Implementing Architectures

Software Architecture

Chapter 9

Objectives

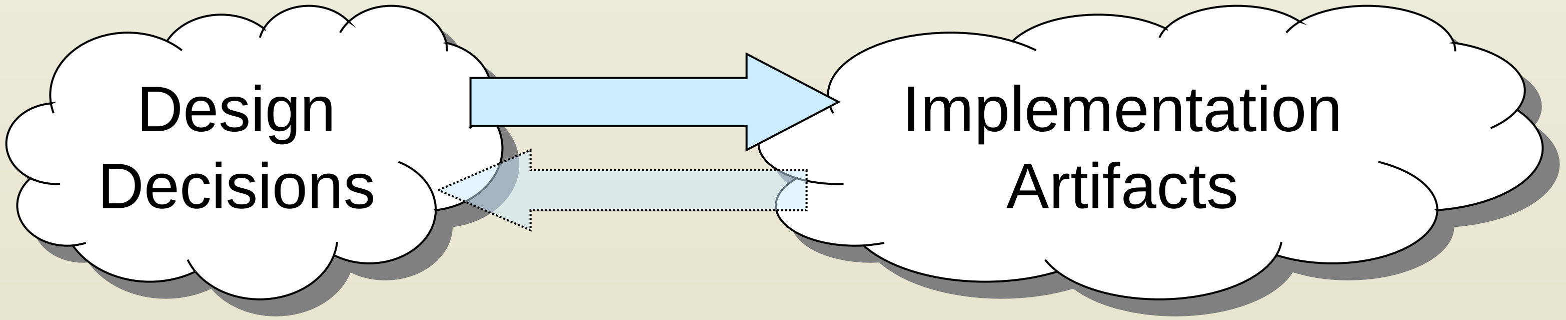
- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

Objectives

- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

The Mapping Problem

- Implementation is the one phase of software engineering that is not optional
- Architecture-based development provides a unique twist on the classic problem
  - It becomes, in large measure, a *mapping* activity



- Maintaining mapping means ensuring that our architectural intent is reflected in our constructed systems

Common Element Mapping

- Components and Connectors
  - Relevant design decisions partition the application’s functionality into discrete elements of computation and communication
  - Programming environments provide mechanisms such as packages, libraries, or classes that are used to partition functionality in implementation
  - The challenge here is to maintain a mapping between the partitions established by the architecture-level components and connectors and the partitions established by the implementation-level packages, libraries, classes, etc.
  - If this mapping is not consistent, then component boundaries may break down causing architectural drift and erosion

Common Element Mapping (cont'd)

- Interfaces
  - Architecture-level interfaces, such as method or function signatures, are straightforward to map into programming-language code
  - However, more complex interfaces, such as state machines or protocols, are harder to map

Common Element Mapping (cont'd)

- Configurations
  - At architecture level, configurations are often specified as linked graphs of components and connectors, specifying interactivity between them
  - The same interactions and topologies must be preserved in the implementation
  - Some programming languages allow one module to refer to another module by way of its interface (e.g., explicitly defined interfaces in Java or function pointer tables in C) or support dynamic discovery and connection between components at run time
  - When such constructs are available, it is often possible to generate implementation-level configurations directly from their respective architecture-level configurations

Common Element Mapping (cont'd)

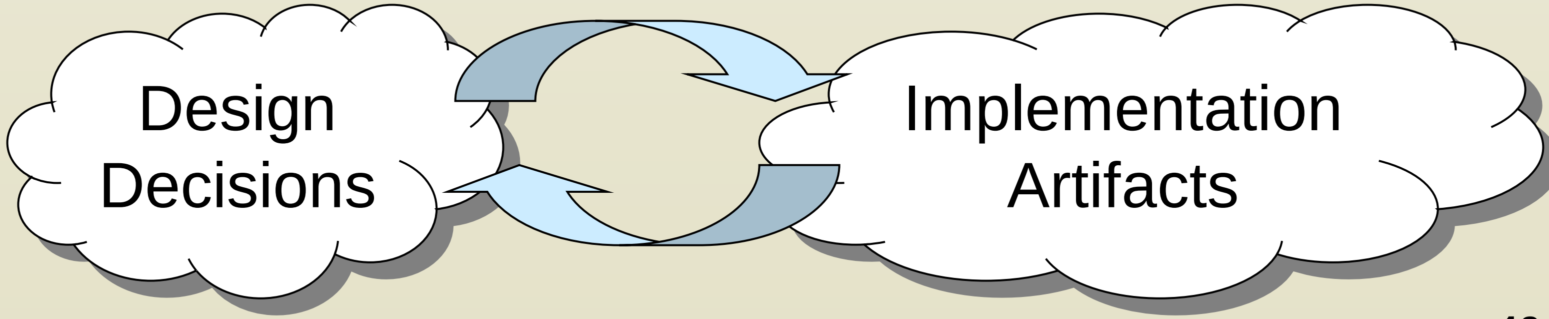
- Design rationale
  - Often does not appear directly in implementation
  - Retained in comments and other documentation
- Dynamic Properties (e.g., behavior)
  - Usually translate to algorithms of some sort (implementation skeletons or even complete implementations)
  - However, formal behavioral specifications often lack bindings to programming-language-level constructs
  - Mapping strategy depends on how the behaviors are specified and what translations are available
  - Some behavioral specifications are more useful for generating analyses or testing plans, rather than for implementations

Common Element Mapping (cont'd)

- Non-Functional Properties
  - Extremely difficult to do since non-functional properties are abstract and implementations are concrete
  - Achieved through a combination of human-centric strategies like inspections, reviews, focus groups, user studies, beta testing, and so on
  - Therefore, refining non-functional properties into functional design decisions, where possible, is important

One-Way vs. Round Trip Mapping

- Architectures inevitably change after implementation begins
  - For maintenance purposes
  - Because of time pressures
  - Because of new information
- Implementations can be a source of new information
  - We learn more about the feasibility of our designs when we implement
  - We also learn how to optimize them



One-Way vs. Round Trip Mapping (cont'd)

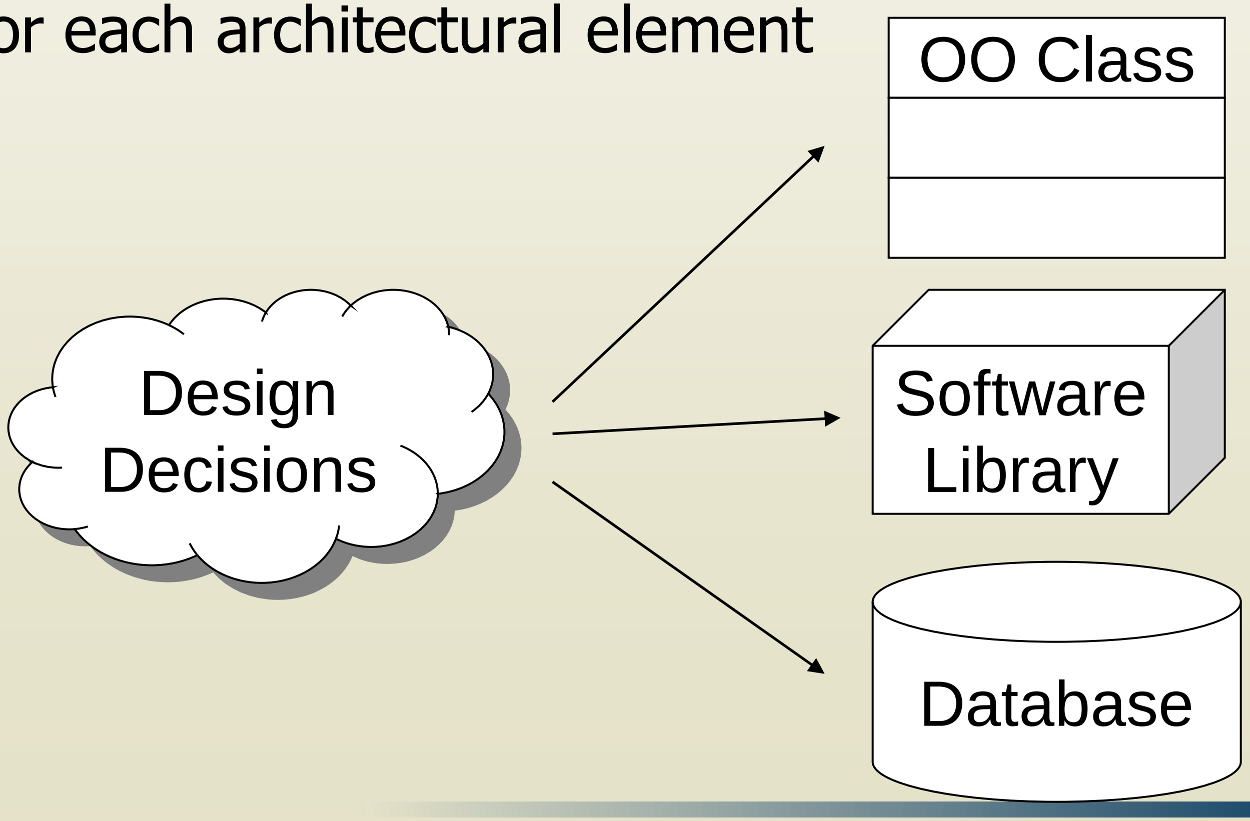
- Keeping the two in sync is a difficult technical and managerial problem
  - Places where strong mappings are not present are often the first to diverge
- One-way mappings are easier
  - Must be able to understand impact on implementation for an architectural design decision or change
- Two-way mappings require more insight
  - Must understand how a change in the implementation impacts architecture-level design decisions

One-Way vs. Round Trip Mapping (cont’d)

- One strategy: limit changes
  - If all system changes must be done to the architecture first, only one-way mappings are needed
  - Works very well if many generative technologies in use
  - Often hard to control in practice; introduces process delays and limits implementer freedom
- Alternative: allow changes in either architecture or implementation
  - Requires round-trip mappings and maintenance strategies
  - Can be assisted (to a point) with automated tools

Architecture Implementation Frameworks

- Ideal approach: develop architecture based on a known style, select technologies that provide implementation support for each architectural element

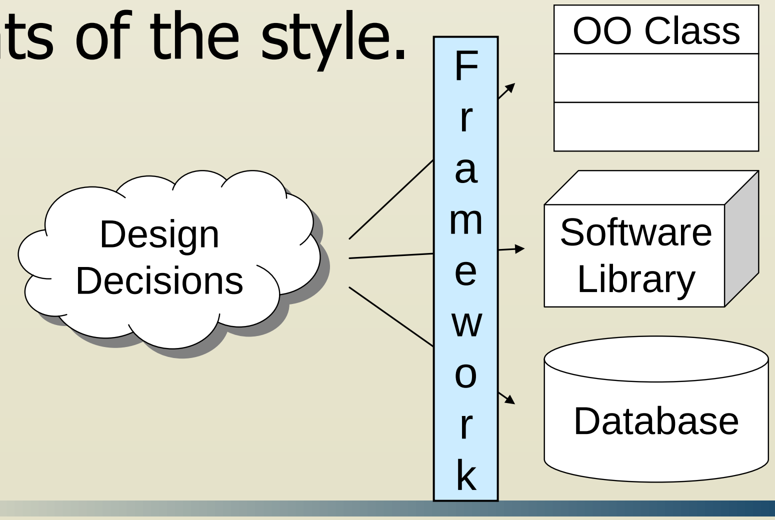


Architecture Implementation Frameworks (cont'd)

- This is rarely easy or trivial
  - Few programming languages have explicit support for architecture-level constructs
  - Support infrastructure (libraries, operating systems, etc.) also has its own sets of concepts, metaphors, and rules
  - Selection of implementation technologies is often driven by extrinsic factors, such as cost, maturity, platform support, organizational culture, etc.
- To mitigate these mismatches, we leverage an *architecture implementation framework*

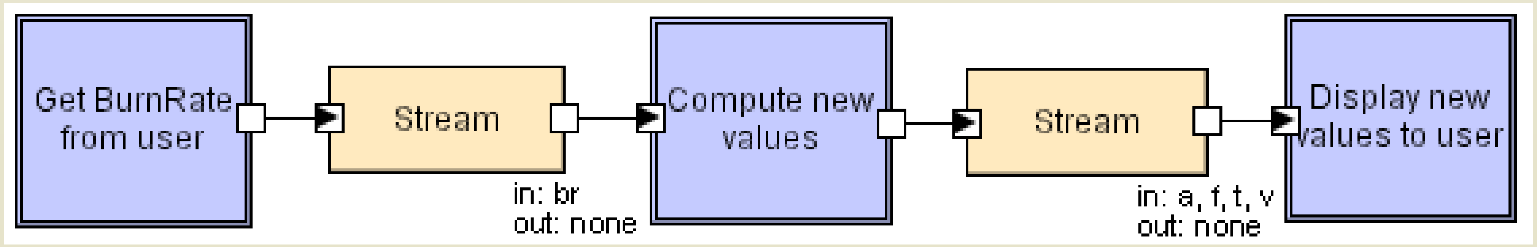
Architecture Implementation Frameworks (cont'd)

- **Definition:** An *architecture implementation framework* is a piece of software that acts as a bridge between a particular architectural style and a set of implementation technologies. It provides key elements of the architectural style *in code*, in a way that assists developers in implementing systems that conform to the prescriptions and constraints of the style.



Canonical Example

- The standard I/O ('`stdio`') framework in UNIX and other operating systems
  - Perhaps the most prevalent framework in use today
  - Actually, a bridge between the pipe-and-filter style (which is character-stream oriented and concurrent) and procedural, nonconcurrent programming languages, such as C



More on Frameworks

- Frameworks are meant to assist developers in following a style
  - But generally, do not *constrain* developers from violating a style if they really want to
- Developing applications in a target style does not *require* a framework
  - But if you follow good software engineering practices, you'll probably end up developing one anyway
- Frameworks are generally considered as underlying infrastructure or substrates from an architectural perspective
  - You won't usually see the framework show up in an architectural model, e.g., as a component or connector
  - However, frameworks often include implementations for common components and connectors (e.g., a pipe connector or an event bus)

Same Style, Different Frameworks

- For a given style, there is no one perfect architecture framework
  - Different target implementation technologies induce different frameworks
    - `stdio` (C) vs. `iostream` (C++) vs. `java.io` (Java)
- Even in the same (style/target technology) groupings, different frameworks exist due to different qualitative properties of frameworks
  - `java.io` vs. `java.nio` (the latter also supports buffering, better synchronization and fast data transfer)
  - Various C2-style frameworks in Java

Evaluating Frameworks

- Frameworks, like any software system, can vary widely along nearly any quality dimension
- Often for the same architectural style and the same environment correspond many frameworks
- Frameworks can be evaluated along a number of dimensions
  - Platform support
    - Once an architectural style has been identified, the availability of architectural frameworks for a target programming language and operating system combination can be determined

Evaluating Frameworks (cont'd)

- Frameworks can be evaluated along a number of dimensions (cont'd)
  - Fidelity (to the target architectural style)
    - How much style-specific support is provided by the framework?
      - Many frameworks are more general than one target style or focus on a subset of the style rules
    - How much enforcement is (should be?) provided?
      - More faithful frameworks are better at avoiding architectural drift/erosion but are more complicated, bigger, or less efficient

Evaluating Frameworks (cont'd)

- Frameworks can be evaluated along a number of dimensions (cont'd)
  - Matching Assumptions
    - Styles impose constraints on the target architecture/application
    - Frameworks can induce constraints as well
      - E.g., startup order, communication patterns …
    - To what extent does the framework make too many (or too few) assumptions?
    - It is important to enumerate the assumptions a framework makes and compare those with the assumptions made by other components, toolkits, libraries, and environments with which the application will interact

Evaluating Frameworks (cont'd)

- Frameworks can be evaluated along a number of dimensions (cont'd)
  - Efficiency
    - Frameworks add a layer of functionality between the application and the hardware it is run on
    - They also pervade target applications and can potentially get involved in any interaction (e.g., mediate all communication between components or dictate the concurrency policy)
    - Useful to run benchmarks; e.g., if a framework can exchange 10,000 messages per minute, it is unrealistic to expect that an application built with that framework will be able to exchange 20,000 messages per minute

Evaluating Frameworks (cont'd)

- Frameworks can be evaluated along a number of dimensions (cont'd)
  - Other quality considerations
    - Nearly every other software quality can affect framework evaluation and selection
      - Size
      - Cost
      - Ease of use
      - Reliability
      - Robustness
      - Availability of source code
      - Portability
      - Long-term maintainability and support

Middleware, Component Models, and Application Frameworks

- A spectrum of technologies exists to integrate software components and provide services beyond those provided by programming languages and operating systems
  - CORBA, COM/DCOM, JavaBeans, .NET, Java Message Service (JMS), etc.
- We will refer to these systems collectively as "middleware"
  - CORBA provides well-defined interfaces, portability, remote procedure call…
  - JavaBeans provides a standardized packaging framework (the bean) with new kinds of introspection and binding, making it possible to compose beans more easily

Middleware and Component Models (cont'd)

- Indeed, architecture implementation frameworks *are* forms of middleware
  - There's a subtle difference in how they emerge and develop
  - Middleware generally evolves based on a set of *services* that the developers want to have available
    - E.g., CORBA: Support for language heterogeneity, network transparency, portability
  - Frameworks generally evolve based on a particular *architectural style* that developers want to use
- Why is this important?

Middleware and Component Models (cont'd)

- By focusing on *services*, middleware developers often make other decisions that substantially impact architecture
- E.g., in supporting network transparency and language heterogeneity, CORBA uses RPC
  - But is RPC necessary for these services or is it just an enabling technique?
- In a very real way, middleware induces an architectural style
  - CORBA induces the 'distributed objects' style
  - JMS induces a distributed implicit invocation style
- Understanding these implications is essential for not having major problems when the tail wags the dog!

Resolving Mismatches

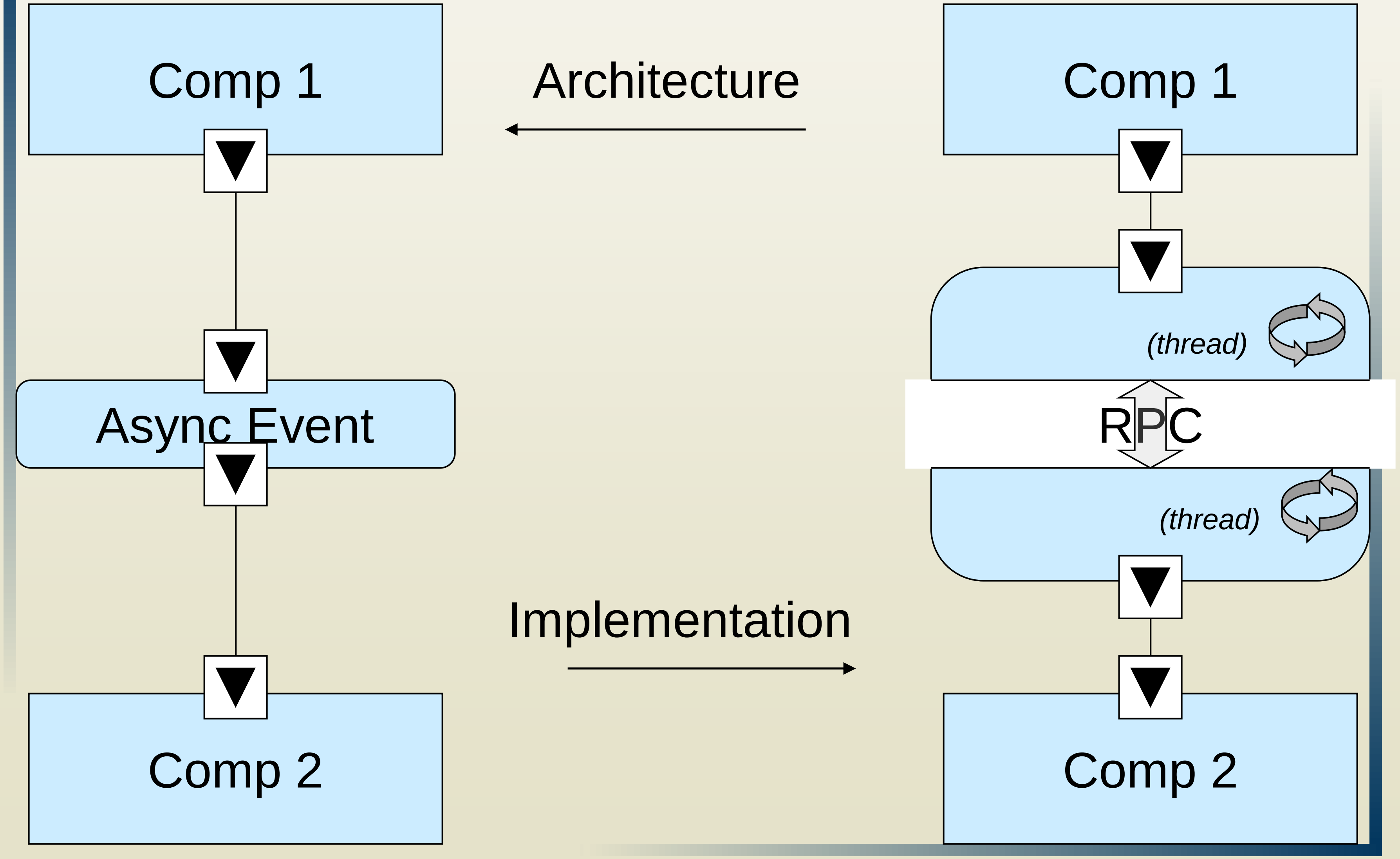
- A style is chosen first, but the middleware selected for implementation does not support (or contradicts) that style
- A middleware is chosen first (or independently) and has undue influence on the architectural style used
- Strategies
  - Change or adapt the style
  - Change the middleware selected
  - Develop glue code
  - Use only a required subset of services offered by the middleware
  - Hide the middleware in components/connectors

Use the middleware as the basis for a framework

Using Middleware to Implement Connectors

- Many middleware packages provide services which are effectively communication centric, allowing heterogeneous components to communicate
- It is possible to use the middleware to implement only the architecture's connectors, rather than the whole architecture (thus, avoiding the middleware to corrupt the architectural design)
- Any connector capabilities not supported by the chosen middleware are implemented directly within the connectors
- In this way, the connectors fulfil the architectural needs rather than bowing to the middleware assumptions

Hiding Middleware in Connectors



Building a New Framework

- Occasionally, you need a new framework
  - The architectural style in use is novel
  - The architectural style is not novel but it is being implemented on a platform for which no framework exists
  - The architectural style is not novel and frameworks exist for the target platform, but the existing frameworks are inadequate
- Good framework development is extremely difficult
  - Frameworks pervade nearly every aspect of your system
  - Making changes to frameworks often means changing the entire system
  - A task for experienced developers/architects

New Framework Guidelines

- Understand the target style first
  - Enumerate all the rules and constraints in concrete terms
  - Provide example design patterns and corner cases
- Limit the framework to the rules and constraints of the style
  - Do not let a particular target application's needs creep into the framework
  - Including application-specific features (that are not part of the style) in a framework, limits the reusability of the framework and blurs the border between application and framework

New Framework Guidelines (cont'd)

- Choose the framework scope
  - A framework does not necessarily have to implement all possible stylistic advantages (e.g., dynamism or distribution)
  - What you choose to implement depends on how you envision the framework to be reused for other apps
- Avoid over-engineering
  - Don't add capabilities simply because they are clever or "cool", especially if known target applications won't use them
  - These often add complexity and reduce performance

New Framework Guidelines (cont'd)

- Limit overhead for application developers
  - Every framework induces some overhead (classes must inherit from framework base classes, communication mechanisms limited)
  - Try to put as little overhead as possible on framework users
- Develop strategies and patterns for legacy systems and components
  - Almost every large application will need to include elements that were not built to work with a target framework
  - Develop strategies for incorporating and wrapping these external resources

Concurrency

- Concurrency is one of the most difficult concerns to address in implementation
  - Introduction of subtle bugs: deadlock, race conditions…
- Concurrency is often an architecture-level concern
  - Decisions can be made at the architectural level
  - Done carefully, much concurrency management can be embedded into the architecture framework
  - This can reduce (but not entirely eliminate) concurrency generated bugs
- Consider our earlier example, or how pipe-and-filter architectures are made concurrent without direct user involvement

Generative Technologies

- These technologies generate (parts of) implementations directly from their designs
- This is an attractive strategy for maintaining consistency when software architectures are mapped to executable code
- However, it is generally not a comprehensive (or easy) solution to implement properly
- There are a number of generative strategies that can be employed in architecture-centric development

Generative Strategies

- Generation of complete implementations of systems or elements
  - If it is feasible, it eliminates architectural drift and erosion, as implementations are simply transformations of the architecture
  - But it requires extremely detailed models including behavioral specifications
  - More feasible in domain-specific contexts
- Generation of skeletons or interfaces
  - With detailed structure and interface specifications
  - Behavioral specifications can be implemented as finite-state automata
- Generation of compositions (e.g., glue code)
  - If a library of reusable component and connector implementations is already available and systems are simply composed from this library, architectural models can generate the configurations, using where necessary glue code

Maintaining Architecture-to-Implementation Consistency

- Strategies for maintaining one-way or round-trip mappings
  - Create and maintain traceability links from architectural implementation elements
    - Explicit links in a database, in architectural models, in code comments can all help with consistency checking
  - Make the architectural model part of the implementation
    - When the model changes, the implementation adapts automatically
    - E.g., a description in a model of how components are to be instantiated and connected can be used as an implementation artifact
  - Generate some or all of the implementation from the architecture
    - By generating component skeletons

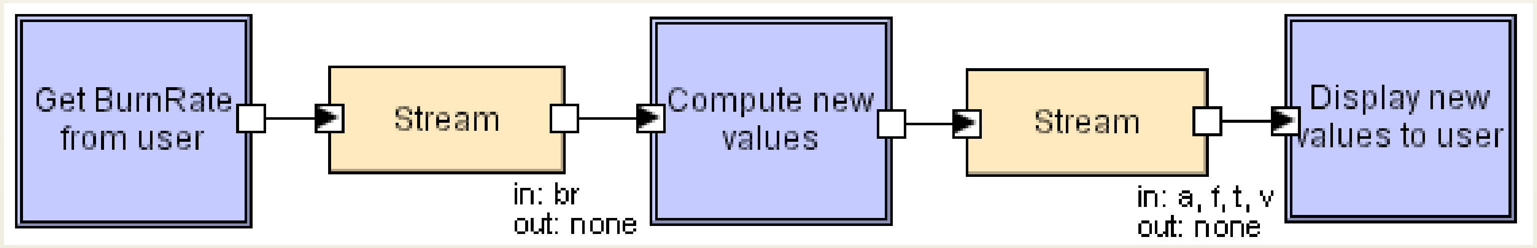
Objectives

- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

Objectives

- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

Recall Pipe-and-Filter



- Components ('filters') organized linearly, communicate through character-stream 'pipes,' which are the connectors
- Filters may run concurrently on partial data
- In general, all input comes in through the left and all output exits from the right

Framework #1: stdio

- Standard I/O framework used in C programming language
- Each process is a filter
  - Reads input from standard input (aka ‘`stdin`’)
  - Writes output to standard output (aka ‘`stdout`’)
    - Also a third, unbuffered output stream called standard error (‘`stderr`’) not considered here
  - Low and high level operations
    - `getchar`(…), `putchar`(…) move one character at a time
    - `printf`(…) and `scanf`(…) move and format entire strings
  - Different implementations may vary in details (buffering strategy, etc.)

Evaluating stdio

- Platform support
  - Available with most, if not all, implementations of C programming language
  - Operates somewhat differently on OSes with no concurrency (e.g., MS-DOS)
- Fidelity
  - Good support for developing P&F applications, but no restriction that apps have to use this style
- Matching assumptions
  - Filters are processes and pipes are implicit. In-process P&F applications might require modifications
- Efficiency
  - Whether filters make maximal use of concurrency is partially up to filter implementations and partially up to the OS

Framework #2: java.io

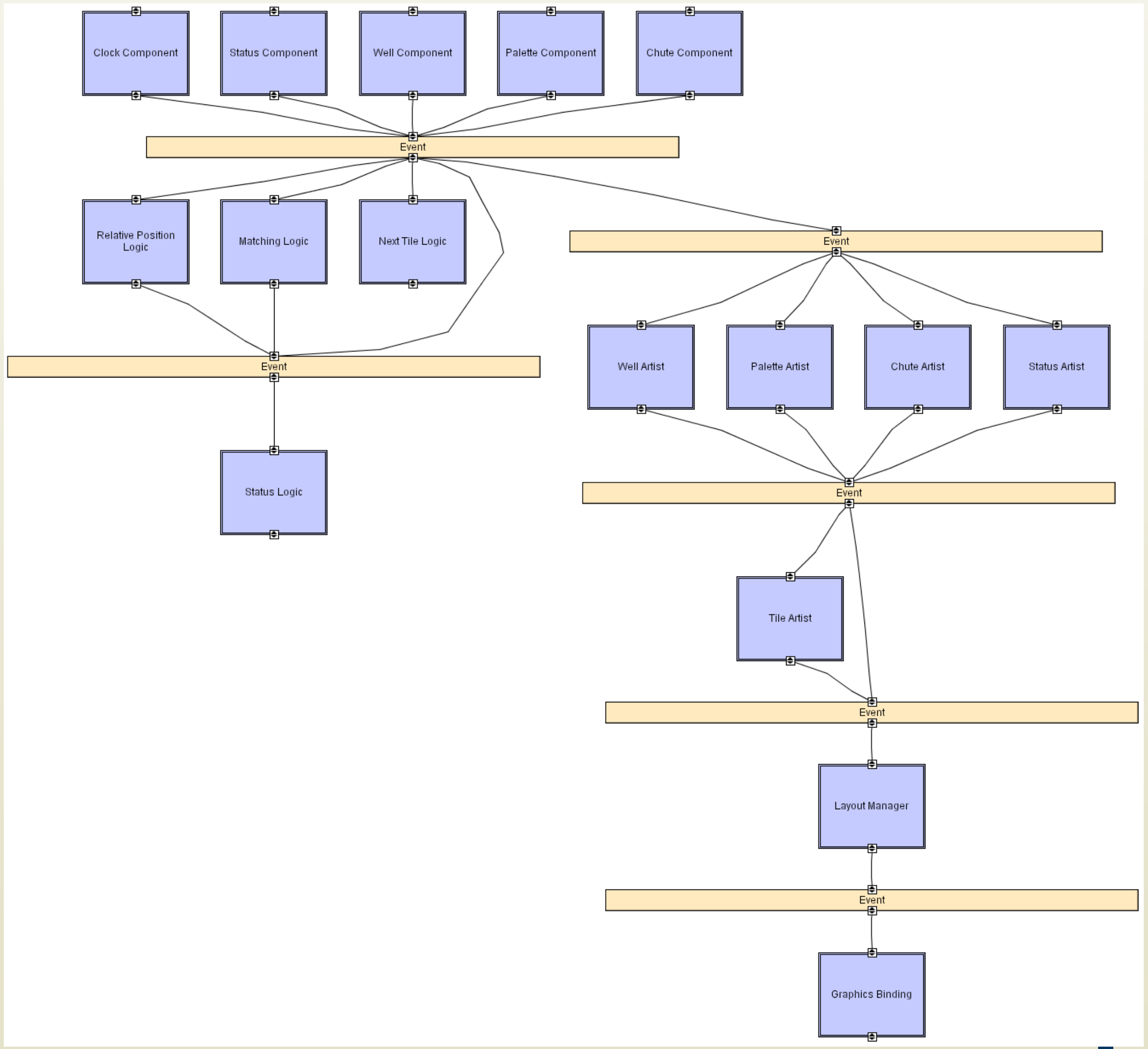
- Standard I/O framework used in Java language
- Object-oriented
- Can be used for in-process or inter-process P&F applications
    - All stream classes derive from `InputStream` or `OutputStream`
    - Distinguished objects (`System.in` and `System.out`) for writing to process' standard streams
    - Additional capabilities (formatting, buffering) provided by creating composite streams (e.g., a Formatting-Buffered-InputStream)

Evaluating java.io

- Platform support
  - Available with all Java implementations on many platforms
  - Platform-specific differences abstracted away
- Fidelity
  - Good support for developing P&F applications, but no restriction that apps have to use this style
- Matching assumptions
  - Easy to construct intra- and inter-process P&F applications
  - Concurrency can be an issue; many calls are blocking
- Efficiency
  - Users have fine-grained control over, e.g., buffering
  - Very high efficiency mechanisms (memory mapped I/O, channels) not available (but are in `java.nio`)

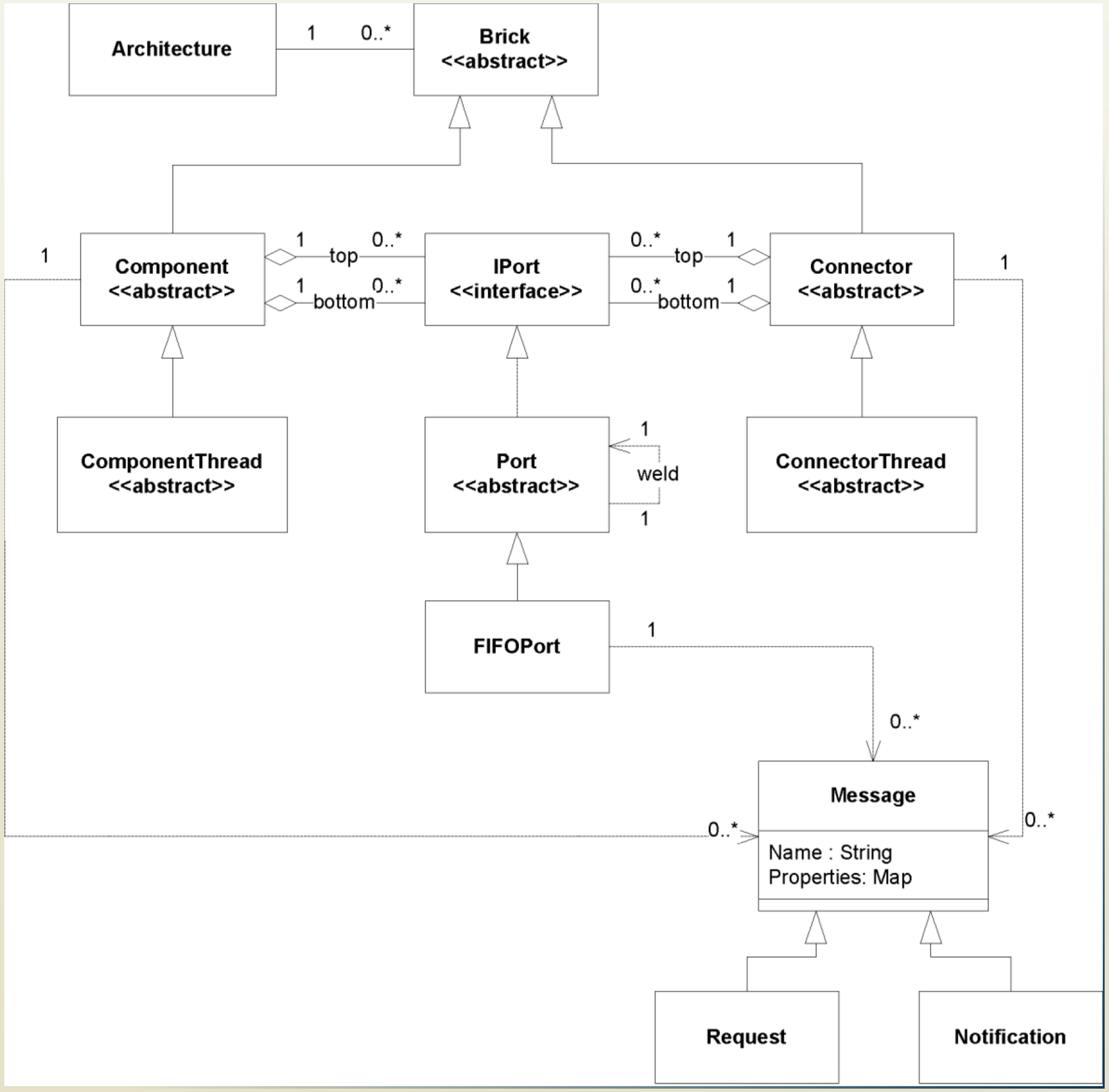
Recall the C2 Style

- Layered style with event-based communication over two-way broadcast buses
- Strict rules on concurrency, dependencies, and so on
- Many frameworks developed for different languages; focus on two alternative Java frameworks here



Framework #1: Lightweight C2 Framework

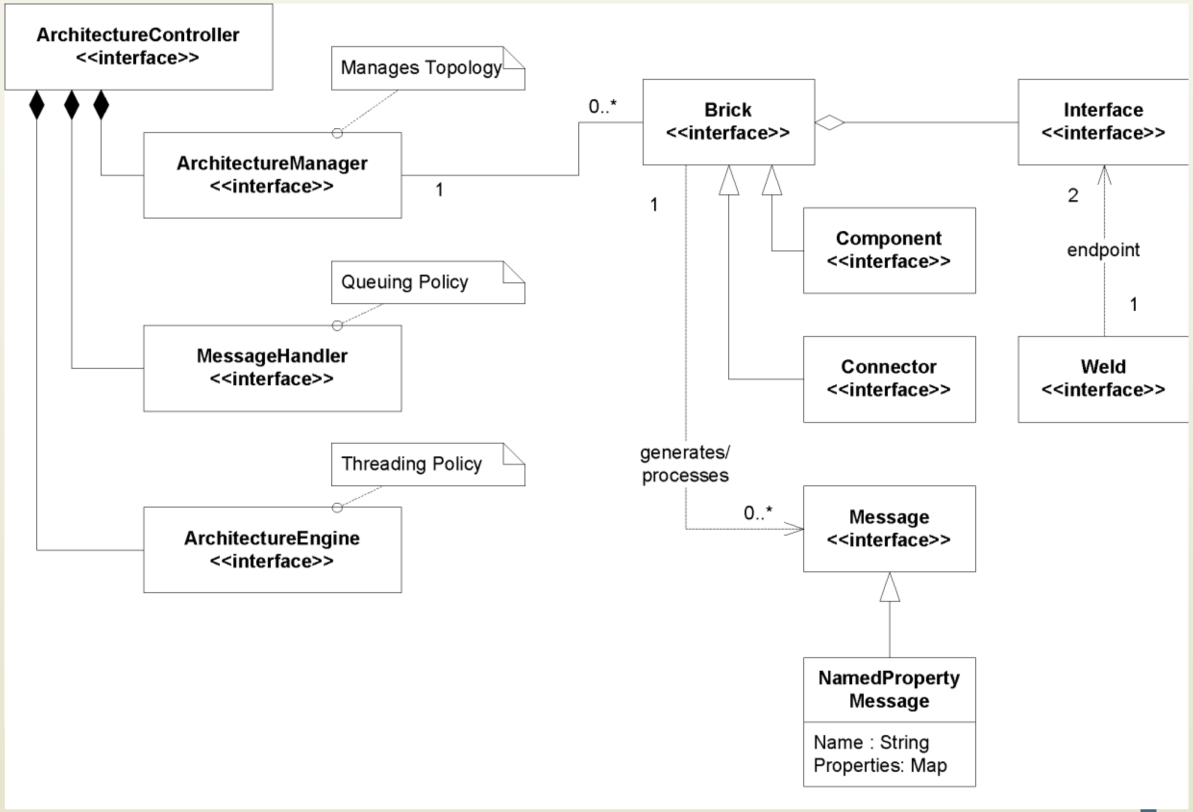
- 16 classes, 3000 lines of code
- Components & connectors extend abstract base classes
- Concurrency, queuing handled at individual comp/conn level
- Messages are request or notification objects



Evaluating Lightweight C2 Framework

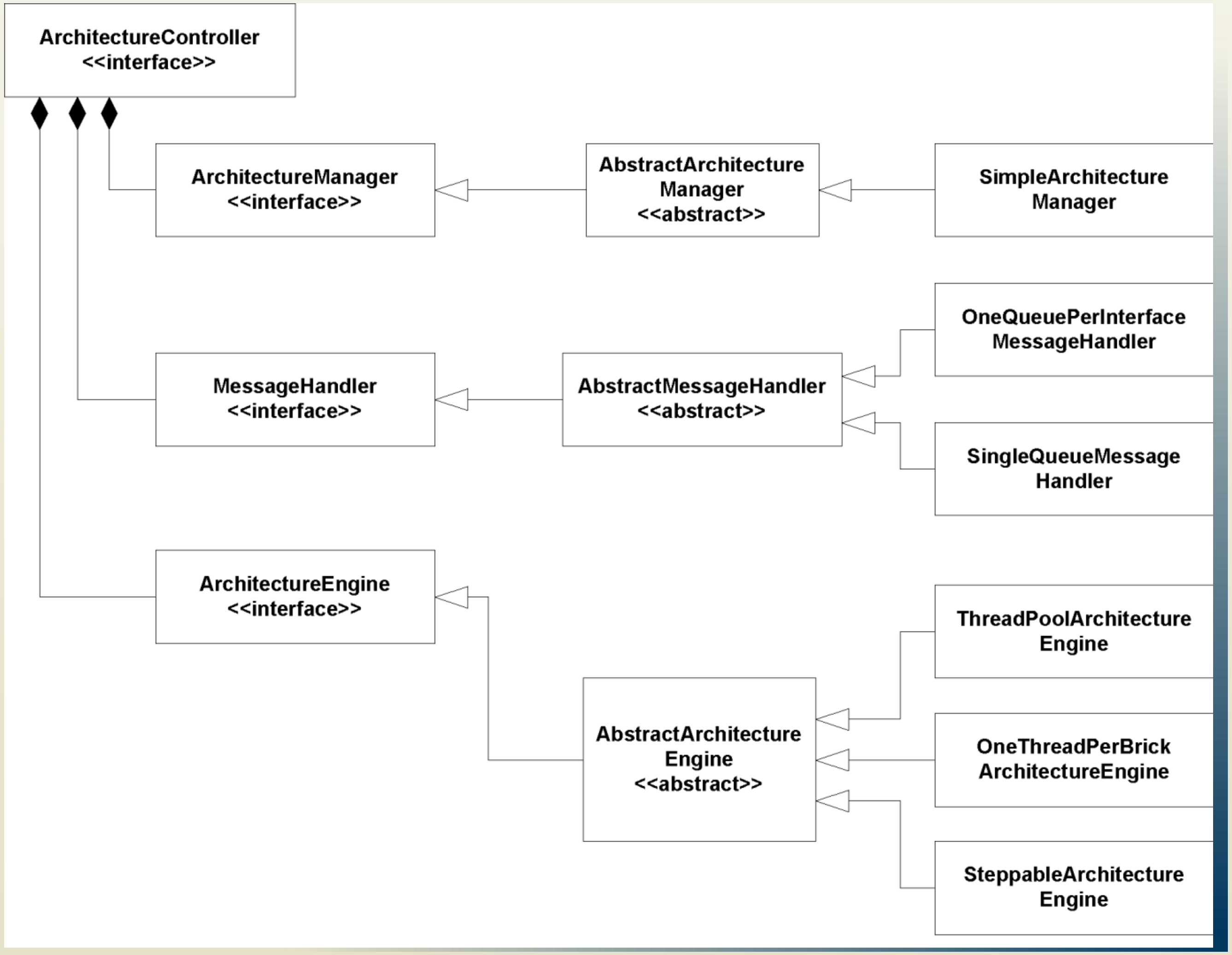
- Platform support
  - Available with all Java implementations on many platforms
- Fidelity
  - Assists developers with many aspects of C2 but does not enforce these constraints
  - Leaves threading and queuing policies up to individual elements
- Matching assumptions
  - Comp/conn main classes must inherit from distinguished base classes
  - All messages must be in dictionary form
- Efficiency
  - Lightweight framework; efficiency may depend on threading and queuing policy implemented by individual elements

Framework #2: Flexible C2 Framework



- 73 classes, 8500 lines of code
- Uses interfaces rather than base classes
- Threading policy for application is pluggable
- Message queuing policy is also pluggable

Framework #2: Flexible C2 Framework (cont’d)


ArchitectureController <<interface>>
ArchitectureManager <<interface>>
AbstractArchitecture Manager <<abstract>>
SimpleArchitecture Manager
OneQueuePerInterface MessageHandler
MessageHandler <<interface>>
AbstractMessageHandler <<abstract>>
SingleQueueMessage Handler
ArchitectureEngine <<interface>>
ThreadPoolArchitecture Engine
AbstractArchitecture Engine <<abstract>>
OneThreadPerBrick ArchitectureEngine
SteppableArchitecture Engine

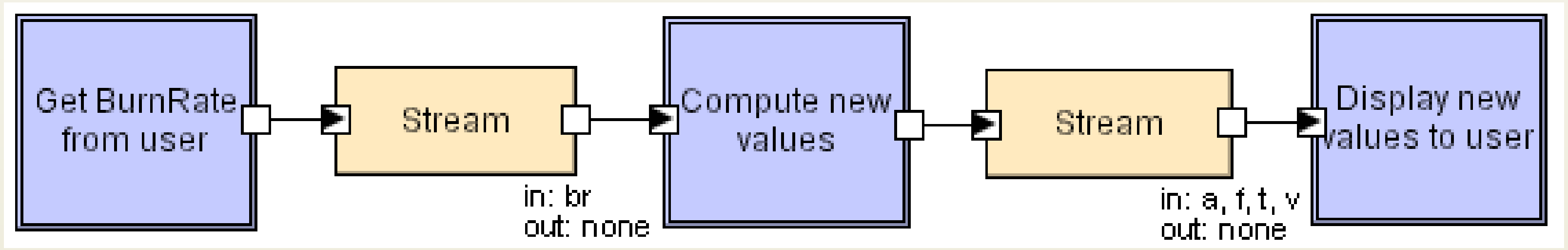
Evaluating Flexible C2 Framework

- Platform support
  - Available with all Java implementations on many platforms
- Fidelity
  - Assists developers with many aspects of C2 but does not enforce these constraints
  - Provides several alternative application-wide threading and queuing policies
- Matching assumptions
  - Comp/conn main classes must implement distinguished interfaces
  - Messages can be any serializable object
- Efficiency
  - User can easily swap out and tune threading and queuing policies without disturbing remainder of application code

Objectives

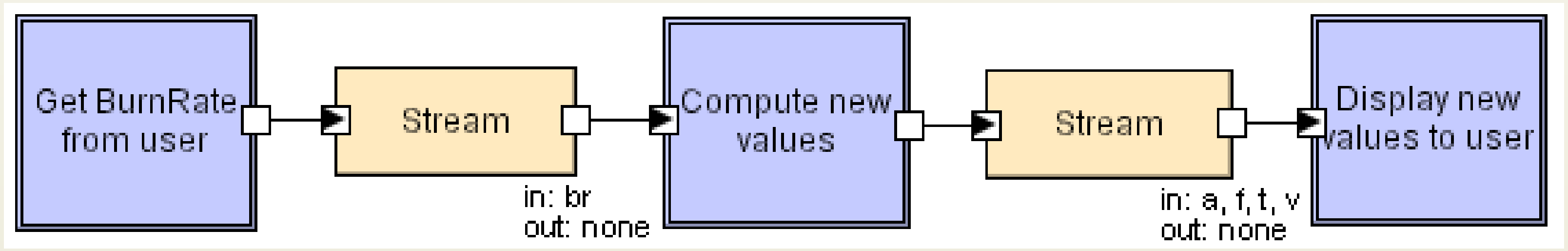
- Concepts
  - Implementation as a mapping problem
  - Architecture implementation frameworks
  - Evaluating frameworks
  - Relationships between middleware, frameworks, component models
  - Building new frameworks
  - Concurrency and generative technologies
  - Ensuring architecture-to-implementation consistency
- Examples
  - Different frameworks for pipe-and-filter
  - Different frameworks for the C2 style
- Application
  - Implementing Lunar Lander in different frameworks

Implementing Pipe and Filter Lunar Lander



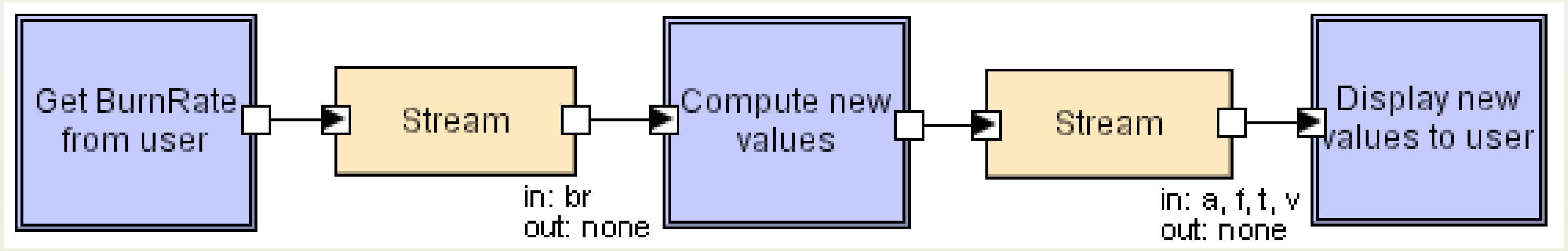
- Framework: `java.io`
- Implementing as a multi-process application
  - Each component (filter) will be a separate OS process
  - Operating system will provide the pipe connectors
- Going to use just the standard input and output streams
  - Ignoring standard error
- Ignoring good error handling practices and corner cases for simplicity

Implementing Pipe and Filter Lunar Lander (cont'd)



- A note on I/O:
  - Some messages sent from components are intended for output to the console (to be read by the user)
    - These messages must be passed all the way through the pipeline and output at the end
    - We will preface these with a '#'
  - Some messages are control messages meant to communicate state to a component down the pipeline
    - These messages are intercepted by a component and processed
    - We will preface these with a '%'

Implementing Pipe and Filter Lunar Lander (cont'd)



- First: `GetBurnRate` component
  - Loops; on each loop:
    - Prompt user for new burn rate
    - Read burn rate from the user on standard input
    - Send burn rate to next component
    - Quit if burn rate read < 0

GetBurnRate Filter

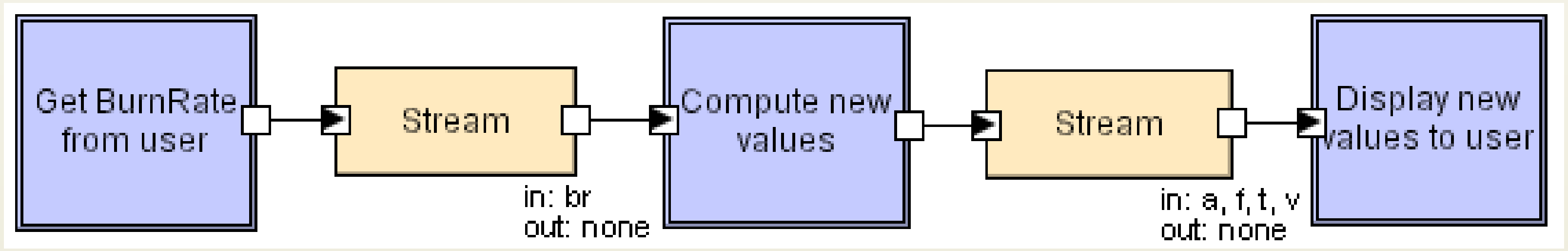
```
//Import the java.io framework
import java.io.*;

public class GetBurnRate{
  public static void main(String[] args){

    //Send welcome message
    System.out.println("#Welcome to Lunar Lander");

    try{
      //Begin reading from System input
      BufferedReader inputReader =
        new BufferedReader(new InputStreamReader(System.in));

      //Set initial burn rate to 0
      int burnRate = 0;
      do{
        //Prompt user
        System.out.println("#Enter burn rate or <0 to quit:");

. . .
```

GetBurnRate Filter (cont’d)

```
//Import the java.io framework
import java.io.*;

public cl
  public

    //Ser
    Syste

    try{
      //F
      Bu
        r

      //S
      int
      do
        /
        S

. . .
```

```
. . .
    //Read user response
    try{
     String burnRateString = inputReader.readLine();
     burnRate = Integer.parseInt(burnRateString);

     //Send user-supplied burn rate to next filter
     System.out.println("%" + burnRate);
    }
    catch(NumberFormatException nfe){
     System.out.println("#Invalid burn rate.");
    }
   }while(burnRate >= 0);
   inputReader.close();
  }
  catch(IOException ioe){
   ioe.printStackTrace();
  }
 }
}
```

Implementing Pipe and Filter Lunar Lander (cont'd)



- Second: `CalcNewValues` Component
  - Read burn rate from standard input
  - Calculate new game state including game-over
  - Send new game state to next component
    - New game state is not sent in a formatted string; that's the display component's job

CalcBurnRate Filter

```
import java.io.*;

public class CalcNewValues{

  public static void main(String[] args){
    //Initialize values
    final int GRAVITY = 2;
    int altitude = 1000;
    int fuel = 500;
    int velocity = 70;
    int time = 0;

    try{
      BufferedReader inputReader = new
        BufferedReader(new InputStreamReader(System.in));

      //Print initial values
      System.out.println("%a" + altitude);
      System.out.println("%f" + fuel);
      System.out.println("%v" + velocity);
      System.out.println("%t" + time);

. . .
```

CalcBurnRate Filter (cont’d)

```
import

public

  publi
    //I
    fin
    int
    int
    int
    int

    try
      B



      /
      S
      S
      S
      S

. . .
```

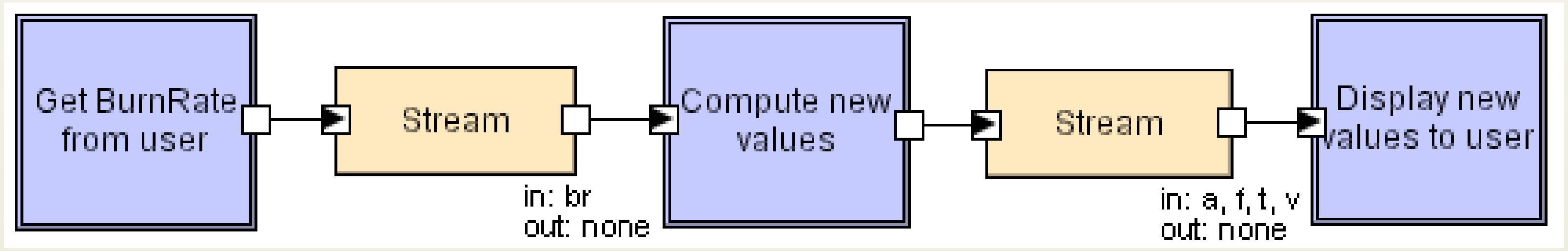
```
      String inputLine = null;
      do{
        inputLine = inputReader.readLine();
        if((inputLine != null) &&
          (inputLine.length() > 0)){

          if(inputLine.startsWith("#")){
            //This is a status line of text, and
            //should be passed down the pipeline
            System.out.println(inputLine);
          }
          else if(inputLine.startsWith("%")){
            //This is an input burn rate
            try{
              int burnRate = Integer.parseInt(inputLine.substring(1));
              if(altitude <= 0){
                System.out.println("#The game is over.");
              }
              else if(burnRate > fuel){
                System.out.println("#Sorry, you don't" +
                  "have that much fuel.");
              }
. . .
```

```
                else{
                  //Calculate new application state
                  time = time + 1;
                  altitude = altitude - velocity;
                  velocity = ((velocity + GRAVITY) * 10 -
                    burnRate * 2) / 10;
                  fuel = fuel - burnRate;
                  if(altitude <= 0){
                    altitude = 0;
                    if(velocity <= 5){
                      System.out.println("#You have landed safely.");
                    }
                    else{
                      System.out.println("#You have crashed.");
                    }
                  }
                }
                //Print new values
                System.out.println("%a" + altitude);
                System.out.println("%f" + fuel);
                System.out.println("%v" + velocity);
                System.out.println("%t" + time);
              }
              catch(NumberFormatException nfe){
              }
. . .
```

```
  else{
    //Calculate new application state
    time = time + 1;
    altitude = altitude - velocity;
    velocity = ((velocity + GRAVITY) * 10 -
      burnRate * 2) / 10;
    fuel = fuel - burnRate;
    if(
      a
      i

      }
      e

      }
    }
  }
  //Print new values
  System.out.println("%a" + altitude);
  System.out.println("%f" + fuel);
  System.out.println("%v" + velocity);
  System.out.println("%t" + time);
}
catch(NumberFormatException nfe){
}
. . .
```

```
          }
        }
      }while((inputLine != null) && (altitude > 0));
      inputReader.close();
    }
    catch(IOException ioe){
      ioe.printStackTrace();
    }
  }
}
```

Implementing Pipe and Filter Lunar Lander (cont’d)



- Third: `DisplayValues` component
  - Read value updates from standard input
  - Format them for human reading and send them to standard output

DisplayValues Filter

```
import java.io.*;

public class DisplayValues{

  public static void main(String[] args){
    try{
      BufferedReader inputReader = new
        BufferedReader(new InputStreamReader(System.in));

      String inputLine = null;
      do{
        inputLine = inputReader.readLine();
        if((inputLine != null) &&
          (inputLine.length() > 0)){

          if(inputLine.startsWith("#")){
            //This is a status line of text, and
            //should be passed down the pipeline with
            //the pound-sign stripped off
            System.out.println(inputLine.substring(1));
          }

. . .
```

```
impo

publ

  pu




. .
```

```
          else if(inputLine.startsWith("%")){
            //This is a value to display
            if(inputLine.length() > 1){
              try{
                char valueType = inputLine.charAt(1);
                int value = Integer.parseInt(inputLine.substring(2));

                switch(valueType){
                  case 'a':
                    System.out.println("Altitude: " + value);
                    break;
                  case 'f':
                    System.out.println("Fuel remaining: " + value);
                    break;
                  case 'v':
                    System.out.println("Current Velocity: ‟ + value);
                    break;
                  case 't':
                    System.out.println("Time elapsed: " + value);
                    break;
                }
              }
              catch(NumberFormatException nfe){
              }
. . .
```

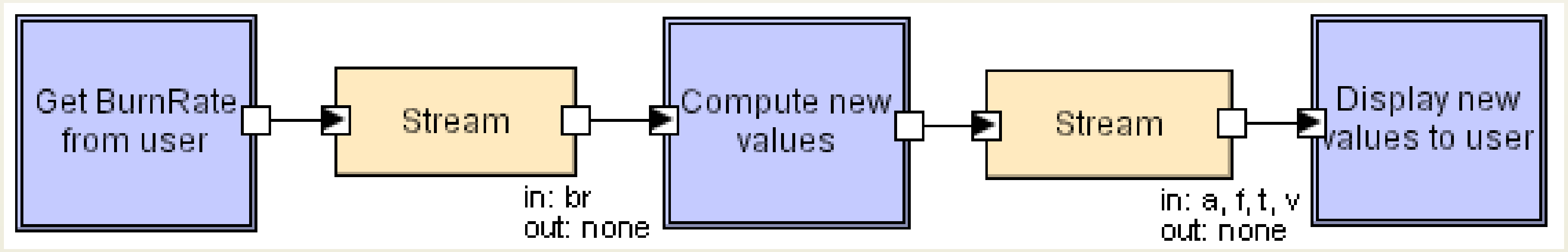
```
impo

publ

  pu

. .
```

```
            else if(inputLine.startsWith("%")){
              //This is a value to display
              if(inputLine.length() > 1){
                try{
                  char valueType = inputLine.charAt(1);
                  int value = Integer.parseInt(inputLine.substring(2));
```

```
                  }
                }
              }
            }while(inputLine != null);
            inputReader.close();
          }
          catch(IOException ioe){
            ioe.printStackTrace();
          }
        }
      }
```

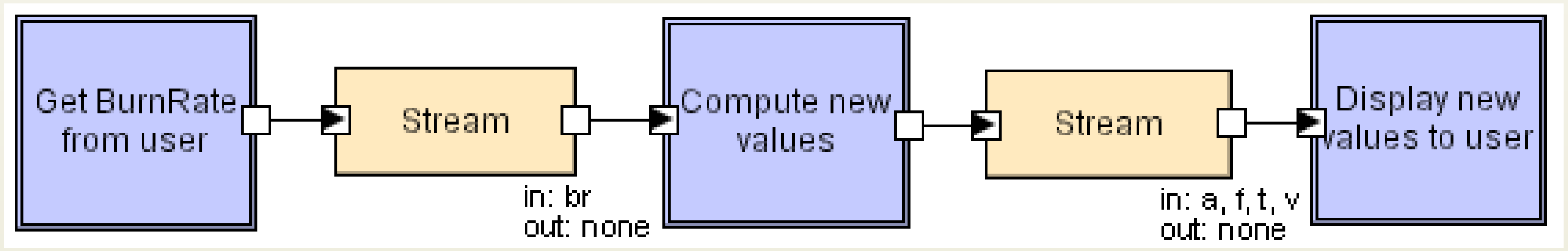
```
                      System.out.println("Time elapsed: " + value);
                      break;
                  }
                }
                catch(NumberFormatException nfe){
                }
. . .
```

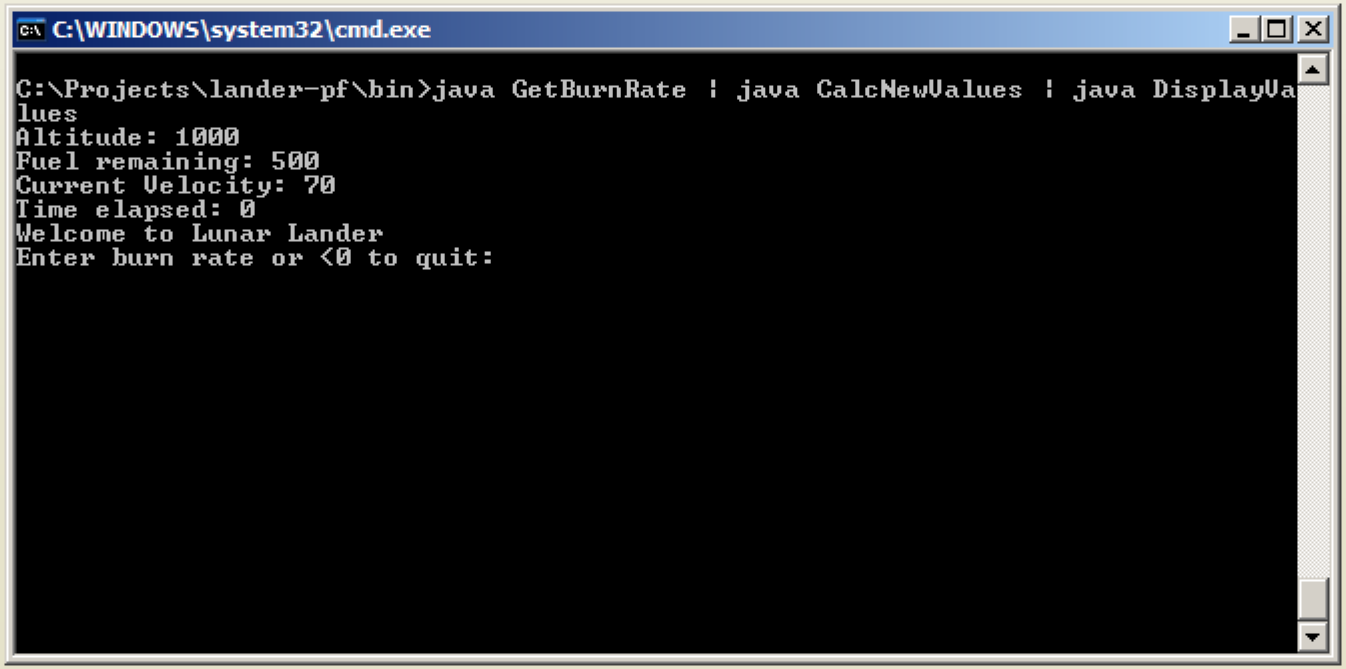
Implementing Pipe and Filter Lunar Lander (cont'd)



- Instantiating the application
  - `java GetBurnRate | java CalcNewValues | java DisplayValues`

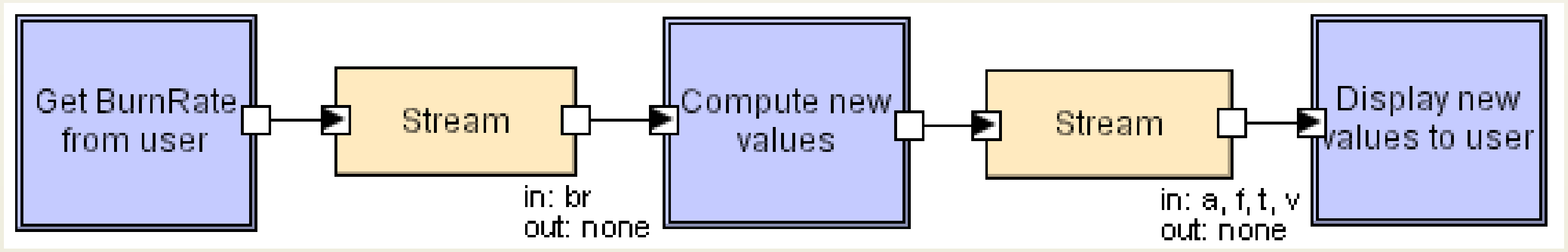
Implementing Pipe and Filter Lunar Lander (cont'd)

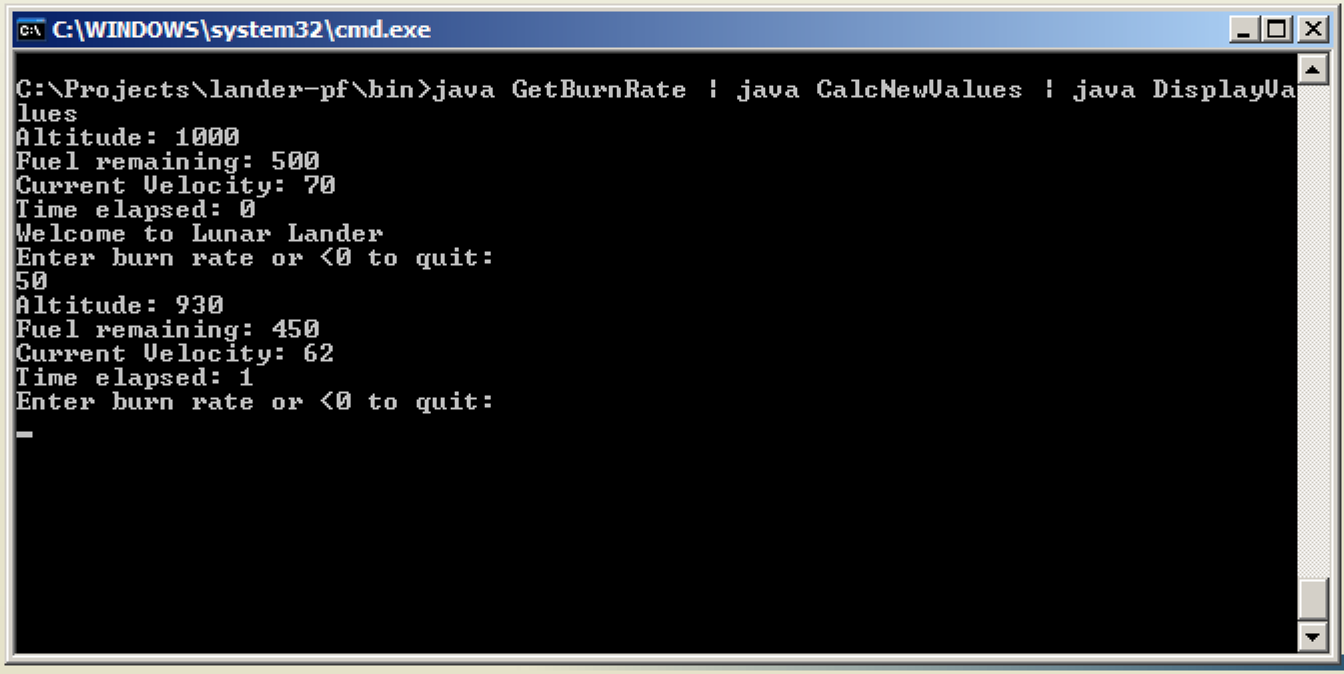



C:\WINDOWS\system32\cmd.exe

```
C:\Projects\lander-pf\bin>java GetBurnRate | java CalcNewValues | java DisplayVa
lues
Altitude: 1000
Fuel remaining: 500
Current Velocity: 70
Time elapsed: 0
Welcome to Lunar Lander
Enter burn rate or <0 to quit:
```

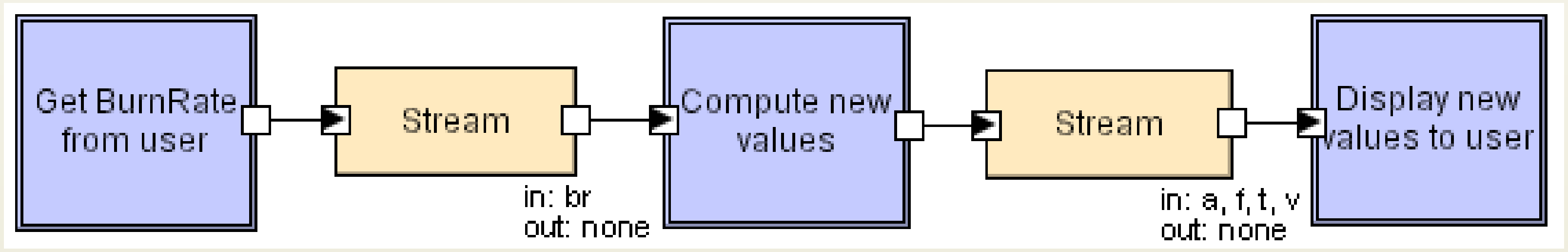
Implementing Pipe and Filter Lunar Lander (cont'd)

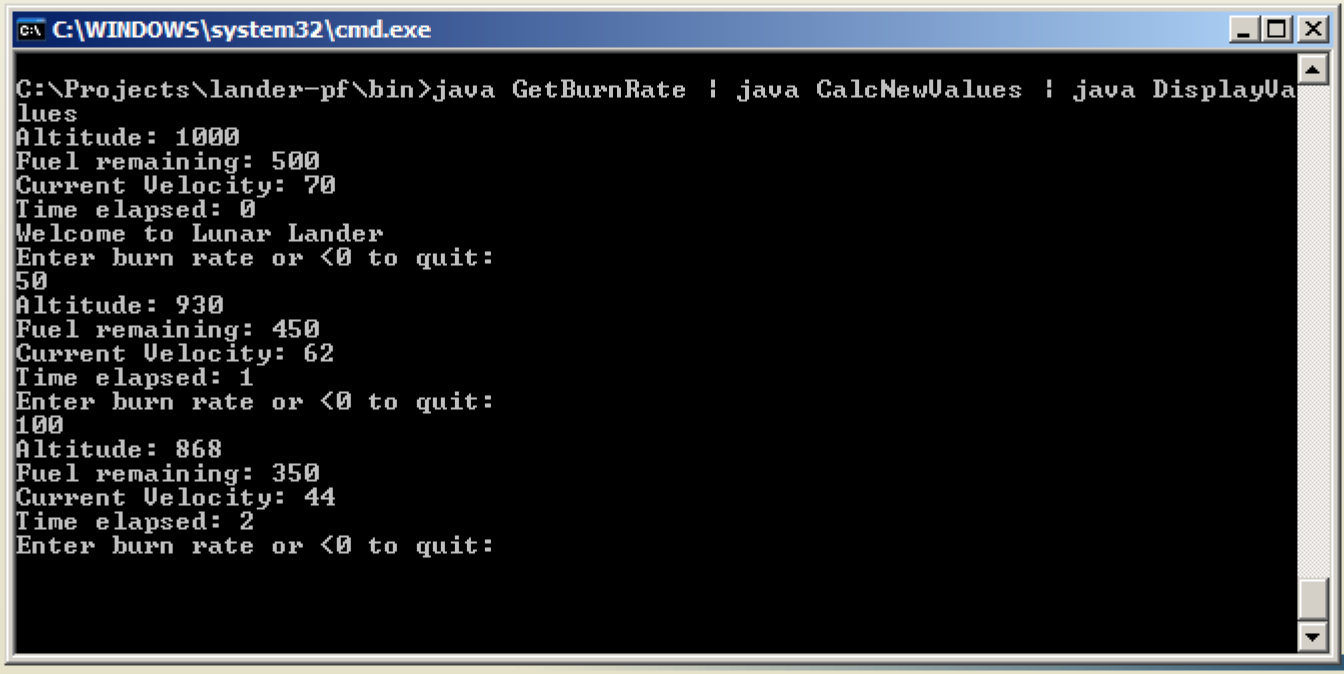




```
C:\Projects\lander-pf\bin>java GetBurnRate | java CalcNewValues | java DisplayValues
Altitude: 1000
Fuel remaining: 500
Current Velocity: 70
Time elapsed: 0
Welcome to Lunar Lander
Enter burn rate or <0 to quit:
50
Altitude: 930
Fuel remaining: 450
Current Velocity: 62
Time elapsed: 1
Enter burn rate or <0 to quit:
_
```

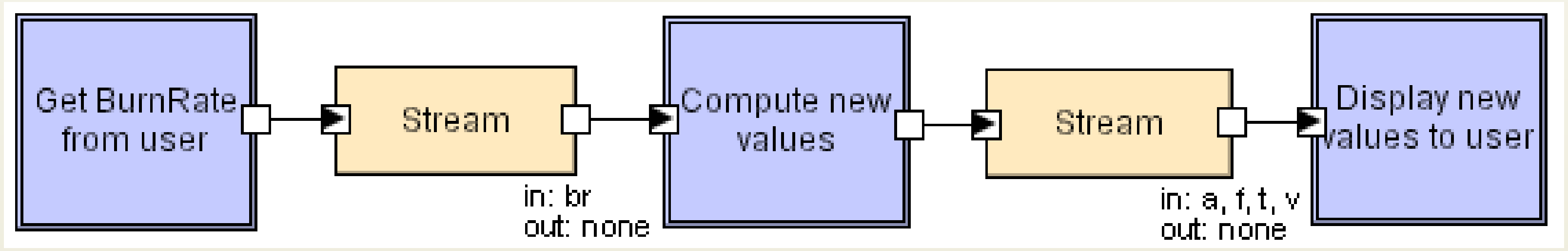
Implementing Pipe and Filter Lunar Lander (cont'd)

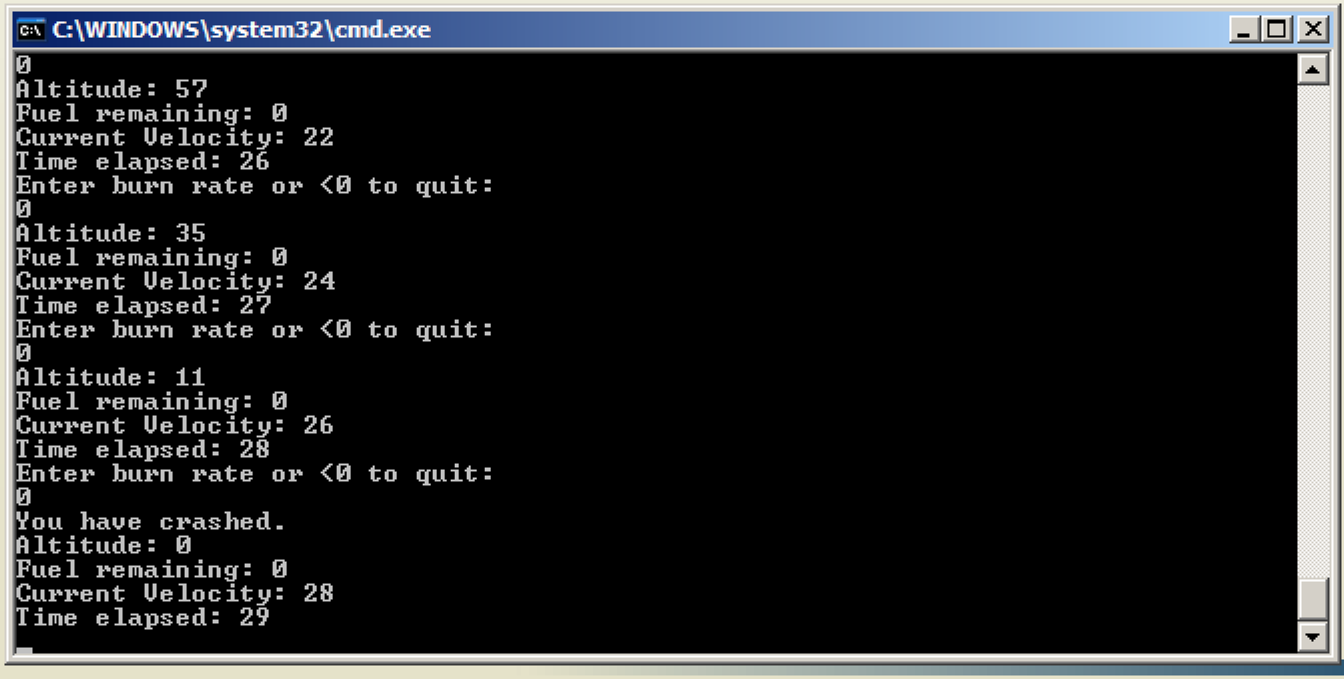




```
C:\Projects\lander-pf\bin>java GetBurnRate | java CalcNewValues | java DisplayVa
lues
Altitude: 1000
Fuel remaining: 500
Current Velocity: 70
Time elapsed: 0
Welcome to Lunar Lander
Enter burn rate or <0 to quit:
50
Altitude: 930
Fuel remaining: 450
Current Velocity: 62
Time elapsed: 1
Enter burn rate or <0 to quit:
100
Altitude: 868
Fuel remaining: 350
Current Velocity: 44
Time elapsed: 2
Enter burn rate or <0 to quit:
```

Implementing Pipe and Filter Lunar Lander (cont'd)





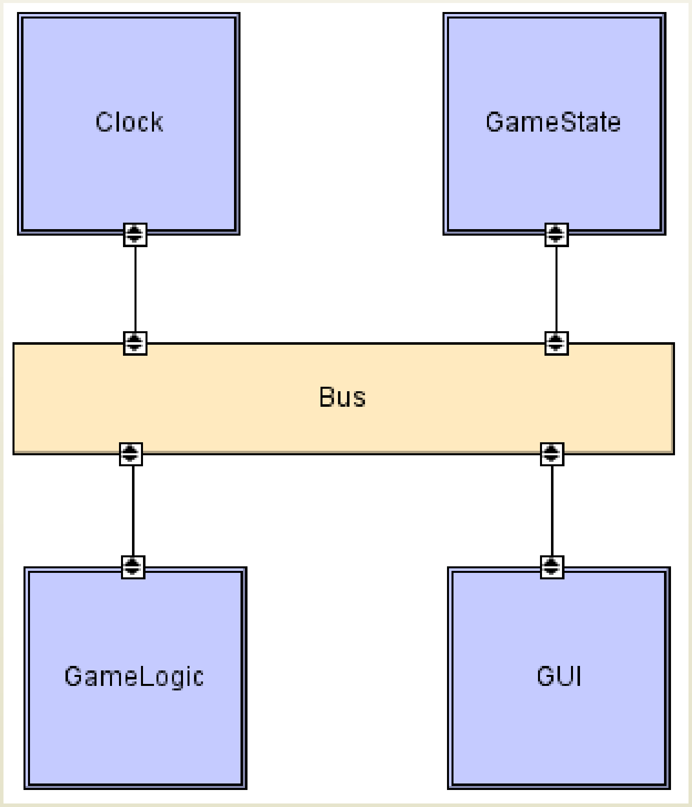
```
0
Altitude: 57
Fuel remaining: 0
Current Velocity: 22
Time elapsed: 26
Enter burn rate or <0 to quit:
0
Altitude: 35
Fuel remaining: 0
Current Velocity: 24
Time elapsed: 27
Enter burn rate or <0 to quit:
0
Altitude: 11
Fuel remaining: 0
Current Velocity: 26
Time elapsed: 28
Enter burn rate or <0 to quit:
0
You have crashed.
Altitude: 0
Fuel remaining: 0
Current Velocity: 28
Time elapsed: 29
```

Takeaways

- `java.io` provides a number of useful facilities
  - Stream objects (`System.in`, `System.out`)
  - Buffering wrappers
- OS provides some of the facilities
  - Pipes
  - Concurrency support
    - Note that this version of the application would not work if it operated in batch-sequential mode
- We had other communication mechanisms available, but did not use them to conform to the P&F style
- We had to develop a new (albeit simple) protocol to get the correct behavior

Implementing Lunar Lander in C2

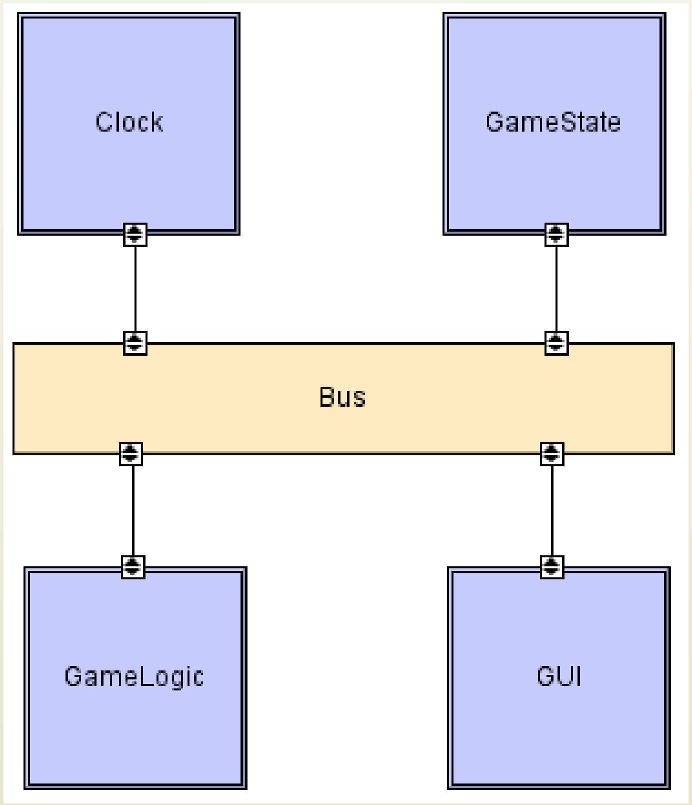
- Framework: Lightweight C2 framework
- Each component has its own thread of control
- Components receive requests or notifications and respond with new ones
- Message routing follows C2 rules



- This is a real-time, clock-driven version of Lunar Lander

Implementing Lunar Lander in C2 (cont'd)

- First: `Clock` component
- Sends out a 'tick' notification periodically
- Does not respond to any messages



```
import c2.framework.*;

public class Clock extends ComponentThread{
  public Clock(){
    super.create("clock", FIFOPort.class);
  }

  public void start(){
    super.start();

    Thread clockThread = new Thread(){
      public void run(){
        //Repeat while the application runs
        while(true){
          //Wait for five seconds
          try{
            Thread.sleep(5000);
          }
          catch(InterruptedException ie){}

          //Send out a tick notification
          Notification n = new Notification("clockTick");
          send(n);
        }
      }
    };
```

Software Architecture: Foundations, Theory, and Practice

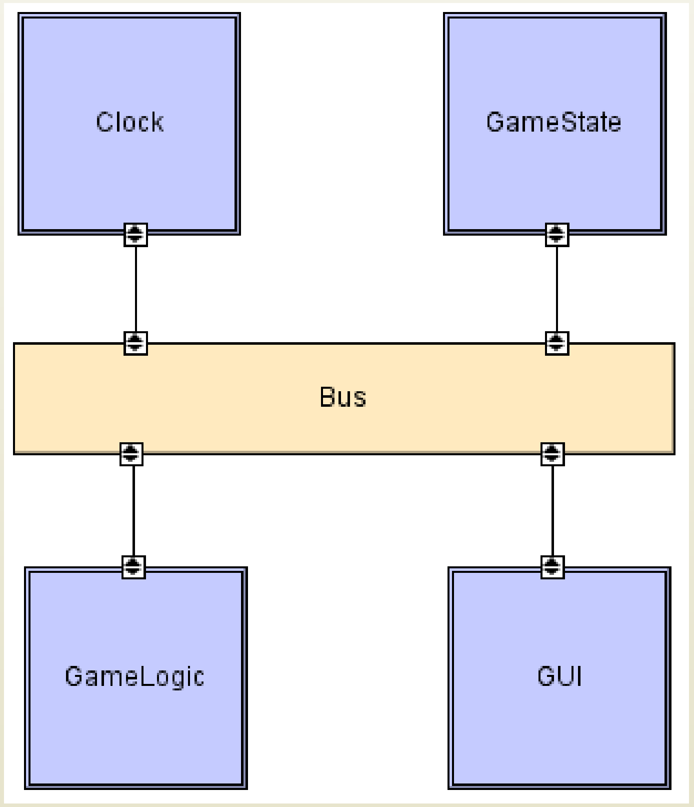
```
import c2.framework.*;

public class Clock extends ComponentThread{
  public Clock(){
    super.create("clock", FIFOPort.class);
  }

  public void start(){
    supe

    Threa
      pub
        /
        w
```

```
    clockThread.start();
  }

  protected void handle(Notification n){
    //This component does not handle notifications
  }

  protected void handle(Request r){
    //This component does not handle requests
  }
}
```

```
          //Send out a tick notification
          Notification n = new Notification("clockTick");
          send(n);
        }
      }
    };
```

Implementing Lunar Lander in C2 (cont’d)

- Second: `GameState` Component
- Receives request to update internal state
- Emits notifications of new game state on request or when state changes
- Does NOT compute new state
  - Just a data store



GameState Component

```
import c2.framework.*;

public class GameState extends ComponentThread{

  public GameState(){
    super.create("gameState", FIFOPort.class);
  }

  //Internal game state and initial values
  int altitude = 1000;
  int fuel = 500;
  int velocity = 70;
  int time = 0;
  int burnRate = 0;
  boolean landedSafely = false;
```

im

pu

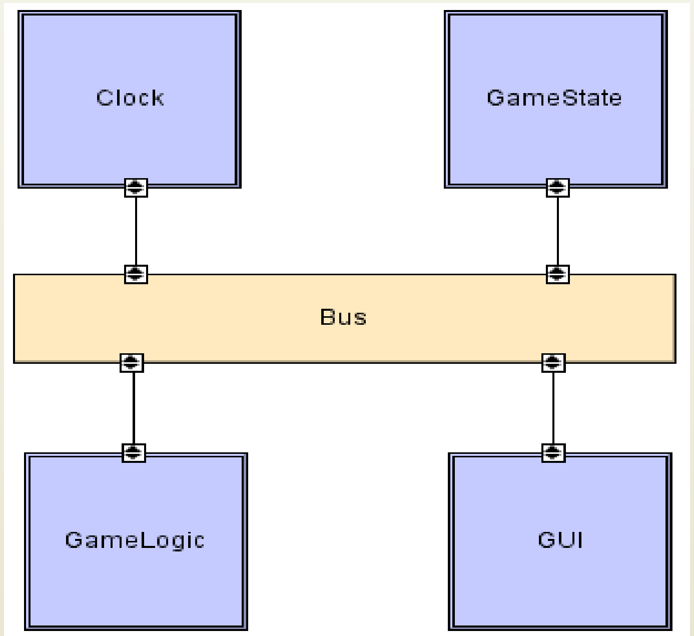
```
protected void handle(Request r){
    if(r.name().equals("updateGameState")){
      //Update the internal game state
      if(r.hasParameter("altitude")){
        this.altitude = ((Integer)r.getParameter("altitude")).intValue();
      }
      if(r.hasParameter("fuel")){
        this.fuel = ((Integer)r.getParameter("fuel")).intValue();
      }
      if(r.hasParameter("velocity")){
        this.velocity = ((Integer)r.getParameter("velocity")).intValue();
      }
      if(r.hasParameter("time")){
        this.time = ((Integer)r.getParameter("time")).intValue();
      }
      if(r.hasParameter("burnRate")){
        this.burnRate = ((Integer)r.getParameter("burnRate")).intValue();
      }
      if(r.hasParameter("landedSafely")){
        this.landedSafely = ((Boolean)r.getParameter("landedSafely"))
          .booleanValue();
      }
      //Send out the updated game state
      Notification n = createStateNotification();
      send(n);
    }
```

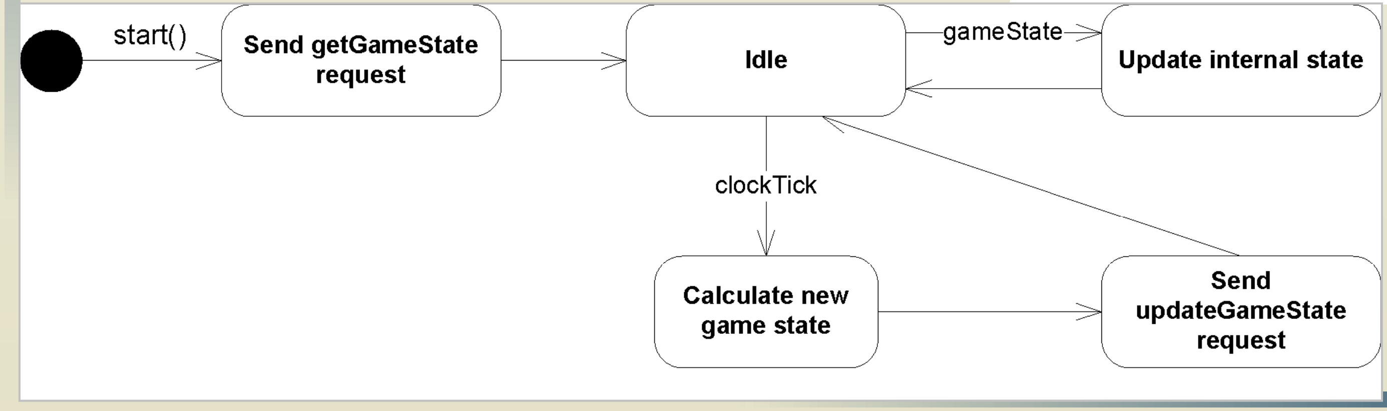
```
prote
    if
im

pu
```

```
    else if(r.name().equals("getGameState")){
      //If a component requests the game state
      //without updating it, send out the state

      Notification n = createStateNotification();
      send(n);
    }
  }

  protected Notification createStateNotification(){
    //Create a new notification comprising the
    //current game state

    Notification n = new Notification("gameState");
    n.addParameter("altitude", altitude);
    n.addParameter("fuel", fuel);
    n.addParameter("velocity", velocity);
    n.addParameter("time", time);
    n.addParameter("burnRate", burnRate);
    n.addParameter("landedSafely", landedSafely);
    return n;
  }
  protected void handle(Notification n){
    //This component does not handle notifications
  }
}
```

Implementing Lunar Lander in C2 (cont'd)

- Third: `GameLogic` Component
- Receives notifications of game state changes
- Receives clock ticks
  - On clock tick notification, calculates new state and sends request up





GameLogic Component

```
import c2.framework.*;

public class GameLogic extends ComponentThread{
  public GameLogic(){
    super.create("gameLogic", FIFOPort.class);
  }

  //Game constants
  final int GRAVITY = 2;

  //Internal state values for computation
  int altitude = 0;
  int fuel = 0;
  int velocity = 0;
  int time = 0;
  int burnRate = 0;

  public void start(){
    super.start();
    Request r = new Request("getGameState");
    send(r);
  }
```

Ga

```
import

public
  publ
    su
  }

  //Ga
  fina

  //Ir
  int
  int
  int
  int
  int

  publ
    su
    Re
    se
  }
```

```
protected void handle(Notification n){
  if(n.name().equals("gameState")){
    if(n.hasParameter("altitude")){
      this.altitude =
        ((Integer)n.getParameter("altitude")).intValue();
    }
    if(n.hasParameter("fuel")){
      this.fuel =
        ((Integer)n.getParameter("fuel")).intValue();
    }
    if(n.hasParameter("velocity")){
      this.velocity =
        ((Integer)n.getParameter("velocity")).intValue();
    }
    if(n.hasParameter("time")){
      this.time =
        ((Integer)n.getParameter("time")).intValue();
    }
    if(n.hasParameter("burnRate")){
      this.burnRate =
        ((Integer)n.getParameter("burnRate")).intValue();
    }
  }
```

Ga

```
  protected void handle(Notification n){
    if(n.name().equals("gameState")){
```

```
    else if(n.name().equals("clockTick")){
      //Calculate new lander state values
      int actualBurnRate = burnRate;
      if(actualBurnRate > fuel){
        //Ensure we don’t burn more fuel than we have
        actualBurnRate = fuel;
      }

      time = time + 1;
      altitude = altitude - velocity;
      velocity = ((velocity + GRAVITY) * 10 –
        actualBurnRate * 2) / 10;
      fuel = fuel - actualBurnRate;

      //Determine if we landed (safely)
      boolean landedSafely = false;
      if(altitude <= 0){
        altitude = 0;
        if(velocity <= 5){
          landedSafely = true;
        }
      }
```

GameLogic Component

```
  protected void handle(Notification n){
    if(n.name().equals("gameState")){
```

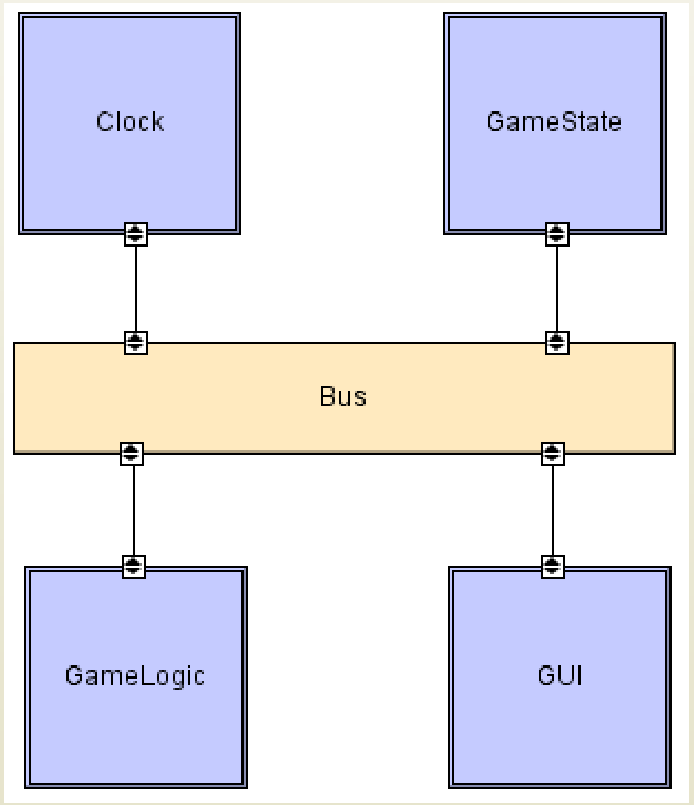
```
else if(n.name().equals("clockTick")){
  //Calculate new lander state values
  int actualBurnRate = burnRate;
```

```
      Request r = new Request("updateGameState");
      r.addParameter("time", time);
      r.addParameter("altitude", altitude);
      r.addParameter("velocity", velocity);
      r.addParameter("fuel", fuel);
      r.addParameter("landedSafely", landedSafely);
      send(r);
    }
  }

  protected void handle(Request r){
    //This component does not handle requests
  }
}
```

```
    if(velocity <= 5){
      landedSafely = true;
    }
  }
}
```

Implementing Lunar Lander in C2 (cont'd)

- Fourth: `GUI` Component
- Reads burn rates from user and sends them up as requests
- Receives notifications of game state changes and formats them to console



GUI Component

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import c2.framework.*;

public class GUI extends ComponentThread{
  public GUI(){
    super.create("gui", FIFOPort.class);
  }

  public void start(){
    super.start();
    Thread t = new Thread(){
      public void run(){
        processInput();
      }
    };
    t.start();
  }
```

```
public void processInput(){
  System.out.println("Welcome to Lunar Lander");
  try{
    BufferedReader inputReader = new BufferedReader(
      new InputStreamReader(System.in));

    int burnRate = 0;
    do{
      System.out.println("Enter burn rate or <0 to quit:");
      try{
        String burnRateString = inputReader.readLine();
        burnRate = Integer.parseInt(burnRateString);

        Request r = new Request("updateGameState");
        r.addParameter("burnRate", burnRate);
        send(r);
      }
      catch(NumberFormatException nfe){
        System.out.println("Invalid burn rate.");
      }
    }while(burnRate >= 0);
    inputReader.close();
  }
  catch(IOException ioe){
    ioe.printStackTrace();
  }
}
```

GI

```
  public void processInput(){
    System.out.println("Welcome to Lunar Lander");
```

```
protected void handle(Notification n){
  if(n.name().equals("gameState")){
    System.out.println();
    System.out.println("New game state:");

    if(n.hasParameter("altitude")){
      System.out.println("  Altitude: " + n.getParameter("altitude"));
    }
    if(n.hasParameter("fuel")){
      System.out.println("  Fuel: " + n.getParameter("fuel"));
    }
    if(n.hasParameter("velocity")){
      System.out.println("  Velocity: " + n.getParameter("velocity"));
    }
    if(n.hasParameter("time")){
      System.out.println("  Time: " + n.getParameter("time"));
    }
    if(n.hasParameter("burnRate")){
      System.out.println("  Burn rate: " + n.getParameter("burnRate"));
    }
```

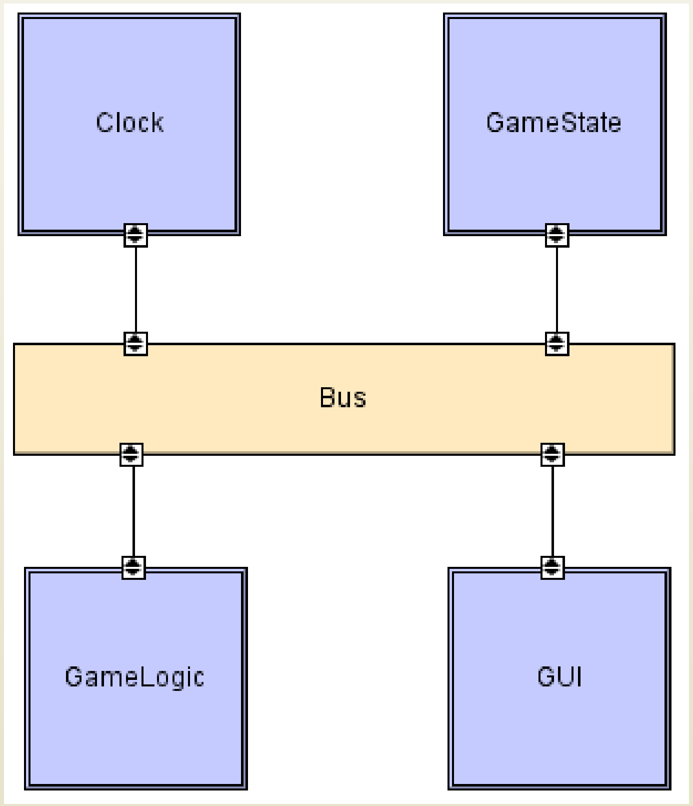
```
        ioe.printStackTrace();
      }
    }
```

```
  public void processInput(){
```

```
      if(n.hasParameter("altitude")){
        int altitude =
          ((Integer)n.getParameter("altitude")).intValue();
        if(altitude <= 0){
          boolean landedSafely =
            ((Boolean)n.getParameter("landedSafely"))
            .booleanValue();
          if(landedSafely){
            System.out.println("You have landed safely.");
          }
          else{
            System.out.println("You have crashed.");
          }
          System.exit(0);
        }
      }
    }
  }

  protected void handle(Request r){
    //This component does not handle requests
  }
}
```

Implementing Lunar Lander in C2 (cont'd)

- Lastly, `main` program
- Instantiates and connects all elements of the system



```
import c2.framework.*;

public class LunarLander{

  public static void main(String[] args){
    //Create the Lunar Lander architecture
    Architecture lunarLander = new
      SimpleArchitecture("LunarLander");

    //Create the components
    Component clock = new Clock();
    Component gameState = new GameState();
    Component gameLogic = new GameLogic();
    Component gui = new GUI();

    //Create the connectors
    Connector bus = new ConnectorThread("bus");

    //Add the components and connectors to the architecture
    lunarLander.addComponent(clock);
    lunarLander.addComponent(gameState);
    lunarLander.addComponent(gameLogic);
    lunarLander.addComponent(gui);

    lunarLander.addConnector(bus);
```

```
import c2.framework.*;

public class LunarLander{

  public static void main(String[] args){
    //Create the Lunar Lander architecture
    Ar


    //
    Co
    Co
    Co
    Co

    //
    Co

    //Add the components and connectors to the architecture
    lunarLander.addComponent(clock);
    lunarLander.addComponent(gameState);
    lunarLander.addComponent(gameLogic);
    lunarLander.addComponent(gui);

    lunarLander.addConnector(bus);
```

```
    //Create the welds (links) between components and
    //connectors
    lunarLander.weld(clock, bus);
    lunarLander.weld(gameState, bus);
    lunarLander.weld(bus, gameLogic);
    lunarLander.weld(bus, gui);

    //Start the application
    lunarLander.start();
  }
}
```

Takeaways

- Here, the C2 framework provides most all of the scaffolding we need
  - Message routing and buffering
  - How to format a message
  - Threading for components
  - Startup and instantiation
- We provide the component behavior
  - Including a couple new threads of our own
- We still must work to obey the style guidelines
  - Not everything is optimal: state is duplicated in `GameLogic`, for example

Summary

- It is imperative that, to the extent possible, the design decisions in the architecture are reflected in the implemented system
- Any conflicts or mismatches must be documented in order to mitigate risks
    - Having conflicting information in the architecture and implementation is more harmful than having an underspecified architecture
- The strongest mappings from architecture to implementation are possible when architectural models become part of the system implementation
    - This is easier to be achieved for structural design decisions but more difficult to achieve for abstract design decisions, such as any non-functional requirements