

The Sequence Pattern

A thesis submitted in partial fulfilment of the requirements
for the M.Sc. in Software Engineering

NAME

Department of Computer Science,
National University of Ireland, Maynooth (Ireland)
e-mail: name@yahoo.com

Abstract

The purpose of this paper is to describe a design pattern named Sequence pattern. This design pattern defines a computational execution model for software components based on the pipeline execution model. It provides support for method aliasing and runtime polymorphism. The Sequence pattern decouples method invocation from method execution. It further decouples structural behaviour from the execution behaviour. This design pattern supports the configuration of collaboration models between heterogeneous software components to be determined at runtime. The Sequence pattern enforces incremental development process and offers openness to new paradigms.

Table of Contents

1.	Introduction	1
2.	Background	1
2.1.	Design Patterns	1
2.2.	Batch Jobs and Pipelines in Software Design	2
3.	The Sequence design pattern	4
3.1.	Intent	4
3.2.	Also known as	5
3.3.	Problem	5
3.4.	Forces	5
3.5.	Solution Structure	5
3.6.	Solution Participants	6
3.7.	Collaborations	8
3.8.	Consequences	9
3.9.	Liabilities	10
3.10.	Implementation	11
3.11.	Sample Source	12
3.11.1	Other implementation issues: Placeholders and Exceptions	14
3.12.	Known uses	15
3.13.	Related patterns	15
4.	Conclusions and Future Work	16
	Appendix A: Bibliography	18

List of Figures

Figure 1: the monolithic block (pseudo code)	2
Figure 2: the Pipeline architecture	3
Figure 3: the Filter and Pipe pattern	3
Figure 4: the Template architecture (pseudo code)	4
Figure 5: the Sequence pattern (pseudo code)	4
Figure 6: the Sequence pattern (UML class diagram)	6
Figure 7: the Shared Repository pattern	6
Figure 8: the Facade pattern	7
Figure 9: the Adapter pattern	7
Figure 10: the Command pattern	7
Figure 11: the Structural behaviour	8
Figure 12: the Execution behaviour	9
Figure 13: the pattern relationships in a pattern language	15
Figure 14: the Sequence pattern (pattern language)	16

1. Introduction

In this paper we will describe the Sequence pattern, a design pattern with similar behaviour to other patterns (pipeline [SHA96], Filter and Pipe [BUS95], Filter [GRA98] and Composite Filter [YAC01]), but overcoming some drawbacks of the former patterns, due to the architecture schema provided by the pattern. The pattern also supports incremental development, due to the separation of concerns between structural behaviour and functional behaviour.

After this introduction, section 2 discusses different pipeline and batch jobs architectures used at present and describes the design pattern rationale. Section 3 presents the actual Sequence design pattern. Finally, section 4 summarizes the paper, drawing some conclusions and outlining further research activities.

2. Background

Since the origins of computing (in the mid 1950), the *batch job system* [SIL98] have been associated with the *developer toolbox*; in this kind of system, the developer used to script some code in order to process several chained jobs. The former systems evolved towards *spool* (Simultaneous Peripheral Operation On Line) *batch systems* [SIL98]; these systems added a queue of batch jobs to be processed in order to optimise the use of the processor resources. Afterwards (in the mid 1960 until present), it was the time for *multiprogramming systems* [SIL98] (based on spooling batch systems to load several jobs at once, and then cycle through them, working on each one for a specified period of time) and the *timesharing systems* [SIL98] (extending the multiprogramming systems technique to allow several users to use the same processor, splitting the processor time between the users).

As we observed, most modern systems are based on a simple concept: the batch job systems. Nevertheless, this concept is flexible enough to allow us to adapt it to new software and hardware paradigms (functional and object paradigms [PRE94], processor architectures such as *Symmetric Multiprocessing* and *Massively Parallel Processor* [PAT98][HWA98]...). For these reasons, we think that realising a *design pattern* of this kind of system would increase the understanding of the system itself, in order to reuse it in new scenarios and improve the maintenance of the existing ones.

2.1 Design Patterns

Why would I describe a system by means of *design patterns*? In order to answer this question, Dirk Riehle [RIE96] gives us the following clue: “A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts”. Although the pattern concept was born almost twenty years before, in the Alexander’s book describing *patterns* extracted from buildings [ALE77], it was ten years later when Kent Beck and Ward Cunningham presented a paper using Alexander’s ideas on *software buildings* [BEC87]. Finally, in the last years, the design patterns technique would become popular between the software engineers thanks to the *Gang of Four* book “Design Patterns – Elements of Reusable Object-Oriented Systems” [GOF94] and the *Gang of Five* book “Pattern-Oriented Software Architecture – A System of Patterns” [BUS96].

According to Buschmann, patterns can be organized using the following classification [BUS96]:

- An *architectural pattern* expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their relationships and includes rules and guidelines for organizing the relationships between them.
- A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
- *Idioms* are low-level patterns, specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using features of the given language.

Design patterns can be seen as a powerful tool to abstract key concepts behind a system. The main idea about design pattern is that most of the systems are based on the same *templates*, used again and again. Design patterns support the analysis of these problems (templates at design level) in order to describe far better the system, easing the reusability and maintenance of them.

Most of the design patterns examined in the real world are considered pattern languages. A *pattern language* is a collection of patterns that refer to each other in such a way that users can use the patterns to build software systems in a similar way as they use natural language to create sentences, paragraphs and books ([ALE79][COP97]). Understanding the relationships between patterns involved in a pattern language eases the understanding of the system and the integration with other systems.

2.2 Batch Jobs and Pipelines in Software Design

We can find an extensive amount of literature describing architectures and designs that implement batch jobs systems and pipeline architectures [SIL98][TAN97][PAT98][SHA96][BUS95]. A pipeline is different from a batch job in that the pipeline execution flow is incremental whereas the batch job execution flow is sequential. As result, pipeline designs get better performance in parallel systems; such systems allow a continuous feed of the pipeline, and as soon as the pipeline element finishes its processing step, it can accept the next processing step, because the elements of the pipeline are independent of each other. However, batch job elements must wait until the end of the ongoing execution flow before accepting the next processing step.

```
main (... ,format,... ,load,...) {
while (!end) {
...
if (format==HTML) toHTML();
else if (format==ASCII) toASCII();
...
if (load==LOCAL) localFile();
else if (load==REMOTE) remoteFile();
...
} endwhile
}
```

Figure 1: the monolithic block (pseudo code)

In the beginning, batch jobs were implemented in (large) monolithic blocks of code. All the chained jobs were coded together in the same main function, or they were uncoupled from the main function using separate functions, but leaving in the main function the control logic of the process, using conditional statements (see Figure 1). Nevertheless, the resulting block was excessively coupled and lacked of coherency. These factors made these systems very hard to evolve, maintain and reuse.

The first intent to formally describe a pipeline by means of a pattern was done by Mary Shaw and her Pipeline architectural pattern [SHA96]. This pattern relies on being able to decompose the problem into a set of processing steps (filters), which transforms one or more input streams incrementally to one or more output streams (see Figure 2). This pattern is often mentioned by UNIX programmers, who use it for prototyping; however, the behaviour of the UNIX filters differs from the “pure” filters because they often consume the entire input stream before producing output.

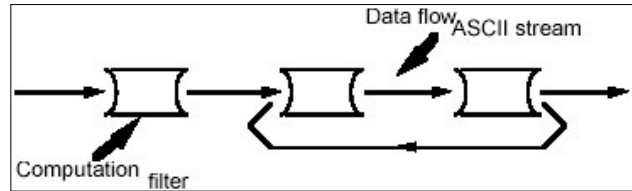


Figure 2: the Pipeline architecture

The Buschmann’s Filter and Pipe design pattern “... provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data are passed through pipes between adjacent filters.” [BUS95]. The same kind of input data can be streamed from various sources and the output data can be forwarded to different targets.

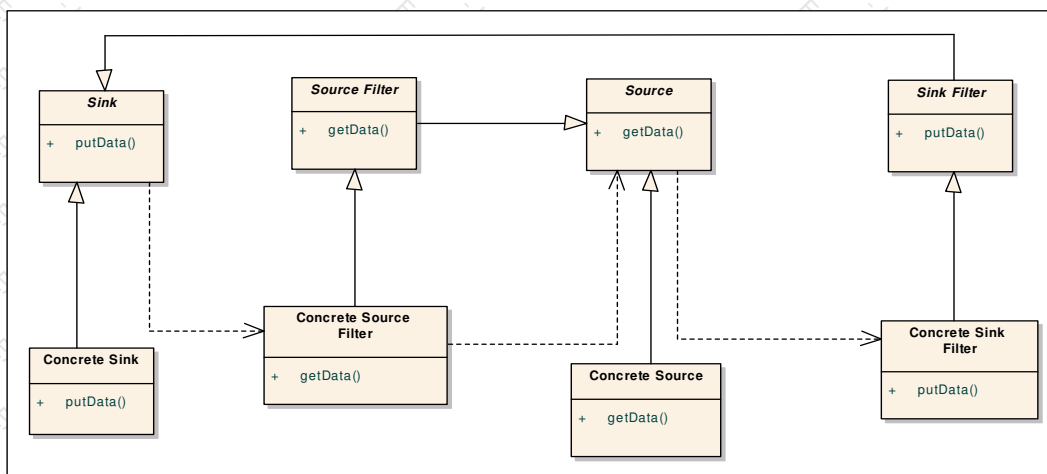


Figure 3: the Filter and Pipe pattern

The structure of a Filter and Pipe pattern consists of four elements: filter, data source, data sink and pipe (see Figure 3). The filters are classified in *active filters* and *passive filters*. The active filter assumes the filter is always pulling / pushing data from the pipelines (likely, using an infinite loop in a separate thread). The passive filters are activated either by a subsequent pipeline element pulling output from the filter, or a previous pipeline element pushing new input data to the filter (push versus pull models). A *data source* is the component that provides the input data to the system. A *data sink* is the component that gathers data at the end of the pipeline. Finally, the *pipes* are the connectors used by the system to communicate its elements.

The Filter pattern [GRA98] is similar to the Filter and Pipe design pattern. The Filter pattern assumes a simpler pipe configuration: whilst in the Filter and Pipe pattern the pipes are considered first-class design, in the Filter pattern the pipes are established as method calls to the following filter. Also the Filter pattern allows the building of complex filter structures using cascading.

The Composite Filter pattern [YAC01] extends the Filter design pattern, using two additional patterns: the Composite pattern and the Strategy pattern (both of them from [GOF94]). This pattern provides a common interface to design any combination of filters (using either composition or sequence of simple filters).

Additionally, the template architecture (as shown in Figure 4) can be used too. In this case, an abstract class contains the process sequence model (the template), and we redefine this template in each concrete class. In this case, the problem consists ending up with an extensive hierarchy of template objects (depending on the amount of filters – processing steps).

Finally, the Sequence pattern (Figure 5) gathers the best features of the last architectures and designs. The amount of filters doesn't interfere with the system because each pipeline is configured at runtime, avoiding implementing each filter variation in a template at compile time. The system stays open to accept new filters, regardless of the interface used by them. Finally, the system offers a simple interface that allows the construction of complex filters.

<pre> abstract template { DoProcess() { FromFile(); Validate(); Process(); Calculate(); Format(); ToFile(); } abstract FromFile(); abstract Validate(); abstract Process(); abstract Calculate(); abstract Format(); abstract ToFile(); } </pre>	<pre> concreteA:template{ ... FromFile () { localFile(); } ... Format () { toHTML(); } } </pre>	<pre> concreteB:template{ ... FromFile () { localFile(); } ... Format () { toASCII(); } } </pre>	<pre> concreteC:template{ ... FromFile () { remoteFile(); } ... Format () { toHTML(); } } </pre>	<pre> concreteD:template{ ... FromFile () { remoteFile(); } ... Format () { toASCII(); } } </pre>
--	---	--	--	---

Figure 4: the Template architecture (pseudo code)

```

facade {
static formatter format;
static loader load;
doLoadRemote (obj) { ... load.Remote (host, path); ... }
doLoadLocal (obj) { ... load.Local (path); ... }
doFormatHTML (obj) { ... format.toHTML (text); ... }
doFormatASCII (obj) { ... format.toASCII (text); ... }
}

client () {
director.replace (Format, facade.doFormatHTML);
director.execute (context);
director.replace (FromFile, facade.doLoadRemote);
director.execute (context);
}
        
```

Figure 5: the Sequence pattern (pseudo code)

3. The Sequence design pattern

Patterns are generally described using *templates* that provide uniform and consistent representation style. The template used here to describe the Sequence design pattern is based on the format proposed in [GOF94], and widely used by the software design pattern community.

3.1 Intent

To provide a sequential execution of services, enhancing the reusability (*plug & play architecture*) and the flexibility (*create once, configure and run multiple times*) of the system.

3.2 Also known as

Pipe and Filter, Filter, Composite Filter

3.3 Problem

We need to build a filtering system (a program that processes its input data through several services), which allows a large number of variations in the assembling of the filters. Each filter is available as an independent software component, however the source code of these components is not available. The sequence in which to apply the filters is known at runtime, and probably once configured, it will repeat the same process filtering sequence for all the input data, until the end of the application.

3.4 Forces

Ability to create complex choreographies. When the service requires large processing steps, it is preferable to split the service in a sequence of small processing steps, to make it easier to re-use and maintain.

Ability to cooperate with any service. The system shall be able to assure the integration of the services, because each service can be implemented by different components, using different method signatures. Furthermore, all the legacy components can be preserved, accepting upcoming new components at the same time

Ability to encourage the re-use of the actual components and available resources. For example, in the case of services with low coupling level, several choreographies could use the same service.

Ability to interact with its context. The system shall be prepared to be integrated in bigger application systems, and for this reason, it shall be able to interact with the rest of the system: accept input data from the application context, and return an output value.

3.5 Solution Structure

The UML class diagram in Figure 6 illustrates the structure of the Sequence pattern. The diagram can be divided into three layers: the *structural* layer involves the Client and theCommand subclasses; theFacade subclasses are at the *behavioural* layer; and finally, the *functional* layer involves the subsystem containing the services used by the system. The resulting system resembles a 3-tier architecture: the front end (structural layer), the back end (behavioural layer) and the database (functional layer).

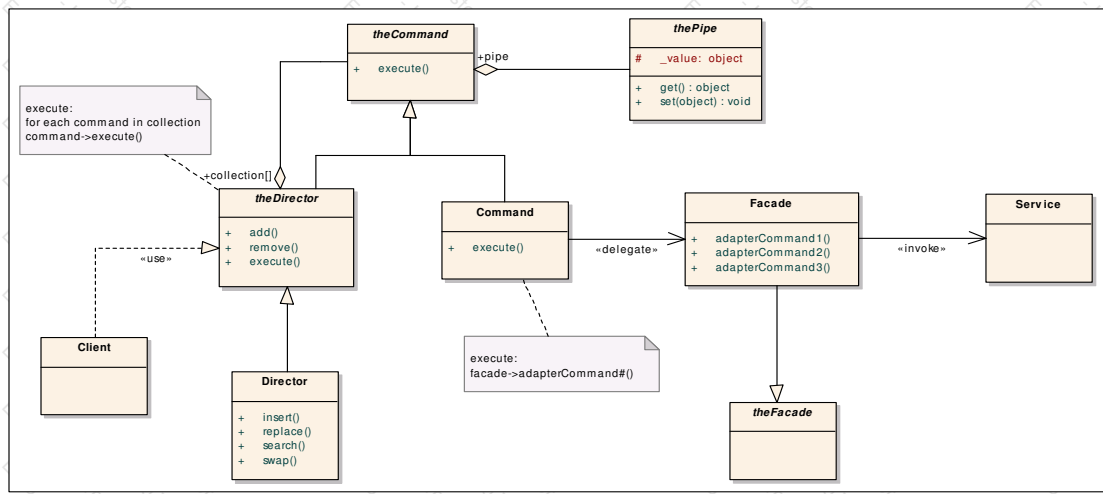


Figure 6: the Sequence pattern (UML class diagram)

3.6 Solution Participants

Following the UML class diagram illustrated in Figure 6, the Sequence pattern can be decomposed into several elements. In this section, we will examine the different elements of the system, and its relationships with some well-known patterns.

Client is responsible for setting up the pipeline (i.e. creating its elements and making the connections). Acts as a high level environmental interface, providing the input data and querying the output.

Service represents a (complex) subsystem of services. All the requests forwarded to the system, eventually are routed to some component belonging to Service.

thePipe encapsulates a shared data location, and implements a communication protocol for the elements contained inside the system. Thus, it serves to store data accessible by the system, and to define an interface to manipulate this data. It is the provider of the application data for the system, and as well the provider of the system output for the application. thePipe is a variation of the Shared Repository pattern (see Figure 7), an architectural pattern that describes a communication “technique” between software components using a shared location [LAL98].

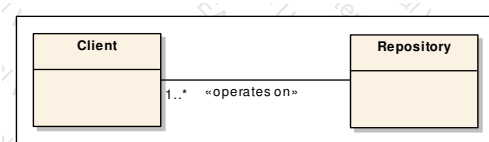


Figure 7: the Shared Repository pattern

theFacade class provides an adapter method for a service request. theFacade introduces an additional level of indirection between Commands and the services. theFacade is based on the Facade design pattern (see Figure 3) described in [GOF94]. This pattern provides a simplified interface to access complex subsystems, but without hiding these subsystems.

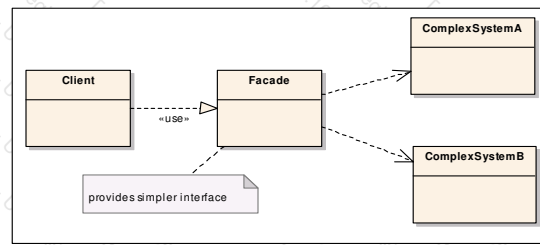
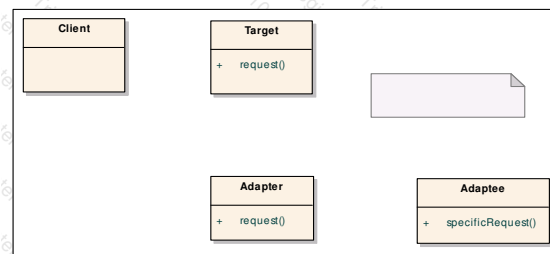


Figure 8: the Facade pattern

Facade is a subclass of the Facade. Usually this will contain adapter methods (based on the Adapter pattern [GOF94] illustrated in Figure 9); these methods process service requests issued by a Command object.



theCommand provides the execution behaviour to the pipeline elements. Following the Command pattern (see Figure 10) nomenclature used in [GOF94], the Client sets up the Receiver (usually an adapter method on a Facade object) of the Command; Command acts like a ConcreteCommand, and it forwards all the service requests towards Receiver; finally the Director acts like an Invoker (invoke the execute method on the Command objects).

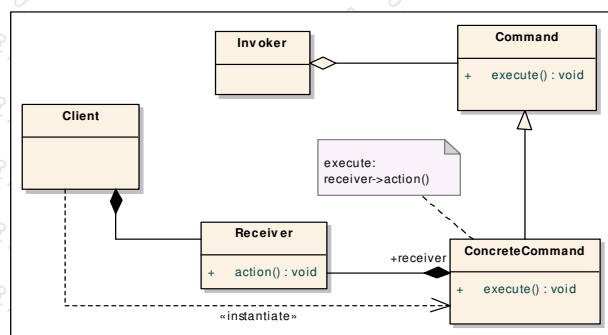


Figure 10: the Command pattern

Command represents a concrete object implementing the Command abstract class. In the context of a pipeline, a Command could be seen like a *placeholder*. For example, a placeholder named “output format” would represent the placeholder for the format of the output data, and it would fit commands like “toHTML”, “toXML” or “toASCII”.

ERROR: rangecheck
OFFENDING COMMAND: .buildcmap

STACK:

-dictionary-
/WinCharSetFFFF-V2TT786613C3t
/CMap
-dictionary-
/WinCharSetFFFF-V2TT786613C3t