw Hill, 1995.

[48] M. Grand. *Patterns in Java, Vol. 1.* Wiley, 1998.

[49] M. Grand. *Patterns in Java, Vol. 2.* Wiley, 1999.

[50] M. Grand. *Java Enterprise Design Patterns.* Wiley, 2002.

[51] C. Gross. *Foundations of Object-Oriented Programming Using .NET 2.0 Patterns.* Apress, 2006.

[52] D. Gross and E. Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 1998.

[53] Object Management Group. The object management group. http://www.omg.org, 2004 [Last Accessed June 2008].

[54] Object Management Group. *The Unified Modeling Language V. 2.0.* http://www.uml.org/#UML2.0, 2004 [Last Accessed June 2008].

[55] Object Management Group. *Model Driven Architecture.* http://www.omg.org/mda, 2008 [Last Accessed October 2008].

[56] The Hillside Group. Patterns home page. http://hillside.net/patterns, 2005 [Last Accessed June 2008].

[57] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity.* Prentice-Hall, 1996.

[58] K. Henney. Patterns in java: One or many. http://www.two-sdg.demon.co.uk/curbralan/papers/javaspektrum/OneOrMany.pdf, [Last Accessed June 2008].

[59] K. Henney and F. Buschmann. Beyond the gang of four. In *18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*. From the abstract., 2003.

[60] J. Hong, D. Duyne, and J. Landay. *The Design of Sites: Principles, Processes and Patterns for Crafting a Customer-centered Web Experience*. Addison Wesley, 2002.

[61] IBM. Rational rose. http://www-306.ibm.com/software/awdtools/developer/rose/index.html, [Last Accessed November 2008].

[62] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[63] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.

[64] JM. Jézéquel, M. Train, and C. Mingis. *Design Patterns and Contracts*. Addison Wesley, 1999.

[65] N. Kerth. Caterpiller's fate: A pattern language for the transformation from analysis to design. In *Pattern Languages of Program Design Vol. 1.*, 1995.

[66] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained*. Addison Wesley, 2003.

[67] P. Kruchten. *The Rational Unified Process – An Introduction, 3rd ed.* Addison-Wesley, 2003.

[68] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[69] D. Lea. *Christopher Alexander: An Introduction for Object-Oriented Designers*. http://gee.cs.oswego.edu/dl/ca/ca/ca.html, 1997 [Last Accessed June 2008].

[70] D. Lea. Patterns-discussion faq. http://gee.cs.oswego.edu/dl/pd-FAQ/pd-FAQ.html, 2001 [Last Accessed June 2008].

[71] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall, 1994.

[72] F. Lyardet, G. Rossi, and D. Schwabe. Patterns for dynamic websites. In *Pattern Languages of Programs Conference*, Monticello, Illinois, USA, 1998.

[73] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, and K. Tan. Generative design patterns. In *17th IEEE International Conference on Automated Software Engineering (ASE)*, 2002.

[74] F. Marinescu. *EJB Design Patterns.* Wiley, 2002.

[75] R. Marinescu. An object oriented metrics suite on coupling. Technical report, Universitatea Politehnica Timisoara, Facultatea de Automatica si Calculatoare, Departamentul de Calculatoare si Inginerie Software, 1998.

[76] R. Martin, D. Riehle, and F. Buschmann. *Pattern Languages of Program Design 3.* Addison Wesley, 1997.

[77] G. Masuda, N. Sakamoto, and K. Ushijima. Applying design patterns to decision tree learning system. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering*, 1998.

[78] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, December 1976.

[79] G. Meszaros and J. Doble. A pattern language for pattern writing. In *Pattern Languages of Program Design, vol.***3**, 1996.

[80] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[81] Microsoft. *Enterprise Solution Patterns Using Microsoft .NET.* Microsoft Press, 2003.

[82] Vico Open Modelling. Composite pattern. http://vico.org/pages/PatronsDisseny/Pattern[Last Accessed June 2008].

[83] J. Noble. Classifying relationships between object-oriented design patterns. In *In Australian Software Engineering Conference (ASWEC)*, 1998.

[84] J. Noble. Towards a pattern language for object oriented design. In *Proceedings of the Technology of Object-Oriented Languages and Systems.* IEEE, 1998.

[85] Visual Paradigm. Visual paradigm. http://www.visual-paradigm.com/, 2008 [Last Accessed November 2008].

[86] College of Information Sciences Pennsylvania State University and Technology. Scientific literature digital library. http://citeseer.ist.psu.edu, 2007 [Last Accessed November 2007].

[87] Code Project. Composite pattern. http://www.codeproject.com/cs/design/CompositePattern.asp, 2006 [Last Accessed June 2008].

[88] Code Project. Composite pattern. http://www.codeproject.com/cs/design/csdespat_2.asp, 2006 [Last Accessed June 2008].

[89] Rational. Rational unified process. http://www-01.ibm.com/software/awdtools/rup/, 2004 [Last Accessed October 2008].

[90] R. Reissing. Towards a model for object-oriented design measurement. In *In ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, pages 71–84, 2001.

[91] D. Riehle. Composite design patterns. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1997.

[92] D. Riehle and H. Zullighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 1996.

[93] L. Rosenberg and L. Hyatt. Software quality metrics for object-oriented environments. *Software Assurance Technology Centre*, 1997. published in Crosstalk Journal, April 1997.

[94] G. Rossi, A. Garrido, and D. Schwabe. Design reuse in hypermedia applications development. In *8th ACM Conference on Hypertext and Hypermedia (Hypertext 97)*, Southampton, UK, 1997.

[95] G. Rossi, D. Schwabe, and F Lyardet. Improving web information systems with navigational patterns. http://www8.org/w8-papers/5b-hypertext-media/improving/improving.html, [Last Accessed June 2008].

[96] W. Royce. Managing the development of large software systems. In *Reprinted in 9th International Conference on Software Engineering*. ACM Press, 1987.

[97] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[98] D. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM. Volume 38, Number 10*, 1995.

[99] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* Wiley, 2000.

[100] R. Schultz and E. Berard. *Mapping the Berard Object-Oriented Method into DoD-2167A.* http://www.ipipan.gda.pl/marek/objects/TOA/2167a/2167a-BOOM.html, 1995 [Last Accessed June 2008].

[101] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis: Modelling the World in Data.* Prentice Hall, 1988.

[102] S. Stelting and O. Maassen. *Applied Java Patterns.* Prentice Hall, 2002.

[103] C. Thilmany. *.NET Patterns: Architecture, Design and Process.* Addison Wesley, 2004.

[104] W. Tichy. A catalogue of general-purpose design patterns. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 23)*, IEEE Computer Society, 1998.

[105] J. Tidwell. Common ground: A pattern language for human-computer interface design. http://www.mit.edu/jtidwell/interaction_patterns.html, 1999 [Last Accessed June 2008].

[106] J. Tidwell. *Designing Interfaces.* O'Riley, 2005.

[107] Brighton University. What are the current object-oriented methodologies? http://burks.brighton.ac.uk/burks/pcinfo/progdocs/oofaq/s37.htm, 1996 [Last Accessed August 2007].

[108] Rice University. Composite pattern. http://www.exciton.cs.rice.edu/UWisconsin/session2/, 2006 [Last Accessed June 2008].

[109] J. Vlissides. *Pattern Hatching: Design Patterns Applied.* Addison Wesley, 1998.

[110] J. Vlissides, N. Kerth, and J. Coplien. *Pattern Languages of Program Design 2.* Addison Wesley, 1996.

[111] M. Voelter, J. Noble, and D. Manolescu. *Pattern Languages of Program Design 5.* Addison Wesley, 2006.

[112] M. Völter, A. Shmid, and E. Wolff. *Server Component Patterns.* Wiley, 2002.

[113] K. Walden and JM. Nerson. *Seamless Object-oriented Software Architecture: Analysis and Design of Reliable Systems.* Prentice-Hall, 1994.

[114] M. Welie. Web design patterns. http://www.welie.com/patterns, 2007 [Last Accessed June 2008].

[115] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys. Volume 30, Number 4*, 1998.

[116] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.

[117] E. Yourdon. *Modern Structured Analysis.* Prentice Hall, 1989.

[118] E. Yourdon. *Yourdon Systems Method: Model-Driven Systems Development.* Prentice Hall, 1993.

[119] W. Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*, 1995.

Appendix A

COMPOSITE COMBINES COMMAND

Related Patterns

Command (See Command Pattern)

Add behaviour to an application or system by encapsulating a request in an object.

Classification type **(Behavioural)**

Behavioural patterns apply responsibility to objects.

Problem Solving Type **(Control)**

- *For what aspects of functionality is the Command pattern responsible?*

The Command pattern deals with the control of execution, and the selection of appropriate methods.

- *How the Command pattern uses and/or controls the functionality of other patterns.*

The Command pattern adds functionality to an application or system. The Command pattern can take control of specific aspects of other pattern components by offering an alternative to controlling behaviour.

- *How it will combine with other patterns to enhance functionality*

The Command pattern will usually share an interface. This could be a combination of the two interfaces of the combining patterns or could be an interface that has common methods.

Association Type **(Combines (Command))**

The interface components of both Composite and Command combine to form a single interface. The Composite element of the pattern supplies the collection object for the combined patterns whilst the Command element invokes functionality on the Leaf elements of Composite. The reality is that the Leaf components become ConcreteCommands that are invoked by a client that issues a command.

Composite - Command Relationship

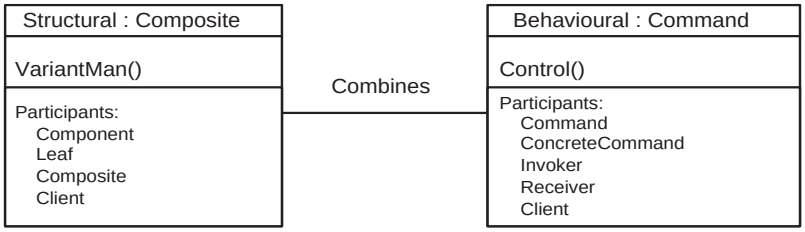

Figure A.1: Relationship between Composite and Command

Examples of Generative Design

Scenario 1

Analysis

Scenario 1 illustrates a simple drawing package where lines, squares and circles can be drawn within a frame. Each drawing item can be individually added to the drawing area by the click of a button, which issues a command to draw the item. Each drawn item can be added to the composite object where it can be used to repaint the drawing area.

Design

Use-Case Diagram

The use-case diagram represents a business process that defines the activities that can be applied to the drawing scenario. In this case, drawing components are created on command and displayed in the drawing area.

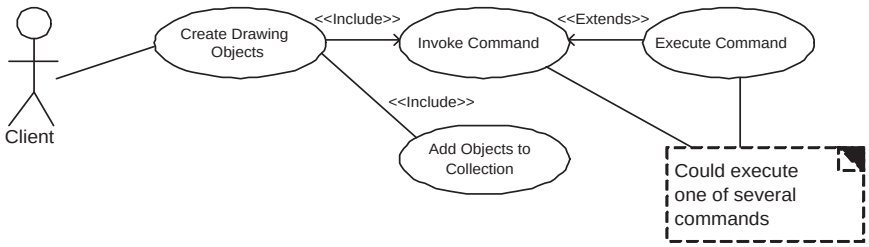

Figure A.2: Use-Case Diagram - Composite combines Command

Class Diagram

The diagram below shows the class components that collaborate to form the structure of the Composite – Command drawing scenario. Three different buttons are created that are used to issue the commands.

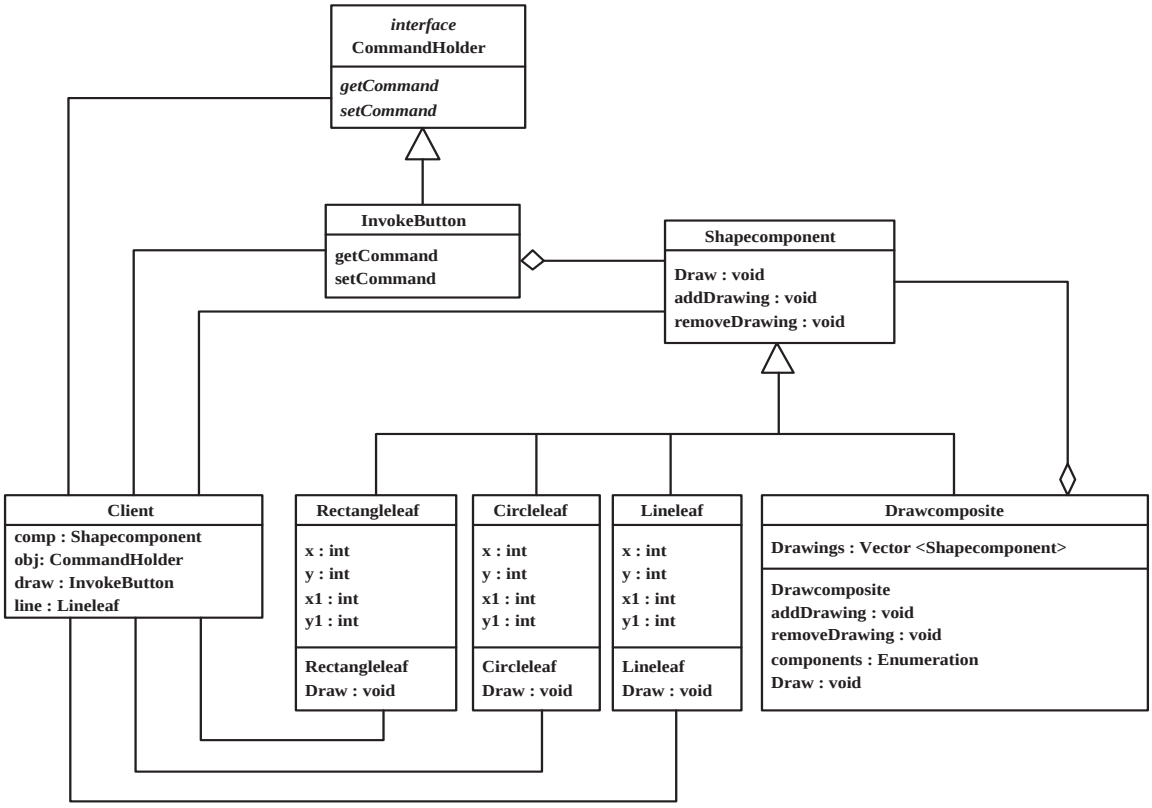

Figure A.3: Class Diagram - Composite combines Command

Sequence Diagram

The interaction diagram shows the sequence of events that occur between the various components that are utilized in this pattern combination. It also shows that the Command decouples the invoking object from the receiving object. When the drawing object has been created it is then added to the Composite object.

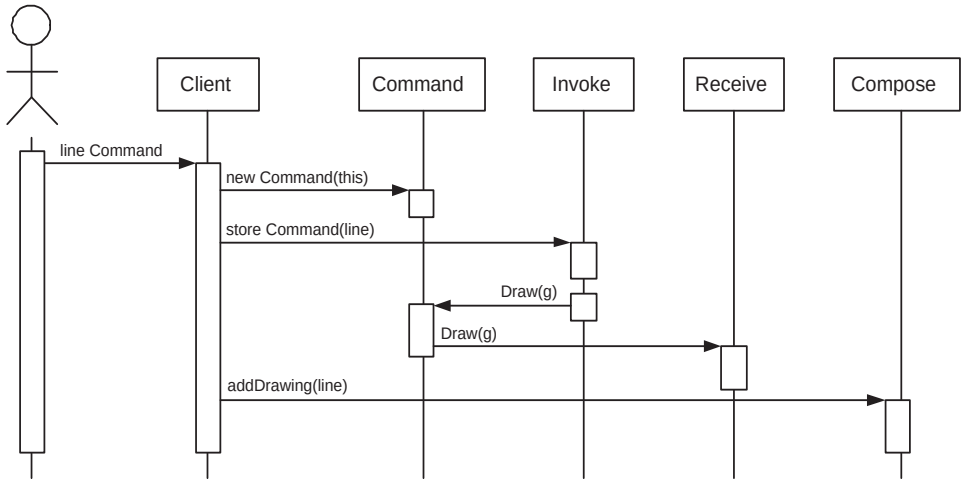

Figure A.4: Sequence Diagram - Composite combines Command

Implementation

This example uses the Composite and Command patterns to demonstrate a simple drawing package. For the purpose of demonstration the components are hard coded into the client but in a live application the components would be created dynamically.

Participants

Client

The Client component is a simple GUI used to create the drawn components. The client implements the Command objects and adds the components created on command to the collection object.

Shapecomponent

The Shapecomponent class specifies an Abstract interface to the components of the Composite and Command. Shapecomponent defines three methods that can be implemented by all sub-classes. The addDrawing(Shapecomponent draw) and removeDrawing(Shapecomponent draw) methods are implemented in the Composite class and the Draw(Graphics g) method is implemented in all sub-classes.

Drawcomposite

Drawcomposite has two functions; one is to add or remove items from the Collection object (the Vector):

- private Vector<Shapecomponent> drawings;

and the other is to call back the items from the collection (print to the frame)

- ((Shapecomponent)components.nextElement()).Draw(g).

In this example items are only added to the collection:

- addDrawing(Shapecomponent draw){drawings. addElement(draw)}.

Lineleaf, Circleleaf, Rectangleleaf

Leaf components represent the drawing objects that are added to the drawing area when issued with a command to do so. Each component defines its own type of drawing object, which is called in the Draw method:

- g.drawLine(x, y, x1, y1).

InvokeCommand

The InvokeCommand stores the ConcreteCommand object which is passed to InvokeCommand as a parameter in a setCommand() method. The InvokeCommand component asks the Command to carry out a request.

CommandHolder

CommandHolder acts as an interface to one or more components that can invoke a command. In this example there is only one Invoke component that activates a button command but there could be others such as menu items.

Receiver

The receiver in this case is the panel on which the drawing objects are drawn

ClientCommand.java (Client)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ClientCommand implements ActionListener
{
  Shapecomponent composite = new Drawcomposite();
```

```
CommandHolder obj;
InvokeButton drawline;
Lineleaf line;

public ClientCommand()
{
  super("Draw commands");
  JPanel jp = new JPanel();
  getContentPane().add(jp);
  jp.setLayout(new BorderLayout());
  JPanel bp = new JPanel();
  jp.add("South", bp);
  PaintPanel cp = new PaintPanel();
  jp.add("Center", cp);
  drawline = new InvokeButton("Draw Line", this);
  line = new Lineleaf(30, 30, 50, 50);
  drawline.setCommand (line);
  bp.add(drawline);
  drawline.addActionListener(this);
  setBounds(200,200,400,200);
  setVisible(true);
}

public void actionPerformed(ActionEvent e)
{
  Graphics g = getGraphics();
  obj = (CommandHolder)e.getSource();

  if(obj == drawline)
  {
    obj.getCommand().Draw(g);
    composite.addDrawing(line);
  }
}

public class PaintPanel extends JPanel
{
  public void paint(Graphics g)
  {
    composite.Draw(g);
  }
}

static public void main(String argv[])
```

```
  {
    new ClientCommand();
  }
}
```

Shapecomponent.java (Component)

```
import java.awt.*;

public abstract class Shapecomponent
{
  public void Draw(Graphics g) {}
  public void addDrawing(Shapecomponent draw) {}
  public void removeDrawing(Shapecomponent draw) {}
}
```

Drawcomposite.java (Composite)

```
import java.awt.*;
import java.util.Vector;
import java.util.Enumeration;

public class Drawcomposite extends Shapecomponent
{
  private Vector<Shapecomponent> drawings;
  public Drawcomposite(){drawings = new Vector<Shapecomponent>();}
  public void addDrawing(Shapecomponent draw){drawings.addElement(draw);}
  public void remove(Shapecomponent draw){drawings.removeElement(draw);}
  public Enumeration components(){return drawings.elements();}

  public void Draw(Graphics g)
  {
    Enumeration components = components();

    while (components.hasMoreElements())
    {
      ((Shapecomponent)components.nextElement()).Draw(g);
    }
  }
}
```

Lineleaf.java (Leaf)

```
import java.awt.*;

public class Lineleaf extends Shapecomponent
{
   private int x, y, x1, y1;

   public Lineleaf(int x, int y, int x1, int y1)
{
     this.x = x;
     this.y = y;
     this.x1 = x1;
     this.y1 = y1;
   }

   public void Draw(Graphics g){g.drawLine(x, y, x1, y1);}
}
```

CommandHolder.java (An interface Component)

```
public interface CommandHolder
{
public void setCommand(Shapecomponent comd);
public Shapecomponent getCommand();
}
```

InvokeButton.java (Invoker)

```
import java.awt.*;
import javax.swing.*;

public class InvokeButton extends JButton implements CommandHolder
{
   private Shapecomponent btnCommand;

   public InvokeButton(String name)
   {
     super(name);
   }

   public void setCommand(Shapecomponent comd) {btnCommand = comd;}

   public Shapecomponent getCommand() {return btnCommand;}
}
```

Appendix B

COMPOSITE COMBINES BUILDER

Related Patterns

Builder (See Builder Pattern)

Simplify complex object creation by defining a class whose purpose is to build instances of another class.

Classification type **(Creational)**

Creational patterns provide flexibility for what gets created, what creates it, how it gets created and when[45].

Problem Solving Type **(Variant Management)**

Variant Management patterns treat different objects with a common purpose in a consistent manner by factoring out their commonality.

- *What objects are being manipulated.*

In this example, the Leaf components of the Composite pattern are being manipulated. Several different objects are being used by one builder to create a pre-defined drawing.

- *Why they are being manipulated.*

In this instance the system can provide access to common graphical structures.

- *What objects will be manipulated through a Combines relationship and how the combination will affect the object.*

The Composite and Builder share an interface and the Builder uses the Leaf components of Composite as a product. An instance of the Composite object is passed through the Builder to store the drawing.

Association Type **(Combines (Builder))**

The interface components of both Composite and Builder combine to form a single interface. The Composite element of the pattern supplies the collection object for the combined patterns whilst the Builder element builds objects from the Leaf elements of Composite.

Composite - Builder Relationship

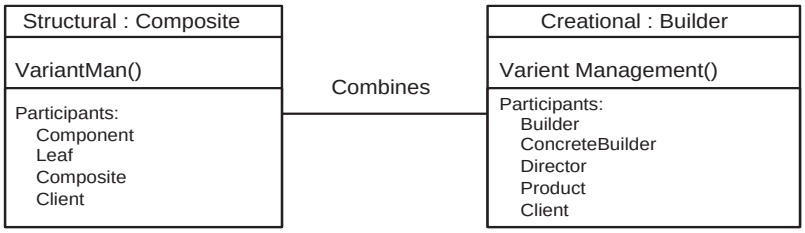

Figure B.1: Relationship between Composite and Builder

Examples of Generative Design

Scenario 1

Analysis

Scenario 1 illustrates a simple drawing package where pre-defined graphics can be drawn within a frame. Different graphical representations can be added to the drawing area when the builder object is called. Each graphic item is be added to the composite object where it can be used to paint the drawing area.

Design

Use-Case Diagram

The use-case diagram represents a business process that defines the activities that can be applied to the drawing scenario. In this case, a graphical is built on request and displayed in the drawing area.

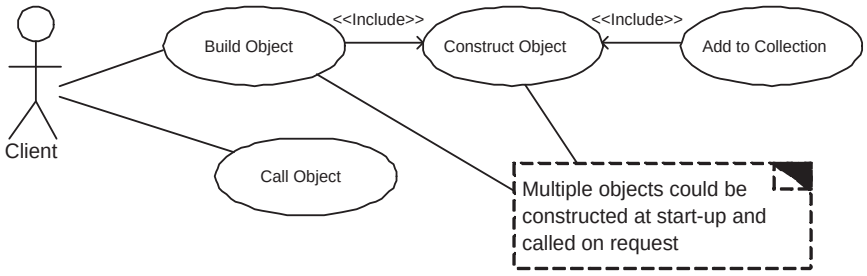

Figure B.2: Use-Case Diagram - Composite combines Builder

Class Diagram

The diagram below shows the class components that collaborate to form the structure of the Composite – Builder drawing scenario.

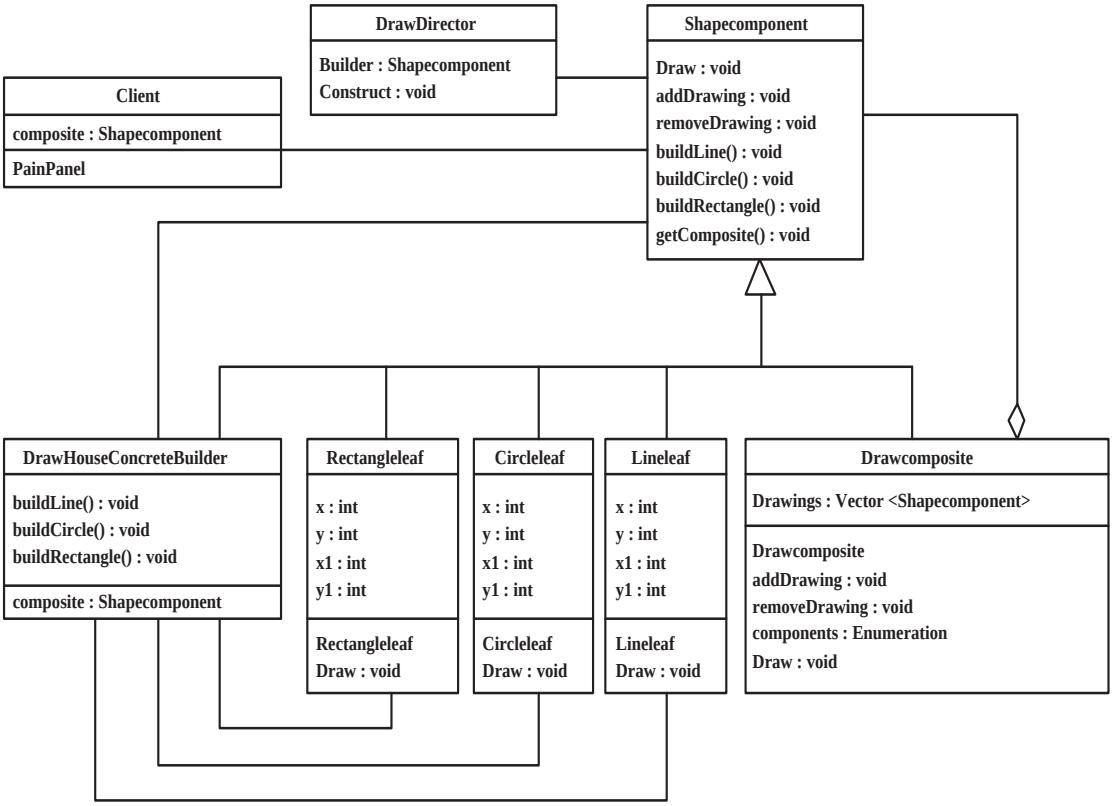

Figure B.3: Class Diagram - Composite combines Builder

Sequence Diagram

The interaction diagram shows the sequence of events that occur between the various components that are utilized in this pattern combination. When the drawing object has been built it is added to the Composite object.

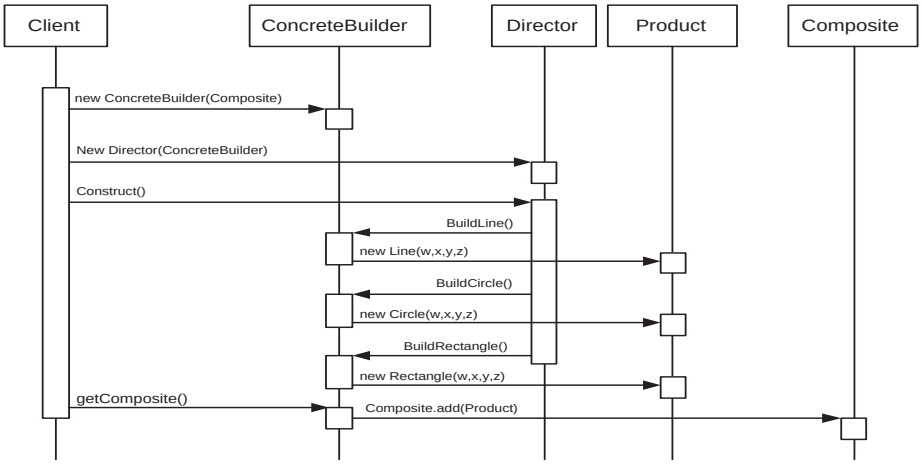

Figure B.4: Sequence Diagram - Composite combines Builder

Implementation

This example uses the Composite and Builder patterns to demonstrate a simple drawing package. The components are hard coded into the ConcreteBuilder and could be in a live application. Alternatively, the components could be created dynamically and stored for future use.

Participants

Client

The Client component is a simple GUI used to display the drawn components. The client implements the Builder which creates the object and adds it to the collection object.

Shapecomponent

The Shapecomponent class specifies an Abstract interface to the components of the Composite and the ConcreteBuilder. Shapecomponent defines seven methods that are a combination of the methods required by both patterns. The addDrawing(Shapecomponent draw) and removeDrawing(Shapecomponent draw) methods are implemented in the Composite class and the Draw(Graphics g) method is implemented in all sub-classes of the Composite part of the combined patterns. The remaining methods are implemented by the Builder part of the combined patterns.

Drawcomposite

Drawcomposite has two functions; one is to add or remove items from the Collection object (the Vector):

- private Vector<Shapecomponent> drawings;

and the other is to call back the items from the collection (print to the frame)

- ((Shapecomponent)components.nextElement()).Draw(g).

In this example items are only added to the collection

- addDrawing(Shapecomponent draw){drawings. addElement(draw)}.

Lineleaf, Circleleaf, Rectangleleaf

Leaf components represent the drawing objects that are added to the drawing area when issued with a command to do so. Each component defines its own type of drawing object, which is called in the Draw method - g.drawLine(x, y, x1, y1).

DrawDirector

The DrawDirector calls the creational methods on its builder instance to have the different parts of the graphical object built.

DrawHouseConcreteBuilder

DrawHouseConcreteBuilder implements all the methods required to create the product - in this case the graphical object.

ClientCommand.java (Client)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ClientBuilder extends JFrame
{
  Shapecomponent composite = new Drawcomposite();

  public ClientBuilder()
  {
    super("Draw Builder");
    JPanel jp = new JPanel();
    getContentPane().add(jp);
    jp.setLayout(new BorderLayout());
    JPanel bp = new JPanel();
```

```
    jp.add("South", bp);
    PaintPanel cp = new PaintPanel();
    jp.add("Center", cp);
    setBounds(200,200,400,400);
    setVisible(true);
  }

  public class PaintPanel extends JPanel
  {
    public void paint(Graphics g)
    {
      Shapecomponent composite = new Drawcomposite();
      Shapecomponent houseBuilder = new DrawHouseConcreteBuilder(composite);
      DrawDirector draw = new DrawDirector(houseBuilder);
      draw.construct();
      composite = houseBuilder.getComposite();
      composite.Draw(g);
    }
  }

  static public void main(String argv[])
  {
    new ClientBuilder();
  }
}
```

Shapecomponent.java (Component)

```
import java.awt.*;

public abstract class Shapecomponent
{
  public void Draw(Graphics g) {}
  public void addDrawing(Shapecomponent draw) {}
  public void removeDrawing(Shapecomponent draw) {}
  public void buildLine() {}
  public void buildCircle() {}
  public void buildRectangle() {}
  public Shapecomponent getComposite() {return null;}
}
```

Drawcomposite.java (Composite)

```
import java.awt.*;
import java.util.Vector;
import java.util.Enumeration;

public class Drawcomposite extends Shapecomponent
{
   private Vector<Shapecomponent> drawings;

   public Drawcomposite(){drawings = new Vector<Shapecomponent>();}

   public void addDrawing(Shapecomponent draw){drawings.addElement(draw);}

   public void remove(Shapecomponent draw){drawings.removeElement(draw);}

   public Enumeration components(){return drawings.elements();}

   public void Draw(Graphics g)
   {
     Enumeration components = components();

     while (components.hasMoreElements())
     {
       ((Shapecomponent)components.nextElement()).Draw(g);
     }
   }
}//The Leaf components of Composite are all very similar. The Rectangleleaf will have g.drawRect(x, y, x1, y1);
```

Lineleaf.java (Leaf)

```
import java.awt.*;

public class Lineleaf extends Shapecomponent
{
   private int x, y, x1, y1;

   public Lineleaf(int x, int y, int x1, int y1)
  {
     this.x = x;
     this.y = y;
     this.x1 = x1;
     this.y1 = y1;
   }

   public void Draw(Graphics g){g.drawLine(x, y, x1, y1);}
}
```

DrawDirector.java (Director)

```
class DrawDirector
{
   private Shapecomponent builder;

   public DrawDirector( Shapecomponent builder ) {this.builder = builder;}

   public void construct()
   {
     builder.buildLine();
     builder.buildCircle();
     builder.buildRectangle();
   }
}
```

DrawHouseConcreteBuilder.java (ConcreteBuilder)

```
import java.awt.*;

public class DrawHouseConcreteBuilder extends Shapecomponent
{
   Shapecomponent composite;

   public DrawHouseConcreteBuilder(Shapecomponent composite) {this.composite = composite;}

   public void buildLine()
   {
     Lineleaf roof1 = new Lineleaf(100, 100, 175, 50);
     Lineleaf roof2 = new Lineleaf(175, 50, 250, 100);
     composite.addDrawing(roof1);
     composite.addDrawing(roof2);
   }

   public void buildCircle()

   public void buildRectangle()
   {
     Rectangleleaf walls = new Rectangleleaf(100, 100, 150, 150);
     composite.addDrawing(walls);
   }

   public Shapecomponent getComposite() {return composite;}
}
```

Appendix C

BUILDER COMBINES COMMAND COMBINES COMPOSITE

The Composite in this example is supplemental to the combining patterns Builder and Command.

Related Patterns

Command (See Command Pattern)

Composite (See Composite Pattern)

Add behaviour to an application or system by encapsulating a request in an object.

Classification type **(Behavioural)**

Behavioural patterns apply responsibility to objects.

Problem Solving Type **(Control)**

- *For what aspects of functionality is the Command pattern responsible?*

The Command pattern deals with the control of execution, and the selection of appropriate methods.

- *How the Command pattern uses and/or controls the functionality of other patterns.*

Adds functionality to an application or system. The Command pattern can take control of specific aspects of other pattern components by offering an alternative to controlling behaviour.

- *How it will combine with other patterns to enhance functionality*

Will usually share an interface. This could be a combination of the two interfaces of the combining patterns or could be an interface that has common methods.

Association Type **(Combines (Command) Combines (Composite))**

The interface components of both Builder and Command combine to form a single interface. However, in addition the interface from the Composite pattern adds to the combination. The Composite element

of the pattern supplies the collection object for the combined patterns. The Command element invokes functionality on the Director element of Builder, which supplies the method calls on the Concrete-Builders. The Leaf components of Composite work as ConcreteCommands / Products that are invoked by a client that issues a command.

Builder - Command - Composite Relationship

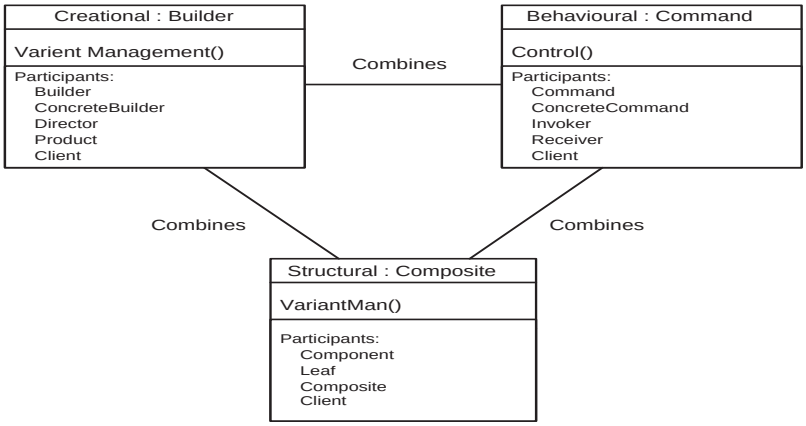

Figure C.1: Relationship between Builder, Command and Composite

Examples of Generative Design

Scenario 1

Analysis

Scenario 1 illustrates a simple drawing package where lines, squares and circles can be drawn within a frame. The drawing artefacts are created by the Builder pattern but are not drawn on the frame until they invoked by the Command pattern. Each drawing item can be individually added to the drawing area by the click of a button, which issues a command to draw the item. Each drawn item can be added to the composite object where it is used to paint the drawing area.

Design

Use-Case Diagram

The use-case diagram represents a business process that defines the activities that can be applied to the drawing scenario. In this case, drawing components are created by the client and only called when an invoke action is activated.

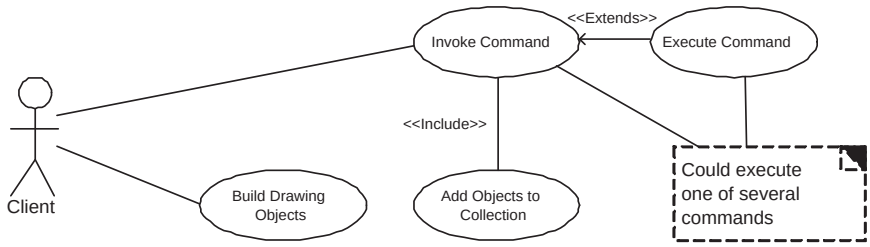

Figure C.2: Use-Case Diagram - Builder combines Command combines Composite

The diagram below shows the class components that collaborate to form the structure of the Builder – Command – Composite drawing scenario. Three different buttons are created that are used to issue the commands.

Class Diagram

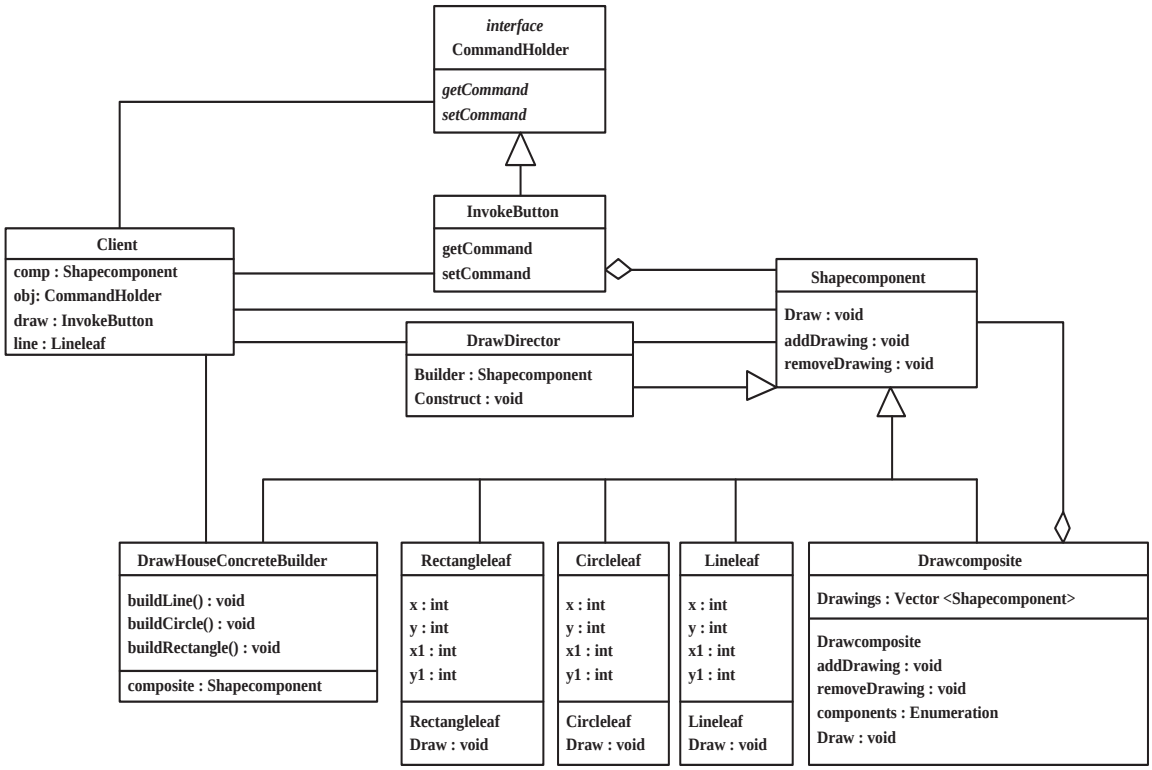

Figure C.3: Class Diagram - Composite combines Command

Sequence Diagram

The interaction diagram shows the sequence of events that occur between the various components that are utilized in this pattern combination. The objects are built and stored in the Composite object until they are drawn on demand. It also shows that the Command decouples the invoking object from the Product. When the a button is activated a command is invoked and the composite object is called.

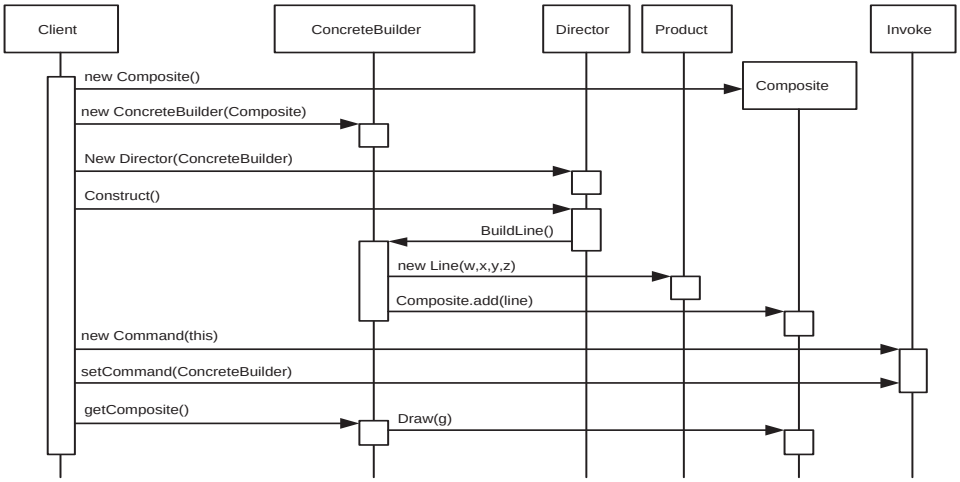

Figure C.4: Sequence Diagram - Composite combines Command

Implementation

This example uses the Builder and Command patterns to demonstrate a simple drawing package. The full Composite pattern is used in the scenario. The composite object holds the drawing objects whilst the Leaf objects of Composite act as Product components. For the purpose of demonstration the components are hard coded into the client but in a live application the components would be created dynamically.

Participants

Client (Receiver)

The Client component is a simple GUI used to build the drawn components. The drawing objects are created automatically and added to the collection object. The client implements the Command objects, which calls the collection object to display the drawing. The receiver in this case is the panel on which the drawing objects are drawn.

Shapecomponent

The Shapecomponent class specifies an Abstract interface to the components of the Command, Builder and Composite. Shapecomponent defines three methods that can be implemented by all sub-classes.

The addDrawing(Shapecomponent draw) and removeDrawing(Shapecomponent draw) methods are implemented in the Composite class and the Draw(Graphics g) method is implemented in all sub-classes.

Drawcomposite

Drawcomposite has two functions; one is to add or remove items from the Collection object (the Vector):

- private Vector<Shapecomponent> drawings;

and the other is to call back the items from the collection (print to the frame)

- ((Shapecomponent)components.nextElement()).Draw(g).

In this example items are only added to the collection

- addDrawing(Shapecomponent draw){drawings. addElement(draw)}.

Lineleaf, Circleleaf, Rectangleleaf (Product)

Leaf components represent the drawing objects that are added to the Composite object during the build operation. Each component defines its own type of drawing object, which is called in the Draw method - g.drawLine(x, y, x1, y1). The Leaf components of Composite act as Product components in the Builder element of the generative pattern.

InvokeButton

The InvokeButton stores the ConcreteCommand object which is passed to InvokeButton as a parameter in a setCommand() method. The InvokeButton component asks the Command to carry out a request.

CommandHolder

CommandHolder acts as an interface to one or more components that can invoke a command. In this example there is only one Invoke component that activates a button command but there could be others such as menu items.

DrawDirector

The DrawDirector calls the creational methods on its builder instance to have the different parts of the graphical object built.

DrawHouseConcreteBuilder

DrawHouseConcreteBuilder implements all the methods required to create the product - in this case the graphical object.

ClientCommand.java (Client)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ClientCommand extends JFrame
{
  Shapecomponent composite = new Drawcomposite();
  CommandHolder obj;
  InvokeButton drawhouse;
  DrawHouseConcreteBuilder houseBuilder = new DrawHouseConcreteBuilder(composite);
  DrawDirector drawHouseDirector = new DrawDirector(houseBuilder);

  public ClientCommand()
  {
    super("Draw Builder - Command");
    JPanel jp = new JPanel();
    getContentPane().add(jp);
    jp.setLayout(new BorderLayout());
    JPanel bp = new JPanel();
    jp.add("South", bp);
    JPanel dp = new JPanel();
    jp.add("Center", dp);
    drawhouse = new InvokeButton("Draw House");
    drawhouse.setCommand (drawHouseDirector);
    drawHouseDirector.construct();
    bp.add(drawhouse);
    ButtonHandler handler = new ButtonHandler();
    drawhouse.addActionListener(handler);
    setBounds(200,200,800,400);
    setVisible(true);
  }

  private class ButtonHandler implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      Graphics g = getGraphics();
      obj = (CommandHolder)e.getSource();

      if(obj == drawhouse){composite.Draw(g);}
    }
  }
```

```
  static public void main(String argv[])
  {
    new ClientCommand();
  }
}
```

Shapecomponent.java (Component)

```
import java.awt.*;

public abstract class Shapecomponent
{
  public void Draw(Graphics g) {}
  public void addDrawing(Shapecomponent draw) {}
  public void removeDrawing(Shapecomponent draw) {}
  public void buildLine() {}
  public void buildCircle() {}
  public void buildRectangle() {}
  public Shapecomponent getComposite() {return null;}
}
```

Drawcomposite.java (Composite)

```
import java.awt.*;
import java.util.Vector;
import java.util.Enumeration;

public class Drawcomposite extends Shapecomponent
{
  private Vector<Shapecomponent> drawings;

  public Drawcomposite(){drawings = new Vector<Shapecomponent>();}

  public void addDrawing(Shapecomponent draw){drawings.addElement(draw);}

  public void remove(Shapecomponent draw){drawings.removeElement(draw);}

  public Enumeration components(){return drawings.elements();}

  public void Draw(Graphics g)
  {
    Enumeration components = components();

    while (components.hasMoreElements())
    {
```

```
      ((Shapecomponent)components.nextElement()).Draw(g);
    }
  }
}
```

The Leaf components of Composite are all very similar. The Rectangleleaf will have g.drawRect(x, y, x1, y1);

Lineleaf.java (Leaf)

```
import java.awt.*;

public class Lineleaf extends Shapecomponent
{
  private int x, y, x1, y1;

  public Lineleaf(int x, int y, int x1, int y1)
{
    this.x = x;
    this.y = y;
    this.x1 = x1;
    this.y1 = y1;
  }

  public void Draw(Graphics g){g.drawLine(x, y, x1, y1);}
}
```

CommandHolder.java (An interface Component)

```
public interface CommandHolder
{
  public void setCommand(Shapecomponent comd);
  public Shapecomponent getCommand();
}
```

InvokeButton.java (Invoker)

```
import java.awt.*;
import javax.swing.*;

public class InvokeButton extends JButton implements CommandHolder
{
  private Shapecomponent btnCommand;

  public InvokeButton(String name)
  {
    super(name);
```

```
  }

  public void setCommand(Shapecomponent comd)
  {
    btnCommand = comd;
  }

  public Shapecomponent getCommand()
  {
    return btnCommand;
  }
}
```

DrawDirector.java (Director)

```
class DrawDirector
{
  private Shapecomponent builder;

  public DrawDirector( Shapecomponent builder )
  {
    this.builder = builder;
  }

  public void construct()
  {
    builder.buildLine();
    builder.buildCircle();
    builder.buildRectangle();
  }
}
```

DrawHouseConcreteBuilder.java (ConcreteBuilder)

```
import java.awt.*;

public class DrawHouseConcreteBuilder extends Shapecomponent
{
  Shapecomponent composite;

  public DrawHouseConcreteBuilder(Shapecomponent composite)
  {
    this.composite = composite;
  }
```

```
    public void buildLine()
    {
      Lineleaf roof1 = new Lineleaf(100, 100, 175, 50);
      Lineleaf roof2 = new Lineleaf(175, 50, 250, 100);
      composite.addDrawing(roof1);
      composite.addDrawing(roof2);
    }

    public void buildCircle()

    public void buildRectangle()
    {
      Rectangleleaf walls = new Rectangleleaf(100, 100, 150, 150);
      composite.addDrawing(walls);
    }

    public Shapecomponent getComposite()
    {
      return composite;
    }
}
```

Appendix D

BUILDER USES COMMAND

Related Patterns

Command (See Command Pattern)

Add behaviour to an application or system by encapsulating a request in an object.

Classification type **(Behavioural)**

Behavioural patterns apply responsibility to objects.

Problem Solving Type **(Control)**

- *For what aspects of functionality is the Command pattern responsible?*

The Command pattern deals with the control of execution, and the selection of appropriate methods.

- *How the Command pattern uses and/or controls the functionality of other patterns.*

Adds functionality to an application or system. The Command pattern can take control of specific aspects of other pattern components by offering an alternative to controlling behaviour.

- *How it will combine with other patterns to enhance functionality.*

Will usually share an interface. This could be a combination of the two interfaces of the combining patterns or could be an interface that has common methods.

Association Type **(Uses (Command))**

The Builder pattern is not using the whole of the Command pattern, it is only using the Invoke component of Command. The build operation of the Builder pattern has a strong influence over the functionality of the combined patterns and as such there is little need for much of the Command components. As such, the Invoke operation works on the Director from the Builder pattern and not a ConcreteCommand from the Command pattern.

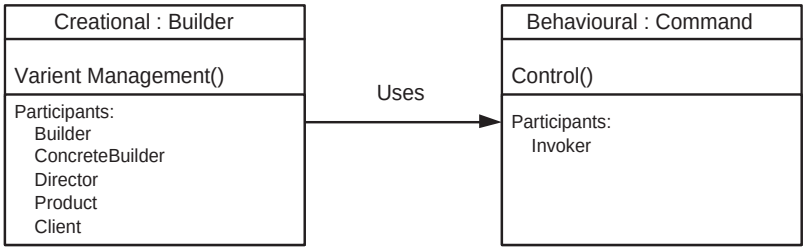

Figure D.1: Relationship between Builder and Command

Builder - Command Relationship

Examples of Generative Design

Scenario 1

Analysis

Scenario 1 illustrates a simple operation where particular styles of coffee can be selected. Each style of coffee is selected at the push of a button, which is intended to simulate the selection process that can be seen on modern cash registers.

Design

Use-Case Diagram

The use-case diagram represents a business process that defines the activities that can be applied to the drinks selection scenario. In this case, drinks are selected by the client at the touch of a button, which is contained in the invoke object.

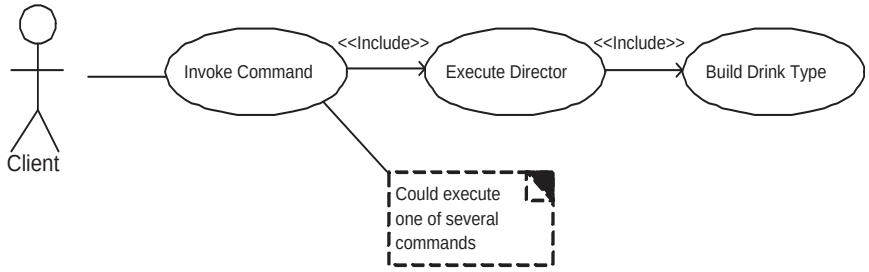

Figure D.2: Use-Case Diagram - Builder uses Command

Class Diagram

The diagram below shows the class components that collaborate to form the structure of the Builder uses Command coffee shop scenario. Three different buttons are created that are used to issue the commands.

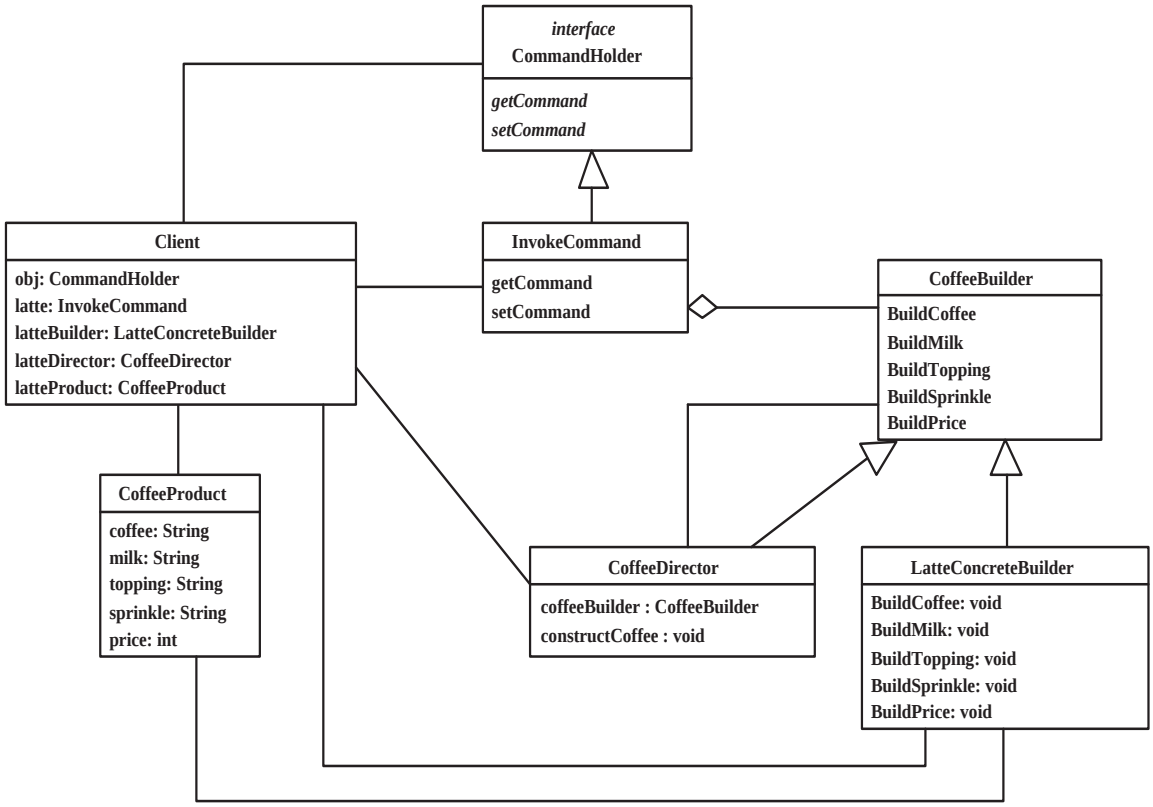

Figure D.3: Class Diagram - Builder uses Command

Sequence Diagram

The interaction diagram shows the sequence of events that occur between the various components that are utilized in this pattern combination. The objects are built and stored in the Composite object until they are drawn on demand. It also shows that the Command decouples the invoking object from the Product. When the a button is activated a command is invoked and the composite object is called.

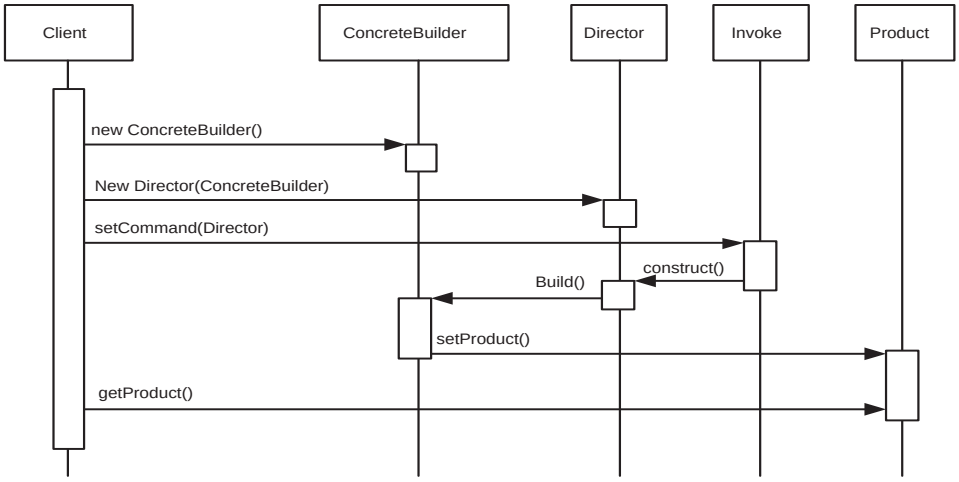

Figure D.4: Sequence Diagram - Builder uses Command

Implementation

This example uses the Builder and Command patterns to demonstrate a simple drink selection package. The Client creates a ConcreteBuilder and Director and uses the buttons of the Invoke command to select a given type of coffee. The Invoke buttons call the construct method of the Director, which issues the action to build the drinks.

Participants

Client

The Client component is a simple GUI used to implement the Invoke commands. Drinks are built and displayed when a button is selected.

CoffeeBuilder

The CoffeeBuilder class specifies an Abstract interface to the components of the ConcreteBuilder. CoffeeBuilder defines five methods that can be implemented by all CoffeeBuilder sub-classes.

CommandHolder

CommandHolder acts as an interface to one or more components that can invoke a command. In this example there is only one Invoke component that activates a button command but there could be others such as menu items.

InvokeButton

The InvokeButton stores the Director object which is passed to InvokeButton as a parameter in a setCommand() method. the InvokeButton component returns a CoffeeBuilder on request.

CoffeeDirector

The CoffeeDirector calls the creational methods on its builder instance to have the different parts of the graphical object built.

LatteConcreteBuilder

LatteConcreteBuilder implements all the methods required to create the product - in this case the a representation of the sale of a drink.

Product

A product is created from the specified methods that are called to create particular drink. The product is called by the client and shown in the display area of the client.

CoffeeClient.java (Client)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CoffeeClient extends JFrame
{
  CoffeeCommand obj;
  InvokeCommand latte;
  LatteConcreteBuilder latteBuilder = new LatteConcreteBuilder();
  CoffeeDirector latteDirector = new CoffeeDirector(latteBuilder);
  CoffeeProduct latteProduct = new CoffeeProduct();
  protected JTextArea textArea = new JTextArea(20,60);
  private final static String newline = "\n";
  JScrollPane scrollPane = new JScrollPane(textArea);

  public CoffeeClient()
  {
    super("Builder Command Coffee House");
```

```
    JPanel jp = new JPanel();
    getContentPane().add(jp);
    jp.setLayout(new BorderLayout());
    JPanel bp = new JPanel();
    jp.add("South", bp);
    JPanel dp = new JPanel();
    jp.add("Center", dp);
    dp.add(scrollPane);

    latte = new InvokeCommand("Latte");
    latte.setCommand (latteDirector);
    latteProduct = latteBuilder.getCoffeeProduct();
    bp.add(latte);

    ButtonHandler handler = new ButtonHandler();
    latte.addActionListener(handler);
    setBounds(200,200,800,400);
    setVisible(true);
  }

  private class ButtonHandler implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      obj = (CoffeeCommand)e.getSource();
      if(obj == latte)
      {
        latteDirector.constructCoffee();
        textArea.append(latteProduct.getCoffee() + newline);
        textArea.append(latteProduct.getMilk() + newline);
        textArea.append(latteProduct.getTopping() + newline);
        textArea.append(latteProduct.getSprinkle() + newline);
        textArea.append("" + latteProduct.getPrice() + newline);
        textArea.append(newline);
        }
      }
    }

  static public void main(String argv[])

  {

    new CoffeeClient();

  }
}
```

CoffeeBuilder.java (Builder)

```
public abstract class CoffeeBuilder
{
   public void buildCoffee()
   public void buildMilk()
   public void buildTopping()
   public void buildSprinkle()
   public void buildPrice()
}
```

CommandHolder.java (An interface Component)

```
public interface CommandHolder
{
   public void setCommand(CoffeeBuilder comd);
   public CoffeeBuilder getCommand();
}
```

InvokeCommand.java (Invoker)

```
import javax.swing.*;
public class InvokeCommand extends JButton implements CommandHolder
{
   private CoffeeBuilder btnCommand;
   public InvokeButton(String name)
   {
     super(name);
   }
   public void setCommand(CoffeeBuilder comd)
   {
     btnCommand = comd;
   }
   public CoffeeBuilder getCommand()
   {
     return btnCommand;
   }
}
```

CoffeeDirector.java (Director)

```
class CoffeeDirector
{
   private CoffeeBuilder builder;

   public CoffeeDirector( CoffeeBuilder builder )
   {
     this.builder = builder;
   }

   public void constructCoffee()
   {
     builder.buildPrice();
     builder.buildCoffee();
     builder.buildMilk();
     builder.buildTopping();
     builder.buildSprinkle();
   }
}
```

LatteConcreteBuilder.java (ConcreteBuilder)

```
public class LatteConcreteBuilder extends CoffeeBuilder
{
   protected CoffeeProduct coffeeProduct = new CoffeeProduct();

   public LatteConcreteBuilder()

   public void buildCoffee()
   {
     coffeeProduct.setCoffee("Latte");
   }

   public void buildMilk()
   {
     coffeeProduct.setMilk("Steamed Milk");
   }

   public void buildTopping()
   {
     coffeeProduct.setTopping("");
   }

   public void buildSprinkle()
```

```
    {
      coffeeProduct.setSprinkle("Vanilla");
    }

    public void buildPrice()
    {
      coffeeProduct.setPrice(200);
    }

    public CoffeeProduct getCoffeeProduct()
    {
      return coffeeProduct;
    }
}

public class CoffeeProduct
{
    private String coffee = "";
    private String milk = "";
    private String topping = "";
    private String sprinkle = "";
    private int price = 0;

    public CoffeeProduct() {}

    public void setCoffee(String coffee) {this.coffee = coffee;}
    public void setMilk(String milk) {this.milk = milk;}
    public void setTopping(String topping) {this.topping = topping;}
    public void setSprinkle(String sprinkle) {this.sprinkle = sprinkle;}
    public void setPrice(int price) {this.price = price;}

    public String getCoffee() {return coffee;}
    public String getMilk() {return milk;}
    public String getTopping() {return topping;}
    public String getSprinkle() {return sprinkle;}
    public int getPrice() {return price;}
}
```

Appendix E

RELATIONSHIP TREES

E.0.1 Structural

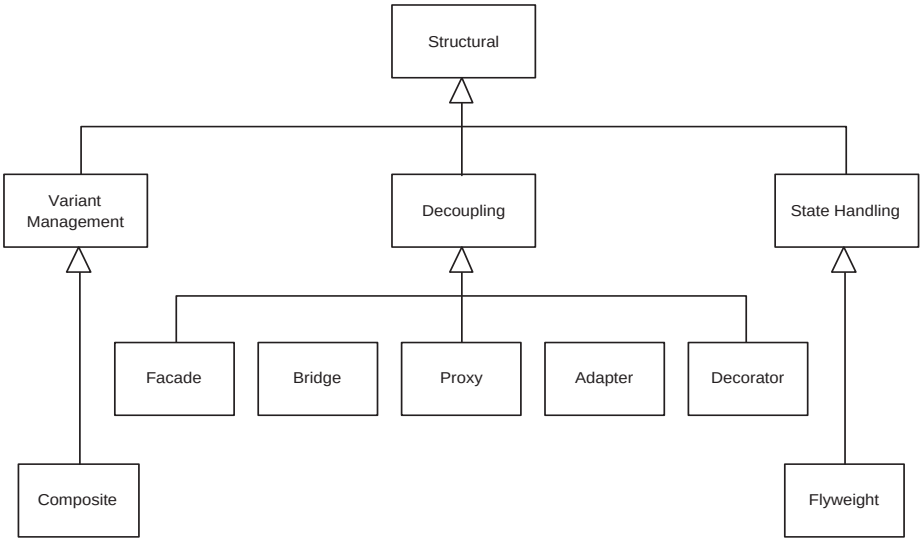

Figure E.1: Structural Hierarchy

E.0.2 Creational

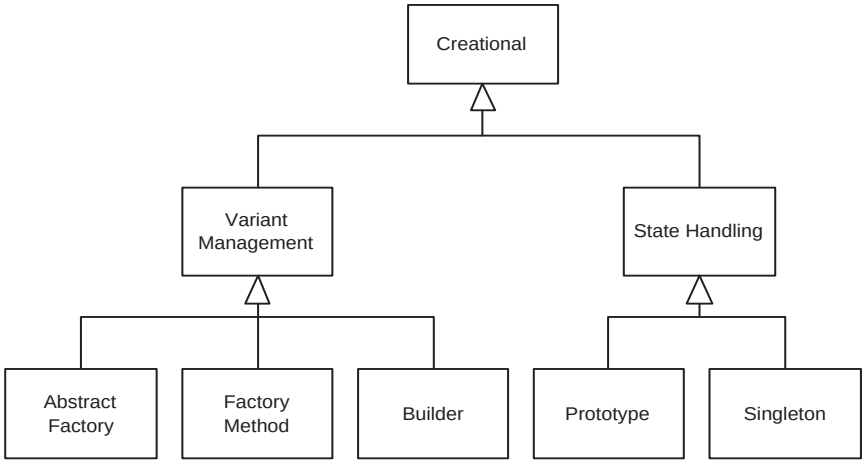

Figure E.2: Creational Hierarchy

E.0.3 Behavioural

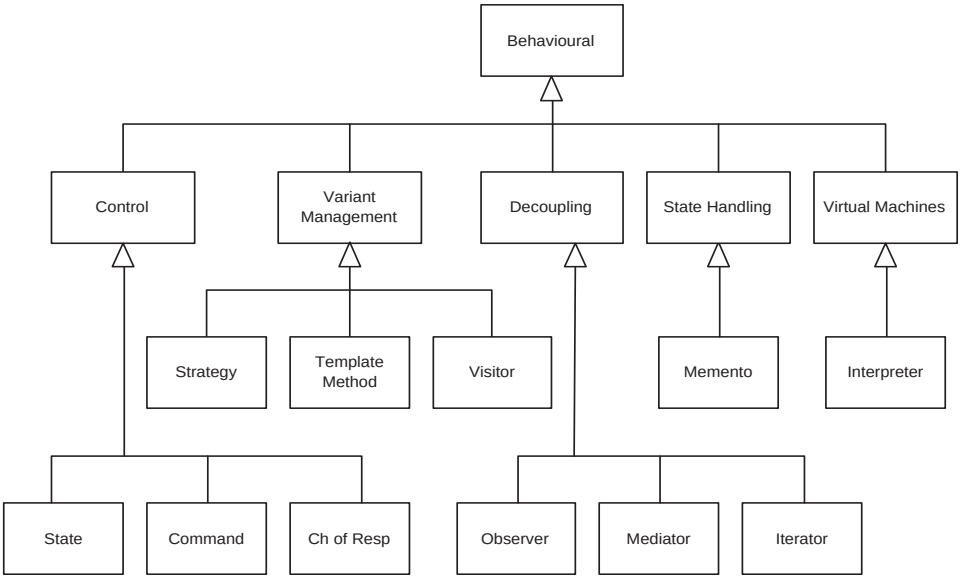

Figure E.3: Behavioural Hierarchy

Appendix F

PATTERN SOURCE CODE AND SCENARIOS

F.1 Source Code – Scenario 1

DrawPanel.java (Client)

```
import java.awt.*;
import javax.swing.*;

class DrawPanel extends JPanel
{
   protected static Shapecomponent composite1 = new Drawcomposite();
   protected static Shapecomponent composite2 = new Drawcomposite();
   protected static Shapecomponent composite3 = new Drawcomposite();
   protected static Shapecomponent colour1 = new Colourdecorator(Color.BLUE);
   protected static Shapecomponent colour2 = new Colourdecorator(Color.RED);
   protected static Shapecomponent colour3 = new Colourdecorator(Color.YELLOW);
   protected static Shapecomponent style1 = new Linestyledecorator(8);
   protected static Shapecomponent style2 = new Linestyledecorator(4);
   protected static Shapecomponent leaf1 = new Lineleaf(10, 10, 20, 20);
   protected static Shapecomponent leaf2 = new Rectangleleaf(20, 20, 40, 40);
   protected static Shapecomponent leaf3 = new Circleleaf(50, 40, 20, 20);

   public void paintComponent(Graphics g)
   {
     composite1.addDrawing(colour1);
     composite1.addDrawing(style2);
     composite1.addDrawing(leaf1);
     composite2.addDrawing(composite1);
     composite2.addDrawing(colour2);
     composite2.addDrawing(style1);
     composite2.addDrawing(leaf2);
     composite3.addDrawing(composite2);
     composite3.addDrawing(colour3);
     composite3.addDrawing(leaf3);
```

```
    composite3.Draw(g);
  }
}
```

Shapecomponent.java (Component)

```
import java.awt.*;

public abstract class Shapecomponent
{
  public void Draw(Graphics g) {}
  public void addDrawing(Shapecomponent draw) {}
  public void removeDrawing(Shapecomponent draw) {}
}
```

Drawcomposite.java (Composite)

```
import java.awt.*;
import java.util.Vector;
import java.util.Enumeration;

public class Drawcomposite extends Shapecomponent
{
  private Vector<Shapecomponent> drawings;
  public Drawcomposite(){drawings = new Vector<Shapecomponent>();}
  public void addDrawing(Shapecomponent draw){drawings.addElement(draw);}
  public void remove(Shapecomponent draw){drawings.removeElement(draw);}
  public Enumeration components(){return drawings.elements();}
  public void Draw(Graphics g)
  {
    Enumeration components = components();
    while (components.hasMoreElements())
    {
      ((Shapecomponent)components.nextElement()).Draw(g);
    }
  }
}
```

Lineleaf.java (Leaf)

```
import java.awt.*;

public class Lineleaf extends Shapecomponent
{
   private int x, y, x1, y1;
   public Lineleaf(int x, int y, int x1, int y1)
   {
     this.x = x;
     this.y = y;
     this.x1 = x1;
     this.y1 = y1;
   }

   public void Draw(Graphics g){g.drawLine(x, y, x1, y1);}
}
```

Circleleaf.java (Leaf)

```
import java.awt.*;

public class Circleleaf extends Shapecomponent
{
   private int x, y, x1, y1;

   public Circleleaf(int x, int y, int x1, int y1)
   {
     this.x = x;
     this.y = y;
     this.x1 = x1;
     this.y1 = y1;
   }

   public void Draw(Graphics g){g.drawOval(x, y, x1, y1);}
}
```

Rectangleleaf.java (Leaf)

```
import java.awt.*;

public class Rectangleleaf extends Shapecomponent
{
   private int x, y, x1, y1;

   public Rectangleleaf(int x, int y, int x1, int y1)
   {
```

```
    this.x = x;
    this.y = y;
    this.x1 = x1;
    this.y1 = y1;
  }

  public void Draw(Graphics g){g.drawRect(x, y, x1, y1);}
}
```

Drawdecorator.java (Decorator)

```
import java.awt.*;

public abstract class Drawdecorator extends Shapecomponent
{
  public Drawdecorator(){}
  public void Draw(Graphics g) {}
}
```

Colourdecorator.java (concreteDecorator)

```
import java.awt.*;

public class Colourdecorator extends Drawdecorator
{
  private Color colour;

  public Colourdecorator(Color colour){this.colour = colour;}

  public void Draw(Graphics g){g.setColor(colour);}
}
```

Linestyledecorator.java (concreteDecorator)

```
import java.awt.*;

public class Linestyledecorator extends Drawdecorator
{
  private int width;

  public Linestyledecorator(int width){this.width = width;}

  public void Draw(Graphics g)
  {
    Graphics2D g2=(Graphics2D)g;
    g2.setStroke(new BasicStroke(width));
```

```
  }
}
```

F.2 Scenario 2

Scenario two is very similar to scenario one. The difference in the two patterns is in how the Composite and Decorator class has been combined into a single class in scenario two.

Analysis

Scenario 2 illustrates a simple drawing package where lines, squares and circles can be drawn within a frame. Each drawing item can be individually decorated or a group of drawing items can be decorated. Each individual item and or groups of items can be collected into a composite object.

Design

Use-Case Diagram

The use-case diagram represents a business process that defines the activities that can be applied to the drawing scenario. In this case, drawing components can be created, decorated and displayed.

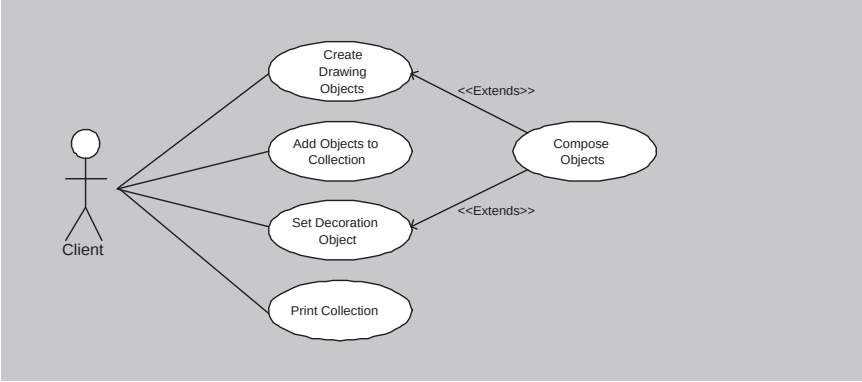

Figure F.1: Use-Case Diagram - Composite combines Decorator

The diagram below shows the class components that collaborate to form the structure of the Composite – Decorator drawing scenario. Three different drawing components can be created and can be decorated with colour and or line sizes can be applied (thickness of lines).

Class Diagram

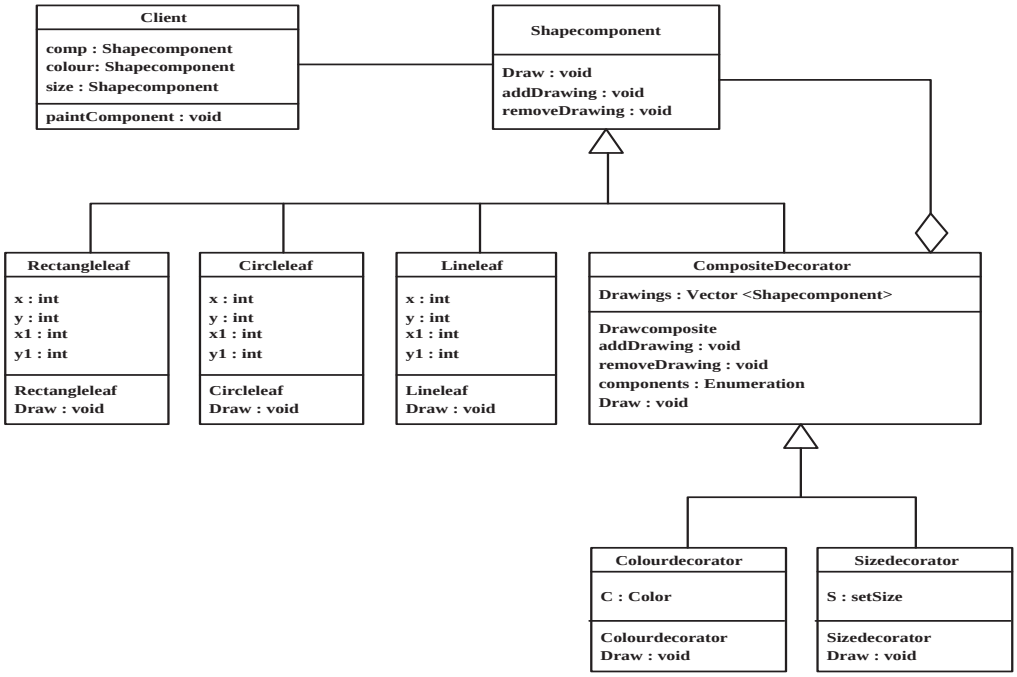

Figure F.2: Class Diagram - Composite combines Decorator

Implementation

This example uses the Composite and Decorator patterns to demonstrate a simple drawing package. For the purpose of demonstration the components are hard coded into the client but in a live application the components would be created dynamically.

Participants

Client

The Client component is a simple driver used to create the drawn components. The client creates the component and decoration objects and adds them to the collection object.

Shapecomponent

The Shapecomponent class specifies an Abstract interface to the main components of the Composite and Decorator. In this respect Shapecomponent is something of a Facade. Shapecomponent defines three methods that can be implemented by all sub-classes. The addDrawing(Shapecomponent draw) and

removeDrawing(Shapecomponent draw) methods are implemented in the Composite class and the Draw(Graphics g) method is implemented in all sub-classes.

CompositeDecorator

CompositeDecorator has two functions; one is to add or remove items from the Collection object (the Vector) - private Vector<Shapecomponent> drawings; and the other is to call back the items from the collection (print to the frame) - ((Shapecomponent)components.nextElement()).Draw(g).

In this example items are only added to the collection - addDrawing(Shapecomponent draw){drawings. addElement(draw)}.

Additionally, CompositeDecorator is the parent class to the concreteDecorator components.

Lineleaf, Circleleaf, Rectangleleaf

Leaf components represent the drawing objects that are added to the composite collection. Each component defines its own type of drawing object, which is called in the Draw method - g.drawLine(x, y, x1, y1).

Colourdecorator, Linestyledecorator

The decoration objects that are used to set the decoration for the drawn components. Each component defines its own type of decoration, which is called in the Draw method - g.setColor(colour). Because they are Shapecomponents, decoration is added to the composite collection as an object.

DrawPanel.java (Client)

```
import java.awt.*;
import javax.swing.*;
class DrawPanel extends JPanel
{
   protected static Shapecomponent composite1 = new Drawcomposite();
   protected static Shapecomponent composite2 = new Drawcomposite();
   protected static Shapecomponent composite3 = new Drawcomposite();
   protected static Shapecomponent colour1 = new Colourdecorator(Color.BLUE);
   protected static Shapecomponent colour2 = new Colourdecorator(Color.RED);
   protected static Shapecomponent colour3 = new Colourdecorator(Color.YELLOW);
   protected static Shapecomponent style1 = new Linestyledecorator(8);
   protected static Shapecomponent style2 = new Linestyledecorator(4);
   protected static Shapecomponent leaf1 = new Lineleaf(10, 10, 20, 20);
   protected static Shapecomponent leaf2 = new Rectangleleaf(20, 20, 40, 40);
   protected static Shapecomponent leaf3 = new Circleleaf(50, 40, 20, 20);
```

```
    public void paintComponent(Graphics g)
    {
      composite1.addDrawing(colour1);
      composite1.addDrawing(style2);
      composite1.addDrawing(leaf1);
      composite2.addDrawing(composite1);
      composite2.addDrawing(colour2);
      composite2.addDrawing(style1);
      composite2.addDrawing(leaf2);
      composite3.addDrawing(composite2);
      composite3.addDrawing(colour3);
      composite3.addDrawing(leaf3);
      composite3.Draw(g);
    }
}
```

Shapecomponent.java (Component)

```
import java.awt.*;
public abstract class Shapecomponent
{
    public void Draw(Graphics g) {}
    public void addDrawing(Shapecomponent draw) {}
    public void removeDrawing(Shapecomponent draw) {}
}
```

CompositeDecorator.java (Composite/Decorator)

```
import java.awt.*;
import java.util.Vector;
import java.util.Enumeration;
public class CompositeDecorator extends Shapecomponent
{
    private Vector<Shapecomponent> drawings;
    public Drawcomposite(){drawings = new Vector<Shapecomponent>();}
    public void addDrawing(Shapecomponent draw){drawings.addElement(draw);}
    public void remove(Shapecomponent draw){drawings.removeElement(draw);}
```

```
  public Enumeration components(){return drawings.elements();}
  public void Draw(Graphics g)
  {
    Enumeration components = components();
    while (components.hasMoreElements())
    {
      ((Shapecomponent)components.nextElement()).Draw(g);
    }
  }
}
```

Lineleaf.java (Leaf)

```
import java.awt.*;
public class Lineleaf extends Shapecomponent
{
  private int x, y, x1, y1;
  public Lineleaf(int x, int y, int x1, int y1)
  {
    this.x = x;
    this.y = y;
    this.x1 = x1;
    this.y1 = y1;
  }
  public void Draw(Graphics g){g.drawLine(x, y, x1, y1);}
}
```

Circleleaf.java (Leaf)

```
import java.awt.*;
public class Circleleaf extends Shapecomponent
{
  private int x, y, x1, y1;
  public Circleleaf(int x, int y, int x1, int y1)
  {
    this.x = x;
    this.y = y;
```

```
    this.x1 = x1;
    this.y1 = y1;
  }
  public void Draw(Graphics g){g.drawOval(x, y, x1, y1);}
}
```

Rectangleleaf.java (Leaf)

```
import java.awt.*;
public class Rectangleleaf extends Shapecomponent
{
  private int x, y, x1, y1;

  public Rectangleleaf(int x, int y, int x1, int y1)
  {
    this.x = x;
    this.y = y;
    this.x1 = x1;
    this.y1 = y1;
  }
  public void Draw(Graphics g){g.drawRect(x, y, x1, y1);}
}
```

Colourdecorator.java (concreteDecorator)

```
import java.awt.*;
public class Colourdecorator extends CompositeDecorator
{
  private Color colour;
  public Colourdecorator(Color colour){this.colour = colour;}
  public void Draw(Graphics g){g.setColor(colour);}
}
```

Linestyledecorator.java (concreteDecorator)

```
import java.awt.*;
public class Linestyledecorator extends CompositeDecorator
{
  private int width;
```

```
  public Linestyledecorator(int width){this.width = width;}
  public void Draw(Graphics g)
  {
    Graphics2D g2=(Graphics2D)g;
    g2.setStroke(new BasicStroke(width));
  }
}
```

F.3 Scenario 3, based on Eckel[39]

Analysis

Scenario 3 illustrates a simple coffee shop where drinks can be ordered to a particular taste. The basic drink contained within a small, medium or large mug can be decorated with particular types of coffee, milk and additives. The cost of individual drinks is composed and via the decorator objects and stored in the composite object. The composite object will display a total sale and the total sales for the day.

Design

Use-Case Diagram

The use-case diagram represents a business process that defines the activities that can be applied to the coffee shop scenario. In this case, basic drinks can be created and extended (decorated) with specific ingredients.

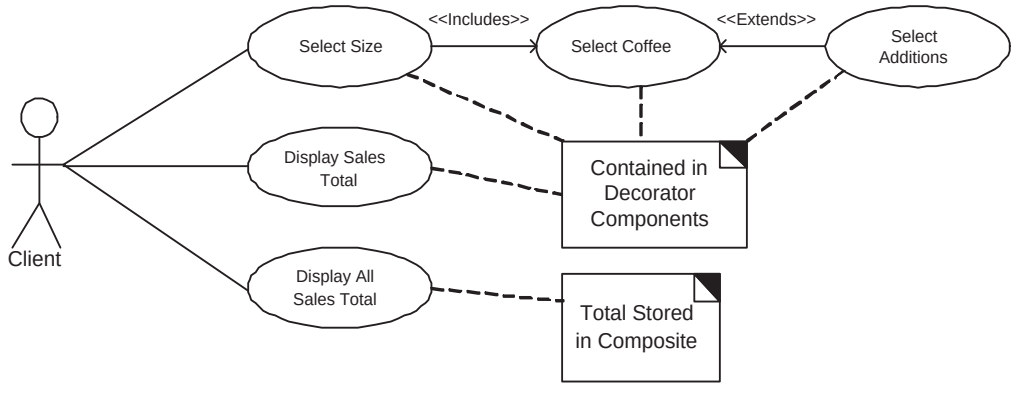

Figure F.3: Use-Case Diagram - Composite combines Decorator

The diagram below shows the class components that collaborate to form the structure of the Composite – Decorator coffee shop scenario. The size of mug can be selected and decorated with required ingredients.

Class Diagram

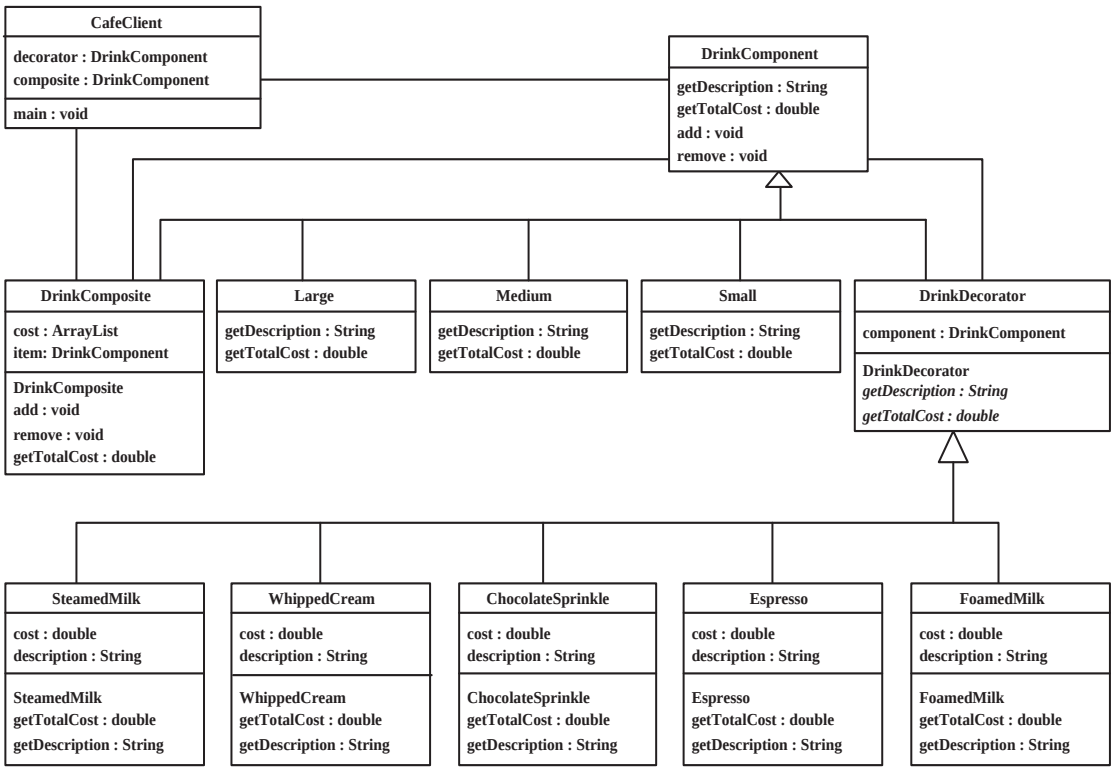

Figure F.4: Class Diagram - Composite combines Decorator

Implementation

This example uses the Composite and Decorator patterns to demonstrate a simple drinks maker. For the purpose of the demonstration the components are hard coded into the client but in a live application the components would be created dynamically.

Participants

Client

The Client component is a simple driver used to create the drinks components. The client creates the drinks as a decoration object. Each component of the decoration object is accessed through the DrinkComponent abstract interface. The client creates the composite component, which is used to store the decorator objects.

DrinkComponent

The DrinkComponent class specifies an abstract interface to the Abstract class DrinkDecorator and Leaf components of the Decorator. DrinkComponent defines two methods that are implemented by all decorator sub-classes. The getTotalCost() and getDescription(). DrinkComponent also specifies two other methods add(DrinkComponent item) and remove(DrinkComponent item), which are implemented by the Composite object.

DrinkDecorator

DrinkDecorator specifies an Abstract interface whose methods are implemented in the Decorator sub-classes. The getTotalCost() and getDescription() methods are declared abstract in the Decorator class as they have no required return value of their own.

SteamedMilk, WhippedCream, ChocolateSprinkle, Espresso, FoamedMilk

Each component defines a specific type of decoration. getTotalCost() returns the cost of the decoration and getDescription() returns a name for the decoration. The constructor of each decoration object takes as a parameter a DrinkComponent, which can be another decoration object or a Leaf object of the Decorator. A drink is built up as a composed object, with the object being closed off by a Leaf node that takes no parameters.

Small, Medium, Large

These components are Leaf components to the Decorator (DrinkDecorator), which in this example is the dominant pattern of the combination. The Leaf component represents an end node of the composed object. As such the component does not take a parameter in its constructor. Like the Decoration components, each method in the Leaf returns a value relative to its purpose.

DrinkComposite

DrinkComposite has two functions; one is to add or remove items from the Collection object (the ArrayList):

```
- private ArrayList cost = new ArrayList();
```

and the other is to call back the items from the collection:

```
- DrinkComponent item = (DrinkComponent)items.next();
```

```
- total += item.getTotalCost();
```

It is in effect only a storage area for DrinkComponents.

CafeClient.java (Client)

```
public class CafeClient
{
  private DrinkComponent cappuccino;
  private DrinkComponent mocha;
  private DrinkComponent latte;
  private DrinkComponent java;
  private DrinkComponent sale1 = new DrinkComposite();
  private DrinkComponent sale2 = new DrinkComposite();
  private DrinkComponent total = new DrinkComposite();

  public static void main(String[] args)
  {
    new CafeClient();
  }

  public CafeClient()
  {
    Sale1();
    Sale2();
    Total();
  }

  public void Sale1()
  {
    cappuccino = new Espresso(new FoamedMilk(new Small()));
    System.out.println(cappuccino.getDescription().trim() + ": GBP " + cappuccino.getTotalCost());
    sale1.add(cappuccino);
    mocha = new Espresso(new SteamedMilk(new ChocolateSprinkle(new WhippedCream(new Medium()))));
    System.out.println(mocha.getDescription().trim() + ": GBP " + mocha.getTotalCost());
    sale1.add(mocha);
    System.out.println("Sale 1 Sub Total = " + sale1.getTotalCost());
  }

  public void Sale2()
  {
    latte = new Espresso(new SteamedMilk(new Medium()));
    System.out.println(latte.getDescription().trim() + ": GBP " + latte.getTotalCost());
    sale2.add(latte);
    java = new Espresso(new FoamedMilk(new ChocolateSprinkle(new WhippedCream(new Large()))));
    System.out.println(java.getDescription().trim() + ": GBP " + java.getTotalCost());
    sale2.add(java);
    System.out.println("Sale 2 Sub Total = " + sale2.getTotalCost());
  }
```

```
    public void Total()
    {
      total.add(sale1);
      total.add(sale2);
      System.out.println("All Sales Sub Total" + total.getTotalCost());
    }
}
```

DrinkComponent.java (Component)

```
public abstract class DrinkComponent
{
    public String getDescription() {return "";}
    public float getTotalCost() {return 0.0;}
    public void add(DrinkComponent item) {}
    public void remove(DrinkComponent item) {}
}
```

DrinkDecorator.java (Decorator)

```
abstract class DrinkDecorator extends DrinkComponent
{
  protected DrinkComponent component;
  DrinkDecorator(DrinkComponent component)
  {
    this.component = component;
  }

  public abstract double getTotalCost();
  public abstract String getDescription();
}
```

Espresso.java (ConcreteDecorator)

```
class Espresso extends DrinkDecorator
{
  private float cost = 2.75;
  private String description = " Espresso";

  public Espresso(DrinkComponent component)
  {
    super(component);
```

```
  }

  public float getTotalCost()
  {
    return component.getTotalCost() + cost;
  }

  public String getDescription()
  {
    return component.getDescription() + description;
  }

}
```

FoamedMilk.java (ConcreteDecorator)

```
class FoamedMilk extends DrinkDecorator
{
  private float cost = 0.25;
  private String description = " Foamed Milk";

  public FoamedMilk(DrinkComponent component)
  {
    super(component);
  }

  public float getTotalCost()
  {
    return component.getTotalCost() + cost;
  }

  public String getDescription()
  {
    return component.getDescription() + description;
  }

}
```

SteamedMilk.java (ConcreteDecorator)

```
class SteamedMilk extends DrinkDecorator
{
  private float cost = 0.25;
  private String description = " Steamed Milk";

  public SteamedMilk(DrinkComponent component)
  {
    super(component);
```

```
  }

  public float getTotalCost()
  {
    return component.getTotalCost() + cost;
  }

  public String getDescription()
  {
    return component.getDescription() + description;
  }

}
```

WhippedCream.java (ConcreteDecorator)

```
class WhippedCream extends DrinkDecorator
{
  private float cost = 0.25;
  private String description = " Whipped Cream";

  public WhippedCream(DrinkComponent component)
  {
    super(component);
  }

  public float getTotalCost()
  {
    return component.getTotalCost() + cost;
  }

  public String getDescription()
  {
    return component.getDescription() + description;
  }

}
```

ChocolateSprinkle.java (ConcreteDecorator)

```
class ChocolateSprinkle extends DrinkDecorator
{
  private float cost = 0.25;
  private String description = " Chocolate Sprinkle";

  public ChocolateSprinkle(DrinkComponent component)
  {
    super(component);
```

```
  }

  public float getTotalCost()
  {
    return component.getTotalCost() + cost;
  }

  public String getDescription()
  {
    return component.getDescription() + description;
  }
}
```

Small.java (Leaf)

```
class Small extends DrinkComponent
{
  public String getDescription()
  {
    return "Small Mug";
  }

  public float getTotalCost()
  {
    return 0.5;
  }
}
```

Medium.java (Leaf)

```
class Medium extends DrinkComponent
{
  public String getDescription()
  {
    return "Medium Mug";
  }

  public float getTotalCost()
  {
    return 0.75;
  }
}
```

Large.java (Leaf)

```
class Large extends DrinkComponent
{
  public String getDescription()
  {
    return "Large Mug";
  }

  public float getTotalCost()
  {
    return 1.0;
  }
}
```

DrinkComposite.java (Composite)

```
import java.util.ArrayList;
import java.util.Iterator;

class DrinkComposite
{
  private ArrayList cost = new ArrayList();
  private DrinkComponent item;

  public DrinkComposite() {}
  public void add(DrinkComponent element){cost.add(element);}
  public void remove(DrinkComponent element){cost.remove(element);}
  public float getTotalCost()
  {
    double total = 0;
    Iterator items = cost.iterator();
    while(items.hasNext())
    {
      item = (DrinkComponent)items.next();
      total += item.getTotalCost();
    }
    return total;
  }
}
```

Appendix G

SOFTWARE METRIC SUITE

The metrics described below are the suite of metrics contained in the Together Architect[16] modelling tool that was used for design components seen in the generative patterns. The descriptions below are those contained in the help file of the tool.

G.1 Basic[16]

- **Class Interface Width (CIW)** CIW is defined as the number of members of the class that belong to the interface of the class. The members that belongs to the interface of the class are the public, non-inherited methods and data members of a class.

- **Lines Of Code (LOC)** LOC is the number of lines of code in a namespace, classifier or method, including comments and white-lines.

- **Number Of Attributes (NOA)** Counts the number of attributes. Inherited attributes may be counted optionally. If a class has a high number of attributes, it may be wise to consider whether it would be appropriate to divide it into subclasses.

- **Number Of Classes (NOC)** NOC counts the number of classes.

- **Number Of Constructors (NOCON)** Counts the number of constructors. You can specify whether to count all constructors or only public, or protected, and so on.

- **Number Of Import Statements (NOIS)** Counts the number of imported packages /classes. This measure can highlight excessive importing and can also be used as a measure of coupling.

- **Number Of Members (NOM)** Counts the number of members, i.e. attributes and operations. Inherited members can optionally be included in the total. If a class has a high number of members, it might be wise to consider whether it would be appropriate to divide it into subclasses.

- **Number Of Operations (NOO)** NOO counts the number of operations. Inherited operations may be counted optionally. If a class has a high number of operations, it may be wise to consider whether it would be appropriate to divide it into subclasses.

- **Number Of Parameters (NOP)** NOP is the number of parameters that build the signature of a method.

- **Number Of Public Attributes (NOPA)** NOPA is defined as the number of non-inherited attributes that belong to the interface of a class.

- **Number of Accessor Methods (NAM)** NAM is defined as the number of the non-inherited accessor methods (properties) declared in the interface of a class. To find accessor methods, NAM relies on the name conventions.

- **Package Interface Size (PIS)** PIS is the number of classes in a package that are used from outside the package . A class uses a namespace if it calls methods, accesses attributes or extends a class declared in that namespace.

- **Package Size (PS)** PS is the number of classes which are defined in the measured package . Inner classes are not counted.

G.2 Cohesion[16]

- **Access of Local Data (ALD)** ALD counts the number of the data accessed in the given method, which is local to the class where the method is defined. Inherited data should be counted too.

- **Class Locality (CL)** CL is computed as the relative number of dependencies that a class has in its own package. In order to compute the metric the CBO value is divided by the total number of classes on which the measured class depends on. Inner classes should not be counted.

- **Lack of Cohesion of Methods 1 (LCOM1)** Takes each pair of methods in the class and determines the set of fields they each access. If they have disjoint sets of field accesses, increase the count P by one. If they share at least one field access, then increase Q by one. After considering each pair of methods:

  RESULT = (P > Q) ? (P - Q) : 0

A low value indicates high coupling between methods, which indicates a high testing effort because many methods can affect the same attributes and potentially has low reusability. The definition of this metric was provided by Chidamber and Kemerer[21].

- **Lack of Cohesion Of Methods 2 (LCOM2)** Counts the percentage of methods that do not access a specific attribute, averaged over all the attributes in the class. A high value of cohesion (a low lack of cohesion) implies that the class is well designed. A cohesive class will tend to provide a high degree of encapsulation, whereas a lack of cohesion decreases encapsulation and increases complexity.

- **Lack Of Cohesion Of Methods 3 (LCOM3)** Measures the dissimilarity of methods in a class by its attributes.

  m - number of methods in a class
  a - number of attributes in a class
  mA - number of methods that access an attribute
  EmA - sum of mA for each attribute

  RESULT = 100*(EmA/a-m)/(1-m)

  The definition of this metric was proposed by Henderson-Sellers[57]. A low value indicates good class subdivision, implying simplicity and high reusability. A high lacking of cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

  If there are no more than one method in a class, LCOM3 is undefined. If there are no variables in a class, LCOM3 is undefined. An undefined LCOM3 is displayed as -1. Methods that do not access any class variables are not taken into account.

- **Package Cohesion (PC)** PC is defined as the relative number of class pair from a package between which a dependency exists.

- **Tight Class Cohesion (TCC)** TCC is defined as the relative number of directly connected methods. Two methods are directly connected if they access a common instance variable of the class.

G.3 Complexity[16]

- **Attribute Complexity (AC)** Defined as the sum of each attribute's weight in the class. You can set up weights for types, the enum type and the array type separately. Use "*" to define

types of a package with all its subpackages . For example, java.lang.* means that the row defines all classes of the java.lang package and its subpackages . To process all types not listed in the table, specify the last row as "*". The row order is important, because checking of attributes goes from the top of the table downwards. (Repetitions of a type aren't counted, so if a specific type follows a more general type that already included it, the specific type isn't counted. For example, java.lang.* won't be counted if it comes after java.* .)

- **Cyclomatic Complexity (CC)** CC represents number of cycles in the measured method. This measure represents the cognitive complexity of the class. It counts the number of possible paths through an algorithm by counting the number of distinct regions on a flowgraph, meaning the number of if, for, and while statements in the operation's body. Case labels for switch statements are counted if the Case as branch property is activated. A strict definition of CC (introduced by McCabe[78]) looks at a program's control flow graph as a measure of its complexity:

  CC = L - N + 2P

  where L is the number of links in the control flow graph, N is the number of nodes in the control flow graph, and P is the number of disconnected parts in the control flow graph. For example, consider a method which consists of an if statement:

```
if (x > 0)
{
  x = x + 1;
}
else
{
  x = x - 1;
}
```

  CC = L - N + 2P = 4 - 4 + 2*1 = 2

  A less formal definition is:

  CC = D + 1

  where D is the number of binary decisions in the control flow graph, if it has only one entry and exit. In other words, the number of if, for and while statements and number of logical and, and or operators.

  For the example above:

$CC = D + 1 = 1 + 1 = 2$

- **Maximum Number Of Branches (MNOB)** MNOB is defined as the maximum number of if-else and/or case branches in the method.

- **Number Of Local Variables (NOLV)** NOLV counts how many local variables are declared within a method.

- **Number Of Remote Methods (NORM)** Processes all methods and constructors and counts the number of various remote methods called. A remote method is defined as a method that is not declared in the class itself or in its ancestors.

- **Response For Class (RFC)** The size of the response set for the class includes methods in the class's inheritance hierarchy and methods that can be invoked on other objects. A class, which provides a larger response set, is considered to be more complex and require more effort in testing than one with a smaller overall design complexity. This measure is calculated as the 'Number Of Operations' + 'Number Of Remote Methods'.

- **Weight Of a Class (WOC)** WOC is the number of non-accessor methods in the interface of the class, divided by the total number of interface members. Inherited members are not counted. The members that belong to the interface of the class are the public, non-inherited methods and fields of a class.

- **Weighted Methods Per Class 1 (WMPC1)** This metric is the sum of the complexity of all methods for a class, where each method is weighted by its cyclomatic complexity. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. Only methods specified in a class are included, that is, any methods inherited from a parent are excluded.

- **Weighted Methods Per Class 2 (WMPC2)** This metric is intended to measure the complexity of a class, assuming that a class with more methods than another is more complex, and that a method with more parameters than another is also likely to be more complex. The metric counts methods and parameters for a class. Only methods specified in a class are included, that is, any methods inherited from a parent are excluded.

G.4 Coupling[16]

- **Access Of Foreign Data (AOFD)** AOFD represents the number of external classes from which a given class accesses attributes , directly or via accessor methods . The higher the AOFD value for a class, the higher the probability that the class is or is about to become an unfocused-class. Inner classes and superclasses are not counted.

- **Access of Import Data (AID)** AID the amount of data members accessed in a method directly or via accessor-methods , from which the definition-class of the method is not derived.

- **Average Use of Interface (AUF)** AUF metric is defined as the average number of interface members of a class that are used by another class. AUF is computed by totalling up the number of used members for each of client-classes and dividing it by the number of client classes (COC).

- **Changing Classes (ChC)** ChC metric is defined as the number of client-classes where the changes must be operated in result a change in the server-class.

- **Changing Methods (CM)** CM is defined as the number of distinct methods in the system that would be potentially affected by changes operated in the measured class. The methods potentially affected are all those that access an attribute and/or call a method and/or redefine a method of given class.

- **Clients Of Class (COC)** COC is defined as the number of classes that use the interface of the measured class. Inner classes are not counted. In the context of this metric, class A uses interface of a class C if (at least) it calls a public method or accesses a public attribute of that class.

- **Coupling Between Objects (CBO)** CBO represents the number of other classes to which a class is coupled to. Counts the number of reference types that are used in attribute declarations, formal parameters, return types, throws declarations, local variables, and types from which attribute and method selections are made. Primitive types, types from java.lang package and supertypes are not counted.

  Excessive coupling between objects is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class coupling should be kept to a minimum. The larger the number of coupling, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. A measure of coupling is useful to determine

how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

- **Coupling Factor (CF)** This measure is from the MOOD (Metrics for Object-Oriented Development) suite. It is calculated as a fraction. The numerator represents the number of non-inheritance couplings. The denominator is the maximum possible number of couplings in a system.

- **Data Abstraction Coupling (DAC)** DAC counts the number of reference types used in the attribute declarations. Primitive types, types from java.lang package and supertypes are not counted.

- **Dependency Dispersion (DD)** DD is the number of other packages on which a class depends. The class depends on a package if it depends on one of the classes from that package.

- **FanOut (FO)** FO counts the number of reference types that are used in attribute declarations, formal parameters, return types, throws declarations and local variables. Simple types and supertypes are not counted.

- **Message Passing Coupling (MPC)** MPC counts the number of method call expressions made into body of the measured method.

- **Method Invocation Coupling (MIC)** MIC is the (relative) number of other classes to which a certain class sends messages.

  $MIC_{norm} = n_{MIC}/(\mathrm{N} - 1)$

  where N is the total number of classes defined in the project, and n_{MIC} the number of classes to which messages are sent.

  Viewpoints: (These viewpoints summarize the impact that coupling has on some external attributes).

  1. **Maintainability**. The maintenance of a strongly coupled class (high MIC value) is more difficult to do because of its dependency on the classes it is coupled to.
  2. **Comprehensibility**. A strongly coupled class is more difficult to understand, as its understanding implies a partial (or sometimes total) understanding of the classes it is coupled to.
  3. **Error-prone and Testability**. Errors in a class is directly proportional to the number of couplings to other classes. Consequently high coupling has a negative impact on testability.

Observations:

1. The proposed definition of MIC is obviously a normalized one. Although this has advantages, but for some viewpoints, like maintainability, it is more important to operate on the absolute values, i.e. the number of classes to which it is coupled.
2. For some viewpoints it might be important to count only the couplings of the system to user-defined classes, i.e. exclude the library classes.

Radu Marinescu[75]

- **Number Of Client Packages (NOCP)** NOCP is the number of other packages that use the measured package. A package uses another package if at least one if its classes is using that package (i.e. calls methods, accesses attributes or extends a class declared in that package).

- **Number Of External Dependencies (NOED)** NOED is the number of classes from other packages on which the measured class depends on. A class A depends on another class B, if class A calls methods and/or accesses attributes and/or extends class B.

- **Number of Client Classes (NCC)** NCC represents the number of classes from other packages that use the measured package. A class uses a package if it calls methods, accesses attributes or extends a class declared in that package.

- **Number of import classes (NIC)** The NIC metric counts the number of external classes from which the given method uses data.

- **Package Usage Ratio (PUR)** The PUR metric is defined as the relative number of classes from the measured package that are used from outside that namespace. The number of uses classes will be divided by the total number of classes in the package: inner classes are excluded. Thus:

  PUR = $\frac{PIS}{PS}$

- **Violations of Demeters Law (VOD)**

  Law of Demeter :

  Definition 1 (Client) Method M is a client of method f attached to class C, if inside M message f is sent to an object of class C, or to C. If f is specialized in one or more subclasses, then M is only a client of f attached to the highest class in the hierarchy. Method M is a client of some method attached to C.

Definition 2 (Supplier) If *M* is a client of class *C* then *C* is a supplier to *M*. In other words, a supplier class to a method is a class whose methods are called in the method.

Definition 3 (Acquaintance Class) A class *C1* is an acquaintance class of method *M* attached to class *C2*, if *C1* is a supplier to *M* and *C1* is not one of the following:

1. the same as *C2*;
2. a class used in the declaration of an argument of *M*
3. a class used in the declaration of an instance variable of *C2*

Definition 4 (Preferred-acquaintance Class) A preferred-acquaintance class of method *M* is either:

1. a class of objects created directly in *M*, or
2. a class used in the declaration of a global variable used in *M*.

Realization note: Direct creation means that a given object is created via operator new.

Definition 5 (Preferred-supplier class) Class *B* is called a preferred-supplier to method *M* (attached to class *C*) if *B* is a supplier to *M* and one of the following conditions holds:

1. *B* is used in the declaration of an instance variable of *C*,
2. *B* is used in the declaration of an argument of *M*, including *C* and its superclasses,
3. *B* is a preferred acquaintance class of *M*.

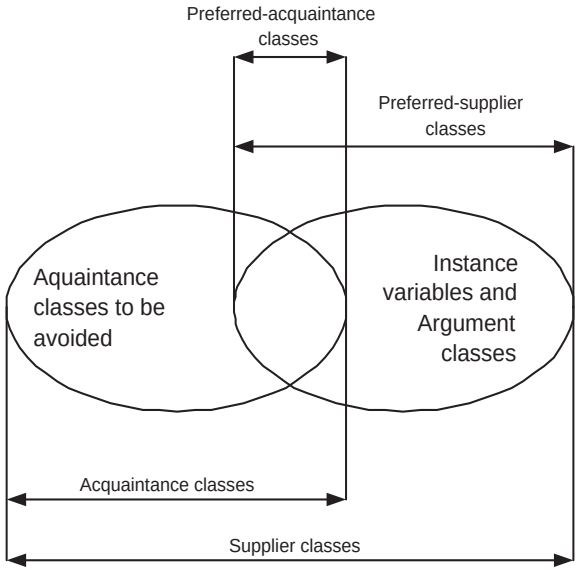

Figure G.1: The relation among the different types of supplier classes

The class form of Demeters Law has two versions: a strict version and a minimization version. The strict form of the law states that every supplier class of a method must be a preferred supplier. The minimization form is more permissive than the first version and requires only minimizing the number of acquaintance classes of each method.

Observations.

1. The motivation behind the Law of Demeter is to ensure that the software is as modular as possible. The Law effectively reduces the occurrences of certain nested message sends and simplifies the methods.
2. The definition of the Law makes a difference between the classes associated with the declaration of the method and the classes used in the body of the method, i.e. the classes associated with its implementation. The former includes the class where the method is attached, its superclasses, the classes used in the declarations of the instance variables and the classes used to declare the arguments of the method. In some sense, there are 'automatic' consequences of the method declaration. They can be easily derived from the code and shown by a browser. All other supplier classes to the methods are introduced in the body of the function, which means these couples were created at the time of concretely implementing the method. They can only be determined by a careful reading of the implementation.

Violations of Demeters Law - VOD

The definition of this metric is based on the minimization form of the Law of Demeter. Based on the concepts defined there, and remembering that the minimization form of Demeters Law requires that the number of acquaintance classes should be kept low, the VOD metric is defined.

Definition 6 (VOD Metric) Being given a class C and A the set of all its acquaintance classes,

VOD(C) = |A|

Informally, VOD is the number of acquaintance classes of a given class. Keeping the VOD value for a class low offers a number of benefits:

1. *Coupling control.* A project with a low VOD value is the sign of minimal "use" coupling between abstractions. That means that a reduced number of methods can be invoked. This makes the methods more reusable.
2. *Structure hiding.* Reducing VOD represents in fact the reducing of the direct retrieval of subparts of the "part-of" hierarchy. In other words, public members should be used in a restricted way.
3. *Localization of information.* A low VOD value also means that the class information is localized. This reduces the programming complexity.

Radu Marinescu[75]

- **Weighted Changing Methods (WCM)** For each method that would be counted by the CM metric, a "weight" is given to it. The weight is defined as the number of distinct members from the server-class that are referenced in that method. WCM is computed as the sum of the weights of all the methods affected by changes.

G.5 Encapsulation[16]

- **Attribute Hiding Factor (AHF)** This measure is from the MOOD (Metrics for Object-Oriented Development) suite. It is calculated as a fraction. The numerator is the sum of the invisibilities of all attributes defined in all classes. The invisibility of an attribute is the percentage of the total classes (excluding the class owner of attribute) from which this attribute is not visible. The denominator is the total number of attributes defined in the project.

- **Method Hiding Factor (MHF)** This measure is from the MOOD (Metrics for Object-Oriented Development) suite. It is calculated as a fraction. The numerator is the sum of the invisibilities of all methods defined in all classes. The invisibility of a method is the percentage of the total classes (excluding the class owner of method) from which this method is not visible. The denominator is the total number of methods defined in the project.

G.6 Halstead[16]

- **Halstead Difficulty (HDiff)** This measure is one of the Halstead Software Science metrics. It is calculated as ('Number of Unique Operators' / 2) * ('Number of Operands' / 'Number of Unique Operands').

- **Halstead Effort (HEff)** This measure is one of the Halstead Software Science metrics. It is calculated as 'Halstead Difficulty' * 'Halstead Program Volume'.

- **Halstead Program Length (HPLen)** This measure is one of the Halstead Software Science metrics. It is calculated as 'Number of Operators' + 'Number of Operands'.

- **Halstead Program Vocabulary (HPVoc)** This measure is one of the Halstead Software Science metrics. It is calculated as 'Number of Unique Operators' + 'Number of Unique Operands'.

- **Halstead Program Volume (HPVol)** This measure is one of the Halstead Software Science metrics. It is calculated as 'Halstead Program Length' * Log2('Halstead Program Vocabulary').

- **Number of Operands (NOprnd)** This measure is used as an input to the Halstead Software Science metrics. It counts the number of operands used in a class.

- **Number of Operators (NOprtr)** This measure is used as an input to the Halstead Software Science metrics. It counts the number of operators used in a class.

- **Number of Unique Operands (NUOprnd)** This measure is used as an input to the Halstead Software Science metrics. It counts the number of unique operands used in a class.

- **Number of Unique Operators (NUOprtr)** This measure is used as an input to the Halstead Software Science metrics. It counts the number of unique operators used in a class.

G.7 Inheritance[16]

- **Attribute Inheritance Factor (AIF)** This measure is from the MOOD (Metrics for Object-Oriented Development) suite. It is calculated as a fraction. The numerator is the sum of inherited attributes in all classes in the project. The denominator is the total number of available attributes (locally defined plus inherited) for all classes.

- **Depth Of Inheritance Hierarchy (DOIH)** The length of the inheritance chain from the root of the inheritance tree to the measured class is the DOIH metric for the class.

- **Method Inheritance Factor (MIF)** This measure is from the MOOD (Metrics for Object-Oriented Development) suite. It is calculated as a fraction. The numerator is the sum of inherited methods in all classes in the project. The denominator is the total number of available methods (locally defined plus inherited) for all classes.

- **Number Of Child Classes (NOCC)** NOCC counts the number of classes directly or indirectly derived from the measured class.

G.8 Inheritance-Based Coupling[16]

- **Inheritance Usage Ratio (IUR)** The IUR metric is a metric defined between a subclass and one of its ancestor classes. It is the relative number of inheritance-specific members from the ancestor class used in the derived class. A member of an ancestor class is an inheritance-specific member if its usage is related to inheritance. There are two identified inheritance-specific members:

  - protected data members and methods;
  - non-private virtual methods.

  The IUR is computed by counting the number of inheritance-specific members of the ancestor class that are used in the subclass, and then divide it by the total numbers of inheritance-specific members from the ancestor. The only usages that are counted are: the access of protected data members, the call of protected methods and the redefinition of a virtual method.

- **Average Inheritance Usage Ratio (AIUR)** AIUR is defined for a derived class as the average value of the IUR metric computed between that class and all its ancestor classes.

- **Total Reuse of Ancestor percentage (TRAp) & Total Reuse of Ancestor unitary (TRAu) Reuse of Ancestors**.

  The RA Metric

  Definition 1 (Reuse of Ancestor-class - RA) The RA metric between a class C and one of its ancestor classes A.

  Explanations The RA metric quantifies the reuse from a super class by totalizing this reuse from all of its methods. The degree to which a method reuses an ancestor class is variable. The way this reuse degree is calculated depends on the goals of the measurement. The metric is parameterised with a family of metrics called Reuse Degree of Ancestor-class (RDA) that evaluates this reuse degree.

  The RDA Metrics

  Definition 2 (Reuse Degree of Ancestor-class) A function expressing the measure of reuse of an ancestor class A in method mth_i of class C is called Reuse Degree of Ancestor-class A in method mth_i.

  RDA : SMF_C X SAC_C –> [0; 1]

  where SMF_C is the set of all member functions (methods) in class C and SAC_C is the set of ancestors classes A for class C.

  Observations Because the stability of the ancestor-class plays an important role from the perspective of the client class, the definition of RDA also considers the stability of ancestors interface.

  The Total RA Metric - TRA

  The RA metric has two parameters: a particular class and one of its ancestor classes. It is necessary to have also a metric that expresses the total reuse (from all the ancestors) for a given class. The definition of this new metric is based on the definition of the already defined RA metric.

  Definition 3 (Total Reuse from Ancestors - TRA) The Total Reuse from Ancestors metric for a class C is defined as the sum of all RA values between class C and its superclasses.

  Radu Marinescu[75]

- **Total Reuse in Descendants percentage (TRDp) & Total Reuse in Descendants unitary (TRDu) Reuse in Descendants**.

The RD Metric

Definition 1 (Reuse in Descendant-class - RD) The RD metric between a class C and one of its descendant classes D.

Explanations The RD metric quantities the totalized reuse of all the members of a class C, in one of its descendant classes. The degree to which a particular member is reused in a descendant class is variable. The way this reuse degree is calculated depends on the goals of the measurement. Analogous to the RA metric, the RD metric is parameterised with a family of metrics called Reuse Degree in Descendant-class (RDD), that quantities this reuse degree.

The RDD Metrics

Definition 2 (Reuse Degree in Descendant Class) A function expressing the measure of reuse of a class member m_C class C in a descendent class D is called Reuse Degree of m_C in Descendant-class D.

RDD : SM_C X SDC_C – [0; 1]

where SM_C is the set of all members in class C and SDCC is the set of descendant classes D for class C .

The Total RD Metric - TRD In the previous sections the RD metric was defined with two parameters: a particular class and a descendant of that class. In the same way that the TRA is defined it is considered necessary to define a metric that expresses the total value for the reuse of a class by all its descendants. There are two viewpoints for the interpretation of this metric.

1. **Maintainability**. A high TRD value for a class indicates that a change in that class has a high impact on the underlying class-hierarchy, i.e. its descendants.
2. **Degree of Member Reuse**. A high TRD for a class indicates that the very most of its members are reused in the sub-classes.

It is observed that because their focus is strongly different it would be quite impossible to have a single definition for TRD. Therefore, a definition is proposed for each one of the two viewpoints:

Definition 3 (Descendants-based Definition of TRD) The Total Reuse in Descendants metric for a class C is defined as the sum of all RD values between class C and its descendants.

Radu Marinescu[75]

G.9 Maximum[16]

- **Maximum Number Of Levels (MNOL)** Counts the maximum depth of if, for and while branches in the bodies of methods. Logical units with a high number of nested levels might need implementation simplification and process improvements, because groups that contain more than seven pieces of information are increasingly harder for people to understand in problem solving.

- **Maximum Number Of Parameters (MNOP)** Counts the highest number of parameters defined for a single operation, from among all the operations in the class. Methods with many parameters tend to be more specialized and so are less likely to be reusable.

- **Maximum Size Of Operation (MSOO)** Counts the maximum size of operations for a class. Method size is determined in terms of cyclomatic complexity, meaning the number of if, for, and while statements in the operation's body. Case labels for switch statements can be optionally included.

G.10 Polymorphism[16]

- **Number Of Added Methods (NOAM)** NOAM counts the number of operations added by a class. Inherited and overridden operations are not counted. Classes without parents are not processed. The large value of this measure indicates that the functionality of the given class becomes increasingly distinct from that of the parent classes. In this case, it should be considered whether this class should genuinely be inheriting from the parent or if it could be broken down into several smaller classes.

- **Number Of Overridden Methods (NOOM)** NOOM counts the number of inherited operations, which a class overrides. Classes without parents are not processed. High values tend to indicate design problems, i.e. subclasses should generally add to and extend the functionality of the parent classes rather than overriding them.

- **Polymorphism Factor (PF)** This measure is from the MOOD (Metrics for Object-Oriented Development) suite. It is calculated as a fraction. The numerator is the sum of overriding methods in all classes. This is the actual number of possible different polymorphic situations. A given message sent to a class can be bound, statically or dynamically, to a named method implementation. The latter can have as many shapes (morphs) as the number of times this same method is overridden in that class's descendants. The denominator represents the maximum

number of possible distinct polymorphic situations for that class as the sum for each class of the number of new methods multiplied by the number of descendants. This maximum would be the case where all new methods defined in each class would be overridden in all of their derived classes.

G.11 Ratio[16]

- **Comment Ratio (CR)** Counts the ratio of documentation and/or implementation comments to total lines of code (comments are included in the code count). You can also specify which type of comments to use for the ratio.
  - Documentation comments are Javadoc comments.
  - Implementation comments are any other type of comments.
- **Percentage of Package Members (PPkgM)** Counts the percentage of package members in a class.
- **Percentage of Private Members (PPrivM)** Counts the percentage of private members in a class.
- **Percentage of Protected Members (PProtM)** Counts the percentage of protected members in a class.
- **Percentage of Public Members (PPubM)** Counts the proportion of vulnerable members in a class. A large proportion of such members means that the class has high potential to be affected by external classes and means that increased effort will be needed to test such a class thoroughly.
- **True Comment Ratio (TCR)** Counts the ratio of documentation and/or implementation comments to total lines of code (all comments are excluded from the code count). You can also specify which type of comments to use for the ratio.
  - Documentation comments are Javadoc comments.
  - Implementation comments are any other type of comments.

G.12 Test Coverage[16]

- **JUnit test Coverage (JUC)** JUC measures JUnit test coverage for methods and classes. For a method, the value of JUC is 1 if the method is directly or indirectly called from any JUnit test case and 0 otherwise. For a class, the value of JUC is the percentage of methods checked with JUnit tests.

Appendix H

ADDITIONAL CASE-STUDIES

H.1 A Simple Case Study using Composite and Builder

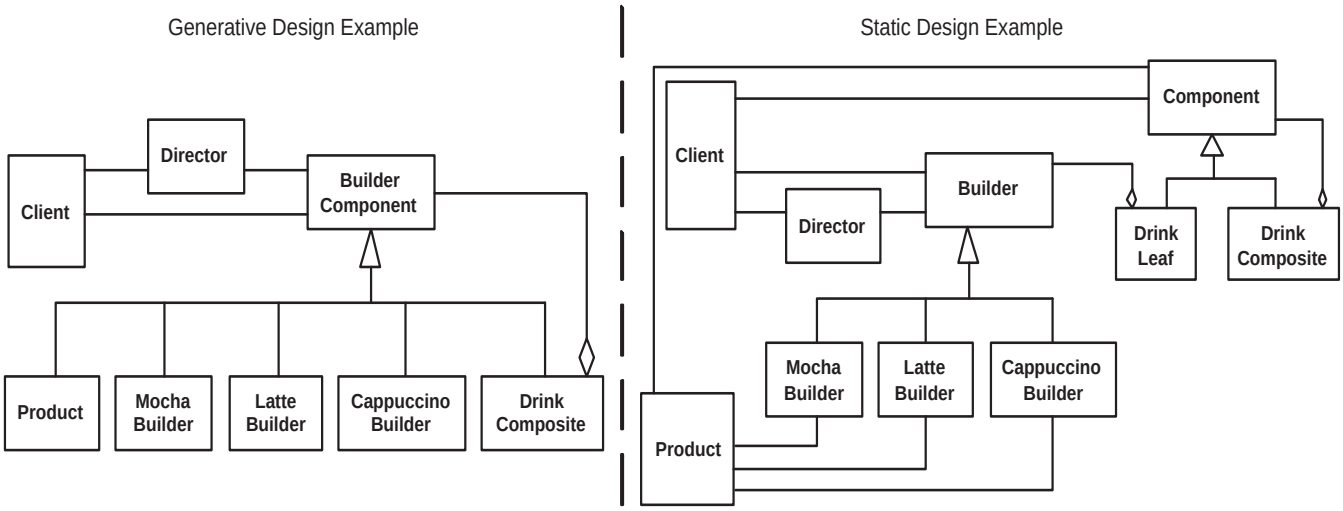

Figure H.1: Generative vs. Static – Composite and Builder

Figure H.1 above provides a class diagram for the comparative examples of the composite and builder patterns used in a generative and static pattern environment. As can be seen from the diagram, the generative example on the left has an interface (BuilderComponent) that is combined from the two interface components that are used in the static example on the right. The three sub-components of the Builder interface class and the Product class from the static pattern example are now leaf components to the DrinkComposite class in the generative pattern example.

In order for the two patterns to work together in the static environment, a product object is created and added to a collection object in the DrinkLeaf component of the composite pattern. As such, multiple product objects can be added to one or more DrinkLeaf components and one or more DrinkLeaf components can be added to a DrinkComposite component. DrinkComposite components can be added to other DrinkComposite components as is intended with a composite pattern.

In the generative example, because any product object that is created is now a leaf component to the DrinkComposite component, it can be added directly to a DrinkComposite object.

Table H.1 shows the overall results of the metrics that were produced from the generative and static examples of the composite and builder patterns described above.

Metric	Generative Patterns	Static Patterns	Difference (%)
CBO	18	20	+
CC	17	17	/
LCOM	88	88	/
LOC	376	403	+6.7%
RFC	13	11	–
WMPC	13	11	–
NOC	8	10	+
EXE SIZE	14.8	16.1	+8.1%

Table H.1: General statistics for the Generative and Static versions of Composite and Builder

The statistics in Table H.1 indicate that the generative pattern will require less testing in respect of the CBO metric but will require more testing in respect of the RFC metric. As can be seen in Table H.2, the higher value for the CBO metrics comes from the client of the static example, which has to communicate with two interface components instead of just one interface component in the generative example. However, the higher value of the RFC metric in the generative pattern comes from the ComponentBuilder interface, which is now having to define two sets of methods for different sub components. The first set of method definitions relate to the CoffeeProduct class, where values are set for the created object. The second set of method definitions relate to the ConcreteBuilder classes, which builds the values into the CoffeeProduct object.

Although there is less coupling in the generative example, as confirmed by the CBO metric, there is a higher degree of complexity. The higher value in the WMPC metric confirms that the ComponentBuilder class is more complex and will therefore require more testing, and if required, more complex maintenance.

Whilst the two separate interfaces (Builder and Component) in the static example still have collective values lower than the ComponentBuilder interface of the generative example, the static example will require some additional testing for the DrinkLeaf component.

The overall viewpoint on this pair of patterns is that the generative example has more points in favour than the static example. This takes into account the reduction in the number of lines of code, the number of classes and the size of the executable file, which are in favour of the generative example.

Class	CBO		CC		LCOM		LOC		RFC		WMPC	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Client	18	20	17	17			118	120	2	2	2	2
	+		/				+		/		/	
Button	6	7	17	17			43	44	11	13	17	17
Handler	+		/				+		+		/	

Class	CBO		CC		LCOM		LOC		RFC		WMPC	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Component	1	*	1	*		*	28	*	13	*	13	*
Builder	*		*		*		*		*		*	
Component	*	1	*	1	*		*	11	*	5	*	5
	*		*		*		*		*		*	
Builder	*	1	*	1	*		*	14	*	6	*	6
	*		*		*		*		*		*	
Drink	*	4	*	2	*	0	*	28	*	8	*	5
Leaf	*		*		*		*		*		*	

Table H.2: Individual statistics for the Generative and Static versions of Composite and Builder

Like the examples in Chapter Seven, the individual class statistics for the like-for-like components in the examples are identical throughout all metric categories, therefore they are not included in Table H.2. In this example this equates to the ConcreteBuilder components, the CoffeeProduct, the CoffeeDirector and the DrinkComposite class. Again, like the previous example the reason for this is modularity, in that each corresponding component provides identical functionality. The only exception in like-for-like components is the client. For the client there is a minor difference in that it communicates with two separate interfaces.

H.2 A Simple Case Study using Command and Builder

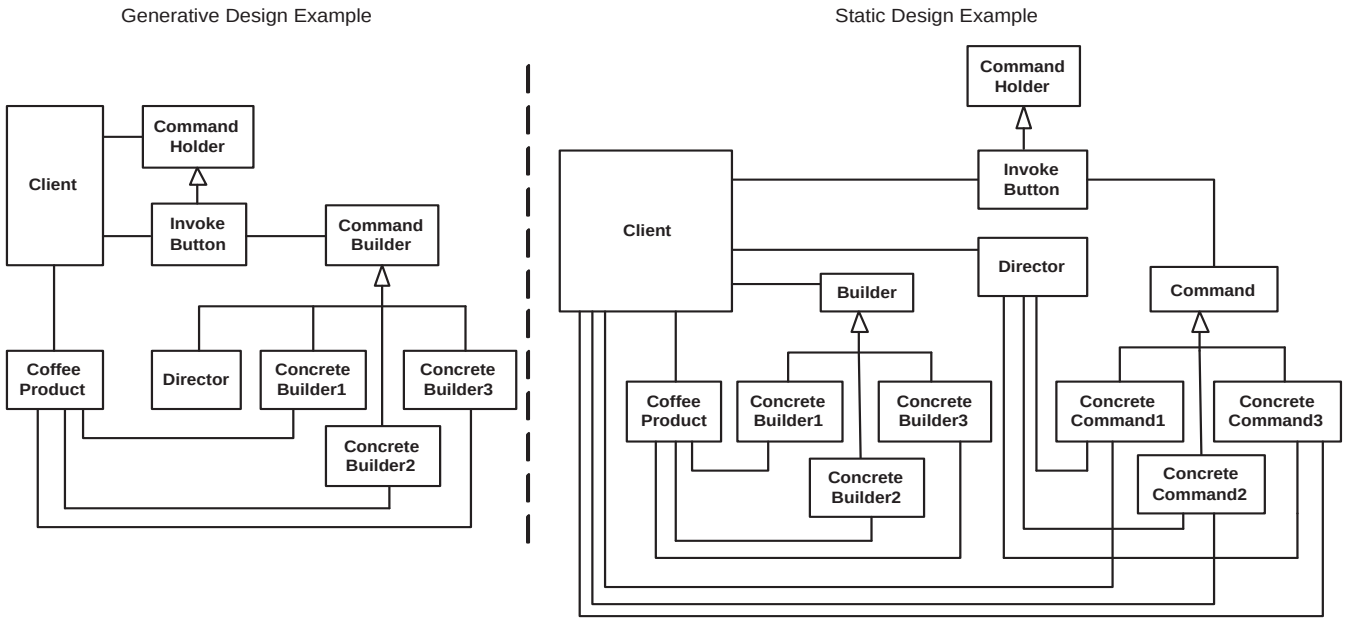

Figure H.2: Generative vs. Static – Command and Builder

Figure H.2 above provides a class diagram for the comparative examples of the command and builder patterns used in a generative and static pattern environment. Although the generative example is being considered as a combination of the two patterns, this is incorrect. In this example, the builder pattern is actually using the command pattern, as defined by the term Pattern X uses Pattern Y in its solution, described in Chapter Four. In stricter terms the builder pattern is only using the InvokeButton class from the command pattern. The evidence for this comes from the CommandBuilder interface, which only defines methods that are applicable to the ConcreteBuilder classes. As such there is no method in the ConcreteBuilder classes that could be considered as being an Execute method that would be applicable to a command pattern.

Also note that in this instance of the builder pattern being used in a generative environment, the director class is a subclass of the interface and not the product, as in the previous example. In the previous example, the product component had to be a leaf component to the composite so it could be added directly to the composite object.

In order for the two patterns to work together in the static environment, the ConcreteCommand components take as a parameter a Director:

```
Builder latteBuilder = new LatteConcreteBuilder();
```

CoffeeDirector latteDirector = new CoffeeDirector(latteBuilder);

LatteCommand latteCommand = new LatteCommand(latteDirector);

Now that the ConcreteCommand has an instance of a Director, a call to the Execute method in the ConcreteCommand will implement the Construct method that will build the Product in the builder pattern – Execute(){latteDirector.constructCoffee();}.

Table H.3 shows the overall results of the metrics that were produced from the generative and static examples of the command and builder patterns described on the previous page.

Metric	Generative Patterns	Static Patterns	Difference (%)
CBO	18	22	+
CC	16	16	/
LCOM	88	88	/
LOC	342	375	+8.8%
RFC	11	11	/
WMPC	11	11	/
NOC	9	13	+
EXE SIZE	13.7	16.2	+15.4%

Table H.3: General statistics for the Generative and Static versions of Command and Builder

From looking at Figure H.2, one might expect that the metrics results for the generative example would be considerably better than those of the static example given the increased number of classes in the static example. However, the results for this experiment were not as expected. The general statistics in Table H.3 indicate that the generative pattern will require less testing and maintenance in respect of the CBO metric only; all other metrics, other than LCOM and EXE SIZE, are of equal value.

As can be seen in Table H.4 the higher value for the CBO metrics comes from the client of the static example, which has to communicate with two interface components instead of just one interface component in the generative example. The equal values in the RFC and WMPC metrics seen in Table H.3 come from the builder classes in both the generative and static examples: namely the CoffeeProduct for the WMPC metric and the ConcreteBuilder classes for the RFC metric. Therefore in respect of the general values the generative example is neither more nor less complex than the static example. However, the general values are not taking into account the command pattern components that do not play a part in the generative pattern example. As such, there is an overhead in terms of the six

attributes itemised in section 7.2, which have to be taken into account in comparing the examples.

Taking the collective values of the ConcreteCommand classes and the Command interface of the static example into account, the static example is certainly more complex than the generative example. Add to this the reduction in the number of lines of code and the size of the executable file, the generative example comes across as an improvement on the static example.

Class	**CBO**		**CC**		**LCOM**		**LOC**		**RFC**		**WMPC**	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Client	18	22	16	16			115	117	2	2	2	2
	+		/				+		/		/	
Button	7	8	16	16			39	40	10	12	16	16
Handler	+		/				+		+		/	

Class	**CBO**		**CC**		**LCOM**		**LOC**		**RFC**		**WMPC**	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Command	1	*	1	*		*	10	*	6	*	6	*
Builder	*		*		*		*		*		*	
Command	*	0	*	1	*		*	4	*	1	*	1
	*		*		*		*		*		*	
Builder	*	1	*	1	*		*	10	*	6	*	6
	*		*		*		*		*		*	
Concrete	*	1	*	1	*		*	9	*	2	*	2
Commands	*		*		*		*		*		*	

Table H.4: Individual statistics for the Generative and Static versions of Command and Builder

Like previous examples, the individual class statistics for the like-for-like components in the examples are identical throughout all metric categories, therefore they are not included in Table H.4. In this example this equates to the CommandHolder interface, the ConcreteBuilder components, the CoffeeProduct, and the CoffeeDirector class. Again, like the previous examples the reason for this is modularity, in that each corresponding component provides identical functionality. The only exception in like-for-like components is the client, which is communicating with two separate interfaces.

H.3 A Simple Case Study using Composite and Command

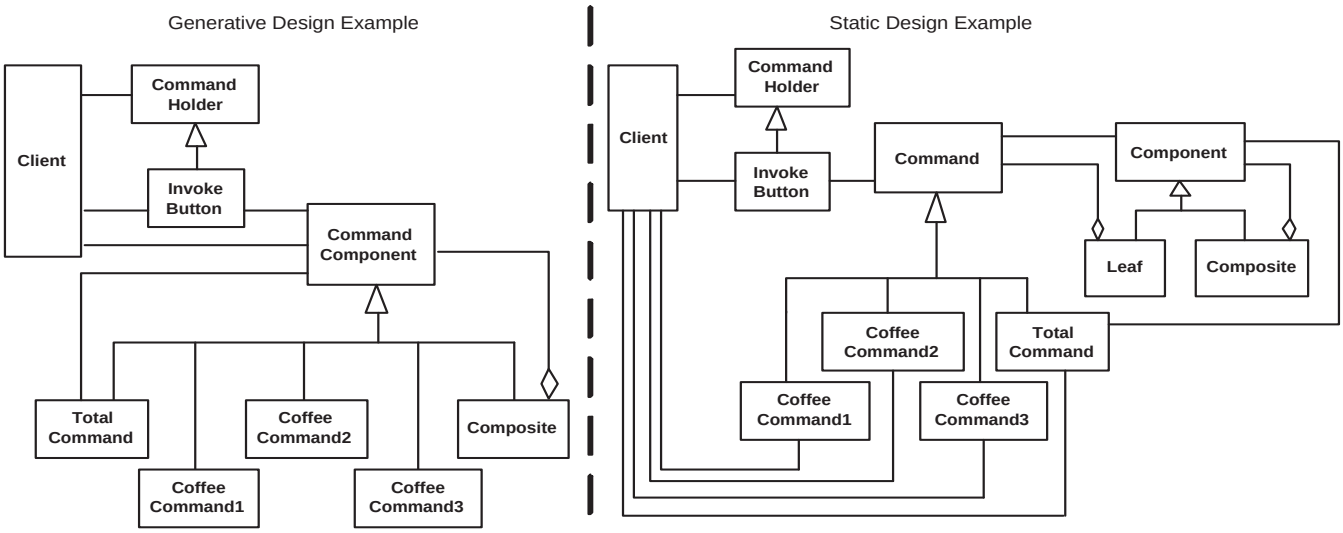

Figure H.3: Generative vs. Static – Command and Composite

Figure H.3 above provides a class diagram for the comparative examples of the composite and command patterns used in a generative and static pattern environment. As can be seen from the diagram, the generative example on the left has an interface (CommandComponent) that is combined from the two interface components that are used in the static example on the right. The four sub-components of the Command interface class from the static pattern example are now leaf components to the Composite class in the generative pattern example.

In order for the two patterns to work together in the static environment, an object is created from the ConcreteCommand components and added to a collection object in the Leaf component of the composite pattern. As such, multiple command objects can be added to one or more Leaf components and one or more Leaf components can be added to a Composite component. Composite components can be added to other Composite components as is intended with a composite pattern.

In the generative example, because any ConcreteCommand object that is created is now a leaf component to the Composite component, it can be added directly to a Composite object.

Table H.5 shows the overall results of the metrics that were produced from the generative and static examples of the composite and builder patterns described above.

Metric	Generative Patterns	Static Patterns	Difference (%)
CBO	18	20	+
CC	2	2	/
LCOM	90	90	/
LOC	249	294	+15.3%
RFC	10	10	/
WMPC	9	8	–
NOC	9	11	+
EXE SIZE	13.2	14.7	+10.2%

Table H.5: General statistics for the Generative and Static versions of Command and Composite

The composite and command examples above, are very similar to that of the composite and builder examples. Like the composite and builder example the statistics in Table H.5 indicate that the generative pattern will require less testing and maintenance in respect of the CBO metric but in this instance are quite even in respect of the RFC metric. Like all previous examples, the higher value for the CBO metrics comes from the client of the static example, which has to communicate with two interface components instead of just one interface component in the generative example. The value of the RFC metric in both pattern examples comes from the Composite component. Although the CommandComponent interface has to define different sets of methods to support the Composite class and the ConcreteCommand classes, the RFC value is less than that of the Composite.

Although there is less coupling in the generative example, as confirmed by the CBO metric, there is a higher degree of complexity. The higher value in the WMPC metric confirms that the CommandComponent class is more complex than other components in the static example and will therefore require more testing, and if required, more maintenance.

However, the two separate interfaces (Command and Component) in the static example have collective values higher than the CommandComponent interface of the generative example. In addition, the static example will require some additional testing for the Leaf component.

As in previous examples, the generative example has more points in favour than the static example. This takes into account the reduction in the number of lines of code, the number of classes and the size of the executable file, which are in favour of the generative example.

Class	CBO		CC		LCOM		LOC		RFC		WMPC	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Client	18	20	1	1			84	85	2	2	2	2
	+		/				+		/		/	

Class	CBO		CC		LCOM		LOC		RFC		WMPC	
	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP	GP	SP
Command	0	*	1	*		*	13	*	9	*	9	*
Component	*		*		*		*		*		*	
Component	*	1	*	1	*		*	12	*	5	*	5
	*		*		*		*		*		*	
Command	*	0	*	1	*		*	10	*	7	*	7
	*		*		*		*		*		*	
Leaf	*	4	*	2	*	0	*	28	*	8	*	5
	*		*		*		*		*		*	

Table H.6: Individual statistics for the Generative and Static versions of Command and Composite

Like in previous examples, the individual class statistics for the like-for-like components in the examples are identical throughout all metric categories, therefore they are not included in Table H.6 above. In this example this equates to the ConcreteCommand components, the InvokeButton, the CommandHolder interface and the Composite class. Again, like the previous examples the reason for this is modularity, in that each corresponding component provides identical functionality. The only exception in like-for-like components is the client. For the client there is a minor difference in that it communicates with two separate interfaces.

Appendix I

AN EXAMPLE DESIGN PATTERN

I.1 Facade (Based on Gamma[45])

Name

Facade

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation[45, 48, 102]

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

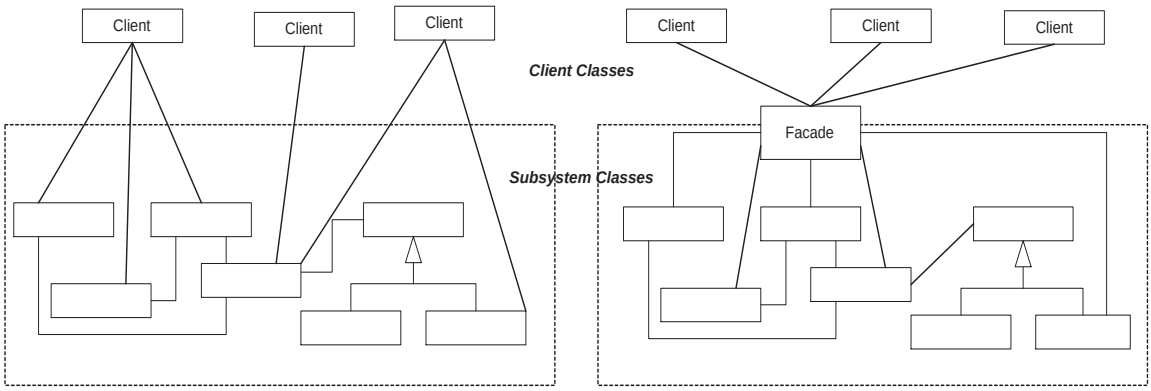

Figure I.1: Facade as an Interface

Dividing a system into several subsystems helps deal with complex systems and provides an opportunity to partition the work. Dividing a system into a number of specialized classes is a good object oriented design practice. However, having a large number of classes in a system can be a drawback as well.

Clients using that system have to deal with more objects thereby increasing complexity. The Facade pattern provides a way to shield clients of a set of classes from the complexity of using those classes. The way it does this is to provide an additional reusable object that hides most of the complexity of

working with the other classes from client classes.

Applicability

Use the Facade pattern when:

- You want to provide a simple interface to a complex subsystem. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customisation will need to look beyond the facade.

- There are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.

- You want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between subsystems by making them communicate with each other solely through their facades.

Structure[48]

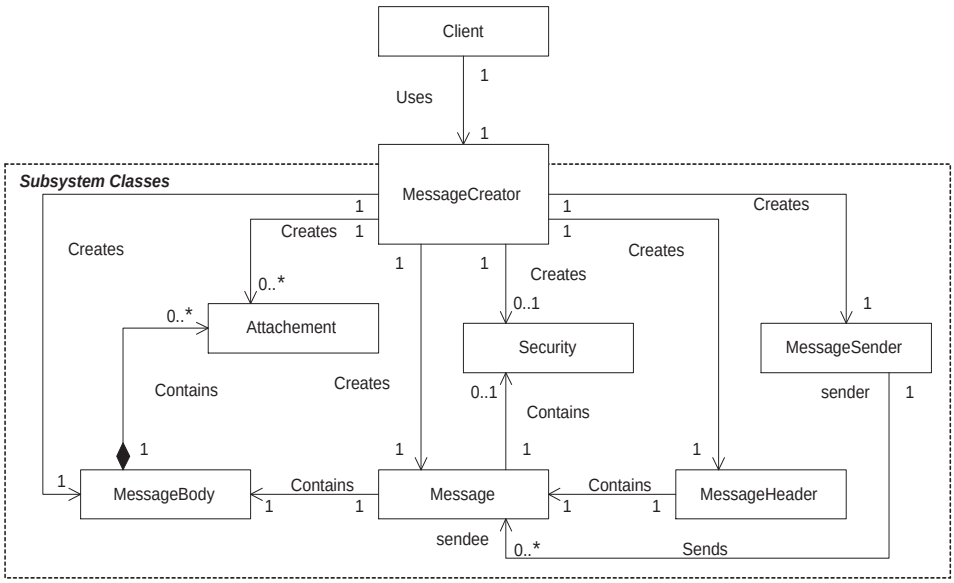

Figure I.2: Message Creator as Facade

Participants

- Facade
  - Knows which subsystem classes are responsible for a request.
  - Delegates client requests to appropriate subsystem objects.
- Subsystem Classes
  - Implement subsystem functionality.
  - Handle work assigned by the Facade object.
  - Have no knowledge of the facade.

Collaborations

- Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- Clients that use the facade don't have to access its subsystem objects directly.

Consequences

The Facade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
2. It promotes weak coupling between the subsystem and its clients. Week coupling lets you vary the components of the subsystem without affecting its clients. Facades help layer a system and the dependencies between objects. They can eliminate complex or circular dependencies. Reducing dependencies with Facade can limit the recompilation needed for a small change in an important subsystem.
3. It doesn't prevent applications from using subsystem classes directly if they need to. Thus you can choose between ease of use and generality.

Implementation

Consider the following issues when implementing a facade:

1. *Reducing client-subsystem coupling.* The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.

   An alternative to subclassing is to configure a Facade object with different subsystem objects. To customize the facade, simply replace one or more of its subsystem objects.

2. *Public versus private subsystem classes.* A subsystem is analogous to a class in that a class encapsulates state and operations, while a subsystem encapsulates classes. It is useful to think of the public and private interface of a class. In the same way we can think of the public and private interfaces of a subsystem.

   The public interface of a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders. The Facade class is part of the public interface, but it is not the only part. Other subsystem classes are usually public as well.

Sample Code[48]

The following code represents `MessageCreator` as the Facade class shown in the class diagram in Figure I.2. Instances of the `MessageCreator` class are used to create and send e-mail messages. It is shown here as a typical example of a facade class.

```
public class MessageCreator
{
    public final static int MIME = 1;
    public final static int MAPI = 2;
    private Hashtable headerFields = new Hashtable();
    private RichText messageBody;
    private Vector attachments = new Vector();
    private boolean signMessage;

    public MessageCreator(String to, String from, String subject)
    {
        this(to, from, subject, inferMessageType(to));
    }
```

```
    public MessageCreator(String to, String from, String subject, int type)
    {
        headerFields.put("to", to);
        headerFields.put("from", from);
        headerFields.put("subject", subject);
    }

    public void setMessageBody(String messageBody)
    {
        setMessageBody(new RichTextString(messageBody));
    }

    public void setMessageBody(RichText messageBody)
    {
        this.messageBody = messageBody;
    }

    public void addAttachment(Object attachment)
    {
        attachments.addElement(attachment);
    }

    public void setSignMessage(boolean signFlag)
    {
        signMessage = signFlag;
    }

    public void setHeaderField(String name, String value)
    {
        headerFields.put(name.toLowerCase(), value);
    }

    public void send(){}

    private static int inferMessageType(String address)
    {
        int type = 0;
        return type;
    }

    private Security createSecurity()
    {
        Security s = null;
        return s;
    }

    public void createMessageSender(Message msg){}
}
```

Known Uses

The `MessageCreator` example in the Sample Code section is a typical example of using facade to create and send email.

Related Patterns

Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-dependent way. Abstract Factory can also be used as an alternative to hide platform-specific classes.

Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator's purpose is to abstract arbitrary communication between colleague objects. It often provides centralized functionality that does not belong to any of them. Mediator's colleagues are aware of and communicate with mediator instead of one another. In contrast a facade merely abstracts the interface to subsystem objects to make them easier to use; it does not define new functionality and subsystem classes do not know about it.

Usually only one Facade object is required. Thus, Facade objects are often Singletons.