Design Patterns in Object-Oriented Frameworks

Object-oriented frameworks provide an important enabling technology for reusing software components. In the context of speech-recognition applications, the author describes the benefits of an object-oriented framework rich with design patterns that provide a natural way to model complex concepts and capture system relationships.

Savitha Srinivasan
IBM

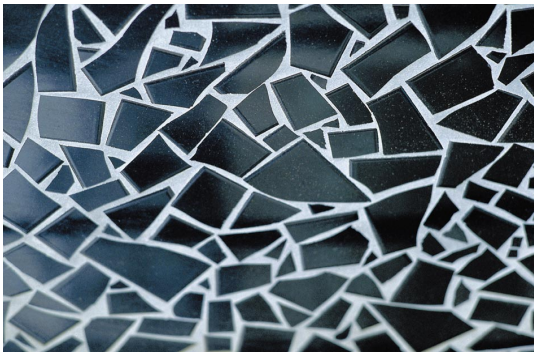
Developing interactive software systems with complex user interfaces has become increasingly common, with prototypes often used for demonstrating innovations. Given this trend, it is important that new technology be based on flexible architectures that do not require developers to understand all the complexities inherent in a system.

Object-oriented frameworks provide an important enabling technology for reusing both the architecture and the functionality of software components. But frameworks typically have a steep learning curve since the user must understand the abstract design of the underlying framework as well as the object collaboration rules or contracts—which are often not apparent in the framework interface—prior to using the framework.

In this article, I describe our experience with developing an object-oriented framework for speech recognition applications that use IBM's ViaVoice speech recognition technology. I also describe the benefits of an object-oriented paradigm rich with design patterns that provide a natural way to model complex concepts and capture system relationships.

OBJECT-ORIENTED FRAMEWORKS

Frameworks are particularly important for developing open systems, where both functionality and architecture must be reused across a family of related applications. An object-oriented framework is a set of collaborating object classes that embody an abstract design to provide solutions for a family of related problems. The framework typically consists of a mixture of abstract and concrete classes. The abstract classes usually reside in the framework, while the concrete classes reside in the application. A framework, then, is a semicomplete application that contains certain fixed aspects common to all applications in the problem domain, along with certain variable aspects unique to each application generated from it.



The variable aspects, called *hot spots*, define those aspects of an application that must be kept flexible for different adaptations of the framework.[1] What differentiates one framework application from another in a common problem domain is the manner in which these hot spots are defined.

Despite the problem domain expertise and reuse offered by framework-based development, application design based on frameworks continues to be a difficult endeavor. The framework user must understand the complex class hierarchies and object collaborations embodied in the framework to use the framework effectively. Moreover, frameworks are particularly hard to

0018-9162/99/$10.00 © 1999 IEEE

document since they represent a reusable design at a high level of abstraction implemented by the framework classes.

But *design patterns*—recurring solutions to known problems—can be very helpful in alleviating software complexity in the domain analysis, design, and maintenance phases of development. Framework designs can be discussed in terms of design pattern concepts, such as participants, applicability, consequences, and trade-offs, before examining specific classes, objects, and methods.

Documenting a framework—on paper or in the code itself—as a set of design patterns is an effective means of achieving a high level of communication between the framework designer and the framework user.[2] Using design patterns helped us establish a common terminology with which we could discuss the design and use of the framework.

DESIGN PATTERNS

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.[2] By their very definition, design patterns result in reusable object-oriented design because they name, abstract, and identify key aspects of a common design structure.

Design patterns fall into two groups. The first group focuses on object-oriented design and programming or object-oriented modeling, while the second group—a more recent trend—focuses on patterns that address problems in efficient, reliable, scalable, concurrent, parallel, and distributed programming.[3] In this article, I focus primarily on the first group, although my colleagues and I used patterns from both categories to address our design problems.

In designing speech-recognition applications, we used patterns to guide the creation of abstractions necessary to accommodate future changes and yet maintain architectural integrity. These abstractions help decouple the major components of the system so that each may vary independently, thereby making the framework more resilient.

In the implementation stages, the patterns helped us achieve reuse by favoring object composition or delegation over class inheritance, decoupling the user interface from the computational component of an application, and programming to an interface as opposed to an implementation.

In the maintenance phases, patterns helped us document strategic properties of the software at a higher level than the source code.

Contracts

Design patterns describe frameworks at a very high level of abstraction. *Contracts*, on the other hand, can be introduced as explicit notation to specify the rules that *govern* how objects can be combined to achieve certain behaviors.[4]

However, frameworks don't enforce contracts; if an application does not obey the contracts, it does not comply with the intended framework design and will very likely behave incorrectly. To make these relationships more clear, a number of researchers have introduced the concept of *motifs* to document the purpose and use of the framework in light of the role of design patterns and contracts.[5,6]

Documenting a framework as a set of design patterns is an effective means of achieving a high level of communication between the framework designer and the framework user.

A FRAMEWORK FOR SPEECH RECOGNITION

We designed a framework for speech recognition applications intended to enable rapid integration of speech recognition technology in applications using IBM's ViaVoice speech recognition technology (http://www.software.ibm.com/speech/). Our primary objective was to simplify the development of speech applications by hiding complexities associated with speech recognition technology and exposing the necessary variability in the problem domain so the framework user may customize the framework as needed.

We focused on the abstractions necessary to identify reusable components and on the variations necessary to provide the customizations required by a user to create a specific application. While this method adheres to the classical definition of what a framework must provide, we consciously incorporated feedback into the framework evolution process.

Speech concepts and complexities

At a simplistic level, the difference between two speech recognition applications boils down to what specific words or phrases you say to the application and how the application interprets what you say. What an application understands is of course determined by what it is listening for—or its *active vocabulary*. Constraining the size of the active vocabulary leads to higher recognition accuracy, so applications typically change their active vocabulary with a change in context.

The result that the speech recognition engine returns in response to a user's utterance is a *recognized word*. In the context of a GUI application, the active vocabulary and the recognized words may be different for each window and may vary within a window, depending on the state of the application.

Several factors contribute to complexity in developing speech recognition applications. Recognition technology is inherently asynchronous and requires adhering to a well-defined handshaking protocol. Furthermore, the asynchronous nature of speech recognition makes it possible for the user to initiate an action during an application-level task so that the

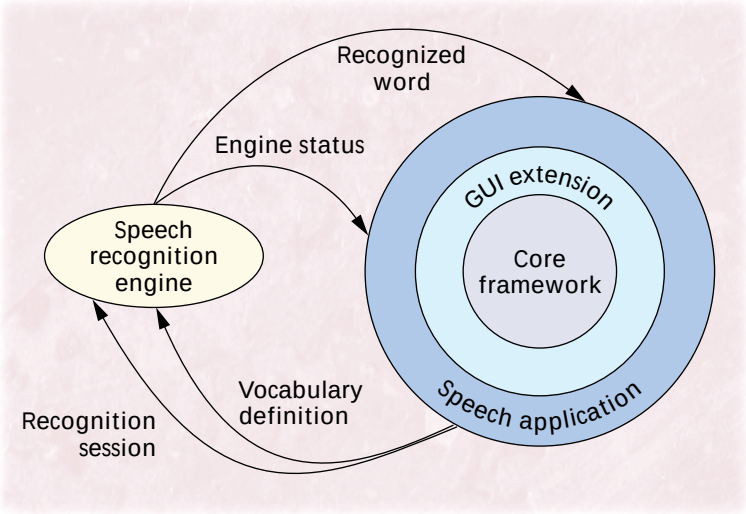


Figure 1. The speech recognition engine and speech application are separate processes. The first layer, the core framework, encapsulates the speech engine functionality in abstract terms, independent of any specific GUI class library. The second layer extends the core framework for different GUI environments in order to provide tightly integrated, seamless speech recognition functionality.

application must either disallow or defer any action until all previous speech engine processing can be completed.

Also, speech-programming interfaces typically require a series of calls to accomplish a single application-level task. And achieving high accuracy requires that the application constantly monitor the engine's active vocabulary. The uncertainty associated with high-accuracy recognition forces the application to deal with recognition errors while the recognition engine communicates with applications. It must do so at a process level and carries no understanding of GUI windows or window-specific vocabularies. The application must therefore build its own infrastructure to direct and dispatch messages at the window level.

Framework abstractions and hot spots

Common application functions include establishing a recognition session, defining and enabling a pool of vocabularies, ensuring a legal engine state for each call, and directing recognized word messages and engine status messages to different windows in the application. The application needs explicit hot-spot identification to handle variations such as the active window, the active vocabulary in a specific active window, the recognized words received by the active window based on the active vocabulary, and the action that an active window must take based on the recognized word.

Figure 1 shows a high-level view of the layered framework architecture that we adopted in order to endow this theory with maximum flexibility and reuse. The speech recognition engine and speech application are separate processes. The first layer, the core framework, encapsulates the speech engine functionality in abstract terms, independent of any specific GUI class library. A GUI speech recognition application usually involves the use of a GUI class library; therefore, the second layer extends the core framework for different GUI environments to provide tightly integrated, seamless speech recognition functionality.

Each GUI speech application requires customizing the GUI extension for the core framework. We modeled the common aspects of the problem domain by abstract classes in the core framework; the application implements concrete classes derived from the abstract classes. The class diagrams in the following sections describe the extensions to the core framework for IBM's VisualAge GUI class library.

We use the prefix "I" as a naming convention for the GUI extension classes. For example, the class ISpeechClient refers to the GUI framework extension class that provides the core SpeechClient class functionality. We use the Unified Modeling Language (UML)[7] notation generated by Rational Rose to represent the class diagrams.

Framework collaborations

The class diagram shown in Figure 2 shows the eventual interface classes for the VisualAge extension to the core framework classes. (The framework evolved over time as we developed various applications.) The IVocabularyManager, ISpeechSession, ISpeechClient, ISpeechText, and ISpeechObserver classes provide the necessary abstractions to direct, dispatch, and receive speech recognition engine events at a level that is meaningful to the application. The ISpeechSession class must establish a recognition session through ISpeechManager prior to any speech functions starting.

ISpeechClient, ISpeechText, and ISpeechObserver provide the necessary abstractions for implementing a GUI application, but do not contain a GUI component. An application creates a speech-enabled GUI window using multiple inheritance from the ISpeechClient class and a VisualAge window class such as IFrameWindow. This inheritance provides speech functionality and speech engine addressability at a GUI window level so that recognized words can be directed to a particular GUI window.

Similarly, an application creates a speech-enabled text widget using multiple inheritance from the ISpeechText class and a VisualAge text-widget class such as IEditControl. This process provides speech functionality and speech engine addressability at a text-widget level. Derived ISpeechObserver classes provide a publish-subscribe protocol necessary to maintain a consistent speech recognition engine state across all windows in the application. The IVocabularyManager class supports the dynamic definition and manipulation of vocabularies. The collaborations of these classes is internally supported by the ISpeechManager class, which behaves in the manner of a Singleton pattern[2] (discussed later) that makes the class itself responsible for keeping track of its sole instance.

Figure 3 shows one of the primary hot spots in the system, the processing of words recognized by the active ISpeechClient. DictationSpeechClient is a concrete class derived from the abstract ISpeechClient

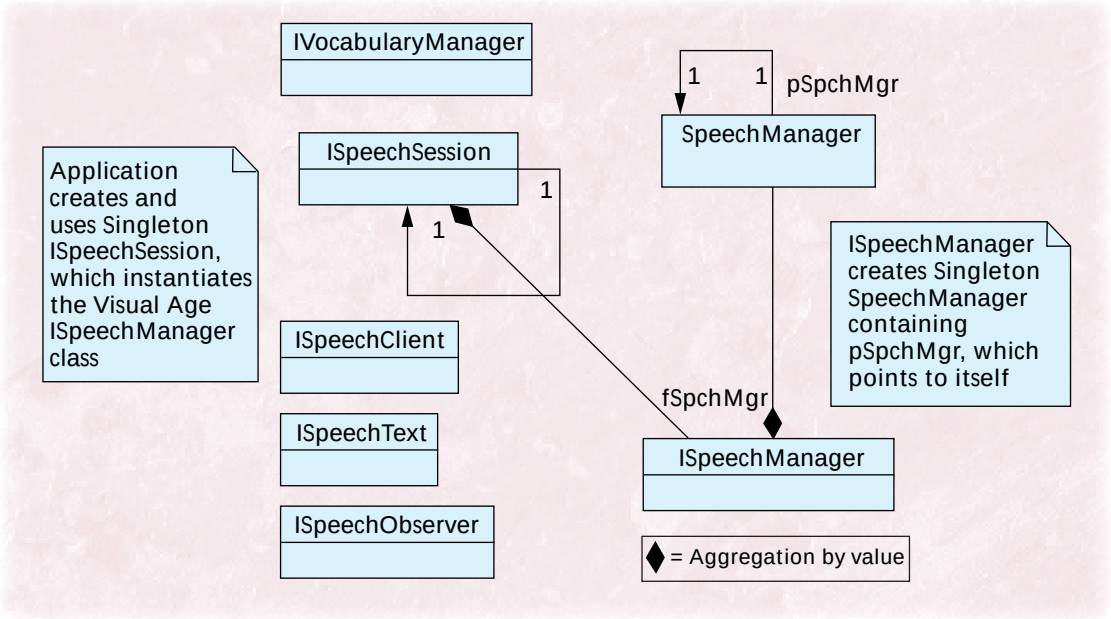


Figure 2. Interface classes for the VisualAge extension to the core framework classes. IVocabularyManager, ISpeechSession, ISpeechClient, ISpeechText, and ISpeechObserver classes provide the necessary abstractions to direct, dispatch, and receive speech recognition engine events at a level that is meaningful to the application. The prefix "I" signals a GUI extension class.

class. The speech engine invokes the pure virtual method, `recognizedWord()`, when it recognizes a word; this gives DictationSpeechClient the ability to process the recognized word in an application-specific manner. This process works as designed only if the application obeys the accompanying contract, which states that before a user speaks into the microphone (which invokes the `recognizedWord()` method), the application must specify the active ISpeechClient and its corresponding vocabularies.

A GUI application might contain several windows, each of which is a multiple-inheritance-derived instance of ISpeechClient and IFrameWindow. In a case like this, the framework user must at all times keep track of the currently active ISpeechClient and ensure that the correct one is designated as being active. Likewise, enabling the appropriate vocabulary based on the active ISpeechClient is the framework user's responsibility.

One way to do this is to designate the foreground window as the active ISpeechClient. The application must track changes in the foreground window and designate the appropriate derived ISpeechClient as active. This means that the application must process the notification messages sent by the IFrameWindow class when the foreground window changes and must also update the active ISpeechClient and vocabulary. Only then will the implementation of the hot spot in Figure 3 ensure delivery of the recognized words to the correct window by calling the active ISpeechClient's `recognizedWord()` method.

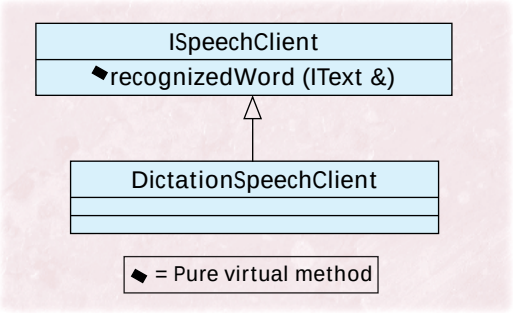


Figure 3. A hot spot that processes words recognized by the active ISpeechClient. DictationSpeechClient is a concrete class derived from the abstract ISpeechClient class. The speech engine invokes the pure virtual method, `recognizedWord()`, when it recognizes a word; this gives DictationSpeechClient the ability to process the recognized word in an application-specific manner.

FRAMEWORK EVOLUTION THROUGH REUSE

We used a single framework to build four applications: MedSpeak/Radiology, MedSpeak/Pathology, the VRAD (visual rapid application development) environment, and a socket-based application providing speech recognition functionality to a video-cataloging tool. Each application we developed had a particular effect on the development and evolution of the framework itself. We used some well-known design patterns to facilitate reuse of design and code and to decouple the major components of the system. Table 1 summarizes some of the design patterns we used and the reason we used them.

Initially, we used the Facade pattern to provide a unified, abstract interface to the set of interfaces supported by the speech subsystem. The core abstract speech classes shown in Figure 2 communicate with the speech subsystem by sending requests to the Facade object, which forwards the requests to the appropriate subsystem objects. Facade implements the Singleton pattern, which guarantees a sole instance of the speech subsystem for each client process.

As the framework evolved, we found ourselves incorporating new design patterns to extend the framework's functionality and manageability.

Radiology dictation application

We designed the MedSpeak/Radiology dictation application—the first framework we created—to build reports for radiologists using real-time speech recognition.[8] We ourselves were the developers and users of the framework, which vastly contributed to our understanding of what the framework requirements were and helped us find a balance between encapsulating problem domain complexities and exposing the

Table 1. Design patterns used in building the speech-recognition framework.

Design pattern used	Reason for using the pattern
Object-oriented patterns	
Adapter[2] (or Wrapper)	Enable the use of existing GUI classes whose interface did not match the speech class interface Create a reusable class that cooperates with unrelated GUI classes that don't necessarily have compatible interfaces
Facade[2]	Provide an abstract interface to a set of interfaces supported by the speech subsystem Abstract speech concepts to facilitate the use of a different speech recognition technology
Observer[2]	Notify and update all speech-enabled GUI windows in an application when speech engine changes occur
Singleton[2]	Create exactly one instance of the speech subsystem interface class per application since the recognition system operates at the process level
Distributed or concurrent patterns	
Active Object[9]	Decouple method invocation from method execution in the application when a particular word is recognized
Asynchronous Completion Token[12]	Allow applications to efficiently associate state with the completion of asynchronous operations
Service Configurator[10]	Decouple the implementation of services from the time when they are configured

variability needed in an application. Specifically, it indicated to us that we needed a clear separation between the GUI components and the speech components, so we defined the concept of a SpeechClient to abstract window-level speech behavior.

We used the class Adapter (or Wrapper) pattern to enable the use of existing GUI classes whose interfaces did not match the application's speech class interfaces.[2] For example, we used Adapter to create a speech-enabled GUI window that multiple inherits from both an abstract SpeechClient class and a specific GUI window class—in this first application, the XVT GUI class library. Thus, we were able to encapsulate speech-aware behavior in abstract framework classes so that GUI classes did not have to be modified to exhibit speech-aware behavior.

Finally, we used the Active Object[9] and Service Configurator[10] patterns to further decouple the GUI implementation from the speech framework. Using the principle embodied in the Active Object pattern, we separated method definition from method execution by the use of a speech profile for each application, which in essence is the equivalent of a lookup table. This delayed the binding of a speech command to an action from compile time to runtime and gave us tremendous flexibility during development.

Similarly, the Service Configurator pattern decouples the implementation and configuration of services, thereby increasing an application's flexibility and extensibility by allowing its constituent services to be configured at any point in time. We implemented this pattern again using the speech profile concept, which set up the configuration information for each application (initial active vocabularies, the ASCII string to be displayed when the null word or empty string is returned, and so forth).

Pathology dictation application

MedSpeak/Pathology was similar to the radiology dictation application, but customized for pathology workflow. We worked closely with the framework users in extending the core framework for a GUI environment different from that of the previous application—namely, the VisualAge GUI class library discussed earlier in the "Framework collaborations" section. The ease with which we accomplished this porting effort validated our Adapter approach to providing speech functionality in GUI-independent classes.

However, it brought out an additional requirement stemming from the specific GUI builder that was being used—that the class form of the Adapter pattern, which uses multiple inheritance, was unacceptable. We therefore modified our usage of the Adapter pattern to the object form (using composition) to accomplish the same result. Instead of defining a class that inherits both Medspeak/Pathology's SpeechClient interface and VisualAge's IFrameWindow implementation, for example, we added an ISpeechClient instance to IFrameWindow so that requests for speech functionality in IFrameWindow are converted to equivalent requests in ISpeechClient.

Furthermore, the need to provide an abstract SpeechText class to provide speech-aware edit control functionality became apparent when we realized the complexity associated with maintaining the list of dictated words. Likewise, the problems associated with maintaining the speech engine state across different windows led to the definition of the SpeechObserver class using the Observer pattern. The Observer pattern defines an updating interface for objects that need to be notified of changes in a subject.[2]

The VRAD environment

The visual rapid application development (VRAD) class of applications that used our framework provided us with insight into our original objectives, which included hiding speech complexities and providing variability. We planned that the speech framework classes would be included in an all-encompassing

object-oriented programming framework, the VisualAge class library. The previous pathology dictation application was one instance of an application that used the VisualAge class library together with the speech framework.

The greatest impediment to the acceptance of the framework by the VRAD users was that the public interface by itself didn't adequately provide the understanding necessary to create speech applications rapidly. For example, the variability provided by the hot spot in Figure 3 requires that the application enforce the designation of the correct ISpeechClient as well as the active vocabulary. Users had to turn to the framework documentation in order to understand the accompanying contracts. Furthermore, our use of the class form of the Adapter pattern, which uses multiple inheritance, forced the users to understand the behavior of the parent classes. This was additional motivation for us to favor the object form of the Adapter pattern.

We therefore modified the public interface in an effort to reflect the client's view, while still ensuring that the necessary contracts were enforced. To accomplish this, we created additional hot spots that helped framework users comply with the internal constraints not ordinarily visible in the external interface but apparent on reading the documentation. The actual implementation of the additional hot spots adhere to standard framework practice of maintaining invariants within the framework and forcing the clients to override abstract methods.

SpeechSession. Figure 4 shows an implementation of the first user-identified hot spot in which the active ISpeechClient must be explicitly specified by the framework user. The new hot spot requires the framework user to create a concrete AppSpeechSession class derived from an abstract ISpeechSession class. SpeechSession hides from the user the speech subsystem's Facade object, which was previously a public framework class. To implement AppSpeechSession, the framework user must define the pure virtual method, `getSpeechClient()`, which the framework invokes to get the active ISpeechClient.

This hot spot necessarily draws the framework user's attention to the fact that an active ISpeechClient must be specified prior to beginning the recognition process, which alleviates the problem of designating the active ISpeechClient when the application starts. However, the behavior of updating the active ISpeechClient when the foreground window changes may also be implemented by the same hot spot. The creation of a new abstract class that derives from ISpeechClient and IFrameWindow can encapsulate the change in foreground window notification from the IFrameWindow class and invoke the `getSpeechClient()` method.

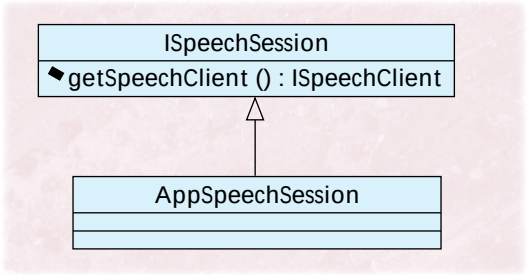


Figure 4. User-identified hot spot: Which ISpeechClient is active? The new hot spot requires the framework user to create a concrete AppSpeechSession class derived from an abstract ISpeechSession class. This hot spot necessarily draws the framework user's attention to the fact that an active ISpeechClient must be specified prior to beginning the recognition process.

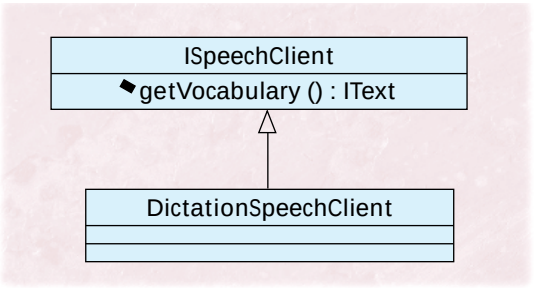


Figure 5. User-identified hot spot: Which vocabulary is active? When a particular ISpeechClient is active, the appropriate vocabulary must be activated. Enforcing this rule by invoking a pure virtual method, `getVocabulary()`, in the active ISpeechClient ensures that the active vocabulary gets updated by the framework user for each new active ISpeechClient.

SpeechClient. Figure 5 shows an implementation of the second user-identified hot spot. When a particular ISpeechClient is active, the appropriate vocabulary must be activated. Enforcing this rule by invoking a pure virtual method, `getVocabulary()`, in the active ISpeechClient ensures that the active vocabulary gets updated by the framework user for each new active ISpeechClient. This method returns the right vocabulary based on the state of that particular ISpeechClient. Here again, this hot spot forces the framework user to specify the active vocabulary prior to beginning the recognition process, which also ensures that there are no surprises regarding active vocabularies and recognized words received by the active ISpeechClient.

Socket interface applications

The socket interface class of applications was different from the others because we used a specific framework application to provide speech recognition services to numerous non-C++ client applications. We built a VRAD application and extended the framework to support a socket interface to read and write operations on a socket connection, thus making irrel-

Table 2. Framework applications summary.

OO framework	Radiology dictation	Pathology dictation	VRAD applications	Socket applications
External framework classes	SpeechClient	SpeechClient, SpeechText, SpeechObserver	SpeechSession, SpeechClient, SpeechText, SpeechObserver, VocabularyManager	SpeechSession, SpeechClient, SpeechText, SpeechObserver, VocabularyManager
Required extensions	Yes	Yes	Yes	No
Employed patterns	Active Object, Adapter, Facade/Singleton, Service Configurator	Active Object, Adapter, Facade/Singleton, Service Configurator	Active Object, Adapter, Asynchronous Completion Token, Facade/Singleton Observer, Service Configurator	Active Object, Adapter, Asynchronous Completion Token, Facade/Singleton, Observer, Service Configurator,
Speech functions	Dictation, Command	Dictation, Command	Dictation, Command, Grammars	Command, Grammars
Speech functionality (percentage)				
in core framework	75	80	90	97
in the GUI extensions	25	20	10	3

evant the runtime environment of the actual speech application. The relative ease with which we were able to add a socket-based string interface for speech recognition in a new SocketSpeechClient class only reinforced our Adapter approach in the framework.

BUILDING RESILIENCE

Table 2 summarizes different aspects of our experience with the various framework applications we developed. With each new application, the number of framework interface classes grew to encapsulate distinct functional areas, which exemplifies the principle of modularizing to conquer complexity. But we didn't capture all abstractions in the problem domain during the initial analysis phase. In fact, some framework applications led to the creation of new speech interface abstractions necessary in the analysis and design phases.

As a result, though, the framework realized more design patterns allowing for greater resilience. We were able to abstract more domain knowledge into the core framework, which minimized the required GUI-specific extensions. The last rows in Table 2 summarize the breakdown of code split between the core framework and the GUI extensions to the framework. Figure 6 shows the movement of the classes from the application/GUI extension to the core framework, which results in the 97 to 3 percent split from the original 75 to 25 percent.

MOTIVATED AND UNMOTIVATED USERS

The unmotivated user's perspective helped further abstract problem domain concepts. Figure 6 shows an increasing abstraction of speech concepts into the core framework classes with each new application. It shows the change in distribution of functionality between the core framework classes, GUI-specific extensions, and the application—as the framework evolved.

The first framework application, Medspeak/Radiology, forced the framework user to understand and create SpeechObserver and VocabularyManager concepts. By the second application, we realized that these concepts were sufficiently abstract to be encapsulated within the framework. At the end of the third iteration, where we faced the most critical users, we had further abstracted SpeechObserver and VocabularyManager classes into the core framework. This was largely driven by an attempt to alleviate the user's confusion in understanding the collaborations between the VocabularyManager, SpeechObserver, and SpeechSession classes.

The unmotivated framework user's perspective can also contribute to the evolution to a black-box framework from a white-box one.[11] Our original white-box framework required an understanding of how the GUI extensions to the core framework classes work, so that correct subclasses could be developed in the application. The VRAD user's reluctance to accept this forced us to favor composition over inheritance in the application. We continued to use inheritance within the framework to organize the classes in a hierarchy, but use composition in the application, which allows maximum flexibility in development.

REDUCING THE LEARNING CURVE WITH HOT SPOTS

Additional hot spots created as a result of the framework user's perspective play an important role in reducing the learning curve associated with a framework. This conclusion appears somewhat paradoxical because hot spots are typically implemented in frameworks using object inheritance or object composition,[1,2] and both suffer from severe limitations in understandability.

A classic problem in comprehending the architecture of a framework from its inheritance hierarchy is

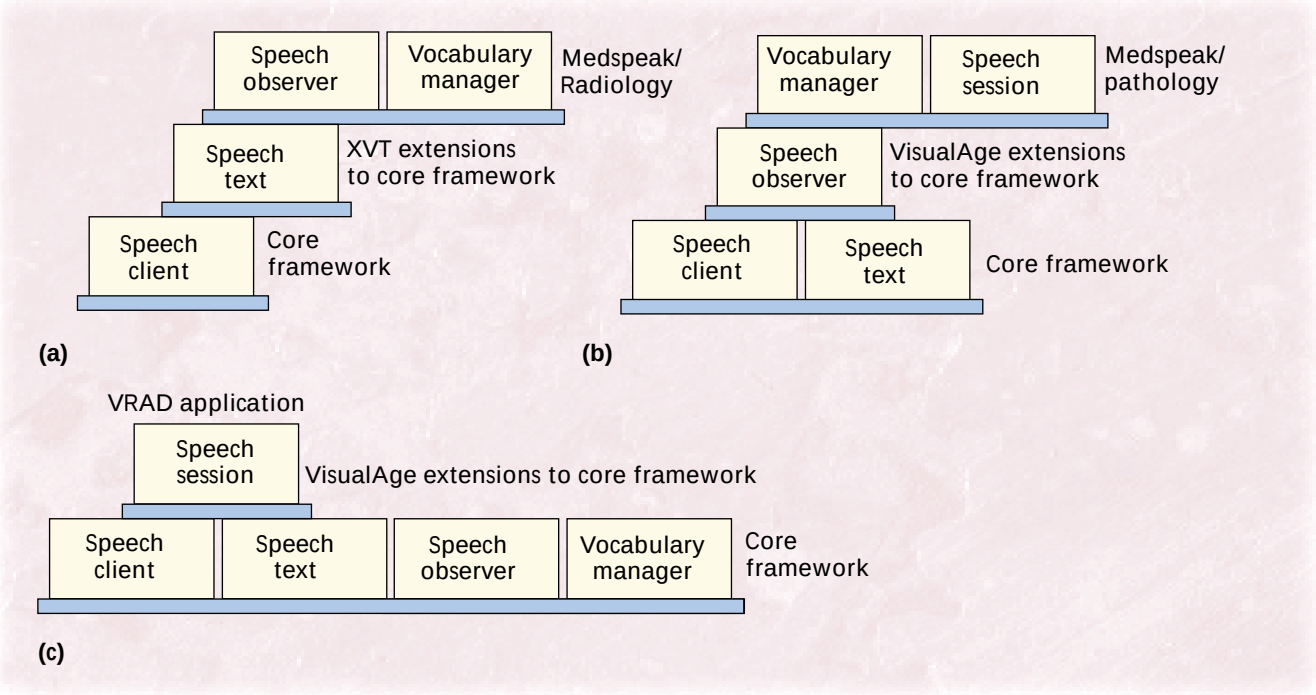


Figure 6. Increasing abstraction of problem domain concepts with each new contract implicit in each application. Each application requires a slightly different architecture: (a) Medspeak/Radiology application, (b) Medspeak/Pathology application, (c) VRAD applications. The Socket application (not shown) simply adds a socket interface class on top of the SpeechSession class as an extension to the VRAD framework classes.

that inheritance describes relationships between classes, not objects. Object composition achieves more flexibility because it delegates behavior to other objects that can be dynamically substituted. However it does not contribute to the understanding of the architecture either, since the object structures may be built and modified at runtime. Despite this, we believe that the hot spots contribute to reducing the learning curve. The primary hot spots address high-level variability, but require an understanding of the underlying contracts and related framework architecture.

Designing for variability appears to be an important facet of the user perspective, too. Applications can be made to obey framework contracts, where previously contracts were merely structured documentation techniques to formalize object collaborations. And since variability is often implemented using design patterns, we were able to achieve an important goal: a common vocabulary for communicating with the framework users. The additional hot spots generated by the user's perspective complement the primary ones and result in a design that necessarily exposes the object collaborations in the framework.

Arguably, the new hot spots identified as the framework evolved could have been identified in the original design. We contend that as domain experts, even though we abstract the common aspects and separate them from the variable aspects, we do so at a sufficiently high level that we mandate a minimum understanding of built-in object collaborations in the framework. Originally, we had identified the hot spot addressing the flexibility required to process recognized words within an application. But this works only if the user understands the underlying collaborations between vocabulary objects and speech client objects documented in a contract. Some of the less glaring hot spots—such as the active vocabulary and active speech client—are apparent as hot spots only when we view the framework from a framework user's perspective. The level of reuse and the productivity gains achieved summarized in Table 2 were obtained by informal user testing, and by evaluating the number of lines of code written and the amount of time spent to develop a new framework application as the framework evolved.

Ultimately, using a common set of abstractions across the analysis, design, and implementation phases and the use of design patterns to capture these abstractions led to better communication and a more efficient technique for handling changes as the framework evolved. Design patterns helped us more effectively communicate the internal framework design and made us less dependent on the documentation. ❖

Acknowledgment

This framework was originally developed as part of the Medspeak/Radiology product with John Vergo at the IBM T.J. Watson Research Center.

References

1. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading, Mass., 1994.
2. E. Gamma et al., *Design Patterns: Elements of Reusable*

Object-Oriented Software, Addison-Wesley, Reading, Mass., 1995.
3. D.C. Schmidt, R.E. Johnson, and M. Fayad, "Software Patterns," *Comm. ACM*, Oct. 1996, pp. 36-39.
4. R. Helm, I.M. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," *Proc. OOPSLA 90*, ACM Press, New York, 1990, pp. 169-180.
5. R.E. Johnson, "Documenting Frameworks Using Patterns," *Proc. OOPSLA 92*, ACM Press, New York, 1994, pp. 63-76.
6. R. Lajoie and R. Keller, "Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert," *Proc. Colloquium on Object Orientation in Databases and Software Engineering*, World Scientific, River Edge, N.J., 1994, pp. 295-312.
7. G. Booch, *Object Oriented Analysis and Design*, Benjamin/Cummings, Los Angeles, 1994.
8. J. Lai and J. Vergo, "MedSpeak: Report Creation with Continuous Speech Recognition," *Proc. CHI 97*, ACM Press, New York, 1997, pp. 431-438.
9. D.C. Schmidt and G. Lavender, "Active Object: An Object Behavioral Pattern for Concurrent Programming," *Proc. Second Pattern Languages of Programs Conf.*, Addison-Wesley, Menlo Park, Calif., 1995.
10. P. Jain and D. Schmidt, "Service Configurator–A Pattern for Dynamic Configuration of Services," *Proc. Third Usenix Conf. Object-Oriented Technology and Systems*, Usenix, Berkeley, Calif., 1997.
11. D. Roberts and R.E. Johnson, "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks," *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, eds., Addison Wesley Longman, Reading, Mass., 1998.
12. T. Harrison, D. Schmidt, and I. Pyarali, "Asynchronous Completion Token," *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, eds., Addison Wesley Longman, Reading, Mass., 1998.

Savitha Srinivasan is a software engineer in the Visual Media Group at the IBM Almaden Research Center in San Jose, California. Her current research interests include using speech recognition and other synergistic techniques, while practicing object-oriented technology. She received an MS degree in computer science from Pace University.

Contact Srinivasan at savitha@almaden.ibm.com.