

# An Analysis of Design Approaches for Crosscutting Concerns

Ruzanna Chitchyan, Ian Sommerville, Awais Rashid  
Computing Department, Lancaster University, Lancaster LA1 4YR, UK  
r.chitchyan@lancaster.ac.uk, {is | awais}@comp.lancs.ac.uk

## Abstract

A number of approaches have been proposed to provide support for crosscutting concerns at the design level. This paper compares some of these design approaches. A set of “good design” criteria is used to identify strengths and weaknesses of the individual approaches. The paper concludes with a proposition to unify the strengths of the discussed approaches into a more effective model.

## 1 Introduction

A number of approaches have been proposed to provide support for crosscutting concerns at the design level. Most of these approaches have been developed to augment the Object-Oriented (OO) design techniques (and techniques based on Object-Oriented design constructs, such as Subject-Oriented approach) with facilities for modularisation of crosscutting concerns.

Each of these approaches for dealing with crosscutting concerns at the design level has its advantages and disadvantages. This paper analyses four of these approaches in order to identify their similarities and differences. A set of “good design” criteria is used to identify the strengths and weaknesses of each individual approach. The analysis leads us to conclude that highly effective separation of crosscutting concerns can be achieved at the design level by combining the best features of these approaches into a unified approach.

In section 2 the selected design approaches are summarised and compared. Section 3 presents the “good design” criteria and analyses the selected approaches with respect to these criteria. Section 4 identifies the features of key importance for the unified model and outlines a possible version for this model. Section 5 summarises the discussion and identifies directions for future work.

## 2: Selected Design Approaches

The approaches discussed in the paper are: Composition Patterns, Aspect-Oriented Component Engineering, Hyperspaces and an approach proposed by Suzuki and Yamamoto. Although several other approaches have been proposed e.g. [9] [10], they suggest introduction of specialised stereotypes for each particular crosscutting concern. These approaches have been left out in favour of a more general one (by Suzuki and Yamamoto)[20].

### 2.1 Brief Summaries of Selected Design Approaches

This subsection presents brief summaries of the four selected design approaches, followed by their comparison.

#### 2.1.1 Composition Patterns

**Summary:** Clarke et al. [2], [7] propose Composition Patterns to design reusable aspects. The approach is based on the observation that there are patterns in the way crosscutting concerns behave with respect to the base designs they cut across. This behaviour holds irrespectively of the subjects crosscut by the concern. Thus, reusable subjects and concerns can be designed independently of each other. Composition rules are then used to compose these designs.

**Design Techniques/Notations:** Composition patterns require extensions to UML and are based on a combination of the subject-oriented design model for composing separate overlapping designs and UML templates [3], [4].

#### 2.1.2 Aspect Oriented Component Engineering (AOCE)

**Summary:** AOCE [11], [13], [15] is based on the idea that each component in a component-based system uses/provides services from/to other components. This approach focuses on capturing the concerns that crosscut many components in a component-based system. These concerns are used to categorise components

in accordance with their contribution towards the overall system properties and reason about the inter-component services. AOCE facilitates:

- identification, description and reasoning about high-level component requirements,
- grouping these requirements in accordance with systemic characteristics (the aspects),
- and refinement of these requirements into design-level aspects, which implement software component services.

The aspect information is encoded into the component interface, allowing other components (as well as developers and users) to dynamically discover and access the services provided/required by any given AOCE component.

**Design Techniques/Notations:** The approach introduces a set of UML meta-model extensions and visual notations used to make the provided and required aspect details explicit [14]. Class diagrams are extended with additional compartments for each kind of aspect that describe the aspect, its details and (optionally) the detail properties. Collaboration and sequence diagrams can be annotated to reflect the relations of the event flows to the provided/required aspect details.

### 2.1.3 Hyperspaces Approach

**Summary:** This approach proposes to use a set of modules that address a single concern (called hyperslice) to encapsulate concerns in dimensions other than the one used for the dominant formalism [16], [22]. The modules within a hyperslice are standard modules written in the chosen formalism, but these modules contain only those units that pertain to the concern addressed by the hyperslice. Hyperslices can overlap, i.e. a given unit in a hyperslice can appear, possibly in a different form, in other hyperslices. Although hyperslices do not always fulfil the completeness requirements of the chosen formalism, they observe “declarative completeness” – declaring (but not necessarily implementing) everything that is used within the hyperslice. All the concerns of importance are modelled as hyperslices, which are then composed to produce a complete system. At the composition stage issues such as overlapping are resolved via composition rules.

**Design Techniques/Notations:** The model can be instantiated in any formalism, at which time the notational constructs to which units and modules map are chosen as well as the way hyperslices are represented. Consequently, there are no standard design notations for this model. However, this model has been instantiated for the object-oriented formalism for which a composition tool, called HyperJ, has been developed [23].

### 2.1.4 Suzuki and Yamamoto’s Model

**Summary:** Suzuki and Yamamoto propose UML extensions to incorporate crosscutting requirements (aspects) into the model and UXF/a - an XML-based aspect description language developed to provide interchangeability of aspect model information between various development tools (e.g. CASE tools, aspect weavers etc.) [20], [21].

**Design Techniques/Notations:** UML extensions are introduced for *aspect* and *woven class*, while the aspect-class relationship is represented by the UML abstraction dependency element with <<realize >> stereotype. An aspect is depicted as a class rectangle with attributes, operations and relationships. The attributes are used by the set of operations, which are the weave definitions for the aspect with the signature of these definitions reflecting the elements affected by the given aspect.

### 2.1.5 Comparison of the discussed approaches

The selected approaches have a key similarity (in addition to the obvious objective to support separation of crosscutting concerns at the design level) in that all of them extend the same base design notation i.e. the UML.

Moreover, three of these approaches can be easily mapped to the forth one: the Hyperspaces approach. This is due to the fact that these three approaches are specifically developed for dealing with the problems of crosscutting concerns within a dominant formalism, while the Hyperspaces approach is more general, allowing instantiation in many formalisms with various notations. The mapping to Hyperspaces is presented below:

Hyperspaces Notations	Composition Patterns	AOCE	Suzuki & Yamamoto's Model
Units	instance variable declarations, methods, etc.	aspect details, aspect detail properties, instance variable declarations, methods, classes, interfaces, etc.	instance variable declarations, methods, etc.
Modules	classes; aspects; interfaces, packages	Components, aspect components	classes; aspects; interfaces, packages
Hyperslices	Composition patterns, subjects	aspects; set of components with same required/ provided services	Core classes; aspects
Hypermodules	subjects, composition patterns with their composition rules	Aspects with rules for matching components with required services to components that provide these services	Core classes with woven aspects

Nevertheless, they also are rather different:

	Hyperspaces	Composition Patterns	AOCE	Suzuki & Yamamoto's Model
Context	Instantiation specific, based on Subject-Oriented approach [8]	Subject-Oriented Paradigm	Component-Based Software Engineering	Design Support for Aspect-Oriented Programming
Design techniques for decomposition	based on Subject-Oriented Design <sup>2</sup> ideas	based on Subject-Oriented Design <sup>2</sup> ideas	Components, Object-Oriented design constructs	Object-Oriented design constructs; & XML constructs
Design techniques for composition	Instantiation specific integration relationships (using extended SOP composition rules);	SOD composition relationships; parameterised subjects; UML templates	Referencing through provided/required services	Assumed use of a weaving tool with no design-level composition rules
Extensions Introduced to UML	Instantiation specific integration relationships (using SOD composition relationships)	Composition relationships from the SOD; parameterised SOD subjects; amended <<bind>> attachment	Aspect, AspectDetail, AspectDetailProperty	Aspect, Woven Class

It is apparent from the above comparison that Composition Patterns and Hyperspaces have more in common than the other approaches.

In the next section we describe a set of “good” design criteria for design approaches that support crosscutting concerns. These criteria are then used to identify the strengths and weaknesses of each of the approaches discussed above.

### 3 Analysis of the Selected Design Approaches

#### 3.1 “Good Design” Criteria

We believe that it is desirable for “good” design techniques for crosscutting concerns to be able to provide a set of notations for encapsulating holistic crosscutting requirements, facilitate understanding of complex systems, be easily reusable and well suited for evolution. These characteristics are essential for efficiently mapping crosscutting requirements to designs and supporting changing requirements at the design level.

Good design techniques should also be supported by an appropriate implementation technology otherwise designs would not adequately link requirements to the code.

The following criteria are used to analyse the extent to which individual design approaches support our desired characteristics:

**Traceability:** ease of tracing crosscutting requirements to designs, design to parts of design models, and finally to code. This criterion is to ensure that the design approach has provided suitable constructs for separation of relevant requirements.

**Change propagation:** ease of change of the complete design when a change is introduced to a part of the model. If a crosscutting concern is well separated, it should be easily modifiable without affecting other concerns and should not be affected by modification of other concerns.

**Reusability:** ease of reusing designs for other systems. Same or similar crosscutting requirements often arise in different systems. The designs produced to satisfy these requirements for one of these systems should be readily reusable in the other systems. This implies that the crosscutting concerns of general nature captured in the design constructs should be context-independent, while the context information itself should be encapsulated in separate units.

**Comprehensibility:** ease of understanding the software system through its designs. The designs of the crosscutting concerns should be meaningful and understandable without referencing other concerns.

**Flexibility:** ability of the design technique to adapt to different separation of concerns techniques, implementation languages, tools etc. A flexible design technique could bring different separation of crosscutting concerns approaches closer together, as well as allow reuse of design artefacts across different approaches, languages and tools.

**Ease of learning and use:** design techniques should be easy to learn and to use without errors. This is necessary to ensure widespread use of the technique.

**Parallel development:** simultaneous development of parts of the system design. When the base and crosscutting concerns can be separately modelled, different types of concerns can be designed simultaneously by different people. This criterion contributes to increased productivity of design teams.

In the subsection below we evaluate our selected design approaches against the above criteria.

### 3.2 Performance per Criteria

- **Traceability**

The Composition Patterns approach clearly encapsulates crosscutting requirements in designated design units, providing for good traceability from requirements to design. If each design unit is implemented independently before composition, the designs are easily traceable to modules at the code level. If, however, the designs are composed and then the integrated designs are implemented, tangling is re-introduced at the implementation level, precluding clear traceability from requirements to code modules.

Unlike Composition Patterns, AOCE does not need to merge crosscutting concerns into other component designs. All designs merely refer to each other, which constantly keeps the component designs intact and easily traceable across their development cycle.

AOCE also provides mechanisms for reasoning about groups of requirements and design level aspects which assist in identification, specification and reasoning about groups of interrelated components. This allows global, system-wide requirements to be captured, constraining more detailed aspects and assisting in tracing these system-wide requirements to the design level [11].

This approach also allows developers to reason about the validity of component configurations (by comparing provides/requires relationships between linked and composed components), once more contributing to traceability of the requirements.

Hyperspaces provide good traceability of both crosscutting and base concerns. Each concern can be clearly traced to a hyperslice design and implementation. However, when all required hyperslices are composed – a complete system is produced where individual concerns could be hard to trace (especially at the implementation level).

As in the case with Composition Patterns, in Suzuki & Yamamoto's model separation of crosscutting concerns into aspect design structures assists in tracing requirements. But, unlike Composition Patterns, this approach references the designated elements to be integrated with the aspects (e.g. methods) in the

signatures of aspect operations. Thus, this approach lacks clear composition rules at the design level, and composition is hard-written into an aspect on case-by-case basis. Traceability is impeded by references to base objects.

	Composition Patterns	AOCE	Hyperspaces	Suzuki & Yamamoto's Model
Summary for Traceability	Crosscutting requirements are clearly traceable to composition patterns, but tangled as designs are merged	Designs do not need merging, but stay intact and are easily traceable across the whole lifecycle	Crosscutting requirements are clearly traceable to hyperslices, but tangled as designs are merged	Partial support: crosscutting requirements are traceable to aspects, which in turn reference affected object elements

- **Change propagation**

Composition Patterns provide an ability to introduce changes into any subject, without invading other subjects. However, when changes are introduced into design subjects the composition relationships have to be re-visited and updated. In the case when the changed design has several layers of composition all of these need to be reviewed or invalid composition could be passed on to other modules.

The AOCE design-level aspects record the services they affect, and may also record aspects whose concerns overlap with their own, allowing designers to track and analyse possible change propagation using these dependency links. Nevertheless, changing a provided service will affect all components that require that service. Thus, although change propagation can be monitored, there are no mechanisms for an efficient change introduction.

The Hyperspaces approach localises changes to a concern within the hyperslice in which this concern is encapsulated. Additional requirements can be introduced non-invasively by merely designing a new hyperslice and composing it into the system. Amendments to the existing hyperslices can also be done by composing in a new hyperslice with the required changes.

Yet, like for Composition Patterns, a change to a hyperslice could require updating of all the composition relationships in which the given hyperslice participates. In the worst case scenario removal of a concern together with its units could require re-writing of units in other hyperslices that use the removed ones.

In Suzuki & Yamamoto's model changes to either aspect or object designs not always could be effected independently as elements of objects are hard-written into aspect operations. However, since the model allows the designs to be translated into UXF/a description and used for other design diagrams/models, tools etc. change in the UXF/a representation could be easily propagated to all designs based on that representation.

	Composition Patterns	AOCE	Hyperspaces	Suzuki & Yamamoto's Model
Summary for Change propagation	Non-invasive changeability, but needs to review composition rules	Change is monitored, but not effectively propagated	Non-invasive changeability, but needs to review composition rules	Partial support for changeability, efficient change propagation via UXF/a

- **Reusability**

Composition Patterns produce reusable designs for crosscutting concerns, which could be composed with any other base design (produced in the formalism of the composition pattern). Use of a composition pattern can be customised both through composition and design amendments to accommodate domain specific requirements to the crosscutting concerns.

AOCE provides a decoupled interaction mechanism, increasing reusability of components due to the information about their provided/required services. While component designs may be reused across different implementation technologies, some designs could require adaptations (e.g. a number of aspects from JViewer will be redundant in EJB as their services are provided as parts of EJB containers).

The Hyperspaces approach is similar to Composition Patterns, in that it improves reusability, by modelling generally useful capabilities separately from special-purpose (domain-specific) ones. Hyperspaces go a step further by allowing any unit or module from any hyperspace to be reused in a different hyperspace.

Suzuki & Yamamoto's model too allows reuse of both aspect and object designs, although the signatures of aspect operations should always be updated. This approach also provides an ability to interchange aspect description information between UXF/a compatible development tools and across the software development lifecycle.

	Composition Patterns	AOCE	Hyperspaces	Suzuki & Yamamoto's Model
Summary for Reusability	provides reusable designs, customisable both through composition and design amendments	Provides a decoupled interaction mechanism, increasing reusability of components due to the information about their provided/required services	provides reusable designs, customisable through composition and design change; allows any unit or module from any hyperslice to be reused in a different hyperslice	provides partially reusable designs, and interchangeability of aspect descriptions between UXF/a compatible tools and across lifecycle

- **Comprehensibility**

Composition Patterns produce clear segregated designs for base and crosscutting concerns. However, when the composition patterns and the base design are composed the tangling is re-introduced. Consequently, the integrated design becomes somewhat complicated reducing comprehensibility.

In AOCE encapsulation of the crosscutting systemic issues in component interfaces assists in understanding the relationships between components. At the same time, although cross-referencing components that provide and request services helps to follow the relationships between components, the designs quickly become cluttered by a web of arrows.

The Hyperspaces approach promotes better alignment of the requirements to designs which contributes to better comprehensibility and traceability of the artefacts across the lifecycle. The approach, therefore, assists in understanding each concern from the viewpoint of the requirement being modelled. Various viewpoint models of the same concern need to be reconciled in order to produce the complete concern design. If there is no composition tool available to automate this process, it could become a complex task, significantly reducing comprehensibility of concern designs and the complete system. Knowledge of the composition relationships is also essential for integrating designs.

Suzuki & Yamamoto's model uses only extensions to UML to assist in separation of crosscutting concerns into dedicated design units. This model does not provide any clear composition rules, instead aspects reference elements of the base objects into with which they will be woven by a weaving tool. Besides, the *woven class* extension can be confusing, as many aspects may be woven to a base class, thus making it difficult to understand which aspect is being addressed in a specific case or is responsible for a specific result.

	Composition Patterns	AOCE	Hyperspaces	Suzuki & Yamamoto's Model
Summary for Comprehensibility	Individual designs are well comprehensible, but integrated designs are less so	aspects assist in understanding the relationships between components but the designs become cluttered with cross-referencing arrows	Same as Composition Patterns	Aspects are separated, but still reference base object elements which reduces comprehensibility. Also, the <i>woven class</i> can be confusing.

- **Flexibility**

Composition Patterns approach is flexible across implementation models: design constructs from this approach have been mapped to AspectJ and HyperJ [5], [6]. However, there has yet been no attempt to map this approach to other AOP techniques, such as Composition Filters and Adaptive Programming.

AOCE too can be realised in various ways. By now the approach has been implemented using a dedicated aspect-extended component framework (JViews), a standard component technology (EJB), and an aspect-oriented programming language (Perceval) [12], [14].

Suzuki & Yamamoto’s model is closely tied to one specific implementation model – the model of AspectJ, bringing up to the design level the specificities of the implementation tool. Due to this reason the present model is not easily mapped to any other techniques or types of implementation tools (unless mechanisms for mapping between AspectJ and other techniques are exploited [1]). However, the design information from this model can be transferred through UXF/a representation and visualised by a different development tool.

The Hyperspaces approach claims to be flexible across any design formalism, but as yet has been formally instantiated only for OO with a Java implementation. Nevertheless, as shown above all the other discussed approaches can be considered specific instantiations of Hyperspaces approach.

By simply including or excluding the particular hyperslices into the system composition Hyperslices approach allows to “mix-and-match” the designs for a customised system providing a non-invasive customisation and adaptation mechanism.

However the fineness of the granularity of the units of decomposition (& composition) is limited to declarations (e.g. functions or members), restricting the flexibility of the approach. Also the requirement of fitting all dimensions of decomposition within a single formalism deprives the approach from the benefits of using the best-suited formalism for a given concern [19].

	Composition Patterns	AOCE	Hyperspaces	Suzuki & Yamamoto’s Model
Summary for Flexibility	flexible across implementation models, not yet mapped to techniques other than AspectJ and HyperJ	could be realised in various ways	flexible across any design formalism, allows “mix-and-match” of designs , but fineness of unit granularity is limited and forcing a single formalism restricts flexibility.	closely tied to a specific implementation model, although information can be transferred between tools/developers due to UXF/a representation.

- **Ease of learning and use**

Use of standard OO notations and diagrams (UML, OCL etc.), facilitates the learning process for all four discussed approaches, although all of these approaches introduce some extensions to the standard UML.

For Composition Patterns and Hyperslices the users also have to learn Subject-Oriented design and composition semantics [18], while Suzuki & Yamamoto’s model requires knowledge of AspectJ (or similar tool) which could imply a higher learning curve.

Although easy to learn, AOCE increases design costs, as more effort is required for identification, documentation and use of component aspects. This could impede widespread use of the technique. Allowing for run-time reconfigurability and dynamicity of the components through encapsulation of the crosscutting systemic issues in component interfaces contributes to the ease of use of these components. This approach also supports check of composed system validity by comparing provided/required services of the components.

In order to instantiate the Hyperspaces model in a given formalism, a complex composition tool for that formalism needs to be developed first, as the alternative of manually specifying composition relationships is not practical.

In respect with this criteria, Suzuki & Yamamoto’s model has an advantage of providing an easy way of communicating information between designers and tools, using UXF/a, e.g. across the internet etc.

	Composition Patterns	AOCE	Hyperspaces	Suzuki & Yamamoto's Model
Summary for Ease of learning and use	Use of standard OO notations/diagrams facilitates learning and use. Learning complicated by need to learn Subject-Oriented design and composition semantics.	Use of standard OO notations /diagrams facilitates learning and use. Use impeded by extra design effort needed, but supported by additional run-time reconfigurability and dynamicity & ease of configuration validation	Use of standard OO notations/diagrams facilitates learning and use. Learning complicated by need to learn Subject-Oriented design and composition semantics.	Use of standard OO notations /diagrams and easy way of communicating information between designers, tools etc. facilitates learning and use. Learning is complicated due to need to learn AspectJ (or similar tool)

- **Parallel development**

Composition Patterns, AOCE, Hyperspaces approaches - all allow the crosscutting concern and the base designs to be developed simultaneously, independently of one another, introducing parallelism to the design stage. While Suzuki & Yamamoto's model, when designing crosscutting concerns, tends to refer to the objects they cut across, reducing the parallelism of design.

After producing separate designs, Composition Patterns approach then simply composes these designs together. AOCE also allows separately designed components to be reviewed as a set to ensure that all required inter-component services are provided for by some of the system component(s).

Similarly, in Hyperspaces the "declarative completeness" rule requires declaration of the elements used in the hyperslice, but at the composition stage it is necessary to introduce checks to ensure they are actually implemented by other hyperslices, or the system could end up trying to use declared but not implemented items.

	Composition Patterns	AOCE	Hyperspaces	Suzuki & Yamamoto's Model
Summary for Parallel development	The approach allows the crosscutting concern and the base designs to be developed simultaneously, independently of one another	The approach allows the crosscutting concern and the base designs to be developed simultaneously and reviewed as a set.	Crosscutting concern and the base designs can be developed simultaneously & independently of one another. But checks are required to ensure that declared items have been implemented	While designing crosscutting concerns, tends to refer to the objects they cut across, reducing the parallelism of design.

#### 4 Unification

In the above sections we have compared and analysed four design approaches for capturing crosscutting concerns. Each of these has its strengths and weaknesses.

We wish to combine the above-discussed approaches into a unified one. This unified design approach for separation of concerns will combine the strength of the discussed models and overcome their weaknesses.

The "good design" criteria analysis conducted in section 3 revealed that Hyperspaces and Composition Patterns have similar strengths and weaknesses. Hyperspaces provide a flexible general separation of concerns model for which the Composition Patterns approach appears to be an instantiation in the OO formalism. Each design for a crosscutting concern in the Composition Patterns approach directly corresponds to a hyperslice in the Hyperspaces terminology.

AOCE approach contributes to increased Reusability by keeping developers/users informed about inter-component provided/required services. Having this information incorporated into the component interfaces, the approach also achieves run-time reconfigurability and dynamicity. It also achieves better *Ease of Use* results by allowing simple way of verifying validity of system configuration.

Suzuki & Yamamoto's model can facilitate *Change Propagation* and *Flexibility* by allowing the design information to be transferred between different tools and visualised. Using a human-readable representation format for these design descriptions, the model also achieves improved *Ease of Use*.

Thus, it would be reasonable to suggest that a unified design approach for separation of crosscutting concerns could be based on the general model of Hyperspaces with Composition Patterns as its OO instantiation. In order to achieve better reusability and ease of use, as well as improved comprehensibility of the composed systems, this model should use AOCE provided/required services method for analysing inter-hyperslice/hypermodule relationships and verifying configuration validity for newly composed systems. It also should use the UXF/a method from Suzuki & Yamamoto's model to get an ability to map designs into a transferable format (UXF/a) which can, in turn, be mapped into different designs for the same system, transferred between design tools across the systems lifecycle and facilitate readability and information exchange for between developers and development tools.

This will not only provide a more efficient design technique, but will also unify the design notations, providing all benefits of standardisation.

## 5 Summary and Future Work

In this paper four approaches for separation of crosscutting concerns at the design level have been discussed and compared. The approaches have been analysed with respect to a set of "good design" criteria and their strengths and weaknesses have been discussed.

It has then been suggested to combine the strengths of these approaches into a unified standard method for designing crosscutting concerns. This model:

- maintains clear separation of crosscutting concerns throughout the development lifecycle, rendering the benefits of traceability, comprehensibility, parallel development, and reuse;
- further facilitates reuse by helping to reason about what services hypermodules provide/ require; to identify other required hypermodules; to exchange aspect design information between developers and development tools, and to validate system configuration;
- supports immediate change propagation from an aspect description to all models based on visualisation of that description;
- maintains flexibility and ease of learning of Composition Patterns and Hyperslices approaches.

However, it should be noted that this unified model does not claim to be either complete or finalised. This model will need to evolve along with the progress in our further understanding of AOSD issues, adopt to the aspect-oriented requirements specification techniques, allow for incorporation of other AOD approaches, etc.

Some of the challenges posed to the AOD are:

- Accommodating other AOD techniques, such as Composition Filters etc. in the unified design model;
- Development of a mechanism to incorporate different formalisms within a single design model, so that the most suitable formalism can be used for designing each given concern;
- Development of a mechanism that would allow for both coarse and fine level composition

These are the problems that we intend to focus on in the near future.

## References:

- [1] C. Chavez, A. F. Garcia, and C. J. P. Lucena, "Some Insights on the Use of AspectJ and HyperJ", Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster University, UK, 2001, Cooperative Systems Engineering Group, Technical Report CSEG/03/01, pp. 20-24
- [2] S. Clarke, R. J. Walker. "Composition Patterns: An Approach to Designing Reusable Aspects" In proceedings of the 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, May 2001.
- [3] S. Clarke. "Extending standard UML with model composition semantics" To appear, in Science of Computer Programming, Elsevier Science, 2002.
- [4] S. Clarke, W. Harrison, H. Ossher, P. Tarr. "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code" In proceedings of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) Denver, Colorado U.S., November 1999.

- [5] S. Clarke, R. J. Walker. *"Separating Crosscutting Concerns across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J"*. Technical Report [TCD-CS-2001-15], Trinity College, Dublin and UBC-CS-TR-2001-05, University of British Columbia. May 2001
- [6] S. Clarke, R. J. Walker. *"Mapping Composition Patterns to AspectJ and Hyper/J"* ICSE 2001, Workshop on Advanced Separation of Concerns in Software Engineering.
- [7] S. Clarke. *"Designing Reusable Patterns of Cross-Cutting Behaviour with Composition Patterns"* OOPSLA 2000, Workshop on Advanced Separation of Concerns.
- [8] W. Harrison and H. Ossher, *"Subject-Oriented Programming - A Critique of Pure Objects"*, Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1993
- [9] J. Herrero, F. Sánchez, F. Lucio & M. Toro *"Introducing Separation Of Aspects At Design Time"*. In proc. Of Aspect and Dimensions of Concerns Workshop at ECOOP, 2000
- [10] W. Ho, F. Pennaneac'h, J. Jezequel, & N. Plouzeau. *"Aspect-Oriented Design with the UML"*. In Proc. of Multi-Dimensional Separation of Concerns Workshop at ICSE, pp. 60-64, 2000.
- [11] J.C. Grundy *"Multi-perspective specification, design and implementation of software components using aspects"*. In International Journal of Software Engineering and Knowledge Engineering, Vol. 20, No. 6, December 2000, World Scientific.
- [12] J.C. Grundy *"An implementation architecture for aspect-oriented component engineering"*. In Proceedings of the 5th International Conference on Parallel and Distributed Processing Techniques and Applications: Special Session on Aspect-oriented Programming, Las Vegas, June 26-29 2000, CSREA Press.
- [13] J.C. Grundy *"Aspect-oriented Requirements Engineering for Component-based Software Systems"*. In Proceedings of the 1999 IEEE Symposium on Requirements Engineering, Limerick, Ireland, 7-11 June, 1999, IEEE CS Press, pp. 84-91.
- [14] J. C. Grundy, R. Patel *"Developing Software Components with the UML, Enterprise Java Beans and Aspects"*. In Proceedings of the 2001 Australian Software Engineering Conference, Canberra, Australia, 26-28 August 2001, IEEE CS Press.
- [15] J.C. Grundy *"Supporting aspect-oriented component-based systems engineering"*. In Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany, 16-19 June 1999, KSI Press, pp. 388-395.
- [16] H. Ossher and P. Tarr *"Multi-Dimensional Separation of Concerns using Hyperspaces"*. IBM Research Report 21452, April, 1999. This paper describes hyperspaces in more detail.
- [17] H. Ossher and P. Tarr. *"Multi-Dimensional Separation of Concerns and The Hyperspace Approach."* In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.
- [18] H. Ossher, M. Kaplan, A. Katz, W. Harrison, V. Kruskal *"Specifying Subject-Oriented Composition"*. Theory and Practice of Object Systems, volume 2, number 3, 1996, Wiley & Sons.
- [19] A. Rashid *"A Hybrid Approach to Separation of Concerns: The Story of SADES"*. Proceedings of the 3rd International Conference on Meta-Level Architectures and Separation of Concerns Reflection 2001, Lecture Notes in Computer Science 2192, pp. 231-249
- [20] J. Suzuki, Y. Yamamoto *"Extending UML with Aspects: Aspect Support in the Design Phase"*. ECOOP 1999, The 3rd Aspect-Oriented Programming (AOP) Workshop.
- [21] J. Suzuki, Y. Yamamoto *"Toward the interoperable software design models: quartet of UML, XML, DOM and CORBA"*. ISESS 1999, 4th IEEE International Software Engineering Standards Symposium.
- [22] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. *"N Degrees of Separation: Multi-Dimensional Separation of Concerns"*. Proceedings of the International Conference on Software Engineering (ICSE'99), May, 1999. This paper describes the domain of multi-dimensional separation of concerns.
- [23] P. Tarr & H. Ossher *"Hyper/J User and Installation Manual"*. <http://www.research.ibm.com/hyperspace>.