Software Reuse in an Industrial Setting: A Case Study

Michael Francis Dunn

Computer Science Report No. RM-90-02
August 1990

SOFTWARE REUSE IN AN INDUSTRIAL SETTING:

A CASE STUDY

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science (Computer Science)

by

Michael Francis Dunn

August 1990

APPROVAL SHEET

This thesis is submitted in partial fulfillment of the

requirements for the degree of

Master of Science (Computer Science)

[illegible]

Michael F. Dunn

This thesis has been read and approved by the examining Committee:

[illegible]

Thesis Advisor

Robert P. Cook

Committee Chairman

[illegible]

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering
and Applied Science

August, 1990

ABSTRACT

The topic of component-based software reuse has drawn a great deal of attention in recent years. It is widely regarded as a promising strategy for improving the productivity and reliability of large-scale application systems development projects. While much has been written on the storage and retrieval of software components, there is very little information available on the nature of reusable components themselves. Sparse, too, is documentation of experiences in developing practical, real-world systems using components.

This work addresses these problems by providing a case study of the introduction of component-based reuse into an industrial software development organization. The work described was undertaken in cooperation with Sperry Marine, Inc. during the first half of 1990. Presented is a description of the Sperry Marine Commercial Engineering division's development environment and mission. A detailed account of how a core set of potentially reusable software artifacts was derived from existing Sperry Marine systems is also presented. The main part of the work, however, explores the issues involved in designing and implementing a system comprised of reusable components. A specific problem domain is described, along with methods for designing, implementing, and evaluating a set of object-oriented components for a spectrum of systems within this problem domain.

ACKNOWLEDGEMENTS

My adviser, John Knight, provided me with patient, insightful support at every step of this project. He was always eager to share ideas, and to steer me in the right direction when I needed it most. My gratitude to him is a debt I can only dream of repaying.

I gratefully acknowledge the grant from Sperry Marine, Inc. which made this research possible. In addition, numerous Sperry Marine employees contributed their valuable time to assist me in the technical issues of this project. John Yancey devoted countless hours to helping me understand the subtleties of real-time navigation software, and was indispensable in providing me with all the resources I needed to make this project a success. Michael Berrang provided much of the code on which the prototype described herein was based. He was also extremely helpful in explaining the basic implementation issues involved in electronic navigation chart management. Other Sperry Marine personnel who gave their assistance were Becky Bird, Bruce Carriker, Jim Corbett, Neal Lebedin, and Steve Scovill.

Special and affectionate thanks goes to my grandmother, Mrs. Frances Terbille, whose love, wisdom, and support helped me weather the rougher storms of higher education.

LIST OF FIGURES

LIST OF TABLES

Table of Contents

1. INTRODUCTION

...every mechanic is familiar with the problem of the part you can't buy because the manufacturer considers it a part of something else.

Robert Pirsig

Zen and the Art of Motorcycle Maintenance

1.1 What Is Software Reuse?

It is common in the software engineering community to hear the phrase *the software crisis* used to describe the difficulties in producing large, reliable software systems. It is equally common to hear *software reuse* posed as a suggestion for helping to deal with this crisis. For years, software developers have noticed that enormous amounts of time and effort go into producing new systems that share many of the characteristics of older systems. The end result of a software reuse strategy is to capture the *work products* that have been developed in earlier systems, standardize them, and use them in new and future systems. The term *work products* as used here refers not simply to source code, but also to the analysis and design that go into every phase of the software lifecycle.

1.2 An Industrial Analogy

To understand the essence of software reuse, consider the airplane. Specifically, think of commercial jet aircraft, the way they are manufactured, and what happens to them during their lifetimes. Jets are very large, complicated machines. It takes a long time to design new models, and many highly skilled people are required to produce and test them. They are expected to last twenty to thirty years, and while they are in service, they could undergo many modifications. For example, airlines typically remodel cabin interiors every few years, and new equipment is added to the cockpit as technology changes and new safety requirements are put in place. Airplanes require continuous maintenance just to stay operational. And, of course, their price tags put them beyond the reach of the average consumer. Furthermore, airlines typically request aircraft on a made-to-order basis.

Now consider large-scale software development. Large software projects typically take years from initial concept to initial release. An embedded system can be in service for as long as the product in which it is embedded. As in the case of airplanes, this could be measured in decades. The skill and experience levels required to produce such systems is enormous, and the use of Ph.D.-level people is not uncommon. A software system almost always requires numerous modifications during its lifetime. These modifications include major functional enhancements as well as problem fixes.

The point of this analogy is that, while both aircraft manufacturing and software development are major industrial undertakings with similar problems, there are two key differences:

- Airplanes are designed modularly using standard components, which eases both the production and renovation process. When USAir and AirCanada order a 727 from Boeing, their requirements might be different, but what they get is still fundamentally a 727. Large software systems, on the other hand, tend to be seen as unique, one-of-a-kind products, and each project usually starts with a clean slate.

- Typically, airplanes can fly when they are delivered. New software products are notoriously error-prone.

The goal of software reuse is to address directly this first issue, and in so doing, indirectly address the second.

1.3 The Reuse Problem

While the reuse of work products from project to project was originally viewed as a promising way to address the difficulties with developing and modifying large systems, efficient and practical implementation of large-scale software reuse has proven to be much harder than anticipated. While the idea has a bewitching intuitive appeal, solutions that take into account the entire software life-cycle remain elusive. As research on the subject progressed throughout the eighties, it became increasingly obvious that this proposed answer to the so-called ''software crisis'' is fraught with problems of its own. Many fundamental questions about component-based development remain open, such as:

- How does one capture the essential characteristics of a problem area, and use this knowledge to design not just one system to address this problem area, but an entire class of systems? The activity of analyzing a problem area for this purpose is known as a *domain analysis* [Nei84].
- How does one create a set of components which can be used in a variety of different systems with little or no change? This problem has a sister problem: How does one analyze already-existing systems to find pieces of code which can be used in systems under development?
- How does one design new systems to take advantage of a set of reusable components?

1.4 Goals of This Thesis

This thesis addresses these questions by providing a case study of a preliminary attempt to introduce software reuse methods into an industrial organization. Of particular interest are methods of *object-oriented development*. In spite of the growing popularity of object-oriented programming in the past several years, there is still very little literature available detailing experiences with this technique in industrial settings.

There is even less data available which focuses on object-orientation as a method for deriving reusable components. That which has been published typically discusses fairly obvious and tractable problem domains, such as windowing software and editors. The problem domain detailed by this study involved a class of embedded, safety-critical systems. Nevertheless, the intent is that the methods presented in analyzing and addressing this domain will be general enough to be used in other domains as well.

Chapter 2 supplies background on the basic issues of reuse, and some examples of how reuse techniques have been applied in industrial settings. Particular attention is paid to component-based architecture and design strategies. Chapter 3 gives a detailed description of the organization under study, and discusses the productivity obstacles that led its programmers to consider reuse as a potentially fruitful development strategy. Methods for evaluating the organization's reuse needs via *code scavenging*, or analyzing existing systems for useful components, are presented in Chapter 4. Chapter 5 constitutes a domain analysis of an application area deemed particularly difficult but very useful by the members of this organization. An object-oriented set of software components was prototyped to address the issues presented in this domain analysis. While Chapter 6 details the method and goals of the component prototyping strategy, the actual design of these components is detailed in Chapter 7. Techniques for assessing the development strategy used to create these components is presented in Chapter 8. A number of these techniques are demonstrated in Chapter 9. A summary and some suggestions for further work is provided in Chapter 10.

2. APPROACHES TO REUSE: AN OVERVIEW

2.1 Chapter Overview

In 1983, ITT organized one of the first major conferences devoted solely to the emerging study of software reuse. Since then, a vast amount of material has been published on the subject. This chapter reviews the literature most relevant to the Sperry Marine study. A number of general overviews are discussed in order to establish an understanding of the basic issues and problems with reuse. The emphasis then shifts to the topic of *component-based development*; what it is, and what design and architecture strategies have been proposed using it as the underlying model.

2.2 Reuse Strategies

Researchers and practitioners have experimented with a number of different reuse strategies. Overviews to the most common techniques are chronicled by Horowitz and Munson [HoM84], Jones [Jon84], and Wegner [Weg87]. Each of these overviews has a different slant. Wegner approaches reuse as a *capital-intensive technology*, borrowing an Industrial Revolution-era analogy that describes heavily investing in parts and equipment, and then recouping the cost of that investment through repeated use. He provides a number of examples of where this is possible in software development, and draws significant parallels between software reuse and reuse in other fields, such as the reuse of proofs in mathematics. Horowitz and Munson do not approach the subject with quite the depth and breadth of Wegner, but describe the basic issues in reuse, along with the varieties. They outline three main varieties of reuse:

- *Code* - The traditional view of reuse as a process of using pieces of code as components which can be interchanged in a variety of different systems. The authors point out several open questions in this area, such as *How does one specify such components?*, *How does one implement such components?*, and *How does one catalog such components?* They have a grim prognosis for this strategy, pointing out the difficulties of modifying old code to suit new requirements.

- *Program Producing Systems* - This includes application generators and domain-specific languages. Such systems often include very high level, interactive front-ends to allow the user to specify a problem using nomenclature specially suited for the problem area in question, rather than a more general, procedural language. The importance of domain analysis is stressed here as a prerequisite to creating both of these types of systems.

- *Innovative Language Features* - Fairly recent language constructs, such as Ada packages and generics, allow the programmer to enforce encapsulation, thus encouraging the use of data abstraction. This can, in principle, lead to a stand-alone, building block approach to software development.

Jones' study focuses on reuse techniques common in industry at the time. He cites a number of empirical studies on what sorts of code were being reused, and what sorts were not. He also discusses architectural and design issues as crucial areas in need of more research. He concludes his study with this poignant comment:

> *The future is hard to predict, but a reasonable prognosis for 1990 would be 50 percent of all code at that time would be reused, and 50 percent would be unique among leading-edge enterprises* [Jon84, p. 55].

2.3 Research on Software Components

The study described by this thesis focuses mainly on reuse from a component, rather than from a language or system generator, perspective. With respect to component-based development, much work has been devoted to the creation and management of software libraries and collections. Studies of such library systems appear in Burton, *et.al.* [BAB88], Frakes and Nejmeh [FrN87], and Tyler and Weiss [TyW89]. Many of the component collections which have been successfully put together consist of general-purpose routines and data structures. Domain-specific collections, such as the CAMP missile guidance components, have also been assembled [MMM87].

Related to library creation is the issue of software representation and its impact on how components can be retrieved from libraries. Prieto-Diaz and Freeman explain the innovative technique of faceted classification in [PrF87]. Frakes and Gandel provide a model that incorporates this and other common techniques in [FrG89].

The issue of what makes a good software component has been described by Wegner in [Weg89]. He describes the techniques of data abstraction, function abstraction, and process abstraction, and discusses Ada as a language that supports these techniques. Parnas, *et.al*, discuss information hiding and interface issues in [PCW89]. They present these techniques in light of their study of the redesign of the flight control software for the A-7E military aircraft.

2.4 Designing Component-Based Systems

While much work has been done on the storage and retrieval of software components, less has been written on how to develop systems from a set of available components. The issues involved are complex. The nature of a software component is difficult to define. It is tempting to think of discrete program units such as C functions or Ada packages as constituting components. However, these units can be combined into still larger units, or *subsystems*, which provide for a major portion of a system's functionality. It is perfectly legitimate to think of these larger units as being components as well.

One can see this principle at work in a number of real-world problem areas. The engine of an automobile serves as a good example. One often thinks of an engine as being a single component. However, removing the engine block reveals a variety of complicated, cooperating sets of smaller components, each constituting a subsystem in its own right. These smaller subsystems can often be divided into yet smaller units. It is this process of identifying related groups of components and conceptually bundling them into larger units that Neighbors identified by the terms *parts* and *assemblies* [Nei81].

The next two subsections explore this component-based development concept further, first by introducing the design issues to consider, and then discussing the interplay between a system's high-level design and its low-level components.

2.4.1 What Does "Component-Based Design" Mean?

As used here, the concept of component-based design can refer to a number of interrelated activities. Among these are:

- Structuring new systems to take advantage of existing components.
- Structuring new systems to yield new components.
- Structuring new systems so that new versions can be derived from them easily.
- Structuring components so that they can be used in a variety of different systems.

The techniques described here, and those described later in the thesis, give insights into how these activities can be accomplished.

2.4.2 The Relationship Between System Architecture and Component Design

It is apparent that the high-level design of a system has a major impact on the design of the components that comprise the system. Parnas provides an elegant demonstration of this in [Par72]. In this classic paper, Parnas provides two decompositions of a Key Word In Context (KWIC) index system. One decomposition uses a traditional top-down structuring, the other uses a strict information hiding approach. Although the two resulting systems are functionally equivalent, the differences between the functions which make up the two systems is remarkable. In the system employing information hiding, crucial design decisions, such as how lines of text are stored during processing, are isolated into single modules. In the top-down decomposition, this and other design decisions are scattered throughout the system, making it difficult to introduce changes to one part of the system without affecting other parts.

The ramifications of this in constructing systems from reusable parts are clear. One can decompose a problem statement into a system design in any number of ways. However, each decomposition is likely to lead to a different set of components. If the designer first decomposes a problem and then later searches a library for a set of components which can be used to create the desired system, he is likely to discover a large number of components which almost do what he needs, but not quite. Oskarsson provides further evidence of this when he says

> *...Attempts to make software modules reusable in radically different system architectures can rarely succeed. Software reusability is only feasible inside a specific architecture or class of architectures...* [Osk89].

2.5 Architectural Strategies

Numerous strategies have been tried for organizing system architectures. Shaw provides some general comments on some of the more common of these strategies in [Sha89]. As more experience is gathered, the trade-offs between these architectures will become better understood. The strategies mentioned below, the *hierarchical, heterarchical*, and *object-oriented* models, have been proposed as promising models to follow when dealing with component-based software reuse.

2.5.1 Hierarchical Model

The term *hierarchical* as used here does not refer to the traditional top-down approach, but rather to the decomposition of systems into independent units of work, or *modules*, coupled with information hiding. As mentioned earlier, Parnas, *et.al.*, discuss this process in detail in [PCW89]. The hierarchy here arises from the fact that each module in the system is composed of many smaller submodules, allowing one to view the decomposition as a tree structure. Each module at each level has its own set of ''secrets'', or design and implementation details subject to future change. By isolating these secrets in specific modules, one can minimize the dependency that a system has on any given design decision. This allows parts of the system to be modified without having to worry about the impact of that change rippling through to other areas of the system. Thus one can view a system as consisting of independent units.

Parnas, *et.al.*, draw a conclusion about this strategy that sheds light on the limits to the reuse and reconfiguration problem in general:

> *Our software is designed to be flexible, not general.... We do not make software that is designed to be used without change in many situations; that would be too inefficient. Instead, we have designed the software so that it is easy to adapt to a new situation. We are able to tune the software to our particular situation, but confine that tuning to specific modules so that the software can be readjusted when the situation changes.*

From the way Parnas uses the word ''inefficient'' here, it is not clear whether he is referring to the efficiency with which software executes, or the efficiency with which it is developed. Both interpretations seem valid. This collision between efficiency and generality is an important pragmatic concern. In much the same way that one cannot make an airplane out of a Volkswagen, one must expect there to be limits on the contexts in which one can use a given module, and the kinds of variations that one can derive from a given system.

2.5.2 Heterarchical Model

Hofstadter describes a heterarchy as "a program which has such a structure in which there is no single 'highest level' " [Hof79]. This also seems to be a good way to describe the architecture documented by Oskarsson in [Osk89]. In this strategy, all modules are classified as one of three types:

- *Data modules* are responsible for handling the access and control of specific data structures used by non-data modules.
- *Interface modules* take care of the details of communicating with the outside world. This might include human users or various kinds of hardware.
- *Activity modules* control particular aspects of the systems behavior.

These modules communicate strictly by asynchronous message-passing semantics. No data is shared between them. This helps to keep them decoupled. The topology of this architecture can either be a star, where all modules are connected to a central module which switches control between them, or a chain, where control moves sequentially from module to module. Adhering to either of these strategies or a combination of the two makes it easy to add new functions or modify existing ones. One does not have to worry about an elaborate set of intermodule dependencies when introducing changes to the system. Thus new versions of the system can be derived easily by using large amounts of code from earlier versions.

2.5.3 Object-Oriented Model

Object-oriented design (OOD) has received much attention in the past several years, and is moving rapidly out of the laboratory and into commercial programming environments. Unfortunately, in spite of all the publicity, OOD still seems to be a buzzword in search of a standard definition. This gives rise to much confusion. In [Boo86], for example, Booch provides a lengthy explanation of object-oriented development techniques using Ada as the language vehicle. However, Ada does not support *inheritance*, the ability to define object templates which derive their data and operations from previously defined templates. Many people insist that this code derivation technique, itself a form of reuse, is a key feature of true OOD. For this discussion, OOD will be defined according to Meyer:

> *Object-oriented design may be defined as a technique that, unlike classical (functional) design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs.* [Mey89]

Reports on industrial experience using OOD techniques have been gradually making their way into the literature. Jette and Smith describe OOD and its impact on reuse in [JeS89]. They describe a system called HyperClass, which is a tool for building interfaces to knowledge-based systems. Implemented in an object-oriented extension to Lisp called Class, HyperClass consists of a number of building blocks useful in creating common user interface objects such as editors, windows, and menus. By defining new classes derived from the classes defining these and other objects in the system, they report an enormous savings in new coding, without sacrificing functional flexibility.

Wybolt's study is another recent addition to this literature [Wyb90]. In this article he discusses an ongoing effort at Cadre Technologies to redesign their Teamwork product to take advantage of the object-oriented features of C++. He states with regard to reuse that the inheritance concept has proven advantageous, but notes that in order to reuse a particular class, one must include the entire inheritance hierarchy in which the class is defined. For those not used to it, this can be a difficult adjustment.

2.6 Summary

This chapter began by discussing general software reuse issues, and quickly narrowed to the issue of designing component-based systems. Particular emphasis was placed on experiences with component-based design in industrial settings. This issue is at the core of the work undertaken during this case study: How effective is a component-based reuse strategy in developing industrial software, and what design paradigms serve as the most effective ways to implement this strategy? The next chapter begins to explore this question in more detail by presenting a sketch of a particular software development organization.

3. AN INDUSTRIAL SOFTWARE ORGANIZATION

3.1 Development Mission

Sperry Marine's Commercial Engineering division designs embedded systems for shipboard navigation management, automatic piloting, and harbor traffic control. Their shipboard systems have been used by oil companies for their tanker fleets, and numerous other commercial shipping enterprises. Their harbor management systems have been customized for such diverse corners of the world as Wales, the Virgin Islands, and Oman.

Shipboard navigation systems are used to assist the crew in all aspects of planning and executing the ship's voyage. This activity includes planning various navigation checkpoints known as *waypoints*. Other functions include the ability to electronically display digitized navigation charts, to digitize paper navigation charts while under way, to display radar input on high-resolution raster screens, and to monitor vital shipboard systems such as the engine room.

The automatic piloting system is closely affiliated with the navigation system. It is coupled with the ship's steering system, and receives input from the navigation system via messages over a local area network. The automatic pilot's primary job is to maintain the proper course preset by the navigator, even in times of bad weather and rough seas.

Harbor management systems are analogous to air traffic control systems. The busier harbors around the globe must deal with a constant flow ships moving into and out of port. Safety dictates that this traffic proceed in and out of these confined spaces in an orderly way. Harbor management systems require numerous radar sites to be placed in strategic locations around the harbor. These radar sites provide target information that gets displayed on the harbor controller's video screen as overlays to navigation charts of the harbor area. Thus the controller gets a complete picture of where ships are in relationship to each other, and to major hazards and geographical features in the harbor area.

3.2 Hardware and Software Platforms

Sperry Marine has a variety of different development and target hardware and software platforms. This reflects the diversity of their business interests and customer requirements. It also reflects a reality common to most software development organizations: technology changes over time, but older systems still require support in the form of occasional enhancements and problem fixes. The platforms which provide this support must be kept available.

3.2.1 Hardware

Much of the commercial development at Sperry Marine is done on IBM PC-class machines running Microsoft's Disk Operating System (DOS), or on Hewlett-Packard workstations running Unix. Processing needs are met by using the Intel 286/386 or the Motorola 68000. Networking needs on the target systems are fairly complex. As was mentioned earlier, for example, navigation management systems are networked together with the automatic piloting systems. This is achieved using an IBM Token

Ring network, with application-level communication support provided by the ''SEANET sockets'' described in [SWC88].

Sperry Marine products increasingly rely on the ability to display high-resolution images of various types, particularly navigation charts. The Raster Scan Radar (RASCAR) product relies on graphics to display radar-input target images; this serves a similar role as traditional ship-board radar systems, but without the usual rotating scan bar. On top of these radar images can be superimposed simple navigation chart outlines. The NEC ACRTC graphics processor is the primary engine driving this equipment. Newer products feature higher resolution displays, faster processing speeds, and the ability to do display list processing in hardware. Sperry Marine has been using Matrox's graphics processor to drive these displays, which are used in their navigation and harbor management systems. They have been considering adding VGA ability to this collection because of its lower cost.

Input and output provides a unique set of challenges because of the environments in which this equipment is used. The people most likely to use these products are not accustomed to using computers. They are people working in often hazardous situations who need ways of getting information into and out of the system as quickly as possible. As much input as is practical is done via touchscreen or mouse. Keyboard input, of course, has to be done when entering information such as passwords or voyage planning information. Input to the shipboard piloting system is especially interesting; given the *steer-by-wire* nature of the system, steering input could be done using the same touchscreen, mouse, and keyboard combination used by other parts of the system. Navigation is a very tradition-bound enterprise, however, and pilots are used to controlling their ships using steering wheels. Providing a different interface would be counter-intuitive to them, and create a significant safety risk.

3.2.2 Software

All development is done in the C language. The compilers used support the ANSI standard.

One of the most important features of Sperry Marine's software environment is the availability of a set of tools to support multitasking. Created locally by Sperry Marine programmers, these MultiTasking Executive (MTX) tools enable the programmer to create separate, independent tasks to manage the processing of various hardware device inputs and outputs, and user input and output. Thus the programmer does not have to create awkward control loops to check for periodic input or provide periodic output.

As mentioned earlier, different versions of particular applications are in active use in the field at any given time. Sperry Marine maintains source code for their systems either by using SCCS on a centralized Unix-based file server, or the Code Management System (CMS) on their DEC VAX.

3.3 Development Process

Programming is done in small teams of about three or four people each. With teams this small, each member is responsible for a very large part of the system. Programmers thus tend to be ''jacks-of-all-trades'', being familiar with hardware internals as well as software.

Hardware and software requirements are provided by Sperry Marine's marketing division. There is currently no single standard design strategy; this is left up to each team. As such, they have no integrated tools to assist with the design process. Design work is typically done in pseudo-code, or via the creation of module interface specification documents. This work is usually done in the context of the traditional *waterfall* model of development [Dav88], although some projects have used rapid prototyping as well.

Sperry Marine has recently started using code reviews as an effort to enhance reliability. Their current review method is fairly typical of methods used in other organizations; each programmer on the review team familiarizes himself with the code prior to the review. Programmers then meet as a group, where one person is designated as the *reader*, one serves as a *moderator*, and the rest serve as *reviewers*. The programmer himself attends as well, although not as an active participant. Code is evaluated for correctness and conformance to standards.

Testing methods vary from project to project, although a large variety of simulation software is available to simulate radar target input and steering commands. A separate group, the Software Quality Assurance department, is available to perform final product testing.

3.4 Difficulties

The diversity of hardware platforms, along with the complexity inherent in developing real-time multitasking software, makes it difficult for Sperry Marine to efficiently develop reliable applications. These difficulties manifest themselves in a number of different ways.

Testing is a challenging problem, given the platforms in which their systems are embedded. It is not practical for them to rent or borrow an oil tanker, embed the system under development, and take it on a trial run every time a new function is added to the code. It is thus important for their simulation software to provide a realistic picture of what awaits the software once it is deployed. Still, simulators are only useful for finding a relatively small class of problems. Code reviews often uncover still others. Nevertheless, as with all large software projects, problems occur after deployment. Most of the ones uncovered in the past have not been so serious as to potentially cause a ship to become grounded, but risk is always present.

One of the most notable development problems, indeed the one that initiated this research, was the amount of perceived duplication of effort taking place from project to project. In spite of the diversity of applications under development, it is clear from the descriptions given earlier that there is still a good deal of functional overlap. A number of general concepts reappear from project to project, such as network message processing, input and output device management, and graphics display management. No good strategy has been found yet to standardize such processing and use the resulting code across different projects. It is strongly felt that such a strategy would go a long way in speeding up the development cycle, as well as improve reliability through the use of well-tested code which has been used in a number of different contexts.

The rest of this thesis documents a beginning attempt to address this problem. The questions of interest are:

- How does one characterize the commonality between these different navigation systems, each with a different purpose, and use this commonality to establish a workable set of software components?
- Once this has been achieved, how does one then use these components in developing new systems in these problem areas?

Given the diversity and complexity of this development environment, no single set of answers is likely to fully satisfy either question. Nevertheless, the next chapter describes an initial effort to address the first of these questions.

4. COMPONENT SEARCH

4.1 Software "Physiology"

In their Summary Report on the 1987 Minnowbrook Workshop on Software Reuse, Agresti and McGarry have this to say on the subject of searching for software components:

> *The major problems with taking advantage of the vast quantity of existing source code are determining what the code does and knowing how to retrieve it. Research is needed in both areas. Software archeologists are needed to sift through the large collections of existing code to identify potentially reusable components.* [AgM88]

The phrase "software archeologists" as used here is amusing but misleading. An archeologist is one who searches for and analyzes historical artifacts, and from these artifacts tries to draw inferences on a particular time and place. Archeologists rely strongly on serendipity to find these artifacts, and once they find them, do not attempt to use them themselves. What would we say about a distinguished Classical scholar who used a priceless ancient Greek vase as his personal coffee mug?

Perhaps a better analogy is that of "software physiologist", one who specializes in the comparative evaluation of a number of different systems from both a structural and functional point of view. The result of such an evaluation is an understanding of how particular classes of systems work, and where they share similarities. One can see just how appropriate the term "physiologist" is here by thinking about the role of the human physiologist. While all people are different, they share the same fundamental biological characteristics. The role of an anatomist is to study the *structure* of the human animal, and group the various organs into systems. The physiologist, on the other hand, is concerned not just with structure but also with *function* and *interaction* between systems. It is this knowledge that makes it possible to correct problems in the human body through medical and surgical techniques.

The role of the software physiologist is to compare a large number of systems from similar problem domains, identify recurring processing patterns, and analyze the contexts in which these patterns are used. In so doing, the physiologist learns how to make intelligent choices about which processing patterns are good candidates for reusable software components, and how future systems in a particular domain can be structured to take best advantage of these components.

Notice that the software physiologist's job is different from that of a *domain analyst.* The domain analyst works from a *metarequirements* viewpoint. He examines the space of problems that a class of systems are designed to solve, and looks for similarities. The physiologist, on the other hand, works from a source code viewpoint. He examines systems which have already been written, compares the different ways in which particular problems have been solved, and tries to draw the best features of these solutions together into a set of components which can be used by future systems.

4.2 An Exercise in Component Evaluation

The initial approach to this case study was to take on the task of the software physiologist. The goal of this first phase was three-fold:

- To derive an initial set of useful components to form the basis of a reusable software library.
- To gain insight into process of comparative source code analysis.
- To get ideas for tools to make studies of this type easier.

The systems under study were representative of those mentioned in Chapter 3. They were:

- *Adaptive Digital Gyropilot (ADG)*

  Shipboard system which provides automatic piloting control.
- *Raster Scan Radar (RASCAR)*

  Shipboard system which provides scanless radar display of local region surrounding ship. It allows simple navigation chart outlines to be superimposed on these radar images.
- *Voyage Management Station (VMS)*

  Shipboard system which provides navigation support. This includes waypoint planning, detailed display of digitized navigation charts, ownship position monitoring, and display of radar-input images of other ships in the area.
- *Vessel Traffic System (VTS)*

  Land-based system which provides support for harbor traffic monitoring. Allows harbor controllers to see radar-input images of ships in the entire harbor area, and where they are in relation to each other, to geographical features and hazards, and to designated restricted areas.

These four systems represent a substantial fraction of Sperry's business. Each one includes several thousand lines of C code (see Table 4.1), and involves real-time input and multitasking. The amount of time it takes to develop and deploy each one of these can be measured in several person-years. They are all subject to an on-going cycle of problem fixes and enhancements. Most important, the safety-critical aspects of these systems should be apparent. Serious flaws in any of these systems could potentially lead to collisions, groundings, or other calamities. The software must be correct.

SYSTEM	TARGET PLATFORM	LINES OF CODE	NUMBER OF MODULES
ADG	Embedded M68000	40K	40
RASCAR	Embedded M68000	60K	70
VMS	Intel 286/386 PCs	125K	150
VTS	Intel 286/386 PCs	60K	80

Table 4.1 Sizes of Systems Examined

4.3 Method

Evaluating these systems for reusable components consisted of two major activities:

- Discussions with the programmers responsible for these systems.
- Manually scanning through source code listings in search of recurring processing patterns.

This section will discuss these activities in detail.

4.3.1 Discussions With Programmers

When undertaking this type of study, an obvious first step is to find out what kinds of applications are being developed by the organization, and what kinds of strategies are being used to do this development.

An informal list of questions was prepared for the programmers responsible for the four systems already mentioned. This list is reproduced in Appendix A. There were several purposes to asking these questions:

- To get programmer perceptions on the current level of software reuse in the organization.

- To shed some light on how software was currently being developed in the organization, and how reuse might be integrated in with the overall development strategy.

- To familiarize us with the variety of systems being developed by the organization.

Discussions were kept informal; the questions on the list served merely to give focus to the conversation. Programmers were not asked to write answers to these questions down themselves; that was left to the interviewer. In spite of the informality, a number of insights were gained. For example, all programmers questioned agreed that an organized reuse strategy would probably improve development cycle time, and indicated that an informal, ''opportunistic'' reuse strategy was already being used. Among the kinds of functions already being shared among applications were:

- Multi-tasking support functions.

- General math and matrix functions.

- Map projection and navigation calculation functions.

A number of functional areas were suggested as good candidates for reuse, but reuse in those areas was currently not being done. This was particularly true in the realm of device drivers.

Along with these insights came an understanding of the Sperry Marine development environment, the products under development, and the issues facing its programmers On the whole, however, this method is ineffective as a rigorous information gathering tool. Some of the questions were too open-ended to lead to good, analyzable answers. Also, recording responses to these questions by hand as the interviewees answer is almost impossible to do accurately. Using a tape recorder, on the other hand, tends to make interview subjects uncomfortable.

In summary, the interview phase was a necessary first step in getting oriented, but did not lead to as many tangible results as the following phase.

4.3.2 Analysis of Source Code

The second major activity consisted of reading through source code in search of candidate software components. An identifier cross-referencer was available to assist with the search. A tree-diagram drawing program was also available, but did not go into sufficient detail to be truly useful. Although the *Insights* section of this chapter discusses the kinds of tools that would be useful for this activity, ultimately one must be prepared to read a large amount of code when performing this exercise. This is exactly what was done. Each of the four systems were scanned manually. Notes were taken on each one, describing system function and recording recurring patterns.

The systems were organized in a traditional procedural fashion. A single *main()* program initiated a number of concurrent processes which handled the primary functions of the system. All systems were structured in a procedural hierarchy, with each *.h* and *.c* module combination constituting a primary organizational unit. This made it fairly easy to understand individual parts of each system, although fully understanding the complex interactions between these parts was infeasible in the given period of time.

However, full system understanding was less important than gaining insights into how this process should be done, and whether it is a worthwhile activity to begin with. The emphasis here was on defining the role of software physiology by doing a comparative evaluation of a number of complex systems. As the next sections show, a number of tangible results arose from this study as well.

4.4 Processing Patterns and Pattern Diffusion

It became apparent early on that certain semantic patterns were repeated in various places throughout the code. As a simple, common example, some systems process data from a variety of different files in a variety of different formats. The underlying pattern, however, was always the same:

```
open file
check for open error
if no error, begin reading and processing data
close file
```

Other patterns were also common. Processing such as receiving user input from menus or screen icons tends to follow the same basic set of steps regardless of where in the code it is done. The same holds true for other processing such as drawing icons on the screen, and processing network messages.

The fact that programs tend to be composed of such semantic patterns has been the topic of much study. Soloway and Ehrlich note that experienced programmers think in terms of *plans*, or sets of conceptual patterns which help them organize programs in terms of what a particular set of statements *mean* when taken together, rather than focusing on the individual statements themselves [SoE84]. Rich and Waters carry the plan concept further in developing the idea of *cliches* in their Programmer's Apprentice project [RiW88]. They speak of formalizing these cliches and collecting them into libraries which can then be consulted when synthesizing new programs.

4.5 Results

The main result of this search was the identification of specific functional categories into which reusable components could be grouped. In many cases, reusable components themselves were identified. Table 4.2 lists and describes these functional categories.

CATEGORY	EXAMPLES
1. Data structure manipulation	Stack and queue management.
2. File management	File open, close, input, output, and scanning.
3. Graphics processing	Window management and shape drawing.
4. General math	Range checking and matrix manipulation.
5. I/O management	Peripheral control and device drivers.
6. List processing	Sorting and searching algorithms.
7. Navigation calculation	Projection transformations and dead reckoning.
8. Network management	Message parsing and checksum calculation.
9. String conversion	Numeric to ASCII transformation.
10. String manipulation	String parsing and substring determination.
11. System management	Multitasking control and memory management.
12. Time Conversion	Convert Greenwich Mean Time to local time.
13. Unit conversion	Cartesian to polar coordinate transformation.

Table 4.2 - Functional Categories And Examples

In all, about 140 specific software artifacts were identified as good candidates for yielding reusable components. Each artifact is expected to yield several reusable components. These artifacts ranged in size from single functions to entire modules. They were organized into a list arranged according to the above categories. This list is reproduced in Appendix B. Included in this list is a section on functional categories without any enumeration of specific components. This reflects the fact that one can often find a problem area which appears to lend itself to code reuse, but cannot locate any reusable components in that area. The area of *device drivers* serves as a good example of this. Numerous low-level drivers have been written for such devices as clocks and communication ports. Yet, it was difficult to find any code that abstracted the higher-level interface characteristics shared by such devices from the lower-level implementation details. Having these interface characteristics available could prove very valuable to programmers creating drivers for new devices. Even though the implementation details would be different, at least the consistency of the driver's interface across a set of applications would give different systems a more uniform feel.

Another interesting feature of this list is the similarity between a number of different functions. The Networking category includes a few *checksum* calculation routines. The String Manipulation category includes a few implementations of a *breakline* program. These are simple examples, and are not intended to point out duplication of effort *per se*, but rather another intriguing phenomenon: *this perceived functional duplication often occurs across problem domain boundaries. Breakline* is implemented in both ADG and VMS, which from a functional standpoint are as dissimilar as one could imagine. Functions which occur often enough in a number of different problem domains often become built-in language functions, freeing the application programmer from further implementation. The *strcpy* function of C is an elementary example of this. One wonders if an organized, large scale scavenging effort of this sort across a wide range of problem areas would yield an entirely new spectrum of potential built-in functions for C, Pascal, and other common languages.

4.6 Insights

On the surface, manually searching through several thousand lines of code written by others might appear to be the most boring and tedious undertaking imaginable. However, a number of valuable insights arose from this study; in many ways, these insights are more significant than the actual code that was yielded. These insights shed a great deal of light on the nature of reusing complex software not originally intended for reuse. Of particular interest are the following:

- The relationship between a component's potential reusability and its use of multitasking semantics.
- The relationship between a component's potential reusability and its use of exception handling semantics.
- The relationship between the component scavenging activity and the software maintenance activity.
- The role of documentation in improving the clarity of a piece of code.
- The relative importance of explicitly-stated software design decisions.
- The partial refutation of a popular claim about the ability to reuse code not intended for reuse.

Each of these items will be discussed in turn.

4.6.1 Reusability and Multitasking

All of the code examined performed multitasking, using the MTX tools. The way mutual exclusion is achieved using these tools is by identifying critical sections and bracketing them by `lock` and `unlock` statements. The implications of this on code reuse are potentially enormous. For one thing, code which constitutes a critical section in one system might not in another. Should components stored in a reuse library include these multitasking statements just in case, or should they be omitted and then added at the reuser's discretion? Also, this method of multitasking is just one of a number of ways. Are some multitasking constructs, such as semaphores, message passing, and monitors, better suited for code reuse than other methods, such as this one?

4.6.2 Reusability and Exception Handling

A significant amount of code is devoted to handling exceptions and undesired events. The C language has no convenient way for the programmer to specify how an exception should be processed when it occurs; he must provide checks for it at each place it is likely to occur. This sometimes makes for awkward, bulky code with a number of `if-then-else`'s solely devoted to this kind of processing. This constitutes a significant aspect of the code's behavior, and has a major impact on the ease with which code can be understood by the reader.

Other languages, such as Ada, Clu, and even PL/I, have more sophisticated exception-handling facilities. It would be worthwhile to study the various types of exception handling mechanisms which have been implemented across a broad language spectrum, and determine the impact each one has on the ability to reuse large sections of code.

4.6.3 Relationship Between Scavenging and Maintenance

In many ways, the component analysis activity is identical to what is traditionally called the software maintenance activity. While the goals of the two processes differ, the cognitive processes are the same. Both involve examining and understanding the inter-relationships between parts of large systems written by other people. As such, a set of software tools designed to assist in the maintenance process will almost certainly prove useful in the component analysis process.

Some examples of such tools are those which display structural information about a system in a graphical, easy-to-read fashion. Cleveland discusses an experimental tool of this nature under development at IBM [Cle89]. The ability to easily determine the defined attributes of an identifier is also important. Identifiers are often defined in one section of code and used in various places all over the system. It would be extremely convenient to have the ability to highlight an identifier using a mouse, and have a window appear describing the attributes of the identifier, and perhaps a list of where else in the code the identifier is used.

From a strictly reuse-oriented perspective, an easy to create and potentially very useful tool would be one which would provide a listing of all programmer-defined functions, and a percentage showing the relative frequency with which they are called. This would let the investigator quickly identify potential reuse hotspots.

4.6.4 Role of Documentation

Surprisingly, the role of formal documentation is rather limited in this kind of study. The type of documentation that is most helpful is a few sentences describing each module, a brief overview explaining how the modules in the system fit together, and a few sentences describing each function of each module. Documentation in excess of this tends to be hard to maintain and probably will not be used anyway. Detailed line comments are almost always unnecessary except when the line in question is particularly counterintuitive. Consistent coding style and descriptive variable names is usually more important than exhaustive documentation in helping others understand what a particular piece of code does.

The Soloway and Ehrlich study of *plans* provides further evidence of this phenomenon [SoE84]. They describe *rules of discourse*, or unwritten standards that experienced programmers subconsciously adhere to in order to produce clear, comprehensible code. These rules of discourse include such truisms as *Variable names should reflect function* and *If there is a test for a condition, then the condition must have the potential of being true*. They describe a set of experiments demonstrating that the ability of experienced programmers to understand even simple pieces of code plummets when these ''rules'' are violated. The implications of this on the scavenging activity are enormous, given the volumes of code that have to be read and understood by the scavenger.

4.6.5 Relative Importance of Design Decisions

A good deal of recent research on software reuse has been devoted to capturing software design decisions to assist the potential reuser in figuring out whether he wants to use a particular component. However, there is a limit to how important this information is in this type of activity. When designing a module, the designer makes hundreds, perhaps even thousands of design decisions. Most of these decisions are made from experience without conscious thought. And most of them are not germane to the module's level of reusability. The decisions which *are* important are those which affect how the component is used in its native environment. Specifically, the potential reuser must understand

- What shared data structures, if any, are being accessed by this component?
- What other support functions are being used by this component?
- Is there a proper sequence that has to be followed in order to use this component? Does data need to be explicitly initialized? Does the system need to be in a particular state?

If the original programmer followed good data abstraction and information-hiding principles, then the first item should not be too problematic. Of course, the older the code,

the less likely that these practices were followed. In this case, it is very helpful indeed to have access to a description of the relevant design decisions, so the reuser can tell how much extraneous code has to be included with the code he really wants. This is true, too, of the second two items.

4.6.6 Contradiction To Previous Work

In a brief paper from the 1987 Minnowbrook Conference, Carle describes a similar scavenging effort in the domain of missile software [Car87]. He draws a number of conclusions about the nature of this activity, one of which appears to contradict our findings. Specifically, he makes the statement *Old Software Is Not Reusable*. This statement is too general and defeatist in nature to be taken literally. Clearly, old software *is* reusable, or at least has strong reuse potential given a re-engineering effort. Even if a particular component cannot be reused verbatim in a new context, the analysis embodied in the component often provides a foundation on which to remodel it somewhat to fit the new context. What is needed is a structured way of carrying out this remodeling effort across a broad spectrum of components.

4.7 Summary

An assortment of program understanding tools would have been helpful in this *software physiology* exercise. Given the volume of code available, there are almost certainly more components available than those identified. However, performing this exercise manually provided good insights into what kinds of tools in particular would help, and how they could be used. As useful as this insight and the other results and insights might be, the scavenging activity should be regarded primarily as an orientation activity rather than the main point of the case study. Nevertheless, this activity laid the foundation for the work described in the next several chapters.

5. ELECTRONIC CHARTS: A DOMAIN ANALYSIS

5.1 Chapter Overview

The previous chapter describes the initial *code scavenging* activity performed on a number of Sperry Marine's commercial software systems. During this process, it became apparent that a large amount of programming was being devoted to the management and graphical display of electronically digitized navigation charts. Sperry Marine has deployed a number of systems which use electronic charts, and several more systems are under development. For this reason, they strongly feel that a set of software components geared toward charts management and easy to integrate into a variety of systems would greatly improve their productivity.

This chapter gives a detailed explanation of the first step in creating those components, namely, the *domain analysis*. According to Prieto-Diaz, a domain analysis is *a process where information used in developing software systems is identified, captured, structured, and organized for further reuse* [Pri90]. A domain analysis provides the foundation of an organized reuse strategy. It is here that the general characteristics of an entire problem area, rather than a single set of system requirements, are made explicit. Given this knowledge, the component developer can create a set of software parts that can be used for a spectrum of related problems, rather than a single problem.

There are many ways to present a domain analysis. Prieto-Diaz reports that common domain notation schemes borrow from the fields of artificial intelligence and database methods. Entity-relationship diagrams, semantic nets, and sets of rules are among these schemes. For the sake of clarity and readability, the domain analysis in this chapter is presented in a narrative style. The importance of this domain is discussed, followed by a description of the domain's salient features. A brief summary section provides a comparison of the requirements of the primary user communities of this domain.

5.2 Electronic Charts

Navigation charts are detailed nautical maps published by various hydrographic services around the world, notably the National Oceanographic and Atmospheric Administration (NOAA), and the Norges SjøKartVerk (NSKV), or Norwegian Hydrographic Service. Their purpose is to provide shipboard navigators with visual representations of coastlines, along with as much information as possible for ships to safely negotiate their way along these coastlines. This information includes such features as buoy locations, wrecks and other hazards, shoals, and navigation channels. Figure 5.1 shows part of a NOAA-prepared chart of New York Harbor.

For the past few years, digitized versions of such charts have been used in both land- and sea-based commercial voyage control systems. In land-based systems, electronic charts, overlaid with information based on radar input, allow harbor managers to get a graphical view of which ships are in the area, and where they are in relation to each other and to major geographic features. Such systems are the nautical equivalent of air-traffic control systems. In sea-based systems, charts allow the navigator to see where his own ship is in relation to other ships, to major land masses, to navigation channels, and to various landmarks and hazards. They also allow a view of where the

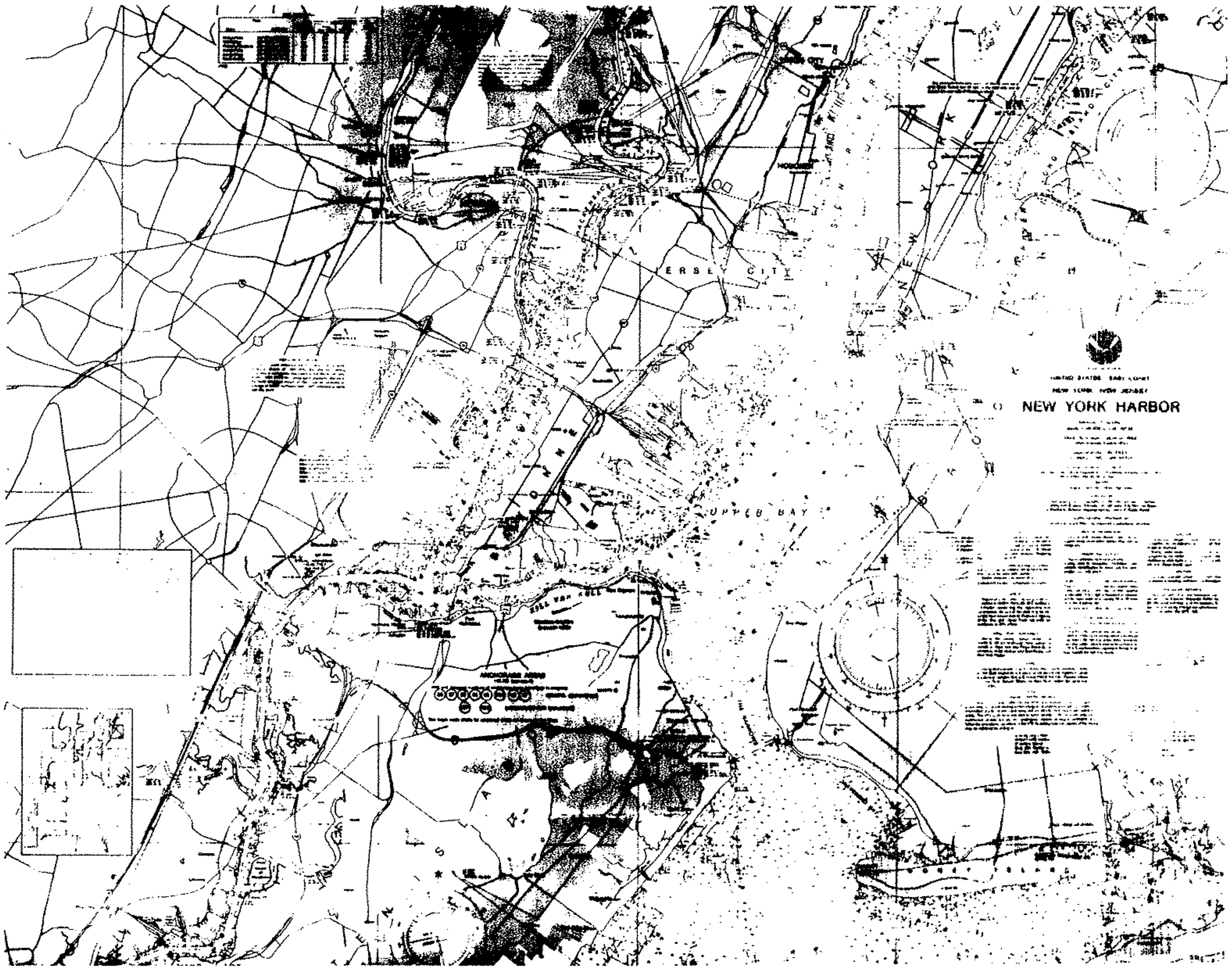


Figure 5.1 NOAA-Prepared Chart of New York Harbor

ship is in relation to a predetermined voyage plan.

5.2.1 Electronic Charts as a Safety Critical Application

The potential uses for electronic charting systems are vast. Users of such systems can derive information from them that inflexible paper charts could never provide. Zoomed and offset views of coastlines, realistic simulations of light house flash rates, and automatic update of hazard information by radio or satellite are just a few possibilities. However, as the possibilities grow, so do the risks.

Obloy discusses the legal ramifications of improper and negligent cartographic techniques in [Obl89]. He presents a number of anecdotes of ships damaged or even stranded because of poorly noted coastal hazards or shoals. With respect to electronic charts, he notes problems which have arisen because of poor software design. In one notable case, a Dutch fishing vessel ran aground because the navigator had entered a planned course into the system and then tracked to it exactly. Unfortunately, there were shallows along the way. At no time did the system issue any warnings that the projected voyage plan would lead to an area of such known and obvious hazards.

This is an example of a level of risk complexity which is absent from conventional charts. Whereas with conventional charts, a cartographic organization can be held liable for misleading or incomplete information, electronic charts introduce the software developer into the picture. Not only is there risk in the chart information being displayed, there is also risk in how overlay information is presented. If overlay information includes superimposed images of moving radar targets, and software problems cause the relative positioning of these targets to be even slightly off, the results could be disastrous.

5.2.2 Problems with Electronic Charts Management

Organizing systems to make use of electronic charting capabilities is a difficult problem. A significant amount of code in current systems is devoted to various aspects of chart processing, such as chart displaying, plotting, digitizing, and overlaying with various objects and symbols. Such operations tend to be scattered throughout the system, rather than isolated in a particular module or set of modules. This makes it hard to borrow code from old systems when creating newer ones. It also leads to complications when introducing enhancements and fixes to these older systems.

Also, different applications use charts for different purposes, as was pointed out earlier. The user and input requirements of such systems vary immensely depending on whether they are land- or sea-based. A land-based system, for example, does not have to be concerned about voyage planning or track history plotting; a sea-based system does. And besides these differences in functional requirements, there is great diversity in the hardware environments on which applications are built, as well as in the application design strategies employed. All these factors combine to make the reuse of chart processing code a difficult technical problem.

A further complication is the great variety of standards used to describe the information that appears on charts. Every hydrographic organization has its own set of codes to note the various kinds of buoys, land features, and hazards. In order to be completely flexible, a system has to be prepared to cope with all of these standards, or have some way of converting these different standards into one consistent format.

As time goes on, the manner in which charts are digitized will become an additional factor to consider. Currently, most charts are digitized into collections of vectors, as the next section explains. In order to show more detail, charts are likely to be represented as bitmaps in future systems. This will happen as display equipment becomes less expensive and larger memories become more common. Systems under development today should be prepared to cope with modifications that might lead to dynamically switching back and forth between the two display methods.

5.2.3 Related Work

Work is currently under way by several organizations to develop the potential for using electronic charts. The NSKV is one of the leaders in this effort, and has sponsored at least one major project to demonstrate the potential for electronic charts usage by the commercial sector. Stene describes this ''North Sea Project'', undertaken in 1987 [Ste87]. In the same place he describes efforts to create an electronic charts database, which would consist of digitized representations of all the coastlines in the world, and would be stored on seven or eight compact disks.

The U.S. Department of Defense has been involved in electronic charts technology for several years. The CIA's World Data Bank-II (WDB II) served as a significant tool for several years. Its lack of resolution and detail, however, limited its usefulness in navigation [Loh89]. More recently, the U.S. Defense Mapping Agency has been creating a new version of WDB II known as the World Vector Shoreline Database, or WVS, which offers higher resolution and more details for military navigation.

5.3 Aspects of Electronic Chart Management

This section gives a detailed description of the issues involved with the management of electronic navigation charts. While the foregoing discussion highlights the important differences between vector charts and raster charts, the immediate area of interest for this project was vector charts. Thus, all subsequent comments in this section refer specifically to vector charts.

The specific areas discussed in this section are:

- The way charts are converted from paper to digital format.
- The way the information conveyed by charts, such as geographical features, sounding information, and human-made objects, are organized internally by an electronic charts management system.
- The various types of manipulations a user should be able to perform on a displayed chart image.
- The various types of informational overlays a user should be able to use in conjunction with a displayed chart image.
- The way a chart system can be used as a navigation information management tool, as well as a chart display vehicle.

5.3.1 Digitizing Charts

Most hydrographic organizations publish and distribute their charts on large paper rectangles of roughly three by four feet. In order to transform these charts from paper format to machine-readable format, it is necessary to go through a *digitization* step. Vector charts are digitized by using a light pen and a digitizer table. The person doing the digitizing uses the light pen to trace along the edges of shoals, landmasses, and other features, clicking the appropriate pen button whenever a vertex is encountered. Each vertex is translated into an (x, y) coordinate pair on a Cartesian plane, where point (0, 0) is considered to be the center of the chart. It is these (x, y) pairs that lend the name *vector chart.* These vector pairs are stored sequentially in files, along with chart projection and calibration information.

5.3.2 Chart Features and Symbols

There are three types of chart entities: *line features, point features,* and *text features.*

- *Line features* are composed of line segments which, taken together, constitute closed or open polygons. Line features are used to represent irregular shapes such as land masses or navigation channels. When a line feature constitutes a closed polygon, it is often referred to as an *area feature.*

- *Point features* are entities which can be represented internally as a single (x, y) coordinate pair. These are used to represent classes of objects whose members are all identical except for their position. Symbols to represent a specific type of buoy, for example, would all have the same size, shape, and color, but each instance of this symbol would show up on the screen in a different location.

- *Text features* are simply strings of text which can be put on a chart to show sounding depths, label legal boundaries, or identify radar targets.

Each feature has a number of different display attributes associated with it, such as *color, fill pattern, line size*, and *line style*. These attributes are all variable from feature to feature, including features of the same type. Land masses, for example, all have different shapes and sizes. However, they might all be displayed using the same color and fill pattern in order to avoid confusion. Such attributes are typically defined to a feature while it is being digitized, although chart applications often have the ability to change them according to the needs of the user. A user displaying charts on a monochrome screen, for example, might have to employ a variety of different fill patterns which he would not have to use if he had access to color.

5.3.3 Chart Image Manipulation

It is often necessary to cause the displayed image of chart to move in various different ways. The more common of these movements are *zooming* and *offsetting*. Zooming is used to focus on a particular section of the chart, or to get a "bird's eye view" of a larger area than is currently displayed. Offsetting describes recentering the displayed chart around a new point.

Associated with zooming is the issue of *decluttering*. When the view is zoomed in close, a lot of detailed information about a local area can be shown. This could include all of the local hazards, buoys, and sounding labels, to name a few. As one zooms out (or more properly, *pans*), a larger portion of the chart is displayed in a constant screen area. The more information that gets displayed, the more cluttered the chart appears, making it hard to read. Decluttering refers to the ability of the user to ''turn off'' the display of certain features to make the chart more readable. Chart systems should allow the user to explicitly select which features he wants to turn off, or automatically turn specific features off once a certain zoom factor is reached.

5.3.4 Overlay Information

As mentioned earlier, in order to take full advantage of the power of electronic charts, the user has to be able to overlay the chart with additional information. This type of information includes

- *Voyage plans* - A tracing along a coastline showing planned waypoints.
- *Track history* - A trail showing where a ship has been during a given time period.
- *Range rings* - A set of concentric circles centered on a particular object or feature. The distance between each ring can represent some number of nautical miles.
- *Grids* - Horizontal and vertical lines marking latitude and longitude positions.
- *Radar targets* - Moving symbols representing the positions of other ships in the area.
- *Ownship symbol* - Moving symbol representing the position of the navigator's own ship.

- *Predicted course* - Projection of ship's track over a given period of time.
- *Restricted areas* - Harbors often have certain areas designated as off-limits to commercial shipping. These are often designated by large circles.

Display equipment must have the ability to draw and erase these overlays without disrupting the underlying chart. This is typically implemented using *color planes* or by drawing the overlays in *XOR mode.*

The fact that relative motion is often involved in overlay display provides an intriguing set of problems. In sea-based systems, target symbols and the ownship symbol are often displayed at the same time. While target symbols are moving relative to each other, they are also moving relative to the ownship symbol. Because of this lack of a fixed reference point, keeping this set of interrelationships organized is a non-trivial task. In land-based systems, the problem is somewhat simplified because there is a constant, fixed reference point. However, moving symbols often have other symbols associated with them. A specific target, for example, might constitute the center point of a set of range rings. As the target moves, the range rings have to move with it. These overlay idiosyncrasies provide an element of complexity absent from a simple graphics processing system.

5.3.5 Calculations and Information Processing

Another useful feature of electronic charts is the fact that they free the navigator from lengthy and complex distance calculations. There is often a need to calculate positions and distances based on information available from a chart. The nature of these calculations will differ depending on the projection system used in creating the chart, the most common being mercator and transverse mercator. The underlying system can now perform such jobs automatically, reducing the chance of human error.

Other possibilities abound. Some of the newer systems are capable of recording and playing back voyage planning, tracking, and other activities. In effect, this is analogous to the role of the "black box" flight recorder on airplanes. This capability can prove extremely valuable when dealing with accident investigations and litigation [Obl89].

5.4 Domain Summary

Table 5.1 provides an encapsulated view of the similarities and differences between the two primary types of electronic navigation chart systems. This view is given from a user capability perspective; that is, it addresses the issue of what functions a user needs to perform on a chart depending on whether he is based on land or sea. As can be seen, even though the required functionality between the two system types often overlaps, their usage sometimes varies significantly because of the difference in perspective.

The issue of feature highlighting serves as an example of this. In a land-based system, the harbor controller might want to highlight ship symbols which are entering a hazardous situation. Ships which are passing too close to each other or to shoals should be highlighted so the controller can issue a warning. In a sea-based system, the same function is necessary, except the pilot of the ship using such a system is concerned about highlighting when circumstances directly affect his ship, not other ships in the area. The underlying rules governing what gets highlighted, and under what circumstances, are very different.

This difference in perspective between the two system types manifests itself throughout the domain.

LAND-BASED SYSTEMS	SEA-BASED SYSTEMS
Radar sites are stationary.	Radar moves with the ship.
Number of charts needed is small. Just enough to provide coverage of the harbor area.	Number of charts needed is large. The ship could be anywhere on Earth.
Number of chart overlays needed is fairly small. Particular needs are: - Current harbor traffic. - Restricted areas. - Moving target safety zones.	Number of chart overlays needed is fairly large. Particular needs are: - Ownship track. - Planned waypoints. - Moving range rings. - Lat/Lon grid.
Status queries, such as distance between two targets, are based on a fixed reference point.	Status queries are based on the ownship as a reference point.
Might need several windows open at same time, each providing a different view of the harbor area.	Multiple views are typically not needed.
Requires the ability to display target position and history vectors.	Must be able to display target and ownship history vectors.
Requires log and playback facility to record target motion.	Requires log and playback facility to record ownship motion.
Requires some way of highlighting particular objects, such as by color, or flashing.	Same. This ability could be used to warn of impending hazards, or as a reminder to perform some action.
Needs some way to select an object for specific information about it.	Same. Such information might include type of buoy or hazard.
Needs zoom and recentering ability.	Same. Often requires ability to automatically recenter chart around stationary ownship symbol, rather than moving ownship on stationary chart.
Needs decluttering ability.	Same. This is especially useful when zooming out from the chart image.

Table 5.1 Land vs. Sea-Based Charting Systems

5.5 Chapter Summary

This chapter has covered the prominent features of the electronic navigation charting domain, why it is important, and how it differs from the more general domain of simple graphics processing. Among the more important aspects of the domain are:

- Electronic navigation charts constitute a safety-critical application area. People rely on the software which processes and displays charts to be correct. If the software is not correct, the result can be loss of human life and property.

- Charts are composed of a number of different features. Each feature is associated with an identifiable standard code, but there are many different standards. A comprehensive chart system must be prepared to process charts created according to a number of different standards.

- Electronic chart systems do not simply display charts, they also move charts around on the display screen relative to fixed points, and control the motion of various overlays on the display. Furthermore, they assist the user in making distance and tracking calculations. So in order to be truly effective, a charting system must constitute a cohesive, integrated nautical management tool for use on both land and sea.

6. ELECTRONIC CHARTS SUBSYSTEM PROTOTYPE

6.1 Overview

The previous chapter provides a narrative description of the domain of electronic navigation charts management. This domain analysis proved to be a crucial stepping-stone to the next phase of this case study. Recall that the focus of this study is on the introduction of a component-based software reuse strategy to an industrial organization. The emphasis is on practice, rather than theory. Taking a domain analysis from a paper description to a working set of software components is a large cognitive leap, and has received little attention in the literature. Thus, the next logical phase of the project was to perform an exercise in creating a prototype set of components from the domain analysis. The idea was to gain an understanding of the issues and problems involved in such an activity. Ultimately, the end-product of this exercise will be a set of reusable components which, taken together, will form a reconfigurable subsystem that could be used in a variety of different charting applications. This chapter describes the nature of the prototyping strategy chosen for these components, along with specific goals of the prototype from an application programmer standpoint.

6.2 Prototyping as an Implementation Strategy

Basili and Turner introduced the concept of *iterative enhancement* in [BaT75]. Iterative enhancement is described as an appropriate implementation strategy to use when creating reasonably large applications where the impact of major design decisions is unclear. Traditional techniques, particularly step-wise refinement, require that the implementor be fully aware of the ramifications of making any given design decision at any given point of the system's creation. This is usually difficult to do in practice, even in familiar problem areas. Iterative enhancement is a technique whereby the implementor creates an entire working model of a small subset of the system, analyzes how well the design decisions made up to that point support the goals of the system, makes corrections as appropriate, and progressively adds functionality to the system. This way, faulty design decisions can be caught and corrected as they become apparent early on, rather than lying dormant until a later stage in the project, when correcting them could be extremely expensive.

This development strategy lends itself to creating components intended for an entire problem domain, as well as for creating specific systems. In this case, there are actually two problems that the iterative enhancement method addresses:

- Designing a specific charting application is complicated enough in and of itself. The graphics requirements, point transformations, real-time requirements, and other complexities mentioned earlier present a number of opportunities for making major design mistakes.
- Some functions are common to all charting applications. Iterative enhancement allows the implementor to create these common functions first, and use them as a platform for adding additional functions for specific applications.

6.3 Selecting A Reuse-Based Architecture

It is desirable for components to be as independent from each other as possible. The looser the coupling, the better the chance of being able to use the components in a variety of different settings. Nevertheless, there has to be an underlying architectural structure that governs how they interact with each other. The components, after all, are intended to be used as a whole, not individually. A number of strategies were considered as potentially good architectures. Chapter 2 describes three of these; *hierarchical*, *heterarchical*, and *object-oriented*. The object-oriented strategy was the one chosen. This section explains the reasons behind that decision.

As noted earlier, object-oriented development methods have received much attention in recent years, but the amount of documented practical experience from a reuse perspective remains sparse. It is apparent the object-oriented design (OOD) provides a natural way of thinking about many software problems. What is less apparent is how well OOD fairs as a reuse strategy when applied to a substantial, complicated problem domain.

Language issues, too, play an integral part in the OOD strategy. One could hardly be expected to implement an object-oriented design in a language that did not support the *class* concept, where a class is defined as an encapsulation of data elements along with a collection of associated functions, serving as a template for specific instances of objects. As discussed in Chapter 2, the concept of *inheritance* should also be supported by the chosen language. For this project, C++ was chosen as the language vehicle. C++ supports both of these facilities, along with a facility for overloading operators and functions, allowing for reasonably generic code. The fact that Sperry Marine programming staff was accustomed to doing their development work in C also gave impetus to this decision. It was felt that the prototype would gain wider acceptance by those intended as its users if it were written in a syntactically familiar language.

So from both a design and a language viewpoint, a study of object-orientation entails a number of reusability questions:

- Is the *class* a reasonable unit of reuse, as opposed to more traditional units such as C functions or Ada packages?
- How easy is it to create an initial set of components with which to construct a set of related systems using object-oriented methods?
- Once a basic set of components has been assembled for the given domain, how easy is it to tailor these components for specific applications under the OOD paradigm?

These and related questions are addressed in more detail in Chapter 8.

6.4 Other Prototype Benefits

Apart from the reuse-based *research* questions, the intent is for the charting components to have practical benefits as well. Once completed, the components are intended for deployment in real applications. This should ease development cycle problems in two ways:

- By providing a set of standard charting services that can be used by all charting applications.
- By helping ease the maintenance of existing applications by encapsulating all chart-related code in a single set of components.

These points are considered separately.

6.4.1 Provide Standard Services to Applications

Recalling the domain analysis, applications using charts require functions in two broad categories:

- Provide various views of a chart, including zoomed views, offset views, and various projections.
- Control the display of various overlays, both moving and stationary.

With these categories in mind, there are a number of potential services which the subsystem ultimately could offer:

- Set base display characteristics

  This refers to how the chart looks when it is first displayed, and includes object color, fill pattern, magnification, center, and orientation. Much of this information is defined when the chart is digitized, and resides in the same file as the rest of the chart. However, some of this information can be changed dynamically and saved in separate configuration files.

- Zoom in or pan out relative to base magnification

- Zoom in or pan out relative to current magnification

- Offset chart relative to base center

- Offset chart relative to current center

- Select a group of objects matching a set of criteria

  For example, the programmer might want to highlight all buoys within 100 yards of the ownship. This is an example of where the subsystem could act not just as a display application, but also as a sort of database with predicate search capabilities..

- Select a specific object or symbol from the chart.

- Delete an object or symbol from the chart.

  This capability is important both when editing a chart, or when decluttering. As mentioned earlier, decluttering is the process of temporarily turning off the display of extraneous information. The chart user might want to do this when he has panned out a certain distance, and chart objects are becoming hard to distinguish. Another example of where this could be useful is daytime sailing versus nighttime. During the day, information on local lights is not very useful, and should not take up chart space.

- Add a new object or symbol to the chart.

  The Coast Guard issues periodic *Notices to Mariners* about new hazards or other shoreline changes. It is important to be able to add such information quickly and easily to existing charts.

6.4.2 Protect Existing Applications From Change

It is important to make such a subsystem as amenable to change as possible. In this particular case, changes could be introduced for many reasons:

- Chart file formatting standards might change.

- New kinds of chart objects might be introduced.
- New operations might be introduced to existing kinds of objects.
- Objects, operations, and facilities might *not* be needed by an application, so the programmer should be free not to include them.

To this end, the prototype design should enforce strict information-hiding. In other words, no part of the system should need to know the internal details of any other part of the system. This is true on two levels.

On the application program level, it should allow the application programmer the ability to manipulate chart objects in a useful way without worrying about how these manipulations are carried out. For example, if the application needs to update the display position of a target ship, the message sent to the subsystem should be to the effect of ''Target 103 is now at <some real-world coordinate position>''. It should then be up to the subsystem to translate this real-world position into a chart position and make the appropriate update. The application is thus blind to such details as how chart coordinates are represented, how a target symbol is represented, and how to move a target symbol.

On the subsystem level, different parts of the subsystem should work as independently as possible. Knowledge about external file formats, display device characteristics, and internal data representations should be isolated into specific components.

6.5 Summary

Creating a prototyped set of components devoted to the domain of electronic navigation charts management has a number of benefits:

- It addresses a growing need for a general set of chart-specific functions which can be used in a number of different applications.

- It provides an opportunity for examining the effectiveness of object-oriented development methods as a software reuse strategy.

- It serves as a good exercise to evaluate the differences between designing parts to fit a specific set of requirements versus designing parts to fit an entire problem domain.

The next chapter goes into detail on the design of the components themselves.

7. SUBSYSTEM DESIGN

7.1 Chapter Overview

While the previous chapter discusses the impetus for creating a prototype set of components devoted to the domain of electronic charts processing, this chapter discusses the details of designing those components. Discussed are the major abstractions of the subsystem, and the components which comprise those abstractions. Some of the major design issues, such as dealing with concurrency and coordinate transformations, are also considered in detail.

Appendix C, which serves as the subsystem's *class catalog*, is referenced throughout this chapter. The catalog provides a brief description of every class used in the system. It also describes the *uses* relationship between classes, and when appropriate, the inheritance structure. The reader can periodically reference this catalog for an implementation-level view of higher-level constructs under discussion.

7.2 Primary Abstractions

The subsystem architecture is based on five separate abstractions:

- *Chart ''database''* - This object is created at run-time, and acts as a collection of slots for feature instances. It also contains chart projection information. It is used as a central repository which other objects can access in order to obtain instances of features, or information about feature attributes, such as color or fill pattern. Note that this is not a ''chart database'' in the sense that it contains information on a number of different charts; rather, it contains all the information needed to display a single chart. An application which displays several charts at a time would need to have as many of these "databases" allocated as it has charts to display.

- *Chart file* - Hides the specifics of processing files in different formats. This object consists of routines tailored to specific formats, and works by searching through the file until it finds the points and attributes describing a feature defined to the subsystem. It then packages this information into a *Feature object* (described later on) and passes it back to the subsystem for inclusion in the chart database.

- *Chart image* - Acts as the intermediary between the chart database and the window in which the chart is displayed. The image object contains a list of chart features to be displayed on the window. The programmer can thus use the image object to turn the display of specific feature types on or off as needed. The image object is also where various "special effects" are implemented, such as zoom and offset.

- *Window* - Contains window position and size information, and methods on how to display various geometric shapes. Windows themselves are device-independent, but when one is created, it needs to be tied to a specific device of the programmer's choosing.

- *Device* - Hides the implementation details of various useful device-level functions, such as window creation, shape drawing, and color display.

7.3 Points - The Fundamental Building Blocks

The fundamental "unit of currency" in the subsystem is the point. Points are used to define features, window positions, and window sizes. They consist of an (x, y) coordinate pair, plus a number of arithmetic operations. Appendix C describes the point class in detail.

In order to manage groups of points, a higher-level construct known as a *point list* was devised. A point list provides a convenient way of bundling groups of related points into a single construct. This is necessary when dealing with points that define features with odd shapes, such as land masses. Access to point lists is provided by *point list iterators*, discussed later. Appendix C also goes into detail on point lists.

7.4 Feature Management

One of the major design concerns was the issue of how to manage and display a variety of different types of chart features, and still keep alive the one underlying, unifying abstraction of "feature". For example, the chart database object should know that it is responsible for storing lists of features, but it should not have to know the details and idiosyncrasies of each type. Such details as whether a feature is represented by one or several points, or whether a feature includes text, should make no difference to how the database stores or retrieves it.

On the other hand, these details matter a great deal when it comes time to display a feature. Features represented as a single point need to be "expanded" into particular shapes before they are displayed. Hazard symbols, for example, are stored as single points in order to save space, but when they are displayed, they might appear as small triangles. So somewhere in the system there needs to be some mechanism that knows that features called "hazards" are actually triangles.

This dual view of features is achieved by using subtyping. Each feature is considered to be a unique type, as established by using C++'s *class* mechanism. Since the main difference between features is how they appear on the screen, what distinguishes each feature type is the fact that they each have different *Draw* methods. The *Draw* routine for each feature type performs whatever expansions are necessary in order for the feature to appear on the screen in the right shape. However, there are cases where individual feature types do not need to be distinguished by appearance. In these cases, it is desirable to keep the code generic by processing data of type *Feature*, rather than having to use a large case statement to handle every possible feature type. C++ makes this possible through its inheritance mechanism. One can set up a base class of type *Feature*, and derive from that subtypes of each particular type of feature. So in sections of code where one is interested in processing feature objects, but does not care if they are land masses or hazards or some other type, he can just specify Feature as the input type. C++ allows a subclass to appear anywhere that its base class is expected.

7.5 Accessing Lists of Items

A processing pattern that arises again and again throughout the subsystem is that of list processing. Points are stored in lists, features are stored in lists maintained by the chart database, and chart files can be regarded as lists searched by the file object. A consistent way was needed to process these lists in a manner transparent to the functions using them. At the same time, some way was needed to allow multiple tasks to access the same list at the same time without interfering with each other. The answer was found by using *iterators*.

An iterator is a control abstraction which hides the details of how particular list structures are implemented. By accessing list items via an iterator, a function does not have to concern itself with whether the list is implemented as an array, a linked list, a tree, or some other structure. All it needs to know is which operations on the iterator will yield the next element of interest. So when a function needs to navigate its way through a list, it first allocates an iterator on that list and uses the iterator as an interface. Since each function has its own personal iterator to work with, and since each iterator maintains knowledge of where it is in the list, an arbitrary number of iterators can be active at the same time on the same list without fear of collisions.

C++ has features which make the creation and use of iterators quite simple. Stroustrup discusses iterator creation and usage in [Str87].

7.6 Coordinate Transformation Strategies

There are three different types of coordinate systems to deal with in this subsystem:

- *Real-world coordinates*, that is, latitude and longitude.
- *Chart coordinates*, typically given in inches or centimeters.
- *Window coordinates*, given in pixels.

At any given time, transformations may need to be done between any two of these systems. How the subsystem is set up to deal with these transformations has a major impact on its usefulness. This proved to be the most difficult design issue in the entire prototype. A few strategies were implemented.

One strategy deals with transformations from chart coordinates to latitude/longitude or window coordinates. These transformations are needed when the chart object is accessed directly. The complication here is that several tasks might be accessing the chart database at one time, and different tasks might need to take different views of the points comprising the chart. A display task, for example, would find it convenient to have the illusion that chart points are stored as window coordinates. Other tasks might find it more convenient to regard points as real-world coordinates. What remains consistent, though, is the fact that any task accessing the points comprising a feature will do so through the chart database's *GetFeature* function. *GetFeature* will respond to the request by returning an iterator on the requested feature. Before doing so, however, it will calculate a *transformation* to perform on the points comprising the feature. A transformation is a class consisting of a function to convert a point from one system to another, and some ancillary data needed in order to carry out this conversion. In order to convert from chart coordinates to window coordinates, for example, the transformation needs to know how large both of them are.

So when a function calls *GetFeature*, it provides a parameter specifying the coordinate system it wants to use to view points, and optionally, the size of the window in which objects are to be displayed. *GetFeature* calculates the transform, creates an iterator on the feature, and sets the transform in the feature's iterator. Each time the iterator returns a new feature instance to the using procedure, it first sets the transform in the feature's point list. The transform is not actually performed on any of the points comprising the feature until the points are actually used. This is desirable because some of the possible transformations, particularly chart-to-latitude/longitude, are relatively expensive.

It might seem that setting all of these transformations in each feature would also be expensive. However, most of the processing involved uses *inline* functions, so there is only a small performance penalty. The space usage is also fairly small, because the transforms on a particular feature can be shared among each feature instance, so pointers are used.

This strategy does not work as well when converting from latitude/longitude to window, or vice-versa. Here one has to go through a two-step process, i.e., convert from latitude/longitude to chart, and then from chart to window. Thus one must use two separate objects which, under the original strategy, are supposed to not have any direct dealings with each other. What to do?

It is here that the image object proved useful, since it serves as the point where a particular chart is tied to a particular window. Any task wishing to perform a transformation from real-world to window coordinates, for example, can use the image object's *LatLonToWindow* method. This method uses the chart object's *LatLonToChart* method, and then converts that result to window units.

7.7 Projections

Earth is a large sphere. Charts are flat rectangles. Projections are used to convert from one view to the other without distortion. There are a number of different projection systems used in cartography, but the mercator and transverse mercator systems are the ones most commonly used in navigation.

When a chart is digitized, the projection information for that chart is included in its file. This includes such information as type of projection, and the latitude/longitude of the chart's center. This data, along with a number of predefined constants, can be used to convert a real-world position into a chart position, and vice-versa.

The functions used to carry out these conversions vary tremendously depending on the projection used. This makes writing generic code a challenge. When dealing with an arbitrary chart, one does not want to have to first check the chart's projection, and then use a set of `if-then`'s to decide which conversion function to use. Such decisions might have to be scattered throughout the code whenever such transformations are needed, and would have to be updated if a new projection type is added.

The answer is to make projection information an abstract data type, and define methods to it named *LatLonToChart* and *ChartToLatLon*. When the chart file is first read in, the projection type is examined and the appropriate class for this projection is defined. From that point on, no other code needs to concern itself with which type of projection to use; it simply has to know that it is dealing with a class subtyped to the

Projection superclass.

7.8 Summary

While this chapter describes the high-level thinking that went into the design of the prototype components, it says nothing about their usefulness. The steps taken to cope with the complexities of the problem area appeared appropriate at the time, but only a rigorous evaluation strategy can determine whether the components are capable of achieving the intended reuse goals. The next chapter provides a framework by which one can evaluate such a set of components.

8. A FRAMEWORK FOR EVALUATION

8.1 Chapter Overview

The previous chapters show that the domain of navigation chart management is not simple. Quite the contrary, systems using navigation charts can have a complex set of requirements. Charting subsystems have to be able to accommodate motion. The chart display itself can move, or it can remain stationary while the information overlaying it moves. The subsystem has be able to support concurrent access by various different tasks. A number of coordinate transformation schemes have to be available. The safety-critical aspects of the application domain dictate that the information provided by the subsystem be clear, correct, and timely.

This chapter creates a framework by which object-oriented development can be evaluated as a software reuse strategy using this problem domain as an example. A set of assessment questions covering a variety of different aspects of the development and evolution process is introduced and explained. An intriguing side issue, namely the effectiveness with which one can use scavenged components to help create object-oriented systems, is also explored.

In the next chapter, a number of the points covered by this framework will be applied to the charting components as a preliminary assessment of their potential usefulness in developing new systems.

8.2 Evaluation Strategy

The intent of this evaluation framework is to establish some questions to help determine if object-oriented development constitutes a reasonable paradigm for reuse in an industrial setting.

There are four fundamental aspects of this project, all of which require quantification. These aspects can be *informally* stated in the guise of the following questions:

- A prototype set of components was created to help in building systems across a domain of applications. How difficult was it to create this initial set?
- Given the requirements for a *specific* application or set of applications, how effective are these components in helping to build them?
- How easy is it to modify the components to accommodate changes in the problem domain?
- What was the role of code scavenging in helping to build these components?

Words such as *easy*, *difficult*, and *effective* are extremely nebulous, especially as used in the contexts above. The creation of an evaluation framework is a step toward assigning rigorous meaning to them. Each section of this framework will discuss these questions in turn. The introduction of each section will describe the various aspects of the problem area. Each of these aspects will be examined in more detail in separate subsections. In Chapter 9, parts of this framework will be used to evaluate the work done so far on the components. However, many of the measurement points presented require a broad base of experience using the components in actual products. It is still far too early in the component's lifecycle to have such experience. It is important, though, to have this framework established near the beginning of the project so as to have a set of tangible goals to strive for.

8.3 Initial Component Development Effort

Developing a set of software components to accommodate an entire domain, rather than a specific system, can be a major undertaking. The developer should be prepared to make many mistakes in the early stages. This section describes some of major areas of evaluation which can be applied to the early stages of a component-based development project. The areas described are:

- Major design problems revealed during implementation.
- Effectiveness of the language used in component development.
- The lessons learned in this stage which might be transferable to other projects of this type.

8.3.1 Major Design Problems

What design problems were encountered, and how were they dealt with?

The most important design problems to work out are those which affect the way in which components interact with each other. These interactions are what determine how naturally they can be used in constructing systems. Other design decisions, such as what sort of data structure to use to represent a collection of related elements, can usually be encapsulated in a separate class and modified later if need be.

8.3.2 Effectiveness of Language Vehicle

How well does the language being used support the goals of the design?

As an old proverb points out, *if the only tool you have is a hammer, you'll tend to see every problem as a nail.* The most immediate tool the programmer has at his disposal is the programming language itself. It is the constructs and facilities of the language which shape the way he thinks about solving the problem. Some languages are clearly and deliberately more suited to some problem domains than others. If the language being used in the early stages is leading to awkward or inefficient code, it is time to consider other alternatives.

8.3.3 Lessons Learned

What insights were gained from developing the initial component set which can be applied to subsequent similar efforts?

An organization developing a set of components for one domain will very likely go through the same exercise for other domains as well. While this case study focused on navigation charting, it could just as easily focused on networking or automatic piloting. It is important to review what aspects of the project were done right and what was done wrong, and treat each iteration as a learning experience.

8.4 Using the Subsystem Components in Applications

The most basic question to answer in this project is *How useful are the components in creating new applications that use charts?* This question can be further qualified by asking a number of smaller questions dealing with specific issues affecting how the components are used. The issues discussed in this section are:

- The effectiveness of the domain analysis in yielding a useful set of components.
- The empirical evaluation of the manner in which the components are used in specific systems.
- The demonstrated time and space efficiency of the components.
- The storage and retrieval of classes versus other types of software work products.
- The determination of the amount of time saved by using the components in genuine development projects.

- The empirical evaluation of the components' reliability.

8.4.1 Effectiveness of the Domain Analysis

The subsystem components cannot be regarded simply as pieces of related code. They must also be thought of as the end result of the domain analysis. If the domain analysis itself was in error or grossly incomplete, it is likely that the parts themselves will not meet expectations. There must be some way of verifying the domain analysis itself in order to gain confidence both in it and in the products that follow from it. One might legitimately ask these questions about the domain analysis as part of the verification process.

- - -

How radically does the charting domain change as customers submit new system requirements? How do these changes affect the usefulness of the components?

In 1986, Congress passed the Tax Reform Bill. Imagine how much accounting software had to be changed to accommodate this traumatic upheaval to the domain of taxation. The lesson is, problem domains are not static. They change as technology changes, as legislation is passed affecting the domain, and as customers identify new needs. A set of components will ultimately become obsolete as a result of this. As this happens, the tendency will be for the programmer to start doing more and more development from scratch, defeating the original purpose of the components. Care needs to be taken, then, to monitor new customer requirements to see how well they match the domain analysis. Major discrepancies indicate that the domain analysis needs to be updated, along with the components themselves.

- - -

Does the domain analysis have strategic value?

In other words, can the domain analysis be used to predict future system requirements? If so, perhaps it is possible to assign to programmers the task of creating components based solely on the domain analysis, rather than on a specific set of system requirements. That way, when a new set of requirements does arrive, it is possible that an appropriate set of components will already be available and ready for use.

8.4.2 Component Usage

Do programmers tend to use certain components of the subsystem quite heavily, while ignoring others?

This situation could point to flaws in the domain analysis. Components might have been created to solve non-existent problems. Another possibility, if this situation occurred, was that certain components do serve a useful purpose, but are implemented so badly that the programmers feel compelled to write their own versions. The reason this question is brought up in the first place is because software libraries often contain a large number of entries, but not every component is used with equal intensity. As the number of charting components increases, the same situation is likely to occur.

- - -

Of the components being used, what percentage require modifications, and why?

It is almost certain that components will have to modified somewhat to fit specific situations. In spite of this certainty, modification should still be regarded as undesirable because of the extra work and complexity involved. It is important to keep track of how prevalent a problem this is, and under what circumstances it occurs. The more modifications the components require in order to be used, the more programmers will regard working with them as an obstacle rather than a blessing.

- - -

Does the percentage of components requiring modification tend to rise, fall, or stay the same over time?

As more components are added to the subsystem, one would hope that it moves closer and closer to the ideal solution space of the problem domain. This could be true if the problem domain stays relatively stable. The domain is likely to be changing as components are being added, making it necessary to modify old components. Traumatic changes, such as the obsolescence of vector charts in favor of raster charts, could result in mass modification or even the abandonment of the current set of components. By understanding modification patterns, one can get an idea of whether the set of components is truly worth having in the first place.

- - -

Have the components proven to be unused or even unusable to certain applications within the problem domain?

This question constitutes another way of quantifying how effective the components are in satisfying the problem domain. It is possible that in some situations, the components simply do not fit the requirements, and modifications would be too complex to be practical. It is also possible that programmers might be ignoring the components for other reasons.

8.4.3 Time and Space Efficiency

Do the components execute with sufficient time and space efficiency?

After all, it does not matter how well-structured the components are if they cannot execute fast enough to satisfy real-time constraints, or if they cannot fit on the target machines.

8.4.4 Library Storage and Retrieval

It is obvious that once created, a set of components must be kept somewhere for all to use. The questions here deal with component library management with respect to object orientation.

- - -

How does a class-based reuse scheme affect the way components are stored in a library?

The fact that classes often inherit attributes from other classes adds a complication to the component storage and retrieval issue that is not present when dealing with more traditional reuse methods. Classes need to be stored such that the inheritance relationship is explicit and easy to manage.

- - -

Is there a significant difference in the ease with which classes can be retrieved from a library versus other, more traditional components such as functions?

In a traditional software library scheme, if the programmer wants a particular function, he locates it in the library under some logically-arranged category, and checks it out. If the programmer wants a *class* which defines a particular object, what does he do? He might not understand the inheritance relationships involved. And his way of thinking about the object being defined by a given class might not match the way the original programmer thought about it.

8.4.5 Savings in Development Cycle Time

How much time is spent analyzing, designing, coding, and testing chart-related parts of new applications relative to the time spent on these activities in other parts of the same system?

One would think that a large, domain-specific set of components would make software development in this domain substantially more efficient. There is always a possibility, however, that the programmer has to spend so much time deciding whether a set of components suits his needs that time is actually lost.

This issue points to the importance of having a comprehensive time tracking system in place to measure how much time is spent on each phase of each project. Such systems are also extremely useful in tracking productivity data in general, not just time save by reusing software. One significant problem with these kinds of systems is programmer acceptance. Programmers typically regard such systems with disdain, seeing them as just more bureaucracy. Significant research is needed on these systems, particularly the user interface, to make them extremely easy and even fun to use. Along with this, work is needed to see how much productivity data can be gathered by the system automatically without the programmer having to enter the data manually.

8.4.6 Reliability

For systems using these components, what is the relative error rate over time of chart versus non-chart code?

Errors uncovered both in development and after deployment need to be tracked to see if there is a significant difference in the quality of the chart components versus other parts of the system. An error tracking system with facilities to do this correlation needs to be created to do this. As with the productivity tracking system mentioned above, care needs to be taken on the interface to this system to make it acceptable to the programmers who will use it.

- - -

Does the error rate of the chart component code tend to rise, fall, or stay the same over time?

The infamous *ripple effect* describes the introduction of new problems into software as old problems are being fixed. This question asks how big a problem the ripple effect poses as the components are modified. The hope is that the components are sufficiently loosely-coupled that changing one part of the system will not affect another.

- - -

What kinds of errors are found in the chart components? Can they be traced to the domain analysis, the design, coding problems, or some other category? How serious are these problems?

A feedback mechanism needs to be established for determining the origin of errors and correcting the source of the error as well as the error itself. The process of assessing the relative severity of a problem, and isolating where in the development cycle the error was introduced, would be difficult to automate. Organizations must be willing to devote a good deal of human resource to discerning the cause of mistakes and learning from them, rather than just correcting them.

- - -

How many problems can be traced to misunderstandings of how the components are supposed to be used?

This item indicates the importance of clear, comprehensive usage specifications for the components. This reduces the chance of the programmer getting the wrong idea about what the component can and cannot do. One way of dealing with this issue is to include usage documentation with the component itself in the component library. An important feature of this documentation should be examples showing how the component is used in specific code fragments. A *man page*-type format that simply describes parameters and options of a component is usually not enough to get a clear idea of how a component should be used in some context. This is especially true if a component is large and consists of several subcomponents.

- - -

How many problems can be traced to modifications of the original components?

It is well known that software is like people in the sense that it tends to get grouchier as it gets older. As changes are introduced, reliability often decreases, sometimes dramatically so. Keeping track of the enhancements made to the components, and which enhancements ultimately lead to failures, could shed some light on how to introduce modifications to the components without damaging them.

8.5 Characterizing Change

Because the subsystem was not designed for a specific application, but rather an entire domain of applications, it must be prepared to undergo numerous changes throughout its life. There must be a structured way of managing these changes. This section describes change management as the *quantification of incremental design decisions*.

Parnas discusses the *program family* concept to describe how versions of a particular applications can be described according to the differences in their design decisions [Par76]. A development strategy that supports systematic reuse should support this family concept by:

- Enabling the programmer to delay certain design decisions when necessary, without stopping progress on the project altogether.
- Enabling the programmer to easily identify the crucial design decisions that make up a system.

- Enabling the programmer to do the requisite backtracking to eliminate the design decisions which are not useful or even counter to his purposes.
- Enabling the programmer to impose new design decisions on the backtracked system to derive new system versions.

Information hiding techniques facilitate the first of these items. But from a practical standpoint, determining what aspects of a system need to be changed to fit a new set of requirements can be a difficult task. The order in which new functions were added to derive a new version is often a crucial piece of information, because functions often depend on the existence of other functions in order to work properly. This section discusses the issue of how to characterize the major functional differences between a set of related applications, and use that knowledge to help derive new versions. The concept of *version tables* is introduced, and an example of how to use them is presented.

8.5.1 System Version Tables

In order to characterize the ease with which a set of components can be reconfigured to form new system versions, one needs a succinct way of capturing the major functional differences between versions. The method proposed here is that of *version tables*.

Version tables capture the object and function information of a set of components, and associate it with specific versions of the system. By also associating each version of the system with a *parent version*, the programmer can backtrack to previous instances of the system where undesired functionality was not present. Presenting information in this format allows the programmer to see the order in which versions were derived, and easily check the dependencies between versions. If the different versions of the components involved are maintained under a configuration management system, the programmer can use the version that matches his needs the closest as the base for modifications.

8.6 Object-Orientation and Code Scavenging

Another set of issues to consider when evaluating this development strategy is the ease with which one can incorporate code scavenged from other systems. The approach described here is object-oriented, yet there is a vast supply of non-object-oriented code available which one might want to use when producing such a system. It is useful to consider some of the changes which can occur to non-object-oriented code when it is used in an object-oriented setting. This section will consider three aspects of this process:

- Data initialization.
- Modification of data by reference parameters
- Use of general functions not specific to any particular object.

8.6.1 Data Initialization

Object-oriented languages, particularly C++, allow the programmer to define implicit data initializations to take place automatically when an object is created. This relieves the programmer of the burden of having to explicitly initialize a set of data items every time he creates new instances of them. Older code is often peppered with special routines for carrying out these initializations, along with calls to these routines. Automatic initialization does not eliminate the need to create the initialization code, but it does eliminate the need for the programmer to remember to make calls to this code before using the new data. This reduces the chance of suffering puzzling problems due to the use of uninitialized data items.

8.6.2 Modification of Data By Reference

It is common in non-object-oriented code to pass data elements into functions which perform some sort of update on those elements. These updates are typically made on data passed in via reference parameters. In object-oriented code, modifying major data objects this way is usually not allowed; data included in objects can only be modified through the object's interface. So functions which modify data via reference parameters often need to have those parameters removed. If a function modifies data this way, it is usually a sign that it should be made a member function of the object to whom the data in question belongs.

8.6.3 General Calculations

Even in the object oriented development paradigm, not everything should be considered part of an object. Functions which convert units of measurement from one system to another, for example, or functions which convert ASCII characters into their numeric equivalents, can be hard to associate with a particular class because they might be used by any number of classes. C++ allows the declaration of *friend* functions to help deal with this problem. A function specified as a *friend* of a class has access to the data hidden within the class, but is not actually a member of the class. This way, a function can be used by a large variety of classes without actually belonging to any of them.

8.7 Summary

The goal of this chapter was to identify the most important areas of evaluation for this type of development strategy. A great deal of emphasis was placed on *change*; how to characterize it, and how to cope with it. The framework presented here will be refined as the project continues, and more questions are likely to be added. The issues raised here should at least give focus to subsequent work. The importance of having an evaluation framework of any kind, subjective as it might be, is that it injects a certain amount of discipline into a traditionally indisciplined process.

The next chapter shows how parts of this framework can be used in evaluating the chart components themselves, and some sample systems constructed using these components.

9. PRELIMINARY COMPONENT EVALUATION

9.1 Chapter Overview

The previous chapter presented a framework by which a component-based development strategy can be evaluated. This chapter applies many of the aspects of this framework to the electronic charting components. The aspects of interest are:

- Initial component development effort.
- Time and space efficiency.
- Analysis of component versions.
- Usage in sample applications.

These aspects were chosen because they do not require a lengthy time frame in which to provide preliminary results. Of these four aspects, the last one is the most revealing because it sheds light on the potential usefulness of the components in practical situations.

To evaluate the aspect, two small systems were implemented using the components. One was intended to capture some of the major functional requirements of a *land-based charting system*, while the other was directed toward *sea-based systems*. Sections 9.5 and 9.6 describe the requirements and implementation of each one, and discusses them in light of the evaluation framework. In no way should these sample systems be construed as fully operational. Their purpose is to demonstrate how the components might be used in such systems, and how well the components satisfy the needs of these systems.

9.2 Initial Component Development Effort

The functional goals of the initial set of components were quite modest, and can be stated thus:

- Read in a Sperry Marine-created chart file.
- Put the features listed in this file into their appropriate "slots" in the *Chart* object.
- Retrieve selected features from the *Chart* object and display them in the appropriate manner in an X-Window on a Sun Workstation. Recall that features will be displayed with different forms, colors, and motions in different systems.

Other functional requirements, such as image manipulation, overlay processing, and real-world coordinate calculation, were then added to this base-level. In evaluating the creation of this base-level, these areas are considered:

- The major design problems encountered.
- The effectiveness of the chosen language in supporting the implementation.
- The lessons learned in the early stages of the implementation.

9.2.1 Major Design Problems

The issue of coordinate system transformations, discussed in Chapter 7, was unquestionably the most difficult design problem in the project. At issue was the determination of a clean way of transforming points between any two of three coordinate systems at any given time. The main problem was the fact that in order to transform a *Chart* coordinate into a *Window* coordinate and vice-versa, a function has to be created that knows the size of both the *Chart* and the *Window*. It was not clear where such a function could be placed and still maintain a natural interface to both the chart and

the window abstractions. The creation of the image abstraction, which associates a particular *Chart* with a particular *Window*, solved this problem, along with the problem of where to implement chart zooming and offsetting.

Another significant problem was how to cope with the addition of new chart features. The components are currently able to process landmasses, shoals, and hazards. It is certain that buoys, text, and other features will be needed in later versions. The question is how to process features of assorted different types without using special-case processing throughout the code. Chapter 7 describes the subtyping strategy used to circumvent this problem. The only special-case processing used for features is done when initially loading these features from files; the *Chart* class needs to know what type of feature it is processing, so it can assign it to the appropriate *Feature* class before storing it.

9.2.2 Effectiveness of Language Vehicle

C++ was a good language choice for the components. It has effective data abstraction and encapsulation properties, and executes efficiently. It also treats classes as first-class entities, so classes can be passed as parameters, assigned to other classes, and nested within other classes. C++'s function overloading proved surprisingly useful in helping to keep the code clear and consistent.

However, translation to C tends to be slow, and the C code produced is unreadable. Also, the compiler version with which the components were developed did not give adequate support to generics, which would have been extremely useful in many places.

9.2.3 Lessons Learned

Using *iterative enhancement* [BaT75] as the development strategy for the initial component set was an excellent choice. However, one must be disciplined in the way one uses this approach. It is important to establish a written set of checkpoints at the outset of the project. At each checkpoint, review progress and problems encountered since the previous checkpoints. Determine which design decisions were flawed, and correct them before going on. This is basically the advice offered by Basili and Turner [BaT75].

9.3 Time and Space Efficiency

An informally stated requirement about the subsystem was that it should load and display a given chart in three to five seconds, and zoom an image in one to three seconds. On average, the subsystem can load and display a complete chart on a networked Sun 3/75 Workstation in six seconds from the time the request is made. An image can be zoomed or panned in about 1.2 seconds. These timing values were obtained by instrumenting test driver code with calls to the C++ *times* function. Values vary depending on the size of the chart being displayed. The slowness in loading is probably due to the way chart features are being loaded into the chart object. Features are stored in dynamically expandable arrays, and the algorithm used to expand the arrays is very inefficient; a significant amount of array allocation and copying is going on, and some of the arrays are fairly large. Changing the feature storage implementation to a collection of linked lists, rather than arrays, would probably lead to a significant speed-up.

The other concern is the amount of space the subsystem takes up in memory. The platforms on which the subsystem would be used in production are currently limited to

640K RAM. When the subsystem components are executed using a simple test driver and a chart is loaded, the space usage goes up to about 360K, as measured using the Unix *top* tool. As with timing measurement, this figure varies depending on chart size. From examining the code, it is clear that this number can be brought down with better memory management techniques.

9.4 Version Analysis of the Chart Subsystem

Table 9.1 shows a simple version table for the chart subsystem. When using this method, the assumption is that the classes and functions of version 0, the *root version*, has been defined in detail. Appendix C, the component *class catalog*, serves this purpose.

As shown here, the development structure is linear. So version 2 has all the capabilities of version 1, along with the ability to display two new feature types. Version 3 has all of these capabilities, plus it can display charts in color, and so on. Suppose we want a hypothetical version 5, which requires all of the capabilities of version 2, along with the ability to do overlays. It is easy enough to backtrack to this version and add the necessary objects and functions to do this. What is potentially harder is if we instead want a version which has all the capabilities of version 2, plus the capabilities of version 4, but without the capabilities of version 3. As it turns out, the ability to display color is orthogonal to the ability to choose a new chart center, so this change would be implemented by removing color capabilities from version 4. We might not be so lucky in all cases, though. Intermediate versions of the subsystem might have capabilities which depend on the existence of other capabilities which we do not want. Care needs to be taken to avoid such dependencies.

Version	Parent Version	New Function	Modification of Class Attributes	Modification of Class Functions	Description of New Class
1	0	Add zoom.	Add current and base zoom factors to Image.	Add zoom in/zoom out and zoom calculation functions.	
2	1	Display shoal and hazard features.	Add shoal and hazard codes to FileScan code conversion table.	Chart.Load needs to recognize shoals and hazards.	Define new hazard class as a derived class of point feature. Define new device fill pattern abstract data types.
3	2	Accomodate color Sun display.	Add color field to Feature data class.	- Add SetColor functions to Chart, Window, Device, and Feature classes. - Add GetColor function to Chart and Feature data class.	-Create abstract data types defining RGB combinations for X. - Derive new class from X device driver class. This class should have all the functions of the monochrome driver, except the initializer defines color mappings.
4	3	Allow user to choose new chart center point.	Add current center value to Image class.	Add ChangeCenter function to Image.	

Table 9.1 Version Table for Initial Components

The ability to display charts in color is a good example of this. It was tempting to modify the monochrome device driver to automatically determine whether it was operating on a monochrome or a color platform, and switch accordingly. But this would mean all subsequent applications would have the ability to do both color *and* monochrome display. This does not sound bad, but most applications run in color. Including the ability to do both could potentially slow performance somewhat, and take up extra space. It would certainly make the display driver code more complicated.

It was decided to create a separate color display driver, which inherited all of the monochrome features and added processing necessary for color. The programmer can thus choose one driver or the other for a given application. If there is a need to include both capabilities, a simple interface object can be written to determine which driver to use, and instantiate the right class at run-time. Thus color and monochrome are regarded as orthogonal display properties, and are kept separate.

9.5 Component Usage in a Simple Sea-Based System

The first sample system created with the components was a simple application designed to demonstrate some of the functions common to sea-based charting applications. This section describes the command and control behavior of the system, and the evaluation criteria applied to it.

9.5.1 Command and Control Behavior

Figure 9.1 is a finite-state automaton diagram which shows the command and control behavior of the simple sea-based system. The arcs represent user-input commands and system transitions; the states represent system modes and calculations resulting from these commands. The major functions depicted in this diagram can be described

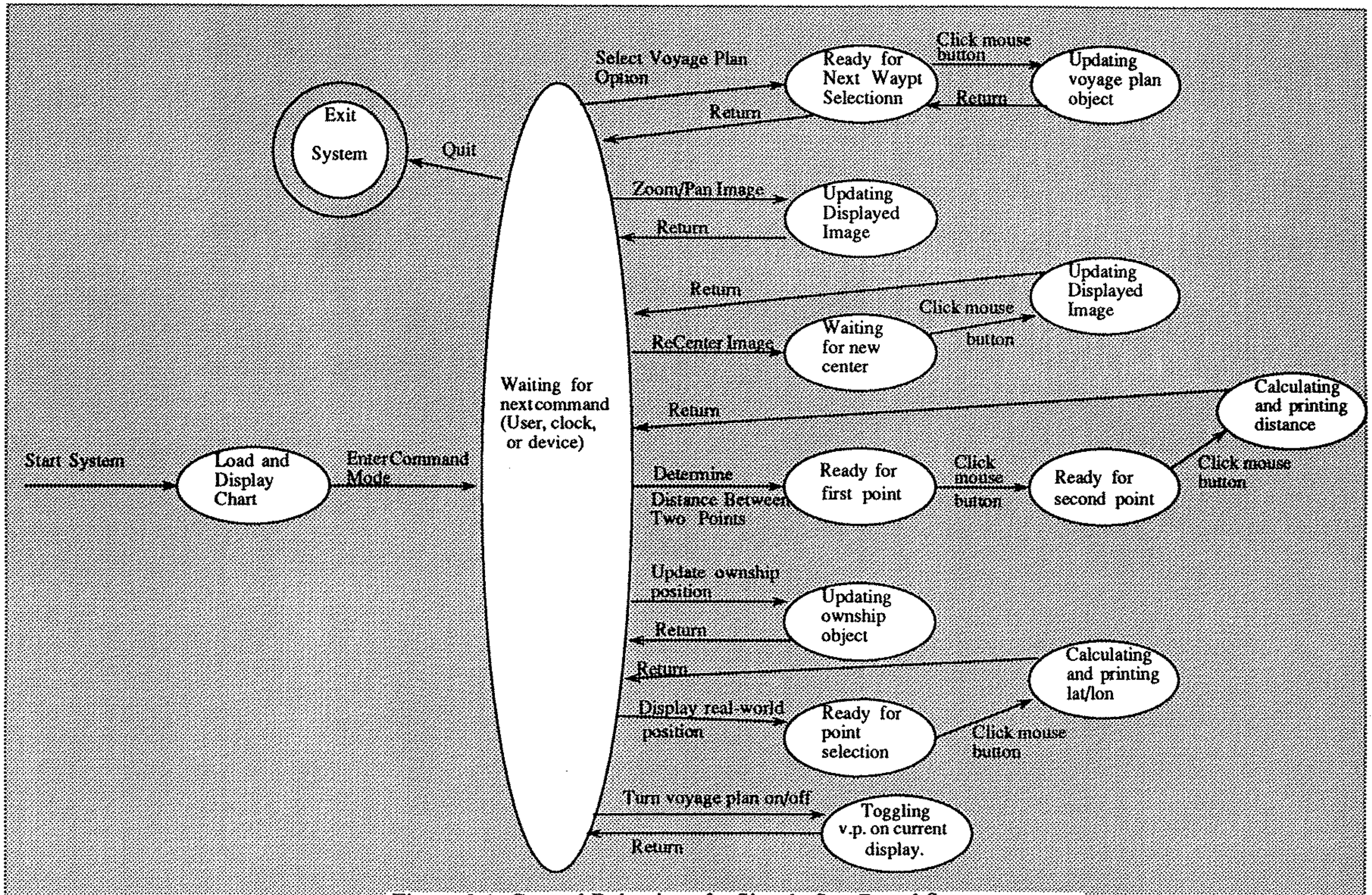


Figure 9.1 Control Behavior of a Simple Sea-Based System

thus:

- *Create new voyage plan*

  Allow the user to trace a path on a displayed chart with the mouse. At each planned waypoint, the user clicks a mouse button. The coordinates of each waypoint are stored in a *voyage plan* object, which can be used for further processing.

- *Turn on/off voyage plan*

  Allow the user to toggle on or off the voyage plan created above. The digitized points are shown connected by line segments.

- *Zoom or pan image*

  Zoom in on or pan out from current center by a pre-set percentage.

- *Re-center image*

  Allow the user to position the mouse pointer on any currently displayed point of the chart, and choose that point as the new window center by clicking a button.

- *Display real-world position*

  Allow the user to position the mouse pointer on any currently displayed point of the chart, and display the real-world latitude/longitude position of that point by clicking a button.

- *Determine distance between two points*

  Allow the user to select any two points on a chart, and display the straight-line distance between those points in nautical miles.

- *Update ownship position*

  Schedule the system to automatically update the display position of the user's ownship on the screen at some time interval. Once the user has chosen to display ownship position, this updating should occur automatically. The user should be allowed to perform other chart tasks while these updates are occurring, so preemptive multi-tasking is desirable. The current implementation does not do multi-tasking, so a movement simulation is provided instead.

  Also, the user should have the option of displaying the ownship's track history, which shows up as an ever-growing tail behind the moving ship symbol.

The first two items and the last item are functions that distinguish in part this system from a land-based system. A land-based user would not need voyage planning functions because he is not going anywhere. And, of course, since he is not on a ship, he would not need any ownship manipulation functions. The zooming, panning and recentering functions demonstrate the manipulations the user would need to perform on the chart image itself. The functions that display real-world coordinates and distances between points demonstrate the capability of electronic charts as nautical information management tools.

9.5.2 Evaluation Criterion - Component Usage

Two questions from the evaluation framework created in Chapter 8 will be dealt with in this subsection:

- Do programmers tend to use certain components of the subsystem quite heavily, while ignoring others?
- Of the components being used, what percentage require modification, and why?

To the first question, all of the original components of the subsystem were used in creating the sample sea-based system. A number of new classes were created as well:

- *Voyage Plan* - Stores the points entered by the user, and displays the projected track on the chart image when requested.
- *Ownship* - Holds current chart coordinate of the ownship object. Also holds the ownship's track history in a list of points. Right now, this class will display the ownship symbol as a circle.
- *Ownship Controller* - For demonstration purposes, this class reads pre-defined ownship track input from a file and updates the ownship symbol on the screen as appropriate.

To the second question, the main functional modification required was the addition of the ability to convert chart coordinates to latitude/longitude points. This required modifications to the following classes:

- *Chart* - Required a *ChartToLatLon* function to invoke conversion function in the *Projection* class used by this chart.
- *Projection* - Required a new function to perform chart-to-world coordinate conversion.
- *Image* - Required a new *PrintLatLon* function to print the results of the conversion in a human-readable format. Also, a *WindowToLatLon* function was added so the conversion from window to latitude/longitude coordinates could be made directly via the *Image* object.

The ability to calculate distances between two selected points also required the modification of a pre-existing class:

- *Image* - Required a new *Distance* function to calculate the nautical mile distance between two points provided in chart coordinates.

Given that there are currently 20 distinct classes in the subsystem, the percentage that required modification was 15%. The rest of the code required for this system was non-subsystem code, and had to be developed from scratch. This included the new overlay classes mentioned above, along with code to process input from the mouse via X-Window library calls. The number of reused lines of code in the components was about 2600. The new classes and application driver code totaled about 457 lines of code. These estimates are rough, and are based on running *wc* against the system's *.h* and *.c* files.

9.6 Component Usage in a Simple Land-Based System

This section is similar to the previous one, with the discussion focused on a sample land-based system using the components.

9.6.1 Command and Control Behavior

Figure 9.2 shows the command and control behavior of the sample land-based system. This system does not require ownship and voyage plan processing functions. It

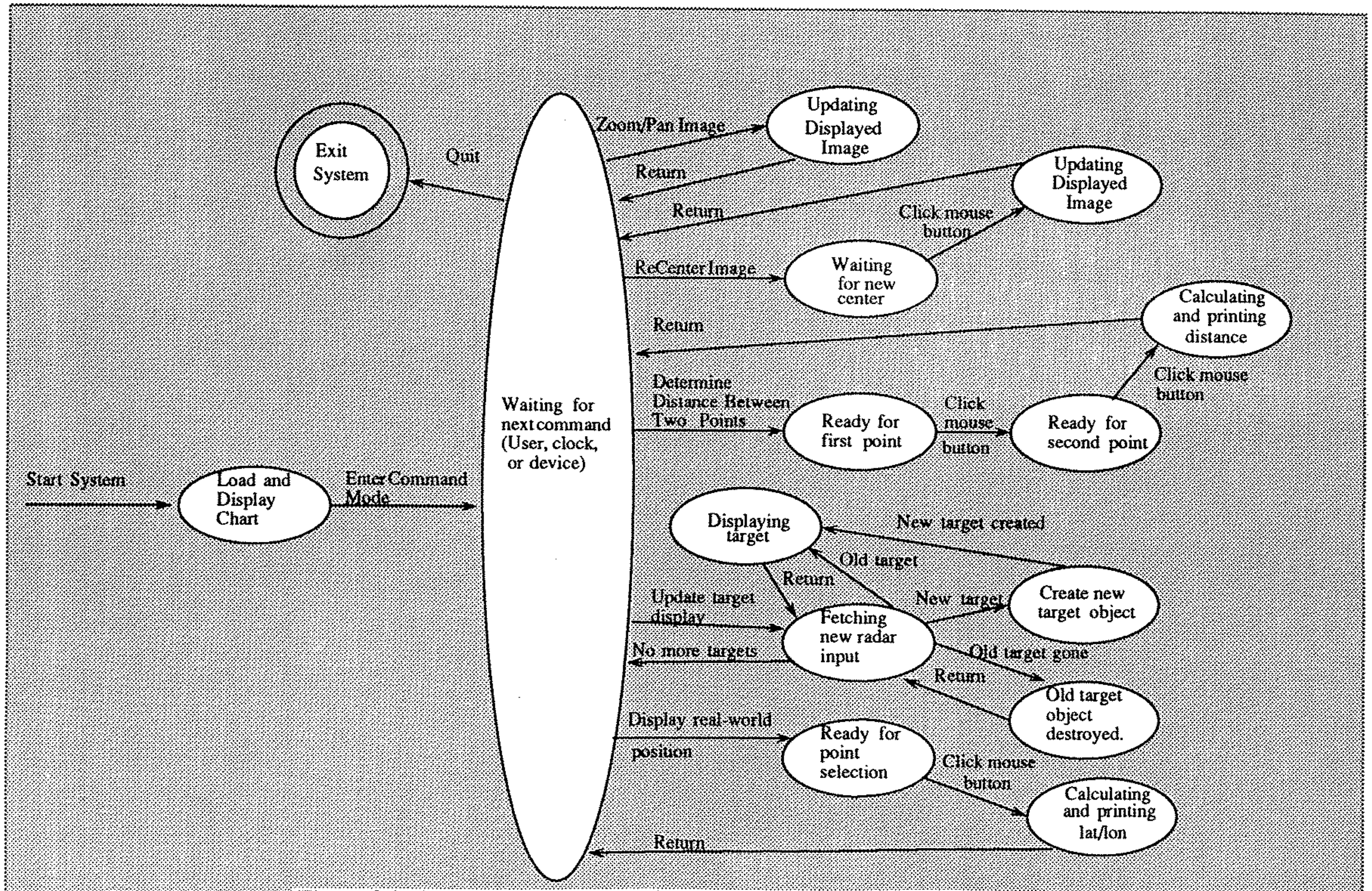


Figure 9.2 Control Behavior of a Simple Land-Based System

does, however, require radar target processing functions. This involves monitoring the motion of multiple targets in the field of view of the radar sites placed in the harbor area. The primary application of such information is collision avoidance.

Target processing is fairly complex. The number of targets being tracked at any one time can be very large. In fact, one recent production system of this type can process up to 2000 targets at one time. New targets can appear at any time. Old targets can go out of range, go behind obstructions, or disappear in various other ways. Most of the targets to display are constantly moving in various directions. The system has to be able to process all of this constantly-changing information in a very short amount of time.

The sample application simulates radar targeting by reading in a file defining target identifiers and corresponding real-world positions. Each target is part of a group of targets called a *scan*. The application reads a scan every three seconds, updating the displayed targets appropriately. Any active targets missing from the current scan are assumed to be gone and are removed from the display.

9.6.2 Evaluation Criterion - Component Usage

The simple land-based system was able to use all of the components used by the sea-based system without modification. The only additional components that needed to be created were the following:

- *Target* - This class was derived from the *Ownship* class created in the previous system. The only difference between the two is that target ships display on the screen slightly smaller than ownship symbols. Their functional behavior is identical, and both require history tracking.

- *Target Controller* - This class takes in radar input, uses the input to create and destroy target symbols as they come into and out of existence, and updates each target as it moves. The target controller stores each target object in a data structure that allows access via a numerical identification. The current implementation of this is an array, although a b-tree might be more flexible in a full implementation.

The modifications made to the *Chart*, *Projection*, and *Image* components were used in this system as well. Thus 0% of the classes in the subsystem required modification. About 2710 lines of subsystem code was reused, including development from the previous system. About 420 lines of new code was required to process targets and serve as an application command processor. Again, these line of code estimates are based on *wc* output against the system's *.h* and *.c* files.

9.7 Summary

The components have proven to be useful in constructing systems which demonstrate some of the important concepts of land and sea-based electronic charting systems. Substantially more work is required before a definitive statement can be made as to how effective the components are in constructing realistic, production-quality systems. The next chapter discusses the various forms this future work might take.

10. CONCLUSIONS AND FUTURE WORK

10.1 Toward a Reuse-Oriented Development Paradigm

Component-based software development may or may not be the solution to the *software crisis* mentioned in the *Introduction*. This work has highlighted some of the more important issues involved in determining the effectiveness of this paradigm. Central to this paradigm are:

- *The derivation of components from existing software.*

  There is a wealth of existing software from which components can be derived. Performing this derivation is a difficult technical problem, and requires a great deal of research. The *Insights* section of Chapter 4 discusses a number of possible research topics.

- *Domain analysis.*

  Components must be suited for a domain, not just a specific application. But one must have reasonable expectations about how wide the range of applicability of a set of components should be. It is not reasonable, for example, to expect to be able to use components intended for navigation charts management in an accounts payable application. On the other hand, domain boundaries are always fuzzy and often gradually merge into each other.

- *Prototyping.*

  One sometimes hears the phrase *canonical designs* used to refer to designs which capture the essence of an entire class of systems, rather than one particular instance. The iterative enhancement strategy used in this project resulted in a prototype which has attributes of such a design.

- *Evaluation*

  Evaluation must be considered an ongoing process rather than a one-time chore. Tools and methods need to be established to facilitate this process. It is important to regard evaluation as a continuous mechanism to avoid obsolescence.

10.2 The Next Phase

The work undertaken in this project marks the beginning of an extended research effort in practical software reuse. This case study has identified a number of issues which subsequent project phases can explore further. Among the more important of these issues are:

- The creation of a comprehensive tool set to assist with the software scavenging effort.
- The creation of a library to serve as a repository for functions, classes, and other software work products.
- The completion of the charts subsystem prototype, and full evaluation of it according to the framework set up in Chapter 8. As this activity is carried out, it should be possible to assess the evaluation framework itself, refining it and organizing it into a set of standard quantitative measurements.
- The creation of innovative productivity and reliability tracking tools to assist in measuring the effectiveness of the components, as well as providing a good base of project data for other studies of this nature.

10.3 Larger Issues

A number of intriguing and more general insights arose from this project which deserve further consideration. These include

- The influence of multitasking semantics on reusability.
- The influence of language exception-handling semantics on reusability.
- The role of change in problem domains, and how to structure components to accommodate change.

Appendix A: Programmer Interview Questions

This is a list of questions prepared as part of the initial information gathering phase.

A.1 Design

- What tools and techniques do you use in creating a design?
- Do you often model off of previous designs? If so, how do you determine what model to use, and what changes to make?

A.2 Coding Strategy

- Do you find yourself writing a lot of similar code for similar functions on different projects? If so, what are some examples?
- Do you keep old code fragments around to model off of?
- Do you sometimes find that a design has to be changed because there is no easy way to code it?
- What kinds of useful functions and routines would you like to see made available to the organization?

A.3 Testing

- What test strategies do you use?
- How do you derive test cases?
- Can you sometimes use the same test cases from project to project?
- Do you have helpful test tools available?

A.4 Modification

- What are some typical changes and enhancements which you have made to existing code?

- When making a change to existing code, how useful do you find the documentation for the code? What are some problems?

Appendix B: Initial Set of Components

The following pages constitute the set of functions and modules yielded by the initial ‘‘code scavenging’’ activity. Also included are some broad areas identified as fruitful reuse areas, but which currently appear to lack general components in the existing code.

Potentially Reusable C Artifacts

Function (Networking)	Module	System	Description
aux_crc	navlines.c	VMS	Cyclic redundancy check algorithm.
compute_checksum	"	"	Computes one-byte additive checksum.
net_rec_navlines	"	"	Receive messages from network.
parse_message	nmea0183.c	"	Parse nmea0183 messages.
message_gll	"	"	Parse nmea0183 GLL messages (GPS, Loran-C)
message_pfd	"	"	Parse nmea0183 PFD messages (engine room).
message_rta	"	"	Parse RASCAR RTA messages (targets).
message_sta	"	"	Parse nmea0183 STA messages (pilot stat).
message_vbw	"	"	Parse nmea0183 VBW messages (speed)
comm_task	tdevices.c	"	Get message from serial port.
network_read	"	"	Get messages from network.
LogInfoChecksum	datalog.c	RASCAR	Return a 16-bit checksum.
Checksum	estartup.c	"	Checksum words from start pos to end pos.
ComputeChecksum	msgio.c	"	Compute one-byte additive checksum.
ComputeCRC	"	"	Compute two-byte cyclical checksum (table)
AuxiliaryComputeCRC	"	"	Compute two-byte cyclical checksum (non-table)
NMEAcksum	psi.c	ADG	NMEA checksum generator and verifier.
LLC network interface	llcio.c	Non-Specific	LLC layer token ring network i/o.
Serial message interface	msgio.c	Non-Specific	Serial message i/o.

Function (Unit Conversion)	Module	System	Description
spd_knots_display	globals.c	VMS	Convert speed units to knots.
spd_display_knots	globals.c	"	Convert knots to speed units.
convert_faw	tplan3.c	"	Convert speed over ground to water speed.
XYtoRhoTheta	globals.c	RASCAR	Convert long integer cartesian to integer polar.
tmfwd	m_lib.c	VMS	Transform geodetic coords to normal gaussian

tminv	"	"	Transform normal gaussian coords. to geodetic coords.
merfwd	"	"	Transform geodetic coords to normal mercator coords.
merinv	"	"	Transform normal mercator coords to geodetic coords.

Function (Time Conversion)	Module	System	Description
convert_date_time_to_gmt	tplan.c	VMS	Converts waypoint's eta from date & time to gmt.
mod_time	tplan1.c	"	Modify gmt to standard or daylight time.
local_time_conv	tplan1.c	"	Convert gmt to local time.
fmt_hr_min	tplan2.c	"	Convert seconds to hours/mins.

Function (String Conversion)	Module	System	Description
LongToString	globals.c	RASCAR	Convert unsigned long integer to dec. ASCII string.
NumToString	"	"	Convert scaled binary fraction to string.
NumToStringXP	tgtdata.c	"	Convert extra-precision number to ASCII.
AngleToString	globals.c	"	Convert semirevs. angle to string.
StringToNum	"	"	Convert string to unsigned number.
StringToAngle	"	"	Convert string to unsigned angle.
fmt_date	m_lib.c	VMS	Convert date in days to string
fmt_deg	"	"	Convert lat or lon in radians to string.
fmt_time	"	"	Convert time in seconds since midnight to string.
str_bits	"	"	Convert numbers in string in integers.
str_data	"	"	Convert string to julian date
str_pos	"	"	Convert string to pos. value in Nort/East ref. sys.
str_time	"	"	Convert string to time val. in seconds since midnight
get_int	play.c	"	Return an int, given string, start pos, and length.
convert_sys_time	tplan1.c	"	Convert sys. date to mm/dd/yy and time to hh:mm:ss
get_time_date	tlog.c	"	Get time & date formatted in string.
TimeToString	datalog.c	RASCAR	Convert time of day from internal format to text.
StringToLong	msgio.c	"	Pack a four-byte string into a long.

Pos2String	navlines.c	VMS	Convert lat or lon coords to message string.
String2Pos	"	"	Convert lat or lon message string to coords.
atod	printf.c	ADG	Convert ASCII string to decimal
itod	"	"	Convert integer to ASCII string (decimal)
utod	"	"	Convert unsigned integer to ASCII string (decimal)
itox	"	"	Convert integer to ASCII string (hex)
itoo	"	"	Convert integer to ASCII string (octal)

Function (Nav. Calculation)	Module	System	Description
get_range_bearing	tplan1.c	VMS	Return range and bearing from great circle or rhumb line funcs.
get_timezone	tplan3.c	VMS	Update timezone global variable.
xtrack_er_gc	tplan3.c	VMS	Calculate cross track error of ownship wrt. voyage plan (GC).
xtrack_er_rl	tplan3.c	VMS	Similar, but for rhumb line track.
conv_to_cart	tplan3.c	VMS	Convert lat/lon point to Cartesian coords.
LineCircleInt	globalr.c	RASCAR	Line-Circle intersection.
LineInBox	"	"	Limit line endpoints to screen boundary.
Compress	"	"	Return point inside screen boundary.
GreatCircle	"	"	Calculate range & bearing
DRLatLon	"	"	Compute new lat/lon from start lat/lon
LatLonDR	"	"	Inverse of DRLatLon
dr	m_lib.c	VMS	Calculate lat/lon position
great_circle	"	"	Calc range and bearing.
rhumb_line	"	"	Calc. course and distance from start to end.
proj_fwd	"	"	Convert a given lat/lon into inches from origin
proj_inv	"	"	Convert inches from org. to lat/lon.
cal_twind	tweather.c	"	Calculate true wind direction and speed.
set_cog_sog	tdevices.c	"	Compute/set speed and course over ground.

trav_time	"	"	Compute time dif. between fix time and cur. time.
Rotate	disptrans.c	RASCAR	Coordinate rotation (float).
RotateX	"	"	Coordinate rotation (fixed).
RotateL	"	"	Coordinate rotation (long).

Function (File Management)	Module	System	Description
read_config_file	navlines.c	VMS	Reads configuration file (DEVICES.DAT)
get_record	nskv_rea.c	"	Get next record from a file.
tfclose	tfile.c	"	Close file.
tfgetc	"	"	Get char from file.
tfgets	"	"	Get string from file.
tfopen	"	"	Open file.
tputc	"	"	Put char to file.
tfputs	"	"	Put string to file.

Function (String Manipulation)	Module	System	Description
breakline	parse.c	ADG	Break string into substrings.
breakline	tcomm.c	VMS	Same. (Different implementation)
ckmsg	parse.c	ADG	Check an input string against a string template.
ckmsg	nmea0183.c	VMS	Same. (Different implementation)
rjust	tscreen.c	VMS	Right justify string using format string.
AppendString	datalog.c	RASCAR	Append a string to another string.
Spaces	"	"	Append specified number of spaces to a string.
XORStr	"	"	Compute 8-bit exclusive-or of string.

Function (I/O Management)	Module	System	Description
dgetc	serio.c	ADG	Get character from a serial i/o port.
dgetl	serio.c	ADG	Get line from a serial i/o port.
dputs	serio.c	ADG	Put string out to serial i/o port.

nfgets	c_file.c	VMS	Get string from network file buffer.
fix_sprintf	tplan1.c	VMS	Fix sprintf bug in Turbo C (see comment).
ptask	tlog.c	"	Output string to printer.
printer	tlog.c	"	Output string to printer.
fprintf	printf.c	ADG	File print formatting.
sprintf	"	"	Buffer print formatting.
printf	"	"	Print formatting.
Input Management	inputapi.c	VTS	Input device interrface.

Function (Password)	Module	System	Description
update_password	password.c	VMS	Update password on disk for current operator.
validate_password	password.c	VMS	Validate password for current user.
encrypt	password.c	VMS	Encrypt a password string.
read_password_file	password.c	VMS	Read password file from disk.

Function (Math)	Module	System	Description
abs16	globalr.c	RASCAR	Absolute value of integer 16
abs32	"	"	Absolute value of integer 32
abs	utility.c	ADG	Integer absolute value.
fabs	"	"	Float absolute value.
min	"	"	Integer min.
fmin	"	"	Float min.
max	"	"	Integer max.
fmax	"	"	Float max.
lim	"	"	Limit magnitude of integer.
flim	"	"	Limit magnitude of float.
in_range	"	"	Bound integer in range.

fin_range	"	"	Bound float in range.
wrap0	"	"	Adjust for 0/360 degree discontinuity.
fwrap0	"	"	Adjust for 0/360 degree discontinuity (float).
wrap180	"	"	Adjust for +/- 180 degree discontinuity.
fwrap180	"	"	Adjust for +/- 180 degree discontinuity (float).
flipbits	"	"	Exchange bits 15&0, 14&1, etc.
ATAN2	globalr.c	RASCAR	Arctan of angle
ATAN2D	"	"	Same, but for float 64's.
madd	matrix.c	VMS	Matrix addition
msub	"	"	Matrix subtraction
mmul	"	"	Matrix multiplication
mscl	"	"	Matrix scalar multiply
midt	"	"	Matrix identity
mzer	"	"	Matrix zero.
mtrn	"	"	Matrix transpose
masg	"	"	Matrix assignment.
reduce	"	"	Gaussian elimination.
solve	"	"	Forward/backward substitution.
minv	matrix.c	VMS	Matrix inversion.
between	plot_off.c	VMS	Determine whether a value is between 2 values.
matherr	globals.c	"	Responses to math exceptions.
test_limits	c_graph.c	VMS	Test and set max and min values.

Function (System Management)	Module	System	Description
init	heap.c	ADG	Initialize a dynamic heap
alloc	heap.c	ADG	Allocate storage for a heap.
free	heap.c	ADG	Free allocated heap storage.
wait_for_event	play.c	VMS	Suspend task until event.
Multitasking primitives	exec.c	ADG	MTX kernel implementation.

Function (Graphics Processing)	Module	System	Description
Screen management	scrnapi.c	VTS	Viewport control and graphics primitives.
Menu management	menuapi.c	VTS	Menu control primitives.

Other Potentially Useful Function Categories

Data structure manipulation
- Stack package
- Queue package

List Processing
- Sorting and searching implementations

Appendix C: Subsystem Class Catalog

C.0.1 Format

This Appendix gives a detailed description of the classes used to compose the charting subsystem. For ease of access, descriptions are listed alphabetically. In order to really understand it, however, one must understand the assembly/subassembly relationships between the components. These are provided below. Another aspect to understanding these components is the inheritance relationships between them. When appropriate, these relationships are made explicit in the sections describing the affected components.

Descriptions of each class consist of up to six parts:

- A brief description of what the class is used for.
- The data that constitutes the class attributes.
- The access functions of this class.
- Overloaded functions, if any.
- Inheritance structure, if any.
- Notes discussing design issues or giving other references.

C.0.2 Assemblies and Subassemblies

The subsystem components can be regarded as five major assemblies, each of which is composed of several subassemblies. These subassemblies, in turn, can also be broken down into subassemblies. The list presented here is indented according to this assembly/subassembly relationship.

- Chart
  - Feature Data
  - Feature Instance List
    - Feature Instance List Iterator
    - Point List
      - Point List Iterator
      - Point
      - Transform
    - Projection
    - Region

- Device

- File
  - Conversion Table
  - Entity Instance
  - File Iterator
  - String

- Image
  - Chart
  - Display Lists
  - Point
  - Region
  - Transform
  - Window

- Window
  - Device
  - Region

C.1 Chart

The central repository for information about the chart to be displayed. Consists of projection information, points defining each feature to be displayed, and data on those features, such as color, fill pattern, line width, and line style. The Chart object is loaded directly from a chart file via the Load function. After that, any task that needs to can query the Chart object for information on any feature using the access functions listed below.

C.1.1 Data:

- *Projection* - Information needed to convert real-world (lat/lon) coordinates into paper chart coordinates (usually inches), and vice-versa.
- *Chart Extents* - Region defining chart max/min points.
- *Feature Instance List* - Has fields for feature code (key value) and list of feature instances.
- *Feature Data* - Contains data on how to display entities. Includes feature code (key value), color, line style, fill pattern, font style, text size, and display list id.

C.1.2 Functions:

- *Insert(Feature Type, Feature)* - .Insert a new instance of a feature into the appropriate list of the *Feature Instance List.*
- *Load(File)* - Load the given file into the database.
- *GetFeature(Feature Type, Point System, Target Region Size)* - Returns iterator on the list containing instances of the specified feature type. The *Point System* parameter is optional; if specified, returned points will be automatically converted into one of chart, lat/lon, or window coordinates. If not specified, they will be returned as chart coordinates. The *Target Region Size* parameter is only required if points are to be converted into window coordinates.
- *GetExtents()* - Returns region defining chart size and position.

- *SetExtents(Minimum point, Maximum point)* - Set opposite corners defining chart region.
- *SetExtents(Region)* - Set opposite corners defining chart region.
- *GetFillPattern(Feature Type)* -Return fill pattern of specified feature.
- *SetFillPattern(Feature Type, Fill Pattern)* - Set fill pattern of specified feature.
- *SetColor(Feature Type, Color)* - Set color of specified feature.
- *GetColor(Feature Type)* - Return color of specified feature.
- *GetProj()* - Return projection data.
- *LatLonToChart(Lat/Lon Point)* - Convert a point with each coordinate in radian units to chart units.
- *LatLonToChart(Lat Value, Lon Value)* - Convert to radian values to chart units..

C.2 Conversion Table

This class defines a simple lookup table used by the File Scanner to see if the incoming feature is defined to the chart subsystem.

C.2.1 Data

- *Array of Table Entries* - Each table entry is a structure with two data elements. One data element is a file feature code, the other is the corresponding subsystem feature code.

- *Max* - Maximum number of file features allowed. This number is used to tell the class constructor how much space to allocate for the array of table entries.

C.2.2 Functions

- *LookUp(File Feature Code)* - See if this file feature code has a corresponding subsystem feature code listed in the table. If so, return the subsystem code. Otherwise return Undefined Entity indicator.

C.3 Device

Serves as interface to device driver functions. Each type of display hardware will have a class derived from this one. These functions are used to implement window creation, shape drawing, and hardware display list manipulation. The only device drivers currently implemented are for X Windows running on either a monochrome or a color Sun.

C.3.1 Data

- *Device handle* - Uniquely identify this device.
- *Foreground Color*
- *Background Color*
- *Drawing Mode* - Two types are currently available, FOREGROUND and OVERLAY.
- *Resolution* - x-by-y pixel range.

C.3.2 Functions

- *Circle(Window, Center Point, Radius)* - Draw a circle of size *Radius* (in pixels) at the specified *Center Point* of the specified *Window*.
- *ClearWindow(Window)* - Erase entire contents of the specified *Window*.
- *Close(Window)* - Remove specified *Window* from the display screen.
- *CreateWindow(Region Definition)* - Create a new window with size and position specified by *Region Definition*.
- *Disconnect()* - Break connection with this device and destroy the device object.
- *DLClose()* - Close the hardware display list which is currently open. Only one can be open at a time.
- *DLDelete(Display List ID)* - Delete the specified display list.

- *DLOpen(Display List ID, Immediate Flag)* - Open the specified display list. If it does not exist, first create it. If *Immediate Flag* is set, then draw display list information on the screen as it is being written to the display list. Otherwise, just write the information to the display list without simultaneous display.

- *DLPresent()* - Return TRUE if this device supports hardware display lists.

- *DLReplay(Display List ID)* - Execute the list of commands stored in the specified display list.

- *Line(Window, Point List)* - Draw a line in the specified *Window. Point List* specified the endpoints of each line segment comprising the line.

- *Polygon(Window, Point List)* - Draw a filled polygon in the specified *Window*. The default fill pattern to use is SOLID.

- *SetBGColor(Color)* - Set the background color.

- *GetBGColor()* - Retrieve the background color.

- *GetResolution()* - Return the device's resolution.

- *SetResolution(Resolution Region)* -

- *SetColor(Color)* - Set the foreground color.

- *SetFillPattern(Fill Pattern)* - Set fill pattern of next shape to be drawn. Default is SOLID.

- *SetDrawMode()* - Set drawing mode to FOREGROUND or OVERLAY.

C.3.3 Notes

- Every device type will be implemented as a class derived from this class.

C.3.4 Inheritance Hierarchy:

$$\begin{tikzcd}
 & \text{Device} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Matrox} & \text{X Windows} & \text{VGA}
\end{tikzcd}$$

C.4 Display List

Allows features to be grouped for display. Display lists can either be real (implemented in hardware), or virtual. This class hides that information from the application and the rest of the subsystem.

C.4.1 Data

- *Display List ID* - Integer identifying this display list.
- *Virtual flag* - Set to TRUE if display lists are not implemented in hardware.
- *Immediate Flag* - Indicates whether list items should be displayed at the same time they are being added to the list.
- *Feature List* - Used if this display list is virtual. Consists of pointers to features, and pointers to display windows.
- *Current Entries* - Number of feature types currently defined to this display list.
- *Maximum Entries* - Maximum number of feature types this list can support.

C.4.2 Functions

- *GetID()* - Return the *Display List ID*.
- *Replay(Chart, Window)* - Execute list of commands in display list. If display list is real, this is simply a call to the device driver, passing it the display list id.
- *Append(Feature, Chart, Window)* - Re-open and add to this display list.
- *Open(Display List ID, Device, Immediate Flag)* - Create and open a new display list. Check if this device supports display lists, and set virtual flag appropriately.
- *Close(Window)* - Stop writing to current display list.

C.5 Entity Instance

Objects of this class are used to store unprocessed features as they are read off the chart file. The idea is for this class to be general enough to accommodate any sort of feature, whether strictly points or also text. Once an *Entity Instance* object is created, it is returned to the Chart's *Load* function where it is converted into the appropriate *Feature* object (each of which include a unique *Draw* function, which is why this step is needed).

C.5.1 Data

- *Color*
- *Fill*
- *Line Width*
- *Line Style*
- *Point List*
- *String*

C.5.2 Function

- *SetColor(Color)*
- *GetColor()*
- *SetLineWidth(Line Width)*
- *GetLineWidth()*
- *SetLineStyle(Line Style)*
- *GetLineStyle()*
- *GetFill()*
- *InsertPoint(Point)*

- *GetPoints()*
- *SetText(String)*
- *GetText()*

C.6 Feature

Serves as the base class for all features, and consists of virtual function definitions. Each class derived from Feature inherits these functions and defines them as appropriate.

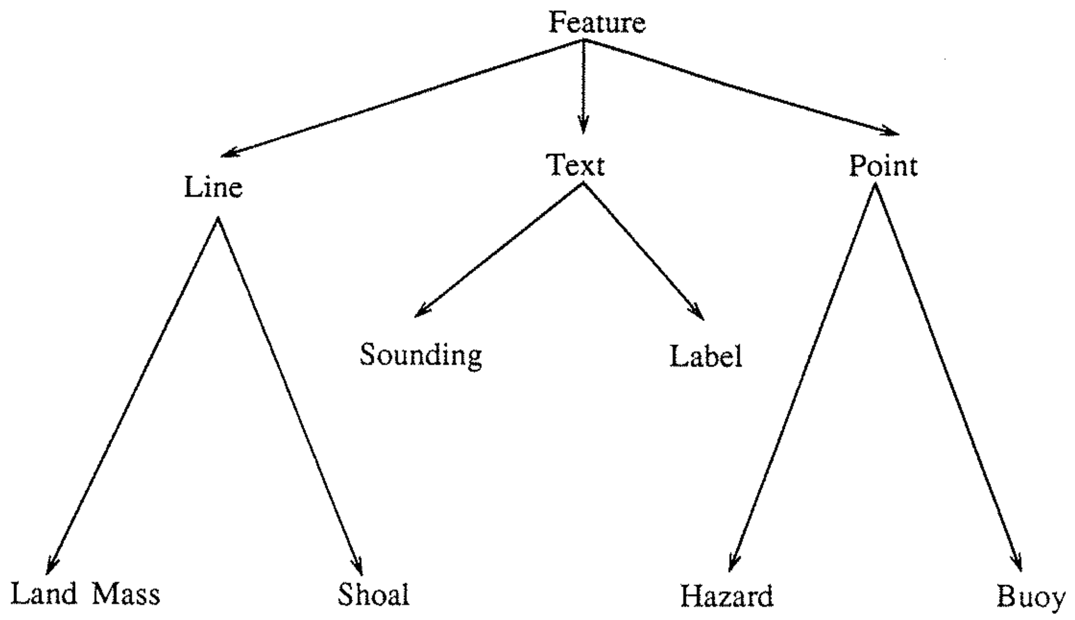
C.6.1 Data

- *Point List* - List of points which define this feature.
- String - Used by *Text Features*.

C.6.2 Functions

- *Get*() - Returns set of points comprising feature. Points are given in chart coordinates.
- *GetText()* - Return string defining a *Text Feature*.
- *Draw(Window)* - Convert set of points to the appropriate shape (in window coordinates) and return.
- *Insert(Point)* - Inserts new *Point* into the *Point List* defining this feature.
- *Set(Point List)* - Set entire *Point List* for this feature at once, rather than one point at a time as is done by the *Insert* function.
- *SetText(String)* - Initialize string defining a Text Feature.
- *SetXform(Transform)* - Initialize the point transformation class for the *Point List* defining this *Feature*.

C.6.3 Inheritance Hierarchy



C.7 Feature Data

Information describing display characteristics of a particular feature type. This object is allocated for each feature type. Collections of these are kept in a table, and keyed according to feature type. All access functions on this object either set the class's attribute data or return, so no additional description is needed here.

C.7.1 Data

- *Color* - Enumerated type.
- *Fill* - Enumerated type; can be SOLID, STIPPLED, CROSSHATCHED, or UNFILLED.
- *Line Style* - Enumerated type. Most likely values are DOTTED, DASHED, or SOLID.
- *Line Width* - Enumerated type. Values can be THIN, MEDIUM, or WIDE.

C.7.2 Functions

- *GetColor()*
- *GetFill()*
- *GetLineStyle()*
- *GetLineWidth()*
- *SetColor(Color)*
- *SetFill(Fill)*
- *SetLineStyle(Line Style)*
- *SetLineWidth(Line Width)*

C.8 Feature Instance List

List of features of a particular type. This class abstracts the specific manner in which features are stored. Currently, feature storage is implemented as an array. This class can be used directly to insert new feature instances, but retrieval of features must be done via a *Feature Instance Iterator* to avoid concurrent access problems.

C.8.1 Data

- *Feature List* - List of chart features. This is currently implemented as an array of pointers to elements of type *Feature*.
- *Space Increment* - Amount of additional space to allocate during list expansion.
- *Max* - Maximum number of elements this list can accommodate.
- *Next Element* - Reference to the next element in the list.

C.8.2 Functions

- *Insert(Feature)* - Put a new feature on the end of the *Feature List*.
- *GetSize()* - Return the current number of elements in the *Feature List*.

C.9 Feature Instance Iterator

Objects of this class are used to navigate through *Feature Instance Lists*. The iterator maintains a record of where it currently is in the list, and returns the next list element whenever the '++' operation is invoked. Each task accessing a particular list will have its own iterator on that list to work with; this avoids concurrent access problems.

C.9.1 Data

- *Feature Instance List* - Specific list object assigned to this iterator. This assignment is made by the user when invoking the iterator's class constructor.
- *Index* - Next list item to return.
- *Transform* - Transformation to perform on points comprising feature.

C.9.2 Functions

- *SetXform(Transform)* - Initialize the transformation to perform on the points comprising the features in the list being iterated.

C.9.3 Overloaded Operators

- '++' - Return next feature in the list.

C.10 File Iterator

Determines which format a given file is in, creates the appropriate file scanner object, and returns each feature in the file to the using function, one feature at a time.

C.10.1 Data

- *File Name* - String indicating file's name and path.
- *File Scanner* - Object allocated by this object to search through file looking for each entity.

C.10.2 Functions

- *GetProjection()* - Return chart projection information to the using function. Projection information must be processed separately from feature information; the manner in which the two types of information are used are radically different from each other.

C.10.3 Overloaded Operators

- '++' - Return next feature in file.

C.11 File Scanner

Scans through the file, looking for entity identifier records. When it finds one, it determines if this entity type is of interest to the chart by comparing its feature code to the code listed in the Scanner's *Conversion Table*, If so, it sends the entire entity back to the file iterator, which forwards it to the calling function. Otherwise, the entity is ignored and scanning continues to the next entity.

C.11.1 Data

- *File Descriptor* - Pointer to input file. This is currently implemented as a C++ *istream.*
- *Conversion Table* - Table listing entity codes as they appear in the file and the corresponding internal subsystem codes.
- *Buffer* - Buffer for current line of file.
- *Entity Instance* - Current entity being read from the file.
- *Entity Count* - Total number of entities in this file.
- *Projection* - Abstract data type for projection information.

C.11.2 Functions

- *atoll(Char Buffer, Type, Coordinate)* - Friend function which converts alphabetic latitude/longitude coordinates into floating point numbers.
- *GetProjection()* -
- *InitProjection()* - Scan file for projection information and initialize *Projection* abstract data type.
- *Next()* - Scan file for next feature occurrence, returning only those entities which are defined to the subsystem.
- *SkipHeader*() - Ignore file header information.
- *Readln*() - Read the next line of the file into a buffer.

C.11.3 Notes

- Operations will need to be added later to allow for file updating.

C.12 Image

Associate a specific chart object with a particular window object. The Image class is responsible for display list manipulation and for ''special effects'', such as zooming and offsetting.

C.12.1 Data

- *Chart*
- *Window*
- *Maximum # of Display Lists*
- *Current # of Display Lists*
- *Current Chart Extents*
- *Current Chart Center*
- *Transform* - Chart-to-window transformation abstract data type.
- *Base Zoom* - Zoom factor to use when chart is first displayed; normally initialized to zero.
- *Current Zoom* - Default zoom factor. Current default is .20.
- *Base Offset* - Offset factor to use when chart is first displayed; current default is 0.
- *Current Offset* - Default offset factor. Current default is .10.
- *Display List Table* - Table of display list entries, indexed by display list id.

C.12.2 Functions

- *Append(Display List ID, Feature Type, Immediate Flag)* - Add a new feature type to the specified display list.
- *Load(List ID, Feature*, Window)* - Load the specified display list with the specified feature such that it displays on the specified window.
- *Load(Chart*, Window)* - Associate the specified chart with the specified

window.

- *Display(List ID)* - Replay specified display list.
- *DisplayAll()* - Replay all display lists in the table.
- *GetWindow()* - Return window object.
- *GetChart()* - Return chart object.
- *ChartToWindow(Chart Point)* - Convert the specified chart point to its corresponding window point.
- *Zoom()*
- *Pan()*
- *ChangeCenter(New Center Point)*
- *ZoomCalc()* - Calculate extents of new chart image. This function is called by both Zoom and Pan.

C.13 Point

This will serve as the base class for other, more specific, types of points. This class provides the basic point arithmetic operators.

C.13.1 Data

- *x value* - x value on Cartesian plane.
- *y value* - y value on Cartesian plane.

C.13.2 Functions

- *GetX()* - Return x-coordinate of point.
- *GetY()* - Return y-coordinate of point.

C.13.3 Overloaded Operators

- '+' - Add the x values and y values of two points and return the resulting point.
- '-' - Subtract the x values and y values of one point from another.
- '*' - Multiply a point by either a scalar or another point. Thus this operation is overloaded twice.
- '/' - Divide a point by either another point or by a scalar. This operation is also overloaded twice.
- '=' - Set one point type equal to another. If the left-hand-side is of a different point type, first transform the right-hand-side to the new type.
- '==' - Check if two points are equivalent.
- '<' - TRUE if the x-coordinate of the lhs point is less than the x-coordinate of the rhs point, and same for the two y-coordinates.
- '>' - Greater-than, which works the same as '<'.
- '!=' - TRUE if either the x or y coordinates of two *Points* are not equal.

C.14 Point List

Points used to define a chart feature usually have to be grouped. The *Point List* class provides a convenient way of achieving this grouping without point users having to worry about whether they are being grouped in an array, a linked list, or some other kind of structure.

C.14.1 Data

- *List of points* - This is currently implemented as a dynamically expandable array.
- *Space Increment* - Amount of extra array space to allocate when maximum has been reached.
- *Max* - Current maximum array size. This increases by *Space Increment* units each time the array is expanded.
- *Next Element* - Index of next empty slot in the array.
- *Transform* - Coordinate transformation to apply on points as they are being read.

C.14.2 Function

- *Insert(Point)* - Add a new point to the point list.
- *SetXform(Transform)* - Initialize the *Transform* attribute.
- *GetXform()* - Return Transform attribute.
- *GetSize()* - Return current number of elements in *Point List.*

C.15 Point List Iterator

In order to retrieve data from a *Point List*, the user must first allocate a *Point List Iterato*r to serve as the interface. As in the case of the *Feature Instance List*, this abstracts the details of the *Point List's* implementation, and also allows for concurrent access.

C.15.1 Data

- *Point List* - Point List on which iteration is being performed. This is defined by the user via the class's constructor.
- *Index* - Current element in list.

C.15.2 Overloaded Operators

- '++' - Yield next element in the list.

C.16 Projection

The *Projection* class is associated with a particular chart, and is used to transform points from real-world coordinates to chart coordinates, and vice-versa. At the moment, only Mercator projection transformations are supported.

C.16.1 Data

- Type of projection
- Center Llatitude and Longitude.
- Chart Scale
- False Northing and Easting
- Flattening Inverse
- Semi-major Axis
- Latitude and Longitude of Chart Scale

C.16.2 Functions

- *LatLonToChart(Point)* - Convert a point in radians to its corresponding chart coordinate.
- *LatLonToChart(char*, char*)* - Convert a real-world coordinate given in degrees, minutes, and seconds to its corresponding chart coordinate.
- *ChartToLatLon(Point)* - Convert a point in chart coordinates to its corresponding real-world coordinate in radians.
- *InitFalse()* - Initialize false northing and easting data.

C.17 Region

Two points needed to define a rectangular region. It is assumed that the minimum point is the lower-left-hand corner and the maximum is the upper-right, but this assumption is kept deliberately vague.

C.17.1 Data

- *Min* - Minimum point of rectangle
- *Max* - Maximum point of rectangle

C.17.2 Functions

- *SetMin(Point)* - Set lower left point.
- *SetMax(Point)* - Set upper right point.
- *GetMin()* - Return lower left point.
- *GetMax()* - Return upper right point.

C.17.3 Overloaded Operators

- '=' - Set one *Region* equal to another.

C.18 String

Provide functions for making string handling more intuitive than those provided by C.

C.18.1 Data

- *Character Buffer* - n-byte character array.
- *Reference Count* - Keeps track of the number of times this string is being referenced. This avoids creating duplicate strings. When the count becomes 0, the string object is destroyed.

C.18.2 Functions

- *GetChars()* - Return the buffer comprising this string.
- *Length()* - Return the number of characters in this string.

C.18.3 Overloaded Operators

- '+' - Concatenate two strings.
- '=' - Copy one string to another.
- '==' - Return TRUE if two strings are typographically equivalent, FALSE otherwise.
- '<<' - Stream a *String* type to standard output.

C.18.4 Notes

- The *String* type can be useful when processing chart file and path names.
- Stroustrup discusses strings on p. 184.
- Dewhurst and Stark discuss strings on p. 80.

C.19 Transform

The Transform class bundles together the data and functions needed to convert a point from one coordinate system to another.

C.19.1 Data

- *Factor* - Amount by which to multiply point.
- *Offset* - Amount to add or subtract to point.

C.19.2 Function

- *SetXform(Source Region, Target Region)* - Calculate the *Factor/Offset* pair for this transformation given the source and target regions.
- *XformOut(Point)* - Transform the specified point to the new coordinate.

C.20 Window

Define a window's size and position, assign it to a device, and draw features on it.

C.20.1 Data

- *Device* - Device this window is assigned to.
- *Window ID* - Identifier of this window; used by device.
- *Extents* - Region defining window size in pixels.

C.20.2 Functions

- *Clear()* - Erase everything in the window.
- *Close()* - Remove window from display and destroy it.
- *GetDevice()* - Return identifier of device this window is defined to.
- *SetExtents(Extents)* - Initialize maximum/minimum coordinates.
- *SetExtents(Point, Point)* - Same.
- *GetExtents()* - Return window region.
- *Open(Device)* - Set *Device* field, and open window on that device.
- *DrawLine(Array of Points)*
- *DrawPolygon(Array of Points)*
- *DrawPoint(Point)*
- *DrawCircle(Point, Radius)*
- *DrawText(Point, Text)*
- *SetFillPattern(Fill)* -
- *SetColor(Color)* -
- *SetDrawMode(Draw Mode)* -

C.20.3 Overloaded Operators

- '<<' - Display data of some type on this window. Type can either be a display list or a shape.

References

[AgM88] W. Agresti and F. McGarry, in *The Minnowbrook Workshop on Software Reuse: A Summary Report*, Computer Sciences Corporation, March 1988, 17.

[BaT75] V. R. Basili and A. J. Turner, Iterative Enhancement: A Practical Technique for Software Development, *IEEE Transactions on Software Engineering SE-1*,4 (December, 1975), 390-396.

[Boo86] G. Booch, Object-Oriented Development, *IEEE Transactions on Software Engineering SE-12*,2 (February, 1986), 211-221.

[BAB88] B. A. Burton, R. W. Aragon, S. A. Bailey, K. D. Koehler and L. A. Mayes, The Reusable Software Library, in *Tutorial: Software Reuse: Emerging Technology*, W. Tracz (editor), Computer Society Press of the IEEE, 1988, 129-137.

[Car87] R. Carle, Reusable Software Components for Missile Applications, *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse*, Blue Mountain Lake, New York, July 1987.

[Cle89] L. Cleveland, A Program Understanding Support Environment, *IBM Systems Journal 28*,3 (1989), 324-344.

[Dav88] A. M. Davis, A Comparison of Techniques for the Specification of External System Behavior, *Communications of the ACM 31*,9 (September 1988), 1098-1115.

[FrN87] W. B. Frakes and B. A. Nejmeh, An Information System for Software Reuse, *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse*, Blue Mountain Lake, New York, July 1987.

[FrG89] W. B. Frakes and P. B. Gandel, Representing Reusable Software: Survey and Model, *Unpublished Manuscript*, 1989.

[Hof79] D. R. Hofstadter, in *Godel, Escher, Bach: An Eternal Golden Braid*, Vintage Books, 1979.

[HoM84] E. Horowitz and J. B. Munson, An Expansive View of Reusable Software, *IEEE Transactions on Software Engineering SE-10*,5 (September 1984), 477-487.

[JeS89] C. Jette and R. Smith, Examples of Reusability in an Object-Oriented Programming Environment, in *Software Reusability: Applications and Experience*, T. J. Biggerstaff and A. J. Perlis (editors), ACM Press, 1989, 73-102.

[Jon84] T. C. Jones, Reusability in Programming: A Survey of the State of the Art, *IEEE Transactions on Software Engineering SE-10*,5 (September 1984), 488-494.

[Loh89] M. C. Lohrenz, The Digital World Vector Shoreline Database: Current and Future Capabilities, *Proceedings of Symposium '89 - Defense Mapping Agency Systems Center*, Herndon, Virginia, May, 1989, 113-122.

[MMM87] *Version Description Document for the Missile Software Parts of the Common Ada Missile Packages (CAMP) Project*, McDonnell Douglas Astronautics Company, 1987.

[Mey89] B. Meyer, Reusability: The Case for Object-Oriented Design, in *Software Reusability: Applications and Experience*, T. J. Biggerstaff and A. J. Perlis (editors), ACM Press, 1989, 1-34.

[Nei81] J. Neighbors, *Software Construction Using Components*, PhD Thesis, University of California, Irvine, 1981.

[Nei84] J. Neighbors, The Draco Approach to Constructing Software from Reusable Components , *IEEE Transactions on Software Engineering SE-10*,5 (September 1984), 564-574.

[Obl89] E. J. Obloy, The Liability of the Electronic Chartmaker for Negligent Charting - Update, *Proceedings of Symposium '89 - Defense Mapping Agency Systems Center*, Herndon, Virginia, May, 1989, 69-86.

[Osk89] O. Oskarsson, Reusability of Modules with Strictly Local Data and Devices - A Case Study, in *Software Reusability: Applications and Experience*, T. J. Biggerstaff and A. J. Perlis (editors), ACM Press, 1989, 143-155.

[Par72] D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM 15*,12 (December, 1972), 1053-1058.

[Par76] D. Parnas, On the Design and Development of Program Families, *IEEE Transactions on Software Engineering SE-2*,1 (March, 1976), 1-9.

[PCW89] D. L. Parnas, P. C. Clements and D. M. Weiss, Enhancing Reusability With Information Hiding, in *Software Reusability: Concepts and Models*, T. J. Biggerstaff and A. J. Perlis (editors), ACM Press, 1989, 141-157.

[PrF87] R. Prieto-Diaz and P. Freeman, A Software Classificaton Scheme for Reusability, in *Tutorial: Software Reusability*, P. Freeman (editor), Computer Society Press of the IEEE, 1987, 106-116.

[Pri90] R. Prieto-Diaz, Domain Analysis: An Introduction, *ACM Software Eng. Notes 15*,2 (April, 1990), 47-54.

[RiW88] C. Rich and R. C. Waters, The Programmer's Apprentice: A Research Overview, *IEEE Computer 21*,11 (November, 1988), 11-25.

[Sha89] M. Shaw, Larger Scale Systems Require Higher-Level Abstractions, *Proceedings of the Fifth International Workshop on Software Specification and Design* , Pittsburgh, Pennsylvania, May, 1989.

[SWC88] R.Simoncic, A.C. Weaver, B.G. Cain and M.A. Colvin, SEANET: A Real- Time Communications Network For Ships, *International Conference on Mini and Microcomputers*, Miami Beach, FL, December 1988.

[SoE89] E. Soloway and K. Ehrlich, Empirical Studies of Programming Knowledge, in *Software Reusability: Applications and Experience*, T. J. Biggerstaff and A. J. Perlis (editors), ACM Press, 1989, 235-267.

[Ste87] O. Stene, The Electronic Chart Test Bed Activities in the North Sea , *Unpublished Memorandum*, 1987.

[Str87] B. Stroustrup, in *The C++ Programming Language*, Addison-Wesley, 1987.

[TyW89] T. W. Tyler and D. M. Weiss, A Tool For Family-Based Reuse, *AIAA Computers in Aerospace VII Conference*, October 1989.

[Weg87] P. Wegner, Varieties of Reusability, in *Tutorial: Software Reusability*, P. Freeman (editor), Computer Society Press of the IEEE, 1987, 24-38.

[Weg89] P. Wegner, Capital-Intensive Software Technology, in *Software Reusability: Concepts and Models*, T. J. Biggerstaff and A. J. Perlis (editors), ACM Press, 1989, 43-97.

[Wyb90] N. Wybolt, Experiences with C++ and OOSD, *ACM Software Eng. Notes 15*,2 (April, 1990), 31-38.