Step out of integration hell –
Protocol Interception Wrapper

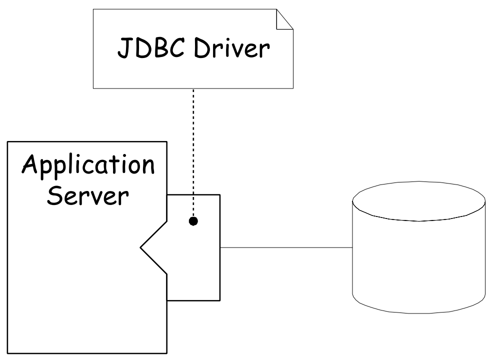
Christian Wege
DaimlerChrysler AG
+49 711 17 92952
wege@acm.org

DaimlerChrysler is a typical IT user rather than an IT producer. We try to base our application systems development on the integration of existing components rather than on custom made solutions. This is part of our effort to limit the number of different technologies within the company. Otherwise they would build up a huge pile of legacy. In our daily work as a strategic IT department this means that we have to find ways to integrate new technologies or products with our infrastructure already in place. Some of the best insights into this job stems from application development projects in which we are involved as part of our job. The Protocol Interceptor Wrapper pattern is the first of a series of patterns which help you to step out of integration hell.

Protocol Interception Wrapper

Context

You build your application from existing commercial-of-the-shelf components. Some of those components can be plugged together via a well-defined interface (like the ones in the Java 2 Enterprise Edition specification) but they are from different sources or vendors (i.e. a hosting system talks to the plugged-in component which implements the interface). For example you might want to understand the details of the protocol between an application server and a JDBC database driver.



Problem

How do you find out how to configure the plumbing mechanism of a hosting system so that it talks correctly to a plugged-in subsystem?

Forces

- Knowing the interface definition between two components is not enough to plug them together if you don't know the specifics of the plumbing mechanism.

- A detailed documentation of this case might not be available (e.g. a combination of components not anticipated by the vendors) or is out of date.
- The definitive specification of the specifics is the source code but it might be unaccessible to you or too complex to understand.
- The configuration options of the hosting system is the main limiting factor for the applicability of this pattern. It must be possible to configure another subsystem instead of the original one.
- The interface may not be well designed which makes it hard to reveal the meaning of the protocol elements.
- To understand the protocol dynamics you have to follow the single steps manually. With a complex protocol this can very easily get intangible.

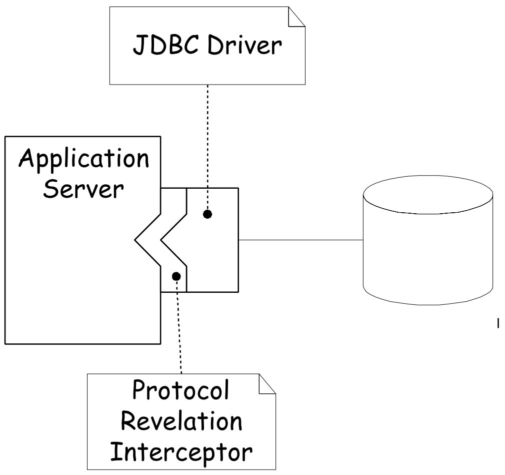
Solution

In place of the original subsystem plug in an interception wrapper which talks the same protocol as the original subsystem. This interception wrapper injects it's logging calls and delegates all calls from the hosting system to the original subsystem. The interception wrapper works much like a specialized proxy which stands in for the original subsystem. [Gamma+95].

Try different configurations of the plumbing mechanism of the hosting system and make it run to see the effects on the dynamics of the protocol between the hosting system and the plugged-in subsystem. Backtrack from what you expect as input for the plugged-in subsystem to configure the plumbing mechanism correctly in the hosting system.

Examples

We had to configure a database JDBC driver plugged into an application server (see figure) and didn't have documentation on how to configure the application server to provide the database driver with the right arguments. The JDBC driver needed a connection string which was put together by the application server based on some configuration parameters. For being able to provide the application server with the right configuration parameters we needed to understand how the connection string was constructed from these parameters inside the plumbing mechanism of the application server. With the Protocol Interception Wrapper we



could set breakpoints at the relevant delegation methods and examine the actually provided parameters. We found out how the connect string and was constructed by the plumbing mechanism of the application server.

In another case we had to understand the protocol between the user manager and the internet portal application framework. The user manager of this framework is responsible for the user and security management of the portal. Our job was to change the authentication to go to another system but keep the rest of the user manager. From reading only the interface we didn’t have enough insight to understand the protocol. With the help of the wrapper we could analyze the dynamics of it.

JInsight is a tool from the IBM alphaworks site to better understand what a Java program is doing [JInsight]. This tool uses an instrumented Java VM to intercept the calls between the components.

COM+ uses interceptors to add transactional capabilities to server-side components. The wrapper intercepts every call to the component to inject the calls to the transaction manager - transparent for the component. [Raj99]

Resulting Context

The Protocol Interception Wrapper pattern provides the following benefits:

1. *Minimal modification of hosting system (Host).*This pattern work independent of the logging or debugging capabilities of the hosting system. The only necessary feature is that the plugged-in subsystem can be configured.
2. *No modification of plugged-in subsystem.* The plugged in component can be used as is. No debugging or logging capabilities are necessary. Thus the component is used in a way very close to a productive operation.
3. *Statistical analysis possible.* Introducing the wrapper into a running system allows to run a large number of tests across the protocol and to analyze the resulting data with statistical methods.

There are some liabilities of the pattern:

1. *The complete Protocol interface has to be implemented by Interception Wrapper.* Implementing an interface means to implement every declared method. This could mean much work for debugging potentially only one method call. As a workaround the PluggedComponent could be subclassed if possible. Then only the relevant methods have to be redefined. Instead of delegating the accepted method calls to a private instance of the plugged component the every redefined method calls its superclasses method. However this workaround only works if the plugged component can be extended in this way.
2. *Introducing the Interception Wrapper might degrade the performance of the system.* The additional level of indirection itself consumes performance. More important is to implement the logging in a performant way. Simple output to the system console or to a file might be too expensive.

Related Patterns

The interceptor is some kind of Proxy which adds the functionality of making the dynamics of the protocol visible. Kent Beck writes about an effort to refactoring the protocol between two classes in [Beck96]. Schmidt describes the Interceptor architectural pattern which "allows services to be added transparently to a framework and triggered automatically when certain events occur" and the Component Configurator pattern which "allows an application to link and unlink its component implementations at run-time without having to modify, recompile, or statically relink the application". [Schmidt+00]

Credits

At ECOOP 2000 Wim de Paauw explained me some details about JInsight. Klaus Kiehne showed me the fastest way to create a custom JDBC driver as described in the examples. Special thanks to Klaus Marquardt who shepherded this paper.

References

[Beck96]	Kent Beck: Smalltalk Patterns: Best Practices. Prentice Hall, 1996.
[Gamma+95]	E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of reusable Object-Oriented Software*. Addison-Wesley, 1995.
[JInsight]	JInsight at alphaworks. Online at http://www.alphaworks.ibm.com/tech/jinsight
[Meszaros+98]	Gerard Meszaros, Jim Doble: A Pattern Language for Pattern Writing. In *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.
[Raj99]	Gopalan Suresh Raj: COM+, 1999. Online at http://www.execpc.com/~gopalan/com/complus.html
[Schmidt+00]	Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for concurrent and Networked Objects*. Wiley&Sons, 2000.