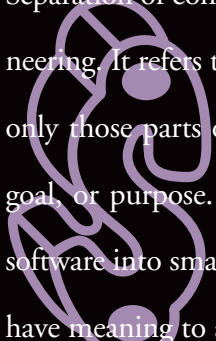


# USING MULTIDIMENSIONAL SEPARATION OF CONCERNS TO (RE)SHAPE EVOLVING SOFTWARE

## Simplifying development, evolution, and integration of Java software using Hyper/J.



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.

Harold Ossher and Peri Tarr

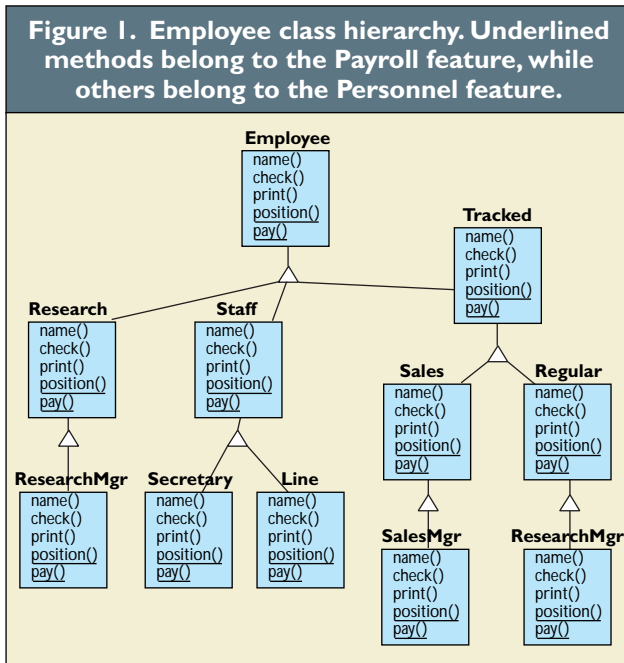
As software becomes more pervasive and its life expectancy increases, it becomes subject to greater pressures to integrate and interact with other pieces of software—often off-the-shelf software that has been written by entirely separate organizations—and to evolve and adapt to uses in new and unanticipated contexts, both technological (new hardware, operating systems, software configurations, standards) and sociological (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, sepa-

## Running Example: Getting Personal with Personnel Software

Our running example is based on some software development and evolution 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.



ration 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 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 life cycle. This article describes an approach called multidimensional separation of concerns (MDSOC), which addresses these issues and permits effective encapsulation of arbitrary kinds of concerns simultaneously—even when the concerns were not anticipated originally—and the integration of separate concerns.

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.

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 as “variant concerns,” which address, for example, different ways of computing taxes in different jurisdictions, or “systemic (or nonfunctional) concerns,” such as distribution and transaction management.

The OO 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 OO 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.

The problems involving poor separation of concerns are addressed by MDSOC [10, 12], which allows developers to decompose their software so it encapsulates all 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 a tool called 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, which allows 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 here.

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, and so forth, even though these carve up the system in substantially different

(though overlapping) ways. New decompositions can be introduced noninvasively as needed during the software life cycle. 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 remainder of this article demonstrates some uses of MDSOC with Hyper/J by showing how it addresses some common, and generally problem-laden, development and evolution scenarios. These scenarios illustrate concretely some of Hyper/J's distinguishing features.

### Separation Anxiety: Noninvasive 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 the two organizations have different constraints suggests the desirability of treating each organization's constraints as customer-specific business rules, which can be noninvasively modified or replaced as needed for new customers.

It is extremely difficult to accomplish these sorts of evolutionary changes using standard OO technology (including design patterns), and doing so would require major, invasive, nonlocal 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, such as Visitor. Even the best developer, however, cannot anticipate every kind of change, for reasons noted earlier. Moreover, even if the devel-

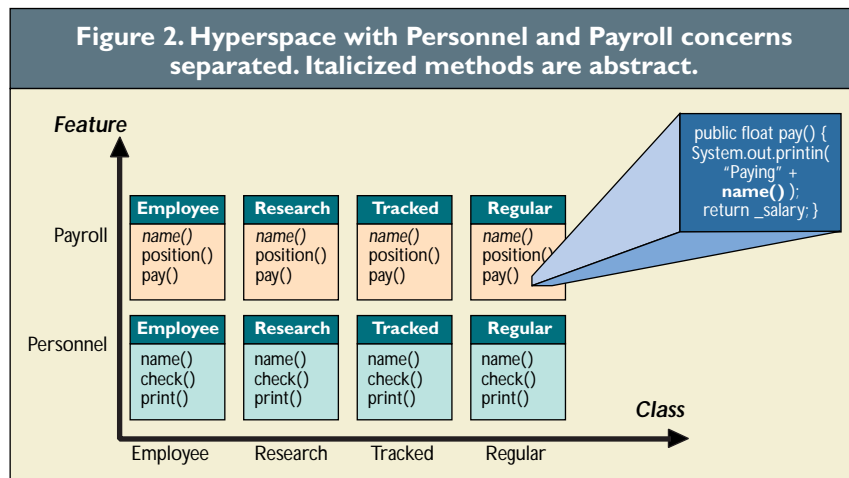
modularizations as needed to reflect new concerns, without disturbing any of the existing modularizations and maintain the existing relationships between concerns.

The first step of on-demand modularization is for the developer to identify those pieces of the existing software that are part of the new concern(s). Figure 1 shows 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—that is, any method, irrespective of its class—named `position` or `pay` belongs to the Payroll concern in the Feature dimension.<sup>1</sup>

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



oper 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.

**Removing a Scattered Feature, and Adding Mix-and-Match in Retrospect.** Hyper/J permits developers to identify and noninvasively 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 modularization: the ability to add new

concern), for each feature, shown as rows in the figure. Each hyperslice encapsulates its own class hierarchy.<sup>2</sup> 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 modularization, Hyper/J makes each hyperslice declaratively complete by inserting abstract declarations for any members referred to, but not implemented within, the hyper-

<sup>1</sup>Developers can choose any names for their dimensions and concerns—there is no predefined Feature dimension.

<sup>2</sup>The hierarchies happen to be identical in structure here, but this need not be the case, as illustrated in the section “Integrating Features with Different Domain Models.”

slice.<sup>3</sup> Declarative completeness is a critical property. First, it means every hyperslice represents a legal Java program that is self-contained, though not necessarily complete. Second, it means hyperslices are loosely coupled, since they never refer 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.

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.

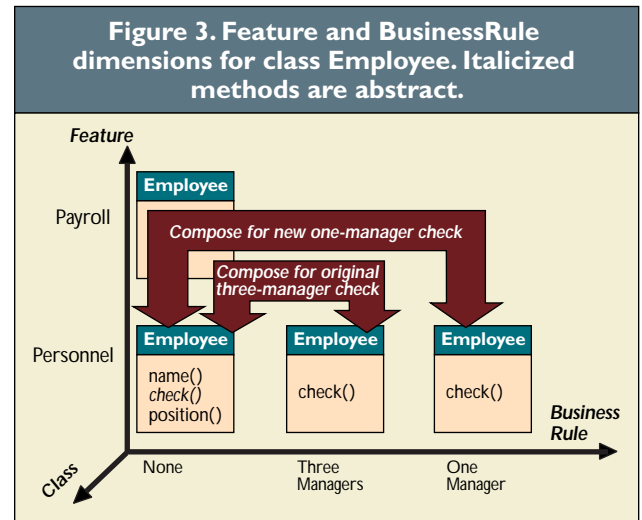
To perform composition with Hyper/J, a developer writes a hypermodule declaration, indicating which hyperslices are to be composed, how their parts are related, and how the composition is to be carried out:

```
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—that is, which parts of these hyperslices correspond to one another. In this case, the general relationship `mergeByName` is used, which is 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 re-created 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 implemen-



tation 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 later in this article.

This example demonstrates how Hyper/J allows developers to adapt existing software to a new context by noninvasively removing or replacing extraneous or inadequate parts, and to achieve “mix-and-match” of features noninvasively, 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.<sup>4</sup>

**Noninvasively Replacing Customer-Specific Business Rules.** Tangled concerns other than features, such as business rules, can also be separated using on-demand modularization, after which they can be

<sup>3</sup>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.

<sup>4</sup>A current limitation of Hyper/J is that it works at the method granularity, so its on-demand modularization mechanism cannot disentangle concerns within a method body. We plan to address this limitation in the 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.

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 in Figure 3.

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
```

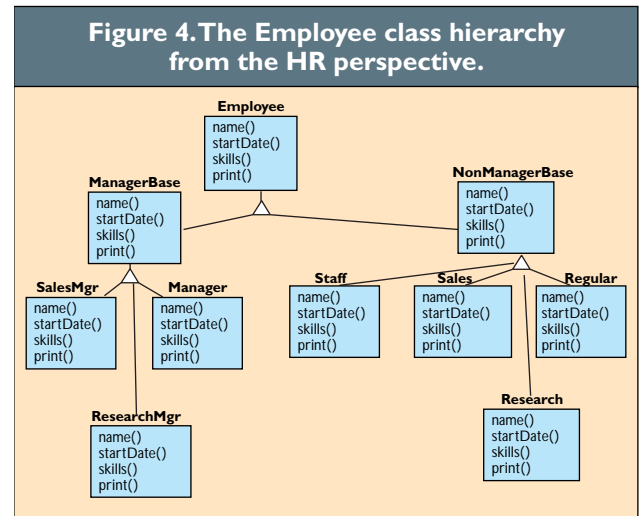
Now we can mix-and-match business rules, creating hypermodules that include either `BusinessRule.ThreeManagers` or `BusinessRule.OneManager`. It would not make sense to include both, however, since these particular rules are mutually exclusive. MDSOC permits the representation of interconcern relationships,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

### 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, one that manages information about employee skills, evaluations, and so forth. Analysis of the HR domain reveals a major distinction between the HR information for managers versus nonmanagers, 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 nonmanager salespeople. The domain model depicted in Figure 4 is thus natural and conve-

nient 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, salespeople, and regular employees, but similar for managers and nonmanagers 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 the original model was wrong, and 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.

**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 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 segments of the

existing personnel feature; for example, 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 (for example, of `print()`) 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 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.

## Related Work

Many current and recent efforts (for example, [1–4, 6–9] and the articles 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 noninvasively; 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.

## Conclusion

 This article introduced MDSOC with Hyper/J and illustrated its use in some development and 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 life cycle, including management 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 a critical goal of software engineering is to produce software 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 toward this ultimate goal.

Hyper/J is available for download, free of charge, from IBM’s alphaWorks Web site: [www.alphaworks.ibm.com/tech/hyperj](http://www.alphaworks.ibm.com/tech/hyperj); also see the MDSOC Web site: [www.research.ibm.com/hyperspace](http://www.research.ibm.com/hyperspace). **C**

## REFERENCES

1. Aksit, M., Bergmans, L., and Vural, S. An object-oriented language-database integration model: The composition filters approach. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1992.
2. Batory, D. and O’Malley, S. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology* (Oct. 1992).
3. Clarke, S., Harrison, W., Ossher, H. and Tarr, P. 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)*, (Oct. 1999).
4. Harrison, W. and Ossher, H. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, (Sept. 1993).
5. Harrison, W., Ossher, H. and Tarr, P. Software engineering tools and environments: A roadmap. In *The Future of Software Engineering*. A. Finkelstein, Ed., ACM, 2000.
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, (Hungary, 2001). Springer-Verlag.
7. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., Irwin, J. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, (Finland, 1997), Springer-Verlag.
8. Mezini, M. and Lieberherr, K. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, (Oct. 1998).
9. Nuseibeh, B., Kramer, J., and Finkelstein, A. A framework for expressing the relationships between multiple views in requirements specifications. In *Transactions on Software Engineering* 20, 10 (Oct. 1994), 260–273.
  10. Ossher, H. and Tarr, P. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2001.
  11. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (Dec. 1972).
  12. Tarr, P., Ossher, H., Harrison, W., and Sutton, S.M. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

---

**HAROLD OSSHER** ([ossher@watson.ibm.com](mailto:ossher@watson.ibm.com)) is a Research Staff Member at the IBM T.J. Watson Research Center in Yorktown Heights, NY.

**PERI TARR** ([tarr@watson.ibm.com](mailto:tarr@watson.ibm.com)) is a Research Staff Member at the IBM T.J. Watson Research Center in Yorktown Heights, NY.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

---