HEADING OFF TRAGEDY:
USING DESIGN PATTERNS AGAINST A MOVING TARGET

Kim G. Woodward
DCS Corporation
1330 Braddock Place
Alexandria, VA 22314
(kwoodwar@dcscorp.com)

presented at The Second World Conference on
Integrated Design and Process Technology on
December 3, 1996

ABSTRACT:

Most designs have a bit of sanity in their development. An orderly analysis followed by a design and then a blessing by the customer prior to coding is often considered the norm. But what about a design for a moving target, something that has not been fully defined and that will continue to change well after your design is implemented? A proposed usage of design patterns along with some common sense rules of designing for change is described to ensure that a minimal set of changes will need to be made in such a situation. A GUI design for a Human Factors simulator for a not completely defined system is given as an example.

I. THE PURPOSE (WHY WE ARE HERE)

This paper is being written because software designers often find themselves in dire straights because of changing requirements outside of their sphere of influence. I work in defense style systems which are more rigidly fixed in terms of their requirements than most commercial systems will ever be. However, there have been times when a full analysis of requirements could not be made due to time constraints, a sudden change in the operation of the system is needed, or IWBNI (It Would Be Nice If) takes over the minds of those who pay the bill. Since the customer is always right you might find yourself alerting the customer to the danger of the change, but you end up doing the change anyway. In any case, a variant of Murphys Law, that I like to use, says that the more closely you have defined the requirements for a system the more requirements you discover that have not been defined. This is known as "The More You Know - The Less You Know" syndrome of analysis.

Just for comparison and contrast, I will be using the description of a software development project that I worked on that had some special changing requirements, as an example. This project was for the development of a GUI simulator for the E-2C Advanced Computer Information System (ACIS) implementation. The E-2C is the U.S. Navy AEW (Airborne Early Warning) aircraft comparable to the USAF AWACS in function. The GUI simulator was to be incorporated into a Human Factors simulator for use in determining how best to spread the operator workload around the aircrew. The peculiar requirements spread for this system is that it was currently in development and would be in development long past the end of our GUI simulator contract. Hence, the requirements would never stay still long enough for us to peg them down. A waterfall approach to development of the functionality was obviously out of the question, because we would not be able to get past the analysis stage. The initial design documents that we obtained indicated that there were more than one hundred separate screens to implement, as well as interfaces to hardware items that had not been fully developed yet. Performing individual functions was not necessarily limited to one way of implementation, and we had no way of knowing if the functions would suddenly be required to be activated by hardware as opposed to software. It was decided that the least risky development effort would be to implement the project in a series of "Builds". Each "Build" would have a unique functionality set and would build on the previous version. Design and implementation would be incorporated into each build. This would lessen the impact of a change downstream. But still the basic design for the system would need to be somewhat stable.

II. THE PREDICAMENT (THE THINGS YOU FIND YOURSELF IN)

The customer has the desire and vision for a certain product and the systems analyst in all of us kicks in to discover exactly what he/she means. Sometimes we get lucky and actually get a good handle on what the customer desires and how the product should look and

© 1996 Society for Design and Process Science

act. Sometimes the customer doesn't even know what he/she wants and it is up to us to put on the Systems Analyst hat and lead him/her by the hand in defining the system. But even at the end of our analysis we find ourselves at the mercies of change. Change WILL happen - you cannot avoid it. Systems which are fixed are doomed to failure because they cannot keep up with the changing desires or needs of the customer. Our example project is not much different, our customer in this case needed to follow the requirements definition of an external agency. His requirements, therefore, were being driven by outside influences which in turn filtered down to our project.

III. THE PONTIFICATIONS (SOAPBOX TIME)

When we find ourselves in these predicaments we usually panic, start polishing our resumes, roll up into a fetal position, or all three. But we should not feel like it is the end of the world. After all, we are engineers and we work with an abstract "thingy" called software that has the amazing ability to look and feel like many different things. Our difficulty is only in knowing how to apply ourselves to the problem of Good Design. A Good Design will allow for the changes that will occur. A Good Design will minimize the changes that need to be done to itself in order to achieve those changes. A Bad Design sometimes is not obvious but it does have some notable characteristics:

"Bad designs aren't bad designs until they are proven to be bad designs. And I mean proven with hard evidence of serious detrimental behavior. {There are} three criterion for this. 1. A bad design is rigid. i.e. it is very difficult to make the kinds of changes that customers need, because those changes affect the fabric of the entire application. The more of the application that is affected by a change, the more rigid the design is, and the poorer the design. 2. A bad design is fragile. Changes made to one part of the application frequently cause failures in other parts of the applications. The higher the probability that a change will result in a failure, the more fragile the application is, and the poorer the design. 3. A bad design is not reusable. 'Desirable' parts of the applications depend upon 'undesirable' parts to the extent that it is safer and less costly for engineers to recreate the desirable functionality than to try to reuse it from this application. The less of the application {that} can be reused by other applications, the poorer the design. (Martin 1996)"

IV. THE PRINCIPLES (HERE IS WHAT YOU DO)

Object-Oriented Design helps with these problems but even it needs to be tailored to be helpfull. We need to avoid a huge amount of redesign when changes come our way. So what are some of the areas that might require redesign within a Bad OO Design in order to allow change to occur? In Gamma (1995), pg 24-25, the following eight common problems which will cause redesign are espoused: (1) Creating an object by specifying a class explicitly. (2) Dependence on specific operations. (3) Dependence on hardware and software platform. (4) Dependence on object representations or implementations. (5) Algorithmic dependencies. (6) Tight coupling. (7) Extending functionality by subclassing. (8) Inability to alter classes conveniently. These difficulties can be circumvented by the use of Design Patterns and Frameworks.

Putting a complete system together requires some form of software architecture in which the program can be set in "mushy concrete" (remember, this thing is going to change). That "mushy concrete" must be flexible enough to allow for change to occur in how the overall system operates. The progamming strategy that is generally used for this type of architecture is the framework. Since non-reuse is one of the "Bad Design" characteristics, the framework that is chosen for the design must be closely reviewed for its characteristics.

"An architectural framework is a reusable design of a program or a part of program expressed as a set of classes. It is often domain specific and it can be considered an object of domain-level reuse. The MVC Paradigm is one of many examples of a framework. A framework consists of classes that collaborate and {has} detailed instructions on how to use and tailor these classes to meet other user needs. Hence, they consist of not just code so they are more abstract than most software, making them more difficult to document.(Palepu 1995)" "Frameworks provide a unifying context that implements some portion of the architecture of an application. (McGregor 1996)"

Strangely enough, reuse and change tend to go hand in hand. If you cannot reuse portions of the existing system, any change to the system will require a great amount of rework. Great is obviously a relative term here. "Reuse of a few relatively sophisticated components, such as frameworks, typically provides more leverage than reuse of many primative components. A single sophisticated component can deliver a larger amount of functionality for the amount of learning required than can a set of unrelated or weakly related components. (McGregor 1996)"

"Object-oriented frameworks represent the scaling up of the fundamental principle of object-oriented programming in that, due to the features of inheritance and dynamic binding, a new class can be implemented by providing only what is different in this class compared to one which already exists. With object-oriented frameworks, the same principle is applied to whole applications or sub-systems, so that the effort to develop a

© 1996 Society for Design and Process Science

new application is proportional to the difference between the particular application and that in the framework. (Ohlsson 1996)"

This difference between the new application and the framework is what gives us our ability to change. If we can keep the level of modification down to a new class, that inherits from other classes to do the specifics of what we want to do, we are that much closer to a solution to our change problem. Containment of change is the means to resolving our delima.

Design Patterns afford you other peoples experience. Design Patterns, sometimes called Pattern Languages, are written solutions for a specific type of problem which may occur in multiple domains. They are a new item in the realm of Object-Oriented Design and are becoming increasingly popular as means to solve some of the most nightmarish problems that exist. Design Patterns used in conjunction with Frameworks serve to avoid the three problems of Bad Design mentioned above: rigidness, fragility, and non-reusability. Even the subject of designing software for modification has been addressed in a design pattern by Brian Foote and Joseph Yoder in Vlissides (1996).

"Patterns are not code reuse; they are design reuse. As such, they can be used to solve similar problems that arise in very dissimilar, completely unrelated systems. Code reuse, on the other hand, has been found to be most applicable across closely related systems within a particular problem domain. (Barnes 1996)"

V. THE PREMISE (WHERE WE ARE HEADED)

To contain the effects of change: (1) concentrate the early stages of design on defining a minimal subset of requirements that are not going to change very much. (2) Design a framework around these requirement subsets with the emphasis on flexibility. Then (3) by using design patterns develop a mechanism to allow the encapsulation of functions such that the functions are easily removed or moved.

VI. THE PROCESS (STEP BY STEP)

The following steps outline the way in which software could be designed to minimize the impact of changes.

A. IDENTIFY THE MINIMAL SUBSET OF REQUIREMENTS FIRST

Some form of requirements definition must be accomplished in order to proceed. Since we dont know all of the requirements we first concentrate on the requirements that we do know. Actually we concentrate on a subset of the requirements that we know will not change appreciably. "Identification of the potentially desirable subsets is a demanding intellectual exercise in which one first searches for the minimal subset that might conceivably perform a useful service and then searches for a set of minimal increments to the system. (Parnas 1979)" Some of these subsets will provide the definition of the requirements for the framework defined in step B. Others will identify functional components that will be encapsulated in step D. Still others will identify the connecting specifications for the patterns defined in step C. It is necessary to view the resulting design as an attempt to build a software family in which minimal changes need to be made in order to add or delete functionality.

"Identifying the minimal subset is difficult because the minimal system is not usually a program that anyone would ask for. If we are going to build the software family, the minimal subset is useful; it is not usually worth building by itself. Similarly, the maximum flexibility is obtained by looking for the smallest possible increments in capability: often these are smaller increments than a user would think of. Whether or not he would think of them before system development, {we are} likely to want that flexibility later. (Parnas 1979)" This step is ongoing during the life cycle of the project. This is where we determine the increments which must be made to our project to meet the ever changing requirements.

In our example, the E-2C ACIS simulator had a requirement in that it was being implemented in Motif (running on Unix) and that the communications between processors would be via TCP and UDP. In addition, the functionality would need to interface with a 3D design package which was controlled by shared memory. There was a desire to not need to specifically know where individual processes were in the system because of logging and control requirements. These requirements were used to define the framework design indicated in step B.

B. DESIGN A FLEXIBLE FRAMEWORK SECOND

In the paper A Pattern for Reuse (McGregor 1996), it is proposed that Architectural Reuse Guide Component Reuse. The steps to achieving this end are described as follows: "(1) Develop an architecture, ... (2) Define a pattern language that describes the architecture, ... (3) Associate Components with Specific Design Patterns, ... and (4) Construct frameworks that instantiate the architecture. (McGregor 1996)" Note that the architecture is formulated from design patterns which in turn control how that frameworks are instantiated. Once we know the requirements subsets we can pick some of them to define the framework. Look for the obvious in choosing the subsets as to whether the subset will go away once the initial part of the system is designed.

© 1996 Society for Design and Process Science

Our example project relied heavily on the use of frameworks built from existing design patterns and class libraries peculiar to its operating environment. The operating environment for the E-2C ACIS was Motif running in a Unix environment. It was decided that the look-feel requirement would best be resolved by using a Motif application generator. This would provide us with an easy way to change the widows, popups, and pulldowns as we encountered change. This constituted the first part of the framework. The second part of the framework definition came from the need to interface with the 3D design package controllable from shared memory and the decision to encapsulate the functionality in several processes. It was determined that the best way of accomplishing this end was to use the Pipes and Filters Architecture design pattern described by Regine Meunier in Coplien (1995). "The Pipes and Filters Architecture, can be used to structure applications that can be divided into several independent, complete subtasks performed in a strongly determined sequential or parallel order. (Coplien 1995)"

C. SEARCH FOR ANSWERS IN PATTERNS LAST

"New and/or inexperienced designers need guidance in creating designs based on reusable components. Reuse is more likely to occur if the design being implemented is a standard approach to a known problem. It is not enough simply to present the designer with a user's guide of reusable components. The designer needs to understand any architectural concepts that underlie the reusable components. (McGregor 1996)" Since Design Patterns are written down and have a clear definition of their use, there is less of a chance that they will be used mistakenly when specific requirements are being addressed. "Patterns provide the linkage between the architecture-level components and the appropriate code-level components, thus resolving the need to locate appropriate reusable units. (McGregor 1996)"

In our example, the design for the pipe, in which information would be moved between the filters (actually individual processes), was determined to be accomplished by the Broker design pattern described in Buschmann (1996). The Broker design pattern provides a means of isolating client and server processes. It accomplishes this by use of the broker process which is the only process which knows where each of the client and server processes reside, either in the same processor or in a different processor. The client and server register with the local Broker process and then use named service calls for the action. Since the design pattern provided Client and Server classes, it was only a matter of inheriting either of the classes to get to the functions that it provided. The general way in which information would pass between processes in a single processor was decided to be via message queues through the local Broker. Therefore the Broker design pattern would need to use a Bridge(151) pattern (see Gamma 1995, pg 151 for details) to "decouple an abstraction from its implementation so that the two can vary independently. (Gamma 1995)" This is a fancy way of saying that we needed to be able to switch between TCP, UDP, and message queues transfers at runtime. A mechanism for reacting to incoming messages was needed for each of the processes and was fullfilled by the use of a modified Blackboard design pattern as described in Buschmann (1996). The Blackboard design pattern is “a collection of independent programs that work cooperatively on a common data structure. Each program is specialized for solving a particular part of the overall system task. ... These specialized programs are fully independent of each other. (Buschmann 1996)” The modification to the pattern allowed the Blackboard design pattern to have a deterministic task completion. The resulting pattern is similar in definition to the Reactor design pattern described by Douglas C. Schmidt in Coplien (1995). In order to provide the capability of the modified Blackboard to be used as a server an Adapter(139) design pattern (see Gamma 1995, pg 139 for details) was used to couple these portions of the design via inheritance. The Adapter pattern class was modified to include provision to use services from the server to obtain the next message in the message queue and place it into the Blackboard pattern data area. In order to easily add or delete functionality, all functions were encapsulated along with inheritance to the Knowledge Source portion of the Blackboard design pattern. The Adapter coupled the Controller portion of the Blackboard pattern with the Server class of the Broker pattern. We used an Abstract Factory(87) design pattern (see Gamma 1995, pg 87, for details) to determine which derived classes, of the Knowledge Source, would be used in each server implementation. Once these design patterns were utilized it became an easy process of adding functionality by: (1) encapsulating a function with a class derived from the Knowledge Source, and (2) defining a class derived from the Adapter with a method which modifies the Abstract Factory creator to make an object of the function class and attaching it to the list in the Controller portion of the Blackboard pattern. The mechanism built into the Blackboard pattern would then call the function object for it to perform its representative function at the proper time. Each of the processes used in the example were simply main programs which called objects of the (2) class above.

D. ENCAPSULATE, ENCAPSULATE, ENCAPSULATE

"Lack of generality is necessary to make the program as efficient or inexpensive as possible. If {the

© 1996 Society for Design and Process Science

designer} must develop a family of products, he tries to isolate the changeable parts in modules and to develop an interface between the module and the rest of the product that remains valid for all versions. The crucial steps are as follows: (1) Identification of the items that are likely to change. These items are termed "secrets." (2) Location of the specialized components in separate modules. (3) Designing intermodule interfaces that are insensitive to the anticipated changes. The changeable aspects or "secrets" of the modules are not revealed by the interface. It is exactly this that the concept of information hiding, encapsulation, or abstraction is intended to do for software. (Parnas 1979)"

"The special implications for our problem are simply that, as far as possible, even the presence or absence of a component should be hidden from other components. (Parnas 1979)"

E. WRITE IT DOWN

The first rule of Systems Engineering is that if it has not been written, it has not been said. This should also be the creed of Software Engineering as well. Not only does it allow you to have data to back up your premise that something was promised but it also helps in those times when you are trying to rack your brain hoping to remember why you decided to use one method versus another in the design in the first place. Having written down the rejected and the accepted components of the design, where you got them from and why will go a long way in keeping you from loosing your mind when the time comes to reuse what you have designed. It will also serve to help you not make stupid mistakes in altering a design that would prevent certain actions from taking place.

It is also necessary to write down the evolving requirements list as it becomes known during the development. This includes cross references to subsets which define how portions of the framework and associated design patterns fit into the overall solution. Keep the notes up to date, don't erase old concepts but make notations to state that they are out of date. It should always be possible to go to the software design notebook and answer the questions WHY and HOW.

F. USE YOUR TOOLS

Some of the tools available today for software engineers have made our lives easier. Application generators, CASE tools, CM tools and the like can be used in various portions of the project during the life cycle to keep a handle on the current state of the requirements as implemented in the system. For instance, the design for the example was done using Rational Rose/C++. Not only can a design be produced with accompanying documentation in the Booch methodology but a shell of the program can be produced by the same tool. The shell can be filled in as the development goes along. However, if we only use the case tools in the beginning and not throughout the life of the project the "currentness" of the design will start to escape us (see step G below).

G. KEEP THE DESIGN CURRENT

Since we are attempting to implement something that is constantly changing, we need to know the current state of the design at any one moment. The design should be made available to those code implementers in the group in hardcopy or in a design repository that can be easily accessed. In addition, it is always nice to tell the implementers that a certain change will affect their part of the project pie. A lot of needless effort may be averted by this simple act of communication. A good way of doing this is with the use of email. Email them when you start a design change to tell them what components will be affected. Email them if the affected components change and then email them when you are about to modify the design book, so that they will be looking for the design change. CC the project manager, if he/she is not you.

H. FORCE CHANGES TO FOLLOW A RULEBOOK

In order to ensure that all changes will be done correctly, i.e., according to the original intentions of the design patterns and framework used, it will be necessary to make up a rulebook as part of the project. This rulebook should indicate step by step procedures on how to add/delete new functions. If encapsulation was used, note the specific interface to the functions, how the object is created (such as Abstract Factory (87) in Gamma (1995)), and what classes the new function should inherit from and how to use the inheritance. Make the rulebook available to the group doing the implementation, and periodically check on them to make absolutely sure that they are following it, even if you get transferred to another project. Why? Because they will talk bad about you behind your back if their changes cause your design to not work properly. Be known as the "Giant Killer", the person who can be counted on to make the software work even when requirements change causes all others to panic - your management might just end up impressed.

VII. THE POTSHOTS (WHAT COULD GO WRONG)

Software developers are notorious for not reading directions and not writing things down. In any Object-Oriented Design or Structured Systems Design, there needs to be an evangelist, preferably the Software Architect who crafted the design in the first place. "The project architect is the visionary, and is responsible for

© 1996 Society for Design and Process Science

evolving and maintaining the system's architecture. (Booch 1994, pg 272)" One of the Software Architects functions is to make sure that the other software developers on the project read the directions in the rulebook, that they continue to use the tools, and that they continually write down points about the system, what decisions were made and why. Without this attention to how things fit together the project may be doomed to failure from the start, even before changes start making life interesting.

VIII. THE POSTULATE (A CONCLUSION OF SORTS)

We have explored some crazy notions that you can indeed get a handle around the problem of requirements change. But do not kid yourself, it will require dilligence on your part. We have explored the notion that if we concentrate on subsets of requirements initially, then form a framework definition from certain of the subsets, that we will have a better handle on change evolution. We noted that after defining the subsets we were in a better position to add design patterns, those written problem resolutions, to accomplish a means of control over change. The rest of the process is function encapsulation.

REFERENCES

Barnes, Eleanor J., Personal Communication, 6/4/96

Booch, Grady,. Object-Oriented Analysis and Design with Applications. The Benjamin/Cummings Publishing Company, 1994.

Buschmann, Frank, et al. Pattern-Oriented Software Architecture. John Wiley and Sons, 1996.

Coplien, James O. and Schmidt, Douglas C., ed. Pattern Languages of Program Design, Addison-Wesley Publishing Co., 1995.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

Martin, Robert C., Personal Communication, 6/27/96

McGregor, John D., Doble, J., and Keddy, A. A Pattern for Reuse. Object Magazine, 1996.

Ohlsson, Lennart, Object Oriented Frameworks, from Frameworks Home Page at the University of Karlskrona/Ronneby, Sweden

Palepu, Ravi, Modeling the Real World: Application of Patterns to Reduce Complexity in the Software Development Process, School of Computer Science Carleton University, April 1995.

Parnas, David L., Designing Software for Ease of Extension and Contraction, IEEE Transactions on Software Engineering, Vol SE-5, No. 2, March 1979

Vlissides, John M., Coplien, James O., Kerth, Norman L., ed. Pattern Languages of Program Design 2, Addison-Wesley Publishing Co., 1996.

© 1996 Society for Design and Process Science