Design Patterns for Wrapping Similar Legacy Systems with Common Service Interfaces

David E. Millard, Yvonne Howard, Swapna Chennupati,
Hugh C. Davis, Ehtesham-Rasheed Jam, Lester Gilbert, Gary B. Wills
School of Electronics and Computer Science, University of Southampton, UK
{dem,ymh,cs,hcd,erj2,lg3,gbw}@ecs.soton.ac.uk

Abstract

Web Services are increasingly being used to create a wide range of distributed systems, many of which involve legacy software. Developing service interfaces for these legacy systems can be difficult, as for interoperability reasons it is advantageous to use a common service interface that is independent of the particular legacy system behind it. This enables other services to interoperate with like legacy systems regardless of their implementation. Unfortunately, similar legacy systems can offer subtly different functionality from each other, making agreeing on a common interface difficult. This paper introduces three design patterns for managing this problem: Lowest Common Denominator*,* Most Popular *and* Negotiated Interfaces*. It formally presents these patterns and reflects on how they have been used within the domain of e-learning to wrap legacy systems that function as databases of objective questions.*

1. Introduction

Distributed systems pose unique challenges for software engineers; components need to agree common protocols, data models, and paradigms of communication in order to work together. Often these components are created by disparate teams of developers, and agreeing on these interoperability issues can be difficult.

Service-Oriented Architectures (SOAs) aim to simplify this problem, by providing a framework in which components publish and consume services using standard protocols and well-defined interfaces. This means that systems can be developed in a modular fashion, and can later be extended to adapt to new challenges, or to provide new functionality [17].

Service-orientation is a philosophical approach to creating distributed systems, but there are a number of SOAs, each using different standards and approaches to providing them at an implementation level. For example, Web Services are based on SOAP, GRID Services are based on OGSI, and REST services are based on HTTP and XML. All of these approaches share a common problem, however, that it is often necessary to wrap existing legacy software in a service interface to make it easily accessible to clients that are using the SOA.

Wrapping a single legacy application can be a simple process of capturing all the system's functionality in a new service interface and then writing some intermediary code that converts from service calls to the proprietary API. However, many legacy systems provide similar functionality and wrapping them all in specific Web service interfaces does not encourage reuse. Different legacy systems can also provide overlapping functionality, and large service interfaces seem bulky and inappropriate.

A better solution would be to devise a number of smaller service interfaces which legacy systems can support as appropriate. The granularity should not be too small, however, (for example, one method per interface) as this adds overhead to the service design, and can be as big an obstacle to reuse as large interfaces (because of the difficulty of finding many services to fulfill what is conceptually one larger service, and the increased risk that one of more small services will be unavailable therefore preventing the larger conceptual service from functioning).

Service designers thus have to balance granularity, defining service interfaces that are complete and therefore robust, but at the same time consolidating the functionality of legacy systems into only a few interoperable interfaces. Design patterns are a semi-

formal method of capturing design practice so that it may be shared and reused in other design exercises.

In this paper we present three design patterns for wrapping legacy systems with common service interfaces. We have developed these patterns during our own work in the domain of e-learning, and e-assessment in particular. E-learning is a rich domain that is beginning to embrace service architectures, and is replete with legacy systems which contain valuable data such as questions, course structures, and student information. We believe that the methods that we have explored and generalised into design patterns will be valuable to other service developers faced with a similar legacy software environment.

Section 2 provides background information on existing Web service design practice, and the use of design patterns within software engineering. Section 3 describes the motivation and context for our work - services development in the domain of e-assessment - as evidence for the need for these types of service pattern. Section 4 presents the patterns themselves: *Lowest Common Denominator*, *Most Popular* and *Negotiated Interfaces*, using the 'Gang of Four' structure. Section 5 describes our experiences and observations from our own implementations of the patterns. Section 6 concludes the paper and summarizes our contribution.

2. Background

There are a number of distinct architectures that subscribe to the service-oriented paradigm. Web services have received a great of recent attention, and are defined around a set of standards (such as SOAP, WSDL, UDDI) developed by the W3C to make functionality available over the Web as simply as data Web services are mostly not secure and stateless. This mirrors the Web approach, and is good for non-sensitive information and ad-hoc systems.

GRID services, on the other hand, assume a highly secure environment, and rely on certificates and authentication bodies to operate [7]. This approach to security makes it possible to build virtual organisations (that exchange and manipulate sensitive information) but can be prohibitively heavyweight for developers wishing to build simpler services and applications.

These two technologies are becoming more closely defined and a new generation of Web Service standards (such as WS_Security) is now being introduced to add a standard layer of authentication and security to Web Services. This will make Web Services more attractive for systems builders as it is possible to build virtual organisations using relatively lightweight middleware.

A third approach to service provision is represented by Representational State Transfer (REST) [6]. This is the name for a methodology rather than a set of standards, where HTTP and XML are used to send and retrieve data to a remote script or application living on a Web server. Web sites such as Google which offer both a REST and SOAP interface report that most activity is through the REST interface, indicating that REST may be good enough for much of current service-oriented practice.

Whichever service architecture is chosen, there remains the common problem of service design, choosing how to decompose a system into co-operating services such that the services are atomic, reusable, and work efficiently together in some greater context.

2.1. Web Service Design Methodologies

Dijkman and Dumas [5] suggest that there are three characteristics that differentiate Service from Component-based design: Services are developed by autonomous teams, they have a coarser granularity, and they are driven by specific business processes.

A number of researchers have suggested that the tight binding between enterprise practice and service workflow can be used to model and develop services. Martin et al. [10] suggest that the best way to implement Web Services in an enterprise is to start with a component-based architecture that exposes business process level services as Web services. Quartel et al [12] use design milestones to develop Web services from explicit business practices.

Others have also explored this type of modeling approach. Wada et al [16] construct a model of the domain and then use this to derive an object design; this kind of modeling can also been used with SOAs to validate a design as fit for purpose [1].

These methods focus upon developing services from a model of the problem domain, but sometimes it is useful to capture actual design strategies for common problems, rather than to reinvent them through detailed modeling.

2.2. Design Pattern Methodology

Design Patterns are a method for effective communication of design rationale, to aid people in reasoning about what they do and to help them understand why they do it in a given context. Schmidt *et al.* propose writing patterns to concentrate on recording the essential patterns successful developers use [15]. Schmidt *et al* also suggest that this is motivated by a number of values:

- *Success is more important than novelty.* It is not just a matter of recording novel ideas but proven patterns that work.
- *Emphasis on writing and clarity of communication.* Patterns are written in a concise standard format to aid communication
- *Quality validation of knowledge.* Software development can be a creative process, with implicit knowledge imbedded in it. Patterns help expose this knowledge.
- *Good patterns arise from particular experience.* Patterns are best developed from the collective experience of a community of developers.
- *Recognizing the human dimension in software development.* Patterns help to recognize the importance of the developer in creating effective software.

Beck *et al* describe their industrial experience which showed that design patterns were very useful for transcending the reuse of personal knowledge to the sharing of knowledge among developers [2]. They found patterns were an effective shorthand for communicating complexity in software development, that they encouraged the use of good practice, and provided a compact means of capturing the essential element of a design. Beck *et al* also make the point that good design patterns are difficult to write for developers who find it difficult to abstract out the key concepts.

Cline makes the point that design patterns are written and categorized by people who really understand them, which can make it difficult for new people to learn where to find relevant patterns [3]

Gomaa *et al* have used the Unified Modeling Language (UML) to describe the components of an interaction pattern [9]. Their definition of a pattern is one that describes a recurring problem, its solution, and the context in which it applies. They used this broad definition to provide a number of patterns for the ways in which components communicate within a client/server system.

Schmidt and Buschmann recognize the synergy between patterns, frameworks, and middleware, yet suggest that there is no hierarchy in the relationship of patterns to frameworks or to middleware [14]. They describe frameworks as a concrete instantiation of a number of patterns, where the patterns steer the design and use of the framework.

While there is no fixed format for describing patterns, they do have four essential elements: a name, a description of the problem, a proposed solution, and a list of consequences [13]. The most common approach to describing patterns is given by Gamaa, *et al* who are commonly referred to as the 'Gang of Four' [8]. As well as the motivation section which includes the rationale for the pattern and a consequences section for recording the trade-offs when using the pattern, the format also records the participants in the pattern, their responsibilities, and their collaborations.

3. Motivation

Our design patterns are motivated by our work on the FREMA project, which is part of the JISC e-Framework initiative [11].

The e-Framework is a collection of services that work together to support applications in the domains of e-learning, e-science, e-research and e-administration. At present it is a mainly a political construct, at the centre of the JISC e-learning strategy, but there are a number of current projects with the aim of defining and/or creating services to populate the framework.

FREMA is a Community Reference Model for the area of e-assessment (shown in Figure 1). It provides a number of descriptions of services within the e-Framework and how they function together to support assessment activities. It is community-based in that it aims to provide a Web forum where new service-designs can be authored, discussed and eventually promoted to full reference model status.

While the current version is a Web site based on an ontological database of resources, the next version will be a semantic wiki fully in the control of the assessment community.

FREMA takes an agile view of service development (emphasizing a rapid and lightweight development cycle). In FREMA, Use Case diagrams are used to capture common problem scenarios within the assessment domain, and these are then converted into a set of Service Responsibility and Collaboration (SRC) cards. SRCs are a high-level, abstract view of a service, which lists all the responsibilities of a service and the collaborations with other services that are needed to fulfill them. UML 2.0 sequence diagrams are used to describe how a number of SRCs work together to fulfill the broader scenario described in the Use Case.

The domain of e-Assessment is a brown field site, in that there many existing systems, protocols and standards in the area. Services must work alongside, or wrap, this existing software if they are to be accepted into real practice and used with current systems.

In a number of cases we had been forced to tackle this problem. The issue is that there is often more than

one software system that fulfills the responsibilities of a given Service. We have looked in particular detail at the area of item banks (open databases of questions), and how different item banks can be wrapped by common query services.

Examples of item banking software include TOIA (a sophisticated Item Management system)[1], E3AN (a simple database of questions adhering to QTI standards [4] and SPAID (a JISC system for storage and packaging of items) [18].

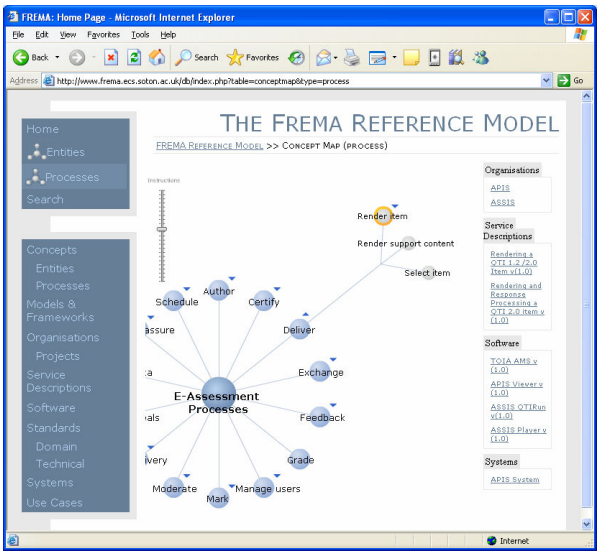


Figure 1: FREMA Web Site

We have made the observation that even for similar systems the intersection of functionality can be small. In terms of service design this means that it is often not possible to have one definitive common interface. It also means that a non-definitive but common interface (covering the intersection of functionality) may not capture the core functionality of either system.

It is necessary to come up with strategies to cope with this problem. We have thus developed three design patterns for wrapping similar legacy systems that can be used depending on the circumstances.

4. The Design Patterns

The following design patterns have been defined in the 'Gang of Four' format. We have been deliberately concise with some of the fields to accommodate the format of an academic paper.

[1] TOIA Homepage: http://www.toia.ac.uk/ (July, 2006)

4.1 Lowest Common Denominator Interface

Pattern Name and Classification: Lowest Common Denominator (LCD) Interface (*Behavioral*)

Intent: To provide the simplest way to create a common interface for two or more software components that are non-identical but which share some common methods.

Also Known As: LCD Interface

Motivation: When integrating existing software components into a Service-Oriented Architecture (SOA) it is necessary to create a Service Interface that captures the functionality of that software and makes it available as a Service. Similar software components should be wrapped with a common interface to enable them to be used modularly within the SOA. The LCD interface is a simple approach to rapidly defining a common interface, with a direct relation between the methods of the common interface and the functionality of the underlying legacy component.

Implementation: A LCD interface is a strict intersection of the functionality of all the legacy components considered. This can be derived by creating interfaces for individual legacy components, normalizing the methods, and extracting those that are common. The data models used in the LCD interface may be different from those wrapped in the legacy systems, although typically the most common approach will be re-used.

Structure: Figure 2 shows a Venn diagram comparing the hypothetical interfaces of two legacy systems with the LCD interface.

Applicability: It is feasible to use a LCD interface when the intersection of functionality between legacy systems includes the functionality that, in the view of an expert, captures the core essence of all the legacy systems considered.

Participants: The pattern applies to at least two software components which have service-like behavior that is similar. It can be generalised to include more components.

Collaboration: The LCD interface can be implemented as an Adaptor-style service. Calls to the LCD interface can be passed directly on to the legacy systems that have been wrapped, although data types may have to be converted and coarse grained methods may have to be devolved into several fine grained calls.

Consequences: The LCD interface is simple to derive, but its effectiveness at capturing the functionality of wrapped legacy systems depends on a high

similarity between the functionality of those systems. It may stifle richness by ignoring novel functionality that is not shared by all. In addition, the likelihood of the LCD interface being effective (capture core functionality) is reduced in proportion to the number of legacy systems being wrapped.

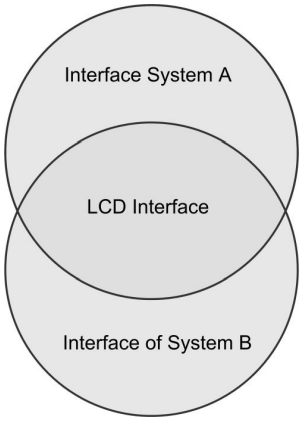


Figure 2: The LCD interface – the intersection of the methods of legacy systems A and B.

Known Uses: This pattern was used within the JISC FREMA project to wrap two item banks (TOIA and E3AN). The code is available from the FREMA website (www.frema.ecs.soton.ac.uk).

Related Patterns: *Adaptor Pattern* [8]: described how classes can be wrapped in Object Orientation, it is a structural pattern focusing on methods of implementation, rather than what parts of the wrapped class should be exposed. *Most Popular Interface* and *Negotiated Interface* are alternative patterns that deal with defining common interfaces for similar software systems.

4.2 Most Popular Interface

Pattern Name and Classification: Most Popular Interface (*Behavioral*)

Intent: To provide a rounded and robust common interface for two or more software components that are non-identical but which share some common methods.

Also Known As: N/A

Motivation: When integrating existing software components into a Service-Oriented Architecture (SOA) it is necessary to create a Service Interface that captures the functionality of that software and makes it available as a Service. Similar software components should be wrapped with a common interface to enable them to be used modularly within the SOA. The Most Popular interface is an approach that produces a compromise interface that reflects the best practice of many legacy systems.

Implementation: A Most Popular interface is an interface whose methods form a set M, such that the intersection of the methods of two or more legacy systems is a proper subset of M. The methods included in M are chosen by a group of experts, to reflect the functionality that they believe would be expected by the community.

Structure: Figure 3 shows a Venn diagram comparing the hypothetical interfaces of two legacy systems with the Most Popular interface.

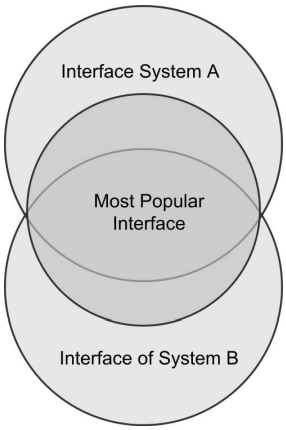


Figure 3: The Most Popular Interface - the methods deemed essential by a group of experts – a subset C, such that the intersection of A and B is a proper subset of C.

Applicability: It is feasible to use a Most Popular interface when there is agreement between experts in a community about the core functionality that should be expected from that type of system.

Participants: The pattern applies to at least two software components which have service-like behavior that is similar. It can be generalised to include more components.

Collaboration: The Most Popular interface can be implemented as an Adaptor-style service. In some cases there will be a mismatch between the functionality represented in the interface and that supported by the wrapped legacy system. There are two possible approaches, to either make the mismatched methods empty calls, that return null, or to replicate the missing functionality with new code (that may utilize the functionality of the

wrapped legacy system in a new way).**Consequences**: The Most Popular interface is complex to derive, and may require a prolonged standardization effort, but it is highly effective at capturing a broad set of capabilities from legacy software and creating a robust and reusable common service. If experts differ then it is possible that many competing common interfaces evolve. It is also possible that in some cases no common view exists. When implementing missing functionality there are two approaches that may be taken:

- The wrapping service might use additional information that was not part of the wrapped legacy system. In this case the new information must be created in order for the wrapping service to work. For example, some item bank services have a "Search by Keyword" method, for those item banks without this method keywords for each item must be created and stored, so the method can be simulated.

- The wrapping service uses existing information within the legacy system in a new way in order to simulate the method. In the "Search by Keyword" example a Term Frequency analysis could be used on the main text of the items, held in the legacy system, to calculate keywords at runtime.

The latter approach is more robust, and can deal with changing data within the legacy system, but may not be appropriate if Quality of Service is an issue.

Known Uses: This pattern was used within the JISC FREMA project to wrap two item banks (TOIA and E3AN). The code is available from the FREMA website (www.frema.ecs.soton.ac.uk).

Related Patterns: *Adaptor* Pattern described how classes can be wrapped in Object Orientation, it is a structural pattern focusing on methods of implementation, rather than what parts of the wrapped class should be exposed. *Lowest Common Denominator Interface* and *Negotiated Interface* are alternative patterns that deal with defining common interfaces for similar software systems.

4.3 Negotiated Interface

Pattern Name and Classification: Negotiated Interface (*Behavioral*)

Intent: To provide a flexible common interface that preserves richness, for two or more software components that are non-identical but which share some common methods.

Also Known As: N/A

Motivation: When integrating existing software components into a Service-Oriented Architecture (SOA) it is necessary to create a Service Interface that captures the functionality of that software and makes it available as a Service. Similar software components should be wrapped with a common interface to enable them to be used modularly within the SOA. The Negotiated interface is an approach that produces a flexible interface which enables all the functionality from all the similar legacy systems to be represented, even though that functionality may be impossible to replicate on some other legacy systems.

Implementation: A Negotiated interface is an interface whose methods represent the union of all methods from two or more legacy systems that have been identified by experts as being important within a domain. The interface also includes methods that allow users of the service to query which methods are supported by the currently wrapped legacy system. This may be done by returning a contract that describes which methods are currently available, or by querying at runtime for the availability of individual methods.

Structure: Figure 4 shows a Venn diagram comparing the hypothetical interfaces of two legacy systems with the Negotiated interface:

Applicability: It is advisable to use a Negotiated interface when there is novel functionality in some legacy systems that experts believe should be reflected in a common interface even though it is not universally supported. However, a Negotiated interface adds runtime complexity, and makes systems less robust, as they may fail if functionality that is required is missing from the wrapped legacy system.

Participants: The pattern applies to at least two software components which have service-like behavior that is similar. It can be generalised to include more components.

Collaboration: The Negotiated interface pattern can be implemented as an Adaptor-style service.

Consequences: The Negotiated interface is cumbersome to define, but avoids complex expert decisions about definitive interfaces. It adds runtime complexity to a service framework, and because of its dynamic nature can destabilize a service-based system (although this can be mitigated by contract-style negotiation that allows for earlier error checking).

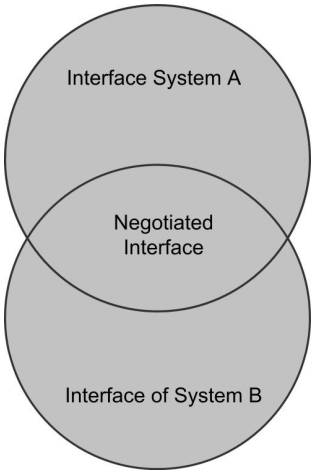


Figure 4: The Negotiated Interface - all the interface methods supported by all major systems – the union of A and B – but with a negotiation interface that allows individual systems to declare whether they support methods at runtime.

Known Uses: This pattern was used within the JISC FREMA project to wrap two item banks (TOIA and E3AN). The code is available from the FREMA website (www.frema.ecs.soton.ac.uk).

Related Patterns: *Adaptor* Pattern described how classes can be wrapped in Object Orientation, it is a structural pattern focusing on methods of implementation, rather than what parts of the wrapped class should be exposed. *Lowest Common Denominator Interface* and *Most Popular Interface* are alternative patterns that deal with defining common interfaces for similar software systems.

5. Experience and Reflections

Within FREMA we wanted to show how web services could be used to wrap legacy systems. In e-Assessment Item Banking is one of the best supported activities. Item banks are databases of questions that can be queried to provide content for either summative or formative assessment. Item Banks are a good example of legacy systems as they often have slightly different query functionality and use different data formats for their questions (although the QTI format is becoming a popular standard). We attempted to wrap two systems, trying each of our three patterns:

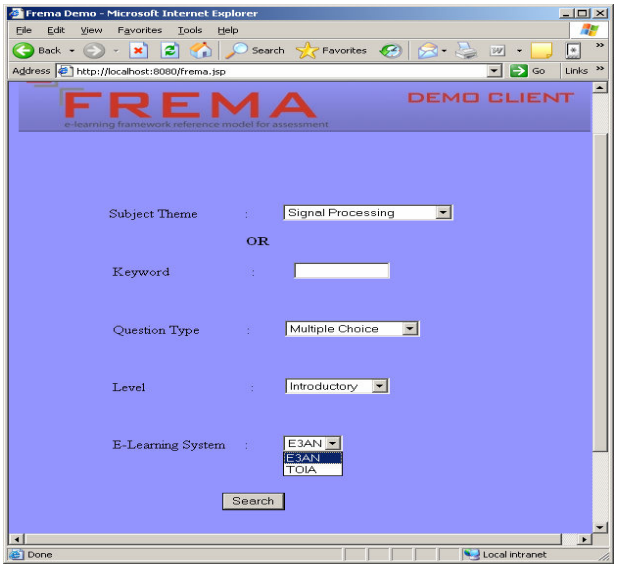
TOIA (Technologies for Online Interoperable Assessment) is a free question management system developed for use by UK HE institutions. TOIA supports the basic idea of grouping items together by subject theme. However TOIA takes this concept of grouping even further by grouping a number of subject themes as a hierarchical *content structure*. For example, there could be a content structure called "Computer Science First Year" which could have a number of subject themes like "Programming", "Computer Basics", "Digital Circuits", etc. In addition to Content Structure & Subject theme, TOIA also supports search by keyword. However keywords are associated with subject themes not Assessment Items. When you search for a keyword in TOIA, you get all the Items that are associated with one or more Subject themes with which that keyword is associated.

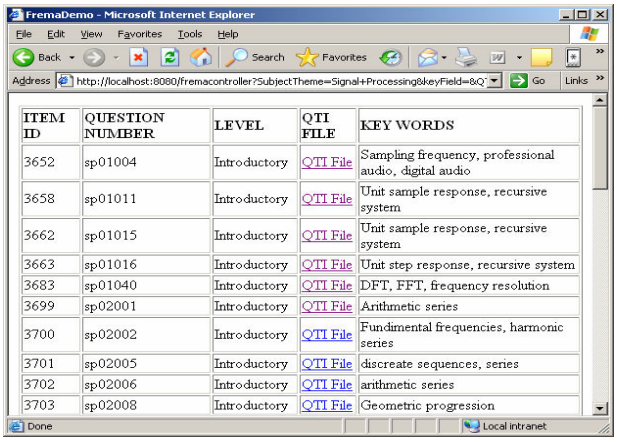
E3AN (Electronics and Electrical Engineering Assessment Network) was an initiative to collect questions around the topic of electrical and electronic engineers, it uses a large open database as its repository. E3AN supports the concept of grouping items by subject theme. However there is no concept of content structure. E3AN also supports the concept of keywords, and associates keyword with individual Assessment Items. Search by Keyword operation in E3AN returns all the relevant Items that are associated with a specific keyword, unlike TOIA where keywords are associated with subject themes.

Originally we also hoped to use SPAID as one of our systems, but the documentation and code where difficult to obtain and in the end we had to concentrate on the two systems that we had available. Although the systems use different formats for their questions, both formats are similar to QTI, and are expressed in XML.

Figures 5 shows the front end to the web service wrappers that we wrote. There is a simple web interface for each pattern that allows queries to be made through the pattern interface. A drop down option allows the user to choose which of the two legacy systems they wish to query. The results are returned with metadata mapped onto the QTI standard. If the user goes through this to the question below they retrieve the original question XML.

Implementing the Lowest Common Denominator (LCD) interface was relatively easy; both systems implement a simple keyword search scoped by question level and type. However, since E3AN does not categorize its questions we do not support searching by subject theme. Implementing the Most Popular interface meant that we were able to extend the interface to include searches by subject theme. Since E3AN does not include a content structure we had to produce this new information in some way. We briefly considered attempting to derive the category from the existing keywords, but since the E3AN database is static we decided that it would be easier to manually classify the questions and store the info in a separate database accessed by the wrapper service.





ITEM ID	QUESTION NUMBER	LEVEL	QTI FILE	KEY WORDS
3652	sp01004	Introductory	QTI File	Sampling frequency, professional audio, digital audio
3658	sp01011	Introductory	QTI File	Unit sample response, recursive system
3662	sp01015	Introductory	QTI File	Unit sample response, recursive system
3663	sp01016	Introductory	QTI File	Unit step response, recursive system
3683	sp01040	Introductory	QTI File	DFT, FFT, frequency resolution
3699	sp02001	Introductory	QTI File	Arithmetic series
3700	sp02002	Introductory	QTI File	Fundimental frequencies, harmonic series
3701	sp02005	Introductory	QTI File	discreate sequences, series
3702	sp02006	Introductory	QTI File	arithmetic series
3703	sp02008	Introductory	QTI File	Geometric progression

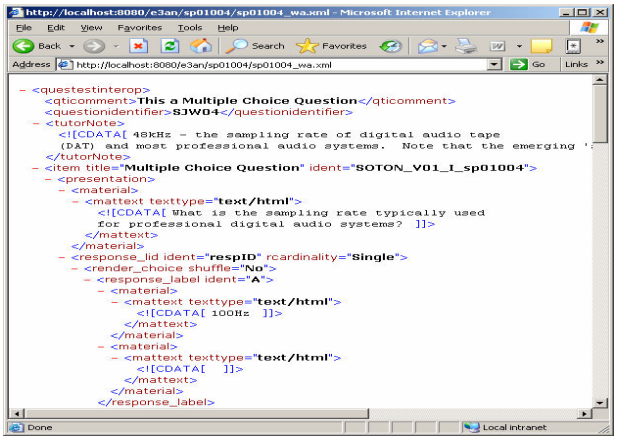


Figure 5: The Web Interface to our wrapper services, the results returned, and Item XML.

Implementing the Negotiated interface involved adding validation to the interface. For reasons of simplicity we did not implement a contract system, but instead added a new validator method that took another method name as a parameter and returned whether it was supported or not. In our interface this was used as the page was loaded in order to determine whether to disable certain search options. This introduced some runtime overhead, but in our system the validation calls were infrequent, and so the added complexity did not adversely effect the application.

Our work demonstrates that all three patterns are viable, and we made the following observations:

- Writing/wrapping a service interface around a legacy system is non-trivial, even for functionally simple systems such as Item Banks, because of the variety of technologies involved. For example, E3AN is based on MS-Access and TOIA uses a proprietary .Net application. There are a number of commercial tools from main stream vendors to service enable legacy systems[2], but these tools are very costly initially to buy and also complex to configure.
- Writing a wrapper around existing systems involves a close understanding of the data model for each of these applications. This can be challenging if the data model is not well documented (for example, reverse engineering from a normalized database, TOIA uses 10+ tables to model its questions). Therefore the complexity of wrapping rises in proportion to the complexity of the data model as well as the interface.
- Even when there is standardization internal representations can vary. Although the QTI specification standardizes the representation of assessment items, these items are stored in application (E3AN, TOIA) databases in diverse ways for performance and scalability reasons. For example, in TOIA an assessment item is called "Question" whereas in E3AN it's called "Item". Similarly, "Subject Theme" is called "Topic" in TOIA. Mapping the terminology used in different systems can be time consuming and in some cases non-obvious.
- Interpretations of standards can sometimes vary. For example, the QTI specification

[2] For example: The Web Sphere Integration platform, and the Oracle Integration Suite

provides a standard for the recording of metadata, but different implementations interpret this in different ways: in some item bank systems, keywords are associated with assessment items and in others they are associated with subject themes

- Support for web service standards is intermittent as tools for implementing Web services are fairly new and not very stable. For example, few IDEs or service containers support WS-I Basic profile, and WS-I secure profile is still being finalized. This situation should improve, but at the moment remains a barrier to creating interoperable legacy wrappers.

6. Conclusions and Future Work

In this paper we have described how wrapping legacy systems is a common problem when introducing service-oriented architectures to a particular domain or community. This is made more difficult as there are often many systems that offer similar functionality (we term these similar legacy systems) and it is desirable to give all these systems a common interface to aid in interoperability and modularity.

Through our work with the JISC e-Framework we have formalized three design patterns for coping with this problem. All three patterns are based on creating specialized services for each legacy system and then normalizing them in terms of data model and terminology.

The Lowest Common Denominator (LCD) pattern, selects only the methods common to all the legacy systems considered. It is simple to create, and rapid to build, as all the functionality required already exists within the legacy systems. However, it can be overly simple, and may miss out valuable functionality that is not common to all legacy systems

The Most Popular pattern selects a set of core methods based on the view of experts. These methods may not be supported by all the wrapped systems and so it may be necessary to write additional functionality into the wrapping service. This pattern depends on a common expert view, which can be difficult to reach, and may still reflect a compromise by the community, but is likely to fulfill the requirements of the majority. It can be expensive to implement, as some wrappers will need to add the additional functionality themselves.

The Negotiated Interface represents all the methods from all legacy systems, but within a negotiation framework such that services can inquire of one another which of their advertised services are available (based on which legacy system is being wrapped). The negotiation may happen on a per method basis, or could be implemented via a system of contracts. This is the most flexible of the patterns, but it adds a run-time overhead, and makes failure checking complex (as the system can fail at run-time).

Combinations of these patterns are possible, for example by implementing a Most Popular interface with Negotiated-style caveats on the non-common methods. Or by using the LCD interface as a contract point within the Negotiated interface to assure a minimum level of co-operation.

Service-Oriented architectures offer an opportunity for communities to create common frameworks of pluggable software components, and thus to interoperate to a new level. However, to bootstrap these efforts it is necessary to include the rich collections of existing legacy software in these new frameworks. It is our belief that the design patterns we have presented here will enable developers to achieve this more easily. While none of the approaches described in the patterns are individually novel, we hope that by expressing them in a formalized way, and in a common context, we may help future service developers to choose an appropriate approach, and to articulate their decisions more effectively.

7. References

[1] Baresi, L., Heckel, R., Thöne, S., and Varró, D. (2003). Modeling and validation of service-oriented architectures: application vs. style. In Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Helsinki, Finland, September 01 - 05, 2003). ESEC/FSE-11

[2] Beck, K., Crocker, R., Meszaros, G., Vlissides, J., Coplien, J. O., Dominick, L., and Paulisch, F. 1996. Industrial experience with design patterns. In Proceedings of the 18th international Conference on Software Engineering (Berlin, Germany, March 25 - 29, 1996). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 103-114..

[3] Cline, M. P. 1996. The pros and cons of adopting and applying design patterns in the real world. Commun. ACM 39, 10 (Oct. 1996), 47-49.

[4] Davis, H. C., White, S. A. and Dickens, K. P. (2002) Engineering a Testbank of Engineering Questions. In Proceedings of ICEE 2002: The International Conference on Engineering Education, Manchester.

[5] Dijkman, R. and Dumas, M (2004). Service-oriented Design: A Multi-viewpoint Approach. International Journal of Cooperative Information Systems 13(4), December 2004.

[6] Fielding, R. T. and Taylor, R. N. 2002. Principled design of the modern Web architecture. ACM Trans. Inter. Tech. 2, 2 (May. 2002), 115-150.

[7] Foster, I., Kesselman, C., and Tuecke, S. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Int. J. High Perform. Comput. Appl. 15, 3 (Aug. 2001), 200-222.

[8] Gamma, E., Helm, R., Johnson R, and Vlissides J. 1995 Design patterns: elements of reusable object-orient software. Addison-Wesley Professional 1995

[9] Gomaa, H., Menascé, D. A., and Shin, M. E. 2001. Reusable component interconnection patterns for distributed software architectures. In Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (Toronto, Ontario, Canada). SSR '01. ACM Press, New York, NY, 69-77.

[10] Martin J, Arsanjani A, Tarr P, and Hailpern B, (2003), "Web Services: Promises and Compromises," Queue vol. 1, pp. 48-58, 2003.

[11] Olivier, B., Roberts, T. and Blinco, K. (2005). The e-Framework for Education and Research: An Overview. Version R1, July 2005. DEST, JISC-CETIS.

[12] Quartel D.A.C., Dijkman R.M., and van Sinderen M.J.. (2004) Methodological Support for Service-oriented Design with ISDL. In: Proceedings of the 2nd ACM International Conference on Service Oriented Computing (ICSOC), New York City, NY, USA, pp. 1-10, 2004

[13] Rossi, G., Schwabe, D., and Garrido, A. 1997. Design reuse in hypermedia applications development. In *Proceedings of the Eighth ACM Conference on Hypertext* (Southampton, United Kingdom, April 06 - 11, 1997). ACM Press, New York, NY, 57-66.

[14] Schmidt, D. C. and Buschmann, F. 2003. Patterns, frameworks, and middleware: their synergistic relationships. In Proceedings of the 25th international Conference on Software Engineering (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 694-704.

[15] Schmidt, D. C., Fayad, M., and Johnson, R. E. 1996. Software patterns. Commun. ACM 39, 10 (Oct. 1996), 37-39.

[16] Wada, H., Suzuki, J., and Oba, K. (2005). Modeling turnpike: a model-driven framework for domain-specific software development. In Companion To the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05. ACM Press, New York, NY, 128-129.

[17] Wilson, S., Blinco, K. and Rehak, D. (2004). Service-Oriented Frameworks: Modelling the infrastructure for the next generation of e-Learning Systems. A Paper prepared on behalf of DEST (Australia), JISC-CETIS (UK), and Industry Canada.

[18] Young R., MacNeill S., Adams D., McAlpie M. 2005. SPAID (Storage and Packaging of Assessment Item Data) Final Report Published by JISC, 28 Oct 2005.