Architecture's Significance

Introduction

For so long now, industry has been preaching the virtues of the object oriented paradigm. As we've been told for quite some time, reuse is the Holy Grail of this widely accepted, and universally promoted, paradigm. While not without it's detractors, object orientation is here to stay. Most of the popular languages today have at least some characteristics that make it "object oriented". In fact, I wonder if a language could survive today if it weren't dubbed "object oriented". It’s quite common to hear co-workers and peers make statements such as, "We should do it this way. It's the new trend." The "way" and the "trend" being object oriented. Please, don't take this wrong. I am definitely an advocate of this powerful technology. But I really think it's time we seriously ask ourselves a question. Why hasn't the object oriented paradigm allowed us to achieve the high degree of reuse it once promised? Is it because we are doing something wrong? Or quite possibly, something else keeps getting in the way.

The Paradox

In, *No Silver Bullet*, by Frederick Brooks, Mr. Brooks states that:

The hardest single part of building a software system is deciding precisely what to build.

In the premier edition of *The Rational Edge*, Walker Royce cites the following:

For every $1 you spend on development, you will spend $2 on maintenance.

While probably not astounding, these statements should be alarming. Two of the most significant activities performed when developing software are also the most inefficient and error prone. Simply stated, the establishment of requirements, and the way in which we deal with changing requirements in relation to system maintenance, is the number one challenge presented to any software development team. Therefore, before we are concerned with creating reusable code, we need to first be concerned with creating a system that matches the user's requirements, and is resilient to changing requirements. For our system will truly be judged as a success or failure, not by how much code we reuse, but by how well it satisfies user's needs. Herein, however, lies the problem.

As stated above, a system's requirements are not static. They are in constant evolution, and are dictated by the demands of the business and the needs of the user. Therefore, our system is judged not only by how well it adheres to the requirements of today, but also by how well it can adapt to the requirements of tomorrow. In other words, our system survives only as long as it can evolve. However, the survivability and evolvability of a system are competing influences. Subsequently, any emphasis we place on writing reusable code today will only be negated by tomorrow's demand to evolve.

Let's explore this more deeply by examining the life of a typical system. Consider a system that has just been deployed, and meets the requirements of today. As these requirements change, our system undergoes a maintenance effort to support these new requirements. The ease with which we are able to accommodate these changes is a direct result of how accommodating to change our system is. A rigid and inflexible system cannot undergo changes in the same graceful fashion that a malleable and dynamic system can. Eventually, as we perform more and more maintenance cycles, the instability of our system may reach a point where the cost to continue evolving the system can no longer be justified. At this point, our system dies. This is illustrated in Figure 1, where we see that the decision to implement a requirements change results in either the maintenance of that system to fulfill these new requirements, producing once again a reliable and stable system, or results in the death of that system. This decision is typically directly related to the cost of the maintenance effort.

Subsequently, it becomes much clearer that our system's internal structure directly impacts the ability of that system to evolve gracefully. The internal structural representation of a system is commonly referred to as a system's architecture, and the situation above describing the two competing influences is the Architecture Paradox. Because of this, we can also now state the following:

The cost to maintain a system is directly related to the resiliency of the system's architecture.

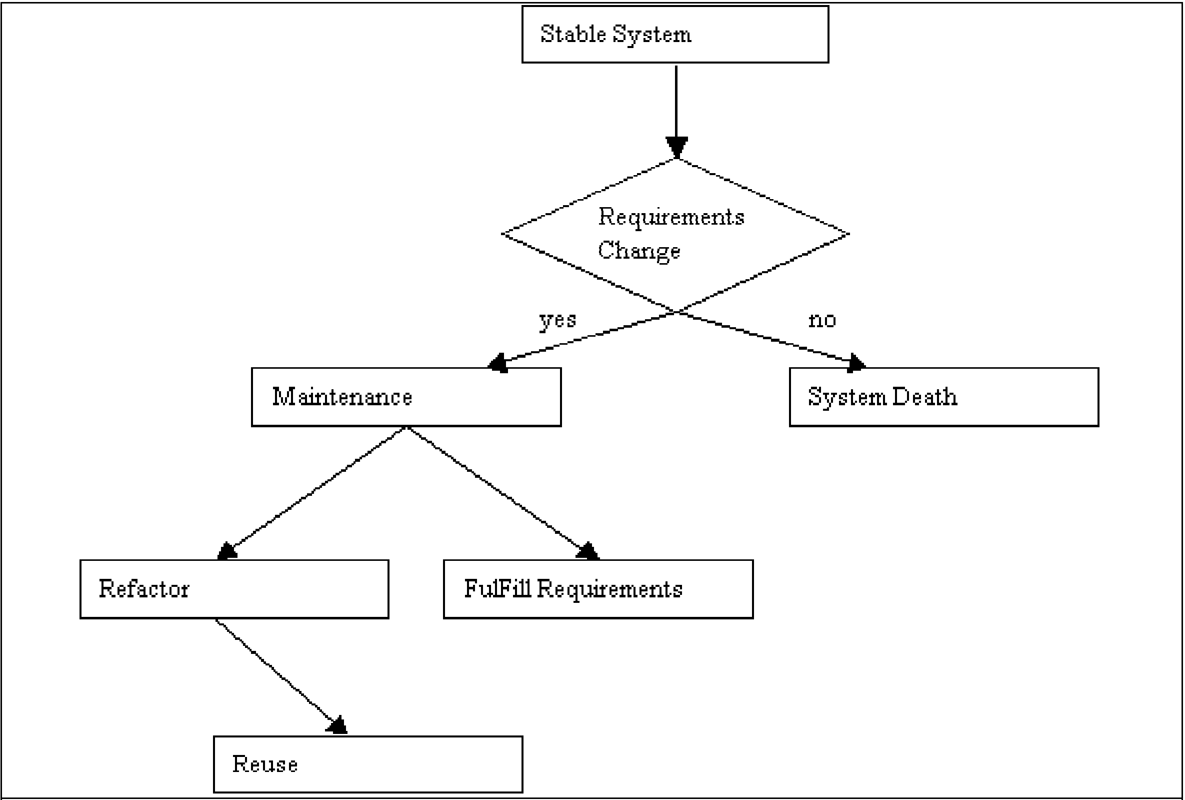

Figure 1

Defining Architecture

Detailing the steps necessary to define a resilient architecture could easily fill a large book, if not a complete volume. Therefore, our intent is not to discuss how to define an entire system's architecture, but to explain what's available to help us do a better job. In fact, the fashioning of the complex structures necessary to represent a software system is a very difficult task. Couple this with the need to design for changes we are unsure will ever occur, and the challenge is only compounded. So, how can we define these resilient architectures?

First, we must take advantage of the flexibility of object orientation. The object-oriented paradigm lends itself quite nicely to creating structures that are extremely flexible. In fact, taking advantage of common principles and patterns can lead us in this direction. For example, combining two simple patterns such as a Strategy and a Factory allows us to create a system that is extremely extensible, yet won't require a lot of maintenance effort. This is an extremely valuable trait of object-orientation, and we should carefully consider using it to design flexible systems that can accommodate change, before necessarily attempting to design for reuse.

Object Orientation and the power behind it is a great tool to help us design for change. However, it's virtually impossible to simply sit down and create code that is flexible enough to bend and flex in the manner we'll need it to. Visual Modeling can aid greatly in providing a simplified representation of our system that can help us see and understand how our system is architected. It's obvious that if we understand something more fully, we will be more effective in maintaining it. Creating these models help to validate that our system's architecture is more resilient by helping us to understand that architecture more fully. This allows us to communicate the system to others, who can then offer valuable insight based on their experiences.

Another tool to help us design for change is refactoring. Refactoring can be thought of as improving the design of existing code. Typically, when refactoring, we do not make any functional changes, but focus strictly on improving a system's internal structure. Refactoring should be done in a very disciplined fashion, not ad hoc, which would be little different from hacking. Taking a disciplined approach to refactoring, diligently and truthfully applying the rules when and where needed, will almost always result in a more architecturally sound system.

In reality, a pragmatic combination of each of the above approaches, in conjunction with other best practices, will work best. Modeling visually to help us resolve any undecided issues works great initially, and throughout development as new challenges are encountered. Inevitably, as the system grows, it will require some refactorings to ensure it maintains the survivability and evolvability necessary to live and expand as the requirements and business demand. Object orientation, with it's flexible principles and patterns, is the paradigm that lends us the flexibility necessary.

In fact, many of us are likely applying each of these techniques already. However, the key is to take the thought and time to apply these techniques to the most critical pieces of the application. It's important that the areas within the application that are most likely to undergo changes in requirements are the areas into which we've built the most flexibility. We really don't need, nor do we want, this flexibility everywhere, as there is ultimately an accompanying degree of complexity associated with it. Therefore, the importance of applying these concepts pragmatically cannot be undermined.

Maintenance and Reuse

By now, we must be wondering where reuse fits into the picture? Have we abandoned all hopes to reuse components within our system? The answer to this is an emphatic "no." However, our approach to reuse may be a bit different than what it was previously. In fact, we cannot simply sit down and write a reusable class library or framework. It's simply not possible. This is primarily because these reusable components must exhibit the most flexibility and resiliency to change. This is extremely difficult, and therefore, these reusable components need to evolve from an existing code base. Consider a statement made in the paper "Patterns for Evolving Frameworks" by Don Roberts and Ralph Johnson:

> *People develop abstractions by generalizing from concrete examples. Every attempt to determine the correct abstractions on paper without actually developing a running system is doomed to failure. No one is that smart.*

Therefore, we must first design a system that functions correctly. Then, the identification of the reuse candidates within that system can take place. Upon identifying the appropriate candidates, we must find the abstractions that are generally applicable, and cull these from the existing design. This too is part of maintenance, however, with a somewhat different focus. Here, we are refactoring to improve our code structure, ultimately achieving reuse. The success with which we will be able to abstract these most useful modules will be directly related to the structural stability of the application. The more rigid the application, the more difficult it will be to extract and create more reusable modules. The more flexible our architecture, the higher the likelihood that the system can be bent, twisted, and restructured to produce the desired result. As we perform this maintenance effort, the UML can play an extremely valuable role in helping to not only understand the system's existing structure, and realize the impact of

Conceptual Difference

Reuse can exist at an entirely different level than does maintenance. Changing requirements typically results in a change to source code, implying the recompilation of a set of classes, after modifying one or more of those classes. Reuse, however, is obtained at the unit of deployment. This unit varies across languages. In Java, this unit is the package. Whether we reuse a JavaBean, or a portion of an API, we need reference to a package. When we reference that package, we are dependent on that package, and it's contents, in its entirety.
Keep in mind that reuse is not a benefit of object orientation, but instead a goal. Reuse can occur within any language, object-oriented or not. The benefit of object-orientation is the flexible architectures we can create with it. From these flexible architectures, reuse is bred.

our efforts, but also to help us work through complex challenges. Now, the true importance of architecture is fully realized, and it becomes more apparent that architectural resiliency coupled with disciplined design is the most effective way to create reusable modules. Therefore, we can now state the following:

Resilient architectures breed reuse.

Now, if we firmly believe that reuse is the Holy Grail of object-orientation because reusing code will allow us to develop applications more quickly, resulting in lower cost, and reduced maintenance. Then, because cost and maintenance are directly related to architectural resiliency, and architecture breeds reuse, the following also holds true:

Establishing our software system's architecture is the single most important technical decision we face when developing software.

Conclusion

We should all agree upon the significant role that our software's architecture plays in the success of our software systems. The architecture contributes not only to the immediate success of a system, but also to its survival, and growth in the months and years to come. Therefore, careful thought and deliberation should be given when establishing the structure of a system. It has the ability to profoundly impact the success of our software development efforts.

References

[SUB99] *Architecture Paradox*. Dr. Subrahmanyam A. V. B., 1999.

[PLOP98] *Patterns Languages of Program Design, Volume 3.* Martin, et. al. Addison-Wesley, 1998.

[ROYCE2000] *The Rational Edge*. "Next-Generation Software Economics", Walker Royce. December, 2000.

[BROOKS] *Computer Magazine*. "No Silver Bullet: Essence and Accidents of Software Engineering", Frederick P. Brooks, 1987.

[FOWLER99] *Refactoring: Improving the Design of Existing Code*. Fowler, Martin. Addison-Wesley, 1999.

[GOF96] *Design Patterns: Elements of Reusable Object-Oriented Software.* Gamma, et. al. Addison-Wesley, 1996.