Structuring I/O Services in Object-Oriented Architectures [*]

Anil A. Gracias, Rushikesh K. Joshi
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Powai, Mumbai - 400 076, India.
{gracias, rkj}@cse.iitb.ac.in

ABSTRACT

Dynamic object creation or state changes often require information related to object's state to be fetched from an environment external to object's context. Similarly, object's state is accessed in external environments through its interface. We discuss a general purpose service architecture for structuring I/O services. The architecture separates the concerns of I/O from object's functionality. The architecture employs event channel type communication, in which, an intermediate I/O channel provides variability to support different types of object factories, observable objects and I/O streams.

Keywords

design patterns, I/O service, intermediate channel, Push/Pull architecture.

1. INTRODUCTION

Objects in an application need to communicate with the external environment for exchange of data. An application fetches input in some form, processes it in its business logic layer and presents results as output in a desired form. If the concerns of functionality and I/O are separated, both can be varied and adapted independently. Though in a given application, I/O streams may communicate over a specific data domain, they may use different forms of data and different styles for communicating. For example, an input stream may be hooked to a command line character sequence, or to a GUI or to a network socket. Similarly, an output stream may be hooked to a GUI, or to a printer or a file.

1.1 Variability in I/O Structuring

The variations in I/O streams can be classified into push style or pull style communication. If an object embeds its I/O implementation scheme, the I/O stream types get tied to a specific choice. For example, consider a text to speech conversion application designed to accept text through GUI and to present the output as on-line synthesized speech. The application would have to be designed with push style input and push style output as shown in Figure 1. This would require the speech engine to be modified if it is to be adapted to a pull style file-based input and a pickup based pull style output, in which, the user may want to pick up the results conveniently at a later time.

[*]This work has been sponsored by Tata Infotech Limited under R & D project code number 02SA011

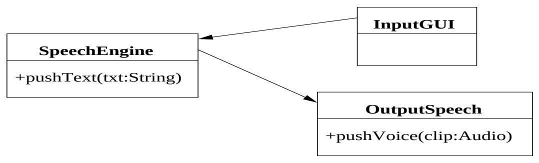


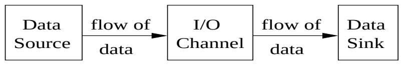
Figure 1: A Speech Synthesizer Application

Moreover, with one style of communication, the form of the streams may vary. For example, we may want the system to be enhanced to accept input from a pdf document in a pull style. Clearly, if the speech engine is overloaded with I/O responsibilities, it becomes cumbersome to manage the adaptations since the concerns of functionality and I/O remain unseparated.

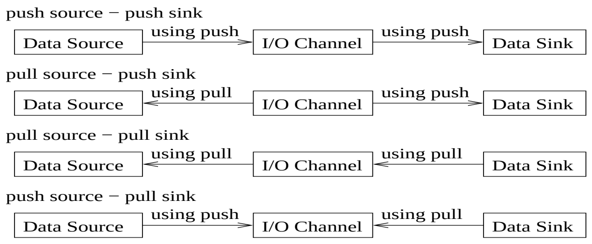
The separation of I/O concerns from functionality not only allows us to carry out adaptations on I/O streams, but also makes it possible to vary the functionality, i.e. speech synthesis in this case, independently of I/O streams. For example, we may want to replace the existing speech engine with a new one with a more pleasing accent, without having to worry about the variations in the data acquisition and delivery mechanisms. Hence to keep the system flexible, it becomes necessary to decouple the concerns of data acquisition and delivery mechanisms from the concerns of functionality. We present the design of an I/O service, which facilitates this separation of concerns. The type of an existing stream may be varied without changing any functional code in the business logic.

2. STRUCTURING I/O STREAMS

The general data flow structure of a system using an I/O channel is shown in Figure 2(a). The I/O channel provides a means of communication between the data source and the data sink. Using the intermediate I/O channel decouples the source from the sink. The channel can be interfaced with a push type source and a push type sink or a pull type source and pull type sink, or even hybrids as shown using representations captured in Figure 2(b). The channel is capable to communicate in any style depending upon the type of source and sink subscribed to it. This makes the source independent of the type of sink. Hence, any changes made at the sink are not noticed by the source and vice-versa. The variations are handled by the channel leaving the communicating components unaffected.



(a) General Data Flow Structure



(b) Different kinds of data sources and sinks handled by I/O channel

Figure 2: General Architecture using I/O Channel

2.1 I/O Channel

The structure of the I/O channel is similar to that of the event channel of the CORBA architecture [1], in the sense that it uses variations of the push and pull style interfaces. The channel implements the PULLCLIENTEC, PUSHCLIENTEC, PULLSERVEREC and PUSHSERVEREC interfaces as shown in Figure 3. Each of the interfaces provides a service to which a component can subscribe to. The channel provides independence from the type of streams by handling both push and pull style within the channel itself. The channel can also provide scalability by allowing multiple streams to subscribe. Depending on the type of streams connected, the channel may need to act as a simple request forwarder, a smart channel or a buffered channel.

The I/O channel has compatible interfaces for clients and servers of both push and pull type. Interface PULLSERVEREC and PUSHSERVEREC respectively define interfaces for pull type and push type servers for the I/O channel. Objects of the type PULLCLIENT and PUSHCLIENT interact with these interfaces respectively. Similarly, interfaces PUSHCLIENTEC and PULLCLIENTEC define interfaces for push type and pull type clients for the I/O channel. Objects of the type PUSHSERVER and PULLSERVER interact with these interface respectively.

3. ENGINEERING OF INPUT ARCHITECTURE

For designing the input of the system, the input streams act as sources producing data to be provided to the channel. The object which is to process this data will act as a sink. The channel forms an intermediate pipe carrying data from the input streams to the data processing object. This section describes static and dynamic models of input architecture based on the design of the I/O channel discussed in the previous section.

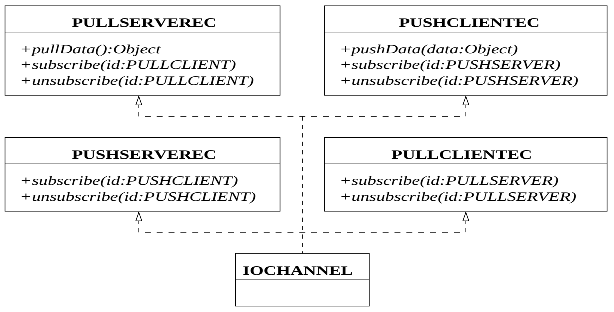

PULLSERVEREC
+pullData():Object
+subscribe(id:PULLCLIENT)
+unsubscribe(id:PULLCLIENT)
PUSHCLIENTEC
+pushData(data:Object)
+subscribe(id:PUSHSERVER)
+unsubscribe(id:PUSHSERVER)
PUSHSERVEREC
+subscribe(id:PUSHCLIENT)
+unsubscribe(id:PUSHCLIENT)
PULLCLIENTEC
+subscribe(id:PULLSERVER)
+unsubscribe(id:PULLSERVER)
IOCHANNEL

Figure 3: Structure of the I/O Channel

3.1 Static Model

The static model of a system using an input channel is shown in Figure 4. The architecture comprises of an application layer requiring to fetch input, an I/O channel and input streams layer.

Application Layer:
Objects requiring to receive data from the input stream are interfaced with one of the server interfaces of the I/O channel. An application object designed to pull data from the I/O streams needs to implement the PULLCLIENT interface and is shown as the PULLCONS class in the Figure 4. Objects of type PULLCONS interact with the PULLSERVEREC of the I/O channel to fetch the required data. If objects in the application layer rather receive data from the input stream, it needs to implement the interface PUSHCLIENT. Class PUSHCONS which implements the PUSHCLIENT interface allows the intermediate I/O channel to push data onto it.

Input Stream Layer:
Input streams, which are to provide the application layer with the required data form a part of the input stream layer. Interfaces PUSHSERVER and PULLSERVER define the interface for push type input stream and pull type input stream respectively. An input stream of push style communication will implement the interface PUSHSERVER and push data to the I/O channel communicating through the PUSHCLIENTEC interface. This class is shown as PUSHIPSTREAM in the Figure 4. An input stream exhibiting pull style communication will implement the interface PULLSERVER and allow the I/O channel to pull data from itself. This class is shown as PULLIPSTREAM.

3.2 Dynamic Model

We capture the behavioral aspect of an input architecture by means of sequence diagrams. Following subsections describe how requests made by the communicating objects are handled by the channel. Depending on the type of application object and the type of input stream interfaced, the input channel alters its behavior.

Input Channel with Push Type Application Object:
Figure 5(a) shows the input channel with a push type of input stream. For this environment, the input channel simply forwards the push message along with the data from the push input stream to the push application object. Figure 5(b) shows the input channel with a pull type of input

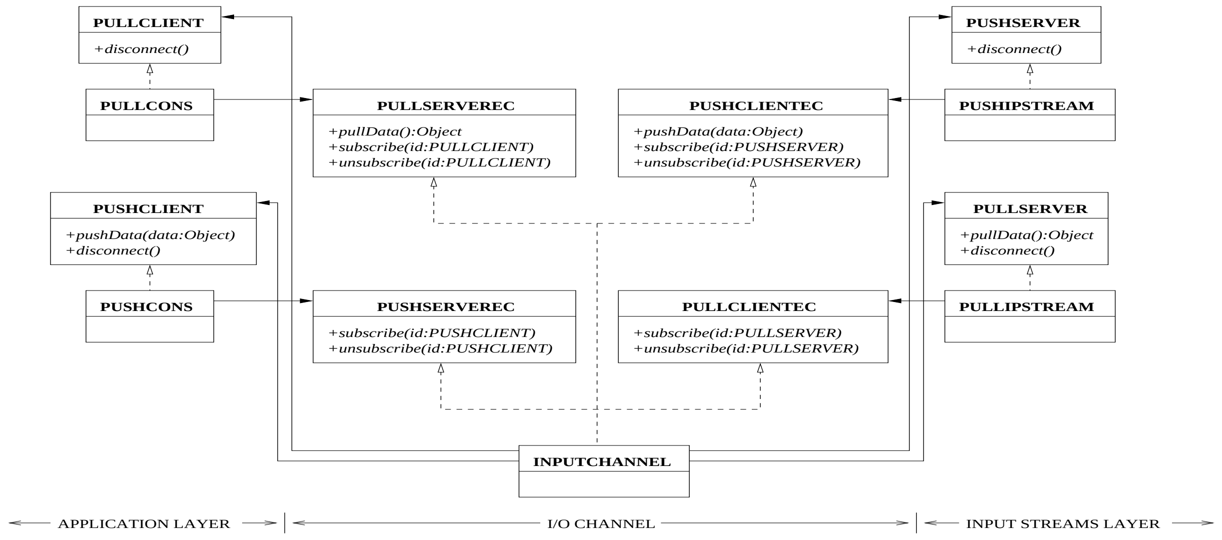


Figure 4: Structure of the Input Architecture

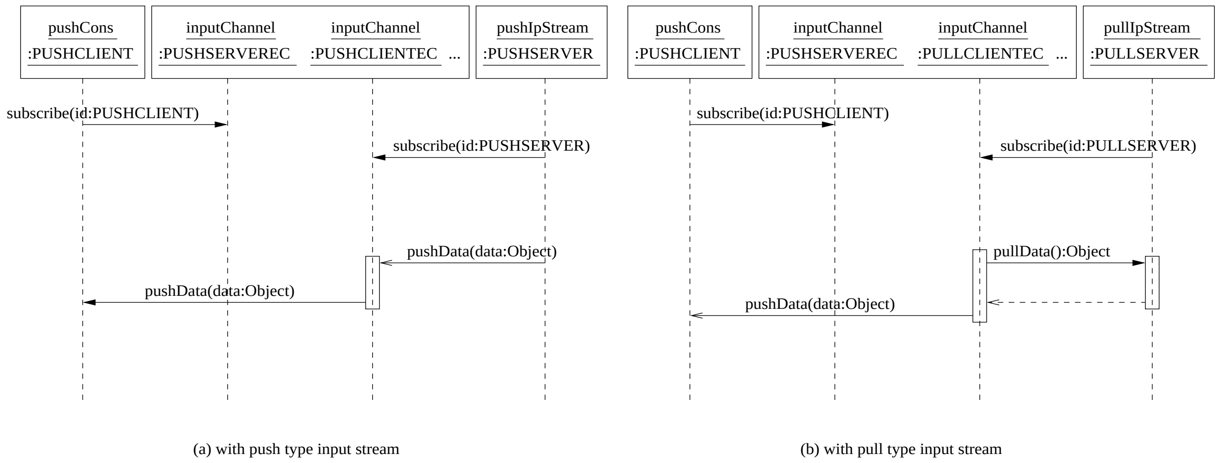


Figure 5: Push Consumer - Input Channel - Input Stream

stream. In this scenario, the input channel acts as an intelligent channel which itself pulls data from the input stream and then pushes it onto the push application object. The pulling may be periodic, wherein the input channel acts as a sampler or the pulling may be governed by some external trigger logic.

Input Channel with Pull Type Application Object: Figure 6(a) shows the input channel with a pull type of input stream. For this environment, the input channel simply forwards the pull message from the pull application object to the pull input stream. The pull application object blocks till the required data is provided to it by the input channel. Figure 6(b) shows the input channel with a push type of input stream. In this scenario both the input stream and the application are active objects and send messages asynchronously. If push message from the input stream occurs before a pull from the application object, the input channel buffers the data obtained from the push message. When a pull request is made from the application object the input channel supplies the buffered data to it. Here, the input channel provides buffering. The pull message from the pull application object can occur first, even before a push from the input stream has been made. In this case, the pull from the application object is blocked till a push message is received from the input stream with the required data.

4. ENGINEERING OF OUTPUT ARCHITECTURE

We have seen how the input channel decouples the input streams from the application layer making the system adaptable for changes in input layer. By using a similar design, the system can be made to adapt to different output streams. Output streams are interfaced with the channel as data sinks. Application objects act as sources and provide data to be delivered to external environment through the output channel. From the designs of the input channel and output channel shown in Figures 4 & 7 respectively, it can be seen that both designs are mirror images of each other. Dynamic model for the output channel will be similar to that of the input channel.

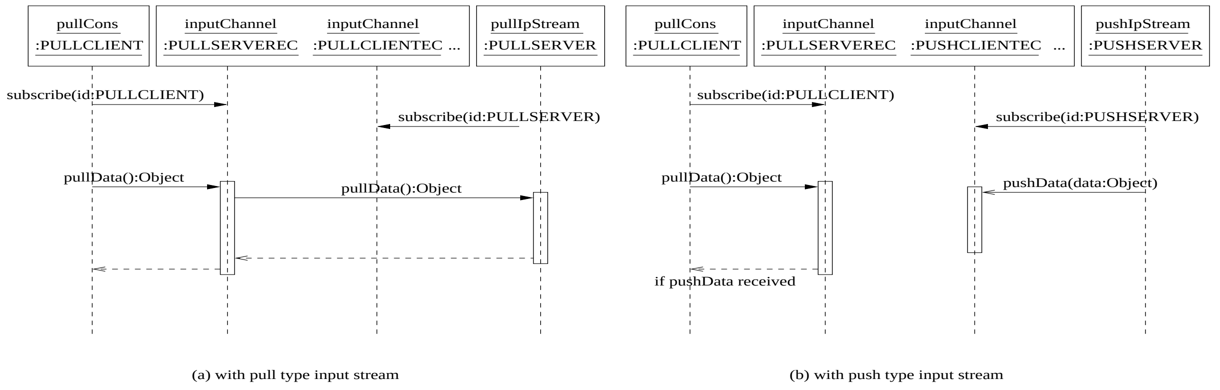


Figure 6: Pull Consumer - Input Channel - Input Stream

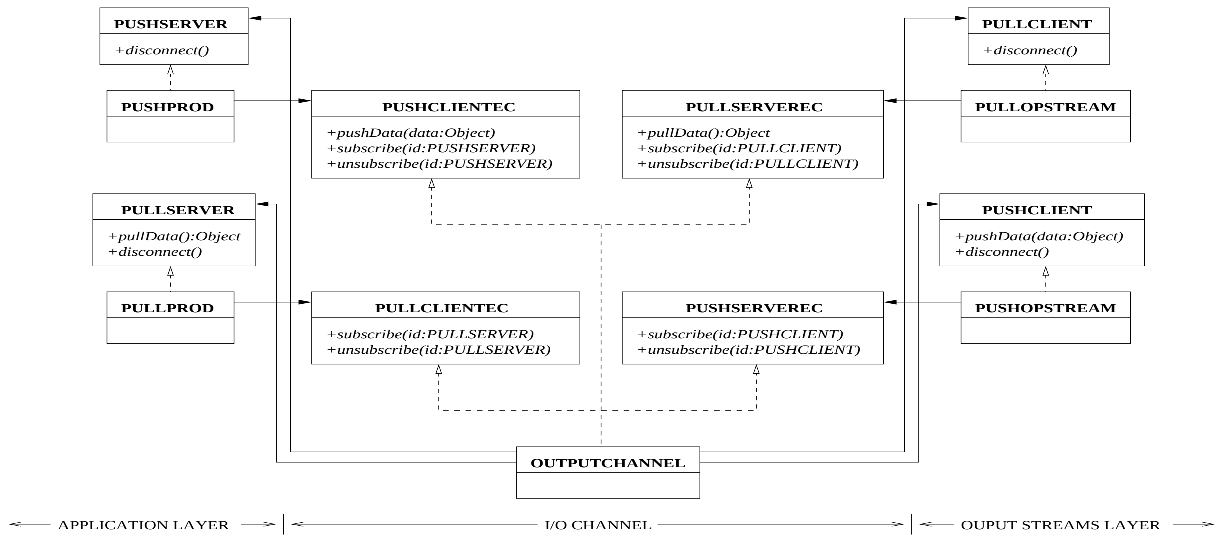


Figure 7: Structure of the Output Architecture

5. INTERWEAVING I/O SERVICES WITH OTHER DESIGN PATTERNS

The I/O Service discussed above can be used in co-operation with other design patterns [3, 6] to enhance the benefits provided by them. The I/O service can be employed in event channel based systems or systems having a client-server type of architecture to provide communication style independence. The design can be used to support style independent peer to peer communication. It is also possible to utilize the I/O channel in creational patterns to fetch the initial state for the object being created. This section illustrates with examples how I/O services can be put to use with other widely used design patterns.

5.1 Pipes and Filters

Pipes and Filters [2, 5] are used to carry out transformations on data streams. Filters act as processing units which can refine or transform data. A series of filters are connected with pipes to carry out complex transformations. Filters may be of pull, push or pull-push type. Depending on the output style of a filter and the input style of the subsequent filter different types of pipes could be required. For example, connecting a push filter with another push filter requires the pipe to do simple forwarding of the data, but if a push filter is to be connected with a pull-push filter, a buffered pipe will be required. The output channel can be used to connect two filters of any type. Using the output channel it is possible to remove a filter of one type and connect a filter of another type without affecting the filter at the other end of the channel. The channel will automatically adapt to the style of communication of the newly attached filter.

5.2 Producer-Consumer

In a producer consumer environment, the producer can create data and have them delivered in either the push or the pull style. Similarly, the consumer can get data in either style. The producer and consumer can communicate with each other in three ways, push-push, pull-pull and push-pull. Inserting the output channel between the producer and the consumer decouples them allowing any type of producer and any type of consumer to be interfaced with each other. In addition to the existing three communication ways, the producer and consumer can communicate in a pull-push fashion, wherein, the output channel is the active element pulling data from a passive producer and pushing the data to a passive consumer.

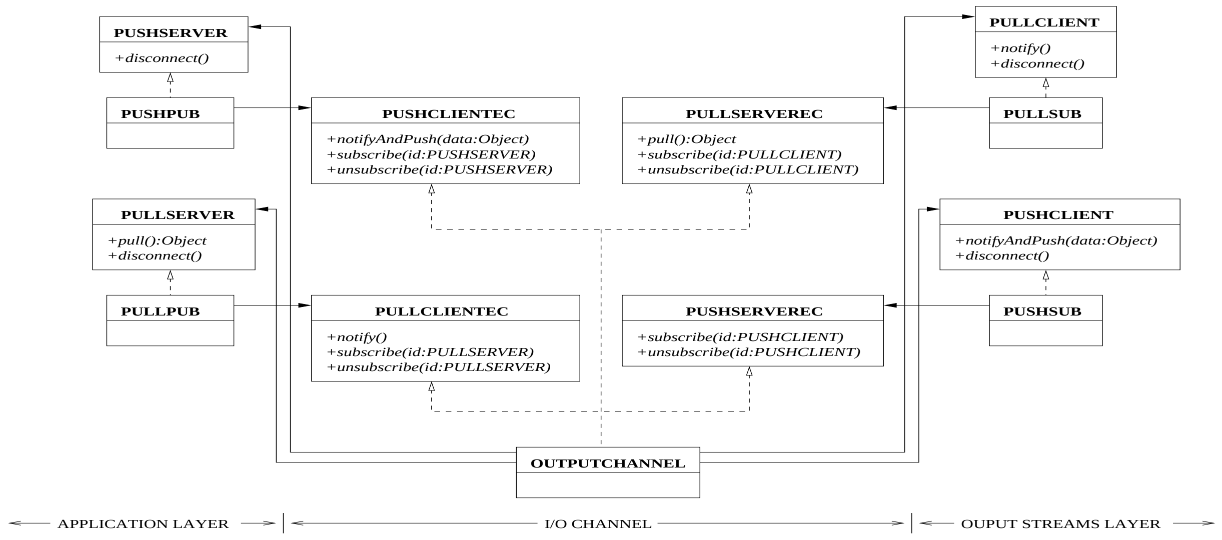


Figure 8: Publisher-Subscriber using I/O Service

5.3 Publisher-Subscriber

The publisher-subscriber design pattern [4] is used when an object wishes to communicate information to a set of interested objects. Objects providing the information are called publishers and the receiving objects are termed subscribers. The publisher notifies all subscribed objects on availability of information. Data is then communicated between the publisher and the subscribers. The data can be communicated in two ways. The publisher can push data along with the notify message to the subscribers or the publisher simply sends a notify message and the subscribers later pick up data from it in pull style. By communicating through the output channel, it is possible to interface any type of publisher with any type of subscribers. The subscription information is also handled by the output channel. The publisher-subscriber pattern using the output channel is shown in Figure 8. The output channel has a few changes. The push method in the PUSHCLIENTEC is to be taken as notifyAndPush and the PULLCLIENTEC has an additional notify message to trigger the pull type client to pull data from the pull server.

5.4 Builder

Builder [3] is used for separating the construction process of an object from its representation. This allows us to use the same creation process to form different representations of the object. The I/O service can be used if the system requires to fetch data from an external environment or from some other object. The builder can be interfaced with an input channel to fetch data external to it for the construction of the object. The builder can act as a pull client which pulls data from the channel when the director of the builder pattern instructs it to construct the object. It is also possible to have the builder as a push client which accepts data pushed onto it by the channel but constructs the object only when the director requests construction of the object. The input streams may be of any type and will be handled transparently by the input channel. The builder is not aware of the types and forms of the input streams.

5.5 Factory Method

The factory method pattern [3] defines the interface of the creator class and delegates the actual construction of the product to subclasses. The concrete creator class which actually creates the product can make use of the input channel for fetching data required to construct the product. The concrete creator can implement the pull client interface and pull data from the input channel when it is instructed to create the product. It is also possible for the concrete creator to implement the push client interface and allow the input channel to push the required data to it. Later when the product is demanded, it can use the data pushed to it to create the product. Since the input channel decouples the factory from the input streams, any type of input stream can be interfaced and varied at the input channel without affecting the factory code.

5.6 Forwarder-Receiver

The forwarder-receiver pattern [2] is used for peer-to-peer communication. Each peer communicates with the other only via forwarder and receivers. The forwarder acts as a push client allowing a communicating peer to push messages to it. The receiver acts as a pull server and waits for the peer to read a message. Hence, every peer needs to interact with its forwarder in push style and with its receiver in pull style. The channel between the forwarder of one peer and the receiver of the other peer is buffered for supporting such a push-pull type of communication between peers. It could be possible that we have peers which wish to communicate its output in pull style and get its messages in push style. Incompatible communication styles between peers would require the forwarders and receivers to be changed accordingly so that they adapt to this new style of communication. By using the I/O channel, it is possible to adapt to such communicating peers without changing any code of the forwarder or receiver. Two I/O channels are required between any two communicating peers. Input channel of one peer will act as the output channel for the other and vice-versa.

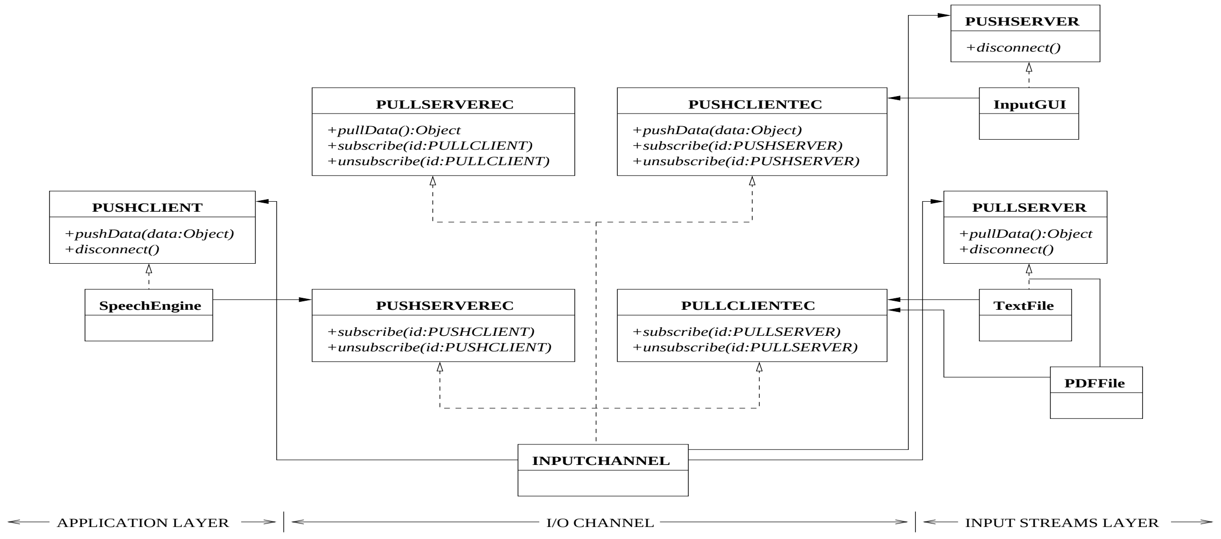


Figure 9: Speech Synthesizer Application using I/O Service

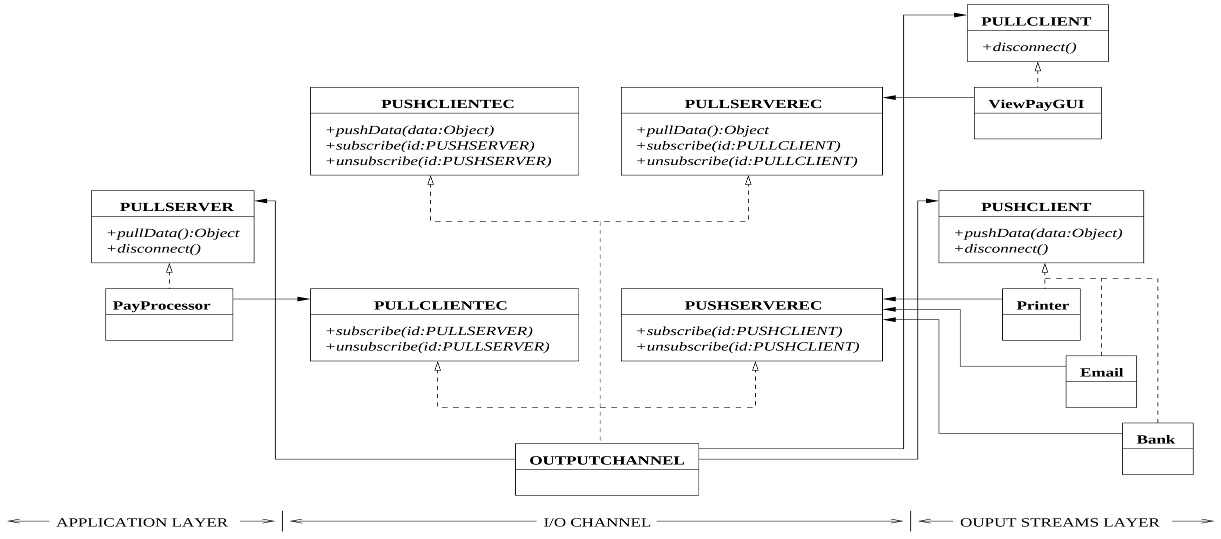


Figure 10: Payroll System using I/O Service

6. EXAMPLE APPLICATIONS

6.1 An Input Architecture

Consider the speech synthesizer application discussed in Section 1, depicted in Figure 1. The new design shown in Figure 9 is a flexible architecture enhanced to adapt to variations in the input layer. The speech engine is decoupled from the mechanisms used to fetch data from the input stream. Hence it is no longer dependent on the type of the input streams. The initial speech engine acts as a push client accepting text from the GUI input. Synthesizing speech from the text file would require the speech engine to act as pull client and pull data from the text file. Each time a new input stream is added the speech engine will have to undergo changes to adapt it. But by using the input channel, any input stream, push or pull type can be added to the system without requiring changes to be made to the speech engine. New streams can be added by simply registering the stream with the input channel making the system scalable. Also, on availability of an enhanced speech engine, the existing speech engine can be easily replaced and it automatically adapts to the existing input streams.

In absence of the I/O Service, the speech engine will be overloaded with all I/O responsibilities. Using I/O Service, these extra responsibilities are delegated to the I/O channel and other objects. The input channel provides stream type independence and can allow multiple streams to be added without changing the speech engine class.

6.2 An Output Architecture

Consider a payroll system, which needs to output data to different output streams. Figure 10 shows the design of the output section of the payroll system. The system allows the user to view the pay details generated for an employee and issue pay slips. The pay details are calculated by a pay processor which acts as a pull server allowing pay information to be retrieved from it. A query class acts as a pull client and fetches data from pay processor. The query class is compatible with the pay processor and both form the pull style environment. To issue pay slips, the information is required to be pushed to the printer. The printer in itself is of push client type. By using the output channel, the task of issuing pay slips through the printer can be incorporated without changing the pay processor class.

New output streams irrespective of its form and type can be added with ease. It may be desirous to send the pay-slip as an email attachment to the employee as a reference. The email client which behaves as a push client can be incorporated into the system. Or further still, the system may wish to directly make payments by crediting the bank account of the employee. This would simply require that a bank class capable of handling transactions be added as a push client. The output channel will provide the bank client with the information required for carrying out the transaction.

7. CONCLUSION

The I/O Service architecture separates I/O concerns from the functional code of objects. This design keeps functional objects independent of both the type and form of the communicating streams by means of a decoupling channel. One can unplug a stream of one type and plug in a stream of a different type without needing to change the existing application object. The functional(application) object has no knowledge of the variations carried out in the type of streams and hence remains unaffected. I/O services when used with other design patterns enrich them providing independence from communication style type and form.

8. REFERENCES

[1] Object management group. *CORBA Services: Event Service Specification, v1.1*, (01-03-01), March 2001.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, England, 1996.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, New York, NY, 1995.

[4] R. Kannan. Managing continuous data feed with subscriber/publisher pattern. *OOPSLA'95, Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, (TR-SE-DNA-95-003), October 1995.

[5] A. Vermeulen, G. Beged-Dov, and P. Thompson. The pipeline design pattern. *OOPSLA'95, Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, October 1995.

[6] J. Vlissides, J. Coplien, and N. Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, Massachusetts, 1996.