Legacy Wrapping

Paul Asman, Federal Reserve Bank of New York, November 2000

In 1999 and earlier, many software shops put new development on hold as they successfully moved existing applications into year 2000 compliance. Even object-oriented shops brought some procedural applications into compliance rather than replace them. The effort points to the widespread and continued use of legacy applications. The success points to the strength these applications retain.

Object orientation and legacy applications have long coexisted. When object orientation first began making inroads into mainstream application development, some OO proponents – notably sales representatives – recognized both the functionality of legacy applications and the investment that had been made in them. These proponents suggested wrapping rather than abandoning the older applications. Other than database management systems, though, few were wrapped.[1] More recently, OO developers have proposed wrapping services of legacy applications, and making these services available as components. But once again, such wrapping has not become widespread.

This should not be surprising, for the rewards of legacy wrapping are elusive. There are technical issues – the process is difficult – and there are social issues, notably an attitude that writing new code is worthier. The four patterns that follow show ways of reaching the rewards. While all the patterns aim towards successful wrapping, they have different scopes. The first two patterns, To Wrap or Not to Wrap and Respect for Wrappers, apply to any legacy wrapping, whatever the context. (Initial attempts at legacy wrapping gave objects access to non-OO applications. Now, however, a wrapper might give Java objects access to a C++ application.) To Wrap or Not to Wrap, the core pattern, offers ways to determine whether wrapping is appropriate. Respect for Wrappers deals with attitudes towards wrapping.

Single Wrappers and Multiple Wrappers, the third pattern, limits itself to wrappers that are themselves part of object-oriented systems, although it applies whether they are wrapping OO or non-OO applications. The final pattern, Wrapping, Not Adapting, is even more narrow: it applies only when OO applications wrap non-OO applications. No patterns in this limited language offer ways of making object-oriented systems accessible to legacy applications as such. That is a different topic, applicable to different situations, requiring

[1] A Gartner Group survey in 1995 (K. Schick, "Wrapping Gifts From the Past," 26 July 1995) "indicated low usage of wrapping techniques … vs. alternative legacy reuse approaches."

different solutions, and yielding different results.[2] It is also of limited utility. The patterns of this language leave legacy applications intact, and require no legacy programming skills to implement.

Pattern names are distinguished by the Arial font.

[2] See, e.g., E. S. Flint, "The COBOL jigsaw puzzle: Fitting object-oriented and legacy applications together," in *IBM Systems Journal*, vol. 36, no. 1, 1997, pp. 49 – 65, available at http://www.research.ibm.com/journal/sj/361/flint.html. Flint proposes three sorts of wrappers: object, procedural, and combination. Procedural and combination wrappers, unlike object wrappers, require changes (albeit "minimal") to the legacy applications. Object wrappers may sometimes benefit from changes to legacy applications but do not require them.

To Wrap or Not to Wrap

Wrappers shouldn't be used everywhere. You must determine whether your situation calls for wrapping.

***

Legacy code is available to meet some need, and you are considering wrapping it. But you need to know whether it's worth the effort.

On 1 January 1999, eleven European nations relinquished the independence of their currencies and adopted the euro. As of that day, the Italian lira, for example, was no longer an autonomous currency, but simply a way of referring to approximately 0.000516456899 euro.[3] As of that day also, Italy, along with Belgium, went from a currency with no decimal places to a currency with two.

On 1 January 2002, euro banknotes and coins will come into circulation, and on 1 July 2002 national banknotes and coins of the participating nations will lose their legal status. In the interim period, though – from 1 January 1999 through 30 June 2002 – banks and other institutions need to handle the euro and its representations in national currencies interchangeably. Were there no interim period, banks could simply have converted from national currencies to the euro. But since the euro and national currencies coexist, banks do not have that luxury.

Banks in participating nations could have rewritten their applications to handle two currencies, but surely this is too high a cost for so short-lived a reward. Italian banks handled a single currency well before 1999, and from July 2002 forward they will handle a single currency again. Wrapping is an expedient solution. A wrapper can provide consistent conversion from one representation of a currency to another without rounding errors. A wrapper can mediate between integer storage for the lira, with no decimal places, and fixed decimal for the euro, with two.

Dual currency systems for nations that have adopted the euro are temporary, and this impermanence favors wrapping. There is no need for an elaborate fix to a problem that will go away on a compact, set schedule. But wrapping has also been used for problems that will not go away so quickly.

Many organizations have chosen windowing to achieve year 2000 compliance. With windowing, years continue to have two-digit storage, but applications access them through wrappers that treat them as having four digits. In calculations and displays, the years stored as 50 through 99 can be treated as 1950 through 1999, and the years stored as 00 through 49 treated as 2000 through 2049. At some point (2020 or sooner for

[3] Rates are fixed in the other direction: there are exactly 1936.27 lire to the euro.

applications that handle 30-year mortgages, in this example), this solution will have run its course. Nonetheless, for many organizations it is a reasonable stopgap measure.

How does an organization decide whether wrapping is appropriate? In the case of the euro, where rewriting applications brings dubious benefit at high cost, it's easy. Furthermore, the problem is common, and wrapping solutions have been worked out, published, and even sold.[4] Similarly, wrapping solutions have been worked out, published, and sold for year 2000 compliance. Those solutions were appropriate for organizations that did not have the resources to make more lasting changes to their otherwise functional systems before 2000.

Sometimes, though, wrappers are ill advised. Consider a case in which two companies in the same line of business merge. Each company will have its own applications to meet common business needs. The newly unified company, however, will want to present one image to its customers. It could wrap some applications so that they will respond to input as presented to the other, unwrapped applications. All applications would then remain active. But all applications would also require maintenance, and the wrappers themselves would introduce even more software to maintain. The cost is simply too great. It's cleaner and cheaper to convert from some applications and retire them.[5]

Even when wrappers seem cheaper they may still be inadvisable. The examples above of appropriate wrappers assume that the applications being wrapped work successfully *on their own terms*. The euro example assumes that the banks had successfully handled a single currency, and the year 2000 example similarly assumes that the systems being wrapped were otherwise functional. But that might not be the case, and if legacy systems do not function properly, wrapping cannot be recommended. Wrappers are useful for gaining access to legacy code, not for repairing it. Using wrappers to fix legacy code introduces a level of indirection in maintenance, and is therefore bad programming practice. The same holds for wrapping code that relies on unsupported technology.[6] Bad or unsupported code should be wrapped only as a last resort. Such wrapping shares the faults of any quick-and-dirty patch, and it's better to hack or start over.

Having sound legacy applications built on supported technology may still not be enough. If you are wrapping components, you need to know what the components are and what they do. Sometimes that's easy, as it is when wrapping a tried-and-true relational database management system.[7] But sometimes applications are so opaque that it simply

---

[4] Sapiens, for example, claims widespread use for the wrapping it provides clients through its EuroMigration tool. For a good description of its wrapping strategy, see its white paper, "Sapiens EuroMigration: 1999-2002 System Migration Strategies," accessible through http://www.sapiens.com/en/solutions/euromigration/.

[5] A different wrapper may be useful in this scenario, though. If customers of both applications require the interfaces that they had before the merger, wrappers can give surviving applications the appearance of deprecated ones.

[6] Perdita Stevens and Rob Pooley argue for this in Divide and Modernise, a pattern posted at http://reengineering.ed.ac.uk/DivideAndModernise.html. It is part of a systems reengineering patterns project found at http://www.reengineering.ed.ac.uk/.

[7] When an object-oriented application uses a relational database, a legacy application – the RDBMS – is wrapped, even if a new database is created for the OO application.

cannot be done, as is often the case with applications written in-house by developers who eschew comments and have since departed. Those applications should not be wrapped. There is also a middle ground where identifying components is possible but difficult. If that occurs for a functionally sound and supported legacy application, you may want to attempt to identify its components, perhaps by applying Find the Architecture in the Development Structure[8]. This pattern addresses the question, "How do you recover the software architecture of a legacy system that is not explicit, or has been lost in the details of a long maintenance lifetime?"

Once you have identified the components, you may be unsure whether they can fit into the framework of the new application, or whether they meet current expectations. An issue of the Butler Forum's *CBD Journal*[9] recommends creating two models, one of the systems as they are, the other of the system that is to be. "[T]hen map the 'as is' model of current systems against [the 'to be' model] to see if they can be reused."[10] If the component you're considering wrapping has no place in the new model, do not wrap it.

If legacy applications survive these tests, they can be wrapped. Wrapped components will be functionally sound and supported, and will meet current needs. Robust COBOL applications often meet these qualifications, and many have been successfully wrapped with GUI front ends. Similar, relational databases have often been wrapped to provide a persistence layer for object-oriented systems.[11] A business might wrap a relational database because it already owns the product and has substantial in-house expertise in it. The business may also be wary of object databases, or untrained in their use. In this situation, wrapping the RDBMS makes sense.

Therefore:

Before wrapping a legacy application, see if it is sound and supported. If it is, and it can provide functionality that you need at a reasonable cost, wrap it.

***

Even if a case can be made for wrapping, developers may disdain it, or work in an environment in which wrapping is thought to be a lesser activity. To promote wrapping, see Respect for Wrappers.

---

[8] O'Callaghan, Alan, and Clazien Wezemen, EuroPLoP '99, posted at http://www.argo.be/europlop/Papers/Final/Ocallaghan.htm.

[9] *CBD Journal*, December 1998. This report was briefly available to the public over the Web.

[10] Note that the 'to be' model is independent of the legacy system. The architecture of the legacy system should not influence the architecture of the new system.

[11] A number of patterns and pattern languages have been written on OO-RDBMS interfaces. See, for example, Kyle Brown and Bruce G. Whitenack's "Crossing Chasms: A Pattern Language for Object-RDBMS Integration," in *Pattern Languages of Program Design 2*, edited by John M. Vlissides, James O. Coplien, and Norman L. Kerth, Addison–Wesley, Reading, MA, 1995 and Wolfgang Keller's "Object/Relational Access Layers" from EuroPLoP '98, available through http://www.coldewey.com/europlop98/Program/workshop2.html#Keller.

If you decide to wrap a legacy application, you need to select a way of wrapping it. See Single Wrappers and Multiple Wrappers.

If you decide not to wrap a legacy application but still wish to replace it, you may find Incremental Delivery[12] helpful.

[12] This pattern is posted on the WikiWikiWeb at http://c2.com/cgi-bin/wiki?IncrementalDelivery.

Respect for Wrappers

Best Adapted Screenplay

Best Original Screenplay

Which Would You Rather Have Won? [13]

You have applied To Wrap or Not to Wrap and have decided to wrap a legacy application, but your development culture looks down on anything but new work.

***

Software developers tend to favor originality over reuse, even when existing work would seem too compelling to ignore. For developers to incorporate existing work into new applications, the experience must feel rewarding and be rewarded.

When objects first began to penetrate corporate consciousness, their boosters touted reuse. The first issue of *Object Magazine*, for example, promised to put its readers "on the fast track to object software development and reuse."[14] Development and reuse were on a par.

But things haven't worked out that way. When we as developers speak of reuse, we usually mean that we create classes (or, more recently, components) that others are able to use. We don't mean that we reuse the work of others, although we do reuse that which no one now thinks to develop, such as standard GUI widgets. Moreover, if someone else has done the work we are asked to provide, we probably don't know. Nor do those asking us.

If we do know that the work has been done, we still might not use it. Few shops reward those who use the work of others as generously as they reward those who create work anew. Even if those commissioning an application tell us of some classes or components that have already been built, we may still resist. What creates value for a manager – prior

[13] The Oscar® statuette is ©Academy of Motion Picture Arts and Sciences®. The video box covers are taken from Amazon.com. The actual names of the two awards are "for the best screenplay based on material previously produced or published" and "for the best screenplay written directly for the screen."
[14] Friedman, Richard P., *Object Magazine*, Vol. 1, No. 1, May/June 1991, p. 4.

investment, avoiding new work when funds are scarce, or office politics in general – may not motivate us as developers. And we can always say that there's something wrong with the existing work. There always is.

However, the ethos of the pattern community includes "an aggressive 'disregard for originality'."[15] An extension of this ethos can create an environment in which developers recognize that value in legacy code should be appropriated. Furthermore, just as members of the pattern community are often evangelical, an extension of this evangelism may help bring management to reward legacy reuse as highly as it rewards creativity. In many shops, proponents of wrapping will be able to show management that wrapping is already used, although it may not be labeled as such. Using relational database management systems for persistence and using windowing for year 2000 compliance are both wrapping solutions, as To Wrap or Not to Wrap shows.

Developers may still disdain wrapping even after management has accepted its virtues. Material rewards would go a long way towards solving this. Many shops have award programs through which management can acknowledge exemplary work. These awards are often publicized, as a way of letting others know to whom - and for what – they should be looking as models. Other shops can put such programs into place, and all can use them to promote wrapping.

Once wrapping becomes established, this pattern can be skipped.

Therefore:

Since legacy wrapping can create value out of existing value, more cheaply and efficiently than creating anew, it should be promoted and rewarded. Promotion and reward will be unnecessary for (though welcome to) those who use patterns, once they see that pattern use and wrapping share virtues.

***

If wrapping is justified – and To Wrap or Not to Wrap should help determine this – you need to select a way of doing so. See Single Wrappers and Multiple Wrappers.

An organization embracing legacy wrapping must change its ways. It will face issues discussed by Opdyke in Fowler's *Refactoring*[16] and by Kane, Opdyke, and Dikel in "Managing Change to Reusable Software."[17] Refactoring and legacy wrapping are both types of reuse, and the organizational forces and solutions overlap.

---

[15] Coplien, James O., *Software Patterns*, SIGS Books & Multimedia, New York, 1996. Coplien quotes Brian Foote.

[16] Fowler, Martin, *Refactoring: Improving the Design of Existing Code*, Addison–Wesley, Reading, MA, 1999.

[17] Kane, David, William Opdyke, and David Dikel, "Managing Change to Reusable Software," PLoP '97, posted at http://jerry.cs.uiuc.edu/plop/plopd4-submissions/P17.html.

Single Wrappers and Multiple Wrappers

You have applied To Wrap or Not to Wrap and identified an application that should be wrapped. Since you work in an environment that affords Respect for Wrappers, you will wrap it.

***

When encapsulating a legacy application whose services may be replaced incrementally, you can use just one wrapper for the entire application or several wrappers, one per service. You can spend a lot of time deciding which way to go.

In the context of this pattern, a wrapper is part of an OO system, and it calls a legacy API. The process has the form shown in Figure 1.[18]

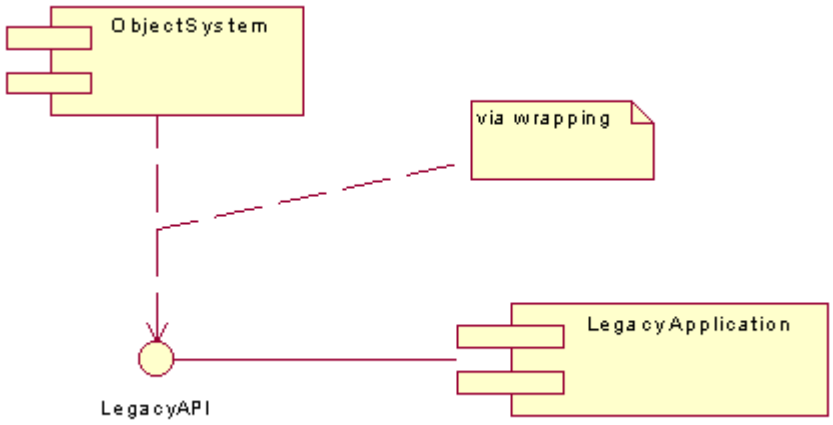

Figure 1: The Wrapping Process

The Figure does not show whether a single wrapper should encapsulate an entire legacy application, or whether multiple wrappers should wrap an application's services individually. The literature diverges on this issue. Mularz, for example, treats a legacy application as "an important functionality," using the singular even while suggesting incremental replacement,[19] and Winsberg writes that an application wrapper "surrounds a complete legacy system, both code and data, by emulating an end-user sitting at a character terminal."[20] On the other hand, Aronica and Rimel, after offering different

[18] This pattern and the next use UML conventions without explanation, in the hope that they are largely self-explanatory and in the belief that those who do not follow all the notation do not need all the details.
[19] Diane E. Mularz, (Legacy) Wrapper, part of her "Pattern-Based Integration Architectures," in *Pattern Languages of Program Design 1*, edited by James O. Coplien and Douglas C. Schmidt, Addison–Wesley, Reading, MA, 1995.
[20] Paul Winsberg, "Legacy Code: Don't Bag It, Wrap It," in *Datamation*, May 15, 1995.

approaches to wrapping legacy code, define systems reengineering as "reengineering and replacing legacy systems either completely or one subsystem at a time."[21]

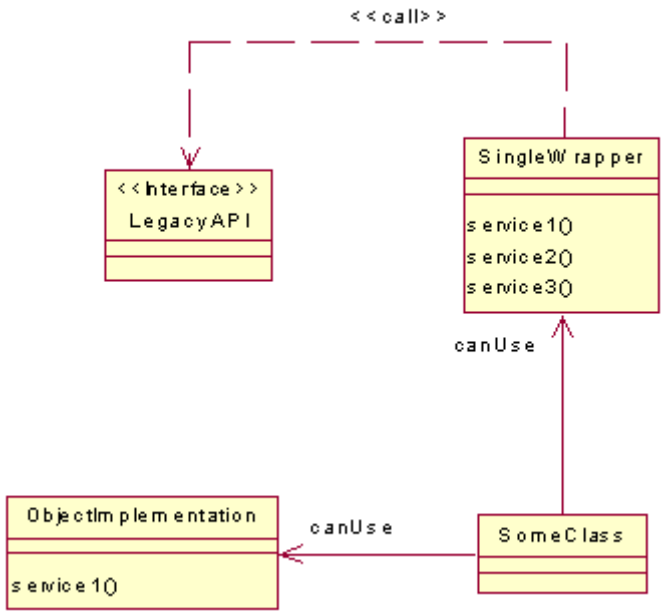

Figure 2: Single Wrapper

A single wrapper encapsulating a legacy application for use in an OO system has the form shown in Figure 2. In this model, one wrapper contains all the methods that invoke the legacy API. When an object requires something from a legacy application, it sends a message to the wrapper, which in turn calls the API. When an alternative implementation of some service becomes available (shown for service 1), the system can send a message to that implementation instead.

Note that this structure does not require wrapping all the services of the legacy application. It only requires that one wrapper provide methods for all the services that are wrapped. This is compatible with the approach recommended by Mowbray and Malveau in Object Wrapper, one of their CORBA design patterns.[22] Mowbray and Malveau state that "the OMB IDL interface should only expose those attributes and operations that are needed by other CORBA clients and services." If you follow this advice, though, be careful that you do not limit the wrapper unnecessarily. When someone else needs some other functionality from the legacy application, the wrapper that you created may not serve the new client.

[21] Ronald C. Aronica and Ronald E. Rimel, Jr., "Wrapper Your Legacy Systems," in *Datamation*, June 15, 1996.
[22] Mowbray, Thomas J., and Raphael C. Malveau, *CORBA Design Patterns*, John Wiley & Songs, New York, 1997.

Note also that the single wrapper, whether complete or limited, does not implement the API. Instead, the wrapper *calls* the API, as indicated by the stereotype on the dependency relationship in Figure 2. The legacy application implements the API, not the object system. The model in Figure 3, which shows the wrapper realizing the API, is invalid.

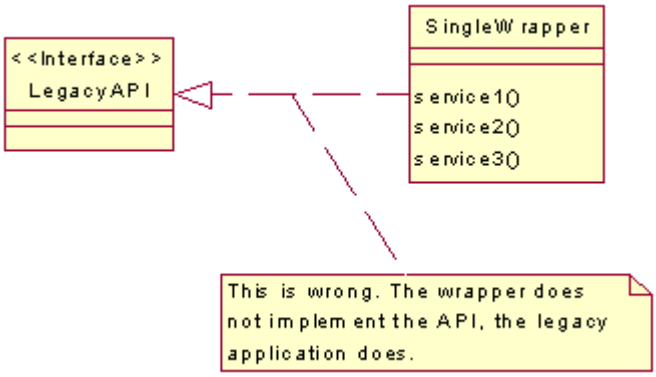

Figure 3: An Incorrect Model

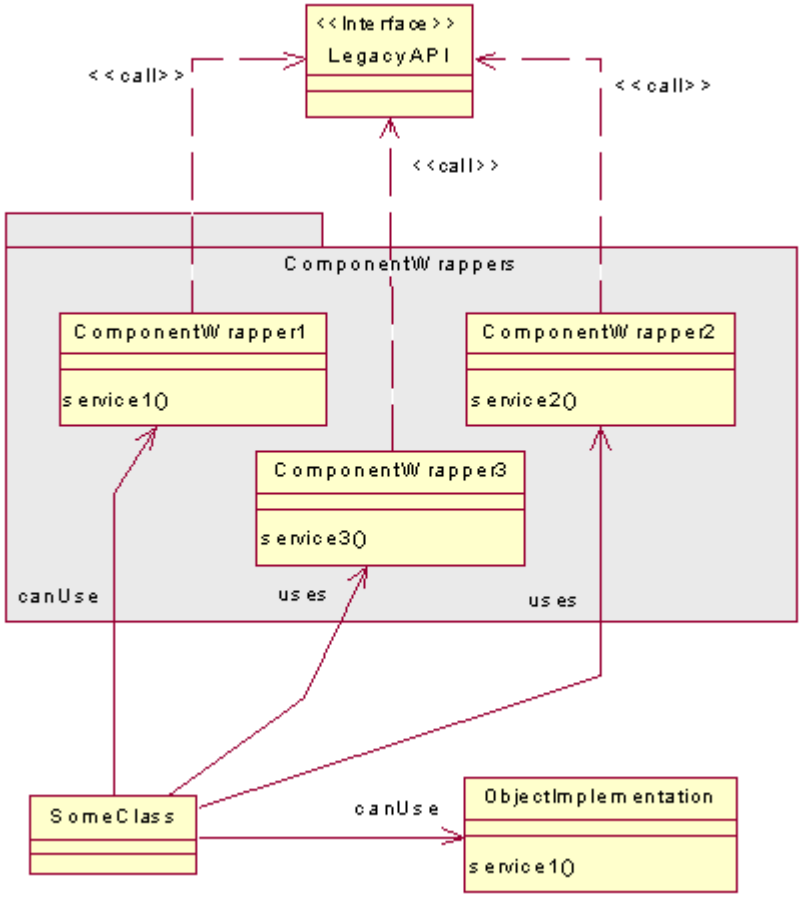

Figure 4: Multiple Wrappers

The alternative to a single wrapper is a multiplicity of wrappers, each encapsulating a particular service performed by a legacy application. Such a schema may have the model shown in Figure 4. In that model, each member of a set of component wrappers encapsulates some service provided by the legacy application, and each calls the legacy API. When a new implementation of a service becomes available (again shown for service 1), an object can use the new implementation. The corresponding wrapper becomes obsolete and can be deprecated. The object continues to use the component wrappers for which no new implementation is available (services 2 and 3 in Figure 4).

Both models equally support incremental replacement of legacy services. In Figures 2 and 4, a wrapper, whether alone or one of many, implements a service through a separate method for that service. When an object no longer needs a legacy application to provide a service, it simply changes one method invocation to another. This would be true even if the wrapper were badly coded and contained only one method for all services. The number of wrappers makes no difference to the process. Neither option aids or threatens incremental replacement of components.[23]

Therefore:

There's nothing about wrapping *per se* that would lead you to use one or many wrappers to encapsulate a legacy application. Determine the multiplicity and identity of wrapper classes, then, in the same way that you determine the multiplicity and identity of classes in any other part of the object domain.

***

Once you've decided to use wrappers, you need to write them. If you are wrapping non-OO applications, use Wrapping, Not Adapting to avoid common pitfalls.

[23] If there is some reason to present several wrappers as one, the *Façade* pattern can be applied to present a united interface. See Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison–Wesley, Reading, MA, 1994.

Wrapping, Not Adapting

You've applied To Wrap or Not to Wrap and have decided to use wrappers to encapsulate a non-OO application. You've chosen between Single Wrappers and Multiple Wrappers and are ready to implement.

***

Some well-known wrapping techniques clearly apply when objects wrap other objects. But you are using objects to wrap a non-OO application, and you don't know whether these techniques will work. You need to distinguish those that will from those that won't.

Your first inclination in building a wrapper around a non-OO application may be to consider two patterns found in *Design Patterns*,[24] Adapter and Decorator. Both are "also known as" Wrapper. A closer look, however, is not encouraging.

Adapter operates solely in the context of objects. In its class version, the adapter inherits from what it adapts. As laid out in *Design Patterns* (and adapted to UML), it has the form shown in Figure 5:

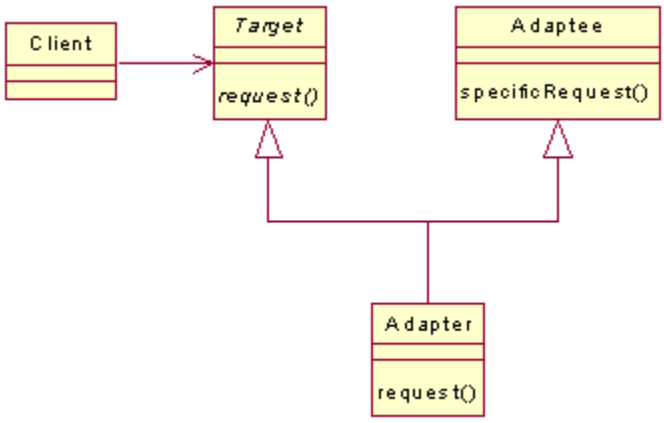

Figure 5: Class Version of Adapter

A legacy wrapper of a non-OO application, on the other hand, wraps a legacy API. Since a legacy API is not an object, a legacy wrapper cannot inherit from it. Hence the class version of Adapter cannot provide the structure for wrapping non-OO legacy software. Indeed, the class version of Adapter is orthogonal to non-OO legacy wrapping, and the two can be used together. Consider for example the last of the "known uses" of Adapter, Bertrand Meyer's "Marriage of Convenience"[25]. There, a FixedStack inherits from both Stack and Array. Such a FixedStack could use a legacy wrapper to receive an array of

[24] Gamma et al., *Design Patterns*, *op. cit*.
[25] Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice Hall, New York, 1988.

values, populate a SQL statement, and pass the array in an update request to a relational database. Adapter would work with the legacy wrapper as shown in Figure 6.

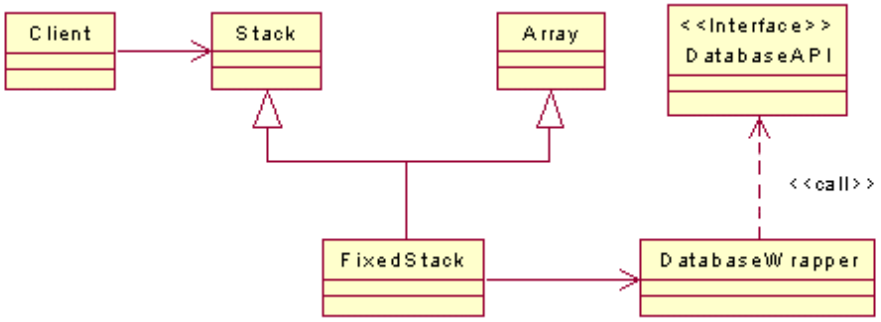

Figure 6: Class Version of Adapter Combined with Legacy Wrapper

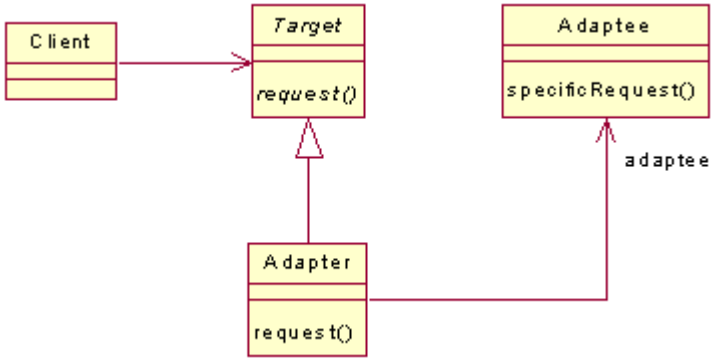

Figure 7: Object Version of Adapter

The object version of Adapter may seem more promising, for there the adapter does not inherit from what it adapts (Figure 7). But once again, this does not meet the needs of legacy wrapping of non-OO applications shown in Figures 2 and 4. As with the class version of Adapter, there is a difference between an adapter contained within an object framework and a wrapper that links objects with non-objects. The object version of Adapter specializes a target to give it access to an adaptee, access the target does not provide without specialization. Objects built from this pattern can work with, but not as, legacy wrappers. In non-OO legacy wrapping, an object needs something that objects do not provide. Decorator has similar limitations.

While Adapter and its kin may not apply directly to wrapping non-OO legacy applications, legacy wrappers are still objects, and familiar principles of object design apply to their construction. In Wrapper Façade, Schmidt presents two main implementation steps: “Identify the cohesive abstractions and relationships among existing functions,” and “Cluster cohesive groups of functions into wrapper façade

classes and methods."[26] These are good principles of object design that apply to wrapping non-OO applications. They apply, for example, to wrapping a relational database management system. RDBMSs do not encapsulate data with methods in the manner of object-oriented languages, but for some functions, such as referential integrity, they have the same effect. An RDBMS will not write values to the database if the values do not preserve referential integrity.

However, an RDBMS does not encapsulate all its functionality as objects do. Schmidt's third (and final required) implementation step is, "Determine an error handling mechanism." A call from an OO system to an RDBMS requires special handling. Relational database management systems send return codes along with data values, and a wrapper of an RDBMS must handle both. Some return codes require no action, but others require notifications and rollbacks. Legacy wrappers must include methods that handle these situations. In a purely object system, return codes and data values would arrive separately, while in an RDBMS they arrive together. Hence wrappers must separate them and pass them on to the appropriate objects.

Therefore:

Don't think that you can rely solely on object-oriented patterns when wrapping non-OO legacy applications. Wrapped applications are not OO components. Legacy wrappers must recast the products of non-OO design into objects before patterns of object interactions apply.

***

Once a wrapper is built, you can specialize it to extend the encapsulated functionality. This follows the solution proposed by Foote and Yoder in Programming-by-Difference, also known as "Wrap It, Don't Bag It."[27]

[26] Douglas C. Schmidt, "Wrapper Façade," *C++ Report*, Vol. 11, No. 2, February 1999, accessible through http://siesta.cs.wustl.edu/~schmidt/patterns-ace.html. This pattern "encapsulates low-level functions and data structures within more concise, portable, maintainable, and cohesive object-oriented interfaces."

[27] Programming-by-Difference is part of "The Selfish Class," in *Pattern Languages of Program Design 3*, edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann, Addison–Wesley, Reading, MA, 1998.

Acknowledgements

When I submitted a version of this paper to PloP 2000, I knew that it had weaknesses, although I didn't know what they were. John Vlissides, who shepherded the paper, saw the major weaknesses and helped find solutions. John also brought in Bill Opdyke, who identified additional weaknesses and remedies. I am grateful to them for their work. They bear no responsibility for the weaknesses that subsequently surfaced.

A workshop at PloP exposed these new weaknesses and helped find solutions to them. I am grateful to the participants in that workshop as well, and exonerate them from any responsibility for the most recent weakness that I have introduced.

Disclaimer

The views expressed in this paper are those of the author and do not necessarily reflect the position of the Federal Reserve Bank of New York or the Federal Reserve System.