Challenge Problems for Separation of Concerns

Jonathan Aldrich

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

jonal@cs.washington.edu

Abstract

A number of techniques have been proposed to achieve a better separation of concerns in programming systems. How does one evaluate the benefits and drawbacks of the various alternatives, and offer guidance to the designers of future systems? This paper proposes a set of challenge problems for separation of concerns, each of which deals with different types or *dimensions* of concerns. Drawing from my experience with large, high-level object-oriented systems and from the separation of concerns literature, this paper classifies separation of concerns problems into a number of categories and chooses representative examples from each. I intend to continue this work by evaluating a range of current systems using these problems as benchmarks, and then using this experience to design better support for separation of concerns in the context of high-level object-oriented languages like Cecil.

1 Challenge Problems

Challenge problems have long been useful in evaluating program design and implementation techniques. Parnas [P72] proposed the KWIC system as a benchmark to evaluate designs for ease of change, and showed that many kinds of change were easier in designs that used information hiding as a criterion, as opposed to a straightforward top-down functional decomposition. Object-oriented languages support good designs in the KWIC system and many others.

However, other kinds of problems are harder to modularize with current language technology. A set of challenge problems is needed to pinpoint the weaknesses of current systems and evaluate techniques for a more clean separation of concerns. In this section, this paper proposes a set of problem categories that capture the wide range of separation of concerns problems in the literature and in practical experience. For each category, a representative challenge problem is chosen and other problems from the literature are mentioned. There is not enough space for a complete specification of each challenge problem, but the brief outlines given should be enough to capture many of the central issues in each problem category; future work includes more precise descriptions of these problems. The categories each encompass a significant range of problems and are largely orthogonal, but some overlap is justified because the different causes or rationales for problems in different categories may require different solutions, in general.

1.1 Functional Concerns

A *functional concern* is a piece of functionality provided for a client. The client may be the user, another program, or even another component of the same program. In an ideal decomposition, a functional concern could be isolated to a single module. However, standard object-oriented decomposition can prevent the programmer from encapsulating a piece of functionality in a single module. In some cases, a functional concern may be spread among several objects. In other cases, a large object may deal with several different functional concerns, which are grouped into one object because they all operate on a single set of data. Because mainstream object-oriented languages require a module to be an object, they cannot modularize many functional concerns well.

A challenge problem for modularizing functional concerns is separating different kinds of operations on a parse tree, a concern that has been suggested in the literature [TOHS99] and has come up in my own compiler work at the University of Washington. A good modularization would separate the methods and data members needed for each operation on the parse tree into its own module, so that one module would contain all the methods for type checking, another for tree display, etc:

```
Statement.typecheck() { ... }                    (in typechecking module)
Expression.typecheck() { ... }
```

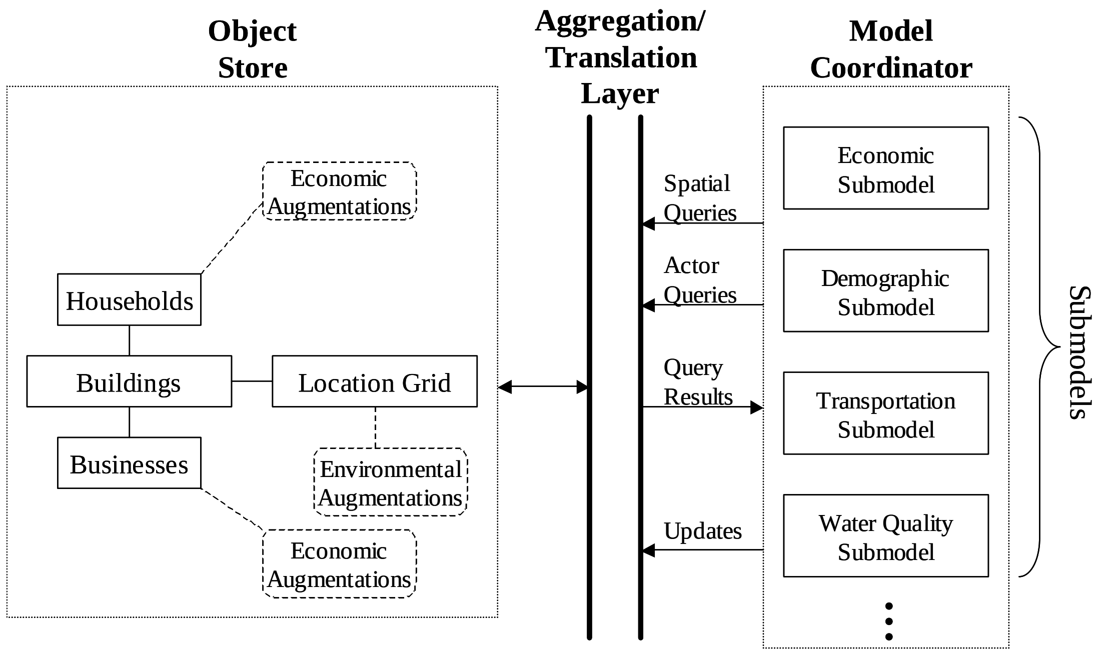

Figure 1. UrbanSim Architecture

Other modules, organized by class as in conventional systems, would contain the core data structure & methods for each class in the parse tree. The Cecil language [C92] designed at UW does not have a module system, but because Cecil has "open objects" it is possible to add methods and members to a class from outside that class's definition file. Thus, it is possible to encapsulate all the methods for performing one operation in a file devoted to a particular functional concern. This problem is also solved well in the Hyper/J [TOHS99] and AspectJ [K00] systems, which provide different ways to group methods by functionality rather than by class.

Other problems in the literature include the various actions of a bottle deposit machine like depositing and printing a receipt [RA92], various 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], and many others.

1.2 Program Organization

Program organization refers to the large-scale structure of the program, at the level of large interacting groups of objects. My interest in this concern arose from difficulty understanding Vortex [DDG+96], a compiler written in 100,000 lines of high-level Cecil code, without any documentation of the application's control-flow or module structure. Large applications often have a large-scale structure (or *software architecture*) in order to separate concerns at the highest level of the application; however, because the source code doesn't deal with any unit larger than a single object, this organization is rarely apparent from browsing the source code. This concern also arises when a single library supports a family of applications that are built by choosing a customized subset of objects and hooking them together to accomplish the desired behavior. Such program configurations are often awkward to specify in conventional object-oriented languages. Large-scale concerns like program organization are complimentary to smaller-scale issues like functional concerns, and play a similar role at a higher level of abstraction.

The challenge problem was inspired by an application simulating urban development over time [NBW00]. Figure 1 shows UrbanSim's software architecture. In order to provide a good structure for the simulation, a shared object store represents the current state of the simulation, and several independent model components implement various aspects of the simulation, such as economics, transportation, and demographics. This application structure is similar to the blackboard software architecture [GS94]. The challenge problem is therefore to explicitly represent a blackboard architecture, with an explicit connection diagram showing how the modules interact 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 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 the role-model work [RA92] can also be thought of as separating program organization concerns at design time.

1.3 Global Properties

Global properties are notoriously difficult to enforce in a modular way using existing technologies. Typically, custom code must be added at every point where the global property could be changed to ensure that the required property continues to hold. This makes it difficult to change system-wide properties, or to implement them in a different way. It may also be tough to guarantee that all possible places where the property could be violated have been found.

Examples of global properties include aliasing properties that prevent a private object reference from escaping a particular module [NVP98], various security properties, constraints that relate the values of variables spread throughout the program, global variables or constants, and synchronization between threads in a multithreaded system. My previous work in compiler analyses for eliminating unnecessary synchronization [ACSE99] was largely inspired by a desire to support cleaner synchronization specifications in an efficient way. Global property concerns are closely related to model checking and type system design.

Constraints are very useful in specifying flexible graphical user interfaces [B93] and are a good challenge problem for separation of global property concerns. The programmer should be able to declaratively specify linear constraints and inequalities between ordinary floating-point program variables:

```
constraint(window.right + offset < graphic.left);
```

When the value of one variable changes over the course of program execution, violating a constraint, an off-the-shelf constraint solver is invoked to find variable values that maintain the constraints, ideally leaving the trigger variable's value unchanged. The user should be able to design custom constraints (such as that two windows cannot overlap in a windowing system) and specify solvers to be invoked when variables involved in the constraint are changed.

1.4 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, tracing executions of a large number of methods, or handling common errors consistently throughout a large code base [LL00].

The challenge problem, invariant checking, was inspired by a performance-related change to the Vortex compiler. In the Vortex optimizer, sets of classes are represented in a number of ways, including singleton classes, all descendants of a particular class, enumerated sets, bit vectors, and other representations. I discovered a performance problem where some classes could not be put into bit vectors, causing very slow $O(n^2)$ behavior for set union operations. Putting these classes into bit vectors changed a key invariant used many places throughout the whole class set representation code, over 1500 lines of complex Cecil code. To verify that the new invariant was not being violated after my changes, I added checking statements to many different operations, but continued to discover bugs, because I had forgotten to add checks to some places where the invariant was affected.

A satisfactory solution to the invariant checking problem 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:

```
inherited invariant(X > 0 && ptr != NULL) on public methods(MyClass);
```

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, there should be a clean way to add an invariant to all public methods of a class without enumerating the methods (in case more are added later) and the invariant should be inherited by subclasses by default, even if they override the methods where the invariant is checked.

Other categories of hard-to-modularize concerns, such as enforcing global properties, can lead to repetitive code. Thus there is some overlap in this categorization. However, repetitive code is a significant issue in its own right, because it creates more work for programmers, makes it easy to forget a crucial repetition here and there, clutters up the resulting code, and makes change more difficult. Thus repetitive code is a useful design and evaluation criterion for systems that enhance separation of concerns.

1.5 System Performance

System performance refers to the performance of an entire subsystem of a program when performing a task for the user, or experiencing the load of a number of system clients. Examples include a user querying a large company database, or a large number of worldwide users purchasing items through a company's e-commerce system.

Conventional object-oriented programming systems can often encapsulate the performance characteristics of a particular data structure, so that it can be searched in $\boldsymbol{O}(\log n)$ time, etc. However, the problem is more difficult when the final performance of a system is a product of a large system of objects working together, and possibly distributed across multiple machines. Performance problems are difficult to characterize in general, and can be among the most challenging issues to debug in a particular system. Because of the diversity of modularity problems caused by performance concerns, this paper does not offer a single challenge problem in this area.

Performance represents a modularity problem because the factors underlying the performance are often spread out through the entire code base of a system, so it is difficult to solve a performance problem by looking at a single, key part of the system. In addition, improving the performance of a system can interfere with modularity in other was, such as by tying objects too closely together. Thus it is important to make the performance characteristics of a system clear from the overall design, and it is important to be able to make performance improvements without compromising the modularity of other concerns.

Although it is difficult to imagine a programming technology that makes all performance problems easy to modularize, technologies designed to address other categories of modularity may improve system performance modularity as a side effect. For example, making the software architecture of a system explicit may make it easier to change the configuration of a system to improve performance, perhaps by duplicating a bottleneck object across multiple different objects. Features for encapsulating repetitive code make it easier to do effective logging, which can pinpoint the performance bottlenecks in a system.

1.6 Special Purpose Concerns

Some *special purpose concerns* have proven especially difficult to modularize because the concern is best specified in a declarative or graphical way, rather than with standard imperative code. For example, graphical user interface design is usually done with a graphical editor, which allows the programmer to lay out components visually on the screen and write snippets of code to connect the components to the core application. When GUIs are specified in imperative code, the result is often very repetitive and difficult to understand or change, because the textual medium is a very poor representation of the intended graphical result. Certain concerns that also fit into other categories (for example, constraints) may be best expressed as a domain-specific language. Finally, object initialization and traversal is often best specified in a declarative syntax [ML98].

The challenge problem for this area is providing a clean specification for a parser. Textual file formats are most easily specified in the declarative BNF syntax, and so the easiest way to write a parser is to annotate a BNF description of the language with actions to take when a particular construct is parsed. Programs such as YACC have been constructed to generate parsers automatically from these annotated BNF files. But tools like YACC do not always fit well into standard development environments, can be difficult to customize, and may not be available for all languages (including Cecil). A good solution to this challenge would support a file format notation recognizably close to BNF within the programming language, and without a lot of repetitive declarations. Actions to be taken should be specified in a notation as natural as the notation used in tools like YACC. The whole thing should be done in the framework of a single extensible language, so that the parsing code is smoothly integrated with the rest of the application:

```
rule class_decl ::= CLASS class_name { /* add to symbol table */ } ...
```

It may be impossible to effectively express all special-purpose concerns in a single, flexible language. Certainly it would be very challenging to seamlessly incorporate a graphical GUI designer into an imperative programming language. However, an extensible language with a way to embed declarative syntax could provide a natural solution to issues like parsing or object traversal, as long as the extensibility mechanism does not make the language significantly harder to understand. Even if a specifically designed domain specific language is still superior for the particular concern, there are costs to using more than one language in a single project that may make a language extension mechanism more attractive. In any case, good support for integration with a variety of domain-specific languages can allow programmers to express each concern in the most modular and appropriate way.

1.7 Context Dependent Behavior

A final concern that can be difficult to modularize effectively is *context dependent behavior*. Object-oriented programming allows different code to be executed depending on one particular *context*—the class of the receiver object. The receiver class context has proven extremely useful in practice. However, other contexts may also be useful in selecting the appropriate behavior. Multi-methods, present in Cecil, CLOS, and other high-level object-oriented languages, allow executable code to be selected based on the classes of all the arguments to a method, not just the class of the receiver object. Predicate dispatching, which chooses code based on variable values and structures, has long been used in functional programming and has recently been integrated into an object-oriented research language [EKC98]. Error handling by specifying exception handlers or by passing closures to be invoked in an error situation also allows custom code to be executed based on an error-handling context.

The challenge problem comes from the security domain. It is often important that sensitive methods can only be called from trusted system classes, or from privileged object instances. To solve this problem, a system should allow methods to be selected based on the calling scope [AT98]. Ideally, the programmer could specify dispatch based on the calling object's class, the particular object instance, or even data within the object. For some callers, the invoked method would perform the indicated task; for others, there might be restrictions on the task or an exception might be thrown.

```
method sensitive() with caller module(System) | privileged_inst { ... }
method sensitive() /* no caller specified */ { throw new Exception(); }
```

2 Conclusion & Future Work

This paper proposed a number of challenge problems for effective separation of concerns in object-oriented systems. It includes a comprehensive categorization of a wide range of important concerns that are difficult to modularize, encompassing most if not all of the examples given in the separation of concerns literature. Different proposed systems designed to separate concerns more effectively have attacked subsets of these issues. For example, the AspectJ [K00] and Hyperspaces [TOHS99] projects both attack functional concerns and repetitive code from different perspectives. Many high-level languages have improved concerns that deal with context dependent behavior, and composition filters [AT98] also seem well suited to this domain. Domain-specific language generators such as those in GenVoca [BST+94] attack special-purpose concerns, as do a number of language extensions such as the traversal strategies in Demeter [ML98]. Type system work, model checking, and logic programming languages have focused on various global properties, and there is some support for these in systems like AspectJ. Systems from the software architecture field have focused on program organization, although many of these solutions are not well integrated into a programming language.

In future work, I intend to evaluate the strengths and weaknesses of a number of different systems using the challenge problems as guides. I will also develop a synthetic benchmark that unifies problems from all the different concern categories into a single application, so that I can see how the different concerns might interact. Using this experience as a guide, I will then look at ways to extend Cecil 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

[ACSE99] J. Aldrich, C. Chambers, E.G. Sirer, and S.J. Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In Proc. of 6th International Static Analysis Symposium, September 1999.

[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.

[BST+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.

[DDG+96] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In Proc. of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1996.

[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.

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

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

[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.

[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

[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.

[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.

[RA92] Trygve Reenskaug and Egil P. Anderson. System design by composing structures of interacting objects. In Proceedings of the 1992 European Conference on Object-Oriented Programming, pages 133--152, 1992.

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

[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.