Chapter 6
6.0 Reverse Engineering

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction [41]. In reverse engineering, the requirements and the essential design, structure and content of the legacy system must be recaptured.

In addition to capturing technical relationships and interactions, information and rules about the business application and process that have proved useful in running the business must also be retrieved. This involves extracting design artifacts and building or synthesizing abstractions that are less implementation dependent. The key objectives in reverse engineering are to generate alternative views, recover lost information, detect side effects, synthesize higher abstractions, and facilitate reuse.

The effectiveness of this process will affect the success of the reengineering project. Reverse engineering does not involve changes to the system or creating a new system, it is the process of examination without changing its overall functionality.

The reverse engineering process, shown in Figure 6.1, begins by extracting the requirements and detailed design information from the source code and existing documents. A requirements document is created and a high level design abstraction is extracted and expressed using dataflow and control-flow diagrams [37]. The recovered design is reviewed for consistency and correctness.
6.1 Patterns

6.1.1 Introduction

Each pattern in the larger language, can, because it is connected to the larger language, help all other patterns to emerge. Patterns can exist at all scales. 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.

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

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

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. It is possible to characterize aspects of this process itself using patterns. Kent Beck [37] highlighted 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.

Booch [22] 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". 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. Belady L [33] also noted that it is important to allow for down stream changes, to avoid design paralysis during the early phases.
It may be possible, however, to characterize this process using a four-layer set of patterns. These patterns would be far from a fullfledged 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 book. 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 that develops 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 section, we define each of these second layer patterns. Then, we define two patterns that apply during the consolidation phase. The consolidation aspects of program evolution have been a focus of the research on object evolution, life cycles, reuse and refactoring. Design guidelines for the consolidation phase have also been documented by others in, for example. Evolve Aggregations from Inheritance Hierarchies, also examined in this section, 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? Is a matrix an array with extra behavior, or does a matrix have a representation, which is an array? 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 seems to be more common, for several reasons.

Create Abstract Superclass is another third-layer pattern defined in this section. 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 (i.e. behavior preserving program transformation) patterns that resolve the forces of this (and similar) patterns. We have found this layered approach helpful in characterizing the program consolidation phase, in understanding how refactorings can be interleaved with additions, and in ensuring that refactorings can be safely applied to object-oriented programs.

6.2 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 introduce the Fractal Model.
The Fractal Model can be thought of as an object-oriented specialization of Boehm's Spiral Model [34]. The Spiral Model is cast broadly, in such a way so as to accommodate 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.

6.2.1 PATTERN: PROTOTYPE A FIRST-PASS DESIGN

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 requirements should be available. There is pressure to produce tangible results relatively quickly.

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.

**Solution**

The initial design of a system should focus on the requirements at hand, with broader applicability 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, 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 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 availability 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 will address this issue.
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.

6.2.2 PATTERN: EXPAND THE INITIAL PROTOTYPE
Context
Successful systems are seldom static. Instead, success sets the stage for evolution.
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.
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.

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), Defe enclosure 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.

6.2.3 PATTERN: CONSOLIDATE THE PROGRAM TO SUPPORT EVOLUTION AND REUSE

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

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 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.

Related Patterns

Table 6.1 lists 13 design rules that are characteristically employed during consolidation. Table 6.2 lists refactorings that can be employed during consolidation.

<table>
<thead>
<tr>
<th>Design Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR1</td>
</tr>
<tr>
<td>DR2</td>
</tr>
<tr>
<td>DR3</td>
</tr>
<tr>
<td>DR4</td>
</tr>
<tr>
<td>DR5</td>
</tr>
<tr>
<td>DR6</td>
</tr>
<tr>
<td>DR7</td>
</tr>
<tr>
<td>DR8</td>
</tr>
<tr>
<td>DR9</td>
</tr>
<tr>
<td>DR10</td>
</tr>
<tr>
<td>DR11</td>
</tr>
<tr>
<td>DR12</td>
</tr>
<tr>
<td>DR13</td>
</tr>
</tbody>
</table>

Table 6.1. Design Rules
<table>
<thead>
<tr>
<th>Category</th>
<th>Refactoring(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Level Refactoring</td>
<td>HR1: Create abstract superclass</td>
</tr>
<tr>
<td></td>
<td>HR2: Subclass and simplify conditionals</td>
</tr>
<tr>
<td></td>
<td>HR3: Capture aggregations and components</td>
</tr>
<tr>
<td>Supporting Refactorings;</td>
<td>SR1: Create empty class</td>
</tr>
<tr>
<td>Create program entity</td>
<td>SR2: Create member variable</td>
</tr>
<tr>
<td></td>
<td>SR3: Create member function</td>
</tr>
<tr>
<td>Delete program entity</td>
<td>SR4: Delete unreferenced class</td>
</tr>
<tr>
<td></td>
<td>SR5: Delete unreferenced variable</td>
</tr>
<tr>
<td></td>
<td>SR6: Delete a set of member functions</td>
</tr>
<tr>
<td>Change program entity</td>
<td>SR7: Change class name</td>
</tr>
<tr>
<td></td>
<td>SR8: Change variable name</td>
</tr>
<tr>
<td></td>
<td>SR9: Change member function name</td>
</tr>
<tr>
<td></td>
<td>SR10: Change type of a set of variables and functions</td>
</tr>
<tr>
<td></td>
<td>SR11: Change access control mode</td>
</tr>
<tr>
<td></td>
<td>SR12: Add function argument</td>
</tr>
<tr>
<td></td>
<td>SR13: Delete function argument</td>
</tr>
<tr>
<td></td>
<td>SR14: Reorder Function arguments</td>
</tr>
<tr>
<td></td>
<td>SR15: Add function body</td>
</tr>
<tr>
<td></td>
<td>SR16: Delete function body</td>
</tr>
<tr>
<td></td>
<td>SR17: Convert instance variables to pointers</td>
</tr>
<tr>
<td></td>
<td>SR18: Convert variable references to function calls</td>
</tr>
<tr>
<td></td>
<td>SR19: Replace statement list with function call</td>
</tr>
<tr>
<td></td>
<td>SR20: in-line function call</td>
</tr>
<tr>
<td></td>
<td>SR21: Change superclass</td>
</tr>
<tr>
<td>Move member variable</td>
<td>SR22: Move member variables to superclass</td>
</tr>
<tr>
<td></td>
<td>SR23: Move member variables to subclass</td>
</tr>
<tr>
<td>Composite refactorings</td>
<td>SR24: Abstract access to member variable</td>
</tr>
<tr>
<td></td>
<td>SR25: Convert code segment to function</td>
</tr>
<tr>
<td></td>
<td>SR26: Move a class</td>
</tr>
</tbody>
</table>

| Table 6.2. Refactoring Patterns |

6.2.4 PATTERN: EVOLVE AGGREGATIONS FROM INHERITANCE HIERARCHIES

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).

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 reusable, more understandable code. During the prototype and expansionary phases of an object's evolution, designers tend to depend heavily on inheritance. Inheritance is often used where aggregation would be better because:

1. 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;
2. It is not obvious that an \textit{is-a} relationship should become a \textit{has-a} relationship until the subclass becomes more mature;
3. 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 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.

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 (e.g., 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.

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).

6.2.5 PATTERN: CREATE ABSTRACT SUPERCLASS

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).

Problem
As programs evolve, abstractions emerge. Abstractions that 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 abstraction. 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 extend 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 defined 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.
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:

- Adding a new class A1, which initially contains no locally defined members;
- Making A1 the new superclass of both C1 and C2;
- Determined the common behavior (functions, or parts of functions) in C1 and C2;
- 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.
- Moving the common functions to A1 and deleting them from the subclasses.

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.
6.3 Forward Engineering

6.3.1 Introduction

The new target system is created by moving downward through the levels of abstraction, a gradual decrease in the abstraction level of system representation by successive replacement of existing system information with more detailed information. This downward movement is actually forward movement through the standard software development process, hence, forward engineering. Forward engineering moves from high level abstractions and logical implementation independent designs to the physical implementation of the system. A sequence from requirements through design to implementation is followed. In this process, the risks are due to the degree and preparation during reverse engineering. Projects are exposed to more risk with the alteration or addition of new requirements.

6.3.2 Re-engineering Approaches

There are three different approaches to software re-engineering. The approaches differ in the amount and rate of replacement of the existing system with the target system. Each approach has its own benefits and risks.

6.3.2.1 Big Bang Approach

The "Big Bang" approach, also known as the "Lump Sum" approach, replaces the entire system at one time, as shown in Figure 6.2. This approach is often used by projects that need to solve an immediate problem, such as migration to a different system architecture.
The advantage to this approach is that the system is brought into a new environment all at once. No interfaces between old and new components must be developed, no mingled environments must be operated and maintained. The disadvantage with this approach is that the result tends to be monolithic projects that may not always be suitable. For large systems, this approach may consume too many resources or require large amounts of time before the target system is produced. The risk with this approach is high, the system must be functionally intact and work in parallel with the old system to assure functionality. This parallel operation may be difficult and expensive to do. A major difficulty is change control; between the time the new system is started and finished, many changes are likely to be made to the old system, which have to be reflected in the new system. It is a lot of work to stay current and not to lose a capability that has been put in the old system. That is, what is being reengineered is likely to change.

6.3.2.2 Incremental Approach

The "Incremental" approach to re-engineering is also known as "Phase-out". In this approach, shown in Figure 6.3, system sections are re-engineered and added incrementally as new versions of the system are needed to satisfy new goals. The project is broken into re-engineering sections based on the existing system's sections.
The advantages to this approach are that the components of the system are produced faster and it is easier to trace errors since the new components are clearly identified. Since interim versions are released, the customer can see progress and quickly identify lost functionality. A benefit is that change to the old system can be easier dealt with, since changes to components that are not being re-engineered have no impact on the current component. A disadvantage to the Incremental approach is that the system takes longer to complete with multiple interim version that require careful configuration control. Another disadvantage is that the entire structure of the system cannot be altered, only the structure within the specific component sections being reengineered.

This requires careful identification of components in the existing system and extensive planning of the structure of the target system. This approach has a lower risk than the Big Bang because as each component starts re-engineering, the risks for that portion of the code can be identified and monitored.

6.3.2.3 Evolutionary Approach

In the "Evolutionary" approach, as in the Incremental approach, sections of the original system are replaced with newly re-engineered system sections. In this
approach however, the sections are chosen based on their functionality, not on the structure of the existing system. The target system is built using functionally cohesive sections as needed. The Evolutionary approach allows developers to focus re-engineering efforts on identifying functional objects regardless of where the tasks reside in the current system. As shown in Figure 6.4, components of the current system are broken by functions and re-engineered into new components.

![Figure 6.4. Evolutionary Reengineering Approach](image)

Figure 6.4. Evolutionary Reengineering Approach

The advantages of Evolutionary re-engineering are the resulting modular design and the reduced scope for a single component. This approach works well when converting to object-oriented technology. One disadvantage is that similar functions must be identified throughout the existing system then refined as a single functional unit. There may also be interface problems and response time degradation since functional sections of the original system are being reengineered instead of architectural sections.

6.4 Organisational Patterns for Moving Forward

6.4.1 Background

Change is one of the few "constants" of software engineering. While managing this change is a challenge for all software-intensive organizations, managing change becomes more difficult when organizations build product-lines.
While reuse helps manage change across the product-line more effectively, managing change for specific components becomes more difficult. When organizations undertake software reuse, the need to manage change for components grows due to the expanded usage and often an expanded life-cycle of a software asset. These changes need to be managed in a coordinated fashion. This section describes six organizational patterns that support software reuse and address these concerns.

6.4.1.1 Develop a Shared Platform;
6.4.1.2 Maintain Reuse Platform Identity;
6.4.1.3 Integrate Reuse and Tie to the Bottom Line;
6.4.1.4 Reuse More than Just Code;
6.4.1.5 Treat Reusable Components like Products; and
6.4.1.6 Merge After Cloning.

Some of these patterns focus on organizational issues, focusing on practices that support managing the rhythms of change and managing cloning to maintain a shared code base.

6.5 Pattern #1: Develop a Shared Platform

6.5.1 Problem Statement
Multiple projects incur redundant development costs by independently implementing identical or similar functionality.

6.5.2 Context
Projects within an organization are loosely-coupled, allowing each project to more effectively focus upon meeting the needs of its set of customers. However
opportunities might be realized to achieve economies of scale by centralizing some functionality in a common platform.

Forces

6.5.2.1 Multiple implementations of similar functionality are often more expensive than a shared solution;
6.5.2.2 Projects want to minimize their development costs;
6.5.2.3 Projects want to maximize the revenue potential for their products and services;
6.5.2.4 The existing organizational structure may not easily support coordination across projects;
6.5.2.5 The needs among projects, while similar, may not be identical - or at least may not appear to be identical to the project staff; and
6.5.2.6 Expertise in common areas may be spread across projects; such staff may be in high demand within their projects.

Solution

6.5.2.7 Bring together staff representing multiple projects, to assess common needs;
6.5.2.8 If the areas of common need are significant, charter the development of a shared platform, funded and staffed either jointly among the application projects, or by a corporate level "core" organization;
6.5.2.9 Define plans for projects to transition onto the common platform; and
6.5.2.10 Maintain close ties between the platform projects and the application projects, to both help focus the platform development and increase the confidence level among application projects that the platform will meet their needs.
6.5.3 Result

A platform is developed to address the needs shared among the application projects. Cost savings (in areas addressed by the shared platform) are realized among the projects, over time. Time-to-market is also reduced. Subsequently, opportunities may also be realized to share application level components among projects. Opportunities to migrate staff among projects may be increased given their shared platform understanding.

6.5.4 Consequences

For the benefits of a shared platform to be realized over time, it is important to Maintain Reuse Platform Identity. An investment is required to establish the platform before the benefits can be realized.

6.6 Pattern #2: Maintain Reuse Platform Identity

6.6.1 Problem Statement

A shared platform has been built, or is envisioned, but potential users, aside from the pilot adopters, are not willing to commit to using the shared platform.

6.6.2 Context

Over time, user needs change, technologies and interfaces change, and designers' insights change. Change is one of the inherent complexities of software. Platforms and frameworks have been recognized as a means for managing change and for reducing development costs and development intervals. As a result the organization has decided to Develop a Shared Platform.

New platforms are often introduced with a pilot adopter. While it is necessary for the platform and pilot application development groups work closely together to make the initial use of the platform successful, it is tempting to associate a
platform project too closely with the pilot application. This association is sometimes realized by placing platform staff and application development staff in a common organization. This close association may benefit the application developers with access to and control over the platform development staff.

From the application developers' perspective, distinctions between platform and application are irrelevant to their customers. It may seem expedient to make arbitrary changes to the platform to accommodate application specific needs, in ways that compromise the generality of the platform. A further expedient may be to merge application and development staff; in the process, platform identity and platform support may be lost.

This close collaboration can cause other potential adopters to see the platform as only suitable for the pilot application, or that the pilot application has too much control. As a result of such perceptions, other applications groups don't commit to using the platform. A warning sign that the platform identify has not been maintained is that the schedules for platform adoption after the pilot application begin to slip.

6.6.3 Forces

6.6.3.1 Costs can be reduced and economies of scale achieved through the use of common platforms;
6.6.3.2 Change is inevitable;
   6.6.3.2.1 ironically, introducing change is hard;
   6.6.3.2.2 pilot projects mitigate some of the challenges of introducing new technology;
6.6.3.3 Platform development groups are driven by multiple, conflicting goals;
6.6.3.4 Near-term focus of the application development groups using the platform;
6.6.3.5 Long term requirements for potential adopters of the platform;
6.6.3.6 Application development groups are focused on near-term deliverables to specific customers and markets. In order to maximize product quality while minimizing product development intervals and development costs, application development groups seek to reduce and eliminate:
6.6.3.7 Unnecessary dependencies between organizations;
6.6.3.8 Unnecessary distractions placed upon their staff;
6.6.3.9 Funding sources often drive the priorities of an organization;
6.6.3.10 Organizations that are closely tied together (e.g. in the same business unit) often have better communication than organizations that are not as close;
6.6.3.11 Potential platform users that are not confident that a platform will be responsive to their needs will not use the platform; and
6.6.3.12 A platform closely tied to a single project may not be seen as responsive to the needs of other users.

6.6.4 Solution
6.6.4.1 Establish a separate organization for the platform;
6.6.4.2 Keep the staff of the platform development organization and the initial application organization separate. There should be good lines of communication, and clear, commonly understood expectations between the two groups;
6.6.4.3 Use stable corporate funding to get the platform started, and then ensure stable funding from the application organizations once the platform is established;
6.6.4.4 There needs to be an advocate for the platform at the corporate level;

6.6.4.5 The platform organization should act like a product organization to engage customers and address customer needs.

6.6.5 Result

If platform identity is maintained, applications are better able to have their needs met, and the value of the platform is greater because it can be used by more applications. If the platform identity is maintained, the platform will be used more widely, which results in reductions in cost and time to market across a larger portion of the organization.

6.6.6 Consequences

Maintaining platform identity may make achieving success with the initial application more challenging.

6.7 Pattern #3: Integrate Reuse and Tie to the Bottom Line

6.7.1 Problem Statement

Incentives to establish reuse are not taken seriously.

6.7.2 Context

The organization has multiple improvement efforts going on simultaneously, e.g. process improvement, new tools, risk management - as well as an organization wide reuse effort. The reuse effort is being ignored and rejected by the organization as just "one more" activity to distract engineers and managers from meeting the business objectives.
6.7.3 Forces

6.7.3.1 Improvement efforts are related - but staff focusing on a particular improvement effort may be unaware of other, related improvement efforts;

6.7.3.2 Organizations are often under tremendous pressure to meet business objectives of completing projects on time and on budget;

6.7.3.3 Some level of reuse often happens, even without a formal reuse program; and

6.7.3.4 Organizations cannot be motivated by mandate.

6.7.4 Solution

6.7.4.1 Reuse leaders should identify fit and ways to serve and be served by other improvement efforts;

6.7.4.2 Coordinate and align the reuse effort with other improvement efforts. The organizational structure should support shared responsibility among the improvement efforts;

6.7.4.3 Identify and communicate common, coordinated recommendations to practitioners regarding the improvements - emphasizing the value of the improvement efforts to the managers and engineers whose support and actions are needed to make the improvement successful;

6.7.4.4 Manage the coordination among the efforts;

6.7.4.5 Track, on an ongoing basis (and require estimates in budget proposals for) how much reuse has been achieved and how much savings have been realized;

6.7.4.6 Integrate reuse throughout the life-cycle, not just at coding.
6.7.5 Result

The engineers and managers involved with the reuse activity receive consistent, easy-to-understand and follow improvement guidance that shows them how to better achieve their business objectives. The reuse effort is more likely to take hold, and deliver the expected results of reduced costs and reduced time-to-market.

6.7.6 Consequences

Even if reuse is tied to other improvement efforts, the improvement efforts still need to be tied to the bottom-line and other motivations of both managers and engineers. If the value of the improvements to the participants is not clearly communicated, the integrated improvement efforts may be more likely to fail. Even with careful coordination and consolidated 'improvement' strategy, the effort may be seen as more complex than each of the efforts individually. The reuse effort could also be canceled if other improvement efforts are deemed more critical (this may not be bad).

6.8 Pattern #4: Reuse More than Just Code

6.8.1 Problem Statement

Engineers don't reuse already developed code components because the components either don't fit with other components, or do not quite fit the requirements.

6.8.2 Context

Managers and practitioners are beginning to make a dedicated, long-term effort to improve results and customer value. Software reuse is seen as a way to achieve these results.
6.8.3 Forces

6.8.3.1 Managers and practitioners often equate reuse with code, not with knowledge capture or organizational learning;

6.8.3.2 If different parts of the organization have very different practices, then they may find it more difficult to share assets and share people among different groups;

6.8.3.3 Code components reflect decisions made during requirements and design;

6.8.3.4 Consistently achieving high levels of process maturity requires an understanding and application of sound requirements and design techniques.

6.8.4 Solution

6.8.4.1 Reuse software assets from all phases of the life-cycle including:

6.8.3.4.1 Processes;

6.8.3.4.2 Checklists;

6.8.3.4.3 Project management templates;

6.8.3.4.4 Requirements; and

6.8.3.4.5 Designs.

6.8.3.5 Establish a repository and other automated tools to share the assets; and

6.8.3.6 Tie reuse to knowledge capture activities.

6.8.5 Result

A number of benefits have been attributed to this approach, although not all organizations realize all of the benefits. Creating and reusing non-code assets makes code reuse more likely. Organizations are also better able to bring in and
train new staff. Reusing non-code assets also results in more direct benefits such as improved cycle-time, consistency and repeatability and reduced costs, risks and surprises.

6.8.6 Pattern #5: Treat Reusable Components like Products

6.8.6.1 Problem Statement

Application developers typically associate components with "use at your own risk stuff" -- they don't trust or use components they don't fully understand.

6.8.6.2 Context

An internal organization is chartered to introduce a new set of behaviors and/or a new technology through their development and management of reusable components. The goal of this activity is to achieve a significant improvement in delivering higher value and better business results. The target adopters are application development organizations. The application development organization is under tight deadlines. Those who supply components must support a diverse group of product developers, each with his or her needs. Even though each component has one owner, others may need to make changes to his or her component when he or she is away to meet deadlines.

A warning sign that this problem is occurring is that component owners do not hear about change requests from component users, either because components are not being used at all, or because a configuration management (CM) system is not being utilized to capture and communicate change requests.

6.8.6.3 Forces

6.8.6.3.1 If individual developers are not confident that component features critical to their work will not change, and that new
component features will be addressed in a timely fashion, then
developers will probably not use the components;

6.8.6.3.2 The supplying organization must add new features and feature sets
to its components to increase its customer base;

6.8.6.3.3 Potential adopters will not use a component if they are at all unsure
that it will be maintained and that it will be managed -- unexpected
defects coming from lack of CM are not cool;

6.8.6.3.4 With a standard configuration management system, only one
component owner has access to a component;

6.8.6.3.5 A standard configuration management approach focuses on
tracking baselines of entire systems, e.g. NamedStableBases, and
not tracking specific components across multiple systems;

6.8.6.3.6 Without a product mentality, component suppliers can easily
become detached from their customers;

6.8.6.3.7 Lack of information raises the perceived risk of using a
component. Perceived risk on the part of (potential) reusers causes
them to lose confidence that a component will meet their needs,
and hence makes them reluctant to reuse the component; and

6.8.6.3.8 Potential reusers are often willing to forego their desire to know
everything about the asset's internals and if they trust the supplier's
credentials and know that the supplier has a competent CM tool
and process.

6.8.6.4 Solution

Treat reusable components like products. A lot is known about how to build
and market products that go within other products, (though reuse efforts still do
believe they are doing something entirely different). For example, the organization
responsible for reuse should gain the confidence of the target customer base through building a high-quality product, tracking and responding to change requests. Gain endorsement from a respected expert and end user, and draw to your team a group of developers that people trust. The reuse organization shares developer's risk through specific commitments to deliver the features promised, managing and responding to change requests, and assuring that critical features will persist. The benchmark is set by product vendors (who certainly are not perfect). Make use of user groups and other communications forums to encourage the platform user community to exchange information and drive consensus regarding proposed platform changes. A flexible CM system and process is a part, but not all of the solution. Developers should be able to view version histories, and outstanding change requests. Users should also be able to use the CM to report defects and make other change requests.

6.8.6.5 Result

Reusable components are presented in a way that developers are more comfortable with. They are willing to try using components because they have a clearer understanding of who is responsible for what. They see the risk of using a component counteracted with endorsements from their peers and experts they respect. Use of reusable components grows from small groups within product organizations to a point where major product features may depend on the components. User community is better informed about specific reusable components. This knowledge reduces their perceived risk to reusing components, and increases component usage. The flow of information helps develop relationships between the component owners and reusers.
6.8.6.6 Consequence

As usage grows, a problem in a component may adversely affect multiple business-critical products.

Example

In the early days of acceptance of C++ in the AT&T development community, it was determined that acceptance would go faster if users were supplied with a set of standard "computer science" functions that would make system development using C++ easier. This set came be known as the C++ Standard Components Library and contained functions like: Lists, Maps (associative arrays), String, Blocks (unbounded arrays with memory allocation as needed) and so on. The Standards Component Library was very popular and was used by hundreds of projects across AT&T. The components were put under configuration management control and change requests (bug reports and enhancement requests) were tracked in the configuration management system, which associated each source change with specific change requests. Any developer assigned a change request could change a component and an audit trail of the change would be kept by the system.

The C++ hotline used the configuration management system to enter customer change requests, track the quality and change history of the components and support the assembly and packaging of new releases. The project's configuration management was more than just the supporting system, it included the process (e.g., review boards) as well. Acceptance was high and cloning was not a problem. The success of the Standard Components Library was based on many factors, which helped build confidence among potential users:

1. Quality of people, many of whom published books and had excellent reputations within the company;
2. The quality of the components and the support that was given. Configuration Management played a supporting role here in that without it, quality and support wouldn't have been as good as it was. Releases of reusable components were not that much different from releases of other products that have to run in multiple environments.

6.9 Pattern #6: Merge After Cloning

6.9.1 Problem Statement

Organizations clone software to add components to quickly respond to customer requests, but the organization does not want to get saddled with the long-term costs associated with cloning.

6.9.2 Context

While cloning offers a means to quickly respond to pressure to develop new features quickly, cloning often has far-reaching consequences. When a developer clones, code is duplicated, complicating product tracking and management. It doesn't take too long before developers build new and complex extensions on the clones making them incompatible with the code base. As a result, code grows rapidly in size and complexity. This dramatically increased the maintenance burden for each product. Periodic merging of clones is needed.

6.9.2.1 Forces

6.9.2.1.1 Rapid response requires that developers have access to and ability to modify existing source code;

6.9.2.1.2 It is often easier to change source by "copy and modify" (ie by creating clones) rather than by extending or generalizing the original source;
6.9.2.1.3 White box access to source is necessary to respond rapidly to multiple quick response requests from end-customers; and

6.9.2.1.4 Generalizing existing components may require coordination not just with the component supplier, but also other users of that component.

6.9.3 Solution

6.9.3.1 Merge the clones regularly;

6.9.3.2 Provide application developers with "white box" access to at least selected parts of the platform source;

6.9.3.3 When cloning, encourage developers to duplicate the smallest portion possible;

6.9.3.4 Establish very clear rules for what types of changes can be made through cloning;

6.9.3.5 Identify clones by watching for signs, such as duplicate functionality, email postings, etc, and then examine the code to determine if cloning is taking place;

6.9.3.6 Allow only one clone of any given component at the same time;

6.9.3.7 Encourage users (application developers) to submit bug reports, bug fixes and suggestions for extensions and generalizations; and

6.9.3.8 Provide support to component users so that they can more easily use the generalized software.

6.9.4 Result

6.9.4.1 Code base improves in robustness and reliability; bugs get fixed and stay fixed. There is a single point of maintenance, which simplifies the task of building products;
6.9.4.2  Reduced maintenance costs. Every line of shared code in use, means that there are 2 or 3 lines of avoided code somewhere else;

6.9.4.3  Application developers can respond to market pressures more quickly than if generalized components were built of modified initially;

6.9.4.4  Reuse suppliers build trust with users; and

6.9.4.5  Code base stays upward compatible.

6.9.5 Consequences
Additional effort must be spent to reincorporate cloned software. There is also a substantial risk that cloning could get out of hand, e.g. cloned chunks are too big, or there are too many clones, and making the practice too expensive to sustain. There may be limits to the number of simultaneous clones that an organization can create and still merge them back into the base. An organization may have reached or passed this limit when the time required to merge the clones slows down, or if a merge cannot be completed before the next scheduled merge. Another warning sign is that merges cannot be completed with major changes.

6.9.6 Example
One organization, in order to meet schedule, cloned their product in order to run the application in an additional operating environment. The organization recognized that in doing so, they had effectively doubled their maintenance costs, and that there was no way that they would be able to support all of the operating environments in their plans. They moved back to a common code-base and reduced their cycle-time from 12-24 months to 12-14 months, and they also reduced their maintenance costs.