Lecture 5

Introduction to Architectural Patterns: Pipes & Filters, Message Queues, Layers
MVC Design Pattern
“One little incident of LISP beauty happened when Allen Newell visited PARC with his theory of hierarchical thinking and was challenged to prove it. He was given a programming problem to solve . . . given a list of items, produce a list consisting of all the odd indexed items followed by all of the even indexed items. [Newell] got into quite a struggle to do the program [with his IPL-V like language]. In 2 seconds I wrote down oddsEvens(x) = append(oods(x), evens(x)).

This characteristic of writing down many solutions in declarative form and have them also be the programs is part of the appeal and beauty of this kind of language. Watching a famous guy much smarter than I struggle for more than 30 minutes to not quite solve the problem his way (there was a bug) made quite an impression. It brought home to me once again that point of view is worth 80 IQ points. I wasn’t smarter but I had a much better internal thinking tool to amplify my abilities. This incident and others like it made it paramount that any tool for children should have great thinking patterns and deep beauty ‘built-in’.”

-Alan Kay, creator of Smalltalk
What is Software Architecture

- A Software Architecture provides a fundamental description of a system, detailing the components that make up the system, the significant collaborations between those components, including the data and control flows of the system.

- A Software Architecture attempts to provide a sound basis for analysis, decision making, and risk assessment of both design and performance.

- Architecture is an asset that constitutes tangible value to the organization that has created it.
What is an Architectural Pattern?

- An enterprise-level solution to common design problems
- A set of design artifacts that together provide mechanisms for solving or avoiding common enterprise-level design problems
- An architectural pattern represents common best practice models for solving problems related to designing enterprise software
- Architectural patterns, like design patterns, promote a common *vocabulary* within which to discuss enterprise design issues, along with constraints on their implementations
- Sometimes, you’ll see architectural patterns called “styles” (cf. Shaw and Garlan). They’re the same thing.
Architectural Interlude

Enterprise Messaging Concepts
Distribution vs. Integration

N-tier distributed applications tend to be tightly coupled, as:

- The tiers depend directly on one another
- The communication tends to be synchronous
- There is a certain performance expectation in terms of timely delivery of information
- The distributed objects of the system tend to form parts of a single application (although components can be leveraged by several different applications)
Distribution vs. Integration

- Integrated applications tend to be *loosely coupled*, as:
  - Multiple applications coordinate in a loosely coupled manner, they can be understood to be “disconnected” from one another
  - Each application is discrete, but may need or provide information from other applications
  - Such communication of data tends to be asynchronous
  - Individual applications can continue what they’re doing while waiting on processing from another application
Messaging

- Technologies such as CORBA, EJB, RPC, Web Services, etc. tend to support n-tier application strategies, and tend to be synchronous in their mechanisms.

- Messaging Technologies tend to be asynchronous and facilitate the integration of separate but collaborating applications.

- Messaging technologies are based on the Mediator Pattern and use it to provide numerous benefits in terms of integration.
Distribution and Integration

- Nevertheless, distributed and integrated technology strategies do share many of the same features, such as:
  - Reliable communication (exceptions)
  - Multithreaded operation
  - An object façade is available for both strategies
Types of Messaging Systems

- Operating Systems (MSMQ in Windows XP, System V IPC Message Queues in Unix)
- Application Servers and Message Oriented Middleware (MOM) (JMS as part of J2EE 1.2, now incorporated into almost all J2EE Application Servers)
- Enterprise Application Integration suites which (may) focus on process workflow such as WebSphere MQ, Microsoft BizTalk, TIBCO, WebMethods, Vitria, etc.
Fundamental Archetypes

- Delivery Metaphors (Messaging Domains)
  - Publish Subscribe
  - Point to Point

- Pipes and Filters Pattern

- Message Queue Pattern
Publish Subscribe Delivery

- Sometimes we want to broadcast the same message to more than one interested recipients (subscriber).
- We want a message to be sent to more than one recipient.
- Sending applications publish messages to the queue (topic), and receiving applications subscribe to queues (topics) to receive messages.
- Messages are delivered to subscribers via the push method (messages are pushed out to subscribers).
- Example: A price change on an exchange needs to be broadcast to several different brokerage firms interested in that price change.
Publish Subscribe Delivery

- NASDAQ Publisher
- DOW Publisher
- Message Queue
- NASDAQ Subscriber
- DOW Subscriber
- DataFlow
Point To Point Delivery

- Sometimes, we want to ensure that *only one* consumer consumes any given message

- We don’t want to broadcast a message to multiple recipients, but rather, want to make sure *only one* recipient receives *each* discrete message

- We can still have multiple (competing) consumers, but P2P guarantees that *only one of them* will get any single message

- Message delivery can either be synchronous (pull via the receive() method) or asynchronous (push via the OnMessage() callback)

- Example: We have multiple Clearing objects, and only want one Clearing object to clear any given trade (i.e., we don’t want trades being cleared *multiple* times)
Point To Point Delivery

Clearing Engine → DataFlow → Message Queue → DataFlow → Clearing Object Instance a

Clearing Object Instance b → DataFlow → Message Queue → DataFlow → Clearing Object Instance a
Pipes and Filters
Pattern

Operative Metaphor:
Conveyor Belt
The Pipes and Filters pattern is a data-flow architectural pattern that views the system as a series of transformations on successive pieces of input data.

Pipes are stateless and serve as conduits for moving streams of data between multiple filters.

Filters are stream modifiers, which process incoming data in some specialized way and send that modified data stream out over a pipe to another filter.

If you are familiar with the decorator design pattern, filters may be seen as decorators. Java’s stream IO is an example of the use of decorators as stream filters.
Details

- Unix background
  - Origination of Pipes and Filters concept with Ken Thompson of Bell Labs

- Active filters (passive pipes)
  - filters do the pushing and pulling from a passive pipe (Unix model)

- Passive filters (utilize active pipes, push and pull)
  - CORBA Event Services model with Push Consumers and Pull Supplier
Motivation:
Batch Sequential Data Processing

In the beginning, there was a void...

UNIX

Process X

/tmp/somefile

Process Y

MVS

JOB A

DASD
BSAM/QSAM

JOB B

Px >/tmp/somefile
Py < /tmp/somefile
Batch Sequential Data Processing

- Stand-alone programs would operate on data, producing a file as output.
- This file would stand as input to another stand-alone program, which would read the file in, process it, and write another file out.
- Each program was dependent on its version of input before it could begin processing.
- Therefore processing took place sequentially, where each process in a fixed sequence would run to completion, producing an output file in some new format, and then the next step would begin.
Pipes and Filters Features

- Incremental delivery: data is output as work is conducted
- Concurrent (non-sequential) processing, data flows through the pipeline in a stream, so multiple filters can be working on different parts of the data stream simultaneously (in different processes or threads)
- Filters work independently and ignorantly of one another, and therefore are plug-and-play
- Filters are ignorant of other filters in the pipeline - there are no filter-filter interdependencies
- Maintenance is again isolated to individual filters, which are loosely coupled
- Very good at supporting producer-consumer mechanisms
- Multiple readers and writers are possible
Benefits

- Fairly simple to understand and implement

- Simple, defined interface reduces complex integration issues

- Filters are substitutable black boxes, and can be plug and played, and thus reused in creative ways

- Filters are highly modifiable, since there’s no coupling between filters and new filters can be created and added to an existing pipeline
More Benefits

- Filters and Pipes can be hierarchical and can be composed into a facade mechanism to further simplify client access.
- Because filters stand alone, they can be distributed easily and support concurrent execution (the stream is *in process*).
- Multiple filters can be used to design larger complex highly-modifiable algorithms, which may be modified by adding new filters or deleting others.
Limitations

- A batch processing metaphor is not inherently limiting, but this pattern does not facilitate highly dynamic responses to system interaction.

- Because filters are black boxes, and are ignorant of one another, they cannot intelligently reorder themselves dynamically.

- Once a pipeline is in progress, it cannot be altered without corrupting the stream.

- Difficult to configure dynamic pipelines, where depending on content, data is routed to one filter or another.
More Limitations

- The “Kiss It Goodbye” scenario: useful for batch-oriented processing, limited support for interactive applications (what happens when already-processed data needs to change? Compare this to a shared-memory solution)

- May force a Lowest Common Denominator for data transmission, forcing some filters to have to modulate the stream before processing (some filters are set to handle XML, others strings, etc.)

- Each filter must internally buffer the data (some all of the data), creating memory issues in longer pipelines

- Filters which require all input to be delivered before they can begin processing (uniq, sort) can require significant buffering overhead
Performance Issues

- Filters force a lowest common denominator philosophy on data streams, usually forcing an ascii format, which can be inefficient.
- If a non-ascii stream is used, often, each filter must pay a price in marshaling and unmarshaling the data.
- Each filter is usually represented as a process, which necessarily incurs some overhead (although the use of threads can mitigate this problem).
- Data is no longer “encapsulated”, but is rather “distributed”.
Participating Classes

- Filters
  - Filters enrich, refine, or otherwise modify streams of data
  - Filters can be seen as complementing the GoF Decorator pattern

- Data Source
  - The original source for incoming data to the pipeline

- Data Sink
  - The final destination for output data from the pipeline
Push Pipeline Scenario
Pull Pipeline Scenario

Data Source: Data Source
Filter1 pull: Filter
Filter2 pull: Filter
Data sink pull: Data Sink

read
filter1
read
filter2
read
Push-Pull Scenario

data source : Data Source
filter1 pull : Filter
filter2 pull/push : Filter
data sink : Data Sink

read ()
filter1
write
filter2

read ()
Push-Pull Methods

Filters can connect to other filters in either of three ways:

- **Push**
  - The server filter will notify the client filter when new data is available and will deliver it

- **Pull**
  - The client filter will contact the server filter when it needs new data
  - this is generally a blocking mechanism

- **Try-Pull**
  - The client filter will contact the server filter when it needs new data, but will NOT block if no data is available
  - this is a non-blocking mechanism
Intermediate Buffering and Persistence

- Buffering Pipe (e.g., CORBA Event Service)
- Issues
  - Persistence
  - Redundancy
  - Efficiency
  - Non-queuing
Message Queue Pattern

Operative Metaphor: Buffered Publish Subscribe
Message Queues in Unix System V
Interprocess Communication

- Message Queues got their start in System V IPC which was first introduced in AT&T Unix SVR2, but is available now in most versions of Unix.
- Microsoft Message Queue is an implementation of message queues on the Windows platform.
- IBM’s MQSeries is another example of a message queue.
- Vitria’s 2 layer approach is built upon message queues (message queues as filters via translators).
- Message Queues represent linked lists of messages, which can be written to and read from.
Unix Message Queues

- A Message Queue is a linked list of message structures stored inside the kernel’s memory space and accessible by multiple processes.

- Synchronization is provided automatically by the kernel, so things like semaphores and other controls mechanisms are not necessary.

- New messages are added at the end of the queue in a FIFO manner.

- Each message structure has a long message type which allows for selective filtering of messages (typed access).

- Messages may be obtained from the queue either in a FIFO manner (default) or by requesting a specific type of message (based on message type).
Message Structs

- Each message structure must start with a long message type:

```c
struct mymsg {
    long msg_type;
    char mytext[512]; /* rest of message */
    int somethingelse;
    float dollarval;
};
```
Message Queue Limits

- Each message queue is limited in terms of both the maximum number of messages it can contain and the maximum number of bytes it may contain.

- New messages cannot be added if either limit is hit (new writes will normally block).

- On Linux, for example, these limits are defined as (in /usr/include/linux/msg.h):
  
  - MSGMAX 8192 /* total number of messages */
  - MSBMNB 16384 /* max bytes in a queue */
Microsoft MSMQ (Falcon)

- Integrated into XP and XP Server, Windows 7 and 8
- Provides Internet messaging which offers
  - reliable delivery (theoretical 1 terabyte buffering)
  - authentication via digital signatures
  - delivery notification and confirmation
- One-to-Many messaging model
- Message Queuing Triggers (associate a queue with a COM component or executable) (danger Will Robinson)
- Load balancing
- HTTP protocol support for delivery (firewall bypass) or HTTPS support over SSL
- URLs can be mapped to a particular queue
Event-Driven Features

- Message Queues are often used to implement event-driven (implicit invocation) systems.
- Consumers of events register interest, and producers of events broadcast events as they arise.
- Implementations can be either priority based (on message type) or interest based (based on registration which requires observers or a type-aware active queue).
CORBA COS Event Service

OMG COS Event Service Specification
The CORBA Event Service

- The COS Event Service allows to *decouple* publishers and subscribers, via an asynchronous message transfer between CORBA distributed objects.

- The Event Service acts as a well-known intermediary between objects that wish to communicate, but do not wish to be tightly coupled.

- Senders of events are called *suppliers*, and receivers of events are called *consumers*.

- The COS Event Service implements the Mediator and Proxy Patterns (GoF).

- The COS Event Service offers what is commonly referred to as the Producer-Consumer model.
Characteristics

- Messages can be accessed by either a Push or Pull metaphor
- Message passing takes place asynchronously (non-blocking model)
- The COS Event Service delivery can be set up to be typed or untyped
- The COS Event Service will automatically buffer received events, offering semi-persisted messaging
- Communication can follow either a Push or Pull model
- Consumers can either synchronously (Pull) or asynchronously (tryPull) obtain messages from the Event Services, or have the Event Service deliver events to them (Push model)
- A 1-1 correspondence between Producers and Consumers is not necessary
The Connection Process

- Regardless of the model, the following steps must be taken to connect to an event channel:
  - The client must bind to the Event Channel, which must already have been created by someone, perhaps the client
  - The client must get an Admin object from the Event Channel
  - The client must obtain a proxy object from the Admin object (a Consumer Proxy for a Supplier client and a Supplier Proxy for a Consumer client)
  - Add the Supplier or Consumer client to the event channel via a connect() call
  - Transfer data between the client and the Event Channel via a push(), pull(), or try_pull() call
The COS Event Service Design

- Suppliers and Consumers each connect to the event service, via a particular channel.

- Each channel can have multiple consumers and suppliers, and all events delivered by a supplier are made available to all consumers.

- An event channel can be thought of as a repository of common interest, the Usenet model is available as a metaphor.

- Suppliers and consumers connect to an event channel, and begin communication through the channel.
Interaction Models

- The Event Channel supports two models of interaction: Push and Pull
  - Push Model: The sender initiates the delivery of the message to the recipient
  - Pull Model: The recipient initiates the delivery of the message from the sender by request

- The Event Channel plays roles on behalf of suppliers and consumers
  - For consumers, it plays the role of a supplier
  - For suppliers, it plays the role of a consumer

- Through this role playing, the Event Channel can decouple the communication between the two interested parties
The Pull Model

- In the Pull Model, the consumer initiates the flow of events from the supplier.
- When a Pull supplier wants to connect to the event channel, the event channel will “pretend” it is a Pull consumer, by creating a consumer proxy for the supplier to talk to.
- The supplier is none the wiser, and simply delivers messages on request to its “consumer”, which is a proxy object within the event channel.
- When a Pull consumer wants to connect to the event channel, the event channel will “pretend” it is a supplier, by creating a supplier proxy for the consumer to talk to.
  - In the Pull Consumer scenario, the Pull Consumer pulls (or tries to pull) events from the Proxy Pull Supplier.
  - In the Pull Supplier scenario, the Proxy Pull Consumer pulls events from the Pull Supplier.
The Push Model

- In the Push Model, the supplier initiates the flow of events to the consumer.
- When a Push Supplier wants to connect to the event channel, the event channel will “pretend” it is a consumer, by creating a *consumer proxy* for the supplier to talk to.
- The supplier is none the wiser, and simply delivers messages to its “consumer”, which is a proxy object within the event channel.
- When a Push Consumer wants to connect to the event channel, the event channel will “pretend” it is a supplier, by creating a *supplier proxy* for the consumer to talk to.
- In the Push Consumer scenario, the event channel *pushes* events out to the consumer via the *push supplier proxy*.
- In the Push Supplier scenario, the supplier *pushes* events out to the event channel via its *push consumer proxy*. 
The Hybrid Model

- **Push Supplier/Push Consumer**: Push suppliers *push* events onto the channel, which in turn *pushes* them out to consumers.

- **Push Supplier/Pull Consumer**: Push suppliers *push* events onto the channel, and pull consumers *pull* events (or try to pull) from the channel when they need one.

- **Pull Supplier/Push Consumer**: The event channel *pulls* events from the Pull Supplier, and then it *pushes* these events out to the Push Consumers.

- **Pull Supplier/Pull Consumer**: The Pull Consumers *pull* events from the event channel, which in turn *pulls* them from the Pull Suppliers.