Reinvent The Wheel

- **AntiPattern Name:** Reinvent the Wheel
- **Also Known As:** Design in a Vacuum, Greenfield System
- **Most Frequent Scale:** System
- **Refactored Solution Name:** Architecture Mining
- **Refactored Solution Type:** Process
- **Root Causes:** Pride, Ignorance
- **Unbalanced Forces:** Management of Change, Technology Transfer
- **Anecdotal Evidence:** "Our problem is unique." Software developers generally have minimal knowledge of each other's code. Even widely used software packages available in source code rarely have more than one experienced developer for each program.Virtually all systems development is done in isolation of projects and systems with overlapping functionality. Reuse is rare in most software organizations. In a recent study of more than 32 object-oriented software projects, the researchers found virtually no evidence of successful reuse

Background

Software and design reuse are significantly different paradigms. Software reuse involves the creation of a library of reusable components, the retrieval of those components, and the integration of the components with a software system. The typical result is a modest amount of reuse around the periphery of the system and additional software development to integrate the components.

Design reuse involves the reuse of architecture and software interfaces in multiple application systems. It requires the identification of horizontal components with uses across multiple application systems.

Design reuse also supports software reuse of the horizontal components without additional development for integration, thus it is a much more effective approach, in that a larger portion of the software system can be leveraged from reusable components.

The term *greenfield* (in Greenfield System, an alias of Reinvent the Wheel) originates from the construction industry. It refers to a new construction site where there are no legacy buildings to introduce constraints on the new building's architecture.

General Form

Custom software systems are built from the ground up, even though several systems with overlapping functionality exist. The software process assumes "greenfield" (build from scratch) development of a single system. Because top-down analysis and design lead to new architectures and custom software, software reuse is limited and interoperability is accommodated after the fact.

Most current software methods assume that developers are building custom software from scratch, and that they are building a single system in isolation. These are called *greenfield system assumptions*.

Greenfield systems inevitably become stovepipes that lack potential for interoperability, extension, and reuse. Greenfield assumptions are mismatched to most real-world software development problems, where legacy systems exist, and interoperation with them is an important requirement for many new systems. Greenfield assumptions also ignore significant reusable software assets in the form of Internet freeware and commercially available software.

Symptoms And Consequences

- Closed system architectures—architectures and software—that are designed for one system at a time without provision for reuse and interoperability.
- Replication of commercial software functions.
- Immature and unstable architectures and requirements.
- Inadequate support for change management and interoperability.

- Extended development cycles involving failed and dead-end prototypes before the architecture is mature enough to support long-term system development and maintenance.
- Poor management of risks and costs, leading to schedule and budget overruns.
- Inability to deliver the desired features to the end user; extensive effort to replicate the functionality already operational in existing systems.

Typical Causes

- No communication and technology transfer between software development projects.
- Absence of an explicit architecture process that includes architecture mining and domain engineering.
- Assumption of greenfield development; in other words, the process assumes that the system will be built from scratch.
- Lack of enterprise management of the computational viewpoint, leading to unique software interfaces in each system.

Known Exceptions

The Reinvent the Wheel AntiPattern is suitable for a research environment and in general software development to minimize coordination costs where developers with different skills work at logistically remote sites.

Refactored Solution

Architecture mining is a way to quickly create successful object-oriented architectures that are robust, product-independent, reusable, and extensible. Most object-oriented design approaches assume that design information is invented as the process proceeds.

In a top-down process, design information is generated from requirements, which may be represented as use cases and object-oriented analysis models. Requirements-driven architecture design is called *architecture farming*. In a spiral process, design information is invented during each iteration. As the spiral process proceeds, architects derive new design information as they learn more about the application problem.

It's fair to say that these approaches *reinvent* much of their design information.

Precursor designs exist for most information systems applications and problems. These designs are in the form of legacy systems, commercial products, standards, prototypes, and design patterns.

Experience proves it is not difficult to identify a half-dozen or more precursor designs for any given application problem. Valuable information is buried in preexisting designs, information that enabled earlier architects to build useful systems. Extracting this information for use in object-oriented architectures is called *architecture mining*.

Mining may be applicable at the application level for certain complex design problems. In some cases, it may be less expensive and risky to exploit existing expertise than to create new code without exposure to preexisting solutions. Mining is applicable at enterprise levels, but less so at global levels, given the reduced access to information resources.

Mining is a bottom-up design approach, incorporating design knowledge from working implementations. Mining can incorporate design input from top-down design processes, too, so that there can be both top-down traceability and bottom-up realism.

Before mining starts, it is necessary to identify a set of representative technologies that are relevant to the design problem. Technology identification can be done by various means, such as searching literature, interviewing experts, attending technical conferences, and surfing the Net. All available resources should be pursued.

The first mining step is to model each representative technology, to produce specifications of relevant software interfaces. We recommend using OMG IDL as the interface notation because it is concise and free from implementation detail.

OMG IDL is also a good design notation for the target architecture because it is language-independent, platform-neutral, and distribution-transparent. Modeling everything in the same notation creates a good basis for design comparison and trade-off.

While modeling, it is important to describe the as-built system, not the intended or desired design. Frequently, relevant design information is not documented as software interfaces. For example, some of the sought-after functionality may be accessible only through the user interface. Other key design lessons may be undocumented, and it is useful to capture this information, too.

In the second mining step, the designs are generalized to create a common interface specification. This step entails more art than science, as the goal is to create an initial "strawman" specification for the target architecture interfaces.

It is usually not sufficient to generate a lowest-common denominator design from the representative technology. The generalized interfaces should resemble a best-of-breed solution that captures the common functionality, as well as some unique aspects inspired by particular systems.

Unique aspects should be included when they create valuable features in the target architecture or represent areas of known system evolution. A robust assortment of representative technologies will contain indicators of likely areas of target system evolution.

At this point, it is appropriate to factor in the top-down design information as one of the inputs. Top-down information is usually at a much higher level of abstraction than bottom-up information. Reconciliation of these differences involves some important architecture trade-offs.

The final step in the mining process is to refine the design. Refinements can be driven by the architect's judgment, informal walkthroughs, review processes, new requirements, or additional mining studies.

Variations

Within an organization, software reuse is difficult to achieve. In a survey of several dozen object-oriented application projects, Goldberg and Rubin found no significant reuse Even if successful, the cost benefits of internal reuse are usually less than 15 percent Industry experience indicates that the primary role of internal reuse is as a investment in software for resale. Where large volumes make the potential savings significant, reuse can shorten time-to-market and support product customization.

On the other hand, we claim that reuse is prevalent, but in different forms: reuse of commercially available software and reuse of freeware. Because of larger user bases, commercial software and freeware often have significantly higher quality than custom-developed software. For infrastructure components upon which much application software depends, this enhanced quality can be critical to project success.

Commercial software and freeware can reduce maintenance costs when the software is used without modification and can be readily replaced with upgraded versions.

Related Solutions

The impact upon management of complexity of architecture mining and the generalization to common interfaces is analyzed in Mowbray 95. Architecture mining is a recurring solution that addresses many of the problems caused by Stovepipe Systems. It is also one of the approaches for defining domain-specific component architectures.

Applicability To Other Viewpoints And Scales

The Reinvent the Wheel AntiPattern puts managers at higher risk, in the form of increased time-to-market and a lower degree of functionality than that expected of the end users. Potential savings from reuse range from 15 to 75 percent of development cost, reduction of 2 to 5 times in time-to-market, and reduction of defects from 5 to 10 times