Decoupling Change from Design

Michael VanHilst and David Notkin
Department of Computer Science and Engineering
University of Washington
PO Box 352350
Seattle, Washington 98195 USA
{vanhilst,notkin}@cs.washington.edu

Abstract

Parnas' seminal 1972 paper, "On the Criteria To Be Used in Decomposing Systems into Modules," identified simplifying change as a critical criterion for modularizing software. Successful designs are those in which a change can be accommodated by modifying a single module. There is a tacit assumption in most of the literature that once a change has been limited to a single module, the cost of making the change is essentially inconsequential. But modules have complexity of their own and are frequently large. Thus, making a change can be expensive, even if limited to a single module.

We present a method of decomposing modules into smaller components for the purpose of supporting change. Although similar to the approach of modularizing programs described by Parnas, our approach is specific to decomposing modules. It is not intended to replace traditional high level modularization but rather to augment it with a second level of modularization where the standard of information hiding can be relaxed. The goal of the method is to make modules easier to change by decomposing them around smaller design decisions—ideally encoding only one design choice per submodule component.

In this paper we show how submodule components can be used to address the issue of change. We also demonstrate how the ability to address change with submodule components is, to a large extent, independent of the design level modularization. Moreover, we show that, at least in some cases, by using submodule components the choice of high level modularization can itself be changed without having to rewrite large amounts of code.

A method of implementation is presented using inheritance, parameterization, and static binding in a way that minimizes implementation dependencies between components. The method supports fine grained decomposition with flexible composability and almost no runtime overhead.

SIGSOFT '96 CA, USA
© 1996 ACM 0-89791-797-9/96/0010...$3 50

1 Introduction

Much of the literature on modularizing programs discusses the goal of isolating potential changes to a single module. But the literature is silent on the simplicity or complexity of making changes *within* a module. The implication is that changing one module is a straightforward task. But is this always the case? Are there design approaches for implementing modules that make them easier or harder to change?

Consider Parnas' KWIC example, which reads lines of text and outputs all the circular shifts of all the lines in sorted order [12]. The preferred modularization of KWIC had five modules—line storage, input, circular shifter, alphabetizer, and output. The dominant reason for preferring this modularization to a more conventional functional decomposition was that it isolated a given set of likely changes to one or at most two modules. For example, to change the internal storage of shifts from an index to actual lines of text, only the implementation of the circular shifter module had to change. But how difficult was it to make that change? No answer to this question was given or even alluded to in the Parnas paper.

Some studies suggest that it is generally easier to replace a module, or add a new one, than to alter the implementation of an existing module [14, 18]. How difficult would it be to build a new circular shifter? The difficulty of changing or building a module necessarily depends on its complexity. Complex modules should be subdivided into smaller, less complex components. What strategy is best for decomposing modules?

A common approach to subdividing modules is to apply the original method of decomposition recursively. But this is not always easy or appropriate. For example, Parnas' preferred KWIC decomposition is based on information hiding and abstract data types. Each module hides the details of a data structure and encapsulates operations on it. For the circular shifter module this means hiding the representation of the list of shifts while providing operations on it. There is one data structure and the operations all use hidden details: it would be hard to decompose it further with this approach.

In object oriented programming, decomposition by subclassing allows a general base class module to be composed with different specialization parts. The circular shifter module could be decomposed into a general list base class and a specialization subclass that adds the behavior of the shifter. A subclass hierarchy forms a tree (or DAG) where each arc represents an increment of specialization. Parnas described a similar graph structure for the family of possible programs using stepwise refinement, where each arc represented a design decision [13]. To change a design decision, one starts at the node before that decision's arc, and continues anew with design decisions, ignoring or revisiting decisions made after that point in the development of the previous version. Decisions that may change are deferred, since later decisions are less disruptive to change. But not every decision can be deferred. For example, changing the shifter's internal storage as described above changes the base class part rather than the specialization.

In this paper we present a method of decomposing modules into smaller components for the purpose of supporting change. Although similar to the approach of modularizing programs described by Parnas, our approach is specific to decomposing modules. It is not intended to replace traditional high level modularization but rather to augment it with a second, lower level of modularization. As will be seen, the approach is qualitatively different in that we relax the standards of information hiding to support finer integration among components within a module. The goal of the method is to make modules easier to change by decomposing them around smaller design decisions—ideally encoding only one design choice per module component. A method of implementation is presented using inheritance, parameterization, and static binding in a way that minimizes implementation dependencies between components. The method supports fine grained decomposition with flexible composability and almost no runtime overhead.

In two earlier papers we compared our method of implementation to frameworks with respect to flexibility and performance [16] and discussed using our approach to implement role based components as an extension to collaboration based design [17]. In this paper, we focus on submodularization as a useful and realizable method of decomposing modules in any design. Our concentration on submodularization implies that the technique naturally scales; that is, it can be applied to as many modules as produced during high level design, with the complexity of applying the technique to any single module depending only on the size and intricacy of that module.

Section 2 describes both a method for implementation and an approach for identifying the appropriate components. Section 3 analyzes submodularizations of the KWIC application with respect to various suggested enhancements. The results of this analysis suggest that the simplicity or complexity of many enhancements is more strongly affected by the use of submodule components than by the design level modularization. Section 4 discusses the approach. Section 5 presents related work, with conclusions presented in Section 6.

2 The Method

Our approach to decomposing modules into components is based on the goal of minimizing the number of design choices per component, with the ideal being one design choice for each component. By design choice we mean a decision that, if changed, would produce a system that was also meaningful, but different from the original in some significant way. Example design choices might be the type of data structure, the type of algorithm, whether a module communicates with another module using local or remote procedure call, etc. In this section we describe the method we use to implement components, and then compose them, that makes this type of decomposition possible.

The KWIC circular shifter module, as described by Parnas, encodes four design choices—the interface imported for accessing lines of text from another module, the algorithm to create the shifts, the data structure to store the shifts, and the interface exported for clients to access the shifted lines. In our approach, the shifter module is composed of four components—GetLineImport, Shifter, ShiftIndex, and GetShiftedLine—that encode each of the four design decisions. GetLineImport provides access to text lines stored in a module exporting a GetLine interface, Shifter creates implicit shifts for each line accessed through GetLineImport, ShiftIndex is the data structure that holds the index of implicit shifts, and GetShiftedLine creates the interface for clients

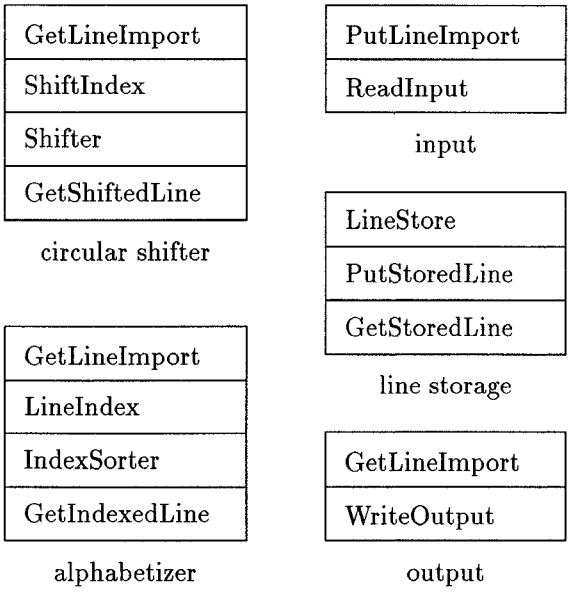

Figure 1: Submodularizations of all five modules in Parnas' KWIC modularization.

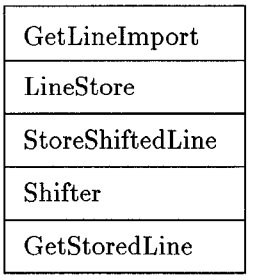

Figure 2: Submodule components in a circular shifter module modified to store lines of text in place of index values.

to access shifted lines using the shift index. Figure 1 shows submodularizations of all five modules in Parnas' KWIC modularization.

With the submodularization of the circular shifter module described above, changing the storage of shifts from implicit shifts to explicit ones requires replacing two of the components, ShiftIndex and GetShiftedLine, with three new components: LineStore to store lines of text, GetStoredLine for the interface through which clients access lines from LineStore, and StoreShiftedLine to translate calls by the Shifter to store implicit shifts into calls to store explicit ones. The new submodularization is shown in Fig. 2. But two of the new components are the same as components in the line storage module. The implementations of those two components can be reused as is. Only the new StoreShiftedLine component needs to be written—a smaller task than changing the entire circular shifter module. This example demonstrates both a narrowing of scope and the opportunity to reuse code among modules.

2.1 Method of Implementation

Mapping submodule components from the design to an implementation requires a method of implementation that supports small reusable components. Unfortunately, with current methods of implementation, supporting large numbers of small reusable components is neither easy nor cost-effective. First, when implementing components it is hard to avoid including dependencies on an application's structure and on the implementations of other components. Dependencies become encoded, for example, when a component's implementation includes the type and location of another component. Second, support for interchangeable components often entails a significant cost that increases with the number of supported components. Costs can include runtime costs for context switching and levels of indirection, as well as complexity costs for the scaffolding added to isolate those components that can change.

Our approach uses a method of implementing components that addresses both component independence and also the cost of composability. The method combines features in an object oriented language—namely inheritance, static binding, and type parameterization—in a stylized way to implement the components and to compose them at compile time to form the modules of an application [16, 17]. Briefly, in our method, components are implemented as subclasses of an initially unspecified superclass—that is to say, the type of the superclass is parameterized. References to the types of other modules are parameterized as well. Components are composed with other components by binding types to parameters. We use the C++ template mechanism for parameterization, and either typedef statements or class definitions for the bindings to types (although the implementation strategy is not specific to C++).

Figure 3 shows the implementation of the Shifter component. The type of the base class is deferred by the SuperType template parameter, allowing calls to components not yet selected. The sequence of class definition statements in Fig. 4 composes the Shifter with the other three components mentioned earlier to form the class of the circular shifter module for the KWIC application. The additional parameter in the

```
template <class SuperType>
class Shifter : public SuperType {
public:
    void shiftLine(int l) {
        int num_words = words(l);
        for(int w=0; w<num_words; w++)
            addShift(l,w,num_words);
    }
    void initializeShift() {
        int num_lines = lines();
        resetShift();
        for(int l=0; l<num_lines; l++)
            shiftLine(l);
    }
};
```

Figure 3: The Shifter component implemented as a C++ class template.

```
class shifter1 : public
    GetLineImport<StoreClass,emptyClass> {};
class shifter2 : public
                    ShiftIndex<shifter1> {};
class shifter3 : public
                        Shifter<shifter2> {};
class ShifterClass : public
                GetShiftedLine<shifter3> {};
```

Figure 4: Parameter bindings to compose components into a class for the initial KWIC circular shifter module.

GetLineImport component is for the type of the line storage module.[1]

Deferring the type of the base class allows the implementation of the Shifter component to call methods in two other components without referring to their names or their types. The implementation assumes only that the needed methods are part of the interface of the inherited superclass. The composition specified in Fig. 4 binds the resetShift() and addShift() calls to methods in the ShiftIndex component. Similarly, the line() and words() calls are bound to methods in the GetLineImport component. (As explained below, GetLineImport is a proxy that calls methods in another module.) This example illustrates how our approach supports component independence by eliminating any dependence that IndexShift might have on the type or location of the other components that it uses.

[1]EmptyClass is a default base class.

```
class shifter1 : public
    GetLineImport<StoreClass,emptyClass> {};
class shifter2 : public
                    LineStore<shifter1> {};
class shifter3 : public
                StoreShiftedLine<shifter2> {};
class shifter4 : public
                        Shifter<shifter3> {};
class ShifterClass : public
                GetStoredLine<shifter4> {};
```

Figure 5: Parameter bindings to compose an alternative circular shifter module using text line storage instead of an index.

Our approach does not require additional runtime scaffolding to support composability. The binding of calls to methods in our implementation is performed at compile time, as shown in Fig. 4. Using compile time binding avoids indirection in the call from one component to another and allows the method call overhead itself to be inlined away. Thus, our method of submodularization adds neither indirection nor context switch overhead. Compile time type checking also prevents interface mismatches from occurring.

Figure 5 shows code used to compose the alternate version of the circular shifter module that stores lines of text instead of an index. In this case, as mentioned above, the LineStore and GetStoredLine components are reused from the line storage module. In this version of the composed circular shifter module, the resetShift() and addShift() calls in the Shifter component are bound instead to methods in the StoreShiftedLine component.

Our method of implementation allows components to themselves be composed of other components. Figure 6 shows the GetLineImport component in terms of two smaller components, Handle and GetLineProxy. The Handle component provides a handle to the module with the actual method bodies, addressing the questions of where and how to access the other module's interface. The GetLineProxy component, as shown in Fig. 7, forwards local calls to line() and words() through the handle, addressing the question of what to import. In the template definition, the BodyType parameter is the type of the module to which the method calls are forwarded. Although the line storage module exports both get and put calls in its interface, here only the get part of its interface is made visible within the circular shifter module.

```
template <class BodyType,class SuperType>
class GetLineImport : public
GetLineProxy<Handle<BodyType,SuperType> >{};
```

Figure 6: GetLineImport in defined as the composition of two smaller components, GetLineProxy and Handle.

```
template <class SuperType>
class GetLineProxy : public SuperType {
public:
    int lines() {
        return handle->lines();
    }
    int words(int l) {
        return handle->words(l);
    }
};
```

Figure 7: Partial implementation of GetLineProxy (see Fig. 6) shows calls being forwarded to the interface of another component through a protected handle pointer.

2.2 Method of Design

For traditional high level design, choosing the right modularization is an art. Decisions that later turn out to be "wrong" can be costly. In our low level submodularization, the task is comparatively straightforward—recursively split components that contain two or more design choices. Here, getting the right level of decomposition is not as critical. If two decisions are split but always coincide, their components can be recombined later to form a single component. If a component is later found to contain separable decisions, it can at that time be replaced by the composition of two or more components providing the same interface. Only the specification of composition encodes the types and numbers of components, and thus only relevant parts of the specification must be changed. As will be shown in the analysis section, even the order in which design decisions are made may not be critical.

Traditional hierarchical decompositions make some changes easier than others. Such tradeoffs force developers to choose the design choices that are most likely to change. In our approach to decomposing modules, no special treatment is offered to make some changes easier than others. All design choices are considered equally likely to change. Once the high level modularization has been chosen, the strategy for submodule decomposition is simply to decompose beyond any changes considered possible.

Perhaps the more difficult part of our approach to submodularization involves deciding how to encode certain types of design decisions. Through experience with using our approach, we are developing classifications for types of design choices and guidelines for addressing each type. The two circular shifter modules, for example, include components that encode four different types of design choices: basic algorithm (Shifter), data structure (ShiftIndex and LineStore), interface translation (GetShiftedLine, GetIndexedLine, GetStoredLine, and StoreShiftedLine), and module interconnection (GetLineImport).

Getting the correct order of composition also poses a challenge. Method implementations must appear earlier in the composition than the calling sites that use them. When the same name is used twice, the order of composition plays a role in assuring that the correct bindings are made. For example, in the circular shifter module, line() and words() methods are defined in both the GetLineImport and GetStoredLine components. The order of composition shown in Fig. 4 binds the calls in Shifter to the methods in GetLineImport, while making the methods in GetStoredLine visible through the module's exported interface. A method of analysis to address the ordering of composition is discussed elsewhere [17].

3 Analysis

In this section we discuss how our submodularization affects the ease or difficulty of making changes. The section is divided into three parts. In the first part, we discuss changes to the KWIC application, applying them to our submodularization of Parnas' preferred modularization. In the second part, we discuss reconfiguring the KWIC application into other, different high level modularizations. In the third part, we discuss the changes again, but applied in the context of the other high level modularizations.

Figure 1 showed the submodularization of all of the modules in Parnas' preferred KWIC modularization. The order in which the components are shown is significant—it corresponds to the order in which they are composed—as, for example, can be seen in the circular shifter composition of Fig. 4. In this submodularization, the circular shifter, alphabetizer, and output modules all access text lines in the line storage module through imported interfaces provided by copies of the GetLineImport component.

3.1 Change

The original KWIC article listed five design decisions that were likely to change. These changes are paraphrased below.

1. Use a different input format (e.g. different end-of-line markers);
2. store lines in compressed form vs. store uncompressed text;
3. have all lines stored in core vs. use disk storage;
4. store index of circular shifts vs. store shifted lines; and
5. alphabetize list once vs. alphabetize on demand.

These changes prove to be fairly straightforward with or without submodularization. In the KWIC article, the first change, changing the input format, affected only the input module. In our submodularization, it affects only the ReadInput component of the input module.

In the original article's preferred modularization, the second change, adding line compression involved changing the implementation of the line storage module. In our submodularization, we simply add a LineCompress component between the GetStoredLine component and the LineStore component in the line storage module's composition. Put and get methods in the LineCompress component override methods of the same name in the LineStore component. The new put methods compress the line before calling their superclass's corresponding method, while the new get methods uncompress the line after calling their supertype's corresponding method. This change does not affect the implementation of the LineStore component, and can be applied to any version of line storage.

The third change, using a disk file for line storage, affects only the line storage module. In our submodularization of the line storage module, this change involves replacing the line storage module's LineStore data structure component. If the interface of the new component matches that of the original LineStore component, no other changes are needed. If the same calls cannot be used, then the two interface components would have to be changed. However, interface mismatches can also be addressed by the addition of a translation component, as was done for adding line compression.

The fourth change, saving shifted lines instead of an index of shifts, was discussed earlier. If this change is applied in addition to either the second or third changes, the changes in the handling of line storage would apply to both the line storage and circular shifter modules. With our component based submodularization, the same components are used for line storage in both modules—only one component has to change, or be added, in either case.

The fifth change calls for changing the sorting strategy from sorting once after all lines are read to some other strategy, such as partial or demand sorting. In the original paper this change was limited to the alphabetizer module. At our lower level, this change in the alphabetization algorithm involves only the IndexSorter component of the alphabetizer module.

A 1992 article by Garlan, Kaiser, and Notkin proposed an alternative modularization of KWIC based on tool abstraction and involving special modules known as toolies[7]. This article pointed out that the changes listed in the original KWIC paper were mostly data oriented, and that functional enhancements would be more difficult for the originally preferred modularization. The paper suggested an additional set of changes to consider. These changes are paraphrased below.

6. Omit shifts that start with members of a set of noise words;
7. include only shifts starting with specific words;
8. omit input lines starting with noise words;
9. include only input lines starting with specific words; and
10. include or exclude individual words.

The sixth change requires omitting shifts that start with certain noise words, while the seventh change includes only shifts starting with certain words. The toolie paper suggested that both of these changes should be included in the circular shifter module of the original modularization, but that adding changes of this type significantly increased the complexity of the module. In our decomposition, the sixth change would add a line omission component, perhaps called OmitShift, between the Shifter component and the ShiftIndex component. The seventh change would add an IncludeShift component in the same location. Both of these filter components override the put method of their superclass and then call the superclass's put method for some shifts, but not for others. To add both changes, both of the components would be added. No disproportional increase in complexity need occur. Similar combinations, such as adding the sixth and seventh changes together with the fourth change, only require adding their respective components.

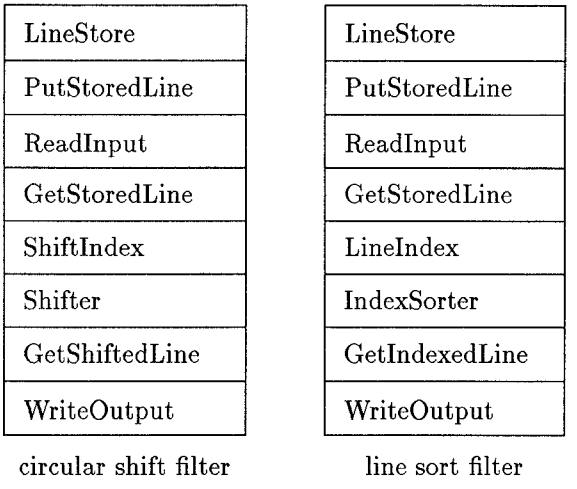

Figure 8: A composition reusing the same components as in the earlier KWIC decomposition to form two filters for a pipe and filter modularization.

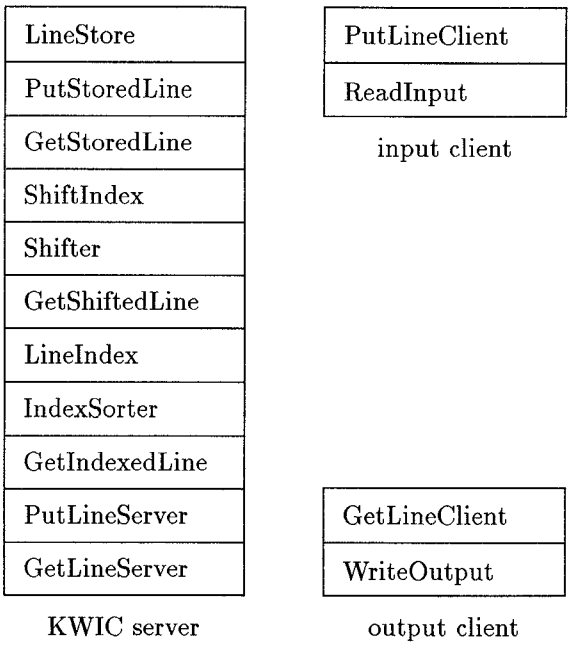

Figure 9: Recomposition of existing components to form a simple client/server modularization of KWIC. Only the client and server components are new.

There might be an efficiency concern in combining the OmitShift, IncludeShift, and StoreShiftedLine components. Each of these components fetches either the first word or the entire line during the process of storing a shifted line. One strategy would be to write text line filter components to be placed on the other side of the StoreShiftedLine component, where the actual line is already being passed with the call. An alternative choice is to replace the GetLineImport component with a GetLine component that caches the last line fetched. The component submodularization makes it possible to consider low level options of this type without excessively adding to the complexity to the module.

The eighth and ninth changes, applying the same type of filtering to the input lines, can be handled in the same way as discussed for the sixth and seventh changes, by adding equivalent filter components to the input module. In this case, text line versions of the filter components, as described above, would be used. The same components could be used to add the change to the line storage module, as suggested in the toolie article, but it is more efficient to do the filtering in the input module.

The tenth change, omitting words from lines, requires a translation component—the line will be stored, but in an altered form. This line trimming component can also be placed in the input module.

Our component based approach satisfies the objective described in the toolie article. "It should be possible to treat each enhancement as an independent unit, with the only interactions being through operations on the shared data structures" [7, p.33]. Each of the enhancements suggested in the toolie article can be satisfied by adding a single component to the implementation. Any combination of the enhancements can be applied by adding the corresponding combination of components.

3.2 Other Modularizations

Although all of the changes discussed so far had relatively straightforward implementations in the existing modularization, there may be changes that are impossible to apply with the existing modularization. For example, in a design based on pipes and filters, where sorting is implemented as a filter, incremental sorting is not an option [15]. The sorting filter cannot both output the lines as they become available and output them in sorted order. How difficult is it to change the modularization? Can existing low level components be used to form a different high level modularization?

Figure 8 shows the composition of two filters for a pipe and filter modularization. The order in which the components are listed for each filter module corresponds to the order in which they would be composed in a series of class definitions similar to those in Fig. 4. Only components from the previous decomposition are used. Had the pipe and filter modularization been implemented first, one additional compo-

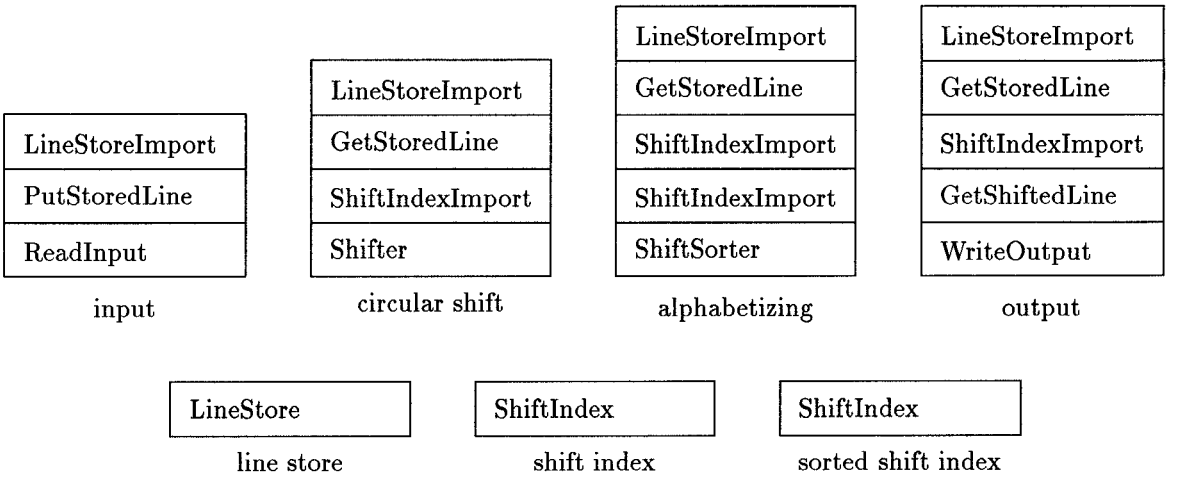

Figure 10: Components of the functional decomposition of KWIC.

nent, GetLineImport, would be needed to reproduce the previous, data oriented, modularization.

Sometimes applications are restructured to allow parts to be distributed over a network or shared among several users. In Fig. 9 the existing components are recomposed to form a simple client/server arrangement. In this composition, storing, shifting, indexing, and sorting are composed in a single module. Two additional pairs of module interconnection components provide the remote procedure call connections between the server module and client modules. All of the other components correspond to components used in the component decomposition of the original KWIC modularization.

Parnas' original KWIC article described a second, supposedly less desirable, modularization [12]. The alternative implementation was modularized around functions or processing steps. Data was stored separately and accessed directly by code in each module. The algorithm was also different in that it used two shift indexes (line number plus shift offset), where Parnas' preferred modularization used one shift index for the shifts and one line index (line number only) for the sorting. Figure 10 shows a component submodularization of this functional modularization. Many of the components that appear in the functional modularization also appeared in the data modularization. If we had implemented this version first and later decided to change to the preferred modularization, we could have done so with the addition of six new components—mostly dealing with the line index version of sorting.

The toolie article suggested yet another modularization with more flexibility for functional changes than the preferred modularization of the original KWIC article. Figure 11 shows the submodularization of a tool based modularization. As in the functional modularization above, this version of KWIC performs sorting on a shift index. The toolie modularization is similar to the earlier functional modularization, but in the toolie version of KWIC the data structures are combined with interface translations to provide interfaces that are less dependent on the details of the data structure implementations. The toolie implementation also uses implicit invocation. Two components have been added to provide the event announcements needed for implicit invocation. Our method of enhancement allows such additions to be made within modules where they have access to the events they need to announce. Again, only a few new components were needed to create this modularization when reusing components from a previous functional modularization.

3.3 Change Revisited

In the previous section we saw that it is possible to reuse the same components for different modularizations. Since design decisions are captured by the components, when a design decision is changed, it should affect the same components in whatever modularization they appear.

Consider the second change, switching between compressed and uncompressed line storage. In our earlier discussion, this change was addressed by adding a LineCompress module between the LineStore component and the Put/GetStoredLine components in the line storage module. In the functional modularization, every module imports the low level interface of the LineStore data structure, and has its

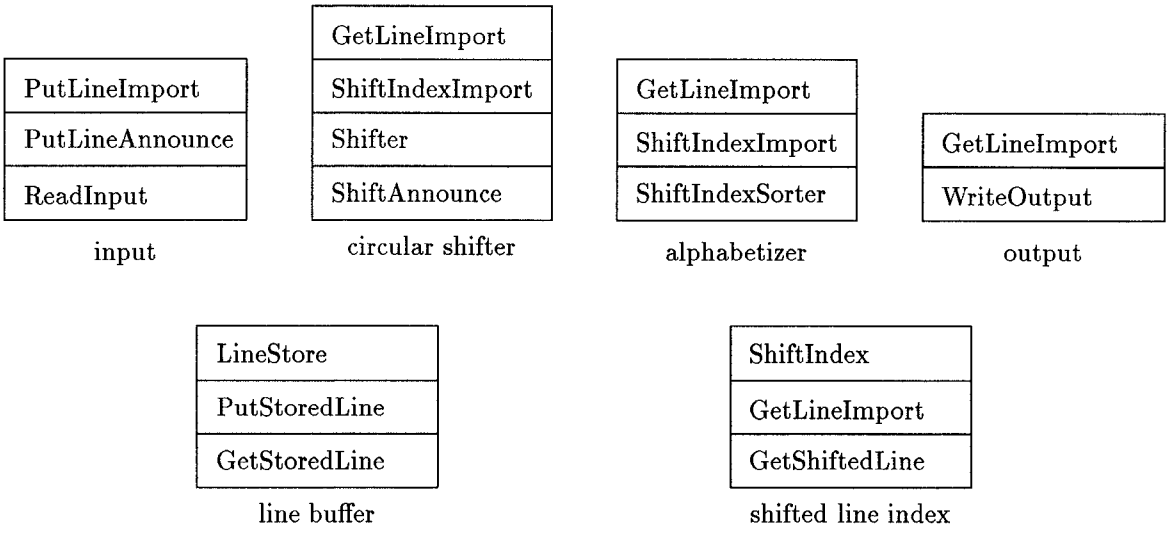

Figure 11: Composition of components to form a tool based modularization of KWIC.

own PutStoredLine or GetStoredLine component. To quote the original article, "the second change would result in changes in every module!" [12, p.1055]. The change affects the same two components; they are just repeated in several modules. In the functional modularization, the change can be applied by adding a LineCompress component between the LineStoreImport component and any Put/GetStoredLine component in every module. It is still the case that only one new component has to be written. In a similar fashion, the LineCompress component would be added twice in the pipe and filter modularization, once in the simple client/server modularization, and once in the toolie modularization.

What about other changes? The first change, changing the input format, affects both modules of the pipe and filter modularization, since they both read lines. But the same ReadLine component is used in both modules. Only one component is affected in the other modularizations.

The third change, switching between core and disk storage affects the LineStore component, and possibly the PutLineStore and GetLineStore components, in our original discussion of this change. The effect is isolated to those same three components in each of the other modularizations.

Consider the fourth change, switching from implicit shifts to storing shifted lines. Our original solution replaced the ShiftIndex and GetShiftedLine components with LineStore and GetStoredLine components, and added a StoreShiftedLine component before the Shifter component. This change is a little more difficult for the functional modularization—it has a ShiftIndexSorter component that would have to be replaced by a new LineSorter, in addition to the other changes. The same situation applies to the toolie modularization, which also has a ShiftIndexSorter component. The difference is due, not so much to the modularization as to the different handling of sorting. With the exception of the pipe and filter design, for which incremental sorting is not an option, the fifth change can be limited to the actual sorting component in each of the remaining modularizations.

The filter component solution approach to changes six through ten, described earlier, can be applied successfully in any of the modularizations. In the toolie modularization, combining omission or inclusion filters in one of the existing toolies actually violated its design principle of one function per toolie. Building a new toolie module for each enhancement is more complicated than the approach taken here of adding submodule components.

With component submodularization, applying combinations of enhancements, such as the second, third, and fourth, does not create additional complications in any of the modularizations described. Combinations of line filters from the sixth through tenth changes can also be applied by adding their corresponding components in any of the modularizations.

4 Discussion

In the previous section we showed how submodule components could be used to address the issue of change. We also demonstrated how the ability to address change with submodule components was, to a

large extent, independent of the design level modularization. In this way, submodularization allows the design level modularization to address other issues such as comprehendability, dynamic behavior, testability, and efficiency, without concern for future change. While submodularization could be applied in an existing design, the redistribution of concerns may also suggest a different design. In another paper, we showed how a design, taken from the literature, had been distorted to accommodate change [17]. Applying our technique resulted in both improved flexibility and a simpler design.

The previous section also showed how the use of submodule components made it possible to change the high level modularization without rewriting large amounts of code. Reducing the effort needed to change an application's modularization reduces the penalty for choosing the "wrong" modularization in an early design phase. Being able to change modularizations quickly could encourage a more exploratory style of application development, particularly when dealing with programs that present complicated structural demands. We have used this technique in our own work to explore difficult questions in the design of an image display application.

Decomposing modules into separate composable concerns benefits module reuse, as well. Currently, large modules are hard to reuse in part because of the number of decisions they encode [6, 9]. Applications that reuse existing modules often must settle for approximate solutions to specific requirements. Having a submodularization that allows decisions to be altered individually makes it easier to adapt large modules to new applications. Applications reusing these modules can be tailored at a finer level of granularity to suit individual expectations.

The basic principle of our approach is to submodularize modules to achieve a fine grained separation of concerns. In doing so, we treat the decomposing of modules as qualitatively different from decomposing applications. Our justification for doing so is that modules present a more narrowly focused scope of concern than applications. Other approaches typically place no restrictions on the style of implementation within a module. In our approach, we relax the standard of information hiding to allow a different level of sharing between components within a module than we allow between modules. For example, while modules are limited to calling each other's interfaces, we expose details of the uses relationships to allow components to intercept calls between other components to replace or interpose new behaviors. C++ further supports this approach with its separate public and protected levels of access.

In this paper, we presented a method of implementation, using a particular language and its features, to show that our approach is feasible. But the same approach of submodule decomposition could be used with other methods of implementation. Our method of implementation addresses the problem of change from one version of a program to the next. But it does not address the issue of dynamic change—that is to say the ability of a program to change while running. For that, other methods must be applied.

We have used our approach successfully on a number of small programs [16, 17]. To test its scalability, we are currently using the approach to implement an image display program modeled after one that was originally written in 30,000 lines of C code. The results, while not conclusive, have been promising. The new design uses 10 to 20 components per module. While the original modules were large and hard to comprehend, the new modules have an extra level of decomposition that isolates well defined concerns in smaller pieces of code. The additive nature of the components makes it easier to construct working partial systems and to incrementally test the code as development proceeds. However, because we have an existing model, and because we are using the opportunity to explore details of the technique, the effort of producing the new program cannot be compared with that of the original. Also, because our efforts involve only a single developer, we cannot say how the approach affects the use of modularization to divide labor.

5 Related Work

A number of approaches have been proposed for implementing design decisions as composable entities.

Multiple inheritance can be used to compose multiple base components. But multiple inheritance is not suitable for our purposes. The type names for intermediate levels of composition are needed to disambiguate the repeated use of a component in the same composition. Also, the linearization of the order of inheritance, provided in CLOS but not in C++, is essential for allowing components to call other components and for resolving name conflicts. Our approach of using type parameters to defer inheritance was described in a paper by Bracha [4], but so far has not been widely exploited.

The Patterns book by Gamma, Helm, Johnson, and Vlissides describes a decorator pattern for applying multiple independent properties to an object [5]. Decorators support dynamic change. But they must be written specifically for the composition in which they are used—the decorators composed by this method

must all subclass from the same superclass and have the same interface. Our components can define their own interfaces and do not have to be anticipated by other components with which they interact, making them more generally usable.

Open Implementation allows users of a module to alter certain decisions of its implementation [9]. The choices are generally encoded in the module itself, or in a corresponding meta-object, and accessed through a separate module interface. The range of choices is limited to those provided by the module's developer. Our approach allows the user to make choices about a module's composition, as well its implementation.

The Predator and P2 systems, by Batory, et al., factor data structures and other domain specific structures into independent components [2, 3]. In the Predator approach, handcrafted generators are used to compose modules from a limited set of choices. The fine grained factoring in this approach is similar to our own, but applicable only to a small set of well understood domain components. Our approach uses the compiler's own template class generating features, and is applicable to every module in an application. New components in our approach can be written by the application developer as they are needed.

Harrison and Ossher's Subject Oriented Programming allows concerns encoded in different views to be merged as a single class [8]. Their system uses a special runtime dispatcher that redirects method calls to achieve the effect of composition. To date, their work has focused more on merging two or more existing programs than creating a new program from little pieces. A similar approach to runtime dispatching is used by the composition filters of Akşet, et al. [1]. Their Sina language offers additional capabilities such as dispatching based on runtime queries, and can encode a wide range of concerns, including persistence and transactions. Method call dispatchers like those used in both these systems provide much flexibility, but they require special runtime support and add extra levels of indirection.

The Demeter system allows program fragments to be flexibly composed in the structure of an application [10, 11]. In Demeter, components encode fragments of the program structure to allow their position in the finished application to be inferred. Our components do not encode any structural dependencies. Instead, we require explicit statements to specify the positions of components. The Demeter model is based on graphs and graph traversals rather than object modules. The Demeter system also requires a special development environment and its own preprocessor.

6 Conclusion

It is commonly accepted that anticipating change is an important criteria in choosing a high level modularization. In this paper we showed that change can be anticipated at a lower level of submodules, allowing the choice of high level modularization to address other important considerations in the design. We looked at the application of a number of changes that were claimed to be more difficult for some modularizations than for others and showed that, addressed at the level of submodules, they were roughly equivalent for a variety of high level modularizations. Moreover, we showed that by using submodules the choice of high level modularization could itself be changed without having to rewrite large amounts of code.

Our approach does not replace the use of high level design. Rather, it augments it, enabling the high level design to address other important criteria with fewer compromises needed to anticipate future change. A method of implementation was presented that supports our approach through fine grained decomposition, flexible composition, and almost no runtime overhead.

References

[1] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 372–395, 1992.

[2] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 191–199, 1993.

[3] D.S. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[4] G. Bracha and W. Cooke. Mixin-based inheritance. In *Proceedings of the 1990 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 303–311, 1990.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[7] D. Garlan, G.E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE Computer*, 25(6):30–38, June 1992.

[8] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the 1993 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 411–428, 1993.

[9] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, January 1996.

[10] K.J. Lieberherr and C. Xiao. Minimizing dependency on class structures with adaptive programs. In *Object Technologies for Advanced Software: Proceedings of the First JSSST International Symposium*, pages 424–441, 1993.

[11] K.J. Lieberherr and C. Xiao. Object-oriented software evolution. *IEEE Transactions on Software Engineering*, 19(4):313–343, April 1993.

[12] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[13] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[14] R. Selby. Empirically analyzing software reuse in a production environment. In *Software Reuse: Emerging Technology*, pages 176–189. IEEE Computer Society Press, 1988.

[15] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[16] M. VanHilst and D. Notkin. Using C++ templates to implement role-based designs. In *Proceedings of the 2nd JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer-Verlag, 1996.

[17] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[18] S.H. Zweben, S.H. Edwards, B.W. Weide, and J.E. Hollingsworth. The effects of layering and encapsulation on software development cost and quality. *IEEE Transactions on Software Engineering*, 21(3):200–208, March 1995.