Life Cycle and Refactoring Patterns that Support Evolution and Reuse

Brian Foote
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
foote@cs.uiuc.edu

William F. Opdyke
AT&T Bell Laboratories
Naperville, Illinois 60566
opdyke@iexist.att.com

Software development can be characterized in terms of prototype (or initial design) phases, expansion phases and consolidation phases. During a consolidation phase, some relationships, initially modeled using inheritance, may be evolved to aggregations. Also, during consolidation, abstrasct classes are sometimes defined to capture behavior common to two or more existing classes. In this paper, we define high-level patterns for the prototype, expansion and consolidating programs. We also define supporting patterns for evolving aggregations from inheritance hierarchies and for creating abstract classes.

1 INTRODUCTION

Each pattern in the larger language, can, because it is connected to the larger language, help all other patterns to emerge.[1]

Patterns can exist at all scales [1].

The patterns are not just patterns of relationships, but patterns of relationships among other smaller patterns, which themselves have still other patterns hooking them together – and we see finally, that the world is entirely made of all these interhooking, interlocking non material patterns.[1]

You see then that patterns are very much alive and evolving. No matter what the asterisks say, the patterns are still hypotheses, all 253 of them – and are therefore all still tentative, all free to evolve under the impact of new experiences and observations [2].

But what guarantee is there that this flux, with all its individual acts, will not create chaos?
It hinges on the close relationship between the process of creation and the process of repair [1].

And, more subtly, we also find that different patterns in different languages, have underlying similarities, which suggest that they can be reformulated to make them more general, and usable in a greater variety of cases [1].

So the real work of any process of design lies in the task of making up the language, from which you can later generate one particular design [1].

The language will evolve, because it can evolve piecemeal, one pattern at a time.[1]

Most of the work to-date on patterns has concentrated on characterizing the recurring functional, structural, and behavioral relationships among objects. Less attention has been paid to how classes and frameworks emerge and evolve. However, truly reusable objects are the result of an iterative, evolutionary process. We believe that it is possible to characterize aspects of this process itself using patterns. We agree with Kent Beck [10] that an emphasis on the transformations that designers can make to existing objects to improve them can be as helpful to designers as depictions of the resulting artifacts.

During a discussion on the patterns mailing list, Booch and Cunningham [11] claimed that many of the objects in a system may be found via a simple examination of the grammatical relationships in the system's specification. Many of the remaining objects, they claim, are uncovered during using analysis tools such as CRC cards. Only a few are found late in the life cycle; however (they concede) these are often of exceptional value, since they embody insights that emerge only from experience, and can "make complexity melt away" [11].

We feel that it is important to add that while the basic identities of many objects may be discovered early, these objects will change and improve as the system evolves. Truly reusable objects emerge as the result of this evolutionary process. As Dennis DeBruler has noted [9], it is important to allow for down stream changes, to avoid design paralysis during the early phases.

We think it may be possible to characterize this process using a four-layer set of patterns. These patterns would be far from a full-fledged pattern language for object-oriented software development. They should instead be thought of as a rough, preliminary sketch of where some of the major landmarks in such a language might be located. A full exposition of these potential patterns is beyond the scope of this paper. We have elected instead to focus upon five of them in detail. Nonetheless, we hope that through our discussion of the contexts these patterns complete, and the patterns they give rise to, the reader may begin to discern the outlines of this nascent pattern language.

A top-layer pattern *Develop software that is usable today and reusable tomorrow* has forces that are resolved by the second-layer patterns *Prototype a first pass design*, *Expand the initial prototype* and *Consolidate the program to support evolution and reuse.*

In this paper, we define each of these second layer patterns, in sections three through five. Then, we define (in sections six and seven) two patterns that apply during the consolidation phase. The consolidation aspects of program evolution have been a focus of our research on object evolution [13], life cycles [14], reuse [16] and refactoring [17, 21, 22].[1] Design guidelines for the consolidation phase have also been documented by others in, for example, [3, 7, 18, 23].

Evolve Aggregations From Inheritance Hierarchies, also examined in this paper, is one of the third-layer patterns that resolves the forces associated with the consolidation process. Inheritance models the *is-a* relation, while aggregation models the *has-a* relation. However, these relations are less distinct than might be thought at first. Is a pixel a point, or does a pixel have a location, which is a point [24]? Is a matrix an array with extra behavior, or does a matrix have a representation, which is an array [12, 21]? Different people give different answers to these questions, and it is common for a person's answer to change over time. On the one hand, both points of view can lead to working programs. On the other hand, they differ in how the resulting designs will be reused and the kinds of changes than can easily be made to them. It is important to be able to change software so it reflects the current point of view. Although it is possible to convert aggregation to inheritance, converting inheritance to aggregation (the focus of this paper) seems to be more common, for several reasons [17].

Create Abstract Superclass is another third-later pattern defined in this paper. During consolidation abstractions common to two of more classes can be moved to a common abstract superclass. This pattern describes that can be done, and what forces must be resolved.

Finally, there is the fourth layer of refactoring (ie behavior preserving program transformation) patterns [21] that resolve the forces of this (and similar) patterns.

We have found this layered approach helpful in characterizing the program consolidation phase, in

[1] While the refactoring examples described herein apply most clearly to C++ programs, we have also researched how these patterns apply to programs written in Smalltalk and CLOS.

understanding how refactorings can be interleaved with additions, and in ensuring that refactorings can be safely applied to object-oriented programs [21].

2 BACKGROUND: OBJECT EVOLUTION

There are three distinct phases in the evolution of object-oriented abstract classes, frameworks and components: a *prototype phase*, an *expansionary phase* and a *consolidation phase*. Associated with each of these phases is a series of high-level patterns that address the forces that must be resolved during the phase. These high-level patterns, in turn, are realized by applying lower-level patterns that resolve these forces. In the process of software development, we have seen these phases iterated and replicated in and among classes, frameworks and applications. This pattern of self-similarity at different levels is typical of fractal curves; hence we refer to our characterization as the *Fractal Model* [14].

The *Fractal Model* can be thought of as an object-oriented specialization of Boehm's *Spiral Model* [4]. The Spiral Model is cast broadly, in such a way so as to accomodate reuse, iteration, and the independent evolution of subsystems. The Fractal Model emphasizes those characteristics of objects that allow them to evolve in ways that traditional software cannot. It is also unique in its emphasis on consolidation and refactoring as essential stages in the evolution of truly reusable components.

In the sections that follow, we will describe our patterns in a format similar to that of Alexander [2]. The subsections below present the context, problem, solution and discussion of related patterns.

3 PATTERN: PROTOTYPE A FIRST-PASS DESIGN

3.1 Context

In order to *Develop software that is usable today and reusable tomorrow*, one must first address the problem at hand. Initial (albeit sketchy) user requirments should be available. There is pressure to produce tangible results relatively quickly.

3.2 Problem

Building systems from the ground up is expensive and time consuming. Moreover, it is difficult to tell if they really solve the problems they were intended to solve until they are complete.

It is rare to see systems built completely from scratch these days. Modern software systems rely on a variety of domain independent components and tools. However, reusable domain-specific objects and frameworks are still relatively rare, particularly outside of the realm of graphical user interfaces.

It should come as no surprise that that is so. Simply designing a system at all is hard. Designing a general, reusable system from first principles is much harder. Designing a system that addresses both the requirements at hand, as well as a broader range of potential future problems pose nearly insurmountable challenges.

3.3 Solution

The initial design of a system should focus on the requirements at hand, with broader applicabilty as a secondary concern. It is important instead to get something running relatively quickly, so that feedback regarding the design can be gotten. This initial prototype can borrow expediently from existing code.

As Brooks notes [6], software should be *grown* not *built*. Successful large systems begin as successful small systems. A good way to get started is to build a prototype.

For object-oriented programs, early prototypes allow designers to get feedback from customers, and enable designers to understand the architectural issues they need to confront. Often, the prototype is a quick, first-pass design, where the emphasis is on finding a set of objects that embody the surface structure of the problem at hand.

The *prototype phase* may involve the application of analysis and design methods (such as [5], [8] and [26]) as well as the development of initial prototype implementation.

During the construction of a prototype, it is common to expediently make use of existing code in order to get something working quickly. Such a strategy depends on not only on the availablility of pre-existing domain independent reusable components like collections, but on an infrastructure of domain-specific artifacts as well. Even in those domains where such code does not exist, code from a related domain might be “borrowed”.

Leveraging existing code to create a new application based on an existing one is sometimes called ”programming-by-difference”. It is fair to ask where such reusable code (which serves as the foundation for an initial design) comes from for domains where none previously exists. The next two patterns address this issue.

3.4 Related Patterns

While this phase can realize a reasonable first-pass set of objects, the designs of these objects still need to be refined and later may need to be redesigned. Examples of patterns that apply in this phase are: *Nouns in the specification imply objects, verbs operations (P1), Build on existing objects using inheritance (P2), Get it running now, polish it later (P3),* and *Avoid premature generality (P4).* (Note that these patterns are not further developed here.) This phase also sets the stage for exploration and consolidation. These are discussed in the following sections.

4 PATTERN: EXPAND THE INITIAL PROTOTYPE

4.1 Context

Successful systems are seldom static. Instead, success sets the stage for evolution.

4.2 Problem

When software addresses an important need, both users and designers may recognize opportunities to apply the software in new ways. Often, addressing these new applications would require some changes to the program – changes that were not envisioned when the software was initially designed. Such software evolution and reuse can undermine a program's structure, and over time, make it more difficult to understand and maintain the software.

During the *expansion phase*, designers often try to reuse parts of a program for purposes that differ from the program's original purpose to varying degrees. In traditional languages, such reuse might be undertaken by making copies of the original code, or by introducing flags and conditionals into the original code. Such activity tends to compromise a program's structure, and make it difficult to understand and change the program later.

4.3 Solution

In object-oriented programs, inheritance is a powerful and useful mechanism for sharing functionality among objects. Placing new code in subclasses can help maintain design integrity, because changes are isolated in these subclass, and the original code in the superclasses remains intact.

Objects can evolve more gracefully than can traditional functions or procedures because exploratory changes can be confined to subclasses. Such changes are less potentially disruptive to existing code that depends on a component.

What often results from the expansion phase is a class hierarchy that models a history of changes. The resulting subclasses are not yet truly general. More desirable, from a software maintenance standpoint, would be an inheritance hierarchy that models a type hierarchy [19].

4.4 Related Patterns

During expansion, patterns such as these come into play: *Subclass existing code instead of modifying it (E1), Build on existing objects using inheritance (E2; like P2), Defer encapsulation for shared resources (E3), Avoid premature generality (E4; like P4)* and *Get it running now, polish it later (E5; like P3).* Note that some of the same patterns that appeared during the prototype phase appear here as well. This reflects genuine underlying similarities between these two phases.

5 PATTERN: CONSOLIDATE THE PROGRAM TO SUPPORT EVOLUTION AND REUSE

5.1 Context

Initial designs are often loosly structured. As objects evolve, insights as to how they might have been designed better emerge.

5.2 Problem

As objects evolve, they are subjected to forces that can undermine their structure if they are left unchecked. Prototypes are often first-pass designs that are expediently structured. During expansion, the introduction of new, sometimes conflicting requirements can muddle the clarity of parts of the original design. The insight necessary to improve objects is often not available until later in the life cycle. Traditional life cycle notions do not address the need to exploit this insight.

Truely reusable objects seldom emerge fully formed from an initial analysis of a given problem domain. More commonly, they are discovered later in the life cycle, or are polished and generalized as a system evolves. As a result, the objects in the system must be changed to embody this structural insight.

Traditional waterfall life cycle models do not accommodate redesign late in the life cycle. Later life cycle models, such as the Spiral Model, embrace iteration, but do not address the unique properties of evolving objects.

Objects evolve differently than traditional programs. This is because they can, and do, change within and beyond the applications that spawn them. Some of these changes add breadth or functionality to the system, others improve its structure or future reusability. It is easy to understand why the latter are often deferred indefinitely. This is unfortunate, because it is these changes that can be of the most enduring value.

Prototypes are loosely structured for a variety of reasons. One is that prototypes often are built to allow the designer to gain an initial sense of the layout of the design space. By definition, the designers understanding of the problem will be immature at this time. Objects found during this phase may reflect the surface structure of the problem adequately, but will need to be refined to do so elegantly. Furthermore, they will need to be reused in order to become reusable.

A second reason for the structural informality of prototypes is that they often are constructed in an expedient fashion out of existing reusable parts. This should not be seen as a bad thing. *"Get it running now, polish it later (P3)."* can be an effective strategy for learning how to employ existing components to address new requirements.

In both cases, the insight necessary to get the objects right is not available up-front. If the process does not accommodate it when it does become available, these rough drafts can become the final ones.

During expansion, objects that have proven useful are redeployed in contexts that differ from their original ones. Since the requirements raised in these contexts were not part of the specification for the original objects, they could not, in general, have been anticipated when these objects were designed. In object-oriented systems, these tend to accumulate around the leaves of the inheritance graph. Over time, the hierarchy can become overgrown with redundant, haphazardly organized code.

5.3 Solution

Exploit opportunities to consolidate the system (by refactoring objects) to embody insights that become evident as the system evolves.

Objects can provide opportunities for reuse that are not available to conventional software. Object-oriented encapsulation encourages more modular initial designs. Inheritance allows changes made to accommodate new requirements to be made in subclasses, where they do not undermine the structural integrity of existing objects.

There comes a time when insight gained during the prototype and consolidation phases can be employed to refactor the system. Refactorings typically do not change the the way the system works, but rather improve its structure and organization.

Experience accrued during successive reapplications of an object (during the prototype and expansion phases) should be applied during a *consolidation phase* to increase its generality and structural integrity. A program's design should be improved; abstract classes and frameworks should emerge or be made more explicit. During the expansionary phase, the size of a system typically increases. During consolidation, it can shrink.

For example, a designer might notice that two methods added during expansion contain duplicated code or data. The designer might factor this common code into a common superclass. Or, a method may have grown larger as the code evolved. A designer might break this code into several methods to increase its level of abstraction, and to provide new places to override behavior. As an object evolves, it is common for new objects to emerge. The next section describes a refactoring that addresses this. Each refactoring can be seen as addressing and correcting forces that, if left unchecked, would undermine the structural integrity of the objects that comprise the system. As a system evolves, disorder and entropy can increase. Consolidation can be seen as an entropy reduction phase.

5.4 Related Patterns

Table 1 lists 13 design rules that are characteristically employed during consolidation. Table 2 lists refactorings that can be employed during consolidation. The next two section presents two of the most common and important refactorings as patterns.

Table 1: Design Rules [16]
DR1. use consistent names
DR2. eliminate case analysis
DR3. reduce the number of arguments
DR4. reduce the size of methods
DR5. class hierarchies should be deep and narrow
DR6. the top of the class hierarchy should be abstract
DR7. minimize access to variables
DR8. subclasses should be specializations
DR9. split large classes
DR10. factor implementation differences into subcomponents
DR11. separate methods that do not communicate
DR12. send messages to components instead of to self
DR13. reduce implicit parameter passing.

Table 2: Refactoring Patterns [21]

Category	*Refactoring(s)*
High Level Refactoring	HR1. create abstract superclass HR2. subclass and simplify conditionals HR3. capture aggregations and components
Supporting Refactorings: Create program entity	 SR1. create empty class SR2. create member variable SR3. create member function
Delete program entity	SR4. delete unreferenced class SR5. delete unreferenced variable SR6. delete a set of member functions
Change program entity	SR7. change class name SR8. change variable name SR9. change member function name SR10. change type of a set of variables and functions
	SR11. change access control mode SR12. add function argument SR13. delete function argument SR14. reorder function arguments SR15. add function body SR16. delete function body SR17. convert instance variable to pointer SR18. convert variable references to function calls SR19. replace statement list with function call SR20. in-line function call SR21. change superclass
Move member variable	SR22. move member variable to superclass SR23. move member variable to subclass
Composite refactorings	SR24. abstract access to member variable SR25. convert code segment to function SR26. move a class

6 PATTERN: EVOLVE AGGREGATIONS FROM INHERITANCE HIERARCHIES

6.1 Context

The class hierarchies that emerge during the prototype and expansion phases are often functional, but neither elegant nor reusable. During the consolidation phase, designers take time to exploit opportunities to clean up the system, improve its structure and comprehensibility, and increase its reuse potential. Evolving aggregations from inheritance hierarchies can play a major role in system consolidation. This pattern can be employed to *Factor implementation differences into subcomponents (DR10), Separate methods that do not communicate (DR11)* and *Send messages to components instead of to self (DR12).*

6.2 Problem

Inheritance sometimes is overused during the early phases of an object's evolution. Changing informal, white-box-based inheritance to black-box style aggregate-component relationships can result in better encapsulated, better structured, more resuable, more understandable code.

During the prototype and expansionary phases of an objects evolution, designers tend to depend heavily on inheritance. Inheritance is often used where aggregation would be better because:

- inheritance is supported at the language level, so using it is easier than constructing aggregates by hand. Since it is a feature of object-oriented languages, programmers are trained to use it when they learn the language. They do not become familiar with design idioms and patterns such as aggregation until they become more experienced.
- it is not obvious that an is-a relationship should become a has-a relationship until the subclass becomes more mature.
- inheritance creates a white-box relationship that makes sharing resources such as operations and variables easy. It does not become clear how best to untangle intra-object coupling that may exist until the object has been used and reused for a while, and the fissures along which new object may be cleaved become more evident.

There comes a time (i.e. the consolidation phase) when designers may notice that parts of an object exhibit a degree of cohesion that suggests that a distinct objects can be factored from the existing hierarchy. The following benefits might be realized if some inheritance relationships were able to be changed into aggregations:

1. cohesion and encapsulation could be improved by changing one large class to two smaller classes
2. aggregates could change their components at runtime, while inherited subparts are static. That is, components can exploit dynamic polymorphism. A component might become a member of a different aggregate as well.
3. separate classes could be reused independently, and may independently evolve. Each may spawn subclasses that can be interchangeable used by the other, since they will communicate only via a public interface.
4. an aggregate might have more than one instance of a given component class.

An example of an inheritance-based relationship that could be cast as an aggregate might be a **Matrix** class. The initial design of such a class might be based on the observation that a **Matrix** is a **TwoDimensionalArray** to which a repertoire of arithmetic operations are added. Hence, **Matrix** might be defined as a subclass of **TwoDimensionalArray** that adds operations like +,

*, and transpose to the inherited methods for accessing and changing array elements. Changing the relationship from an inheritance based relationship to aggregation can take advantage of the fact that the **TwoDimensionalArray** subpart is being used essentially intact as a state repository for the **Matrix** abstraction. Making this part of the **Matrix** a component can permit alternate representations for this repository, such as **SparseArrays** or even stateless identity objects, to be used in place of **TwoDimensionalArrays**.

6.3 Solution

Change inheritance-based relationships into aggregate-component relationships by factoring parts of an existing class into a new, component class. Perform these changes in such a way as to ensure that the program will still work as it did before.

Suppose that **A** is a subclass of **C**. **A** can reuse behavior of **C** by:

1. adding an instance of **C** as a component variable of **A**.
2. replacing references to variables and functions inherited from **C** with references to the component
3. removing the inheritance link between **A** and **C**.

For example, the **Matrix** class is a subclass of **TwoDimensionalArray**, with an inherited variable *arrayRepr* and inherited functions *get* and *put*. An instance of class **TwoDimensionalArray** is added as a component variable of **Matrix**. References to the inherited members of class **TwoDimensionalArray** are replaced by references to members of its new component variable. Then, the superclass of **Matrix** is changed (eg, to another class, or to null if Matrix is now a top-level class).

Ensuring that the program will still work after the changes are performed is easy for steps 1 and 3, but more difficult for step 2. where references to inherited variables and functions must be replaced not only in **A** (or **Matrix**) but also in its clients. One way to make step 2 easier is to abstract access to the variables inherited by **A** (or **Matrix**), and change the accessing functions to point to the members of the component variable.

6.4 Related Patterns

Changing inheritance-based relationships to aggregate/component relationships can require that a number of supporting refactorings be applied to a program. Creating an instance of the component class and populating it employs the pattern *create member variable (SR2)*. Changing the superclass of the aggregate class employs the pattern *move class (SR25)*. Other related patterns include *create member variable (SR2)*, *create member function (SR3)*, *delete unreferenced variable (SR5)*, *delete a set of member functions (SR6)*, *add function body (SR15)*, *move member variable to superclass (SR22)*, *move member variable to subclass (SR23)* and *move class (SR25)*. Changes to argument lists and member names may also be necessary, employing the patterns *change variable name (SR8)*, *change function name (SR9)*, *add function argument (SR12)*, *delete function argument (SR13)* and/or *reorder function arguments (SR14)*. Abstracting access to variables employs the pattern *abstract access to member variable (SR23)*.

7 PATTERN: CREATE ABSTRACT SUPERCLASS

7.1 Context

As noted for the prior pattern, the class hierarchies that emerge during the prototype and expansion phases are often functional, but neither elegant nor reusable. One way to clean up inheritance hierarchies during the consolidation phase is to define abstract classes that capture behavior common to one or more existing classes. This pattern can be employed to satisfy the following design rules: *Class hierarchies should be deep and narrow (DR5)*, *The top of the class hierarchy should be abstract (DR6)* and *Subclasses should be specializations (DR8)*.

7.2 Problem

As programs evolve, abstractions emerge. Abstractions are appear in two or more classes are often implemented differently, and are often intertwined with code that is specific to a class. Unless abstractions are consolidated in one place, code duplication persists and it hard to reuse the abstaction.

Systems grow with age. As they grow, the same abstraction may appear in more than one place in a program. This may happen because:

- one common programming practice is to entend a program by copying existing code and modifying it. As this happens, code gets duplicated.
- on multi-person projects, different project members may implement the same functionality independently in the parts of a program for which they are responsible.

During the consolidation phase, these common abstractions are sometimes discovered. If the abstractions were consolidated in one place, several benefits might be realized:

- Defining the abstraction *in one place* reduces the program's size and possibly its execution time
- Separating out the abstraction makes it easier to understand and reuse.
- If the abstraction (or its implementation) is flawed, it need only be fixed *in one place.* One problem with the copy-and-modify approach to software development is that errors in the original code get copied along with the code. If the error is subsequently discovered and fixed in one place, it may still persist somewhere else.
- If throughout a program abstractions are separated out and made explicit, it can make the entire program easier to understand and evolve.

An example of where this pattern might be applied is where two classes **DenseMatrix** and **SparseMatrix** are defined. Suppose that **DenseMatrix** was define first, then later **SparseMatrix** was defined by copying **DenseMatrix** and modifying it. These two classes contain common behavior and duplicated code. An abstract superclass **Matrix** could be defined that captures the behavior common to these two classes [22].

7.3 Solution

Factor abstractions common to two or more classes into a common abstract superclass. Perform these changes in such a way as to ensure that the program will still work as it did before.

Suppose that classes **C1** and **C2** share a common abstraction. An abstract superclass can be defined by:

1. adding a new class **A1**, which initially contains no locally defined members;
2. making **A1** the new superclass of both **C1** and **C2**;
3. determined the common behavior (functions, or parts of functions) in **C1** and **C2**;
4. changing (as needed) function names, argument lists, function bodies and the attributes of reference variables so that functions that implement common behavior (in **C1** and **C2**) are implemented identically.
5. moving the common functions to **A1** and deleting them from the subclasses.

For example, during the evolution of the *Choices* file system framework [20] two classes **BSDInode** and **SystemVInode** were defined to support two different file formats. This pattern was applied to move common behavior into a common superclass *Inode*. While some of the steps in applying this pattern were trivial, changing the function bodies was not. The *mapUnit* function was defined in both classes, included much common code but also a few differences. The differing code was split off into new functions, and (in mapUnit) the differing code segments were replaced by calls to these functions, in order to make the function definitions in the two classes conform [22].

7.4 Related Patterns

Creating the abstract superclass may employ the patterns *create empty class (SR1)*, *create member variable (SR2)*, *create member function (SR3)*, *delete unreferenced variable (SR5)*, *delete a set of member functions (SR6)*, *change variable name (SR8)*, *change member function name (SR9)*, *change type of a set of variables and functions (SR10)*, *change access control mode (SR11)*, *add function argument (SR12)*, *delete function argument (SR13)*, *reorder function arguments (SR14)*, *replace statement list with function call (SR19)*, and *move member variable to superclass.*

8 DISCUSSION

To reiterate, our emphasis on consolidation does not mean that one should abandon the use "up front" of disciplined design and analysis techniques. On the contrary, one should apply discipline in the up front stages, while realizing that the design won't remain fixed throughout a program's evolution. Over time, insights are gained and programs are evolved to address new problems that were not understood when the programs were initially designed. The focus on consolidations is not so much to "fix mistakes" as it is to improve a program's structure to accommodate change.

In our aggregation pattern we discuss how inheritance is overused and sometimes is incorrectly used. Our pattern addresses how to fix one common misuse of inheritance - but in proposing this pattern, are we (improperly) encouraging an undisciplined use of inheritance, with the idea that one can "fix things later"? We think not. As noted earlier, "is-a" relationships are not always clearly distinct from "has-a" relationships. Points of view change over time, which does not imply that the original use on inheritance was incorrect.

C++ implements subtyping using subclassing. However, inheritance in C++ can also be (and sometimes is) used to implement programming-by-difference - a variant on the copy and modify approach to program development. We agree with Liskov [19] and others that inheritance should primarily be used to represent subtyping relationships - however, in practice inheritance is not always used this way. Our patterns allow one to more clearly reflect typing relationships in programs.

In summary, in this paper we have characterized the evolution of object-oriented programs in terms of three distinct phases (prototype, expansion and consolidation). We defined a high-level pattern for program consolidation, and also defined a consolidation pattern for evolving aggregations from inheritance hierarchies.

It has been widely recognized that aggregates are superior to inheritance for expressing some structural relationships [25]. Black box components can better support encapsulation than the white-box nature of inheritance. Also, the ability to replace old components with new ones helps in realizing the benefits of polymorphism at run time.

Gamma et. al. [15] have compiled a catalog of two dozen structural design patterns. The emergence of aggregate/components relationships, together with forwarding methods is a prominent, recurring theme in a sizable number of their patterns. Given the ubiquitous nature of this relationship, we hope to see better linguistic support for aggregation.

9 ACKNOWLEDGEMENTS

Ralph Johnson supervised both of our research projects, and provided review comments on several drafts. John Brant, Gabrielli Elia, Brian Marick, Don Roberts and other members of Ralph Johnson's patterns seminar provided insightful review comments on a later draft, as did Ken Auer and the PLOP '94 reviewers. AT&T Bell Laboratories supported William F. Opdyke's refactoring research at the University of Illinois under the full-time doctoral support program.

References

[1] C. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[2] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language.* Oxford University Press, 1977.

[3] Paul L. Bergstein. Object-preserving class transformations. In *Proceedings of OOPSLA '91*, 1991.

[4] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5), May 1988.

[5] Grady Booch. *Object-Oriented Design.* Benjamin/Cummings, 1990.

[6] Frederick P. Brooks. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1987.

[7] Eduardo Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach.* PhD thesis, University of Geneva, 1991.

[8] Peter Coad and Ed Yourdon. *OOA - Object-Oriented Analysis.* Prentice-Hall, 1990.

[9] Dennis Debruler. Review comments on this paper. PLOP '94.

[10] Kent Beck et al. Patterns postings related to aggregations. email exchange on patterns@cs.uiuc.edu.

[11] Ward Cunningham et al. *When Are Objects Found?* email exchange on patterns@cs.uiuc.edu.

[12] Brian Foote. *An Object-Oriented Framework for Reflective Meta-Level Architectures.* Ph.D. thesis in preparation, University of Illinois at Urbana-Champaign.

[13] Brian Foote. Designing to facilitate change with object-oriented frameworks. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[14] Brian Foote. A fractal model of the lifecycle of reusable objects. In *OOPSLA '93 Workshop on Process Standards and Iteration (J. Coplien, R. Winder and S. Hutz, organizers)*, Washington, D.C., 1993.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[16] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[17] Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In *Proceedings of ISOTAS '93: International Symposium on Object Technologies for Advanced Software*, November 1993.

[18] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[19] Barbara Liskov. Data abstraction and hierarchy. In *Addendum to the Proceedings of OOPSLA '87*, 1987.

[20] Peter W. Madany. *An Object-Oriented Framework for Filesystems.* PhD thesis, University of Illinois at Urbana-Champaign, 1992. Also Technical Report No. UIUCDCS–R–92–1751, Department of Computer Science, University of Illinois at Urbana-Champaign.

[21] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992. Also Technical Report No. UIUCDCS–R–92–1759, Department of Computer Science, University of Illinois at Urbana-Champaign.

[22] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of CSC '93: The ACM 1993 Computer Science Conference*, February 1993.

[23] Roxanna Rochat. In search of good Smalltalk programming style. Technical Report CR-86-19, Tektronix, 1986.

[24] J. P. Rosen. What orientation should ada objects take? *Communications of the ACM*, 35(11):71–76, November 1992.

[25] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA '86*, pages 38–45, November 1986. printed as SIGPLAN Notices, 21(11).

[26] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.