Protecting an Endangered Species: Design Patterns

Do you know what happens to your cherished design patterns once your application is delivered and enters the hard, wildlife of exploitation, software evolution, and maintenance? Life is a jungle for the application code and health in the ecosystem of permanent software evolution, rapid maintenance, and changing software maintenance staff. It is likely that their life expectancy turns shorter than ever as the application evolution changes hands.

When you carefully crafted your design patterns, your intent was to exploit the experience of the "Gang of Four" masters and others by using proven, rock-solid arrangements of objects and their documented tradeoffs. Your goal was probably to help promote easier program changes and object reusability through these shared solutions.

However, findings in the field show that your interest and knowledge of design patterns — their purpose, their use, and their benefits — is not always shared as it could be. Lack of knowledge and misunderstanding of major design patterns is a plague. And the consequence is that many times once the team that originally developed the application hands it over to a new staff the life expectancy of the design patterns falls rapidly.

Several researchers have worked on this issue, mainly by proposing automatic detection and documentation of design patterns. Many thesis and research projects that tried (and still try) to automatically detect design patterns in existing code in order to document them. As an example, you can look at this 2007 thesis by Marcel Birkner titled "Objected-Oriented Design Pattern Detection Using Static And Dynamic Analysis In Java Software."

More recently, and using advanced learning techniques, a team from the School of Computing of DePaul University, Chicago, Ill. produced a "Tactic-Centric Approach for Automating Traceability of Quality Concerns." This system automates the documentation of "tactics," a higher level of design patterns. The Chicago team explains in its research paper that:

"Unfortunately, software architectures tend to degrade over time as maintainers modify the system without understanding the underlying architectural decisions. Although this problem can be mitigated by manually tracing architectural decisions into the code, the cost and effort required to do this can be prohibitively expensive."

At CAST, we regularly analyze recent object-oriented IT applications as well as aging ones, and we have seen the effect of both lack of understanding of design patterns and missing design-to-code traceability.

This is why the protection of design patterns out in the wild is one of our concerns at CAST. And we decided to provide such a protection for both our customers and our own software
platform. However, we think that instead of writing or generating paper documentation that will hardly be read, it is necessary to automatically alert development teams and project managers each time that a design pattern is in danger. The idea came while working with customers on how to improve the evolutive maintenance of a mission critical application. To protect their design patterns, they had the idea to use CAST Architecture Checker to help monitor these precious pieces of code in their applications.

The very first pattern they checked was their custom implementation of the classic Model-View-Controler (MVC) pattern. Here is how it works. CAST Architecture Checker is used to define layers and authorized dependencies (or forbidden dependencies) gathered in an architecture model. This model is then checked either within CAST Architecture Checker or within CAST Application Intelligence Platform after static code analysis of all the programming languages in the application code. The violation dependencies are then displayed for remediation.

The idea was to define layers containing each class of the pattern, define the authorized dependencies, and activate the check to detect a violation of the pattern. In the case of the MVC pattern, we've defined three layers: one for model classes, another for view classes, and one for controller classes. When defining layers, we tell CAST Architecture Checker how to find M classes, V classes, and C classes using inheritance and naming convention or other conventions. Based on these definitions, the architects can check interactively for unwanted dependencies or integrate the checks into the application health check performed automatically (for example, each night or before new deliveries).

Architecture Checker became a pattern checker! So we started using it that way at CAST too. The idea is to check our own patterns to avoid any errors and degradation of the software platform. One of the patterns checked using CAST Architecture Checker on our own application source code is the interpreter pattern. The interpreter pattern, originally documented by the “Gang of Four,” is heavily used in the CAST family of 16 source code analyzers that cover the application analysis from end to end.

Using CAST quality platform on our own application source code is an operation we call “CAST on CAST.” This design pattern protection schema made part of CAST on CAST checks enables the entire development team to implement changes, continually evolving these analyzers, while being sure that precious patterns are not in danger.

As CAST Architecture Checker enables cross-languages checks, we are also looking at the protection of framework design patterns which involves programming languages, APIs, and XML configuration files.

How about you? Have you experienced any endangered design patterns? Which ones would you like to see protected in your application? Tell us in a comment.

Tags: Architecture Checker, CAST Application Intelligence Platform, Model-View-Controller pattern, object oriented, object re-usability, quality concerns, Software development design patterns, software...