Proceedings of the 9th International Conference on Applied Informatics
Eger, Hungary, January 29–February 1, 2014. Vol. 2. pp. 283–292
doi: 10.14794/ICAI.9.2014.2.283

Building scalable distributed network systems using a modified pipeline design pattern*

Balázs Kreith

University of Debrecen, faculty of Informatics, Debrecen Hungary
kreith.balazs@inf.unideb.hu

Abstract

This paper attempts to do three things. The first is to examine the so called pipeline design pattern for designing and implementing applications. The second is to modify this pattern in order to make it available in programming languages supporting single inheritance only. The third goal is to use this pattern in a context of developing distributed network systems in which several devices and participants communicate and work simultaneously. We have developed our own framework in C# based on the pipeline design pattern and used it to build distributed network systems. We show the usability of the new pattern and suggest some directions for the future.

Keywords: pipelining, flow-based programming, internet of things, developing distributed applications, device communication framework

1. Introduction

Pipelining is a generally used technique in low-level hardware programming[1]. This implementation technique uses a sequence of parallel executed stages. (Figure 1).

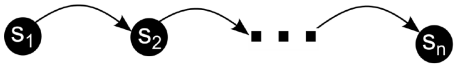

Figure 1: Sequential processing of an arbitrary task

The parallel execution of stages $(s_1, s_2, \ldots, s_n)$ allows us to increase the number of executed tasks in a given range of time.

*The work was supported by the TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project. The project has been supported by the European Union, co-financed by the European Social Fund.

Each stage contains a set of instructions, an input and an output. Every stage input is the output of the previous stage. The data transmission method between the stages can be various [2], here we focus on the asynchronous method.

In high-level programming languages, several frameworks [3,4,5] and concepts [6,7] exist, and a design pattern for pipelining has been introduced by Vermeulen et. al. [2]. This describes an object oriented concept implemented in C++ [8], and using reusable objects for stages. Applying this pattern to decompose the process implementation and its interaction, a better modularity may be achievable, along with the parallelization possibility in processing and in execution both. This provides greater flexibility and scalability and it increases the modularity of the software and allows us to implement new features easier while maintaining the application integrity. This design pattern uses multiple non-virtual inheritance, which is not allowed in popular languages like Java or C#.

In this paper we rediscover this design pattern, examine its possibilities further and redesign it using single inheritance. The modified pipeline design pattern is implemented in .NET, but other high-level object oriented languages can use it as well. At the end of our discussion we show an example. The rest of the article is organised as follows: Section two describes the original design pattern, its terminology and advantages. In section three the structure of the pattern is changed and new concepts are introduced. The concrete realization is discussed in section four, where simple examples are given too. In section five we show how to use this pattern in developing a complete system by building a full process network. Finally we describe the possible future works.

The main contribution of this paper is introducing a modified pipeline design pattern and showing its efficiency.

2. Overview of the original design pattern

Pipelined processes are built up by sequentially connected stages (Figure 1.). Each stage is producing, consuming, transforming, or filtering data for the task it is created for. Every stage has an input, an output, or both. By connecting these interfaces, a pipeline is defined and it can be used for data transfer.

2.1. Terminology & structure

In Vermuellen et al. [2], stages are called processing elements. If a stage has only input or only output, it is called an endpoint pipeline processing element (EPPE). Stages that have both input and output are called interior pipeline processing elements (IPPE). In implementation, inputs are inherited from the Reader and outputs from the Writer class. Two processing elements are connectable if one part is a Writer and the other part is a Reader.

Depending on the method of how data are entering from a predecessor stage's output into the successor's input, three transfer types are defined: push-flow, pull-flow or push-pull hybrid flow (commonly referenced as "data-flow", see [2]). In

the push-flow concept, a predecessor stage's Writer sends data to its successor's input (which receives it). In the pull-flow concept, a successor stage's Reader demands data from its predecessor's output. (which supplies it). To perform data-transfer, one part (input or output) must initiate the transmission. This part is called the active part. The other side, which is fulfilling the request is called the passive part. Extending the Reader and Writer classes with their possible behaviours, ActiveReader, ActiveWriter. PassiveReader and PassiveWriter classes are created. These are EPPEs. By combining these classes using multiple inheritance, the following IPPEs are created: ActiveReaderPassiveReader, ActiveReaderPassiveWriter, PassiveReaderPassiveWriter and PassiveReaderActiveWriter. For the Booch diagram of these elements see [2]. For a better understanding, the next section describes an example.

2.2. Example of usage

To realize a push-flow mechanism, an ActiveWriter must connect to a PassiveReader. A pull-flow operation is realizable by connecting a PassiveWriter to an ActiveReader.

A data-transfer is initiated by an ActiveReader or an ActiveWriter. A stage is executing its process after acquiring data. Parallel execution of stages is done by using threads.

For synchronization, a buffer has to be introduced between two threads. This is a PassiveReaderPassiveWriter. An example of constructing a pipelined image processing is shown in Figure 2.

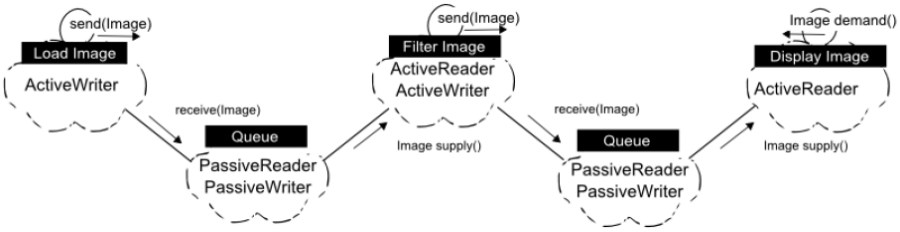

Figure 2: Pipeline design pattern for image filter processing

In the example shown in Figure 2, the tasks are the following: loading images filtering them and then saving the results. Here, a push-pull hybrid flow mechanism is used. The Load Image stage is using push-flow by sending the loaded image into the first queue. When the Filter Image stage is ready for execution, it demands an image from the first queue. After the filter procedure is done, it sends the filtered image to the second queue. Finally, the Display Image stage, which is using a pull-flow, continuously demands an image from the second queue and displays it. Here, every stage that contains an active interface is running on its own thread. A queue between two threaded stages is used as a buffer for synchronization. Queues, Load Image, Save Image and Filter Image elements are concretized stages inherited from a basic element. The stages' processes use the functor idiom

[9] in the implementation. Queues can be FIFO, LIFO, priority queue, etc. The different threads run parallel, and the buffers ensure the synchronization between these threads, thereby the time needed for filtering several images continuously is reduced.

The connection methods of different data-flow mechanisms (one reader to many writers, or reversely) gives further possibilities for designing applications that perform a process.

2.3. Further possibilities and limitations

In the data-transfer mechanism, one active part can send data to or demand data from one passive part, but one passive part may receive from or supply many active parts. This indicates that in a push-flow mechanism, one PassiveReader may receive data from many different ActiveWriters, and in a pull-flow mechanism, one PassiveWriter may supply many ActiveReaders. The first case (one PassiveReader to many ActiveWriters) opens the possibility for multiplexing messages from different senders, and the second case (one PassiveWriter to many ActiveReaders) can be used for distributing data amongst many stages.

Regarding the example described in Figure 2, let us suppose the following: The load and image stages are much faster than the filter image stage. For that reason, we can introduce many filter stages and connect them to the first and second queues. This can be done without any structural modifications on the concrete classes, with slight modifications in the implementation.

Going further, we may want to distribute the calculation amongst several computers, or we may want to create a more complex network between the stages for controlling the data-flow.

To Distribute the computation we need to create stages, which demand data from the first queue via a network connection and send the result data to the second queue. These new stages need to handle network connections, data transmissions, connection failure, etc.

Suppose we count the number of images available in the first queue and want to pause the load state if the number of available images, for instance, is more than five. Using the pipeline concepts, we want to send information from the first queue to the Load image stage whether we need more image or not. For that reason, we have to add new interfaces. This can be done by inheriting the appropriate base classes, however we cannot use one base class more than once, which limits our possibilities.

We modified the original design pattern for two reasons: First, we changed the inheritance hierarchy in order to implement it in object oriented languages that allow only one base class (i.e.: Java, C#). Secondly, we reorganized the structure to extend the flexibility of the original design pattern.

3. Modifications on the original design pattern

Our approach for redesigning is based on the decomposition of responsibilities in the pipelining design. We determine three area of responsibilities: Processing, Connection, Transmission.

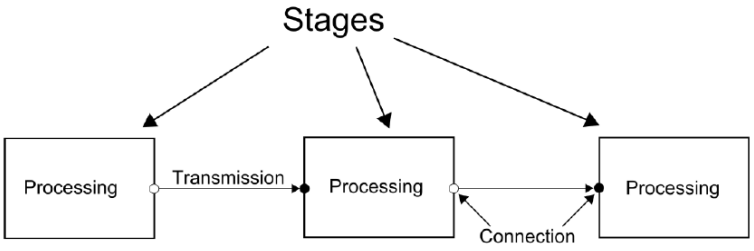

Figure 3: Modifications related to processing, transmitting and connecting stages

3.1. Redesigning the original structure

A stage is created to perform a task. We focus on this in our design's inheritance hierarchy in order to create abstract stages. The transmission and connection methods are implemented in different classes and they are referenced as an attribute by a derived abstract stage. The connection is responsible for establishing a connection between two stages. It is realized in a class named Plug. Depending on how many interfaces we need, a stage may contain as many Plugs as necessary. A plug may be bound or it may be connectable. A connectable plug connects and sends (or demands) data to (or from) a bound plug. This is similar to the previously mentioned connection between an active and a passive part. Plugs bind a transmission or connect to one via a bound plug. The transmission's main responsibility is to ensure data exchange between two stages. They can use different technologies to fulfil this task (delegates, socket communications, COM interfaces, etc.).

This separation isolates the structure of connecting two stages, transmitting data between them and performing an operation on a received (or requested) data.

3.2. Reorganizing the inheritance hierarchy and introducing new elements

For the three possible data-flow mechanisms, we created three packages: Pushers, Pullers and Adapters. The Pushers and Pullers packages realize push-flow and pull-flow mechanisms, while elements of Adapters are used for realizing a push-pull hybrid flow (Figure 3).

Both the Pushers and Pullers packages contain the following four basic elements: Transmitter, Transceiver, Receiver and Threader. Transmitter is a stage, which

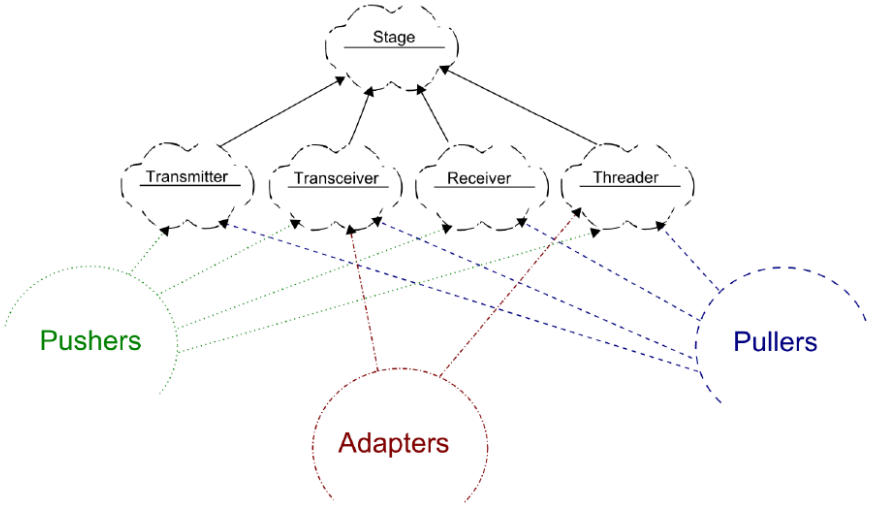

Figure 4: Boch diagram of the modified pipeline design pattern

has output only, and Receiver is a stage, which has input only. They are EPPEs. We created two basic IPPEs for Pushers and Pullers packages: Transceiver and Threader. Both have one input and one output. Transceiver performs a process on data that goes through a stage. Threaders are active objects [10] running on their own thread and they contain a buffer. The main task for a threader is to separate its predecessor and successor stages into two different threads. The push-flow type Transmitter and the pull-flow type Receiver are also active objects. Transceivers, being a pull-flow type Transmitter and a push-flow type Receiver, are passive objects. They are activated by their input or output. The push-flow type Transmitter and the pull-flow type Receiver are active objects.

The Adapters package contains IPPEs for two possible cases. When a push-flow element is desired to connect to a pull-flow element, then only time the buffer needs to store the data sent by a predecessor push-flow stage is until the successor pull-flow stage demands it. This element is called an Accumulator. The opposite of this case is when a pull-flow type stage wants to connect to a push-flow type stage. In that case, an element is necessary for demanding data from its predecessor pull-flow stage and sending it to a push-flow type successor. This element is called a Relayer, and it is an active object. Accumulators and Relayers are IPPEs.

The above mentioned elements (Transmitter, Receiver, Transceiver, Threader, Accumulator, Relayer) are basic abstract elements. A summary of all basic elements used for realizing push-flow, pull-flow or hybrid-flow mechanisms are shown in Figure 5.

We introduced a graphical notation for different types of basic abstract elements as it is shown in Figure 4. Rectangles indicate stages. Thicker lines indicate active objects. Stages, which are acting actively also contain a thread notation inside

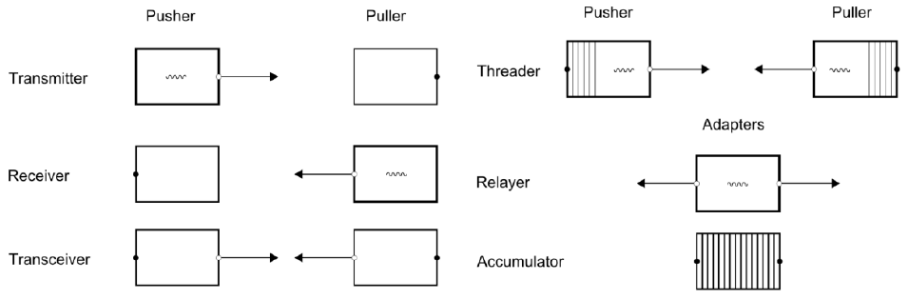

Figure 5: Abstract elements of the modified pipeline design pattern

of their rectangles. Circles on the stages' left and right side indicate plugs that are responsible for connecting stages with each other. Empty circles are indicating connectable plugs, and filled circles indicate bound plugs. A connectable plug can connect to a bound plug. Circles on the stages' left side always indicate inputs. On the ride side outputs are indicated. The transmission is represented by an arrowed line. The direction of the transmission is depending on the type of the data-flow. These lines from plugs indicate transmissions. If the arrow points to the right, it is a push-flow mechanism, and if it points to the left, it is a pull-flow mechanism. Consequent lines inside a stage indicate buffers. Buffers are used by Threader and Accumulator stages.

The above described modifications along with the introduced basic abstract elements open the possibility for designing an application using pipelined processes such as image processing. The above described graphical notation is developed for documenting an application design.

4. Creating applications using the modified pattern

Generic programming is used in the implementation. Stages are only connectable if the types of the data going through the interfaces, which we want to connect with each other, match. Decomposition of Plugs and Transmissions allows us to create a flexible design to the application we want to realize.

4.1. Creating a simple application

In the following, we show a concrete program that uses the modified pipeline design pattern and realizes the example introduced in section 2 (load-filter-save image).

In Figure 6, the previosly described image filtering process is shown by using our graphical notation. The code belonging to this example is shown in code snippet 1. The abstract elements introduced in previous sections are used as base classes of the concrete elements (LoadImage, FilterImage, SaveImage). The connection between the stages is configured in the main program.

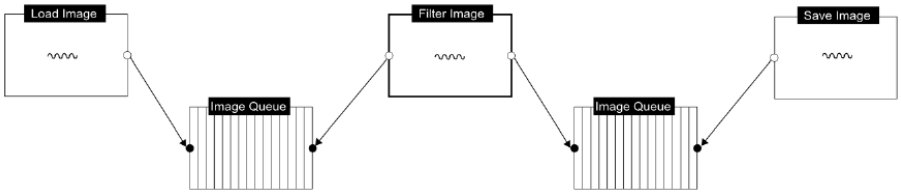

Figure 6: Modified pipeline design pattern for image filter processing

```
class LoadImage :
Pushers.Transmitter<Image>
{
    protected override Image Processing()
    {
        //...load image process...
    }
}

class FilterImage :
Adapters.Relayer<Image, Image>
{
   protected override Image
Processing(Image input)
   {
      //...filtering image process...
   }
}

class SaveImage : Pullers.Receiver<Image>
{
   protected override void
Processing(Image data)
    {
       //...save image process...
    }
```

```
class Program
{
 static void Main(string[] args)
 {
   var loading = new LoadImage();
   var queue1 = new Accumulator<Image>();
   var filtering = new FilterImage();
   var queue2 = new Accumulator<Image>();
   var saving = new SaveImage();

   loading.Output.Connect(queue1.Input);
   filtering.Input.Connect(queue1.Output);
   filtering.Output.Connect(queue2.Input);
   queue2.Output.Connect(saving.Input);

   loading.Start();
   filtering.Start();
   saving.Start();
  }

}
```

Code snippet 1: Source code of the image filter processing

4.2. Creating distributed network application

The decomposed Transmission and Plugs allows us to distribute the application amongs several hosts. This can be done by creating our own Transmission and using it in the Connection method or we can create a component for this purpuse.

In Figure 7, two additional components are added to the previosly described example: A TCP sender and the TCP receiver components. The application task is distributed among the hosts (Host1, Host2) using the additional components.

Another advantege of the decomposition of Plugs and the Transmission is to add as many plugs to a stage as much we want. More plugs of one stage indicate more interfaces, thereby a graph of process interactions may be described by using these elements..This allows us to build many kind of applications and services.

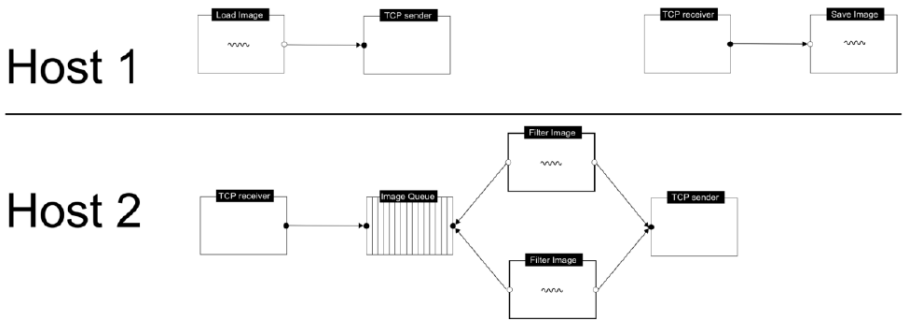

Figure 7: Distributed filter processing applications using the modified design pattern

5. Conclusions

Here we described modifications of the pipeline design pattern, in order to use it in programming languages like C# and Java. With these modifications a concept for building application is introduced and a flexible design is created for te applications. The pattern is applyable for designing many kind of applications and services with different languages and platforms, also it is a usable concept for distributing the performance amongs several hosts.

References

[1] FOUNTAIN, Terence, and Peter Kacsuk. Advanced computer architectures: a design space approach. Pearson Education India, 1997.

[2] Vermeulen, Allan, Gabe Beged-Dov, and Patrick Thompson. "The pipeline design pattern." Proceedings of OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems. 1995.

[3] Win32, S. D. K. "Programmer's Reference-MSDN Library,"." Anonymous Pipes,"(C) 1996 (1992): 1-27.

[4] Lauterburg, Steven, et al. "A framework for state-space exploration of Java-based actor programs." Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, 2009.

[5] Thurau, Martin. "Akka framework."

[6] Agha, Gul Abdulnabi. "Actors: a model of concurrent computation in distributed systems." (1985).

[7] Morrison, J. P. "Flow-based programming: A new approach to application development, self-published: EAN-13 978-1451542325, 2010." (2012).

[8] Stroustrup, Bjarne. The C++ programming language. Pearson Education India, 1995.

[9] Coplien, Jim. "Advanced C++ programming styles and idioms." Technology of Object-Oriented Languages, International Conference on. IEEE Computer Society, 1997.

[10] Lavender, R. Greg, and Douglas C. Schmidt. "Active object–an object behavioral pattern for concurrent programming." (1995).