**Reuse Patterns and Antipatterns**

By [**Scott Ambler**](http://www.drdobbs.com/sdonline/authors.html#sambler), February 01, 2000

[**Post a Comment**](http://www.drdobbs.com/reuse-patterns-and-antipatterns/184414576#disqus_thread)

You can achieve high levels of reuse within your organization, but only if you choose to succeed.

Most software development projects exceed their schedules or budgets. Depending on project size, between 25 percent and 50 percent of projects fail, where "failure" means that the project is canceled or exceeds its schedule estimates.

Software reuse–specifically, reuse of proven solutions–is arguably the only way to get to the root of the problem of software project failure. This is where patterns and antipatterns have roles to play. A pattern is a common approach to solving a problem that is proven to work in practice. Conversely, an antipattern is a common approach to solving a problem that leaves us worse off than when we started.

Consistent reuse requires a change in mindset. Developers must be willing to work together, to reuse each other’s work, to help the reuse efforts of their organizations and to plan to reuse items wherever possible. When you begin a project, you should first determine what portions of your application can be reused from elsewhere. Perhaps someone else has built what you need, or perhaps you can purchase it. The flip side of the coin is that you must be willing to share your work with other people so that they can reuse it. A good team leader will constantly be on the lookout for opportunities for reuse and will promote and reward reuse within his team. An excellent approach is to look for reuse during inspections: During a model walkthrough, look for opportunities for inheritance and pattern reuse, and during code walkthroughs look for component and code reuse. Reuse is an attitude, not a technology.

You can achieve high levels of reuse within your organization, but only if you choose to succeed. It’s very easy to choose to fail–following any one of the reuse antipatterns listed below is a decision to fail–but very difficult to choose to succeed. You need to manage your reuse efforts, put processes in place that support reuse, mature your culture so that reuse is a desired practice, and invest in the purchase and/or development of a wide range of reusable items.

Note: Although patterns and antipatterns are typically presented in templated format–with sections for the name, description, initial context, solution and resulting context–for the sake of brevity, I present them here to you in a less detailed, "degenerate" format.

|  |  |
| --- | --- |
| **Pattern** | **Description** |
| *Reuse-Friendly Culture* | An organization that recognizes that it is honorable to reuse the work of others when appropriate, disgraceful to develop something that could have been found elsewhere, and virtuous to consider the needs of others by generalizing appropriately. Organizations with a *Reuse-Friendly Culture* recognize that reuse is a cross-project, infrastructural effort based on cooperation and teamwork. |
| *Robust Artifact* | An item that is well-documented, built to meet general needs instead of project-specific needs, throroughly tested, and has several examples to show how to work with it. Items with these qualities are much more likely to be reused than items without them. A *Robust Artifact* is an item that is easy to understand and work with. |
| *Self-Motivated Generalization* | Developers often generalize an item (that is, make it potentially reusable by others) out of pride in their work. This is very common within the open source community, in which developers create source code that they share universally. Peer recognition for quality of work is often more important to a developer than a monetary reward. |
| *Senior Reuse Engineer* | The developer of a reusable artifact must have the skills necessary to develop a *Robust Artifact* and be able to support its use by other developers. This skillset is typically found in senior developers who have a wide range of development and maintenance experience, who have a software engineering background, and who have successfully reused the work of others on actual projects. |

|  |  |  |
| --- | --- | --- |
| **Antipattern Name** | **Description** | **Refactored Solution** |
| *Reuseless Artifact* | An artifact believed to be reusable, often because it is *Declared Reusable*, which is not reused by anyone. | Someone other than the original developer must review a *Reuseless Artifact* to determine whether or not anyone might be interested in it. If so, the artifact must be reworked to become a *Robust Artifact*. |
| *Repository-Driven Reuse* | The belief that creating a reuse repository, a mechanism that stores and automates management of potentially reusable items, will drive reuse within your organization. Often a result of *Reuse Comes Free*. | Many organizations achieve significant reuse simply by maintaining a directory of reusable artifacts, and people often find these artifacts through word of mouth and not a fancy search mechanism. This makes it doubtful that a reuse repository is any sort of prerequisite for success. You need a *Reuse-Friendly Culture* to achieve high-levels of reuse, not a nifty new tool. Yes, a repository does provide a search facility and configuration management, but these features only support reuse, they don't guarantee it. |
| *NIH Syndrome Excuse* | Developers of a *Reuseless Artifact*claim that others don't reuse it because those developers didn't create it themselves—the "not invented here" (NIH) syndrome. | Professional developers constantly seek to reuse the work of others because it frees them up to work on the domain-specific portions of their own applications. People will readily reuse *Robust Artifacts*, not items that are only *Declared Reusable*. |
| *Declared Reusable (also known as "If You Build It, They Will Come")* | The belief that something is reusable simply because you state that it is so. Often a result of *Reuse Comes Free*. | Although this approach does engender some reuse, the typical result is a collection of *Reuseless Artifacts*. Reuse is about quality, not quantity. You know that something is reusable only after it has been reused; reusability is in the eye of the beholder, not the eye of the creator. |
| *Reward-Driven Reuse* | The belief that all your organization needs is a reward program to achieve high levels of reuse. | Most bonuses work out to less than minimum wage, when calculated on an hourly basis, so it's doubtful that people do it for the money. *Self-Motivated Generalization* and reuse of *Robust Artifacts* are far more common in practice. Trust your coworkers. They'll do the right thing when given the opportunity. |
| *Production Before Consumption* | The belief that you can start by building reusable artifacts. | The reality is that you need to invest heavily to make reuse a reality. Dedicate resources to develop *Robust Artifacts*. Grow and support a *Reuse-Friendly Culture*. Put configuration management and change control processes in place. Reuse driven from the top down requires infrastructure management processes such as organization-level architectural modeling. |
| *Code Reuse Only* | The belief that you can only reuse code. | Of the many items that you can reuse, such as components and documentation templates, code reuse is typically the least productive. You can reuse a wide variety of artifacts throughout the entire software life cycle. |
| *Project-Driven Reuse* | The limiting of generalization decisions to the scope of a single project. | Yes, a single project may be able to obtain some level of reuse on its own, but reuse is a multi-project effort. Generalizing a date routine, or a use-case template, or a user-interface design standards document offers little value to the project. The benefits of generalization efforts are often realized by the projects that come later. You need the infrastructure-oriented viewpoint of a *Reuse-Friendly Culture*. |