Visual Separation of Concerns through Multidimensional Program Storage

Mark C. Chu-Carroll
IBM T. J. Watson Research Ctr
19 Skyline Dr, Hawthorne NY
mcc@watson.ibm.com

James Wright
IBM T. J. Watson Research Ctr
19 Skyline Dr, Hawthorne NY
jwright@watson.ibm.com

Annie T. T. Ying
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
aying@cs.ubc.ca

ABSTRACT

Aspect-oriented software development (AOSD) has primarily focused on linguistic and meta-linguistic mechanisms for separating concerns in program source. However, the kinds of concern separation and complexity management that AOSD endeavors to achieve are not the exclusive province of programming language design.

In this paper, we propose a new model of concern separation called *visual separation of concerns (VSC)*, which is based on a new model of program storage. By altering the mechanisms used to store and manipulate program artifacts, much of the capability of concern separation can be captured without performing any linguistic transformations. We also describe our implementation of VSC, which is based on Stellation, an experimental software configuration management system. The VSC approach combined with software configuration management can have advantages over conventional approaches by avoiding program transformations, by providing persistent storage of features such as concern maps, and by enabling new techniques for concern identification and manipulation.

1. INTRODUCTION

Separation of concerns is one of the central tenets of proper software design and engineering. As software complexity has increased, tool and language designers have developed new techniques for managing that complexity through the separation and management of distinct concerns.

One of the most recent efforts in this direction is aspect-oriented software development (AOSD)[29], which is based on the recognition that concerns are often difficult to separate because they follow different fundamental semantic structures. Since most current programming languages require semantic structures to be reflected in the basic syntactic structure of a system's code, and dictate that there is exactly one dominant semantic structure to the system, this means that some concerns must be implemented in a way that cuts across the basic structure of the system. Tarr et al have termed this problem *the Tyranny of the Dominant Decomposition*[31].

AOSD 2003 Boston, MA USA

AOSD addresses this problem by allowing cross-cutting concerns to be managed, either by allowing a cross-cutting concern to be sliced out of a program, or by allowing concerns to be implemented separately and then integrated together. Most techniques in both categories are based on some form of semantic or linguistic transformation of the program, either slicing aspects out of the program into independent semantic units, or through the use of language extensions or meta-languages to describe how semantically independent implementations of different concerns should be composed into a single integrated semantic unit.

We propose a different approach for aspect-oriented separation of concerns, called *visual separation of concerns (VSC)*. VSC presents separate *views* of cross-cutting aspects, allowing programmers to read and edit aspects in isolation, while leaving the semantic structure of the system untouched. The visual technique is based on a new model of program storage that separates notions of storage, language semantics, and program organization. By doing this, it is possible to allow programmers to view their system through multiple, overlapping, editable organizations of the system, each of which presents concerns separated by a different criteria; each organization represents a decomposition of the system according a different *dimension of concern*.

2. THE PROBLEMS WITH COMPOSITIONAL AOSD

Typically, aspect oriented systems separate concerns *linguistically*, by either extending programming languages, or by providing a compositional metalanguage. In the former approach, typified by AspectJ[20], constructs are added to the language which programmers use to implement cross-cutting aspects, which are then composed into the primary semantic structure of the system at compile time. In the latter approach, typified by HyperJ[23] and Composition Filters[4, 5] concerns are implemented separately as complete components using standard programming language semantics, and those separate concern implementations are then composed together at compile time through the use of a meta-language describing how the different concerns should be integrated.

We believe that these compositional approaches have two weaknesses: invisible semantic transformation, and *scatter*. We will discuss each of these in greater detail.

It should be noted that we do not believe that the visual technique is a replacement for compositional AOSD systems; compositional techniques provide powerful capabilities that can yield significant benefits in dealing with the complexity of large software systems. However, we believe that the benefits of the visual technique are synergistic with those of compositional AOSD tools. We believe

that in many cases, the benefits of the visual technique are large enough that full compositional techniques are not needed; and further that when this is the case, avoiding the program-transforming properties of composition is a significant benefit.

2.1 Invisible Semantic Transformation

By invisible semantic transformation, we mean that compositional approaches generate executable code in which code may have been inserted by aspect composition which has significant effects, but where the potential for these effects are not signaled in the pre-composition source code.

For example, in AspectJ, programmers implement aspects by specifying join points in standard Java code, and providing a body of code to be inserted at program locations that match that join point specification. The target of the composition is a standard Java source file, with no indication of the possibility of other code being inserted. A programmer unfamiliar with the aspect composition scheme in use could easily be confused, because composed code can alter the visible state of the system in ways that appear impossible in the original code. There is no indication in the basic Java code that the code in the class source file is not semantically complete. The programmer needs to understand the full aspect system in order to understand their code, and there is no direct way of identifying all of the aspects being composed except by looking at the build process.

Thus, in compositional systems, the final executable structure of the program does not precisely match the structure expected from reading the source code of any standard semantic unit of the program. While the benefits of AOSD are significant, the problem of invisible semantic transformation, particularly during the ongoing process of software maintenance, should not be discounted.

2.2 Scatter

Scatter, while closely related to invisible semantic transformation, is in some sense a deeper issue. It is a direct manifestation of the application of abstraction in programming language design. The same linguistic tools that provide the capability to separate concerns also produce a non-trivial negative effect on the readability and traceability of code. *Scatter* is our term for the effects of abstraction and indirection in programming languages causing the flow of control through a program to become increasingly indirect. We call it scatter because the common manifestation of this problem is that the flow of control for a particular operation is scattered through many different storage units, making it difficult to view or understand as a consistent whole.

In early programming languages, programs were quite small and monolithic, and control flow was straightforward and easy to follow. Over time, monolithic programs became too complex, and structured programming introduced named subroutines to help manage complexity. Understanding specific control flow became a bit harder, but there were significant gains in both expressiveness and in the manageable complexity of the system. In our terms, this introduced the first layer of scatter: instead of a straightforward control flow, control now jumped in and out of named units, and understanding the execution stack became crucial to understanding program execution.

Object-orientation introduced a significant new layer of scatter: with objects, a name now referred to a family of type-related subroutines, with a particular subroutine selected dynamically at execution time. To understand flow, the programmer needed to understand the type structure of the system, and the ways in which names would be bound during execution. This means that to understand an operation, a programmer needs to follow its code through multiple

```
package proj.util;
import java.util.Map;
import java.util.HashMap;
public class StringMap {
    public StringMap() { ... }
    public void put(String key, String value) { ... }
    public String get(String key) { ... }
    protected Map _content = new HashMap();
}
```

Figure 1: Example of division of a Java class into fragments

class definitions, which may be scattered through multiple source files. In a complex program, understanding control flow could involve viewing dozens of source files.

Aspect-orientation can introduce yet another layer of scatter. A particular operation may still involve methods defined in many different classes, but now, each of those methods may be assembled from a collection of aspect sources. To understand the system, the programmer must understand *all* of the classes in *all* of the separate concerns, and how they will all be composed together.

Each step in this process of programming language evolution introduced significant, expressive tools that allowed programmers to work with vastly more complex systems. But at the same time, each step introduced more scatter, and this scatter produced significant new issues of its own. The scatter effect must be considered when designing and implementing a system. When introducing a new aspect or a separating out a new concern, programmers need to consider whether the benefit of that separation outweighs the added scatter.

A major advantage of the visual concern separation approach is that it counteracts scatter. With a VSC tool, a programmer can view code with concerns separated when it is valuable to separate it, but they can also view the code through a non-separated non-scattered view.

3. VISUAL SEPARATION OF CONCERNS: A NON-COMPOSITIONAL APPROACH TO AOSD

We propose a different approach, which we call visual separation of concerns *(VSC)*. Instead of providing linguistic methods of separating concerns through modification of the programming model, VSC is based on separating concerns by modifying the *storage* model.

In conventional tools, programmers store code using source files. The location of an element of code within a source file can express either semantic properties (class membership, name scope), or organizational properties (related methods will often be closely co-located). A file-based storage system forces programmers to place program elements in exactly one position, strictly limiting storage structure as an expressive mechanism. We believe that this is a problem very similar to the tyranny of the dominant decomposition, and have thus termed it *the tyranny of source files[9].*

In the VSC approach, storage is not in terms of files, but in terms of smaller fine-grained elements called *fragments*. Fragments consist of atomic units of code from an underlying programming language, with a linkage to their semantic context. Unlike the similarly named fragment construct of programming languages like

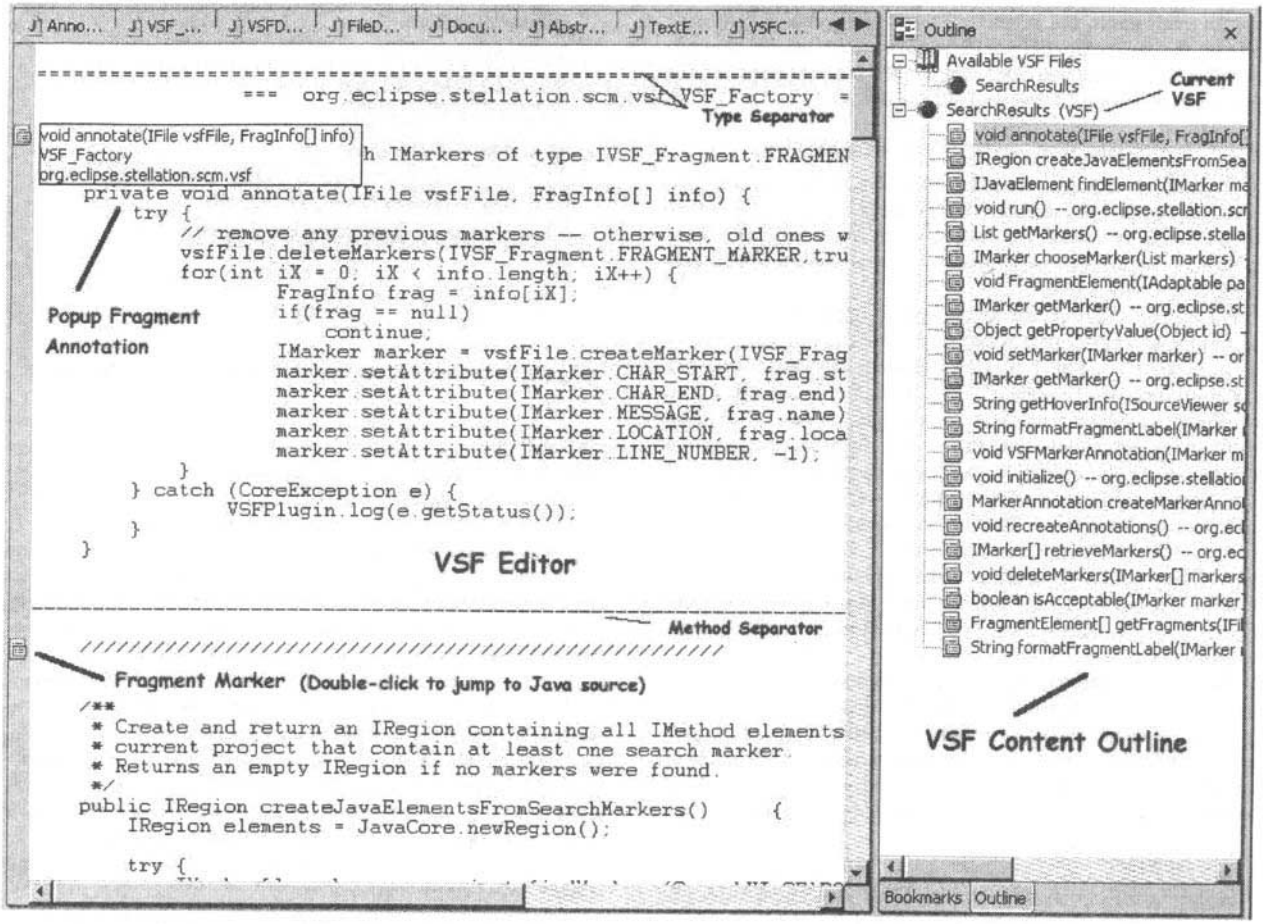

Figure 2: Screenshot of a VSF Editor with an Aggregate Outline

Beta[22], code fragments are *not* a semantic construct: a fragment is specific atomic segment of code in a particular semantic context determined by its position in a canonical view of the program. The size of a fragment is dependent on the underlying language: it consists of the smallest identity-preserving element in the language structure. For example, in Java, fragments consist of methods, fields, etc. (The division of a java source file into fragments is illustrated in figure 1.) Programmers then view and manipulate code through source-file like code views, called *virtual source files,* which are generated from dynamically selected collections of fragments. Programmers use the dynamic selection mechanism to produce code views that reflect decompositions of the system according to different dimensions of concern. These code views have *no* semantic effect on the program: programmers view collections of fragments through VSFs, but the fragments are still compiled and executed in their original semantic context.

For example, figure 2 presents a screenshot of the Eclipse IDE extended to edit VSFs. The VSF Editor (left) presents views a set of source fragments selected by some criteria (in this case, methods using the interface *IMarker*). Fragments which meet the criteria are included in the VSF view, separated by a narrow horizontal separator; a thicker separator is used to denote type (source file) boundaries. The Content Outline on the right lists all fragments in the current VSF. The underlying source code semantics are unaltered.

VSC is a powerful technique enabling programmers to view concerns in isolation, without transforming the structure or semantics of the program. Although the VSC technique does not have the full power of compositional approaches, it allows programmers to gain much of the benefit of concern separation without confronting the additional complexity of compositional systems, or the scatter related problems that they cause.

The VSC model has one very significant advantage over the compositional techniques: it enables the creation of concern views, similar to the viewpoints of Finkelstein[15], that integrate products from different phases of the development process. For example, a concern view can present the documentation of a concern, the analysis requirements that led to the concern, design diagrams describing the implementation of the concern, and the code that implements that concern together in one consistent view or set of views.

3.1 Separating Concerns without Composition: an Example

For example, consider figure 3, which represents a illustratory subset of code implementing a distributed software development environment. In this system, the UI is dynamically assembled from

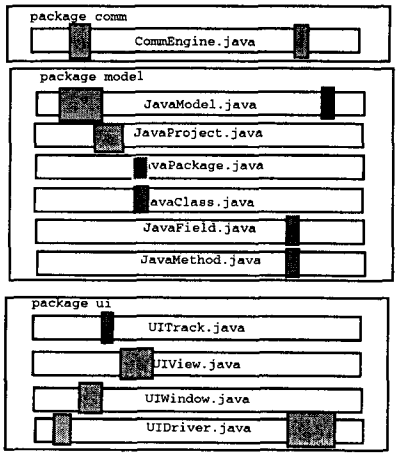

Figure 3: An illustration of scatter in an object-oriented application

contributions provided by dynamically loaded components. The horizontal boxes correspond to Java source files and the package directories containing those source files in a conventional file-based storage system. The irregular shaded boxes correspond to code involved in the menu generation process. In the standard source-file view (in the complete system), this menu generation process is scattered through 30 different source files.

The shaded code could be considered a menu generation concern or a aspect. But in practice, this is a very abstract, highly scattered concern which involves: (a) pieces of the basic communication system including message generation and dynamic subscription; (b) the production of XML documents describing menus (c) the registration of event types and listeners for the UI; and more! In an AspectJ-like model of AOSD, there is no useful way to capture an aspect or concern like this, because this aspect itself involves so many other aspects. Any encapsulation of this using Aspect-J aspects will create a huge amount of scatter, almost certainly adding enough scatter-based complexity to the system to outweigh the benefits of encapsulation.

In a system like HyperJ, menu generation in this system corresponds to a hyper-slice or hyper-module: a collection of code that when viewed through the lens of menu generation is presented as a complete concern. This is a better solution than that afforded by AspectJ, because it does allow this abstract concern to be encapsulated and represented; however, there are numerous interactions between this code and other code in the surrounding context of each of the highlighted chunks. To understand the full impact and interaction of this code in surrounding context, a developer must be able to understand the code both in isolation as an encapsulated concern, and in context as a part of the complete components where the various fragments of the menu generation will execute. In the HyperJ model, this is possible: hyperslices can overlap, and the same code can be viewed in the context of different hyperslices. However, because of the semantic transformation of the code in the process of slicing, the programmer will need to carefully reconcile the two different transformed code selections.

In the visual separation model, the system can generate an encapsulated view that presents the complete menu generation concern as a single virtual source file view. At the same time, the programmer could see the standard Java source organization in another view. Changes made in either view will immediately appear in both. The programmer can freely switch between views, without having to deal with any reconciliation: the code seen in both views is identical, untransformed.

The VSC model can thus reduce the scatter issue: languages may scatter code, but VSC tools can gather code together when necessary, while still allowing the programmer to reap the linguistic benefits of concern separation. Further, programmers can avoid introducing scatter in places where visual separation is sufficient, and linguistic separation is not necessary. Finally, when the use of advanced tools like HyperJ or AspectJ is necessary, visual separation can help reduce the scatter complexity introduced by the features of these languages: programmers can select views that co-locate all of the code that will be composed into a single execution unit.

3.2 Requirements for Visual Separation of Concerns

For a system to support visual separation of concerns, there are a small number of requirements:

1. *Fine-grained fragment storage.* A VSC system must store program artifacts as fine-grained units. The core VSC functionality is based on dynamically re-arranging the code into different views, and changes made to a fragment in any one view should be immediately reflected in all other views containing that fragment. This means that the basic storage model must be built around fine-grained storage and manipulation of program fragments.

2. *Aggregation.* A VSC system must provide some mechanism for combining collections of fine-grained program artifacts into larger source file-like views.

3. *Dynamic Fragment Selection.* A VSC system must provide some mechanism for selecting the set of fragments to be presented in a particular view. Possible realizations of this mechanism include (but are not limited to) a query language, a pattern-matching facility, or a browser tool for selecting related artifacts.

4. *Fragment-aware editing tools.* The VSC model is fundamentally based on a radically different storage model. The system needs some kind of program editing tool that allows programmers to work with the virtual source files generated by the system.

5. *Internalization/Externalization facilities.* Most common tools used by developers, including compilers, debuggers, and profilers do not support the sort of fine-grained storage systems provided by a VSC system. Thus, the system must provide some mechanism for externalizing sets of fine-grained artifacts as standard source files for manipulation by conventional tools.

4. VISUAL SEPARATION OF CONCERNS AND SOFTWARE CONFIGURATION MANAGEMENT

The primary focus of our work has been on advanced software configuration management *(SCM)* functionality in the Stellation system. Our visual separation of concerns model is derived from our work on multidimensional software configuration management, as described in [7, 8, 30, 10, 9]. We believe that a software configuration management system tightly integrated with a programming

SlotType = AtomicType | CollectionType
AtomicType = PrimitiveType | SemanticType | UnionType
PrimitiveType = Integer | String | Text | Binary
SemanticType = *language specific atomic type (e.g., java_method_decl)*
UnionType = AtomicType "or" AtomicType ...
CollectionType = Set of AtomicType | List of AtomicType

Figure 4: Aggregate Slot Types Summary

environment is the appropriate platform for building visual separation of concern tools. SCM systems are among the most ubiquitous tools used by software developers, and are the primary mechanism that programming teams use for storing programs and sharing work. Thus, for most developers, integrating VSC into an SCM system provides the most natural interface to this functionality.

Beyond the basic VSC functionality we will describe, integrating it with SCM provides several unique facilities. These are made possible by taking advantage of the versioning capabilities of the SCM system to provide versioning of both the fine-grained units, and the larger aggregates assembled from those fine-grained units. In particular, it enables a flexible mechanism for managing the history of concern mapping within a system; for using data mining and machine learning mechanisms for concern identification; and even for providing stronger integration of compositional AOSD systems with other tools.

4.1 Implementing VSC though SCM in Stellation

We are implementing our VSC model in the Stellation SCM system. Stellation is an experimental SCM system being developed as a research technology subproject of the Eclipse development environment[12].

Stellation is currently being implemented with VSC support for Java. Our implementation of the fine-grained versioning requirement is straightforward, but our mechanisms for aggregation,fragment retrieval, and externalization require elaboration.

4.1.1 Aggregation in Stellation

A detailed description of the Stellation aggregation mechanism can be found in [10].

Aggregation is our mechanism for combining groups of fragments into larger structures. This aggregation facility is required for building dynamic views. But an aggregation facility is useful for far more than just simple source views: in fact, aggregation is the key that enables many of the particularly powerful capabilities of Stellation. Aggregation is a general purpose metadata mechanism that allows Stellation to manage relationships and concern maps, to provide a mechanism for other tools to integrate thier data into a Stellation repository, and to provide data integration facilites for linking data from multiple sources or multiple phases of the development process.

This aggregate system allows programmers to define types of aggregates to represent different repository data and metadata structures. The type system allows programmers and tools to differentiate between aggregates created for different purposes, and to allow aggregates to have enough structure to represent complex data and relationships. An aggregate type looks like a type definition in a programming language: it consists of a set of named slots, each of which declares a type and a merge operator. Slot types are summarized in figure 4. A set of sample aggregate type declarations are illustrated in figure 5a.

Programmers can dynamically define and evolve aggregate type definitions. Any aggregate type whose slots contain artifacts that can be viewed using the Stellation UI can be presented as a view. For example, a developer could define an aggregate type to represent the relationship between a set of of requirements, and the fragments of code that implement those requirements. Once represented as an aggregate, that relationship can be versioned, maintaining the correct associations throughout the evolution of the system; it can also be presented as an editable view within the programming environment.

4.1.2 Artifact Selection in Stellation

To take advantage of the aggregate mechanism, both users and tools require some mechanism to dynamically create and populate aggregates. In order to do this, Stellation provides a query language, and allows programmers to populate the slots of an aggregate using queries. The query language is extension based, allowing programmers to provide components for adding new predicates and new conjugates to the query language.

In addition, since the query language itself is implemented as an extension component, developers can add new forms of queries beyond the "built-in" query language, including pattern matchers, query-by-example, or even a full logic programming language.

An example of query-based aggregate generation can be seen in figure 5b. This example illustrates how our aggregation mechanism creates source-file-like aggregate views that present a cross-cutting concern in a single source file, in this case producing the virtual source file corresponding to the scattered concern in figure 3. The UI for this system assembled menus by having a method named "populateMenus", which sent a message asking for contributions to the menu from different components, and then waited for contributions sent as responses. Following the control flow of the process of generating menus for a real system was extremely complicated: it involved methods from approximately 30 classes. In a conventional storage system, that required the programmer to trace code through 30 different files! In Stellation, the programmer can select a functional view that presents all code involved in this process in one place.

The work on Stellation has been largely independent of work in aspect-oriented programming languages - but it is noteworthy that the query language, which is being developed based on specific functionality requirements from users, naturally acquired constructs similar to AspectJ's join point specifiers, as seen in this example. More examples of our query language, and a description of how the query language can be implemented efficiently are found in [10].

4.1.3 Artifact Externalization in Stellation

Externalization is the capability to take an aggregate artifact within a Stellation repository and translate it into a form which can be used by external tools. There are two key pieces to this process: *export* (take an aggregate within the repository and translate it into its external form), and *import* (take an externalized aggregate, and translate that back into an internal form, generating a new version if the aggregate was modified).

Our approach to externalization is based on the use of standard XML tools. The XML community has done extensive work on document transformation using XSLT[13] and XML formatting objects[25]. We export all aggregates into an XML format defined by an aggregate schema, and then allow developers to use an XML tool to translate the resulting document into a desired format. Import is handled similarly: developers use external tools to translate their external form back into the Stellation XML schema, and the result is reintegrated into the repository. We plan to provide im-

```
aggregate java_class
    name: [conflict] String
    package: [conflict] java_package_decl
    imports: [union] java_import_decl
    decl: [conflict] java_class_decl
    members: [linear] java_member List
}

aggregate java_viewpoint {
    name: [conflict] String
    description: [linear] Text
    members: [dynamic] java_member List
}

aggregate bug_report {
    title : [conflict] String
    severity : [largest] Integer
    description : [conflict] Text
    subject_code : [dynamic] java_member_decl Set
    test_data : [union] test_case Set
}

aggregate specifies_relationship {
    specification : [union]Z_fragment Set
    implementation : [union]java_member_decl Set
}
```

(a) Aggregate Type Examples

```
menu_mgmt = new java_viewpoint {
    name = "menu management"
    description = "Menu management component of functional decomposition"
    members = all m :java_method |
                m calls subscribe("menu.contribution.request", *, *)
                OR m.name = contributeMenu OR m.name = UIDriver.populateMenus
                OR m calls sendMessage("menu.contribution", *, *)
}
```

(b) Aggregate population examples

Figure 5: Aggregate Type Declaration and Population Examples

```
<Aggregate type="java_class" id="23">
  <Field name="name"><String>project.util.StringMap</String> </Field>
  <Field name="package">
    <Semantic type="Text" label="java_package_decl" id="27">
    package project.util;
    </Semantic></Field>
  <Field name="imports"><Set type="java_import_decl">
    <Semantic type="text" label="java_import_decl" id="42">
    import java.util.Map;
    </Semantic>
    <Semantic type="text" label="java_import_decl" id="43">
    import java.util.HashMap;
    </Semantic>
  </Set></Field>
  <Field name="decl">
    <Semantic type="Text" label="java_class_decl" id="29">
    class StringMap implements Map extends HashMap
    </Semantic>
  </Field>
  <Field name="members">
    <List type="java_member">
      <Semantic type="text" type="java_member" id="48">
        public void putString(String key, String value) { ... }
      </Semantic>
      <Semantic type="text" type="java_member" id="49">
      public void getString(String key) { ... }
      </Semantic>
    </List>
  </Field>
</Aggregate>
```

Figure 6: Example of XML Externalization Format in Stellation

```
package project.util;
import java.util.Map;
import java.util.HashMap;
/*+id=29*/
class StringMap implements Map extends HashMap {       /*+id=48*/
   public void putString(String key, String value) {      ... }
/*+id=49*/
    public void getString(String key) { ... }
}
/*+AGGINFO
<Aggregate type="java_class" id="23">
    <Field name="name" type="String"  value="project.util.StringMap"/>
    <Field name="package" type="java_package_decl">package project.util       </Field>
     <Field name="imports" type="java_import_decl Set">
        <member id="42">import java.util.Map</member>
        <member id="43">import java.util.HashMap</member>
     </Field>
     <Field name="decl" type="java_class_decl" id="29"/>
     <Field name="members" type="java_member List">
          <memberref id="48"/>
          <memberref id="49"/>
      </Field>
</Aggregate>
```

Figure 7: An example of a final externalized form in Stellation

port/export tooling for several common formats.

In order to support round-trip operations involving non-XML external formats, we expect that developers will use external formats containing marker tags which identify the boundaries between areas corresponding to distinct XML elements. Figure 6 illustrates our standard XML externalization format, and 7 illustrates a possible externalized Java format produced using an XML transformer. This file is standard java syntax, with the addition of markers and a file trailer to support round-trip operations. It is easy to write a script that translates from this format to a simple XML format, which is then transformed back into Stellation aggregate form using XSLT tools.

4.2 Using VSC and SCM for Concern Identification

The versioning history of fine-grained artifacts stored in a VSC supporting SCM system can be used to identify concerns. The work on this use of fine-grained SCM information is still preliminary, but we believe it has great potential. The basic concept is that it is possible to identify patterns from the development history of the system, and to use those patterns to infer information about the structure of the system and relationships between program artifacts.

Concern discovery can be performed using this mechanism in several ways, including at least the analysis of change co-occurence and pattern similarity recognition. Change association analysis is based on the observation that related artifacts tend to change together. When a set of artifacts tend to consistently change together through the development of the system, it is likely that they are members of a common concern. Pattern similarity recognition is a notion based on the work of Griswold et al on information transparency[18], which is based on the property that code addressing a particular concern will often exhibit a particular pattern: by recognizing recurring patterns in code, a system can infer potential relationships between artifacts that exhibit those patterns. These kinds of inference mechanisms can be implemented using data mining and machine learning techniques. While these techniques could have been applied using standard SCM systems, this would not have been useful for aspect discovery, because it would only be able to identify relationships between complete source files, rather than between collections of methods. The increased precision made possible by fine-grained storage makes this techique significantly more powerful in general, and in particular, makes it useful in the context of aspect-identification. The remainder of this section focuses on the change association analysis that is used for aspect-identification.

Change association between artifacts can be discovered by applying association rule mining[1] on repository versioning data. Association rule mining is a technique for the so-called *market basket problem.* Intuitively, market basket data corresponds to a database of transactions in retail organizations. The goal of the technique is to discover rules of the form "when a customer purchases a set of items X in a transaction, they are likely to also purchase a set of item Y in the same transaction." This can be applied to program history by considering an atomic change unit (a single semantic change spanning multiple artifacts) in the history as a transaction containing the changed artifacts. In this setting, association rules generated are of the form: "when a programmer modifies a set of artifacts X in a change, they will likely also modify artifact Y in the same atomic change.". If the change association is strong enough, then we can infer that there is a likelihood of a concern relationship between X and Y.

To demonstration a concern relationship that could be inferred through these techniques, we will discuss examples taken from the current Stellation system source code. Stellation uses a relational database for storage. The system is based on an extension architecture, where components are dynamically assembled into a running system. Extensions can create and manipulate tables in the database, and they can also manipulate both the intrinsic basic database tables used by the system core, and by other extensions which they depend on. The implementation of a particular extension touches on many different concerns: database storage, history, logging, caching, retrieval, indexing and analysis.

When an extension is modified, there are certain change relationships that tend to hold. An example of such a relationship is that when a method involved in the storage concern is modified, the other methods will tend to change as well, because a change to how data is stored in the database may affect the other methods that access that database. Another example is that when code in-

volved in the indexing and analysis concern is modified, code that uses the analysis results for retrieval will frequently change. Both of these change patterns are indicative of concern relationships. We believe that this technique is particularly useful because it can discover concern relationships that would be difficult to discover using semantic analysis techniques. The two examples we cited above are both relationships that involve modifications to code written as literal strings passed to JDBC database access methods. Such relationships could not be derived from conventional program analysis without specifically implementing a JDBC/SQL source code analyzer.

5. SUPPORTING COMPOSITIONAL AOSD WITH SCM

As we discussed earlier, the use of VSC tools is not a replacement for compositional aspect-oriented programming languages and metalanguages. For the implementation of large, complicated software systems, the greater power of compositional techniques is necessary.

In such a situation, SCM-based visual separation of concern tools can be combined with compositional systems. Such a combination provides powerful capabilities, including reducing scatter-related difficulties, maintaining history of the evolution of the system and its concerns, and allowing programmers to store metadata reflecting the connection between program artifacts and artifacts from other stages of the software process.

We will describe two examples of the synergy between VSC and compositional AOSD systems: persistent concern mapping, and scatter reduction.

5.1 Concern Mapping

In the HyperJ system, concerns are mapped into a multidimensional hyperspace. Each dimension of a hyperspace corresponds to a particular set of concerns and criteria for separation. A key step in building a system using the HyperJ module is the process of mapping concerns into their appropriate positions in the hyperspace, and mapping program artifacts into the appropriate concerns. This process is called *concern mapping*. This is done using a mechanism for specifying the program artifacts that belong to a concern using a query-like mechanism. Concern mapping is also a fundamental concept of the Cosmos model of Sutton and Rouvellou[19], but Cosmos goes further, and extends the notion of concern mapping to include artifacts from different phases of the development process, and also to maintain relationships between different concerns as they move through the process.

Concern mapping can be combined with a VSC SCM system by representing the concern maps using aggregate structures, and using the concern selection language as a fragment selection mechanism in the SCM system. Doing this provides a versioned form of a concern map similar to the Cosmos model, but has several advantages over the use of independent concern mapping tools:

- It is simple for programmers to experiment with different specifications for concern mapping, and to easily view and judge the results.
- The concern map is versioned by the SCM system, allowing programmers to see how the set of concerns and the way that the mapping of program artifacts onto concerns has evolved over time.
- The concern map can store both program artifacts and artifacts from other stages of the software process. For example, programmers can map both Java code and UML design diagrams into a common concern map, and see the correspondence between concerns in the design and the implementation. This makes the concern map of a Cosmos model into something tangible that can be viewed, manipulated, and used as a guide for performing development tasks.

5.2 Combating Scatter

As we discussed in section 2, using compositional AOSD can introduce difficulties involving the introduction of unexpected semantic elements, and scattering of control flow. To understand a system implemented using compositional AOSD, the programmer must know about all of the aspects that make up the system, and how they will be composed. This issue has been recognized by other tool builders, and systems to assist aspect-oriented software development have provided ways of seeing where code will be composed. For example, the AspectJ development tools for Eclipse[11] provide a visual indication in an outline view when a method is the target of a piece of advice code.

By using a VSC tool, programmers can get a much better view of the system, and how advice will be integrated. For example, when viewing a system implemented using AspectJ, the system can automatically generate views that co-locate methods and the selections of advice that will be composed with those methods. From the converse point of view, the system can use a join-point as a selection expression, and generate a view illustrating what pieces of code will be affected by a given aspect or a given fragment of advice.

Using such a facility, the impact of the scatter effect on the comprehensibility of the system can be dramatically reduced. The programmer can understand exactly what aspects will affect a given piece of the system, and what parts of the system will be affected by a particular aspect.

6. RELATED WORK AND OPEN QUESTIONS

Many of the basic techniques involved in the visual separation of concerns is based on existing technologies.

The basic idea of fine-grained storage was explored by other systems including COOP/Orm[2] and ENVY[24], but neither of these systems allowed dynamic custom view generation. Fine-grained storage with some degree of dynamic views originated in environments like Smalltalk80[16], and the expanded functionality of dynamic views based on queries has been explored both the CMU Gwydion project[28], and the Desert environment[27, 21].

The aggregate mechanism used by Stellation bears some relation to the type specification language of Adele[14, 3]. In addition to the structuring and merge management features that we provide, Adele's types have significantly richer semantics for managing aggregate object behavior, history, and interactions.

The scatter problem has also been addressed by others. Murphy et al[6, 26] have done a variety of work to help identify scattered code related to a concern, and have built tools and proposed language extensions to reduce the impact of scatter. Griswold[17, 18] has proposed a programming technique called *information transparency* and implemented supporting tools that use pattern recognition in programs to identify code related to a scattered concern, and gather that code together into a single view. This work is similar to ours both in using views to reassemble scattered concerns, and in using pattern-recognition to discover those concerns. We believe that the pattern recognition technique used by Griswold can be enhanced by the use of the machine-learning techniques we proposed in section 4.2.

We believe that our visual separation of concerns model produces a significant step forward from the work of earlier systems, both by providing richer functionality, and by combining existing

functionalities in new ways. However, there are several open questions that we plan to address, both in the fundamental VSC functionality, and in the realm of new features that can be provided using VSC.

1. **What are the best ways of expressing desired selections of artifacts?** We currently use a query language, and will likely support the HyperJ pattern matching language. But there are a wide range of methods that programmers may want to use to select the set or artifacts to present in a view. We plan to explore alternatives, and provide a variety of mechanisms for artifact selection.

2. **What kinds of user interface support are needed to provide the maximum benefit of visual concern separation?** VSC can be a powerful aid in combating scatter, but for it to yield its promise in this area, it must be possible to trivially shift between different views containing the same artifact in order to properly understand that artifact in context. Discovering the correct UI paradigm for presenting this and other capabilities of VSC is quite important.

3. **Can VSC be used as a communication tool between members of large development teams?** One of the fundamental goals of the Stellation project has been supporting collaborative software development. We believe that VSC can be a powerful tool for allowing one developer to communicate her viewpoint on the code to another, or to present a viewpoint that illustrates the cause of a bug.

4. **What other features of a system can be discovered through the machine-learning/data mining history approach described in section 4.2?** We believe that this approach is useful for at least concern discovery, predictive impact analysis, and predictive error analysis. But we also believe that as an extremely new technique for taking advantage of the information available in the history of a system, there are likely to be other problems that can be approached through this technique.

7. CONCLUSIONS

In this paper, we presented a new model of concern separation called the visual separation of concerns (VSC). VSC allows concern separation by changing the basic storage model of the system, storing programs as fine-grained artifacts rather than source files, and allowing programmers to dynamically build views composed from collections of fine-grained elements. The VSC model thus allows concern separation without introducing new language constructs, languages, or modifications to the semantics of programming languages.

We described some of the problems of existing approaches to concern separation, including the *scatter* problem, and described how VSC can avoid and even counteract scatter. We also described how VSC systems can work synergistically with existing compositional systems.

We also discussed some new research directions enabled by the the VSC model, including improved concern mapping, and concern discovery through development history analysis. Finally, we discussed some open issues to be addressed by future work.

8. REFERENCES

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[2] B. Magnusson and U. Asklund. Fine grained version control of configurations in COOP/Orm. In *ICSE '96 SCM-6 Workshop*, pages 31–48, 1996.

[3] N. Belkhatir, J. Estublier, and W. Melo. Adele 2: A support to large software development process. In *Proceedings of the 1st International Conference on the Software Process*, 1991.

[4] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[5] L. Bergmans, M. Aksit, and B. Tekinerdogan. *Software Architectures and Component Technology: the State of the Art in Research and Practice*, chapter Aspect Composition Using Composition Filters, pages 357–382. Kluwer, 2001.

[6] Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. Explicit programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 10–18. ACM, 2002.

[7] M. Chu-Carroll and S. Sprenkle. Coven: Brewing better collaboration through software configuration management. In *Proceedings of FSE 2000*, 2000.

[8] M. C. Chu-Carroll. Supporting distributed collaboration through multidimensional software configuration management. In *Proceedings of the 10th ICSE Workshop on Software Configuration Management*, 2001.

[9] Mark C. Chu-Carroll. Separation of concerns: an organizational approach. In *Proceedings of the OOPSLA 2000 Workshop on Advanced Separation of Concerns*, 2000.

[10] Mark C. Chu-Carroll, James Wright, and David Shields. Supporting aggregation in fine grained software configuration management. In *Proceedings of SIGSOFT FSE 10*, 2002. To appear.

[11] The Stellation project homepage. Webpage at "http://www.eclipse.org/stellation".

[12] Eclipse platform technology overview. Technical report, OTI, Inc., July 2001.

[13] James Clark (editor). XSL transformations (XSLT) version 1.0. W3c recommendation, W3C, November 1999.

[14] J. Estublier and R. Casallas. *Configuration Management*, chapter The Adele Configuration Manager. Wiley and Sons, Ltd., 1994.

[15] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedike. Viewpoints: a Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engienering*, 2(1):31–58, March 1992.

[16] A. Goldberg and D. Robson. *Smalltalk 80: the Programming Language*. Addison Wesley Longman, Inc., 1989.

[17] W. G. Griswold, Y. Kato, and J. J. Yuan. AspectBrowser: Tool Support for Managing Dispersed Aspects. Technical Report CS99-0640, UCSD, Departmentof Computer Science and Engineering, 1999.

[18] William G. Griswold. Coping with crosscutting software changes using information transparency. In *LNCS 2192: Proceedings of Reflection 2001*, pages 250–265. Springer Verlag, 2001.

[19] S. Sutton Jr. and Isabelle Rouvellou. Modeling of software concerns in cosmos. In *Proceedings of the 1st international*

conference on Aspect-oriented software development, pages 127–133. ACM, 2002.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP*, June 1997.

[21] Y. Lin and S. Reiss. Configuration management with logical structures. In *Proceedings of ICSE 18*, pages 298–307, 1996.

[22] O. Lerhmann Madsen, K. Nygaard, and B. P. Miller. *Object-Oriented Programming in the Beta Programming Languge*. Addison-Wesley, 1993.

[23] 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*. Kluwer, 2000.

[24] OTI. ENVY/Developer: The collaborative component development environment for IBM visualage and objectshare, inc. visualworks. Webpage: available online at: "http://www.oti.com/briefs/ed/edbrief5i.htm".

[25] Dave Pawson. An introduction to XSL formatting objects. Webpage at "http://www.dpawson.co.uk/xsl/sect3/bk/index.html", 2001.

[26] M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependenceis. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, 2002.

[27] S. Reiss. Simplifying data integration: the design of the Desert software development environment. In *Proceedings of ICSE 18*, pages 398–407, 1996.

[28] R. Stockton and N. Kramer. The Sheets hypercode editor. Technical Report 0820, CMU Department of Computer Science, 1997.

[29] R.E. Filman T. Elrad and A. Bader (editors). Special section on Aspect Oriented Programming. *Communications of the ACM*, 44(10):28–97, October 2001.

[30] P. Tarr, W. Harrison, H. Ossher, A. Finkelstein, B. Nuseibeh, and D. Perry, editors. *Proceedings of the ICSE2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, 2000.

[31] P. Tarr, H. Ossher, W. Harrison, and Jr. S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, 1999.