RC 22130 (99123) July 30, 2001 **Computer Science**

IBM Research Report

The Shape of Things to Come: Using Multi-Dimensional Separation of Concerns with Hyper/J to (Re)Shape Evolving Software

Harold Ossher, Peri Tarr
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

The Shape of Things to Come: Using Multi-dimensional Separation of Concerns with Hyper/J to (Re)Shape Evolving Software

Harold Ossher
Peri Tarr
IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, New York, 10590
{ossher, tarr}@watson.ibm.com

1. Introduction

Separation of concerns [11] is a key guiding principle of software engineering. It refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose. Concerns are the primary criteria for decomposing software into smaller, more manageable and comprehensible parts that have meaning to a software engineer.

As software becomes more pervasive and its life expectancy increases, it becomes subject to greater pressures to *integrate* and *interact* with other pieces of software—often off-the-shelf software that has been written by entirely separate organizations—and to *evolve* and *adapt* to uses in new and unanticipated contexts, both technological (e.g., new hardware, operating systems, software configurations, standards) and sociological (e.g., new domains, business practices, processes and regulations, users). When the concerns a software engineer has in a particular context are ones that have been identified and *encapsulated*, evolution is simpler and less costly—changes are localized and easier, and the impact of change is smaller. Reuse is facilitated because developers can reuse exactly what they need and not be burdened with extraneous parts, which might be costly or incompatible with the reuse context. Integration is also simplified, because developers need only address the relevant interactions among components. Thus, while separation of concerns to aid initial development is important, as it allows developers to manage software complexity, separation of concerns to promote evolution, integration, and reuse is even more critical, as the majority of software engineering effort is expended on these activities.

Much of software evolution, reuse and integration are of an *unanticipated* nature. This is not necessarily because of poor design, but rather because the world is changing so fast that it is impossible to predict the paths of software evolution and use with accuracy. It is not even possible to predict exactly which concerns will arise or become important during the lifetime of a system. It *is* certain that different kinds of concerns will be relevant to different developers in different roles, with different goals and tasks, and/or at different stages of the software lifecycle.

This paper describes an approach called *multi-dimensional separation of concerns* (MDSOC), which addresses these issues. It permits effective encapsulation of arbitrary kinds of concerns simultaneously, even when the concerns were not anticipated originally, and the integration of separate concerns. We illustrate the properties and use of MDSOC with a running example, introduced in Section 2, where we also discuss the relationship to AOP. Sections 3 and 4 explore evolution scenarios, and show how Hyper/J™, our system for supporting MDSOC for Java™ developers, facilitates them. Section 5 briefly covers related work, and Section 6 presents some conclusions.

2. Running Example: Getting Personal with Personnel Software

Our running example is based on some evolutionary scenarios involving a personnel system for a large, international organization. The software includes a variety of features, which could be implemented as separate tools or applications, for use by different branches of the organization. Initially, these are:

- The personnel feature, which manages basic information about employees, such as name, ID, and management chain. It enforces certain business rules, such as a requirement for each employee to have at least one and at most three managers.
- The payroll feature, which maintains salary and tax information. It also enforces some business rules, such as minimum and maximum wage requirements and tax regulations.

Figure 1 depicts these features, implemented using standard object-oriented technology. A class is used to represent each kind of employee, and each feature is implemented as one or more methods defined on those classes.

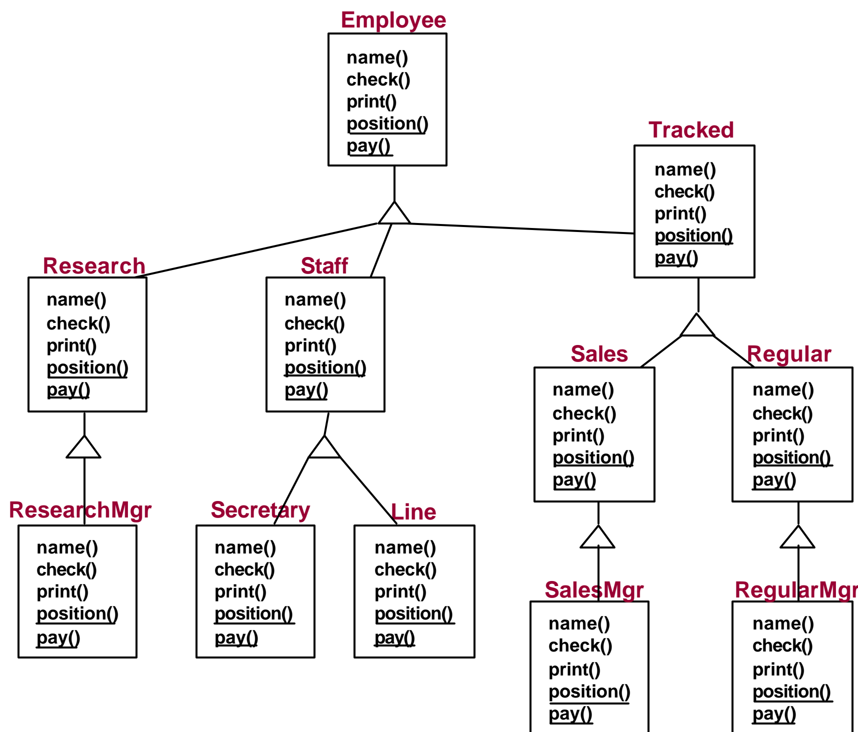

Figure 1: Employee class hierarchy. Underlined methods belong to the Payroll feature, while others belong to the Personnel feature.

This software reflects many different kinds of concerns, and each is relevant to different people, at different times, with different tasks. Each kind of employee is a "data concern," or object. Personnel and payroll are "feature concerns," which tend to reflect end-user concerns and perspectives. Each business rule is a "business rule concern," which often reflects management, business process, and semantic consistency issues. These are the only concerns we will discuss in this example, but there are usually many more, such "variant concerns," which address, for example, different ways of computing taxes in different jurisdictions, or "systemic (or non-functional) concerns," such as distribution and transaction management.

The object-oriented paradigm allows the data concerns—the employees—to be encapsulated within classes, so that all the software associated with each data concern is localized. The other kinds of concerns, however, cannot be represented effectively in the object-oriented paradigm. We have called this phenomenon the *tyranny of the dominant decomposition* [12], because one dominant way of decomposing the program—by class—imposes a structure on the software that makes it difficult or impossible to encapsulate other kinds of concerns—like features and business rules—effectively. The tyranny results in problems associated with poor separation of concerns—difficult, costly evolution, low reuse, complicated integration, and brittle software—because the software does not, in fact, separate the necessary concerns.

Multi-dimensional separation of concerns (MDSOC) [10,12] addresses this problem. It lets developers decompose their software so that it encapsulates *all* of the relevant kinds (*dimensions*) of concerns *simultaneously*, without one dominating the others. MDSOC also includes a powerful composition capability, to allow developers to integrate these separated pieces. Our approach to achieving MDSOC is called *hyperspaces*, and its realization for Java™ is Hyper/J™ [10]. A key goal for Hyper/J was not to modify or extend the Java language. Instead, Hyper/J supports MDSOC for *standard* Java software, developed using any methodologies and tools. Hyper/J operates on "class files," not source, allowing extension, extraction, adaptation and integration of off-the-shelf binary Java components.

A key difference between MDSOC with Hyper/J and AOP as described in the literature [7] and exemplified by AspectJ [6], is that AspectJ supports augmentation of a *single model*, whereas Hyper/J supports integration of *multiple models*. In AspectJ, one starts with a distinguished "base" hierarchy—"the model"—and uses separately-coded *aspects* to augment its classes and methods. This is especially powerful when a single aspect cuts across many classes, allowing a single, localized specification of scattered behavior. The fact that the model dictates the structure makes specification of the aspects convenient, but introduces limitations. Aspects augment classes, but cannot augment one another, so aspects are not composable. Aspects often cannot be understood without reference to the model, which may limit their reusability. Also, all aspects in a system are coded relative to the same model. In reuse and integration situations, however, one usually does not have a single model across all components: separately-developed reusable components employ whatever domain model is most appropriate for their purpose, so they must be adapted to the reuse context, and differences between components being integrated must be reconciled. Even when extending a system by adding a new feature, it is sometimes valuable to use a different model specifically tailored to that feature, as illustrated in Section 4. Hyper/J allows a developer to compose a collection of separate models, called *hyperslices*, each encapsulating a concern by defining and implementing a (partial) class hierarchy appropriate for that concern. The models typically overlap, and might or might not cut across one another. Each model can be understood in isolation. Any model can be augmented by composition with others: Hyper/J does not require a distinguished base hierarchy, and makes no distinction between "classes" and "aspects," allowing any hyperslices to extend, adapt and be integrated with one another as needed. Crosscutting behavior can be specified by composing a single method in one model with multiple methods in another. This symmetric treatment of all concerns in MDSOC is a key feature, promoting evolution, integration, and reuse.

Another unique feature of MDSOC is its ability to handle multiple decompositions of the same software simultaneously: some developers can work with classes, others with features, others with business rules, variants, etc., even though these carve up the system in substantially different (though overlapping) ways. New decompositions can be introduced non-invasively as needed during the software lifecycle. Other approaches that provide flexible means of decomposing systems still support only one (or few) decomposition(s) at a time, determined when the software is written.

The rest of this paper demonstrates some uses of MDSOC with Hyper/J by showing how it addresses some common, and generally problem-laden, evolution scenarios. These scenarios illustrate concretely some of Hyper/J's distinguishing features.

3. Separation Anxiety: Non-Invasive Separation of Concerns in Retrospect

Suppose the developers of the personnel system are approached by a different organization, seeking similar software, but with some different requirements:

- They need not maintain payroll information, since a subcontracted organization handles this. They cannot afford the now-irrelevant payroll feature as baggage within their system.
- They have some different business rules. In particular, each employee must have exactly one manager.

The first requirement suggests the need to *remove* the payroll feature from the system. Moreover, it suggests the need to *mix-and-match* the payroll feature, since some clients want it while others do not. The second requirement indicates a need to *modify* the existing well-formedness constraints. The fact that the two organizations have different constraints suggests the desirability of treating each organization's constraints as customer-specific business rules, which can be non-invasively modified or replaced as needed for new customers.

It is extremely difficult to accomplish these sorts of evolutionary changes using standard object-oriented technology (including design patterns), and doing so would require major, *invasive, non-local* changes. This is because the concerns these changes affect—the payroll feature and the business rules—were not encapsulated, because they did not align with the dominant decomposition, the class hierarchy.

One could argue that a savvy developer would have anticipated this type of change and would have enabled it through the judicious use of design patterns, like Visitor. Even the best developer, however, cannot anticipate every kind of change, for reasons noted earlier. Moreover, even if (s)he could, it would be undesirable to encapsulate every potentially useful concern or provide every possible extension hook: the cost of doing so would be prohibitive, both in terms of added complexity and reduced runtime performance. This kind of evolution scenario—where a

new kind of concern, not previously encapsulated, is needed to effect a particular kind of change—is, therefore, common and important.

3.1. Removing a Scattered Feature, and Adding Mix-and-Match in Retrospect

Hyper/J permits developers to identify and *non-invasively* encapsulate new concerns at any time, including concerns that affect and are scattered across, and tangled within, existing software. We call this capability *on-demand remodularization*: the ability to add new modularizations as needed to reflect new concerns, without disturbing any existing modularizations.

The first step of on-demand remodularization is for the developer to identify those pieces of the existing software that are part of the new concern(s). Figure 1 shows that the capabilities associated with the payroll feature concern are implemented by the `position()` and `pay()` methods that occur in many of the classes in the Employee hierarchy. These methods, and the classes containing them, should all be part of the Payroll feature; the others are part of the Personnel feature. Developers express this information in Hyper/J using a *concern mapping*:

```
package Personnel:  Feature.Personnel
operation position: Feature.Payroll
operation pay:      Feature.Payroll
```

This concern mapping indicates that, by default, all members of the classes and interfaces in the Java package Personnel belong to the Personnel concern in the Feature dimension. The subsequent statements override this to say that any operation—i.e., any method, irrespective of its class—named `position` or `pay` belongs to the Payroll concern in the Feature dimension.[1]

When processed by Hyper/J, this concern mapping results in the creation of the hyperspace shown in Figure 2. Hyper/J creates a separate module, called a *hyperslice* (essentially, an encapsulated concern), for each feature, shown as rows in the figure. Each hyperslice encapsulates its own class hierarchy.[2] The classes in the Payroll hierarchy contain the `position()` and `pay()` methods, while the classes in the Personnel hierarchy contain all the other methods present in the original system. There is one exception: each class in the Payroll hierarchy also contains an *abstract* `name()` method. This is because `pay()` methods invoke `name()` methods (see Figure 2). When performing on-demand remodularization, Hyper/J makes each hyperslice *declaratively complete* by inserting abstract declarations for any members that are referred to, but not implemented within, the hyperslice [3]. Declarative completeness is a critical property. First, it means that every hyperslice represents a legal Java program that is self-contained, though not necessarily complete. Second, it means that hyperslices are loosely coupled, since they never refers directly to one another—for example, the Employee.`pay()` method in the Payroll feature refers to the local, abstract `name()` method, not to the one defined in the Personnel.Employee class. Because hyperslices do not refer to one another directly, developers can transparently replace one with another that is compatible, facilitating reuse and evolution.

---

[1] Developers can choose any names for their dimensions and concerns—there is no predefined Feature dimension.

[2] The hierarchies happen to be identical in structure here, but this need not be the case, as illustrated in Section 4.

[3] Introducing an abstract method for declarative completeness will never introduce more methods, because its body is not included. If the method's class must be instantiable within the hyperslice, a method whose body just throws a special exception is inserted instead of a true Java abstract method. Hyper/J treats such methods as abstract.

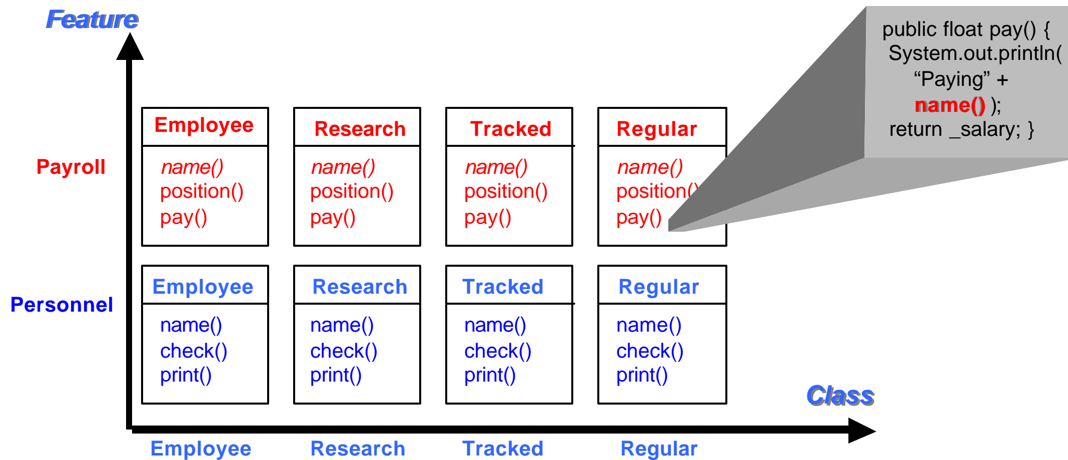

Figure 2: Hyperspace with Personnel and Payroll Concerns Separated. Italicized methods are abstract.

The Payroll and Personnel concerns are now disentangled and can be manipulated independently. The Personnel hyperslice is a complete program and can be shipped to the new customer by itself, without the Payroll concern, as the customer requested. The Payroll hyperslice, on the other hand, is a legal Java component, but it cannot run by itself—it requires implementations for the abstract `name()` declarations to be provided by *composition* with other hyperslices.

To perform composition with Hyper/J, a developer writes a *hypermodule declaration*, indicating which hyperslices are to be composed, how their various parts are related, and how the composition is to be carried out:

```
hypermodule PayrollPlusPersonnel
   hyperslices: Payroll, Personnel;
   relationships:
      mergeByName;
end hypermodule
```

The first two lines name the hypermodule and list the hyperslices to be composed. The relationships section lists all the ways in which the hyperslices are related—i.e., which parts of these hyperslices *correspond* to one another. In this case, the general relationship `mergeByName` is used. `MergeByName` is a shorthand for specifying a collection of relationships: "all entities in different hyperslices that have the same name correspond," and "merge all corresponding entities into a single new entity." Thus, for example, the classes Payroll.Employee and Personnel.Employee correspond, because they have the same name, and they will be merged into a new class, PayrollPlusPersonnel.Employee. This new class will contain all the members of Payroll.Employee and Personnel.Employee. Similarly, the `name()` methods in the two Employee classes correspond, and so the concrete method in the Personnel hyperslice will implement the abstract one in the Payroll hyperslice. The composed class hierarchy is the one we started with, shown in Figure 1.

This particular composition merely recreated the original software, as it existed before we extracted and encapsulated the Payroll concern. It is also possible, however, to compose Payroll with different concerns—ones that provide, for example, a different implementation of the `name()` method. More interesting compositions can also be accomplished; for example, where the class hierarchies do not match, or where classes in different hyperslices define different implementations of the same methods, or where crosscutting behavior specified in a single place is composed into multiple methods. Some of these will be described in subsequent sections.

This example demonstrates how Hyper/J allows developers to adapt existing software to a new context by non-invasively removing or replacing extraneous or inadequate parts, and to achieve "mix-and-match" of features non-invasively, even when the software was not originally designed for it. The new payroll concern can now be treated

as a first-class, encapsulated feature. This ability to identify, encapsulate, and manipulate new concerns in existing software is a key benefit of the MDSOC approach.[4]

3.2. Non-Invasively Replacing Customer-Specific Business Rules

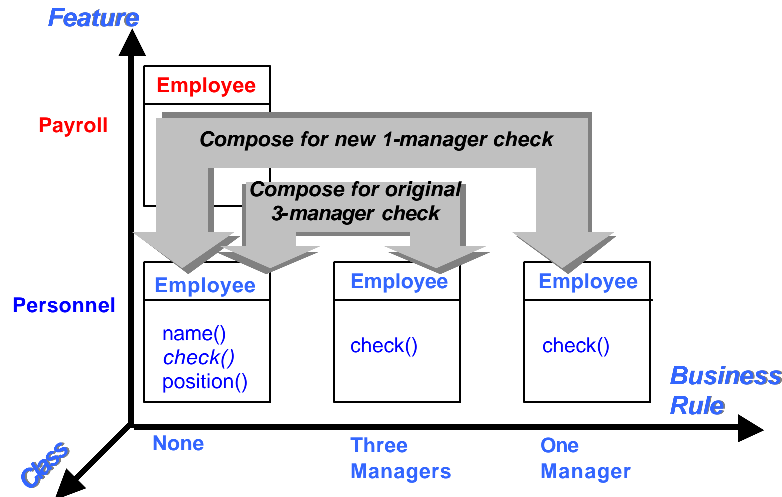

Figure 3: Feature and BusinessRule Dimensions for class Employee. Italicized methods are abstract.

Tangled concerns other than features, such as business rules, can also be separated using on-demand remodularization, after which they can be composed with, or omitted from, the software as desired. Suppose, for example, that the original business rule (which requires employees to have 1-3 managers) is implemented in the Employee `check()` method. The concern mapping

```
method Personnel.Employee.check:  BusinessRule.ThreeManagers
```

designates the existing check() method as a business rule concern, as shown in the middle column of Figure 3. The leftmost column, the "None" BusinessRule concern, contains all the code that has nothing to do with business rules. Even though the check() method is placed in the ThreeManagers concern, it is called by code in the None concern, so Hyper/J inserts an abstract declaration for declarative completeness, shown in italics.

The new customer imposes a different business rule—that each employee has exactly one manager. We can implement this alternative by defining a new `check()` method in a new Java class and package, and then mapping it to a different concern in the BusinessRule dimension, as shown in the rightmost column of Figure 3:

```
method SingleManagerPkg.Employee.check:  BusinessRule.OneManager
```

Now we can mix-and-match business rules, creating hypermodules that include either BusinessRule.ThreeManagers or BusinessRule.OneManager. It would not make sense to include both, however, since these particular rules are mutually exclusive. MDSOC permits the representation of inter-concern relationships, like this mutual exclusion relationship [10]. It is our intent to support the representation and checking of many kinds of inter-concern relationships in future versions of Hyper/J.

[4] A current limitation of Hyper/J is that it works at the *method* granularity, so its on-demand remodularization mechanism cannot disentangle concerns within a method body. We plan to address this limitation in future by incorporating a refactoring tool, which the developer can use to split methods as needed. Methods that are split thus, or written separately to start with, can then be composed to form single methods as desired.

4. Keeping Things in Perspective: Integrating Features with Different Domain Models

For the next evolution scenario, suppose the Human Resources (HR) department of the original company requests a new feature, to manage information about employee skills, evaluations, etc. Analysis of the HR domain reveals a major distinction between the HR information for managers versus non-managers, which must be reflected in the domain model. Other distinctions are secondary. For example, Sales Managers and Research Managers have much more in common, from the HR perspective, than Sales Managers and non-manager salespeople. The domain model depicted in Figure 4 is thus natural and convenient to implement the HR perspective.

As illustrated in Figure 1, however, the class hierarchy of the original personnel system was defined based on how salaries are computed: the computation is different for researchers, sales people and regular employees, but similar for managers and non-managers in each category. We are therefore confronted with a mismatch between the desired model for the HR domain and the existing, implemented domain model. Does this mean that the original model was wrong, and that we must refactor it to reflect the requirements of the new feature? No! It was, and is, correct for its purpose: modeling payroll computation. The key issue is that different situations require different perspectives and, hence, different domain models. One might argue that a more general initial domain model could have accommodated both perspectives and better facilitated evolution. That is possible, but developers often do not have the time or insight to identify a suitably general domain model up front. Also, general, flexible models are often more complicated to use for any given purpose than a specialized model customized for that purpose.

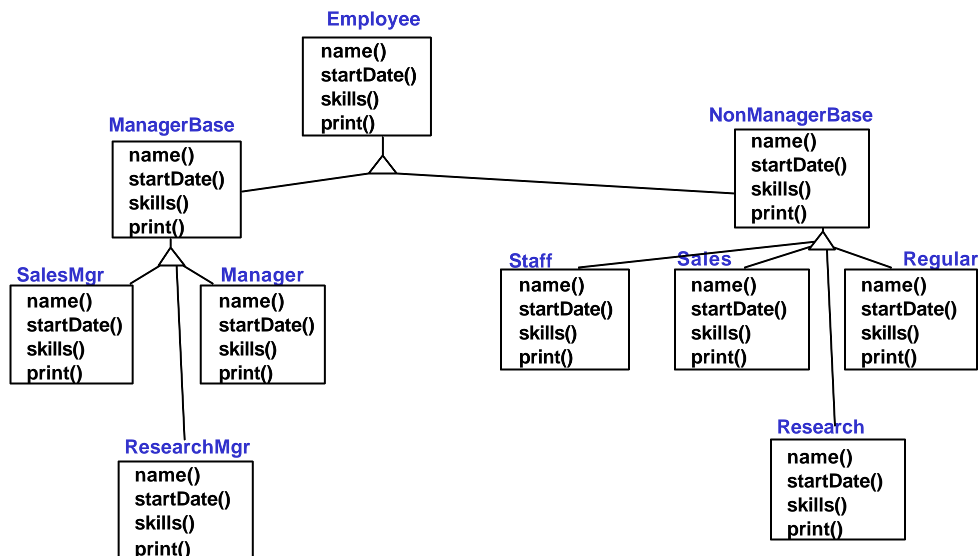

Figure 4: The Employee class hierarchy from the HR perspective.

4.1. Reconciling Different Perspectives

Using Hyper/J, the developers design and implement the new HR feature using the class hierarchy shown in Figure 4. They keep the HR feature implementation completely separate from that of the existing personnel and payroll features, by implementing them in separate Java packages. This separation both facilitates the design and implementation of the new HR feature by shielding its developers from the details of the other parts of the software, and it ensures that the HR feature can be "mixed and matched"—it can be included or excluded in different versions of the system—as desired. This new HR feature contains parts that *overlap* parts of the existing personnel feature; e.g., they both define Employee classes with `print()` methods. Representation of overlapping concerns is important, since it enables the different features to implement only those parts (e.g., of `print()`) that are relevant to them, and to integrate those implementations as appropriate. Standard Java—even with design patterns—does not support the representation and integration of overlapping concerns.

To create a new system that includes the Personnel and HR features, a developer must specify how the different perspectives relate to one another—specifically, how their classes (and their members) correspond to one another,

and how they are to be integrated. This is done using relationship specifications in a hypermodule. In this case, many of the classes, like Employee, correspond directly by name, despite having different positions in the two class hierarchies. The classes Personnel.RegularMgr and HR.Manager should also correspond, even though their names are different, because they represent the same concept. A few classes appear in one feature but not the other: for example, Personnel.Secretary and Personnel.Line have no direct correspondents in the HR feature. In all cases, corresponding entities are intended to be merged, except that the developers decide to supercede the `name()` method from the HR feature with the one from the Personnel feature because, in the integrated system, the Personnel feature should have control of employee names. The corresponding `print()` methods from the two features will also be merged, resulting in a combined `print()` that displays both Personnel and HR information. The developers define the following hypermodule:

```
hypermodule PersonnelPlusHR
   hyperslices:
      Feature.Personnel, Feature.HR;
   relationships:
      mergeByName;
      equate class Feature.Personnel.RegularMgr, Feature.HR.Manager;
      override operation Feature.HR.name with Feature.Personnel.name;
end hypermodule
```

When run through Hyper/J, this hypermodule specification causes the Personnel and HR features to be integrated to form a composed class hierarchy that integrates all of the corresponding classes and their corresponding members. The composed hierarchy is constructed to respect and preserve all of the ancestor relationships that were present in the original Personnel and HR hierarchies, and the composed methods are generated to work correctly from both perspectives. This approach incurs performance overhead only for those features actually used; traditional approaches to extension and mix-and-match, like design patterns, require "hooks" that incur some overhead even when not used.

5. Related Work

Many current and recent efforts (e.g., [1,2,3,4,6,7,8,9] and the papers in this special section) address the inadequacy of modularization mechanisms in standard software languages. Each makes unique contributions depending upon the driving domain and the problems emphasized. Some key distinguishing features of MDSOC with Hyper/J are: the ability to extract and encapsulate concerns from existing software non-invasively; the declarative completeness approach to loose coupling; the symmetric treatment of concerns; the ability to specify relationships among, reconcile mismatches between, and integrate concerns; and the applicability to standard Java software, even when source code is not available. These properties make it especially well suited to facilitating evolution, reuse and integration, as well as initial development.

6. Conclusions and Future Work

This paper introduced MDSOC with Hyper/J and illustrated its use in some evolution scenarios. MDSOC also facilitates reuse and integration: hyperslices can encapsulate collaborations, design patterns and other useful units of reuse, and composition relationships allow for adaptation of reusable components and reconciliation of differences among components to be integrated.

Further research is needed to make these uses a practical reality, however. One key issue is moving beyond code to separation of concerns across the software lifecycle, including maintenance of relationships across artifacts. Other issues include coping with tangling within method bodies, identifying and addressing issues of concern interaction and interference, and exploring the limits of reconciliation and integration.

At present, software is like *clay*: it is soft and malleable early in its lifetime, but eventually it hardens and becomes brittle. At that point, it is possible to add new bumps to it, but its fundamental shape is set, and it can no longer adapt adequately to the constant evolutionary pressures of our ever-changing world. We believe that a critical goal of software engineering is to produce software that is more like *gold*—malleable and flexible for life. We call such software "morphogenic," because it is able to adapt to new contexts by changing shape [5]. Only when software retains its ability to evolve, adapt, and reshape itself to uses in new or different contexts will we truly achieve the

benefits of good software engineering and clean separation of concerns. MDSOC represents a key advance towards this ultimate goal.

For more information: See the MDSOC web site, http://www.research.ibm.com/hyperspace. Hyper/J is available for download, free of charge, from IBM's alphaWorks web site: http://www.alphaworks.ibm.com/tech/hyperj.

Acknowledgements: We are grateful to Stan Sutton for reading and commenting on an earlier version of this paper, and to the reviewers for many valuable suggestions.

References

1. M.Aksit, L.Bergmans, S.Vural. "An object-oriented language-database integration model: The composition filters approach." In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1992.

2. Don Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components." *ACM Transactions on Software Engineering and Methodology*, October 1992.

3. Siobhán Clarke, William Harrison, Harold Ossher and Peri Tarr. "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code." In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, October 1999.

4. William Harrison and Harold Ossher. "Subject-oriented programming (a critique of pure objects)." In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA),* September 1993.

5. William Harrison, Harold Ossher and Peri Tarr, "Software engineering tools and environments: A roadmap." In *The Future of Software Engineering* (A. Finkelstein, ed.), 263–277, ACM, June 2000.

6. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. "An Overview of AspectJ." In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Hungary. Springer-Verlag. June 2001.

7. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. "Aspect-Oriented Programming." In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland. Springer-Verlag LNCS 1241. June 1997.

8. Mira Mezini and Karl Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development." In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA),* October 1998.

9. Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications." In *Transactions on Software Engineering*, vol. 20, no. 10, pp. 260-273, October 1994.

10. Harold Ossher and Peri Tarr. "Multi-Dimensional Separation of Concerns and the Hyperspace Approach." *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2001. (To appear.)

11. David L. Parnas. "On the Criteria To Be Used in Decomposing Systems into Modules." *Communications of the ACM*, vol. 15, no. 12, December 1972.

12. Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. "*N* Degrees of Separation: Multi-Dimensional Separation of Concerns." In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.