Supporting Aspects in Program Storage

Mark C. Chu-Carroll
IBM T. J. Watson Research Center
mcc@watson.ibm.com
September 19, 2002

Abstract
The fundamental abstractions used by the majority of programming languages to store computer programs has not changed significantly in 40 years. Software configuration management systems reflect this, forcing a unidimensional file-based decomposition of the system for storage purposes. By applying the same techniques of indirection that have driven programming design to program storage, an SCM system can provide a collection of facilities that I call multidimensional software configuration management, which both resembles and supports aspect-oriented software development. In this paper, I will describe Stellation, a new system under development at IBM Research, and how it supports multidimensional software configuration management.

1 Introduction
The history of program language design can be viewed as a process of adding layers of indirection into programming. Programs started out in the early days of FORTRAN and COBOL as monolithic bodies of code. Structured programming came along, and introduced the idea of separating out chunks of a system as procedures and giving them names. Object orientation went further, and added indirection to the names, so that a name represented a collection of alternative procedures, from which one would be selected dynamically during execution. Aspect-oriented software development goes even further, turning the calls to dynamically bound procedures into named entities.

In the meantime, the ways in which we view, store, edit, and manage source code has barely changed from the days of punched cards. Programs started out as stacks of cards, and made the leap to named source files containing lists of lines. But from there, progress essentially stopped: the process and basic tools for actually writing code have not changed significantly in forty years!

Our project at IBM is trying to change that. The focus of our work is a system called Stellation, which is a software configuration management system tightly integrated with the Eclipse programming environment. Stellation discards the notion of source files as a storage mechanism, and adds the kinds of indirection that have revolutionized programming languages to the tools that manage program artifacts. By doing so, mechanisms very similar to aspect-oriented programming become available to tools throughout the software process, enabling a wide variety of novel capabilities, including the integration of aspect-oriented programming itself.

In this paper, I will briefly describe the basic mechanisms of Stellation, and how they relate to aspect-oriented software development. A more detailed overview of an early version of Stellation can be found in [1], and a more detailed overview of the Stellation aggregate mechanism will appear in FSE 2002.

2 Multidimensionality in Stellation
The basic feature of Stellation that enables aspect-oriented features for code manipulation is what I call multidimensionality. Multidimensionality is the ability to view and manipulate code using multiple overlapping viewpoints. Each viewpoint presents code in a different organizational structure, representing a different decomposition into concerns, or a different organizational or functional relationship between program artifacts.

Multidimensionality is enabled by a combination of fine-grained storage, flexible metadata constructs for annotating program artifacts, mechanisms for creating versioned aggregate structures, and dynamic queries for creating and populating new viewpoints. With these features, multidimensionality presents a functionality quite similar to aspect-oriented programming tools like HyperJ[5].

In particular, multidimensionality enables the visual separation of concerns. Similar to the generation of hyperslices in HyperJ, Stellation allows dynamic creation of program views that present a cross-cutting concern as a single source-file like view. The query expressions for...
Figure 1: Fragments in Java

describing these views strongly resemble the expressions used to generate hyperslices in HyperJ, or join-point specifiers in AspectJ.

In the rest of this paper, I will go into greater detail describing the facilities of Stellation that enable multidimensionality, and how they can support the separation of concerns and aspect-oriented software development.

2.1 Fine-grained storage.

Instead of having the code organization of the system dominated by its layout in source files for storage, programs and related development artifacts are stored as small, fine-grained units called fragments. The exact granularity of storage is language specific: the smallest self-contained unit of code in the language. For example, in Java, the size of a fragment is a method or field declaration, as illustrated in figure 1.

In Stellation, all storage is in terms of fragments. The use of fragment-based versioning allows us to separate storage from organization. By managing code in terms of fragments, Stellation is capable of providing a wide variety of fully equal views of the system, each suited toward a different purpose or viewpoint. This multiple viewpoint facility resembles some early work related to aspect-oriented software development[4,2], and can be used for providing a way of visually identify, separating, versioning, and manipulating aspects or concerns in a software system.

2.2 Aggregation

Aggregation is our mechanism for combining groups of fragments into larger structures. This aggregation facilities is required for a fragment based SCM system. Based on experimental data, using fragment granularity for artifacts increases the number of artifacts managed by the system by between one and two orders of magnitude. Some mechanism is required for managing this increased number of artifacts without overwhelming the programmer with complexity.

The simple need for a tool for managing the large number of artifacts is not the only motivation for our aggregation system; in fact, aggregation is the key that enables many of the particularly powerful capabilities of Stellation. Aggregation allows Stellation to represent dynamically generated collections of source code that is presented in a source-file like form. It also allows the generation of views that integrate data from multiple sources or multiple phases of the software process.

Aggregates also represent an extremely powerful metadata facility for representing relationships between software artifacts. For example, an aggregate can 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.

Finally, aggregates form the basic data structure used by Stellation itself for providing a variety of facilities for supporting collaboration. For example, an aggregates can represent the set of artifacts placed under a lock for coordination between programmers.

The Stellation aggregate system allows programmers to define types of aggregates to represent different data structures. An aggregate definition looks like a type definition in a programming language. An aggregate is a set of named slots, each of which declares a type and a merge operator. (For details on merge operators, see [?].) Slot types are summarized in figure 2a. A set of sample aggregate type declarations are illustrated in figure 2b.

2.3 Metadata Properties

In order to allow Stellation to correctly and rapidly identify the program artifacts that belong to a particular viewpoint or relationship, the system requires some mechanism for allowing programmers and other tools to annotate artifacts with arbitrary metadata.

Stellation does this using a simple property list mechanism: each artifact in the repository has an associated list of named properties, whose values are essentially arbitrary sized blocks of text. (It is our expectation that the majority of metadata in the system will be stored in XML form.)

Metadata properties range from semantic summaries of program fragments that will be used for information retrieval over the program repository, to links to other related artifacts, to programmer annotations describing their intentions for future changes to an artifact, to system data like artifact modification times.
B = new bug_report {
    title = "thread leak"
    severity = 5
    description = "Messaging threads not being terminated"
    subject_code = all x :java_fragment |
            x creates DeliveryThread
            OR x invokes DeliveryThread.run OR
            (exists y : java_fragment |
                  field(x) AND waits(y, x))
            OR x = Engine.shutdown

    test_data = TestCase2, TestCase5, TestCase13
}

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", *, *)
}

Figure 3: Example of query based aggregate population

2.4 Dynamic Selection and Source Queries

To take advantage of the aggregate mechanism, both users and tools require some mechanism to dynamically create and populate aggregate objects. In order to do this, Stellation provides a query language, and allows programmers to populate the fields 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 backtracking predicate programming language like Prolog.

Some examples of query-based aggregate generation can be seen in figure 3. Example one illustrates how our mechanism for fine granularity combined with typed aggregates allows the SCM system to create versioned artifacts representing concepts like software bug reports, tying together the bug report, the code that is likely to be the source of the bug, and the test cases that illustrate the bug. By reifying this as a versioned artifact, the system history can now show the introduction of the bug, the evolving history of how the programmers narrowed down its cause, and the final changset that corrected the bug.

The second example illustrates how our aggregate mechanism creates source-file-like aggregate views that present a cross-cutting concern in a single source file. This example comes from a real software system - the predecessor of the Stellation system. In this system, operations were performed using an asynchronous messaging system. 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 small methods generating contributions from approximately 15 classes. In a conventional SCM system, that required the programmer to trace code through 15 different files! In Stellation, the programmer can select this 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 extremely similar to AspectJ’s join point specifiers[3], as seen in these examples.

3 Supporting AOSD in Stellation

In addition to the synergy between Stellation’s multi-dimensional functionality and AOSD programming languages, there is a simple way of implementing languages for aspect-oriented program composition as Stellation extension components operating as aggregate to aggregate transformers, with the source aggregates populated using Stellation queries.

For example, to implement an AspectJ-like language in Stellation, one can define an aggregate type consisting of a slot containing a join-point specifier (which acts as a
SlotType = AtomicType | CollectionType
AtomicType = PrimitiveType | SemanticType | UnionType
PrimitiveType = Integer | String | Text | Binary
SemanticType = language specific atomic type (e.g., java_methoddecl)
UnionType = a set of atomic types
CollectionType = Set of AtomicType | List of AtomicType

(a) Aggregate slot types summary

aggregate java_class {
    name : [conflict] String
    package : [conflict] java_packagedecl
    imports : [union] java_importdecl
    decl : [conflict] java_classdecl
    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
}

(b) Aggregate type declaration examples

Figure 2: Aggregate Slot Types

query selecting the fragments where the advice will be inserted; a slot specifying the type of advice (before, after, or around); a slot containing the advice to be composed into the target code; and a slot containing the set of fragments selected by the join point specifier.

In this scheme, a complete aspect would be a construct much like the java_class aggregate in figure 2b, containing a set of individual composition aggregates, and java declarations.

4 Conclusion

In this paper, I presented a brief overview of multidimensional software configuration management in Stellation. By applying the same principle of indirection that led programming languages to aspect-oriented programming, Stellation can introduce new aspect-like features to software configuration management. These features enable powerful new capabilities for software configuration management, including multiple overlapping source organizations/views, visual separation of concerns, explicit fine-grained relationship representation, tighter integration of the system with other tools such as bug management, and better communication between programmers. In addition, the resulting system is capable of providing extremely flexible support for aspect-oriented software development, including the ability to combine the SCM system with the aspect compiler/software composition system.

References