SWE 264P | Distributed Software Architecture | Winter 2026
Lab 1: Pipe-and-Filter | Prof. Sam Malek

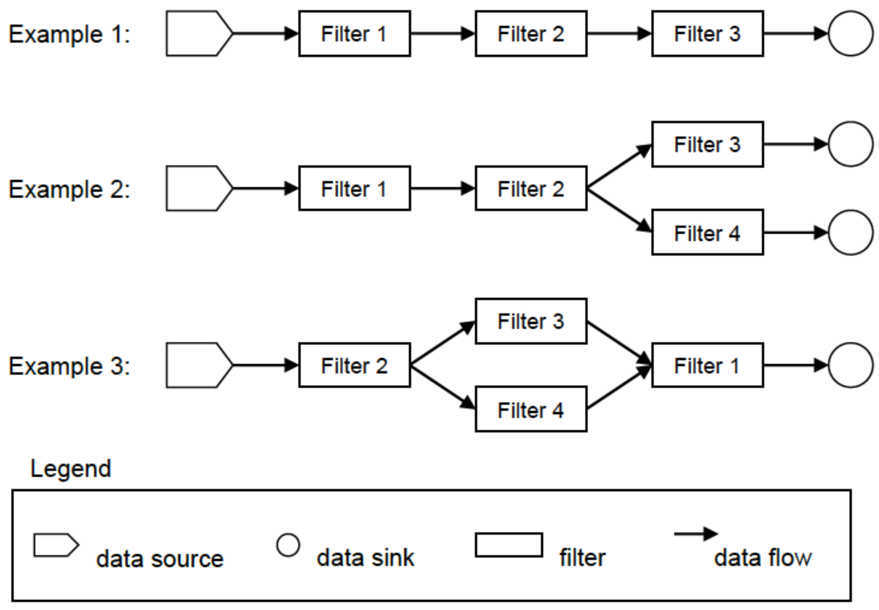
Problem Description

The objective of this assignment is to develop an appreciation of architectural patterns/styles and their impact on systemic properties. This assignment will use the pipe-and-filter architectural pattern as an exemplar. This assignment will be implementation-oriented, allowing you to experiment with a particular pipe-and-filter implementation strategy in order to gain a clearer understanding of the issues associated with carrying an architectural design through to code. Note that this is not a programming class, and so the emphasis of this assignment is on the architecture issues – **this cannot be stressed enough**. Simply re-writing the entire implementation will indicate a lack of understanding of the core architectural concepts presented in class.

For this assignment, you will be provided a working sample system that uses a (coding) framework supporting the pipe-and-filter paradigm. The application domain for this assignment is signal processing, as described below. Your task is to extend the existing framework to architect and build the systems specified in the requirements below. While you may discuss design decisions with your colleagues, the **lab must be done individually**.

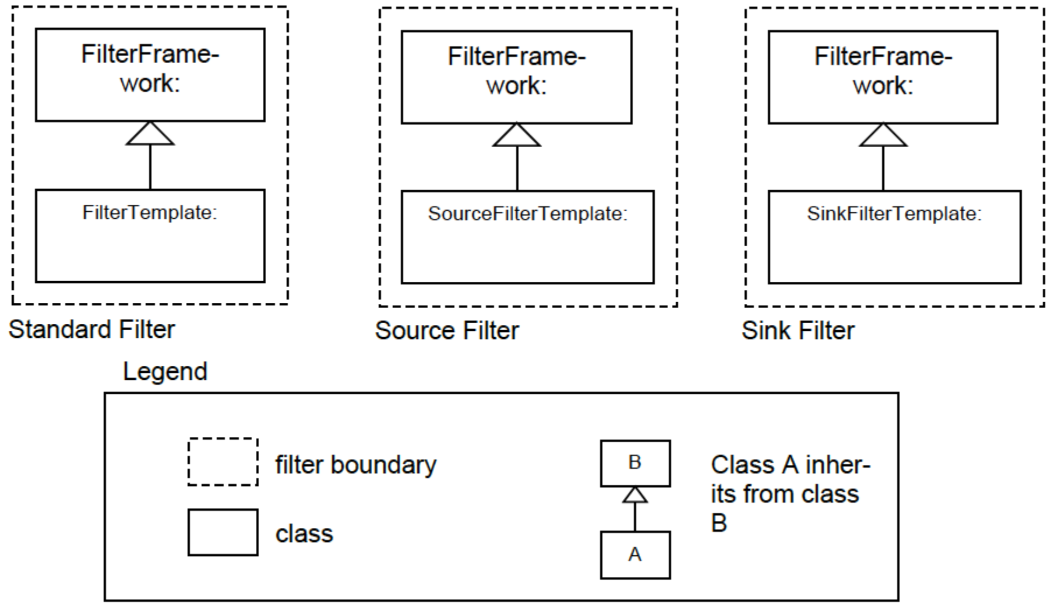
Business Context and Key Architectural Approaches

The principal stakeholder for this system is an organization that builds instrumentation systems. Instrumentation is a typical kind of signal processing application where streams of data are read, processed in a variety of ways, and displayed or stored for later use. A key part of modern instrumentation systems is the software that is used to process byte streams of data. The organization would like to create flexible software that can be reconfigured for a variety of applications and platforms (for our purposes, we can think of “platforms” as processors). For example, one application might be to support instrumentation for an automobile that would include data streams that originate with sensors and terminate in the cabin of the auto with a display of temperature, oil pressure, velocity, and so forth. Some subset of filters for this application might be used in aviation, space, or maritime applications. Another application might be in the lab reading streams of data from a file, processing the stream, and storing the data in a file. This would support the development and debugging of instrumentation systems. While it is critically important to support reconfiguration, the system must also process streams of data as quickly as possible. To meet these challenges, the architect has decided to design the system around a pipe-and-filter architectural pattern. From a dynamic perspective, systems would be structured as shown in the following examples.



The “data sources” in these systems are special filters that read data from sensors, files, or that generate data internally within the filter. All filter networks must start with a source. The “filters” shown in these examples are standard filters that read data from an upstream pipe, transform the data, and write data to a downstream pipe. The “data sinks” are special filters that read data from an upstream filter, but write data to a file or device of some kind. All filter networks must terminate with a sink that consumes the final data. Note that streams can be split and merged as shown in these examples.

The organization’s architect has developed a set of classes to facilitate the rapid development of filters and applications that can be quickly tested and deployed. These libraries have been provided to you. In addition, there are several examples that have been provided to illustrate the use of these classes. The class structure (static perspective) for filters is as follows:



The FilterFramework class is the base class for all filters. It contains methods for managing the connections to pipes, writing and reading data to and from pipes, and setting up the filters as separate threads. Three filter "templates" have been established to ease the work of creating source, sink, and standard filters in a consistent way. Each of these filter templates describes how to write code for the three basic types of filters. Note that the current framework does not support splitting or merging the data stream. A fourth template, called the "PlumberTemplate" shows how pipe-and-filter networks can be set up from the filters created by developers. The "Plumber" is responsible for instantiating the filters and connecting them together. Once done with this, the plumber exits.

Data Stream format

The system's data streams will follow a predetermined format of measurement ID and data point. Each measurement has a unique id beginning with zero. The ID of zero is always associated with time. Test files have been provided that contain test flight data that you will use for the project. The file data is in binary format – you can use a Hex editor to read these files. The table below lists the measurements, IDs, and byte sizes of the data in these files.

ID	Data Descriptions and Units	Type	Number of Bytes
N/A	Measurement ID: Each measurement has an ID which indicates the type of measurement. The Measure-ment IDs are listed in this table in the left column.	Integer	4
000	Time: This is the number of milliseconds since the Epoch (00:00:00 GMT on January 1, 1970).	long Inte-ger	8
001	Velocity: This is the airspeed of the vehicle. It is measured in knots per hour.	Double	8
002	Altitude: This is the vehicle's distance from the surface of earth. It is measured in feet.	Double	8
003	Pressure: This is atmospheric pressure external to the vehicle. It is measured in PSI.	Double	8
004	Temperature: This is the temperature of the vehicle's hull. It is measure in degrees Fahrenheit.	Double	8
005	Pitch: This is the angle of the nose of the vehicle rela-tive to the surface of the earth. A pitch of 0 indicates that the vehicle is traveling level with respect to the earth. A positive value indicates that the vehicle is climbing; a negative value indicates that the vehicle is descending.	Double	8

Data in the stream is recorded in frames beginning with time, and followed by data with IDs between 1 and n, with n≤5. A set of time and data is called a frame. The time corresponds to when the data in the frame was recorded. This pattern is repeated until the end of stream is reached. Each frame is written in a stream as follows:

Frame 1	ID: 000	Time	ID: 001	Data	...	ID: n	Data
Frame 2	ID: 000	Time	ID: 001	Data	...	ID: n	Data
				:			
Frame F	ID: 000	Time	ID: 001	Data	...	ID: n	Data

Installing the Source Code

NOTE: These instructions assume that you have already installed Java SE 8 or higher.

First, extract the files from Lab1.zip file into a working directory. You will see three directories: *Templates*, *Sample*, *DataSets*. The *Templates* directory contains the source code templates for the filters described above. The *DataSets* directory has all of the test data that you will need. You can use a hex editor such as iHex[1] (on Mac) or FS Hex Editor[2] (on Windows) to read the contents of the binary data file. The directory *Sample* contains a working pipe-and-filter network example that illustrates the basic framework. To compile the example in the *Sample* directory, open a command prompt window (or start a Linux command line terminal), change the working directory to *Sample*, and type the following:

```
javac *.java
```

The compile process above creates the class files. After you compile the system, you can execute it by typing the following:

```
java Plumber
```

Sample is a basic pipe-and-filter network that shows how to instantiate and connect filters, how to read data from the Flightdata.dat file, and how to extract measurements from the data stream.

The output of this example is measurement data and time stamps (when each measure was recorded) – all of this is written to the terminal. If you would like to capture this information to assist you in debugging the systems, you can redirect it to a file as follows:

```
java Plumber > output.txt
```

In this example, the output is redirected to the file output.txt.

Of course, to compile and run *Sample*, you may use any IDE you prefer, e.g., Eclipse and IntelliJ

Task: Design and Construction

Your task is to use the existing framework as the basis for creating two new systems. Each new system has one or more requirements. In each system, please adhere to the pipe-and-filter architectural pattern as closely as possible. Make sure that you use good programming practices including comments that describe the role and function of any new modules, as well as describing how you changed the base system modules. Good programming practices will be appreciated.

System A

Revise the sample pipe-and-filter network to read the data stream in FlightData.dat file in a Source Filter and write the output to a text file called OutputA.csv in a Sink Filter. Format the output as follows (no need to round the values):

```
Time,Velocity,Altitude,Pressure,Temperature
```

[1] https://apps.apple.com/us/app/ihex-hex-editor/id909566003
[2] https://www.funduc.com/fshexedit.htm

```
yyyy:MM:dd:HH:mm:ss:SS,VVV.vvvvv,AAA.aaaaa,PPP.ppppp,TTT.ttttt
yyyy:MM:dd:HH:mm:ss:SS,VVV.vvvvv,AAA.aaaaa,PPP.ppppp,TTT.ttttt
yyyy:MM:dd:HH:mm:ss:SS,VVV.vvvvv,AAA.aaaaa,PPP.ppppp,TTT.ttttt
...
```

System B

Modify System A by revising the Middle Filter to filter "wild jumps" out of the data stream for altitude. A wild jump is a variation of more than 100 feet between two adjacent frames. For wild jumps encountered in the stream, replace it with the average of the previous two altitudes. Note that if the wild jump occurs in the second frame (i.e., there is only one previous altitude available), simply replace the current altitude with the previous altitude. The Middle Filter should (1) write the records of wild jumps (with the original value, before replacement) to a text file called WildPoints.csv using the same format as System A, and (2) send the updated data stream to the Sink Filter. Modify the Sink Filter of System A to write the output received from the Middle Filter to a text file called OutputB.csv by formatting the output as shown below, annotating any replaced values with an asterisk:

```
Time,Velocity,Altitude,Pressure,Temperature
yyyy:MM:dd:HH:mm:ss:SS,VVV.vvvvv,AAA.aaaaa,PPP.ppppp,TTT.ttttt
yyyy:MM:dd:HH:mm:ss:SS,VVV.vvvvv,AAA.aaaaa*,PPP.ppppp,TTT.ttttt
yyyy:MM:dd:HH:mm:ss:SS,VVV.vvvvv,AAA.aaaaa,PPP.ppppp,TTT.ttttt
...
```

Packaging and submission

- Systems A and B should be clearly separated in terms of implementation. Place each implementation in a different folder for each of the systems.
- Provide a README file clearly describing how to set-up and execute your systems. We will test your programs on our computers with test data sets that are similar to the data sets you have been provided with. If we are unable to follow your instructions (and/or they are incorrect), you will be penalized or receive a zero for the assignment.
- Upload everything in a compressed zip file via canvas.

Grading Criteria

- The correct implementation and operation of the solutions - we will test your solutions with our test data.
- The degree to which your solutions adhere to the pipe-and-filter architectural pattern where possible to do so.

Each task of the assignment is weighted as follows:

- System A: 40 points
- System B: 60 points