Separation of Concerns: Techniques, Issues and Implications

Aseem Daga, Sergio de Cesare and Mark Lycett

Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex UB8 3PH, United Kingdom

ABSTRACT

System decomposition is one of the fundamental abstraction principles of software engineering. One way in which decomposition can be achieved and managed is through a concern-based approach. With reference to software, systems concerns can be viewed as distinct system aspects or features (e.g., functional and non-functional concerns). Concern-based decomposition has become central to software development because of the benefits that such an approach can potentially provide. In order to achieve such benefits, it is necessary to be able to represent and manage the different concerns separately—known as *separation of concerns*. Such separation is difficult to obtain given that software systems tend to encapsulate multiple dimensions of concerns that cut across functional requirements. One solution regularly applied is to use techniques that separate concerns along different dimensions and then integrate them with a base dimension. To apply this scenario consistently across the whole development process, the various techniques must be flexible and adaptive at both conceptual and implementation levels. This paper argues that these techniques lack in flexibility because the underlying principles that define them are not yet fully developed. In order to demonstrate this limitation, the paper draws on the strengths and weaknesses of some of the major techniques for separation.

Reprint requests to: Sergio de Cesare, Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex UB8 3PH, U.K.
e-mail: {Sergio.deCesare, Aseem.Daga, Mark.Lycett}@brunel.ac.uk

Such techniques are based on a set of criteria that underlie important aspects of the separation principle. A critical analysis based on the criteria will provide an indication of the areas in which specific techniques have limited ability. Based on the outcomes, implications of the study and future work are presented.

KEYWORDS

multidimensional separation of concerns, software development, comparative analysis, aspect-oriented programming, composition filters, subject-oriented programming

1. INTRODUCTION

The increasing complexity in the requirements of software applications has considerably augmented the variety of concerns that the designer needs to deal with while developing applications (Hursch & Lopes, 1995). Concerns are domains that are used as decomposition criteria for organizing software into manageable and logical parts (Tarr et al., 2001). Concerns represent areas of interest within a software application based on some system concept, goal, or purpose. For example, concerns can be distinguished based on *features* (such as user needs), *aspects* (such as synchronization, scheduling, and distribution), or *roles, viewpoints* and so on (Turner et al., 1998, Kiczales, 1997). Concerns can reflect any significant domain of software development, including all levels of the development life cycle (e.g., business modeling, analysis, design, and implementation).

As the number of concerns that are recognizable in software-systems development has increased, the task of classifying such concerns along a limited number of specific non-overlapping dimensions has become greatly more difficult. The reason for clearly and distinctly separating software concerns is associated with the need to decompose systems into manageable units in order to achieve benefits such as ease of change, more localized maintenance intervention and reuse. This problem is known as *Separation of Concerns* (SoC). Many researchers in recognition of this problem have defined new dimensions, which in principle are claimed to provide a solution for a clear-cut SoC. These dimensions, along with their associated design and implementation techniques, include adaptive programming, aspect-oriented programming (AOP), composition filters, role modeling, subject-oriented programming (SOP), and the multi-dimensional approach. These techniques have emerged as enhancements to object-oriented (OO) design and programming by providing SoC along newly defined dimensions. (Aksit et al., 1992; Anderson & Reenskaug, 1992; Baniassad & Murphy, 1998; Batory et al., 2000; Czarneki & Eisenecker, 2000; D'Souza & Wills, 1998). The SoC paradigm helps to separate concerns at all levels within the software life cycle. For example, it allows the developer to provide a clean separation between the basic functional and non-functional properties of the system (Kiczales et al., 1997). Moreover, separation helps to bring the different dimensions of concerns under a single umbrella, giving them a generalized and more comprehensible picture.

Despite a good deal of research, the question of how to achieve a complete and accurate separation of concerns remains an issue within the software development community. The issue in a large part relates to the limited ability of the techniques in separating all forms of concerns across the whole development process. All modern software formalisms provide separation to an extent, but a complete and consistent separation of concerns remains a non-existent reality. When separating concerns, there are issues of identification, decomposition, and composition. The problem is not just about separating concerns; the process of weaving and interrelating concerns is equally important. Problems arise when concerns overlap. We argue here that although the above issues are relevant, they are not cleanly specified within the different separation techniques. The manner in which concerns are implemented is not consistent with what is specified at the design stage because of very limited traceability between the design and implementation aspects of different techniques. This paper compares some of the major SoC techniques. The comparison is based on a set of well-recognizable criteria for achieving a complete SoC. In such a manner, the limitations of current techniques can be analyzed in order to formulate ideas as to resolving those issues that hinder the progress of the separation principle.

This paper is structured as follows. Section 2 defines the concept of separation of concerns in detail. Section 3 highlights the major criteria on which to judge the different separation techniques. Section 4 elucidates and critically analyzes the major separation techniques. Finally, the implications and conclusions of this work are presented.

2. SEPARATION OF CONCERNS

Decomposition is a fundamental principle of software engineering. Decomposition involves breaking down the system into smaller and more manageable parts, which at a requirements level generally corresponds to subdividing the overall problem into sub-problems that can be addressed independently. Concerns are abstractions of a given problem (Tekinerdogan & Aksit, 2000). They are a set of coherent issues based around a problem that has to be resolved. Concerns are generic abstractions but are highly relative to the problem. As a result, what might be considered a concern for a given problem might not be relevant at all for another problem. The definition of concern, however, does not provide a clear distinction of the granularity at which concerns should be viewed. The distinction is based more on the practical utility or the view taken about the concern. As a result, depending on the view taken of the definition, a concern can be either a small task like "Print a document" or a complete software artefact like requirements understanding. A concern can be any element that is relevant in the development of a system.

Concerns as they stand can be separated into basic concerns and special purpose concerns (Hursch & Lopes, 1995; Aldrich, 2000). Basic concerns within software are usually responsible for providing basic functionality and services to the client. The client of the system can be either the user, another program, or some other component of the system (Aldrich, 2000). The primary objective of basic concerns is to answer the functional needs of the system in a traditional manner and usually do not depend on any other aspect of the system. The design style they support is to break the software down into components that can be called upon to perform some function. These components specify what is really important to the system and work accordingly. Classes and objects exist as a prime example for basic concerns. Classes and objects fit within the majority of existing programming languages and provide the perfect platform for decomposing the system into small parts, in which each part is assigned to perform some function. Both data and methods are defined within the classes, thus encapsulating (or hiding) the details from the outside world. Furthermore, inheritance and overriding mechanisms allow classes to modify and extend certain features without changing their inherent organization. Other examples can be application features and services, sub-routines, procedures, and functions.

Special purpose concerns are generally attributed to serve or manage the basic concerns within the system or fulfilling some special requirements for the system. Special concerns are usually performance related and accordingly play an auxiliary role to the basic concerns. Being crosscutting in nature, special purpose concerns do not affect any single module or domain but rather spread throughout the system functionality. Examples of special purpose concerns within the system can be concurrency or synchronization, persistency, real-time constraints, distribution, location control, or configuration, failure recovery, and replication.

Over the years, concerns have often been viewed in terms of software implementation. With all programming languages supporting decomposition along a single dimension, concerns are decomposed based around that predefined dimension. *Dimension* can defined as a combination of the decomposition unit and the means to assemble the units to make it a running program. For example, traditional systems based on the object-oriented paradigm often modularize the concerns based on a single dimension like class (Constantinides et al., 2000). This form of decomposition based on a single dimension can be acceptable when the problem can be solved through simple interfaces. With concurrent programming based on component tech-nology, interactions no longer localize within a single modular unit, but rather tend to cut across the whole of system functionality. The reason for the latter being that component interaction in itself is based on a number of properties like synchronization, scheduling, and distribution, which affect the perfor-mance of the whole system; defining these along a single dimension is quite a challenging task. This manner of looking at concerns through a single dimension has been referred to as the "*the tyranny of dominant decom-position*" within the literature (Ossher et al., 1999). This view has made it quite difficult for designers to manipulate these properties under a single dimension and has resulted in many significant problems for the development and deployment of software (Kiczales, et. al., 1997; Bergmans & Aksit, 2000, Ossher & Tarr, 2000).

As a direct consequence of the above limitation, it has become important to look at concerns from different dimensions or angles. Recently, several new techniques have emerged for organizing and composing concerns along different dimensions. These techniques are based on the premise that programming all concerns within a single block of code is often the reason for increased complexity and tangling problems. By abstracting and separating the concerns, programming then becomes substantially less complex and code can be effectively reused. Such techniques are assumed to directly support decomposition along different dimensions while remaining orthogonal to the basic concerns. The value of these techniques, however, will clearly depend on overcoming two major hurdles. Firstly, with SoC techniques still at its early stages, the knowledge of key concerns and how they should be separated is very limited. Secondly, there exists little or no high-level scenarios or design mechanisms whereby concerns can be understood both individually and in relation to other aspects of the system. As a result, concerns are often just directly implemented without understanding and knowing what effects it may have on the overall system.

3. COMPARATIVE FRAMEWORK

Comparative criteria have been identified to effectively compare SoC techniques. These criteria serve a dual purpose: (1) to describe the main aspects of each technique; and (2) to form the basis of evaluation and comparison of the techniques analyzed. Table 1 provides a list highlighting the selected criteria along with the reasons as to why each one is of value in describing and evaluating SoC techniques.

TABLE 1

Criteria selected and t heir relevance to SoC

Criteria	Relevance to the Separation of Concern principle
Understanding of the technique and relevant description	Identification of Concern; Understanding of the sub-parts of a concern
Design Knowledge	Encapsulation and Decomposition of Concern; Design Capability
Composition of Concern	Integration of Concerns; Composition Rules
Language Support	Provides knowledge about implementation abilities of the technique; Highlighting the relationship between design and implementation
Overlapping Concern	Establishes the relationship between intersecting concern

The criteria are divided into five categories: (1) understanding the technique, (2) design knowledge, (3) composition of concerns, (4) language support, and (5) overlapping concerns. The themes of the criteria are predominantly project based, highlighting the relationship among the developers, the software artefact(s), and the methodological underpinnings. The following subsections detail each criterion adopted for the comparative analysis.

3.1 Understanding the Technique

Comparing the relevance of any technique to its cause requires a thorough understanding of the technique itself, requiring the technique to identify which dimension(s) of concerns it has to target and to identify how the units are to be populated within the concerns. This identification makes explicit all the concerns of interest along with the detailed understanding of the requirements of the various concerns.

3.2 Design Knowledge

Design knowledge of the system has often been referred to as one of the most important and beneficial activities within the system lifecycle (Booch, 1994; Coleman, 1993; Daniels, 1994). With design being the mapping ground between requirements and implementation, a clean and unambiguous design model can provide benefits ranging from early assessment of requirements to a clear definition of the coding principles and management of issues like traceability, complexity, and evolution (Rumbaugh et al., 1990). In addition, two major benefits are noteworthy. A clear expression of the design knowledge provides a better encapsulation of the major concerns. Secondly, as the goal is to develop an entire system, decomposition usually implies recomposition: a model that is decomposed makes the integration of those decomposed parts easy and more understandable, resulting in a system that is operational. Design knowledge is imperative for any system, more so for large and evolving systems (Tarr et al., 1999).

3.3 Composition of Concerns

Composition of concerns looks at what kind of rules the technique employs to compose different concerns. The SoC principle would be less beneficial without the integration of decomposed parts as noted by Jackson (1990), who asserts the importance by saying "*having divided to conquer, we must reunite to rule*". Integration mechanisms describe how these abstractions can be combined with each other and with other functionality of the system. For example, how a crosscutting concern can be weaved back with the functional aspects of the system or with other crosscutting concerns.

Due to the cross cutting nature shown by the majority of the concerns, the integration process is as relevant and important as identifying the concerns themselves. Whereas the thrust on decomposition has been evident while designing systems (Gruenbacher et al., 2000), addressing the composability or integration mechanism remains an issue with a majority of the techniques. Many questions are being raised as to how the integration mechanism should be carried out. How should the basic and special concerns be woven together? Should the integration be carried out at the source code or the object code level? What rules and principles should be applied while carrying out such procedures? What design framework should it follow for the integration process? These and a host of other issues have made composition of concerns an important area of focus, and a number of publications have started to address the composability problems within systems (Mullet et al. 1995; Nierstrasz & Tsichritzis, 1995).

3.4 Language Support

The construction of complex, evolving software systems requires clear and unambiguous language constructs to manage the gap between the high-level design and its low-level implementation. The above three criteria provide the identification, encapsulation, and integration of various concerns at a high level. A fluent relationship between high-level and low-level repre-sentations provides traceability and alignment between requirements, design, and implementation. This relationship within traditional systems, however, is very rarely enforced. The essence of the problem with OO design techniques is that they traditionally align well with OO code and not with requirements. This makes the alignment between requirements and code very poor, which leaves room for the designers to diverge from the requirements, thus differentiating the designed system from the actual implemented one (Ossher et al., 1999).

3.5 Overlapping Concern

One of the primary aims of separation of concern is to identify and modularize those parts of a system that can be based around some purpose, goal, or scope. This goal involves identifying different dimensions of concerns and developing solutions orthogonally to avoid the problems of code tangling and scattering of code. This approach, however, has effects when composing overlapping concerns within the systems. Most concerns within systems just do not exist in isolation; they are interrelated in different ways. For example, a piece of functionality might be associated with two or more concerns. The presence of overlapping concerns raises many questions within the minds of the designer about how they should be eventually tackled. Should multiple copies of the code be executed if all concerns are composed into a system? Does overlapping indicate a relationship between concerns, such that one concern must be included if another concern is included? Should the code be executed only if all concerns are composed into the system? As a result of these questions and many more, different techniques need to counter and provide a suitable answer to overlapping concerns.

4. CRITICAL ANALYSIS OF MAJOR SOC TECHNIQUES

Having established the criteria on which to base our study, this section critically examines four major techniques that support SoC: Subject-Oriented Programming (SOP), Aspect-Oriented Programming (AOP), Composition Filters (CF) and Multi-Dimensional Separation of Concerns (MDSOC). One reason for having different techniques is that concerns being dependent on the nature of the software are context–sensitive (Ossher, 2001). They depend on the type of the software being developed, the stage of the development cycle, the users, and the developers of the application. As a result, many dimensions of concerns based on the relevance of the situation and requirements will be needed. Although each technique defined below achieves SoC, the manner in which they achieve it is not identical. The techniques differ in their application of the separation principle. One major attribute of these techniques is that they are very recent and are in constant evolution. The following subsections critically examined each technique by applying the framework presented in Sec. 3. Table 2 provides a matrix of the different techniques and the manner in which the techniques provide solutions based on the criteria highlighted.

4.1 Subject Oriented Programming (SOP)

Subject Oriented Programming evolved as a program-composition technology to support building OO systems as compositions of subjects to improve

TABLE 2

Comparative Analysis of SoC Techniques

Criteria	Technique			
	Subject Oriented Programming	**Aspect Oriented Programming**	**Compositional Filters**	**MDSOC**
Understanding of the technique and the relevant description	High Level Understanding quite fluent	Understanding quite low and abstract	Mere extension of the OO and AOP techniques	Hyperspaces. Separation achievable in any dimension.
Design Knowledge	Separation in form of design subjects	Separation in form of aspects	Separation in form of filters. Clear distinction between interface and implementation.	Separation in the form of a hyperslice.
Composition of Concern	Using composition rules based on needs and requirements	Join Point introduced between base object and aspects	Filters are composed and attached as objects to the base class.	Hyper-module. Uses composition rules.
Language Support	Primarily a design technique; Only support available in some languages	Aspect/J - A Java based extension. Aspects can be composed into other languages as well.	Language independent. Any language can be used.	Hyper/J - A Java based extension.
Overlapping Concern	Provided in abstract terms but not clearly explained	Not Available	Available, though highlighted only theoretically.	Yes

the misalignment gap between requirements and code. A subject is a collection of classes or class fragments whose hierarchy models the domain in its own, subjective way (Harrison & Ossher, 1993). Subject Oriented Programming permits standard design models to be decomposed into smaller, potentially overlapping, units, called design subjects. Unlike the units of modularity present in OO design languages, design subjects are chosen to align with the requirements, thus minimizing the traceability problem associated with OO system. Other benefits often associated with SOP-designed systems are facilitating unplanned extensions, multi-site development, and modularizing the system in terms of different dimensions like features, requirements, etc.

Each design subject encapsulates a single, coherent piece of functionality and defines only the states and behaviors that are intrinsic to the object. The reasoning is that extrinsic elements are outside the scope of relevance of the object and thus should not form a part of the object definition. This view, however, has limitations as sometimes the objects have to be extended to perform specialized behavior (Robillard, 2000). The individual design subjects are then decomposed into code subjects using standard OO design and coding techniques. This approach provides very good traceability through requirements, design, and code as each corresponds directly, and within a single subject, to standard object-oriented design or code.

Individual code subjects then can be combined to form cooperating groups called compositions. The composition is guided by composition rules that specify in detail how the subjects are to be combined. Any form of mismatch and overlapping of concerns is resolved within the composition process. Starting from the low-level definition pertaining to a single subject, general-level composition rules are defined that are applicable to the whole system. The code subjects are then composed and implemented to produce the entire system. Any new requirements or changes to existing requirements results in the creation of a new design subject and thus follows the same procedure to enhance the existing design. Although this mechanism allows for a greater flexibility, difficulty with this implementation relates to the composition of the units that belong to different dimensions, which then have different rules for composition.

The implementation of code subjects is largely done using standard OO languages. Subject Oriented Programming primarily is a design technique and as a result is largely language independent. This independence, however, has limited the applicability of SOP as large-scale implementation often needs some form of tool support. Support for SOP is available in C++ as an extension of the IBM VisualAge for C++ Version 4 (VAC++) compiler and environment.

4.2 Aspect-Oriented Programming (AOP)

Aspect-oriented programming (AOP) is a programming technique that has been widely claimed to make it easier to reason about, to develop, and to maintain certain types of application code (Kiczales et al., 1997). Aspect-oriented programming is based on the concept of aspects, which enables the designer to talk about the global properties that are spread over the application (Pawlak et al., 2000). Kiczales et al. (1997) define aspects as the properties that affect the performance and semantics of the components of the system in a systematic way. Due to this global nature, aspects can be thought of as horizontal slices that cut across various components within a system. This makes it possible to program the concerns in a modular way and achieve the usual benefits of separation of concerns (Kiczales et al., 2001). Typical aspects illustrate many non-functional attributes of the system like user interfaces, collaborative work, distribution, persistency, concurrency, and other emergent concerns (Kiczales & Lopes, 1998). Aspect definition by contrast remains a rather vaguely used term, however. Although aspects are argued to highlight all forms of concern, the majority of concerns that are highlighted are non-functional in nature (Rashid, 2001).

Aspect-oriented programming highlights and captures the interaction between components and aspects within the source code (Kiczales et al., 1997). It distinguishes the concept of classes, which encapsulates the functional properties of the system from aspects, which highlight the non-functional requirements. Once identified, the aspect is expressed within a separate piece of code along with its related data structures and behaviors. After coding the aspects, join points are identified to weave them back with the functional components. Join points are well defined points in the execution of the program like objects calls and methods. To compose aspects, it is necessary to distribute the elements of the aspect where they are needed.

Aspects specify data structures and operations or parts of operations that are specific to a concern, together with instructions on how to relate these elements to a base program. This, however, also introduces the problem of aspect composability. Although AOP evolved as a programming technique to overcome the tyranny of dominant decomposition, its own decomposition of concern (aspects) limits this view. Multiple decomposition of the same software in different dimensions is not allowed. As a consequence, AOP's capability of dealing with overlapping concerns is limited.

Aspect-oriented programming provides explicit language support to highlight the different crosscutting properties of the components. Most work on aspects is conducted through AspectJ, an aspect-oriented extension to Java that can be used within different environments (Kiczales et al., 1997). AspectJ allows the programmer to introduce concerns in the form of aspects, along with attributes and methods, just like the classes in Java. These aspects once defined are then woven back with the base program using the aspect weaver to compose the application. The weaver however only provides either static or dynamic weaving of aspects. While static weaving is viewed to provide better performance, dynamic weaving facilitates incremental weaving and makes debugging easier, which makes both forms necessary (Bollert, 1999; Constantinides et al., 2000).

The usage of AOP as a separation technique has been amply verified in cases for which the scope of the techniques has been well defined and the tasks are clearly highlighted (Walker et al., 1999). There are, however, limitations over using AOP in large projects, as they are still largely untested. The lack of generalization among the techniques hinders the application and does not specify what kinds of problem they are best suited to solve.

4.3 Composition Filters

The composition filter (CF) model evolved as an extension to the conventional OO model and more recently the AOP technique through the addition of object filters to incorporate complex crosscutting properties of the system (Aksit & Bergmans, 2001; Kiczles et al., 1997). The model essentially consists of one or more input and output filters that sits on top of the base program controlling the message flow. It provides modular and orthogonal extensions for handling various special concerns within the base program. The composition-filter model and its application has been applied to various cross-cutting concerns and other problems, such as inheritance and delegation (Aksit & Tripathi, 1988), coordinated behavior (Aksit et al., 1993), real-time specifications (Aksit et al., 1994), and reusable synchronization specifications (Bergmans, 1994).

The technique is primarily developed around the concept of filters, which in essence are objects supported through a filter class, definable on concern basis. The purpose of filters is to manage and affect message sends and receives. In particular, a filter specifies conditions for message acceptance or rejection, and determines the appropriate resulting action. The model follows the OO paradigm by having a separate interface and implementation. The interface part provides the object declaration and consists of methods, instance variables, conditions, internal and external objects, and one or more filter declarations.

The separation of the interface from the implementation makes the CF model language independent. This approach, however, also makes the design model rather fixed in terms of its capability, which makes the model quite inflexible in terms of any evolutionary needs of the system. The implementation part provides definition for the interface by explaining the conditions set on the methods for message sending and receiving. The system makes sure that a message is processed by the filters before the corresponding method is executed—once a message is received, it has to pass through a set of input filters, and before a message is sent, it has to pass through a set of output filters. The actions taken on acceptance or rejection of a message by a filter depend on the class of the filter. Separation is achieved by defining a filter class for each concern. For example, a RealFilter will be proposed to affect the real time aspects of incoming messages. Each filter in essence has to carry out responsibilities associated with that particular concern. Support for the CF model is available in the Sina language (Koopmans, 1995). Support has also been extended to Smalltalk (Dijk & Mordhorst, 1995) and C++ (Glandrup, 1995) languages. Although the CF model has been able to solve specific sets of concerns like synchronization and delegation, doubts remain regarding the ability of the model to modularize other forms of concerns. Moreover, the development of the CF model as a means to achieve orthogonal solutions has meant that it has limited ability when handling overlapping concerns. Defining the concerns independently would make it difficult for the model to compose different concerns simultaneously.

4.4 Multi-dimensional Separation of Concerns (MDSOC)

Modern software systems elicit many dimensions of concerns like features, roles, and data structures that are deemed important during the life-time of the system. Although existing separation techniques have managed to separate these concerns, such techniques still show traces of dominant decomposition and dependence on OO-based design mechanisms. Multi-dimensional separation of concerns (MDSOC) overcomes the limitation of dominant decomposition by allowing the designer to decompose the system into different and arbitrary forms of concerns without any concern acting as a dominating force (Ossher & Tarr, 2001). Multi-dimensional separation of concerns permits clean separation of multiple, possibly overlapping and interacting concerns simultaneously, with support for remodularization to encapsulate new concerns at any time. This realization promotes reuse, improves comprehension, reduces the impact of change, eases maintenance and evolution, improves traceability, and opens the door to system refactoring and re-engineering. Thus, it addresses some of the fundamental limitations in software engineering. Multidimensional separation of concerns introduces the concept of hyperspace to alleviate the problem of dominant decomposition (Ossher & Tarr, 2001). The concept is supported through hyperslices, which allow for the explicit identification and encapsulation of concerns, and on hypermodules, which provide integration or composition of concerns.

Hyperslices in essence are modules that allow the designer to decompose the system and encapsulate concerns in dimensions other than the dominant one. Each hyperslice represents a single concern and contains units that provide information about that particular concern. Hyperslices can overlap, thus supporting simultaneous decompositions along multiple dimensions. This allows a software implementation to be represented as a collection of hyperslices along as many dimensions as required. Once all the hyperslices are identified within the system, they must be integrated together to produce the complete artefacts or the complete system. A hypermodule is a set of hyperslices and clearly defined composition rules that dictate the integration policy between different units of hyperslices. They identify and guide as to how the units within the hyperslices are related and the manner in which they should be integrated. This hyperspace approach of MDSOC is supported by HyperJ, a tool that supports MDSOC for any Java based development. It provides the ability to identify concerns, specify modules in terms of those concerns, and synthesize systems and components by integrating those modules. It operates on standard Java class files without need of source and produces new class files to be used for execution.

The embryonic nature of the MDSOC however nullifies its immense potential. Apart from a few pilot projects, there exists no practical application of the technique to highlight its ability. Small-scale implementations or examples that explain the technique remain the only link between the technique and its set objectives. Future research into the practical ability of the technique is required. The technique needs to establish the management of concerns across different artefacts, if possible at a much higher level. It needs to address and clearly explain how the code will cope with the tangling of different methods when handling overlapping concerns along with reconciliation and interference issues. At the code level, it will have to see whether the application of the language has increased the complexity, and moreover, whether the language can express all forms of concerns. As the technique gets more exposure, the impact of MDSOC on development methodologies, software processes, and software architecture will decide the future of MDSOC as a quality separation technique.

4.5 Other Techniques

Many other techniques like Viewpoints (Nuseibeh et al., 1994), design patterns (Gamma et al., 1995), and frameworks also provide separation of concern. The viewpoint technique is based on requirements engineering and helps to encapsulate the designer's view of requirements artefacts and the requirement building process. Viewpoints have many strong points, such as more focus on design issues, as well as an orthogonal view of concerns. Viewpoints, however, are limited in terms of implementation, and the approach is primarily concerned with how the concerns are viewed. Similarly, design patterns are limited as they do not provide a very flexible decomposition of concerns. Moreover, some design patters can be implemented using SOP (Robillard, 2000).

5. IMPLICATIONS FOR THEORY AND PRACTICE

Both theoretical and practical contributions can be drawn from the comparative analysis of Sec. 4. The analysis provides the basis to identify the deficiencies of current SoC techniques as a whole, as well as directions for future research in this area.

5.1 Implications for Theory

5.1.1 *Better co-ordination of concern within business/software architecture(s).* The analysis has revealed that most techniques provide SoC in a way that is not very well integrated with the development of the system as a whole. Even though SoC techniques publicize designing the concerns in a modular fashion, a complete separation between the functional and non-functional aspects of the system is often unhealthy and unfeasible. This problem creates difficulty both in terms of overall understanding of the system and of composing the different parts of the system. A more practical manner in which SoC can be applied is to highlight the concerns architecturally at the conceptual and the implementation stage irrespective of the dimension to which they belong (Hursch & Lopes, 1995). This approach provides better management of concerns across requirements, design, and coding of the system. Highlighting concerns at the conceptual level provides a better way to manage them at the implementation stage. Separating the concerns at the conceptual stage helps the designer to identify the different concerns that have to be catered for in the system. Moreover, an abstract picture of the different concerns can be gathered along with the underlying relationships between different concerns. This will then allow techniques like MDSOC, which does cater for overlapping concerns, to design solutions, with a clear knowledge of what exactly is the concern being designed for but still maintaining a degree of independence.

5.1.2 *Better Management of Techniques.* The embryonic nature of the techniques have meant that although every step forward provides a better way of describing and implementing the technique, it often does not allow the technique to complete the whole circle. The analysis has clearly identified the strength and weaknesses of the different techniques. Overall analysis has revealed that only MDSOC provides a relative coverage across all the criteria specified. Although AOP shows a better coverage in terms of application and usability, it is severely hampered because it does not cater for overlapping concerns, and the aspect definitions do not allow decomposition along additional dimensions other than non-functional requirements. The SOP and the CF model have limited or no tool capability, which makes the usage of the technique not very viable. Both are dependent on support available from other languages. Moreover, the lack of practical usage casts doubts over the general applicability of these techniques.

The analysis clearly opens two avenues for researchers and academics. If the viability of the technique is the primary aim, then the focus should be to design and implement pilot projects keeping the strength and weaknesses of the techniques in mind. If the aim is to provide grounding for the techniques, then working on the weaknesses and limitations should be the primary focus area.

5.2 Implications for Practice

In terms of practical implications, the paper clearly highlights the limited applicability of SoC techniques as very little exists in terms of practical exposure for the techniques. Overall analysis has revealed that the understanding of the technique is relative to the nature of the work involved. The techniques are fluent where the scope is well defined. The usage of AOP and MDSOC as separation techniques has been amply verified in cases for which scope of the techniques has been well defined and the tasks are clearly highlighted (Walker et al., 1999). There are, however, limitations over using these techniques in large projects, as they are still largely untested. The lack of generalization among the techniques hinders the application and does not specify what kinds of problem they are best suited to solve. This paper has highlighted some of the important aspects of the techniques, which can be beneficial for the designer when using the techniques.

6. CONCLUSION

Separation of Concerns is one of the core principles of software engineering. Separation of Concerns aims at overcoming one of the limitations of traditional approaches by identifying and separating distinct features and aspects of the system, allowing developers to focus on individual concerns. To be effective, SoC techniques must be consistent throughout the whole development process. This consistency requires a thorough understanding of the various dimensions of concerns the technique is aiming to resolve, decomposing and composing the concerns by using predefined rules, and applying design techniques to portray their high-level understanding. Other requirements include tool support and the design and implementation solution they provide for overlapping concerns.

The paper has provided a comparative evaluation of four major SoC techniques. The broad conclusions that can be drawn from this analysis concern (1) the level of maturity of the techniques themselves, (2) lack of integration and coordination of such techniques within the overall development process, and (3) the ability of SoC techniques to represent throughout the system's life cycle different dimensions of concerns. As a consequence, research should be directed in each of the above areas. Firstly, SoC techniques must be applied across large industrial projects in order to test their scalability and understand from the data that emerges how the limitations described in this paper can be overcome. Secondly, SoC techniques must be integrated within full-fledged software development methodologies and at all levels in order to achieve full integration of such techniques with more consolidated development approaches like object-orientation. Finally, orthogonality between dimensions of concerns is likely to be achieved with the development of more sophisticated mechanisms for decomposing, composing and weaving different types of concerns.

REFERENCES

Aksit, M., Watika, K., Bosch, J., Bergmans, L. and Yonezawa, A. 1993. Abstracting object-interactions using composition-filters, in: *Object-based distributed processing, Lecture Notes in Computer Science,* edited by Guerraoui, R., Nierstrasz, O. and Riveill, M., Springer-Verlag, 152–184.

Aksit, M., Marcelloni, F., Tekinerdogan, B., Vuijst, C. and Bergmans, L. 1994. Designing software architectures as a specializations of knowledge domains, University of Twente, *Memoranda Informatica,* 95–44.

Aksit, M. and Tripathi, A. 1988. Data abstraction mechanisms in SINA/ST, *Proceedings of the OOPSLA '88, ACM SIGPLAN Notices*, **23,** 265–275.

Aldrich, J. 2000. Challenge problems for separation of concerns, *Proceedings of the OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, USA.

Anderson, E. and Reenskaug, T. 1992. System design by composing structures, *Proceedings of the European Conference on Object-Oriented Programming ECOOP*, Utrecht, the Netherlands, 133–152.

Batory, D., Johnson, C., MacDonald, B. and von Heeder, D. 2000. Achieving extensibility through product-lines and domain-specific languages: a case study, *Proceedings of ICSR 6*, Vienna, Austria, 117–136.

Bergmans, L. 1994. University of Twente, Enschede, The Netherlands.

Bergmans, L. and Aksit, M. 2000. Composing software from multiple concerns: a model and composition anomalies, *Proceedings of the ICSE '2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, Limerick, Ireland.

Bollert, K. 1998. On weaving aspects, *Proceedings of the Aspect Oriented Programming Workshop ECOOP '99*, Lisbon, Portugal.

Booch, G. 1994. *Object-oriented analysis and design with applications*, Menlo Park, California, USA, Benjamin-Cummings.

Coady, Y., Kiczales, G. and Feeley, M. 2000. Exploring an aspect-oriented approach, *Proceedings of the Advanced Separation of Concerns Workshop OOPSLA 2000*, Minneapolis, Minnesota, USA.

Constantinides, C.A., Bader, A., Eldrad, T.H., Netinant, P. and Fayed, M. 2000. Designing an aspect-oriented framework in an object-oriented environment, *Proceedings of the ACM Computing Surveys Symposium on Application Frameworks*, **32,** 41.

Czarnecki, K. and Eisenecker, U.W. 2000. Separating the configuration aspect to support architecture evolution, *Proceedings of the Workshop on Aspects and Dimensions of Concern at ECOOP'2000*, Cannes, France. [Position paper]

Daniels, J. and Cook, S. 1994. *Designing object systems: object-oriented modelling with syntropy*, London, UK, Prentice-Hall.

Dijk, W.V. and Mordhorst, J. 1995. University of Twente, Enschede, the Netherlands.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design patterns: elements of reusable object-oriented software*, Boston, Massachusetts, USA, Addison-Wesley.

Glandrup, M. 1995. University of Twente, Enschede, the Netherlands.

Harrison, W. and Ossher, H. 1993. Subject-oriented programming (a critique of pure objects), *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications OOPSLA '93*, Washington, DC, USA, ACM Press, 411–423.

Hursch, W. and Lopes, C. 1995. *Separation of concerns*, College of Computer Science, Northeastern University, Massachusetts, USA, 21. [Technical Report NU-CCS-95-03]

Jackson, M. 1990. Some complexities in computer-based systems and their implications for system development, *Proceedings of the International Conference on Computer Systems and Software Engineering EuroComp '90*, Tel-Aviv, Israel, IEEE Press, 344–351.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J. 1997. Aspect-oriented programming, *Proceedings of the European Conference on Object-Oriented Programming ECOOP*, Jyväskylä, Finland, Springer-Verlag, 220–242.

Koopmans, P. 1995. Department of Computer Science, University of Twente, Enschede, the Netherlands.

Lopes, C.V. and Kiczales, G. 1998. Recent developments in AspectJ. *Proceedings of theEuropean Conference on Object-Oriented Programming (ECOOP '98)*, Brussels, Belgium. [Position paper]

Mullet, P., Malenfant, J. and Cointe, P. 1995. Towards a methology for explicit composition of metaobjects, *Proceeding of the OOPSLA'95, ACM Sigplan Notices*, **30**, 316–330.

Murphy, G.C., Walker, R.J., Baniassad, E.L.A., Robillard, M.P., Lai, A. and Kersten, M.A. 2001. Does aspect-oriented programming work? *Communication of the ACM*, **44**, 75–77.

Nierstrasz, O. and Tsichritzis, D., editors. 1995. *Object-oriented software composition*, Hertfordshire, UK, Prentice Hall.

Nuseibeh, B., Kramer, J. and Finkelstein, A. 1994. A framework for expressing the relationships between multiple views in requirements specification, *Transactions on Software Engineering*, **20**, 760–773.

Ossher, H. and Tarr, P. 2000. Hyper/J: Multi-dimensional separation of concerns for Java, *Proceedings of ICSE*, Limerick, Ireland, 734–737.

Ossher, H. and Tarr, P. 2001. Using multidimensional separation of concerns to (re)shape evolving software, *Communication of the ACM*, **44**, 43–50.

Pawlak, R., Duchien, L., Florin, G., Martelli, L. and Seinturier, L. 2000. Distributed separation of concerns with aspects components, *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, IEEE Press, 276.

Rashid, A. 2001. Computing Department, Lancaster University, Lancaster, UK, 1–11.

Robillard, M.P. 2000. Department of Computer Science, University of British Columbia, Vancouver, BC, Canada, 1–15.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddi, F. and Lorensen, W. 1991. *Object-oriented modeling and design*, London, UK, Prentice Hall.

Tarr, P., Ossher, H., Harrison, W. and Sutton, S. M. 1999. *N* Degrees of separation: multi-dimensional separation of concerns, *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, California, USA, 107–119.

Tekinerdogan, B. and Aksit, M. 2000. Separation and composition of concerns through synthesis-based design, *Proceedings of the OOPSLA' 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, USA.

Turner, C. R., Fuggetta, A., Lavazza, L. and Wolf, A. L. 1998. Feature engineering, *Proceedings of the 9th International Workshop on Software Specification and Design*, Ise-Shima Isobe, Japan, 162–164.

Walker, R., Baniassad, E.L.A. and Murphy, G. 1998. University of British Columbia, Canada.