Vendor Lock-In

- **AntiPattern Name:** Vendor Lock-In
- **Also Known As:** Product-Dependent Architecture, Bondage and Submission, Connector Conspiracy
- **Most Frequent Scale:** System
- **Refactored Solution Name:** Isolation Layer
- **Refactored Solution Type:** Software
- **Root Causes:** Sloth, Apathy, Pride/Ignorance (Gullibility)
- **Unbalanced Forces:** Management of Technology Transfer, Management of Change
- **Anecdotal Evidence:**We have often encountered software projects that claim their architecture is based upon a particular vendor or product line.
  Other anecdotal evidence occurs around the time of product upgrades and new application installations:
  "When I try to read the new data files into the old version of the application, it crashes my system." "Once you read data into the new application, you can never get it out again." "The old software acts like it has a virus, but it's probably just the new application data." "Our architecture is.. What's the name of our database again?"

Background

A worst-case scenario of this AntiPattern would occur if your data and software licenses were completely allocated to online services, and one day, a modal dialog box popped up.

Interactive word processing became more popular than formatting language technologies (like SGML) because it allows the user to see the final format on the computer screen and print an exact copy of the on-screen appearance. This capability is called "What you see is what you get" (WYSIWIG).

A pervasive variation of Vendor Lock-In is the phenomenon called "What you see is sort-of like what you get" (WYSISLWYG, pronounced "weasel wig"). Recently, given the desktop dominance of Microsoft, product misfeatures cause the printed versions of documents to vary significantly

from their on-screen appearance. For example, symbols in drawings can change or disappear, and embedded objects are often printed as command strings (like "{EMBEDDED POWERPOINT FIGURE}").

Documents from different versions of the same Microsoft product can cause support problems on corporate networks and system crashes. Many companies discourage or outlaw co-mingling of product versions. It is difficult to avoid this form of vendor lock-in and its product misfeatures because of organizational dependence on Microsoft's products for document interchange.

General Form

A software project adopts a product technology and becomes completely dependent upon the vendor's implementation. When upgrades are done, software changes and interoperability problems occur, and continuous maintenance is required to keep the system running.

In addition, expected new product features are often delayed, causing schedule slips and an inability to complete desired application software features.

Symptoms And Consequences

- Commercial product upgrades drive the application software maintenance cycle.
- Promised product features are delayed or never delivered, subsequently, causing failure to deliver application updates.
- The product varies significantly from the advertised open systems standard.
- If a product upgrade is missed entirely, a product repurchase and reintegration is often necessary.

Typical Causes

- The product varies from published open system standards because there is no effective conformance process for the standard.
- The product is selected based entirely upon marketing and sales information, and not upon more detailed technical inspection.

- There is no technical approach for isolating application software from direct dependency upon the product.
- Application programming requires in-depth product knowledge.
- The complexity and generality of the product technology greatly exceeds that of the application needs; direct dependence upon the product results in failure to manage the complexity of the application system architecture.

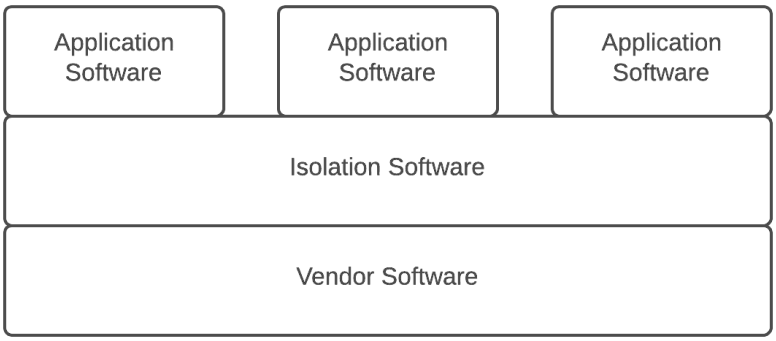
Known Exceptions

The Vendor Lock-In AntiPattern is acceptable when a single vendor's code makes up the majority of code needed in an application.

Refactored Solution

The refactioned solution to the Vendor Lock-In AntiPattern is called *isolation layer*. An isolation layer separates software packages and technology. It can be used to provide software portability from underlying middleware and platform-specific interfaces. This solution is applicable when one or more of the following conditions apply:

- *Isolation of application software from lower-level infrastructure*. This infrastructure may include middleware, operating systems, security mechanisms, or other low-level mechanisms.
- *Changes to the underlying infrastructure are anticipated within the life cycle of the affected software;* for example, new product releases or planned migration to new infrastructure.
- *A more convenient programming interface is useful or necessary*. The level of abstraction provided by the infrastructure is either too primitive or too flexible for the intended applications and systems.
- *There a need for consistent handling of the infrastructure across many systems*. Some heavyweight conventions for default handling of infrastructure interfaces must be instituted.
- Multiple infrastructures must be supported, either during the life cycle or concurrently.



The solution entails creating a layer of software that abstracts the underlying infrastructure or product-dependent software interfaces. This layer provides an application interface that completely isolates the application software from the underlying interfaces.

The application interface should implement a convenient language-specific interface to desired capabilities. The layering software should ensure default handling of some infrastructure calls and parameters, but expose other details when appropriate.

This isolation layer is used across multiple system development projects to assure interoperability, consistency, and isolation. To implement it, migrate the isolation to new infrastructures as necessary; also, update the isolation layer when the infrastructure is updated. In all cases, maintain the same application software interface, regardless of infrastructure changes.

It is also necessary to install gateways between multiple infrastructures that must be supported concurrently, and to install forward and reverse gateways during infrastructure migration

The benefits of this solution are:

- Mitigation of the risks and costs of infrastructure migration.
- Precluding obsolescence caused by to infrastructure changes.
- Reduction of the risk and cost of software upgrades required by infrastructure changes.
- Assurance of a less labor-intensive and more inexpensive programming interface to most application programmers.
- Support for the concurrent use of multiple infrastructures, transparently.
- Enforcement of coordinated default handling of flexible interfaces and parameters.

- Separation of infrastructure knowledge from application knowledge, thereby enabling a small team of infrastructure developers to maintain the isolation layer, while the majority of programmers have a customized interface to the layering software.

Other consequences of this solution are that:

- The isolation layer must be migrated and maintained, potentially on multiple platforms and infrastructures.
- The developers who define the initial isolation layer interfaces must be coordinated.
- Changes made to the application interfaces must be coordinated.

Variations

This solution is often used at the global level in commercial products and technologies. Typically, the isolation layer enables the vendor to provide a convenient language-specific interface to a lower-level technology. Some of this convenience comes in the form of default handling of lower-level interfaces that are more flexible than necessary for most applications.

For example, the HP Object-Oriented Distributed Computing Environment (OO DCE) product comprises an isolation layer, and presents a C ++ interface to application developers. Underlying this interface is an isolation layer of software that is built upon the C language DCE environment.

Calls to the C ++ APIs can invoke several underlying DCE procedure calls; in particular, just two calls are needed to initialize OO DCE security service interfaces. The underlying isolation layer, in turn, makes more than 50 calls to DCE APIs to achieve this initialization with the legacy DCE security service.

The isolation layer solution is most applicable at the enterprise level. However, individual systems have applied this solution to provide middleware isolation; for example, the Paragon Electronic Light Table (ELT) product uses an isolation layer above the Common Desktop Environment (CDE) middleware infrastructure, called ToolTalk. By isolating ToolTalk, Paragon can easily migrate its product to a CORBA infrastructure and support both CORBA and ToolTalk infrastructures.

Example

The following examples are three known uses of the isolation layer solution:

1. The ORBlite framework, based on HP ORB plus, isolates application software from multiple language mappings and network protocols Thus, it was able to support multiple language mappings for C ++ , given the evolution of the OMG mappings during the adoption and revision process
2. Even though OpenDoc is no longer in the product strategy of its creator, some interesting technical approaches were used, including the isolation layer solution. The OpenDoc Parts Framework (OPF) institutes a higher-level C ++ programming interface to the OpenDoc compound document interface, defined in ISO IDL. OPF includes interfaces to operating system functions (including display graphics), as well as OpenDoc functions.In doing so, OPF enables source code portability interfaces for middleware, windowing, and operating systems. Compound document parts written using OPF can be ported via recompilation and linking to OS/2, MacOS, and Windows95. A testing capability called LiveObjects, from the Component Integration Labs, the consortium responsible for OpenDoc, assures component portability and interoperability.

3. EOSDIS (Earth Observing System Data and Information System) is a large-scale information retrieval project funded by NASA. The EOSDIS middleware abstraction layer was used to isolate application software and evolving middleware. Initial prototypes used a beta-test CORBA product.These prototyping efforts proved unsuccessful, largely due to difficulties in using the beta-test product. Although program management acknowledged the need for future CORBA support, a proprietary object-oriented DCE extension was chosen for short-term implementations. Management also did not want to rely entirely on proprietary interfaces.
   The situation was resolved through the addition of a middleware abstraction layer that masked the choice of middleware from EOSDIS application software; it hid differences in object creation, object activation, and object invocation.

Related Solutions

This pattern is related to the Object Wrapper pattern , which provides isolation to and from a single application to a single object infrastructure. The isolation layer pattern provides insulation of multiple applications from multiple infrastructures. This pattern is also related to the Profile pattern , where an isolation layer can be viewed as a particular enterprise profile for the use of middleware.

The isolation layer can be thought of as one the levels in a layered architecture In contrast to most layers, this is a very thin layer that does not contain application objects. Typically, an isolation layer serves only as a proxy for integrating clients and services with one or more infrastructures. The Proxy pattern is described in Buschmann (1996).

Applicability To Other Viewpoints And Scales

The impact of Vendor Lock-In on management is a loss of control of the technology, IT functionality, and O&M budgets to the dictates of vendor product releases. Vendor Lock-In also impacts risk management. The AntiPattern is often accepted with the understanding that future features will be implemented. Unfortunately, these feature often deliver later than expected or needed, if ever.

Vendor Lock-In affects developers by requiring they have in-depth product knowledge. But this knowledge is transient; it will be made obsolete by the next product release. Thus, developers are expected to be on a continuous learning-curve treadmill about product features, product bugs, and product futures.