Software Frameworks

Instructor: Yongjie Zheng
March 23, 2020

CS 441: Software Engineering

Definitions of Software Frameworks

- A framework is a reusable, "semi-complete" application that can be specialized to produce customer applications.
- A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.
- A framework is a set of cooperating classes, some of which may be abstract, that make up a reusable design for a specific class of software.
- A framework is the skeleton of an application that can be customized by an application developer.

Frameworks in General: Extensibility

- Depending on how application code and framework code are integrated, there are *white-box frameworks* and *black-box frameworks*.
- White-box frameworks (i.e. class frameworks): application code is integrated with framework code via inheritance.
  - Application code are usually *subclasses*.
- Black-box frameworks (i.e. component frameworks): the integration is done via object composition.
  - Application code are usually *plug-ins*.

Frameworks in General: Inversion of Control

- The framework code usually has the thread of control and calls the application code (i.e. *inversion of control*).
- This is also known as the Hollywood principle: “Don’t call us - we call you”.
- This is an important difference between software frameworks and normal libraries (which are usually called by the application code).
- Potential problem: what if two different frameworks are used in a system and both of them want to claim the thread of control?

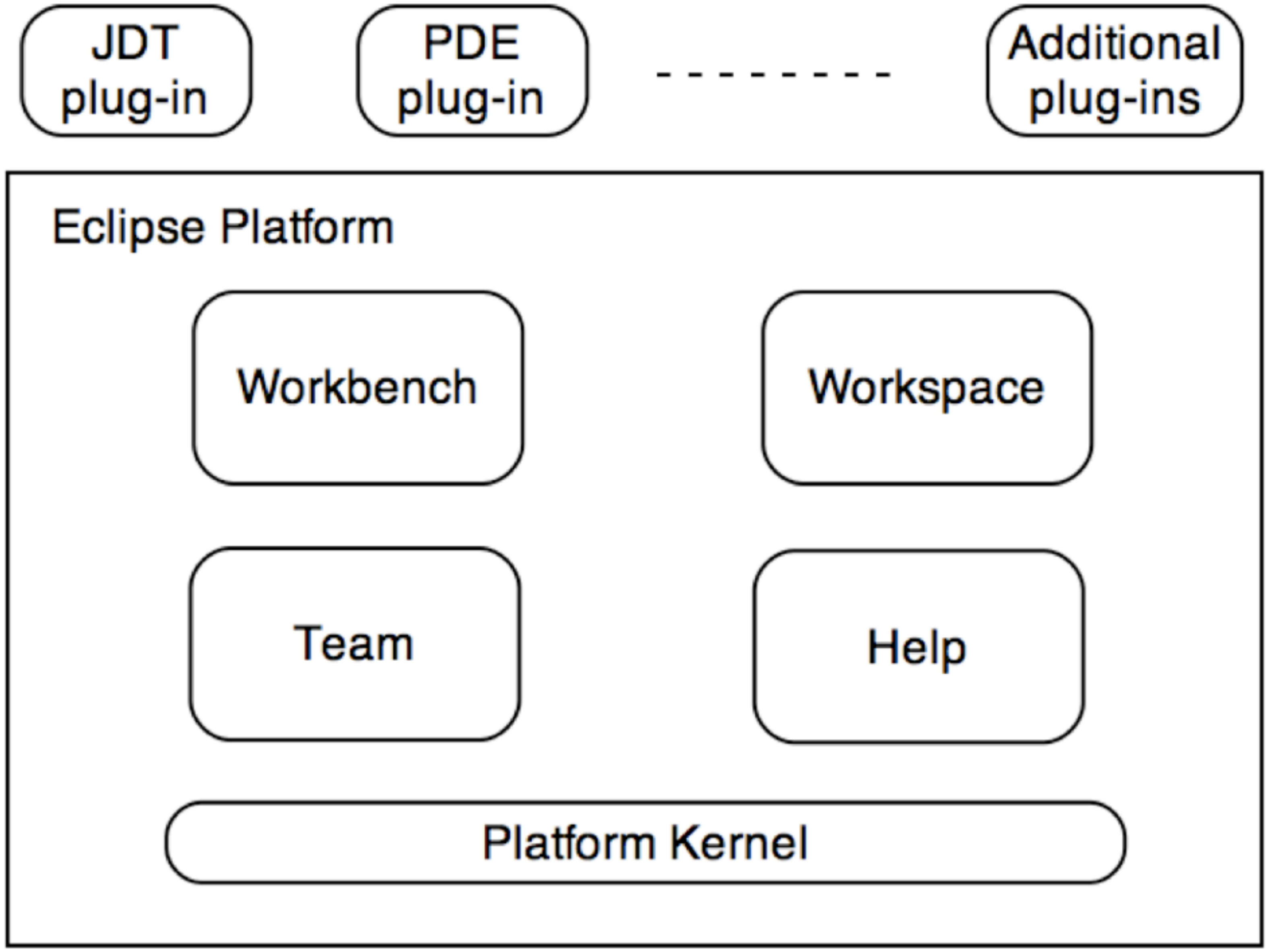
Frameworks in General: Challenges

- Code bloat: the framework code needs to be deployed together with the developed system, even if only a portion of the framework is actually used.
- Efficiency: frameworks add a layer of functionality between the application and the hardware it runs on.
- Frameworks facilitate reuse. However, it is often more expensive to build a framework than to build a single application.

Existing Frameworks

- The OSGi framework used in Eclipse
- Frameworks for the pipe-and-filter architecture style
  - Nearly every programming language implemented on every major operating system is bundled with a library or module that serves as an architecture framework for the pipe-and-filter style.
  - The standard I/O framework (**stdio**) of the C programming language.
  - The **Java.io** framework of Java.
- JavaScript Frameworks: Angular, React, Node.js, Dojo

Eclipse Infrastructure



More about Eclipse Infrastructure

- In essence, Eclipse is a small kernel surrounded by hundreds (and potentially thousands) of *plug-ins*.
- The kernel (runtime system) is based on *Equinox*, an implementation of the *OSGi* specification.
- Even the entire Eclipse user interface is built in the form of plug-ins.
  - Based on the Standard Widget Toolkit (SWT) (instead of Java AWT or SWING).
  - JFace viewers are used to present object-oriented data.

Why do we need OSGi?

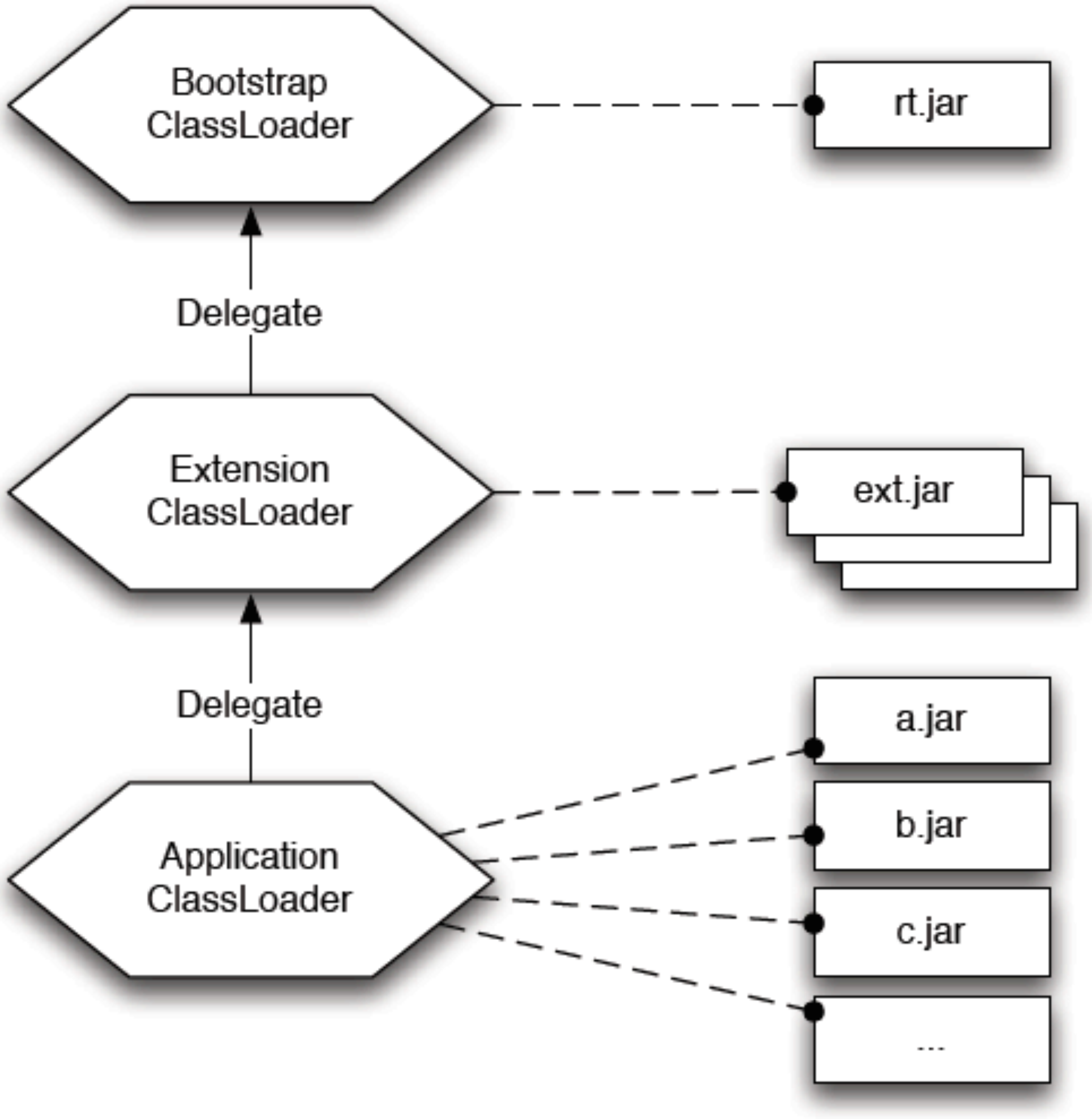
- To build a Java application as complex as Eclipse, modularity is very important.
  - Modularity: encapsulation, highly cohesive, loosely coupled.
- Java's existing mechanisms are not sufficient to build modular applications.
- OSGi is a module system based on which we can build modular applications (e.g. Eclipse) in Java.

Java's current support of modularity

- A large or enterprise Java application is often deployed as a number of JAR files, and each JAR contains a list of class files.
  - Is a JAR file really a module?
- All the JAR files share the same classpath and are dumped into a flat list when JVM runs.
  - E.g. java -classpath a.jar;b.jar;c.jar HelloWorld
- Major limitations of JARs
  - **Mutual dependency**, **Version Information**, **Information Hiding**.

Java’s Class Loading Procedure

- Parent-first delegation (tree-based class loading)
  - Bootstrap class loader: loading all the classes in the base JRE library, e.g. everything with a package name beginning with java, javax, etc.
  - Extension class loader: loading classes from the “extension” libraries.
  - Application class loader: loading from the “classpath”.



The standard Java class loader hierarchy.

Specific steps of loading a class (e.g. java -classpath log4j.jar;classes org.example.HelloWorld)

1. The JRE asks the application class loader to load a class.
2. The application class loader asks the extension class loader to load the class.
3. The extension class loader asks the bootstrap class loader to load the class.
4. The bootstrap class loader fails to find the class, so the extension class loader tries to find it.
5. The extension class loader fails to find the class, so the application class loader tries to find it, looking first in log4j.jar.
6. The class is not in log4j.jar so the class loader looks in the classes directory.
7. The class is found and loaded, which may trigger the loading of further classes — for each of these we go back to step 1.

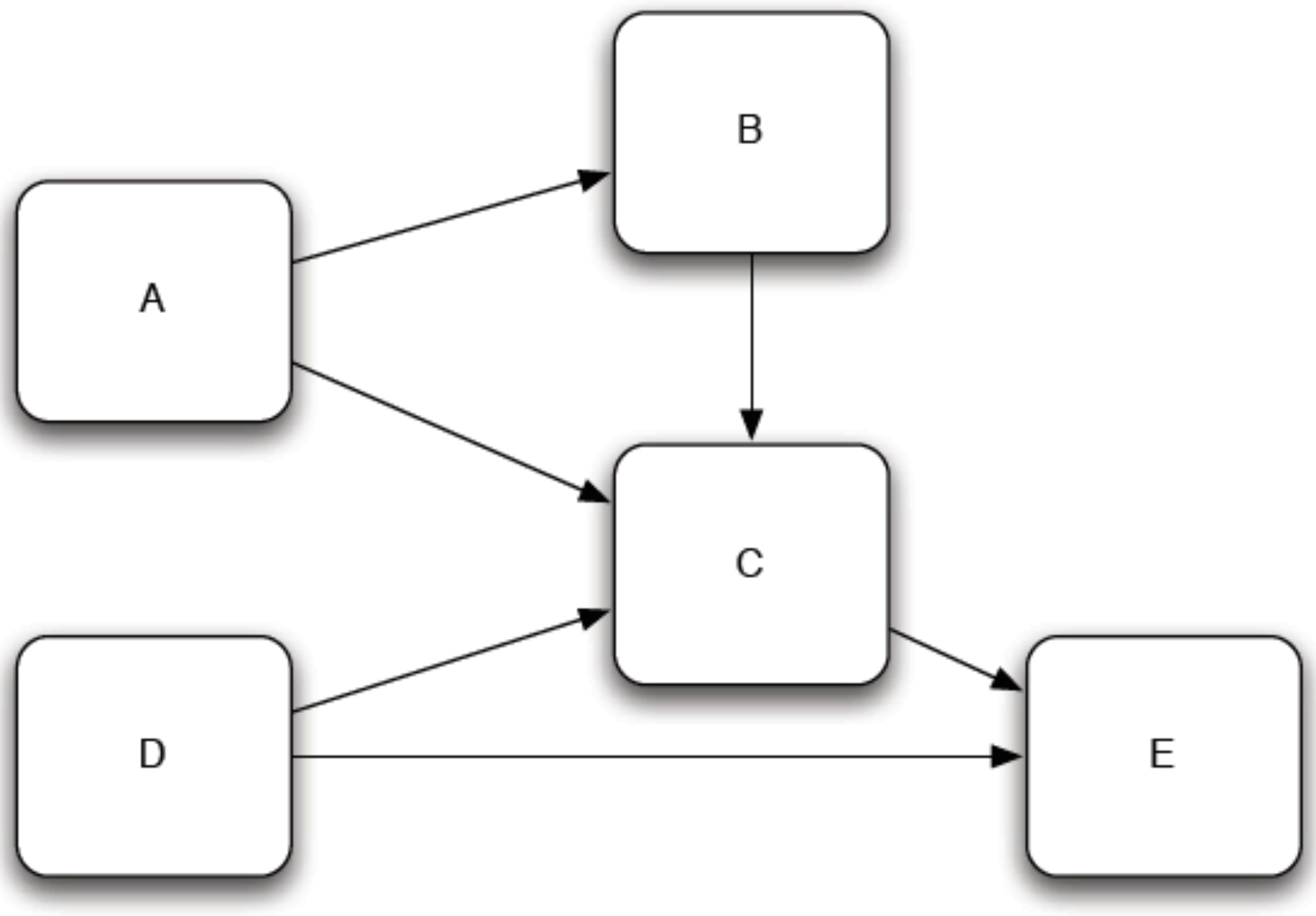
Potential Problems

- Lack of explicit dependency: one JAR may depend on the functionality provided by other JARs
- Lack of version information: multiple versions of the same JAR file may exist.
- Lack of information hiding: everything in a JAR is completely visible to all other JARs.
  - Conflicting classes: two classes of the same name may exist in two different JAR files.

The *JAR Hell* Problem

Example: java -classpath a.jar;b.jar;c.jar HelloWorld

Library A (in a.jar) works with Version 2 of library B (in b.jar), but library C (in c.jar) can only work with Version 3 of B . In a standard Java application, A and C must use the same version of B.



The OSGi class loader graph

Open Services Gateway initiative (OSGi)

- Each module has its own classpath.
- In OSGi, modules are referred to as bundles.
- Physically, a bundle is just JAR file. In addition, it contains a file named MANIFEST.MF that specifies: the *name* of the bundle, the *version* of the bundle, the list of *imports* and *exports*, etc.
- The OSGi framework will take responsibility for matching up the import with a matching export.
- The class loading procedure follows a **graph**, instead of a **tree**.

OSGi, cont.

OSGi hides everything in a JAR (bundle) unless explicitly exported. A bundle that wants to use another JAR must explicitly import the parts it needs. By default, there is no sharing.

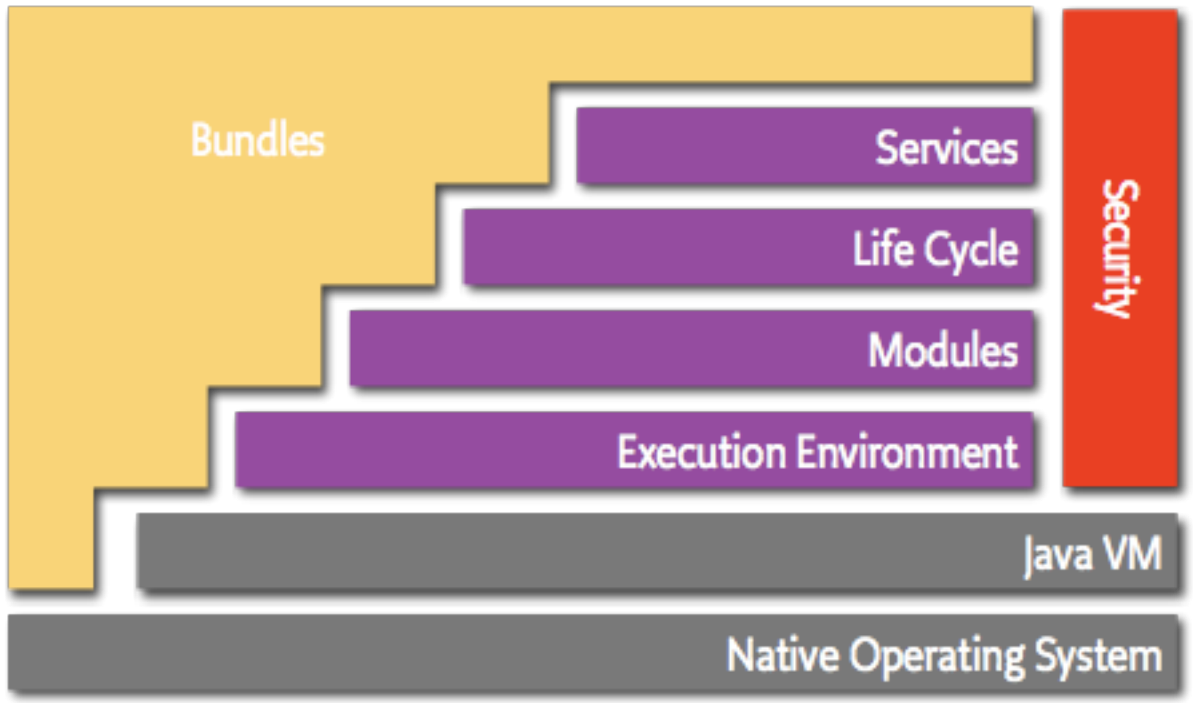
OSGi, cont.

One of the key advantages of OSGi is its **class loader model**, which uses the metadata in the manifest file. OSGi does not have a global class path. When **bundles** are installed into the OSGi Framework, the metadata is processed by the module layer and the declared **external dependencies** are reconciled against the **versioned** exports declared by other installed modules. The OSGi Framework determines the dependencies based on the manifest, and calculates the **independent** required class path for each bundle. This approach resolves the shortcomings of plain Java class loading by ensuring that the following requirements are met:

- Only packages explicitly exported by a particular bundle, through the metadata, are visible to other bundles for import.
- Each package can be resolved to specific versions.
- Multiple versions of a package can be available concurrently to different clients

From: http://pic.dhe.ibm.com/infocenter/rsahelp/v8/index.jsp?topic=/com.ibm.osgi.common.doc/topics/cosgiarchitecture.html

OSGi, cont.



The OSGi architecture from http://www.osgi.org/Technology/WhatIsOSGi

The Standard I/O Framework

- Recall the pipe-and-filter style,
    - Filters run in parallel.
    - Filters are activated when data becomes available.
    - Filters retrieve and send data through byte streams.
- In contrast, the C programming language
    - Is single threaded.
    - Uses call-return control flow for procedures.
    - Stores and retrieves all data from memory by address.
- The operating system provides services for
    - Concurrency at the process level, and at least two data streams for each process (“standard input” and “standard output”).

The Standard I/O Framework

- The stdio library provides
  - The abstraction of *streams* that includes a consistent interface for programmers to interact with different sources of I/O in their programs. For example, *getchar*(...), *putchar*(...), *scanf*(...), *printf*(...), etc..
  - Example streams: standard input, standard output, and standard error.
  - You can specify what files or processes correspond to these streams using the pipe and redirection facilities.

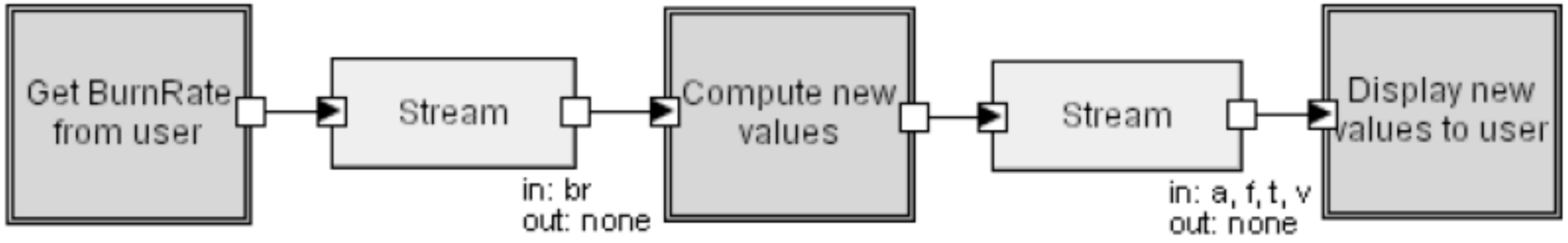
The java.io Framework

- Different from stdio in the sense that
  - The Java programming language is multithreaded, object-oriented, and bundles code and data within objects.
  - The design of the java.io library is different
    - Two base classes: *InputStream* and *OutputStream*.
    - Subclasses are created for different data sources and different requirements. For example, *BufferedInputStream*, *DataOutputStream*, *FileInputStream*.

The java.io framework can implement pipe-and-filter in two different forms

- In-process
  - Each filter is implemented as a Java thread.
  - Internal streams are used for pipes.
  - The system configuration is done in the *main* method of the program.
- Multiprocess
  - Each filter is a separate operating system process.
  - The operating-system-provided streams (standard input and standard output) are used for pipes.
  - The system configuration is done in a command line.

Implementing Lunar Lander in the pipe-and-filter style using the java.io framework



Code: GetBurnRate.java ; CalNewValues.java ; DisplayValues.java

Run: java GetBurnRate | java CalcNewValues | java DisplayValues