JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 6, July - August 2006

Patterns of Anti-Patterns?

Mahesh H. Dodani, IBM Software, U.S.A.

1 LOOKING FOR COMMON WORST PRACTICES

"To err is human. We all make mistakes but the most successful companies learn from them. This 'worst practice' guide is intended to help you learn from the mistakes that others have made, so that you can identify and avoid them. Alternatively, you could follow all the guidance and really mess up!" – http://www.oursouthwest.com/SusBus/worstpg.html

Over the last decade we have witnessed the maturity of both best practices (as manifested in the patterns movement http://en.wikipedia.org/wiki/Design_pattern_(computer_science)) and worst practices (as manifested in the anti-patterns movement http://en.wikipedia.org/wiki/Anti-pattern.) As we evolve through software engineering (moving from functional to objects to components to services) practices, we can see trends emerging in both the patterns that can lead to successful implementations as well as the anti-patterns that show the common mistakes that we make.

Patterns address the need for solutions to common problems in a well-defined context. Ideally, the solution is easy to use and adapt to the particular context. Therefore, developing a pattern can be seen as a bottom-up process, where a recurring solution is used to address (at least three) common problems, and the solution is then abstracted into a pattern.

Antipatterns, on the other hand, identify common problems with solutions, and then show how to refactor the solution to get rid of the problem. Therefore, developing a antipattern is a top down process, where the (at least three) problems with a recurring solution are identified, and then a best practice refactoring of the solution is developed to address the problems. The problem, solution, and refactored solution then get put together into an antipattern.

Figure 1 diffrentiates between patterns and antipatterns as well as shows their relationship. A pattern starts with a problem and documents a repeatable successful solution to it. The solution generates some benefits, consequences and possible problems. An antipattern demonstrates a frequently used solution to a problem that has a negative effect. The antipattern describes the causes that led to the worst practice and also shows how to prevent or correct the solution. Note that over time and when applied within a new context a pattern can evolve into an antipattern.

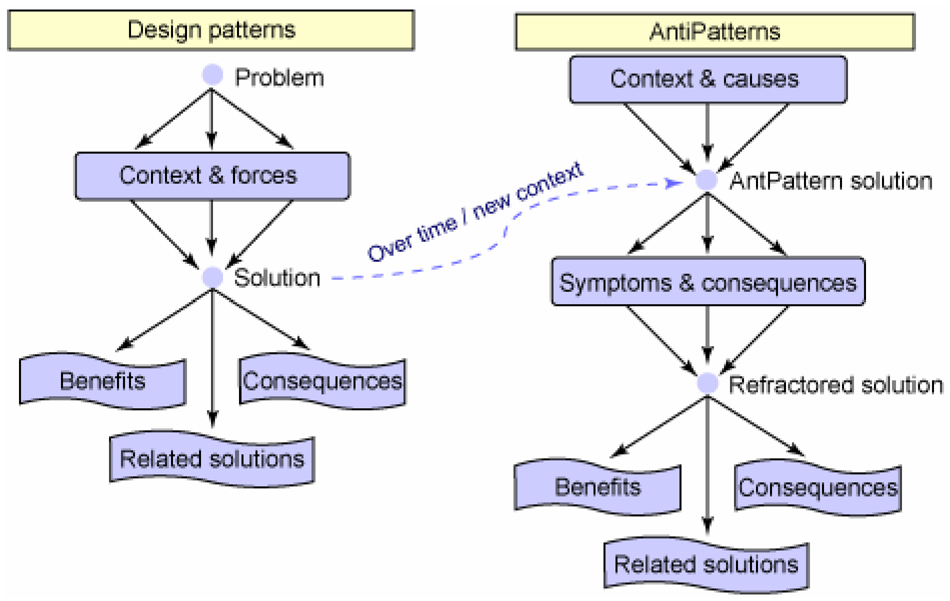

Figure 1: Patterns and Anti-Patterns (sourced from http://www.antipatterns.com)

To ensure that we learn from our past mistakes and not repeat them as we evolve our software engineering practices, we should look at "patterns" of anti-patterns.

2 PATTERNS OF ANTI-PATTERNS

As we have evolved through software engineering approaches, we continue to find best practices that facilitate flexibility, ease of change and better alignment of IT with business needs. The following have emerged over the years as driving principles of good software engineering practice:

- Program to an interface and not to an implementation. This principle puts the onus of designing reusable components to the art of designing good interfaces. Our software engineering journey has provided us support from patterns, frameworks, architectures, and tools to help ensure that good interfaces can be designed and maintained.
- Use composition to extend behavior. This principle decouples extended behavior from the original behavior thereby allowing the extended behavior to change without impacting the original.
- Minimize coupling between atomic components to ensure that changes are localized and do not propagate. This separation of concerns allows each component to focus on specific capabilities and facilitates the management of

these components without any dependencies to other components. Related to this principle is the ability to separate collaboration/control behavior and logic from the underlying business behavior and logic. This approach (especially when defined “declaratively”) supports changing of collaborative behavior without the need to re-code the underlying logic.

- Iterative and incremental approaches applied to every aspect of software engineering is the key to success. Built into these iterative and incremental approaches are the metrics to measure conformance to principles, use of best practices, and the effectiveness of the process. These metrics facilitate continuous improvement of the approaches through each increment and iteration.

It is interesting that many of the “patterns” of antipatterns are based on a violation of the above principles and best practices – basically, as we evolve our software engineering approaches, we tend to repeat the same mistakes. For this article, we will focus on two antipatterns and use the mini antipattern template as described in http://www.antipatterns.com:

- AntiPattern Name: What shall this AntiPattern be called by practitioners?
- AntiPattern Problem: What is the recurrent solution that causes negative consequences?
- Refactored Solution: How do we avoid, minimize, or refactor the AntiPattern problem?

AntiPattern Name: Iterative/Incremental Development, Waterfall Style

AntiPattern Problem: As we discussed above, one of the best practices that has emerged over the years is the use of iterative and incremental methods for software engineering. These methods have evolved over the years, culminating in the current agile methods (http://www.agilealliance.org/home), which facilitate individuals interacting with each other and in close collaboration with their customer to build incremental versions of the system and quickly respond to needed changes. Unfortunately, in practice it is difficult to adopt and follow iterative and incremental approaches. This problem is exacerbated by the newer approaches (e.g. SOA) where the domain for applying these methods has grown to include all parts of the enterprise. As an example, the incremental and iterative SOA lifecycle (see http://www-128.ibm.com/developerworks/webservices/newto/) includes modeling the business itself, modeling the business services and processes that are exposed, assembling the components needed to realize the services and processes, deploying the services and processes to the operating infrastructure, managing the deployed environment, and monitoring the environment to collect metrics that can be used for optimizing the business processes as well as the IT environment. The methods themselves typically focus on a particular aspect of the entire domain -- so, we have methods for business modeling, business process and service modeling, assembling and testing of the process and services, and IT service management. These methods are further augmented by governance and management methods. Even though each method itself may be incremental and iterative, the combined approach typically is applied in a waterfall fashion. This leads to the same problems that we faced in purely waterfall methods -- analysis paralysis, slow response to changes, etc.

Refactored Solution: Build a roadmap that provides the overall blueprint for how an organization will move towards SOA. Building such a roadmap requires a well defined model that describes capabilities against maturity levels (for example, see the Service Integration Maturity Model described in http://www-128.ibm.com/developerworks/webservices/library/ws-soa-simm/.) The roadmap uses the maturity model to plot a path that leads an enterprise from its current state to their desired state. The enterprises' state is defined in terms of characteristics and capabilities established in the maturity model, and the path is defined in terms of a prioritized set of projects and an incremental/iterative process (along with a timeline.) Such a roadmap then provides the "glue" to ensure that the entire journey is incremental and iterative, as well as allowing for the individual parts of the approach to be accomplished using iterative and incremental methods.

AntiPattern Name: RAD-DAR Love (Also known as Not Invented Here)

AntiPattern Problem: Software engineering requires a good understanding of Reusing Abstractions for Development as well as Developing Abstractions for Reuse (RAD-DAR.) Reuse is an integral part of any software engineering practice, and systematic reuse approaches have been attempted, with mixed results, throughout software engineering history. The main reasons why reuse has failed include:

- A poor understanding of reuse, both from the perspective of creating reusable components as well as using reusable components to build other components. This problem typically manifests itself in reuse repositories that contain components that have never been reused (outside of the group that created them.)
- Lack of integration of reuse within the software engineering practices and methods. Reuse is often seen as a separate consideration (or worse an optional step) within the software engineering method. This also applies to the harvesting and updating of assets from usage experience in the field. This problem typically manifests itself in reuse being practiced only by specialized teams.
- Insufficient support for reuse within the organization. This lack of support can manifest itself from an organizational perspective where issues of funding, sponsorship, and governance are addressed; as well as from a tool perspective where issues of storing, searching, using, publishing and measuring reuse are addressed. This problem typically manifests itself in a perpetual cycle of seeding reuse programs that shows promise when applied to a small program, but can never be fully implemented across the entire enterprise.

Refactored Solution: Reuse must be systematically integrated into the fabric of the enterprise. Governance must ensure that the appropriate decisions are made around reusing components as well as building reusable components; along with important considerations for funding, ownership, and incentives. Part of this governance is the establishment of a board that defines common standards for a reusable component, best practices for reuse, and the measurements that can be used to determine the effectiveness of reuse and to continously improve the practices. Throughout the software engineering lifecycle, ensure that reuse is a major consideration. Furthermore, practitioner guides are needed to articulate not only how to reuse, but also the difficulties in reusing components and the characteristics of reusable components. After the development is complete, ensure that newly developed components are made available to others. This allows teams reusing the components to evaluate and provide feedback on the reusability of the components. Tools are needed to support all aspects of reuse within the enterprise. Repositories are needed to publish, search, and understand the characteristics of the reusable component. Reusable components and cookbooks on how to use them should be integrated into the modeling, design, and development tools. Established reuse metrics should be automatically collected, and displayed through dashboards appropriate for the particular role of the decision maker and to improve the reuse processes and practices.

3 PRACTICE MAKES PERFECT

In summary, it is important for us to learn from our mistakes through each evolution of software engineering approaches to ensure that we are not repeating them. To understand these we need to look at both the best (or emerging) practices as well as the common mistakes (or worst practices.)

Understanding "patterns" of common mistakes (antipatterns) and ensuring that we have well defined refactored solutions to address these antipatterns will help us successfully implement each increment of evolving our software engineering practices.

About the author

Mahesh Dodani is a software architect at IBM. His primary interests are in enabling communities of practitioners to design and build complex on demand business solutions. He can be reached at dodani@us.ibm.com.