Notes on Software Patterns: Chapters 1-2 Introduction and Pipes

H. Conrad Cunningham

16 April 2022

Contents

Copyright (C) 2022, H. Conrad Cunningham
Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla

1 Introduction to Patterns

1.1 Chapter Introduction

The goal of this chapter is to introduce the basic concepts and terminology of software patterns as used in software architecture, software engineering, and programming.

The chapter approaches patterns mostly from the perspective of object-oriented programming using languages such as Java and Scala. Classic works on software "design patterns" include the Gamma et al. (i.e., the "Gang of Four") [3] and Buschmann et al. (i.e., "Siemens") [2] books. In these notes, we primarily use the terminology of Buschmann et al. [2].

Similar approaches may be used in functional languages such as Haskell, but often functional languages will use first-class and higher-order functions to express the patterns.

The accompanying set of HTML slides (not fully updated in 2022) is the following:

- Introduction to Patterns (HTML)

1.2 What is a Pattern?

When experts need to solve a problem, they seldom invent a totally new solution. More often they will recall a similar problem they have solved previously and reuse the essential aspects of the old solution to solve the new problem. They tend to think in problem-solution pairs.

Identifying the essential aspects of specific problem-solution pairs leads to descriptions of problem-solving *patterns* that can be reused.

The concept of a pattern as used in software architecture is borrowed from the field of (building) architecture, in particular from the writings of architect Christopher Alexander [1]. Buschmann et al. [2] defines pattern as follows in the context of software architecture:

> "A *pattern for software architecture* describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate."

Where software architecture is concerned, the concept of a pattern described here is essentially the same concept as an *architectural style* or *architectural idiom* in Shaw and Garlan [9].

In general, patterns have the following characteristics [2]:

- A pattern describes a solution to a recurring problem that arises in specific design situations.

- Patterns are not invented; they are distilled from practical experience.
- Patterns describe a group of components (e.g., classes or objects), how the components interact, and the responsibilities of each component. That is, they are higher level abstractions than classes or objects.
- Patterns provide a vocabulary for communication among designers. The choice of a name for a pattern is very important.
- Patterns help document the architectural vision of a design. If the vision is clearly understood, it will less likely be violated when the system is modified.
- Patterns provide a conceptual skeleton for a solution to a design problem and, hence, encourage the construction of software with well-defined properties.
- Patterns are building blocks for the construction of more complex designs.
- Patterns help designers manage the complexity of the software. When a recurring pattern is identified, the corresponding general solution can be implemented productively to provide a reliable software system.

1.3 Descriptions of Patterns

Various authors use different formats (i.e., "languages") for describing patterns. Typically a pattern will be described with a schema that includes at least the following three elements [2]:

1. Context
2. Problem
3. Solution

1.3.1 Context

The *Context* element describes the circumstances in which the problem arises.

1.3.2 Problem

The *Problem* element describes the specific problem that arises repeatedly in the context.

In particular, the description describes the set of *forces* repeatedly arising in the context. A force is some aspect of the problem that must be considered when attempting a solution. Example types of forces include:

- requirements the solution must satisfy (e.g., efficiency)
- constraints that must be considered (e.g., use of a certain algorithm or protocol)

- desirable properties of a solution (e.g., easy to modify)

Forces may complementary (i.e., can be achieved simultaneously) or contradictory (i.e., can only be balanced).

1.3.3 Solution

The *Solution* section describes a proven solution to the problem.

The solution specifies a configuration of elements to balance the forces associated with the problem.

- A pattern describes the static structure of the configuration, identifying the components and the connectors (i.e., the relationships among the components).
- A pattern also describes the dynamic runtime behavior of the configuration, identifying the control structure of the components and connectors.

1.3.4 Aside: Extending pattern elements

In a helpful tutorial on pattern writing, Wellhausen and Fiesser [13] separate out the Forces and Consequence as separate element and state that the following elements must be present in the given order:

- *Pattern Name* gives an evocative name for the pattern.
- *Context* describes the circumstances in which the problem occurs.
- *Problem* describes the specific problem to be solved.
- *Forces* describe why the problem is difficult to solve, giving different considerations that must be balanced to solve the problem.
- *Solution* describes how the solution to the problem works at an appropriate level of detail.
- *Consequences* describe what happens when a software designer applies the pattern. It gives both the possible *benefits* and possible *liabilities* of using the pattern.

All of the above elements except Consequences are also prescribed by the Mandatory Elements Present pattern from Meszaros and Doble's *A Pattern Language for Pattern Writing* [4]. Their Optional Elements When Helpful pattern suggests use of optional elements such as Examples, Code Samples, and Rationale. Following their Readable References to Patterns pattern, we refer to a pattern using its name in small capital letters.

1.4 Categories of Patterns

Patterns can be grouped into three categories according to their level of abstraction [2]:

1. Architectural patterns
2. Design patterns
3. Idioms

1.4.1 Architectural patterns

Buschmann et al. [2] defines an architectural pattern as follows:

> "An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them."

In early work, Shaw and Garlan [9] used the term *architectural style* instead of architectural pattern.

An architectural pattern is a high-level abstraction. The choice of the architectural pattern to be used is a fundamental design decision in the development of a software system. It determines the system-wide structure and constrains the design choices available for the various subsystems. It is, in general, independent of the implementation language used.

Examples of architectural patterns include the following.

- The PIPES AND FILTERS (or PIPELINE) pattern [2:53,8,9,15] defines a structure for systems in which an independent set of computations—called *filters*—transform one or more input streams—passing along *pipes*—incrementally to create one or more output streams.

  See the separate notes on the Pipes and Filters pattern or the set of Powerpoint slides for more a more detailed discussion of this pattern.

  In the UNIX operating system [6], for instance, a filter is a program that reads a stream of bytes from its standard input and writes a transformed stream to its standard output. These programs can be connected together (e.g., in the interactive shell program) with the output of one filter becoming the input of the next filter in the sequence via the pipe mechanism. Larger systems can thus be constructed from simple components that otherwise operate independently of one another.

- The LAYERED SYSTEMS [2:31,8,9] pattern organizes a system hierarchically with "each layer providing service to the layer above it and serving as a client for the layer below" [9].

  Communication protocols, operating systems, virtual machines, and application programming interfaces (APIs) are often designed and implemented as layered systems [2].

- The BLACKBOARD pattern [2:71,8,9,16] defines a structure in which "a collection of independent programs"—the *knowledge sources*—"work cooperatively on a common data structure"—the *blackboard* [2].

  A BLACKBOARD pattern is one of two subcategories of the REPOSITORY pattern [8,9]. In a BLACKBOARD system, the state of the blackboard (i.e., the repository) triggers the activity of the knowledge sources (i.e., the independent programs). Such a structure is often useful in artificial intelligence applications.

  The other subcategory of the Repository pattern is for situations in which the types of the inputs trigger the independent programs. In such systems, the repository can be a traditional database that simply organizes the data collected.

- The MODEL-VIEW-CONTROLLER [2:125,14] pattern defines a structure for interactive user interface programs (such as GUIs or Web applications). It decomposes the application into three types of components [2]:

  - The Model contains the application's core functionality (i.e., its logic) and data (e.g., accesses the application's database).
  - A View represents the data contained in the Model for display to a user.
  - A Controller handles user inputs associated with a View and interacts with the View and Model to carry out the user's commands.

  An application has a single Model and one or more Views. Each View has a unique Controller to enable the user to manipulate both that View and the Model.

The Buschmann et al. book [2], Qian et al. book [5], and Shaw paper [9] elaborate on these and other architectural patterns. (**This link** is to a local copy of a preprint of the Shaw paper.)

1.4.2 Design patterns

Buschmann et al. [2] defines design pattern as follows:

> "A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context."

A design pattern is a mid-level abstraction. The choice of a design pattern does not affect the fundamental structure of the software system, but it does affect the structure of a subsystem. Like the architectural pattern, the design pattern tends to be independent of the implementation language to be used.

A design pattern might not, however, be independent of the programming paradigm. A pattern that is meaningful for a statically typed, object-oriented language without first-class functions may not be as meaningful for a dynamically typed language or for a functional language with first-class functions.

Examples of design patterns include the following.

- The ADAPTER (or WRAPPER) pattern [3:139,10,17] converts the interface of one existing type of object to have the same interface as a different existing type of object.

  For example, suppose a Java program has a base class `Stack` to represent stack data structures. Also suppose the program has instances of the builtin class `Vector` that we wish to use as `Stack` objects. We can implement a new subclass of `Stack`, say `VectorAsStack`, that wraps a `Vector` object and implements the operations of `Stack` by delegating them appropriately to the `Vector` object but hides the non-stack features of the `Vector`.

  As another example, consider Schmid's paper "Creating Applications from Components: A Manufacturing Framework Design" [7]. The application framework presented in the paper uses the ADAPTER pattern. It adapts the portal robot machine class so that its instances can be used in place of transport service class instances.

- The ITERATOR pattern [3:257,11,18] defines a way to access the elements of a container (data structure) sequentially without exposing the container's representation.

  Iterator objects for standard collections are now common in most programming language libraries. Programmers can also implement iterators for their own custom collections.

- The STRATEGY (or POLICY) pattern [3:315,12,19] defines an interface to a family of related algorithms so that any algorithm in the family can be dynamically substituted for another at runtime.

  Suppose we have an container class `C` whose elements we wish to be able to sort using one of several different sorting algorithms selected at runtime. To apply the STRATEGY pattern, we design `C` so that it delegates the sorting of its elements to a method `sort()` on an object of type `Sorter` stored its instance variable `mySorter`. We then implement a different subclass of `Sorter` for each soring algorithm of interest. We can dynamically change `C`'s sorting behavior by assigning a different `Sorter` object to `mySorter`.

  We also see this pattern used in the Schmid article [7]. Schmid's third transformation involves breaking up the application logic class `ProcessingControl` into several subclasses of a new `ProcessingStrategy` class. The specific processing strategy can then be selected dynamically based on the specific part-processing task.

  See the Powerpoint slides on the STRATEGY pattern for more information.

1.4.3 Idioms

Buschann et al. [2] defines idiom as follows:

> "An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language."

An idiom is a low-level abstraction. It is usually a language-specific pattern that deals with some aspects of both design and implementation.

In some sense, use of a consistent program coding and formatting style can be considered an idiom for the language being used. Such a style would provide guidelines for naming variables, laying out declarations, indenting control structures, ordering the features of a class, determining how values are returned, and so forth. A good style that is used consistently makes a program easier to understand than otherwise would be the case.

In Java, the language-specific iterator defined to implement the `Iterator` interface can be considered an idiom. It is a language-specific instance of the more general ITERATOR design pattern.

Another example of an idiom is the use of the COUNTED POINTER (or COUNTED BODY or REFERENCE COUNTING) technique for storage management of shared objects in C++. In this idiom, we control access to a shared object through two classes, a *Body* (representation) class and a *Handle* (access) class.

An object of the *Body* class holds the shared object and a count of the number of references to the object.

An object of a *Handle* class holds a direct reference to a body object; all other parts of the program must access the body indirectly through handle class methods. The handle methods can increment the reference count when a new reference is created and decrement the count when a reference is freed. When a reference count goes to zero, the shared object and its body can be deleted. Often the programmer using this pattern will want to override the `operator->` of the handle class to give more transparent access to the shared object.

We can use a variant of the COUNTED POINTER idiom to implement a "copy on write" mechanism. That is, the body is shared as long as only "read" access is needed, but a copy is created whenever one of the holders makes a change to the state of the object.

1.5 What Next?

TODO

1.6 Acknowledgements

I wrote the first version of these notes for my Spring 1998 graduate Special Topics in Software Architecture class. I based the notes, in part, on:

- Chapter 1 of the “Siemens” book [2]
- Chapter 2 of Shaw and Garlan [9]

I revised the notes somewhat for related courses in 1999, 2000, 2001, 2002, and 2004. Also, in 2004 I revised the notes, created slides, and included them as a part of the materials supported by a grant from the Acxiom Corporation titled “The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)”. My ALSACE research team included PhD students Yi Liu and Pallavi Tadepalli and MS students Mingxian Fu and “Melody” Hui Xiong.

In Spring 2017, I adapted the earlier notes to use Pandoc-flavored Markdown. In Spring 2018 I revised the notes slightly to fit in with the other documents for the CSci 658 course.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on possible textbooks based on the course materials I had developed during my three decades as a faculty member. In January 2022, I began refining the existing content, integrating separately developed materials together, reformatting the documents, constructing a unified bibliography (e.g., using citeproc), and improving my build workflow and use of Pandoc.

In 2022, I also added the aside based on Wellhausen and Fiesser’s tutorial and expanded the discussion of the example patterns.

I maintain this chapter as text in Pandoc’s dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed

1.7 Terms and Concepts

TODO: Update

Pattern (for software architecture), architectural pattern (or architectural style), design pattern, idiom. Pattern Context, Problem, and Solution. Pattern Name, Forces, and Consequences. Pipes and Filters, Layers, Blackboard, and Model-View-Controller architectural patterns. Adapter, Iterator, and Strategy design patterns. Counted Pointer (or Reference Counting, Handle-Body) idiom.

2 Pipes and Filters Architectural Pattern

2.1 Chapter Introduction

The goal of this chapter is to present the PIPES AND FILTER architectural design pattern. This presentation follows the presentation in Buschmann et al. [2].

The accompanying set of Powerpoint slides (not fully updated in 2022) is the following:

- Pipe and Filters Architectural Pattern (Powerpoint)

2.2 Definition

Buschmann et al. [2] defines the PIPES AND FILTERS architectural pattern as follows:

> "The *Pipes and Filters* architectural 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. Recombining filters allows you to build families of related filters."

2.3 Context

The context consists of programs that must process streams of data.

2.4 Problem

Suppose we need to build a system to solve a problem:

- that must be built by several developers
- that decomposes naturally into several independent processing steps
- for which the requirements are likely to change

The design of the components and their interconnections must consider the following forces [2]:

- It should be possible to enhance the system by substituting new filters for existing ones or by recombining the steps into a different communication structure.
- Components implementing small processing steps are easier to reuse than components implementing large steps.
- If two steps are not adjacent, then they share no information.
- Different sources of input data exist.
- It should be possible to display or store the final results of the computation in various ways.

- If the user stores intermediate results in files, then the likelihood of errors increases and the file system may become cluttered with junk.
- Parallel execution of the steps should be possible.

2.5 Solution

The Solution involves the following steps:

- Divide the task into a sequence of processing steps.
- Let each step be implemented by a filter program that consumes from its input and produces data on its output incrementally.
- Connect the output of one step as the input to the succeeding step by means of a pipe.
- Enable the filters to execute concurrently.
- Connect the input to the sequence to some data source, such as a file.
- Connect the output of the sequence to some data sink, such as a file or display device.

2.6 Structure

The *filters* are the processing units of the pipeline. A filter may enrich, refine, or transform its input data [2].

- It may enrich the data by computing new information from the input data and adding it to the output data stream.
- It may refine the data by concentrating or extracting information from the input data stream and passing only that information to the output stream.
- It may transform the input data to a new form before passing it to the output stream.
- It may, of course, do some combination of enrichment, refinement, and transformation.

A filter may be active (the more common case) or passive.

- An *active* filter runs as a separate process or thread; it actively *pulls* data from the input data stream and *pushes* the transformed data onto the output data stream.
- A *passive* filter is activated by either being called:
  - as a function, a *pull* of the output from the filter
  - as a procedure, a *push* of output data into the filter

The *pipes* are the connectors—between a data source and the first filter, between filters, and between the last filter and a data sink. As needed, a pipe synchronizes the active elements that it connects together.

A *data source* is an entity (e.g., a file or input device) that provides the input data to the system. It may either actively push data down the pipeline or passively supply data when requested, depending upon the situation.

A *data sink* is an entity that gathers data at the end of a pipeline. It may either actively pull data from the last filter element or it may passively respond when requested by the last filter element.

(See the Class-Responsibility-Collaborator (CRC) cards for these elements on page 56 of Buschmann et al. [2])

TODO: Consider whether the above paragraph is needed or can be replaces without depending upon the book completely.

2.7 Implementation

Implementation of the pipes-and-filters architecture is usually not difficult. It often includes the following steps [2]:

1. *Divide the functionality of the problem into a sequence of processing steps.*

   Each step should only depend upon the outputs of the previous step in the sequence. The steps will become the filters in the system.

   In dividing up the functionality, be sure to consider variations or later changes that might be needed—a reordering of the steps or substitution of one processing step for another.

2. *Define the type and format of the data to be passed along each pipe.*

   For example, Unix pipes carry an unstructured sequence of bytes. However, many Unix filters read and write streams of ASCII characters that are structured into lines (with the newline character as the line terminator).

   Another important formatting issue is how the end of the input is marked. A filter might rely upon a system end-of-input condition or it may need to implement their own “sentinel” data value to mark the end.

3. *Determine how to implement each pipe connection.*

   For example, a pipe connecting active filters might be implemented with operating system or programming language runtime facility such as a message queue, a Unix-style pipe, or a synchronized-access bounded buffer.

   A pipe connecting to a passive filter might be implemented as a direct call of the adjacent filter: a push connection as a call of the downstream filter as a procedure or a pull connection as a call of the upstream filter as a function.

4. *Design and implement the filters.*

   The design of a filter is based on the nature of the task to be performed and the natures of the pipes to which it can be connected.

   - An active filter needs to run with its own *thread of control.* It might run as as a “heavyweight” operating system process (i.e., having its own address space) or as a “lightweight” thread (i.e., sharing an address space with other threads).

   - A passive filter does not require a separate thread of control (although it could be implemented with a separate thread).

   The selection of the *size of the buffer* inside a pipe is an important performance tradeoff. Large buffers may use up much available memory but likely will involve less synchronization and context-switching overhead. Small buffers conserve memory at the cost of increased overhead.

   To make filters flexible and, hence, increase their potential reusability, they often will need different *processing options* that can be set when they are initiated. For example, Unix filters often take command line parameters, access environment variables, or read initialization files.

5. *Design for robust handling of errors.*

   Error handling is difficult in a pipes-and-filters system since there is no global state and often multiple asynchronous threads of execution. At the least, a pipes-and-filters system needs mechanisms for detecting and reporting errors. An error should not result in incorrect output or other damage to the data.

   For example, a Unix program can use the `stderr` channel to report errors to its environment.

   More sophisticated pipes-and-filters systems should seek to recover from errors. For example, the system might discard bad input and resynchronize at some well-defined point later in the input data. Alternatively, the system might back up the input to some well-defined point and restart the processing, perhaps using a different processing method for the bad data.

6. *Configure the pipes-and-filters system and initiate the processing.*

   One approach is to use a standardized main program to create, connect, and initiate the needed pipe and filter elements of the pipeline.

   Another approach is to use an end-user tool, such as a command shell or a visual pipeline editor, to create, connect, and initiate the needed pipe and filter elements of the pipeline.

2.8 Example

An example pipes-and-filter system might be a retargetable compiler for a programming language. The system might consist of a pipeline of processing elements similar to the following:

1. A *source* element reads the program text (i.e., source code) from a file (or perhaps a sequence of files) as a stream of characters.

2. A *lexical analyzer* converts the stream of characters into a stream of lexical tokens for the language—keywords, identifier symbols, operator symbols, etc.

3. A *parser* recognizes a sequence of tokens that conforms to the language grammar and translates the sequence to an abstract syntax tree.

4. A *"semantic" analyzer* reads the abstract syntax tree and writes an appropriately augmented abstract syntax tree.

   Note: This element handles context-sensitive syntactic issues such as type checking and type conversion in expressions.

5. A *global optimizer* (usually optionally invoked) reads an augmented syntax tree and outputs one that is equivalent but corresponds to program that is more efficient in space and time resource usage.

   Note: A global optimizer may transform the program by operations such as factoring out common subexpressions and moving statements outside of loops.

6. An *intermediate code generator* translates the augmented syntax tree to a sequence of instructions for a virtual machine.

7. A *local optimizer* converts the sequence of intermediate code (i.e., virtual machine) instructions into a more efficient sequence.

   Note: A local optimizer may transform the program by removing unneeded loads and stores of data.

8. A *backend code generator* translates the sequence of virtual machine instructions into a sequence of instructions for some real machine platform (i.e., for some particular hardware processor augmented by operating system calls and a runtime library).

9. If the previous step generated symbolic assembly code, then an *assembler* is needed to translate the sequence of symbolic instructions into a relocatable binary module.

10. If the previous steps of the pipeline generated a sequence of separate binary modules, then a *linker* might be needed to bind the separate modules with library modules to form a single executable (i.e., object code) module.

11. A *sink* element outputs the resulting binary module into a file.

The pipeline can be reconfigured to support a number of different variations:

- If source code preprocessing is to be supported (e.g., as in C), then a *preprocessor* filter (or filters) can be inserted in front of the lexical analyzer.
- If the language is to be interpreted rather than translated into object code, then the backend code generator (and all components after it in the pipeline) can be replaced by an *interpreter* that implements the virtual machine.
- If the compiler is to be retargeted to a different platform, then a backend code generator (and assembler and linker) for the new platform can be substituted for the old one.
- If the compiler is to be modified to support a different language with the same lexical structure, then only the parser, semantic analyzer, global optimizer, and intermediate code generator need to be replaced.

  Note: If the parser is driven by tables that describe the grammar, then it may be possible to use the same parser with a different table.
- If a load-and-go compiler is desired, the file-output sink can be replaced by a *loader* that loads the executable module into an address space in the computer's main memory and starts the module executing.

Of course, a pure active-filters system as described above for a compiler may not be very efficient or convenient.

- Sometimes a system of filters can be made more efficient by directly sharing a global state. Otherwise the global information must be encoded by one filter, passed along a pipe to an adjacent filter, decoded by that filter, and so forth on downstream.

  In the compiler pipeline, the symbol table is a key component of the global state that is constructed by the lexical analyzer and needed by the phases downstream through (at least) the intermediate code generator.
- Sometimes performance can be improved by combining adjacent active filters into one program and replacing the pipe by an upstream function call (a passive pull connection) or a downstream procedure call (a passive push connection).

  In the compiler pipeline, it may be useful to combine the phases from lexical analysis through intermediate code generation into one program because they share the symbol table. Performance can be further improved by having the parser directly call the lexical analyzer when the next token is needed.
- Although a piece of information may not be required at some step, the availability of that information may be useful. For example, the symbol table information is not usually required during backend code generation,

interpretation, or execution. However, some of the symbol table information, such as variable and procedure names, may be useful in generation of error messages and execution traces or for use by a runtime debugging tools.

2.9 Variants

So far we have focused on single-input single-output filters. A generalization of the pipes-and-filters pattern allows filters with multiple input and/or multiple output pipes to be connected in any directed graph structure.

In general, such dataflow systems are difficult to design so that they compute the desired result and terminate cleanly. However, if we restrict ourselves to directed acyclic graph structures, the problem is considerably simplified.

In the UNIX operating system shell, the `tee` filter provides a mechanism to split a stream into two streams, named pipes provide mechanisms for constructing network connections, and filters with multiple input files/streams provide mechanisms for joining two streams.

TODO: Bring the following up to date?

Consider the following UNIX shell commands. On a Solaris "Unix" machine (late 1990's), this sequence sets up a pipe to build a sorted list of all words that occur more than once in a file:

```
# create two named pipes
mknod pipeA p
mknod pipeB p
# set up side chain computation (running in the background)
cat pipeA >pipeB &
# set up main pipeline computation
cat filename | tr -cs "[:alpha:]" "[\n*256]" \
             | tr "[:upper:]" "[:lower:]" | sort | tee pipeA | uniq \
             | comm -13 - pipeB | uniq
```

- The `mknod` commands set up two named pipes, `pipeA` and `pipeB`, for connecting to a "side chain" computation.
- The "side chain" command starts a `cat` program running in a *background* fork (note the `&`). The program takes its input from the pipe named `pipeA` and writes its output to the pipe named `pipeB`.
- The main pipeline uses a `cat` filter as a source for the stream. The next two stages use filter `tr` to translate each sequence of non-alphabetic characters to a single newline character and to map all uppercase characters to lowercase, respectively. The words are now in a standard form—in lowercase, one per line.

- The fourth stage of the main pipeline sorts the words into ascending order using the `sort` filter.
- After the sort, the main pipeline uses a `tee` filter to replicate the stream, sending one copy down the main pipeline and another copy onto the side chain via `pipeA`.
- The side chain simply copies the words from `pipeA` onto `pipeB`. Meanwhile the main pipeline uses the `uniq` filter to remove adjacent duplicate words.
- The main pipeline stream and the side chain stream are then joined by the `comm` filter. The `comm` filter takes two inputs, one from main pipeline's stream (note the `-` parameter) and another from `pipeB`.
- Invoking the `comm` filter with the `-13` option cause it to output the lines that appear in the second stream (i.e., `pipeB`) but not the first stream (i.e., the main pipeline). Thus, the output is an alphabetical list of words that appear more than once in the input file.
- The final stage, another `uniq` filter, removes duplicates from the final output.

2.10 Consequences

2.10.1 Benefits

The pipes-and-filters architectural pattern has the following benefits [2]:

- *Intermediate files unnecessary, but possible.* File system clutter is avoided and concurrent execution is made possible.
- *Flexibility by filter exchange.* It is easy to exchange one filter element for another with the same interfaces and functionality.
- *Flexibility by recombination.* It is not difficult to reconfigure a pipeline to include new filters or perhaps to use the same filters in a different sequence.
- *Reuse of filter elements.* The ease of filter recombination encourages filter reuse. Small, active filter elements are normally easy to reuse if the environment makes them easy to connect.
- *Rapid prototyping of pipelines.* Flexibility of exchange and recombination and ease of reuse enables the rapid creation of prototype systems.
- *Efficiency by parallel processing.* Since active filters run in separate processes or threads, pipes-and-filters systems can take advantage of a multiprocessor.

2.10.2 Liabilities

The pipes-and-filters architectural pattern has the following liabilities [2]:

- *Sharing state information is expensive or inflexible.* The information must be encoded, transmitted, and then decoded.
- *Efficiency gain by parallel processing is often an illusion.* The costs of data transfer, synchronization, and context switching may be high. Non-incremental filters, such as the Unix `sort`, can become the bottleneck of a system.
- *Data transformation overhead.* The use of a single data channel between filters often means that much transformation of data must occur, for example, translation of numbers between binary and character formats.
- *Error handling.* It is often difficult to detect errors in pipes-and-filters systems. Recovering from errors is even more difficult.

2.11 What Next?

TODO

2.12 Acknowledgements

I wrote the first version of these notes for my Spring 1998 graduate Special Topics in Software Architecture class. I based the notes, in part, on:

- The Pipes and Filters pattern description in Section 2.2 of the Buschmann et al. (i.e., "Siemens") book [2]
- The Pipes and Filters pattern description in Section 2.2 of Shaw and Garlan [9]

I revised the notes somewhat for related courses in 1999, 2000, 2001, 2002, and 2004. Also, in 2004 I revised the notes, created slides, and included them as a part of the materials supported by a grant from the Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)". My ALSACE research team included PhD students Yi Liu and Pallavi Tadepalli and MS students Mingxian Fu and "Melody" Hui Xiong.

In Spring 2017, I adapted the earlier notes to use Pandoc-flavored Markdown. In Spring 2018 I revised the notes slightly to fit in with the other documents for the CSci 658 course.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on possible textbooks based on the course materials I had developed during my three decades as a faculty member. In January 2022, I began refining the existing content, integrating separately developed materials together, reformatting the documents, constructing a unified bibliography (e.g., using citeproc), and improving my build workflow and use of Pandoc.

TODO: Modify acknowledgements appropriately for 2022+ updates.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed

2.13 Terms and Concepts

TODO

References

[1] Christopher Alexander. 1977. *A pattern language: Towns, buildings, construction.* Oxford University Press, Oxford, UK.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-oriented software architecture: A system of patterns.* Wiley, Hoboken, New Jersey, USA.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley, Boston, Massachusetts, USA.

[4] Gerard Meszaros and Jim Doble. 1998. A pattern language for pattern writing. In *Pattern languages of program design 3*, Addison-Wesley, Boston, Massachusetts, USA, 529–574.

[5] Kai Qian, Xiang Fu, Lixin Tao, Chong-Wei Xu, and Jorge L. Diaz-Herrera. 2010. Software architecture and design illuminated. Jones & Bartlett Learning, Burlington, Massachusetts, USA.

[6] Dennis M. Ritchie. 1984. The UNIX system: The evolution of the UNIX time-sharing system. *AT&T Bell Laboratories Technical Journal* 63, 8 (1984), 1577–1593.

[7] Hans Albrecht Schmid. 1996. Creating applications from components: A manufacturing framework design. *IEEE Software* 13, 6 (1996), 67–75.

[8] Mary Shaw. 1996. Some patterns for software architecture. In *Pattern languages of program design 2*, John M. Vlissides, James O. Coplien and Norman L. Kerth (eds.). Addison-Wesley, Boston, Massachusetts, USA, 255–270.

[9] Mary Shaw and David Garlan. 1996. *Software architecture: Perspectives on an emerging discipline.* Prentice-Hall, Englewood Cliffs, New Jersey, USA.

[10] Source Making. 2022. Adapter design pattern. Retrieved from https://sourcemaking.com/design_patterns/adapter

[11] Source Making. 2022. Iterator design pattern. Retrieved from https://sourcemaking.com/design_patterns/iterator

[12] Source Making. 2022. Strategy design pattern. Retrieved from https://sourcemaking.com/design_patterns/strategy

[13] Tim Wellhausen and Andreas Fiesser. 2011. How to write a pattern? A rough guide for first-time pattern authors. In *Proceedings of the 16th Europe an conference on pattern languages of programs* (EuroPLOP '11), Irsee, Germany, 1–9 (Article 5).

[14] Wikpedia: The Free Encyclopedia. 2022. Model-view-controller. Retrieved from https://en.wikipedia.org/wiki/Model-view-controller

[15] Wikpedia: The Free Encyclopedia. 2022. Pipeline (software). Retrieved from https://en.wikipedia.org/wiki/Pipeline_(software)

[16] Wikpedia: The Free Encyclopedia. 2022. Blackboard (design pattern). Retrieved from https://en.wikipedia.org/wiki/Blackboard_(design_pat tern)

[17] Wikpedia: The Free Encyclopedia. 2022. Adapter pattern. Retrieved from https://en.wikipedia.org/wiki/Adapter_pattern

[18] Wikpedia: The Free Encyclopedia. 2022. Iterator pattern. Retrieved from https://en.wikipedia.org/wiki/Iterator_pattern

[19] Wikpedia: The Free Encyclopedia. 2022. Strategy pattern. Retrieved from https://en.wikipedia.org/wiki/Strategy_pattern