Sorting out Concerns*

Lee Carver and William G. Griswold
{lcarver, wgg}@cs.ucsd.edu

Computer Science and Engineering
UC San Diego
La Jolla, CA 92093

Abstract

Realizing the dream of additive, rather then invasive, software development requires support for the components and composition methods used to implement production quality software. The invasive changes common in most development indicate the inadequacy of standard practice. Several recent proposals (Hyperspaces, Aspect Oriented Programming) suggest that the problems are inherent in hierarchical nature of the standard mechanisms. These proposals define new mechanisms that can support a richer set of software construction methods.

These new mechanisms have been used to guide a dissection of Gnu `sort.c` (2145 text lines). The goals of this experiment include the identification and analysis of concerns and composition mechanisms used in real, production quality software. Preliminary results indicate that the new proposals do a good job of describing many forms of program composition. Although 60 concerns have been identified, the interactions between 27 of them can be described by a single hyperspace. This hyperspace describes most of the program's external behavior. It consists of one dimension with 20 configuration concerns cross-cutting 3 dimensions defined by 7 feature concerns. However, some compositions are more troubling. In 3 cases, adding a concern inserts code into the middle of already established behaviors. These compositions appear to violate the notion that join points can be restricted to individual methods or operations.

1. Introduction

Any non-trivial software system is the conglomeration of concerns and algorithms. Ideally, each concern and algorithm would be separately specified and then composed to form the final system. Such an approach has great intellectual appeal and numerous pragmatic advantages. The advantages include support for evolutionary software development, increased reuse of code and other artifacts, improved robustness, and shorter time to market.

Unfortunately, too much of development involves invasive and repetitive modification of a program text. The inability to isolate the modifications for a change indicates an inability to maintain a separation of concerns. A key factor in this inability is inadequate techniques for organizing concerns and defining their composition. Ideally, each concern is encapsulated as a single, self-contained software module. Several mechanisms for defining modules are in common practice. These include layering[1], stepwise refinement[2], and information hiding[8]. These mechanisms underlie much of the progress in software engineering, and they are fundamental to the notion of object-oriented programming.

An important insight is that standard practice allows only some concerns to be separated. The conventional mechanisms rely heavily on hierarchical structures. This does allow ready separation of many concerns. However, it is inadequate when there are multiple independent concerns. Features, performance, and error handling are independent and common concerns in many production quality applications. Standard practice uses the features to define a module hierarchy. This forces the performance and error handling concerns to be dispersed throughout the feature modules. These dispersed concerns are often referred to as cross-cutting aspects. The use of a single hierarchy forces independent concerns to be dispersed across the software modules.

Recently, several new mechanisms have been proposed for organizing and composing concerns. These new mechanisms directly support non-hierarchical collections of concerns. Aspect oriented programming (AOP)[4] and subject oriented programming (SOP)[5] provide mechanisms to integrate modules from independent hierarchies. The concern hyperspace model supports a behavioral definition of concerns and allows concerns to be organized into multiple dimensions[7].

These mechanisms create a new context for understanding program structure. We expect that these new mechanisms will allow us to completely describe `sort.c`'s structure. This description will be complete in the sense that every character in the source is associated with at least one concern. Its modest size (2145 text lines) allows for a detailed analysis. Despite its small size, it offers a rich set of capabilities: large file support, complex sort keys, and excellent performance. We expect hyperspaces to provide simple representations for concern interactions and AOP/SOP to provide mechanisms for generating the application from the identified concerns.

Traditional notions of software composition and architecture fail to describe all of the interconnected behaviors in the program. The interactions between `usage()` text, command line parsing, configuration options, and sorting features cannot be described hierarchically. The hyperspace model captures these connections with four dimensions. The traditional mechanisms also fail to explain the program's synthesis or evolution. The transition from an object model to source code is essentially a black art. Hyperspaces and hypermodules offer a way to experiment with clusters of concerns. AOP and SOP allow a developer to realize code from these clusters.

We expect that this experiment will validate the theories of software hyperspaces and to provide practical experience with AOP and SOP. Early results indicate that hyperspaces, AOP,

*This research was supported in part by NSF grant CCR-9508745 and took place in part while the second author was on sabbatical with the AOP group at Xerox PARC.

and SOP do good job of supporting the methods used to implement production quality software.

The following sections provide details on this experiment. Section 2 introduces the concepts used throughout this document. Section 3 describes how concerns are identified and isolated from the rest of the program source. Section 4 presents some preliminary results based on the concerns that have been identified. The final section presents some conclusions and suggests areas for future work.

2. Concepts

In order to enumerate and classify the concerns in `sort.c` a concise definition of concerns is required. The general notion of a concern is any coherent issue in the problem domain. This definition covers very complex situations, but provides little guidance for the enumeration of the practical concerns within a program.

The complementary notion of minimal subsets and minimal increments provides the required guidance for enumerating concerns. It is especially appropriate for concern hyperspaces, since minimal subsets define a mapping from program text to concerns. Each subset or increment defines a single concern. The code that implements each increment is the code that addresses that concern.

Concerns

An essential technique in software engineering is the separation of concerns[3]. In normal practice, this is mostly a philosophical approach to the task. Clear guidance on the use and practice of this technique can provide a more constructive element in software development. A practical definition of a concern is needed.

The basic notion of a concern is that it is any coherent issue in the problem domain. This definition allows the notion of concern to deal with arbitrarily complex concepts. Unfortunately, it gives no guidance on which concerns are reasonable and which concerns should be separated. The utility of a concern is a practical matter. "Handle Errors" is obviously useful for many programming tasks. "All occurrences of the letter *s*" might be useful in a translation task.

In the hyperspace model, concerns are associated with software artifacts and are expected to answer the query: Does software entity X address concern Y? It is often convenient to generalize this to the case where the queried software entity is not a realized artifact. It might be an abstract construct or a derived element. However, practical considerations require that some representation be provided for an entity that is manipulated by a hyperspace implementation.

Several common terms can be defined as specializations of the term concern. A *requirement* ("sort lines of text") is a concern that is defined externally to the software. A *capability* ("compare two keys") is a concern that is (expected to be) bound to executable code. An *aspect* is primarily a concern that affects a dispersed set of modules. These terms are useful when a restricted set of concerns is under discussion. Concerns can also be used to model a whole raft of other software terms like mode, feature, module, component, and function. These terms denote concerns only in the sense that they define a concise predicate over the software entities.

The concerns that we have identified come from a single artifact: the code. This limits the class of concerns that can be identified. The identified concerns are largely functional capabilities such as sorting, key extraction, and file transfers. Concerns related to global issues are difficult to attach to specific program text. This includes the "-ilities" such as maintainability, supportability, and traceability. We assume that these concerns exist in other artifacts (requirements, designs), even when these artifacts are not physically realized.

Minimal Increments

The basic notion of a concern is too general to identify the concerns that are useful for software development. We have found that the complementary notion of *minimal subsets*[9] is an effective way to identify concerns that are practical to software development.

The minimal subsets method is normally used to construct software. The first step is to define a minimal subset of the desired capabilities that might conceivably perform a useful service. Next, one defines a set of minimal increments to the system. The goal in defining the core subset and the increments is to avoid components that perform more then one function. By including or excluding different sets of increments, a developer can readily create a whole family of applications[10].

Hyperspaces and minimal subsets work well together. Each subset and increment defines a concern. The software entities that address such concerns are simple to determine. The code that implements the minimal increment is the code that addresses that concern. In reverse, hyperspaces provide an effective way to manage a large number of small increments.

The dissection of `sort.c` reverses the construction process. Capabilities that are not part of the minimal subset are relegated to an increment. The minimal sorting program simply reads a small file, sorts the lines, and writes the result. This minimal program contains a number of minimal subroutines. The minimal sort routine appears to be a 51-line merge sort. Other minimal routines include managing the operating system transfers, reading the file, and writing the result.

3. Dissection

The first goal of this experiment is a catalog of useful concerns that completely describes `sort.c`. Every character in the source text should be assigned to at least one concern. Our progress on this takes advantage of an understanding of the basic features and the apparent subsets in the program. Any capability outside the minimal subset must be a separate concern. The mapping of concerns to source text has been a largely manual process.

The Corpse

A brief summary of `sort.c`'s capabilities and structure is a useful guide for the remaining discussion. The syntax for the traditional UNIX sort is:

```
sort –mcubdfinrtx [+pos [-pos]] … [-o output]
    [-T directory] [file]
```

The `-m`, `-c`, and `-u` options activate the merge only, check only, and unique records behavior. The `+pos` and `–pos` pairs

define multiple pairs of keys. The `-dfinbr` options change the key comparison behavior. These changes include blank compression, character translation, and reverse ordering. By default, sort reads from standard input, writes to standard output, and uses `%tmp%` as the location for intermediate files. The `[file]`, `-o`, and `-T` options allow the user to specify alternatives. The other options tweak the standard behaviors in specialized ways.

The Gnu `sort.c` extends these options is a number of ways. It adds the POSIX `-k` option for specifying a key. It supports additional comparison criteria, including numeric and month name ordering.

The implementation is quite efficient and robust. In order to minimize system resource usage, large files are broken into sections (approximately 32K lines) that are sorted and then merged. The internal processing of `sort.c` follows this pattern:

```
read parameters;
for each input file {
   break the file into sections;
   sort each section into a temporary;
   }
for each cluster of 16 temporaries
   merge the temporaries;
```

Temporary files are carefully managed so that the final result is directed to the output file, not to another intermediate. There are numerous internal checks for special conditions, such as an input file specified as the output file. Overall, it is an extremely robust program.

Concern Identification

The notion of minimal subsets is the primary criteria for identifying a concern. Any capability beyond the minimal subset must be a separate concern. This standard can also be applied to isolated to concerns and occasionally results in fine-grained structure within a single concern.

The minimal version of `sort.c` that retains sorting capabilities is:

```
read a "small" file from stdin;
sort the input;
write results to stdout;
```

This program does not use any command line options, any temporary files, or allow any specification of keys. It does provide a useful subset of the full program, sorting files that are smaller then one section. This could be combined with independent programs to section a file, manage a group of files, and merge a group of files to achieve the original capabilities.

The minimal program can be further reduced in a number of ways. One can isolate the sort algorithm. This yields a 51-line module based on `sortlines()`, an implementation of merge sort. Other concerns to isolate include the file input support, the file output support, and various aspects of operating system coordination.

No base concern is readily apparent in the minimal subset. All of the routines present in the minimal subset could be used separately in other programs. Several of them could sport multiple implementations. For example, the file input module reads the entire file into virtual memory. Some systems can implement this directly with memory mapped IO. Not even the minimal program:

```
int status; void main() { exit(status); }
```

defines a stable base concern. It would be re-implemented for a non-UNIX environment or a shared library application.

Concern Isolation

Concern isolation creates a mapping from each concern to the program text that implements the capability. In the initial dissection, this was a largely manual process. It required detailed inspection of the program text and manual recording of the responsible source code. All mappings refer to the original Gnu-provided source code.

Visual inspection is often adequate to isolate small minimal increments. Larger concerns often require multiple passes to discover all of the code that implements a given capability. `grep`-based tools can help find all uses of critical terminology, and the C compiler can help find unexpected interactions between components. Better tools, especially with larger programs, would aid this process. For example, data flow and control flow analysis tools could help ensure that all program segments are identified.

For most concerns, even functional ones, the responsible source code is several disjoint code segments. Often there are separate segments for declarations, initializations, and the implementation. A range of line numbers identifies most segments. Occasionally a code segment involves only part of a line. This normally occurs when independent variables are declared on the same line.

Use of the unmodified source code has been an overly strict standard. It does prevent the unintended modification of the program behavior, but it unnecessarily complicates the isolation of concerns. Declarations should be rewritten one to a line. Certain comma expressions should be transformed to statements. Some repeated statement sequences could be consolidated into functions. These transformations should have no effect on program behavior or performance.

4. Preliminary Results

At present, we have identified 60 concerns in `sort.c`. These are largely functional concerns. Several uncounted concerns that cross-cut the main functional hierarchy will be added as the dissection proceeds.

Some of the most interesting results are the different forms of composition required by the program. The vast majority of composition steps simply add new behavior. However, the added behavior invariably affects multiple independent concerns. Simple hierarchical composition does a poor job of describing these interactions. Compositions based on AOP or hyperspace models is much more successful at describing these interactions.

A few troubling compositions redefine existing behaviors, rather then adding capabilities. These intrusive compositions have interactions that go beyond simple operation-level wrapping. Sometimes the join point is a sequence of locations, rather then a single point in the code. Other cases profoundly change the requirements of a component or introduce additional inputs to an operation.

Name	Description
Sort algorithm	The implementation of the sort algorithm.
File input	Read an entire file into virtual memory.
File output	Write a block of bytes.
OS coordination	Receive and release the execution thread.
Multiple sort orders	Support fields and different sort orderings.
Command line parsing	Decode `argv[]` elements.
Usage message	Output a simple help message.
Program name	Support external program names.

Table 1

Concerns

The currently identified concerns are, for the most part, elements of the functional hierarchy. The focus on minimal subsets is the major cause of this functional decomposition. We highlight a few of the identified concerns in Table 1.

The majority of the 60 concerns are unique to the sorting process. Only 18 of these concerns seem likely to occur non-sorting programs. However, sorting is a common subcomponent of many applications. The sorting concerns should be organized into clusters of related capabilities that can be used by future applications.

Additional cross-cutting concerns that we expect to isolate include error handling and performance concerns. It may also be possible to isolate some concerns that address global software issues. Some sections of the program are readily seen to implement the portability and maintainability concerns.

Additive Compositions

Much of the behavior in `sort.c` could be constructed as the result of a series of composition steps. The majority of these composition steps are strictly additive. Although the program is constructed using only hierarchical mechanisms, these mechanisms are largely inadequate to describe the interactions introduced by each incremental concern. The richer models provided by AOP, SOP, and hyperspaces are required to describe the interactions among concerns in production code.

The simplest compositions are those that occur when one feature extends only one other feature. This situation can be adequately described by traditional hierarchical mechanisms. In `sort.c`, this applies only to certain high-level views of the program architecture. Abstracting away the real world concerns of error handling, performance, and user interface can provide models that are describable with strictly hierarchical mechanisms. The fine-grained concerns are implemented with hierarchical mechanisms, but they all interact with features from different hierarchies. Even simple features like setting the temporary directory extend the user interface component and the usage message component.

The next step up in complexity occurs when one feature extends multiple features. This form of composition uses operation-level join points[6]. These compositions can be described using AOP or SOP mechanisms. The error handling, overwrite protection, and parts of the multiple sort order concerns can be described by this form of composition.

The next level of compositional complexity is cross-product interactions between multiple features. Hyperspaces provide a powerful mechanism for describing this form of composition. This mechanism is appropriate for many of the capabilities in `sort.c`. One 4 dimensional hyperspace provides a simple model for 27 of the 60 identified concerns. The 20 concerns associated with configuration options, such as case folding, define one dimension. The user input dimension is defined by two concerns: command line parsing for global options and command line parsing for individual fields. The dimension containing the usage message concern has only the one concern. Other points in this dimension are unimplemented. The feature dimension ties together four additional concerns that address sorting modes, input handling, file names, and field support. The numerous configuration concerns that cross-cut each of the other dimensions makes hyperspaces a better description of these interactions.

Invasive Compositions

The addition of some concerns cannot be described with the standard composition mechanisms. In these extensions, code changes are not limited to operation level weaving. These extensions change the internal behavior of existing code in a more complex fashion. One case collects and transforms sequences of operations. Another case adds new requirements that force the use of a specific sort algorithm. These concerns tend to inject code into the middle of existing capabilities.

This form of composition indicates that unexpectedly complex interactions exist between multiple concerns. Some occurrences introduce new global requirements. Other cases occur when the natural composition order is reversed with respect to the concern interdependencies. This reversed order case might indicate that the developer has used an inappropriate composition order. Changing the order of composition can convert some invasive compositions to additive compositions. Further research is needed to distinguish these cases.

A case of invasive composition occurs with the program name concern. The program name concern deals with an external name used to run a program that differs from the internal name given by the developer. In UNIX, `argv[0]` is the name used to run a program. Well-behaved UNIX programs use this value when they display messages for the user.

As a simplified example, adding the program name concern transforms error messages from

```
fprintf(stderr, "sort: failed to open %s\n",
   szFilename);
```

to

```
fprintf(stderr, "%s: failed to open %s\n",
   argv[0], szFilename);
```

This is an extremely intrusive change to the code. Two coordinated changes are required: one to the format string and one to the parameter list. Both of these changes can have complex interactions with the pre-existing context. In `sort.c`, this context includes two occurrences of the program name intermixed with other independent format substitutions.

This case is further complicated by the question of composition order. The conventional ordering adds the program name concern to a system that already supports error messages. This emphasizes the significance of the error messages, but leads to invasive behavior changes. Reversing the composition order results in an additive extension; error messages simply use the program name concern. However, this case should not be dismissed as an instance of poorly ordered composition. The reverse order fails to capture both conventional practice and conventional understanding of the interactions between the concerns. The criteria for determining the appropriate composition order are an open issue.

A more complex case of intrusive composition occurs within the basic sort function. The minimal sort program compares every byte on every record to determine sort order. The minimal `sortlines()` routine could implement any sort algorithm. When keys and fields are added, parts of a record might be ignored and the need for a stable sort is introduced. The addition of keys and files forces the sort routine to use the merge sort algorithm.

A third case occurs because of a performance interaction between `sortlines()` and its driver `sort()` routine. To minimize memory allocation costs, a single work array is allocated in `sort()` and reused within `sortlines()`. This optimization requires extra parameters in `sortlines()` and eliminates some memory allocation code that would otherwise have been present. The insertion of parameters is an unexpected change to an existing capability. However, this may be an artifact of the function-based programming style used in `sort.c`.

Coping with Multiple Join Points

The addition of the program name concern seems to require unconstrained changes at the expression level. Such transformations complicate, rather then simplify, program construction. This complexity can be reduced if `printf()` is treated as an optimization. The error message above can be rewritten as a sequence of `fputs()` statements:

```
fputs("sort", stderr);
fputs(": failed to open ", stderr);
fputs(szFilename, stderr);
fputs("\n", stderr);
```

Adding the program name concern substitutes the statement `fputs(argv[0], stderr);` for the first statement. Text output is especially susceptible to this kind of transformation. `printf()` statements can be decomposed into a sequence of `fputs()` statements. These can be further decomposed into a sequence of `fputc()` statements.

The appropriate transformation appears to be that any (sub-) sequence of operations can be a join point for an added concern. Although this greatly expands the range of join points, it is much simpler than unconstrained expression level changes. The addition of concerns that independently transform overlapping sequences is a potential problem that needs further investigation.

5. Conclusions and Future Directions

This detailed analysis of a production application has been a very interesting approach to software engineering. Non-hierarchal descriptions of software systems are essential when describing the feature interactions in production code. Even simple functions impact multiple capabilities from separate areas of concern. The notions of hyperspaces and join points make it feasible to describe these interactions.

Although many features can be introduced through additive composition mechanisms, there are a few cases that violate this rule. The addition of the program name concern and the stable sort concern cause intrusive changes to existing code. It appears that sub-sequences of operations must be admissible as join points.

Much work remains to be done. The identification and isolation of concerns from `sort.c` needs be completed. Several of the most strongly tangled concerns are still jumbled together in one knot. In particular, the code that implements fields, keys, and their various sub-concerns is currently one large cluster of code segments.

The creation of a comprehensive hyperspace model is still pending. The hyperspace model has been effective for describing the interactions between features, command line parsing, and usage message text. These interactions form a dense region in the hyperspace model. The model needs to be expanded to cover the more hierarchical regions of software composition. We expect this to define sparse areas in the hyperspace.

The re-implementation portion of the experiment has just begun. We intend to develop the new sort as an evolutionary product. A program implementing the full set of features will be constructed by incremental addition of new capabilities. This should provide an opportunity to experiment with different composition sequences. We hope that this process will help define the language features required to implement production quality applications.

This kind of experiment should be repeated on a number of other programs. Additional experiments on other programs of similar size would validate the techniques and should expand the set of known concerns. A separate direction is to expand the size of the application that is analyzed. Modern commercial software often exceeds a million lines of code. A useful intermediate step would be an application with 10,000 to 50,000 lines. The extraction process would need to be partially automated to support a body of source at this size.

References

[1] E. W. Dijkstra, The Structure of the "THE"-Multiprogramming System, *Communications of the ACM*, 11(5): 341-346, May 1968.

[2] E. W. Dijkstra, Notes on Structured Programming, In *Structured Programming*, Academic Press, pp. 1-82, 1972.

[3] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin, Aspect-Oriented Program-

ming, In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[5] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal, Specifying Subject-Oriented Composition, *TAPOS*, 2(3): 179-202, 1996.

[6] H. Ossher, P. Tarr. Operation-Level Composition: A Case in (Join) Point. In *Proceeding of the Aspect-Oriented Programming Workshop at ECOOP'98*, http://www.parc.xerox.com/spl/projects/aop/ecoop98, pp. 116-119.

[7] H. Ossher, P. Tarr. Multi-Dimensional Separation of Concerns. IBM Research Report RC 21452, 16 April 1999.

[8] D. L. Parnas, On the Criteria To Be Used in Composing Systems into Modules, *Communications of the ACM*, 15(12): 1053-1058, December 1972.

[9] D. L. Parnas, Design and Specification of the Minimal Subset of an Operating System Family, *IEEE Transactions on Software Engineering*, SE-2(4): 301-307, March 1976.

[10] D. L. Parnas, Designing Software for Ease of Extension and Contraction, *IEEE Transactions on Software Engineering*, SE-5(2): 128-137, March 1979.