Evaluating Module Systems for Crosscutting Concerns

Jonathan Aldrich

Ph.D. General Examination Report
September 28, 2000

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, WA 98195 USA

jonal@cs.washington.edu

Abstract

Although object-oriented programming techniques support modular decomposition of data types well, there are a number of programming concerns that cannot be cleanly modularized using conventional language mechanisms. This paper classifies these concerns into several categories, and describes examples and a prototypical *challenge problem* from each category. It then describes several recent techniques designed to improve program modularity in the face of these concerns. The strengths and weaknesses of each technique are evaluated with respect to the challenge problems. The paper concludes with a discussion of future research directions.

1. Object-Oriented Modularity

A *module*, as used in this paper, is a programming environment mechanism for decomposing a large software program into smaller pieces, which the environment can automatically compose into a complete program later.

Modules yield a number of programming benefits [P72]. The most important benefit is the intellectual leverage gained by separating the different concerns (programming issues) within a large program into smaller modules. Effective *separation of concerns* makes a program easier to understand, change, and debug. A second benefit comes from *information hiding*, which reduces dependencies between modules, supporting program evolution and independent development of different parts of a program. Other benefits of modules include separate compilation, incremental development, easier unit testing, improved re-use, and intellectual property protection.

Object-oriented programming separates data representation concerns very well using the concept of an *abstract data type*. Since data representations are very likely to change over the course of program evolution, object-oriented systems separate a particular data representation choice from other parts of a program by putting it together with related code in a source file and hiding the representation choice behind an interface. When the data representation

changes, these mechanisms often limit the effect of that change to the particular object involved. Object-oriented inheritance also enables a related group of objects to reuse a data representation scheme.

Although object-oriented programming effectively separates data representation concerns, there are other kinds of concerns that are difficult to separate into modules using conventional object-oriented language technology. In the next section, we describe and classify a number of challenging modularity problems. Section 3 describes several techniques designed to address these concerns, and section 4 evaluates the techniques against selected challenge problems. Section 5 discusses future research directions in modularity, and section 6 concludes.

2. Modularity Challenges

This section classifies programming concerns (that is, the problem domain for modularity) into a number of categories. For each category, we provide a brief description, relate it to other categories, give a number of example modularity problems, and choose a representative *challenge problem* that captures the essence of the category.

2.1. Data Representation

Data representation concerns deal with how program data should be organized; this paper assumes the reader is familiar with this concern. Existing object-oriented languages, especially those with parameterized types, modularize many data representation concerns well. Parnas [P72] proposed the KWIC index system as a challenge problem to evaluate different modularizations of a data-centric program.

2.2. Functional Concerns

A *functional concern* is a piece of functionality, such as a requirement or use case, provided for a client, such as the user, another program, or another component. While a single object is the unit of modularity in mainstream object-oriented languages, the implementation of a functional concern may cross object boundaries or may only be part of an object's implementation. Thus, it is difficult to isolate the functional concern's code in a single module.

A challenge problem for modularizing functional concerns is separating operations on a program representation [TOHS99]. In typical program representations, there are many different kinds of nodes like assignment and function call nodes, and operations like typechecking, various optimization passes, and code generation can be applied to the different nodes. A good modularization would separate the typechecking operations for each node into one module,

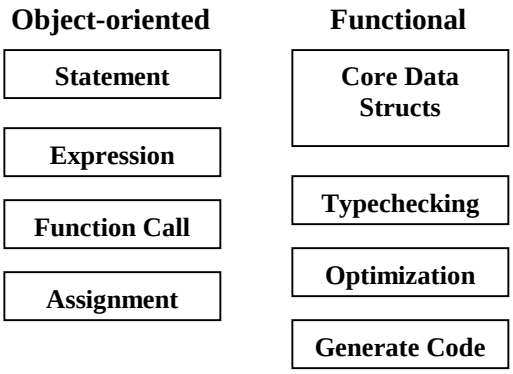

Figure 1. Object-oriented and Functional Decompositions of a Program Representation

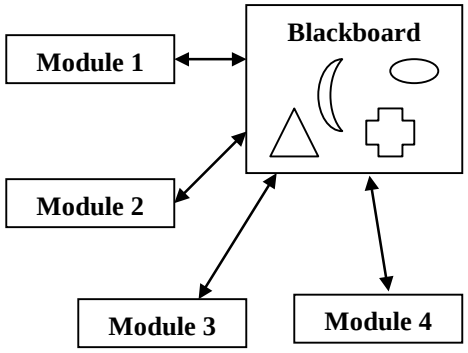

Figure 2. A Blackboard Software Architecture

the code generation operations in another module, etc, effectively separating the functional concerns implemented by the different operations. Tools might even support both an object-oriented view and a functional view (Figure 1).

Other example problems include the various actions of a bottle deposit machine like depositing and printing a receipt [AR92], composable pieces of data structure functionality such as searching and allocation [SB98], user interaction concerns such as directory listings and error messages in an FTP server [LM99], adding an operation to a shape hierarchy [FF98], functional testing of a use case scenario, and many others.

2.3. Program Organization

Program organization concerns address the large-scale structure of the program: how do modules, each composed of many objects, interact on a high level? Large-scale program structure (as opposed to small-scale functional and data representation concerns) is an important concern for program understanding and modular analysis, yet most languages do not support modular structure beyond the size of a single object, obscuring their high-level structure [MN97]. Work on software architecture description languages [MT00] does address this concern, but is not tightly integrated with conventional programming languages. For lack of space, this paper does not survey the area. Effective support for program organization includes specifying relationships between program components separate from the components' code, and clarifying the relationship between components using optional restrictions on the kinds of inter-component communication.

A challenge problem, inspired by an urban simulation program [NBW00], is to specify a blackboard software architecture [GS94] as in Figure 2, with an explicit connection diagram showing how a set of simulation modules interacts with a shared data store. The system should enforce the application constraint that the modules do not directly communicate, but only make modifications to the shared data. It should be easy to make changes like

setting up another data store in the same program with a different set of simulation modules. Ideally, a tool would also be provided to graphically visualize and edit the program structure.

Other program organization problems include the other common architectural styles in Garlan & Shaw's catalog [GS94], including pipe & filter, event-driven, and layered architectures. The abstraction and vertical extension techniques in role models [AR92] can be thought of as encapsulating program organization concerns at design time.

2.4. Global Properties

Global properties are non-local invariants over program data and execution. Code to enforce global properties is often awkward, error-prone, and difficult to change, because custom code must be added at every point where the property could be violated. Techniques for enforcing global properties include customized language constructs, static type systems, dynamic constraint solvers, inference systems like Prolog, and model checking.

Examples of global properties include aliasing properties that prevent a private object reference from escaping a particular module [NVP98], various security properties, global variables or constants, and synchronization between threads in a multithreaded system.

Constraints over program variables are very useful in specifying flexible graphical user interfaces [B93] and enforcing constraints is a good challenge problem for modularizing global property concerns. The programmer should be able to declaratively specify linear constraints and inequalities between ordinary floating-point program variables. When the value of one variable changes over the course of program execution, violating a constraint, an off-the-shelf constraint solver like Cassowary [BB98] is invoked to find variable values that maintain the constraints, ideally leaving the triggering variable's new value unchanged. The user should be able to design custom constraints (such as that two windows cannot overlap in a windowing system) and specify the solver to be invoked when variables involved in a constraint are changed.

2.5. Repetitive Code

Certain concerns almost invariably lead to ubiquitous repetitive code throughout a system. Examples include checking the invariants of a data structure or system of objects, logging application behavior [LM99], serializing object structures into a file, tracing executions of a large number of methods [K00][OT00], or handling common errors consistently throughout a large code base [LL00]. Repetitive code also arises, in a less pervasive way, when several similar but not identical modules or collaboration patterns exist in a single application. Mezini and

```
// challenge: by adding a separate
// module to the program, cache
// return values to make this
// function run in O(n) time.
public static long fib(long n) {
    if (n <= 1)
        return 1;
    else
        return fib(n-1)+fib(n-2);
}
```

Figure 3. Fibonacci Performance Challenge Problem

Lieberherr cite a pricing collaboration example that may arise several times in slightly different forms within a single application, or in a family of related applications [ML98]. Repetitive code can be a symptom of not modularizing global properties or functional concerns like error handling, but it is worth separate consideration because it can independently arise, and because it causes particularly severe program maintenance problems.

A satisfactory solution to the challenge problem, invariant checking, would allow the programmer to specify an invariant in one place and automatically apply it to a large set of methods through some pattern-matching technique. For example, it should be possible to add an invariant to all public methods of a class without enumerating the methods, in case more methods are added later. It should be very easy to remove the invariant once testing is complete, or to change the invariant as changes are made to the code. Finally, the invariant should be inherited by subclasses by default, even if they override the methods where the invariant is checked.

2.6. Performance Concerns

Performance concerns arise when restructuring for optimization results in a less modular design. Restructuring often ties different concerns too closely together, making understanding & change difficult. Examples of restructuring that interferes with good modularity include caching computation results, specialization, loop fusing, etc. Performance problems can also involve a system of interacting objects; if program organization concerns have not been addressed, it can be very difficult to find bottlenecks in a monolithic program.

The problem space is very diverse, so it is difficult to capture the whole range with a single example. The chosen challenge problem is improving the performance of the expensive recursive `fibonacci` function, shown in Figure 3. A clean design would leave the recursive function definition as is, and separately apply a reusable caching library to store previously computed results, improving the function's runtime to O(n).

2.7. Declarative Specifications

Some concerns have proven especially difficult to modularize because the concern is best specified in a declarative or graphical way. Examples include graphical user interface design, constraints, and object initialization

```
Exp ::= Exp + Term { $$ = $1 + $2 }
Exp ::= Term { $$ = $1 }

Term ::= Term * Fac { $$ = $1 * $2 }
Term ::= Fac { $$ = $1 }

Fac ::= num { $$ = $1 }
```

Figure 4. YACC-like BNF grammar and actions for a simple calculator

and traversal [ML98]. Imperative specifications of these concerns tend to be unclear, repetitive, and awkward. While it may be impossible to express every concern in a single language, a language feature supporting custom declarative syntax extensions could express many idioms easily.

A challenge problem for this area is providing a clean specification for a parser. Because textual file formats are most clearly specified in the declarative BNF syntax, the cleanest way to write a parser is to annotate a BNF description of the language with actions to take when a particular construct is parsed. Figure 4 shows the grammar and parse actions for a simple calculator. A good solution would integrate naturally with an extensible host language, be recognizably close to BNF, support good performance, and avoid repetition.

2.8. Context Dependent Behavior

Experience has shown that selecting code implicitly depending on the current execution context leads to more modular designs than a tree of `if` statements. While object-oriented programming supports selecting code according to one context—the receiver of a message—many other contexts are possible, including the classes of arguments to a multi-method, argument values and structure [EKC98], the current exception handler, or a closure.

A challenge problem from the security domain is to select code based on the caller of a sensitive method. To solve this problem, a system should allow methods to be selected based on the calling scope [AT98], including the caller's class, the particular calling object, or an object several levels up the call chain. For some callers, the invoked method would function normally; for others, there might be restricted functionality or an exception might be thrown.

2.9. Discussion

The classification above is intended to highlight patterns in a wide range of modularity problems, to help language designers focus on related classes of modularity problems instead of single problem instances. Although many of the problems cited in the separations of concern literature can be classified in this system, it is not necessarily exhaustive. Many modularity problems fall into more than one of the above categories; for example, many global properties are best specified in a declarative way. A problem like concurrency may include program

```
class Meth extends Node {
  accept(Visitor v)
    { v.visitMeth(this) }
}

class Typeck extends Visitor {
  visitMeth(Meth n) { ... }
  visitStmt(Stmt n) { ... }
  visitFnCall(FnCall n) { ... }
}
```

Figure 5. Visitor Pattern for Typechecking[1]

organization concerns such as the number of threads and how they interact, global property concerns involving synchronization constraints on groups of objects, repetitive locking code, behavior may depends on the thread context, and system-wide performance issues. Techniques that focus on a particular class of modularity problems should improve modularity in the relevant aspects of a multi-faceted problem like concurrency.

Our classification deals with the modularity problem domain, not the solution domain, so one kind of solution may not solve every problem in a given category. For example, global properties can be enforced in dramatically different ways: statically with type systems, dynamically with constraint solvers, or outside the language with static verification tools.

3. Advanced Module Systems

This section describes several programming systems with unique features that support modular programming, and evaluates how well they support different classes of concerns.

3.1. Design Patterns

Design patterns [GHJV95] are programming idioms that solve particular problems in object-oriented software design. Although they are typically less elegant than a language-supported solution, design pattern solutions can be used in any object-oriented language. For example, the Visitor pattern [GHJV95] allows functionally related methods to be grouped together in a single class, and makes it easy to extend the functionality of a group of classes. Figure 5 shows the Visitor code for typechecking an abstract syntax tree (AST), part of the functional concern challenge problem. Typechecking methods are encapsulated inside the `Typeck` visitor. To typecheck a method, the programmer passes a `Typeck` object to the `accept` method of the `Meth` object. This method calls `visitMeth` on the visitor, which typechecks the method node and calls `accept` recursively on the statements in the method. Other visitors can implement various optimization and code generation passes in an extensible way.

---

[1] Our examples are abbreviated for clarity and space; many (but not all) came from actual implementations.

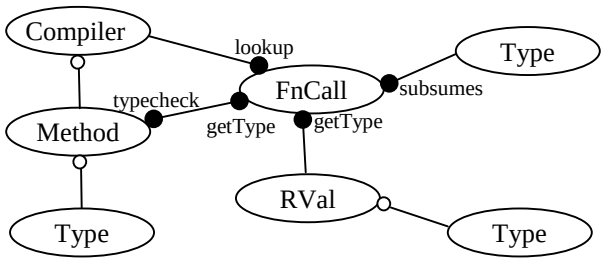

Figure 6. Role Model for Typechecking a Function Call

Unfortunately, the Visitor pattern gives up the data structure extensibility strength of object-oriented languages. Krishnamurthi et al. designed the Extensible Visitor pattern [KFF98] using Factory Methods to overcome this drawback, and suggested a syntactic sugar language mechanism to simplify the cumbersome implementation details. Other patterns such as Facade and Mediator help with program organization, but they cannot fully solve this challenge because they provide no guarantees, they are not explicit in the implementation, and they tend to be most appropriate at smaller scales. Other pattern-like idioms such as using get/set methods for object fields help with global properties and context-sensitive behavior concerns.

3.2. Role Models

Andersen and Reenskaug [AR92] introduced the *role model* design technique as a way to express a single functional concern in an object-oriented system, and later compose it with role models implementing a different concern. In their system, a role is a particular responsibility that contributes to a role model's behavior, and is usually implemented by a single object. A role model is made up of two or more roles, and role models can be composed to form larger role models. Their design technique includes an abstraction mechanism for collapsing a group of roles into a single role to provide a higher-level view of the system, offering some design-time support for program organization. An intuitive graphical notation is very helpful for understanding the design of the system.

Figure 6 shows a role model for typechecking a function call node. Each role in the role model is depicted with an oval, and lines connecting roles represent relationships. An oval means that a role knows about a collaborating role, and a filled oval is annotated with a method interface for the other role. In this role model, the method role calls typecheck on the function call role. The function call looks up the designated method in the compiler's tables, gets the type signature of the method, checks the types of the actual parameters, and then determines if the actual types are subsumed by the formal types.

```
template <class Super>
class Typecheck : public Super {
public:
  class Stmt : public Super::Stmt {..}
  class Meth : public Super::Meth {
  public:
    Meth(Stmt *s) : Super::Meth(s) { }
    void typeck() {
      ((Stmt *)stmt)->typeck();
    }
};};

typedef Typecheck < Base > Rep;

Rep::Meth*m=new Rep::Meth(stmt);
m->typeck();
```

Figure 7. Mixin Layer Implementation of Typechecking

VanHilst and Notkin [VN96] describe an implementation technique for role models using C++ templates. A role is implemented as a class, including all the relevant variables and methods. Roles are composed by inheritance, and to make roles reusable, the superclass of a role is specified in a template argument, providing a form of mixin class [BC90]. Since the actual objects it interacts with are usually subclasses of the interacting roles, the class is templated by the final types of the other classes. The actual composition of roles is done in a sequence of template class instantiations separate from the role definitions.

Smaragdakis and Batory [SB98] describe *mixin layers*, a variant of this implementation technique. Their solution groups roles into role models container classes. In Figure 7, the `Typecheck` role model class is templated by a role model superclass. Each role within the class such as `Meth` extends the corresponding role in the role model's superclass. Some ugly casts are required due to C++'s type system; for example, `stmt` was inherited from `Super::Meth` and so its static type is `Super::Stmt` rather than `Stmt`. Role model classes are composed using template instantiation, typically using a `typedef` statement to reduce template verbosity.

The mixin layer technique is more structured than VanHilst and Notkin's, and can use simpler instantiations. However, it makes even more elaborate use of templates and other complex C++ features, potentially reducing understandability, it is tough to refer to the final composed classes from within a role, and it is difficult to integrate role models that involve different but overlapping sets of classes. In both implementations it can be challenging to understand how the pieces compose together, suggesting that language and tool support may be needed.

```
                nodes.cecil
object meth isa node;
object stmt isa node;
object fncall isa node;

              typecheck.cecil
method typeck(n@:meth) {
  statements.do(&(s:stmt) {
    s.typeck();
}}
method typeck(n@:stmt) { ... }
method typeck(n@:fncall) { ... }
```

Figure 8. Cecil Implementation of Typechecking

3.3. Cecil

Cecil [C92] is a high-level object-oriented language designed to support rapid prototyping of extensible systems. Currently, there is no module system implemented in Cecil that supports encapsulation and information hiding. However, the features of the language support separation of many kinds of concerns very well. Cecil allows methods to be defined anywhere, so programmers can group methods by functionality rather than by object, if appropriate. Figure 8 shows how object (class) declarations can be grouped into the `nodes.cecil` file, while all the typechecking methods of these nodes can be naturally grouped into the `typecheck.cecil` file. Assignments to fields are implemented as calls to automatically generated get/set methods, which makes it very easy to intercept these assignments to support the global property challenge problem. Cecil also provides support for many kinds of context-sensitive behavior through multi-methods, predicate dispatching, and closures.

```
signature BBOARD = sig ...

functor MODULE1
  (structure B : BBOARD)
  :> MODULE
  where type Board.t = B.t = ...

structure MyBoard :> BBOARD = ...

structure MyModule1 = MODULE1
    (structure B = MyBoard)

val mb = MyBoard.create ...
val mm1 = MyModule1.create mb

mm1.sim(...)
```

Figure 9. ML Blackboard Architecture Modules

3.4. ML

ML [MTHM97] is a functional programming language with a powerful module system that represents the high-water mark of conventional modularity. Since it is functional in style, it supports decomposition of functional concerns well but may lack some of the benefits of object-orientation. A parameterized module (functor) in ML can be instantiated into a module instance (structure) many times by applying it to a structure with the appropriate interface. The Fox Project [B95] implemented a TCP/IP stack in ML to demonstrate that ML's module system supports both good modular design and performance concerns like sharing the same data structure between modules without exposing it to module clients. ML supports our functional challenge problem well.

As shown in Figure 9, the explicit composition of modules can improve program organization in the simulation challenge problem. The `BBoard` signature defines the interface for the central bulletin board, which is defined in a separate structure definition. `MODULE1` is parameterized over an unknown structure implementing the `BBoard` signature. Before `MODULE1` can be used, it must be instantiated (as `MyModule1`) with a particular implementation of the bulletin board. Datatypes and functions defined in the modules can then be instantiated and invoked. The explicit module composition statements make the structure of the program clearer, and show dependencies and communication patterns between components. However, it is still possible for modules to communicate through a "backdoor" by using the interface of `MODULE2` (for example) within the definition of `MODULE1`, so better software architecture support is still needed.

```
(define Typechecking
 (unit (import Meth Stmt FnCall)
    (export TMeth TStmt TFnCall)
   (define Tmeth
     (class* Meth () (stmt)
       (public [typeck
         (lambda ()
           (send stmt typeck)))
...))))

(define Base+Typechecking
 (compound-unit
  (import)
  (link [B (Base)]
        [T (Typechecking
              (B Message)
              (B Stmt)
              (B FnCall))])
  (export (T (TMeth Meth))
          (T (TStmt Stmt))
          (T (TFnCall FnCall)))))
```

Figure 10. Units Implementation of Typechecking

3.5. Units

MzScheme's Units [FF98] module system supports modular extension of functional concerns in the context of an object-oriented extension to Scheme. A unit is a parameterized module that includes a set of formal placeholders for imported classes, a series of definitions, and a set of exported classes. The actual imports for a unit are specified in a separate linking expression. MzScheme supports mixins [BC90], where classes defined in a unit may inherit from imported classes, despite the fact that these imports are undefined until the unit is linked with other units. The mixin construct allows a superclass to be extended without relying on the exact details of the superclass, while the separate linking language allows a mixin to be applied to many different superclasses to produce different resulting classes, and also allows an import to be replaced with an updated or enhanced class without modifying the importing module.

Like ML, MzScheme provides full support for our functional challenge problem and partial support for our program organization problem. Figure 10 shows the typechecking problem implemented with Units. The `Typechecking` module imports existing `Meth`, `Stmt`, and `FnCall` classes, defines subclasses that include `typeck` methods, and exports the subclasses. A *compound unit* links two ordinary units together by connecting their imports and exports; for example, the `Message` class from the `Base` module (abbreviated `B`) is passed to the `Typechecking` module as the `Meth` import. The compound unit can then export a subset of the linked objects, possibly under new names. Their module system is partly implemented with Scheme's macro facility [KCR98], which also supports declarative specifications that can be programmatically transformed into executable code by user-defined meta-programs. Macros can also be applied to our performance challenge problem.

3.6. Adaptive Components

Adaptive Plug and Play Components [ML98] focus on the need to modularize functional collaborations in a reusable way. In adaptive programming, object traversals are specified by a list of the classes to visit, and actions to be performed before the traversal, during each object visit, and when returning a traversal result. These *traversal strategies* reduce the dependency of a collaboration on a concrete class structure. The class structure that a component uses is declared in terms of formal placeholder classes, methods and fields in a separate, abstract interface, further reducing the dependency on concrete classes. Collaborative behavior is specified in terms of these placeholders, so it can be reused in a large number of actual contexts. Components are composed separately, and mixin functionality allows a component to modify an unspecified super-component. Thus, adaptive components are very effective at packaging re-usable slices of behavior, and applying them to diverse places in the class hierarchy of a program. The Demeter/Java [DRG00] tool also supports custom language extensions for initializing object data.

```
APPC Typechecking {

 Interface Class Graph:
  Method = <stmts> Stmt
  Stmt { /* interface */ }

 Behavior Definition:
  Method {
   main-entry void typeck() {
    from Method to Stmt {
     init { ... }
     at Stmt { ... } } }
}

Compiler::+ { void typeck() =
 Typechecking with {
   Method = ConcreteMethod;
   Stmt = Assign { typeck = ck};
}}
```

Figure 11. Adaptive Components Implementation of Typechecking

A Typechecking APPC is shown in Figure 11. The classes involved in the typechecking collaboration are described abstractly in the Interface Class Graph section of the APPC. First, a structural specification declares a `stmts` field of `Method` that references one or more objects of type `Stmt`. An interface declares the methods of `Stmt` that are used in this APPC. The collaboration's methods are specified in the Behavior Definition section. The `main-entry` describes `typeck`, the primary function in the APPC. In this case, `typeck` implements a traversal strategy, visiting each `Stmt` reachable from this `Method`. The system automatically infers how to implement the traversal from the structural interface and the actual class structure at run time. The APPC is linked into the `Compiler` application with the `::+` operator. The linking specifies the actual names of the classes involved and their members.

3.7. Hyperspaces

Hyperspaces [TOHS99] were developed at IBM to enable better separation of concerns. The Hyper/J tool [OT00] implements Hyperspaces for Java. Code for different concerns is written in separate standard Java source files, which are then compiled by any Java compiler and composed by Hyper/J to form a final program. A hyperspace file lists the Java classes to be composed, and a concern mapping file describes which packages, classes and methods belong to each concern. Composition is specified by a hypermodule file, which describes which concerns should be integrated, and the mapping between corresponding elements of each concern. Integration relationships can include merging (execute both methods in a specified order, with another method integrating the return values), overriding, renaming, and bracketing with before/after methods. Pattern matching allows one method to be integrated into a large set of others.

kernel.DataStruct.java

```
class DataStruct { /* main impl */ }
```

invariant.DataStruct.java

```
class DataStruct {
  public void pre(String meth) {
    assert(x+y<=100,"entering "+meth);
} ... }
```

invariant.cm (concern mapping file)

```
package invariant : Func.invar;
package kernel : Func.kernel;
```

invariant.hm (hypermodule file)

```
relationships:
  mergeByName;
  bracket "*" with (
    before Func.invar.DataStruct.pre(
              $OperationName),
    after Func.invar.DataStruct.post(
              $OperationName),
    "DataStruct");
```

Figure 12. Hyper/J Implementation of Repetitive Invariants

Figure 12 shows a Hyper/J implementation of class invariants. The core functionality of the `DataStruct` class is defined in an ordinary Java file, and the invariant is defined in a Java file of the same name but in a different package (renaming facilities are specified but not fully implemented). A concern mapping file partitions the classes into the functional (`Func`) concerns `Kernel` and `Invariant`. The hypermodule file specifies relationships between the classes; by default, classes and methods are merged by name, so the two `DataStruct` classes will be merged into one. The `bracket` specification instructs the Hyper/J compiler to insert calls to the `pre` method before each operation (the wildcard `*` operator) in "`DataStruct`", and to insert `post` calls after.

Hyper/J has excellent support for separating functional concerns and repetitive code, including features present in no other system. It can even conceptually (if not physically) separate methods in legacy code without changing or even using the source code. Hyper/J can only be applied at boundaries between methods, making it hard to support global properties that depend on variable values that may change within a method. Although Hyper/J doesn't support first-class software architecture, it does offer more program-organization capability than the other systems

```
class Sensitive
  filters
    f1 : Error = {
      privil => {self.special},
      true => {self.ordinary} };
  states
    privil
      return sender.subtypeOf(
                    Privileged);
  methods
    special() { ... }
    ordinary() { ... } ...
end;
```

Figure 13. Calling Context Sensitivity with Composition Filters

considered because its concern integration features can be used to connect modules. Dealing with all the specification files, adding extra declarations so the individual concerns can be compiled, and visualizing the way concerns integrate were difficult in this system. Improved tool support or a more advanced base language could go a long way to making Hyperspaces practical.

3.8. Composition Filters

Composition Filters [AT98] provides message filters that can intercept, transform, and redirect any incoming message to custom handling code. Their declarative specification is more powerful, reusable, and convenient than mechanisms like Smalltalk's `doesNotUnderstand` message or Java's runtime-generated message filters. Filters can take actions such as signal an error, dispatch to a custom method (similar to `doesNotUnderstand`), encapsulate a transaction, or wait until a condition is true. A Boolean condition method controls whether a filter takes action for a particular message or passes that message on to the next filter in a compositional sequence. Filters have been used to restrict access to certain methods based on the caller, implement a synchronization protocol, or to implement object database-like functionality. Filters can forward messages to a component object, providing some support for separation of functional concerns with role models, etc. Composition filters are well suited to separating many kinds of concerns, including our repetitive code, performance, and context-sensitive behavior challenges.

The `Sensitive` class in Figure 13 defines a single message filter that protects the `special` method from access by unprivileged callers. The `Error` filter rejects all messages except those specified in the filter definition. Message receptions for which the `privil` state is true are allowed to dispatch to the `special` method, but any message is allowed to dispatch to the `ordinary` method. The `privil` state is a side-effect free predicate that checks to verify that the `sender` (a language keyword) is a subclass of `Privileged`.

```
abstract aspect Caching {
  abstract pointcut functions();
  around() returns Object
           : functions() {
    args = thisJoinPoint.params;
    ... thisJoinPoint.runNext();
  }
  Hashtable valueCache = ...;
}

aspect CacheFib extends Caching
                of eachJVM() {
  pointcut functions() :
      executions(* fib(..));
}
```

Figure 14. A Caching Aspect in AspectJ

```
aspect Constraint {
  pointcut varChanged(int val) :
    sets(int MyClass.x)[val]
    || sets(int MyClass.y)[val];

  after(int newVal) :
    varChanged(newVal) {
      Solver.invoke(newVal,
                thisJoinPoint.field());
  }
}
```

Figure 15. Constraint Satisfaction in AspectJ

3.9. AspectJ

AspectJ [K00] is an aspect-oriented programming [KLM+97] extension of Java that supports crosscutting concerns. AspectJ supports functional concerns well; the `aspect` construct can encapsulate member functions or data that are related in functionality, even though they might be part of different classes or nested within different functions. A declarative pattern-matching facility allows repetitive code to be specified in a modular way, and generic aspects can be written that are reused by inheritance and specialization. Context-specific information like the method name and arguments allows generic code to be customized, and control-flow aspects allow information to be recorded implicitly in a caller and later affect a callee. Not all AspectJ solutions are perfectly clean, but they are decent in view of the limitations of the Java host language.

AspectJ provides an elegant solution to our performance challenge problem, caching the results of an expensive computation (Figure 14). The abstract `Caching` aspect defines the caching protocol. A `pointcut` declaration will be overridden by sub-aspects to define which functions are to be affected by the `Caching` aspect. The `around` method is invoked first whenever one of the specified functions is called. It examines the parameters of the method (using the `thisJoinPoint` aspect mechanism) to see if they are currently in the cache; if so, the `around` method returns immediately without invoking the actual callee. Otherwise, the callee is invoked with `thisJoinPoint.runNext()` and the result is cached before returning from the `around` method. The `CacheFib` aspect extends `Caching` by overriding the functions `pointcut` to refer to all executions of functions named `fib`. The `ofeachJVM()` declaration specifies that there is one instance of the `CacheFib` aspect per

JVM, which is sufficient since `fib` is a static function. The aspect could also be instantiated for each object of a particular type in order to keep a separate cache for each object.

Figure 15 shows a constraint aspect that invokes a constraint solver whenever the value of a constrained variable is changed. This functionality is specified but not yet implemented in AspectJ.

3.10. Other systems

Numerous other systems have focused on better separation of concerns. GenVoca [BVT+94] and Intentional Programming [S95] are meta-programming environments that support domain specific languages through syntax tree manipulation. They support declarative specifications and a number of other concerns very well, although the code manipulations may be challenging to specify and hard to understand. In order to limit the scope of this paper, we do not describe these metaprogramming systems in detail. Languages like BETA [M93] and Objective C have unique features that help to structure certain types of code. The System Modeling Language [LS83] and its successor Pebble [LB88] were early linking languages for parameterized modules similar to those in ML. Jigsaw [BL92] is a theoretical framework for modularity that defines a number of module operators, such as merging modules, modifying modules, instantiating a module, and renaming a module's members; although impractical as a language, it is a good standard for evaluating a module system's orthogonality. Contracts [H92] are another way to express interactions among groups of objects. A contract specifies a framework of objects, including invariants over their data, that can be specialized to produce variants of an algorithm. Open Implementation [K96] is a design methodology for modules that can adapt to the needs of different clients by exposing a secondary interface that allows clients to specify an appropriate implementation strategy.

4. Evaluation

Table 1 summarizes how well each module system solved our challenge problems. A black circle in the table means that the system provides an elegant solution to the problem. A half-circle indicates that there is a partial solution, but it is awkward, it does not solve all the issues in the challenge problem, or the supporting mechanism is specified but not implemented. In this section, we discuss the solutions to each problem and discuss remaining issues in separating the different classes of concerns.

System	Functional Concerns	Program Organization	Global Properties	Repetitive Code	Performance Concerns	Declarative Specification	Context Behavior
Design Patterns	◑	◑	◑	○	◑	○	◑
Mixin Layers	●	○	○	○	○	○	○
Cecil	●	○	●	○	○	○	○
ML	●	◑	○	○	○	○	○
MzScheme Units	●	◑	○	○	◑	●	○
Adaptive Components	●	◑	○	○	◑	◑	○
Hyper/J	●	◑	○	●	●	○	○
Composition Filters	◑	○	○	●	●	○	●
AspectJ	●	○	◑	●	●	○	●

Table 1. Support for challenge problems among module systems (●full ◑partial ○none)

4.1. Problem Solutions

Many different systems solved the functional challenge problem well, due to the importance of the problem and the length of time is has been around. Among object-oriented systems, Cecil solves this problem most cleanly, since ordinary methods can be added to objects from any source file, not just where the object is declared. Adaptive Components provide what is possibly the most reusable and evolvable solution, as an operation like typechecking can be easily specified without relying on the exact structure of the abstract syntax tree. ML and MzScheme Units provide excellent solutions that include strong encapsulation of the functional code. Other systems also provided very good solutions, although sometimes with a bit of inconvenient syntax.

The most powerful module system, Units, provides the best solution to the program organization problem, by specifying different modules in the blackboard architecture in modules with explicit imports and exports that make it clear where communication occurs. These modules are linked together in a separate specification that helps to clarify the whole application's structure. Full support for this problem comes only from the Software Architecture Description Language domain [MT00], which this paper does not survey.

Cecil provides the best solution to our global properties challenge problem, because the universal get/set methods for object fields make it easy to intercept variable mutations that may violate constraints. AspectJ has a mechanism that provides an excellent solution, but it is not yet implemented.

AspectJ, Hyper/J, and Composition Filters all provide clean ways to intercept method dispatch for a large set of methods and add code to be executed before a method. This ability is essential to good solutions to the repetitive code and performance challenge problems.

Of all the systems surveyed, only MzScheme, through its macro capability, supports fully flexible user-defined declarative specifications. Demeter/Java provides declarative specifications for object data, but there's no easy way to embed code, e.g., for parser actions. Composition filters provide the cleanest way to choose behavior based on the calling object, by testing the `sender` keyword in a message filter. AspectJ provides control-flow aspects, which is an object associated with all control flow from the entry to a procedure, through arbitrary levels of called procedures until the procedure returns. These aspects can enhance the behavior of methods the control flow touches in a very flexible way.

4.2. Remaining Problems

One or more of the module systems we considered solved all but one of our challenge problems; do they offer any remaining challenges?

Although individual systems may solve particular problems well, integrating the features of those systems into a coherent whole is a non-trivial task. For example, integrating parameterized modules and mixins with a strong type system is an open problem [FF98], and multi-methods almost certainly pose further difficulties. Software architecture description languages have yet to be tightly integrated with conventional programming languages. Existing systems that support pattern matching to eliminate repetitive code, such as AspectJ, Hyper/J, and Composition Filters, provide no static encapsulation mechanism, and it is an open problem how this should be done.

More importantly, this paper's challenge problems were chosen to be at the edge of the attainable in order to effectively distinguish between the different systems. Considerably more difficult problems remain in each category. Functional concerns that influence the middle of a method's code, rather than the beginning or end, are tough to factor out with current systems. Actual solutions to the constraint problem are still nowhere near as clean as in constraint-imperative languages [L97], and problems like synchronization, security, or aliasing properties are even more challenging. It would be very premature to claim that any system does away with repetitive code completely, and performance concerns like loop fusing defy the most powerful compiler and self-modifying code systems. Very few languages have the wide range of context-dependant behavior that Cecil provides.

Finally, the systems themselves are experimental, and more language design is needed to smooth off the rough edges. Early versions of AspectJ were like a grab bag without a clear rationale for including or excluding features. The language has become more structured, orthogonal, and comprehensive in its scope, but its design continues to

develop and is not as clean or orthogonal as it might be because its host language, Java, lacks features like closures and generic types. Both AspectJ and Hyper/J need better tool support to deal with the complexity of crosscutting concerns. Composition Filters has a complex and unusual object model; practical experience is needed to determine if it is effective.

5. Future Work

There are several avenues of future work that we could effectively pursue: further evaluation of the different systems, tool support for crosscutting modules, and the design of a module system for Cecil.

5.1. Other evaluation metrics

Our challenge problems effectively evaluate the expressiveness and breadth of features in the systems with respect to important classes of programming concerns. At a small scale, they evaluate other metrics such as ease of tool use, clarity and verbosity of the language, reuse, and adding features without modifying existing code. However, many of these metrics are best evaluated on a full program that includes sub-problems from diverse concern classes. The specification of such a program, and its implementation in the various systems, would be a good way to evaluate ease of change, code reuse, evolution of code without changing the source, challenges in integrating different features, information hiding, verbosity, ease of static analysis, and tool support.

5.2. Tool Support

Interesting future work remains in tool support for separation of concerns. Visualization tools are important for understanding program organization, and also for seeing how crosscutting concerns compose and interact. A graphical module linking language would isolate program organization concerns more effectively, making large programs far easier to understand and change. Finally, efficient and separate compilation of crosscutting concerns is a major unexplored research area.

5.3. Modules for Cecil

Because of our experience with Cecil, a key area where we can make a research contribution is designing support for separation of concerns in the context of a high-level, orthogonal object-oriented language. A module system for Cecil would preserve the language's support for multi-methods, open objects and static types, building on

existing research [MC99] to allow separate compilation. It would support first-class, parameterized, hierarchical modules with a separate linking language powerful enough to express software architecture. Modules would express different interfaces to clients, extenders, and perhaps "privileged clients." The language would support separation of concerns through mixins, before/after/around methods, pattern matching, and other facilities. XML could be used for documentation purposes as in the new C# language [HW00], as well as to provide an extensible language syntax or even to form the module linking language.

We expect to encounter a number of open problems in module design work for Cecil:

- How can parameterized modules be combined with multimethods?
- How can static parameterized types coexist with parameterized modules and mixin inheritance?
- How can a system that supports separation of crosscutting concerns also provide effective encapsulation? What is the appropriate scope of crosscutting code?
- When one crosscutting module affects many others, how can this be specified? A functional, applicative style (as in ML) seems inappropriate.
- When crosscutting concerns interact, how can the programmer detect possible conflicts, and specify an ordering or priority to resolve the conflicts?
- How to express functionality such as: control-flow aspects in AspectJ, separating the class name from new statements, merging class hierarchies in Hyper/J, restricting objects from escaping a module, and supporting declarative specifications?
- How can modules containing crosscutting concerns be separately and efficiently compiled?

6. Conclusion

This paper proposed a number of challenge problems for effective separation of concerns in object-oriented systems. It includes a categorization of important concerns that are difficult to modularize, encompassing many of the examples from the separation of concerns literature. The paper evaluated the strengths and weaknesses of a number of different systems using the challenge problems as guides. One direction of future work involves defining a benchmark program that unifies problems from all the different concern categories into a single application, in order to find out how the different concerns might interact. Another direction is to look at ways to extend Cecil and

its associated tools with better support for separation of the different categories of concerns. The orthogonal high-level object-oriented features and advanced type system of Cecil should be a benefit in designing such a system.

References

[AR92] Egil P. Anderson and Trygve Reenskaug. System design by composing structures of interacting objects. In Proceedings of the 1992 European Conference on Object-Oriented Programming, pages 133--152, 1992.

[AT98] M. Aksit and B. Tekinerdogan. Solving the Modeling Problems of Object-Oriented Languages by Composing Multiple Aspects Using Composition Filters. ECOOP'98 AOP workshop position paper, 1998.

[B93] Bjorn Freeman-Benson, "Converting an Existing User Interface to Use Constraints", Proc. ACM Symposium on User Interface Software and Technology, Atlanta, Georgia, November 1993.

[B95] Jeremy Buhler. The Fox Project. ACM Crossroads 2.1, September 1995.

[BB98] Greg J. Badros and Alan Borning, "The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation", University of Washington tech report 98-06-04, 1998.

[BC90] Gilad Bracha and William Cook. Mixin-based Inheritance. Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications, 1990.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets Inheritance. Proc. International Conference on Computer Languages, pages 282--290, San Francisco, April 1992.

[BVT+94] D. Batory, S. Vivek, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. IEEE Software, 11(5):89--94, September 1994.

[C92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. Proc. European Conference on Object-Oriented Programming, 1992.

[DRG00] The Demeter Research Group. Online Material on Adaptive Programming and Demeter/Java. http://www.ccs.neu.edu/research/demeter/, 2000.

[EKC98] Michael D. Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. Proc. 12th European Conference on Object-Oriented Programming, Brussels, Belgium, 1998.

[FF98] Findler and Flatt. Modular Object-Oriented Programming with Units and Mixins. Proc. International Conference on Functional Programming, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[GS94] David Garlan and Mary Shaw. An Introduction to Software Architecture. Carnegie Mellon University technical report CMU-CS-94-166, 1994.

[H92] Ian M. Holland. Specifying Reusable Components Using Contracts. Proc. 6th European Conference on Object-Oriented Programming, 1992.

[HW00] Anders Hejlsberg and Scott Wiltamuth. C# Language Reference. Available at http://msdn.microsoft.com/library/prelim/csref/vcoriCReference.htm, 2000.

[K00] Gregor Kiczales. AspectJ™: aspect-oriented programming using Java™ technology. JavaOne, June 2000.

[K96] G. Kiczales. Beyond the black box: open implementation. IEEE Software 13.1 pp 8-11, January 1996.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised report on the algorithmic language Scheme. ACM SIGPLAN Notices 33(9), September 1998.

[KFF98] Shriram Krishnamurti, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. Proc. European Conference on Object-Oriented Programming, 1998.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. Proc. 11th European Conference on Object-Oriented Programming, 1997.

[L97] Gus Lopez. The Design and Implementation of Kaleidoscope, A Constraint Imperative Programming Language. Ph.D. dissertation, University of Washington tech report 97-04-08, April 1997.

[LB88] Butler Lampson and Rod Burstall. Pebble, a kernel language for modules and abstract data types. Information and Computation, 76(2/3):278--346, Feb/Mar 1988.

[LL00] Martin Lippert and Cristina Videira Lopes. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. Proc. 22nd International Conference on Software Engineering, 2000.

[LM99] Albert Lai and Gail Murphy. The structure of features in Java code: An exploratory investigation. Position paper for the OOPSLA '99 workshop on multi-dimensional separation of concerns, 1999.

[LS83] Butler Lampson and Eric Schmidt. Practical use of a polymorphic applicative language. Proc. 10th ACM Symposium on Principles of Programming Languages, Austin, 1983.

[M93] Ole Lehrmann Madsen. An overview of BETA. Book chapter, http://www.daimi.au.dk/~beta/Papers/BetaOverview/betaoverview.abstract.html, 1993.

[MC99] Todd Millstein and Craig Chambers. Modular Statically Typed Multimethods. Proc. 13th European Conference on Object-Oriented Programming, Lisbon, Portugal, June 1999.

[ML98] M. Mezini, and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1998.

[MN97] Gail C. Murphy and David Notkin. Reengineering with Reflexion Models: A Case Study. Computer 30, 8, pp. 29-36, August 1997.

[MT00] Nenad Medvidovic and Richard Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. on Software Engineering, 2000.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The Definition of Standard ML (Revised). MIT Press, 1997.

[NBW00] Michael Noth, Alan Borning, and Paul Waddell. An Extensible, Modular Architecture for Simulating Urban Development, Transportation, and Environmental Impacts. http://www.urbansim.org/Papers/. Submitted for publication, August 2000.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. Proc. 12th European Conference on Object-Oriented Programming, Brussels, Belgium, 1998.

[OT00] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach. Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.

[P72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 15(12): 1053-1058, ACM Press, 1972.

[SB98] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin-Layers. Proc. European Conference on Object-Oriented Programming, 1998.

[S95] Charles Simonyi. The Death of Computer Languages, The Birth of Intentional Programming. Microsoft Research Technical Report MSR-TR-95-52, September 1995.

[TOHS99] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. Proc. 21st International Conference on Software Engineering, May 1999.

[VN96] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications, 1996. Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications, 1998.