Object – Oriented Design with UML and Java

Part XVI - Architecture

Copyright © David Leberknight

Version 2023

Architecture

- Enterprise Strategy and Budget
- Major Infrastructure / Cloud or Hybrid
- System Boundaries / Bounded Contexts
- Staffing and Team Topologies
- Integration with and Transformation away from Legacy Systems
- Maintenance / Operational Costs
- Networks and Security
- Programming Language(s) and the Technology Stack
- Fault Tolerance / SLAs / Resilience
- Capacity Planning / Scalability / Load Balancing
- Evolvability / Loose Coupling
- Testability
- Infrastructure as Code
- Containers (Docker - Kubernetes)

Architecture

The architecture phase is where a project’s managers, designers and advisors apply their collective ***experience*** to make long-term decisions.

An indication of a good architecture is that the major subsystems and interfaces remain stable over time.

Architecture patterns are more ***strategic*** than ***tactical*** design patterns.

- Plan for change.
- Separate concerns.
- Plan to develop iteratively.
- Consider function and form.
- Think: services and loose coupling.
- Balance economic and technical constraints.
- Keep it simple, and use the right tool for the right job.

Time to Market

There are conflicting forces that plagues most software projects:

- the need to develop software quickly and cheaply.
- the need to develop high-quality software.

The costs associated with a "bad" first deployment can be extremely high.

- From the date of first delivery, the project has a "bad" legacy system (ouch!).
- But "time to market" is a powerful force.

We use a "pattern" to resolve these conflicting forces… ***iterative development***.

But even with iterative development, business decisions regarding the system architecture are usually made early in the project; and these decisions are often hard to revisit, as they tend to involve expensive hardware purchases, expensive database licenses, expensive consultants, etc…

Frameworks

A ***Framework*** is a set of classes with well defined collaborations, where many abstract classes are designed to be specialized for each (re)application.

Roles are defined, players may change.

- Some design decisions have been made and can’t easily be changed.
- Might make use of metadata for dynamic configuration, often using XML.
- Excellent code reuse, with caveats:
  - It takes a lot of effort to design a framework that can be (re)used by many different kinds of applications; most frameworks are targeted towards specific kinds of applications.
  - It usually takes multiple iterations over several different applications to arrive at a general, useful and intuitive set of framework classes.
  - Frameworks typically employ “White Box” reuse - you often must know how the framework classes work in order to extend their functionality; such extensions often involve subclasses of existing framework classes.

Frameworks

Examples:

- A Nintendo game system, with a plug-in cartridge for each game.
- The Java AWT and Swing, for doing GUIs in Java.
- The Spring Framework, a Java EE container. It’s good. Check it out.

- Enterprise Java Beans (EJB3), for fast, scalable, and secure servers.
- Google's Guise for dependency injection.
- JBOSS – Another free, open source, JMS & Java EE implementation.
- Microsoft’s .NET

This stuff evolves rapidly, professionals should constantly learn and adapt.

Example Framework: The Java AWT

- Provides infrastructure to detect user events and notify registered *observers*.
- Provides infrastructure that makes applets and windows possible.
- Provides infrastructure that calls update() / paint() when necessary.
- Provides a robust set of graphics classes.
- Provides a decent set of widgets, layout managers, and other components.
- Provides the ability to create custom components.
- Provides numerous other helper and support classes to make life easier.

A Button widget knows how to redraw itself as having been pushed; the application knows what to do about the Button having been pushed.

JavaScript / TypeScript UI frameworks are more commonly used today.

Frameworks

The goal is to separate (decouple) the framework from the business objects as much as possible, to provide flexibility in light of the following:

- with multiple *views* of the same *model*, if one view alters something, the other views must get updated immediately and automatically;
- database transactions must contain objects from multiple screens;
- the database "schema" and the object model must vary independently;
- the UI and the object model must vary independently;
- adding or changing a view must not affect other views;
- a direct manipulation UI, such as a CAD editor, must have sophisticated, nested windows into the design, with varying levels of detail;
- multiple applications need to share business objects across the network;
- business rules change quickly in the dynamic marketplace; the software must adapt equally quickly.

Horizontal Layers

Many designs begin with domain analysis and use cases, representing ***vertical*** slices of functionality. A characteristic of a vertical design is that there is little or no code reuse; each slice has its own UI, business logic and persistence code. There is a lot of duplicated effort in such designs, and all programmers need to know about all aspects of the system.

Horizontal services, on the other hand, are part of the reusable infrastructure of a system, and are generally independent of any specific (vertical) functionality.

Horizontal services may be ***layered*** on top of one another:

- For example, a Persistence Layer might be designed on top of a Network Communications Layer, which might be built using TCP/IP.
- The Persistence Layer might also make use of other reusable services, such as a Transaction Manager, and an Error Manager.
- It is common to have “business layers” and “data access layers.”

Horizontal Layers

- Horizontal services help to manage change, as they tend to be immune from changes in the vertical domain functionality. New features can be added more quickly by leveraging these services.
- Horizontal services help to manage complexity when they are designed to be reusable, versatile, and easy to use.

The idea is that classes in one layer only interact with classes in the same layer or adjacent layers, with no cyclic dependencies. This helps to reduce complexity for the programmer, as (s)he need not worry about any other layers. For example, the Persistence Layer programmer need not know about TCP/IP, as the Communications Layer already provides all of the needed services at a higher, more convenient level of abstraction.

Metadata

- Metadata is data about data.
- Metadata can be used to make an application much more flexible by storing data which can be read by the application to dynamically (re)configure itself at runtime.
- Metadata can be used, for example, to describe GUI screens along with the objects which reside on those screens, how the screens are laid out, etc…
- Metadata can describe how objects map to relational database tables.

By using metadata, an application can be reconfigured without touching the source code! A trained user can edit an XML file and the application is instantly reconfigured. No recompile is necessary.

The downside, of course, is the extra time and effort spent designing the metadata itself, along with the corresponding framework, and such things as metadata editors, test harnesses, etc.

Toolkit

A ***Toolkit*** is a set of reusable and well-tested classes or *components* which can be used as implementation building blocks.

Note: some of these toolkits provide “framework classes” for GUI applications:

- Java’s Abstract Window Toolkit (AWT).
- Microsoft Foundation Classes (MFC).
- Java Foundation Classes (JFC) & Swing components.
- The C++ Standard Template Library (STL).
- java.util.concurrent.*, java.io.*, java.net.*, javax.swing.*, etc…
- Xerces XML parser.

Toolkits provide excellent code reuse.

Components

A ***Component*** is a fully tested object that has been designed to be (re)usable and easy to integrate, and should have (some of) the following features:

- A well-specified, standard *interface* for interoperability.
- Configurability of *properties* and *behaviors*.
- Internal and external event handling.
- Security.
- Persistence (within transactions).
- Execution inside a *Container*, such as Java EE or a Web browser.
- A UI which can be manipulated visually at *design / deployment time*.
- Version control.
- Compatibility with one or more of the industry's component models.
- Internationalization (I18N) and Localization (L10N).

Components

A component can be:

- a simple UI *widget*, such as a text-edit or a push-button.
- a *container* for other components, such as a Panel or a Frame.
- a POJO (Plain Old Java Object) - almost any class, really.

Many Java Beans and ActiveX controls can be customized without writing code, at design / deployment time, by specifying values for certain configurable *properties*.

For example, when using an ActiveX control to connect to a legacy database server, upon specifying the database name, a list of tables to select from might appear, all at *design* / *deployment time*.

Internationalization (I18N)

Another important feature for a reusable component, especially one designed to play on the Internet, is *Internationalization* (I18n) and *Localization* (L10n).

- Java provides support for this with *Unicode* (instead of ASCII), and the classes java.util.Locale and java.util.ResourceBundle.
- Third-party tools also exist that provide “virtual keyboards” for typing in other languages. It is possible to integrate such tools to be used by a Bean’s custom PropertyEditor…

Commercial Middleware

Information Week says Middleware is:

"1) a hodgepodge of software technologies;

2) a buzzword;

3) a key to developing Client / Server applications."

Examples:

TCP/IP, CORBA, HTTP, .NET, RMI, MQ, EJB3, JBOSS, ...

- Middleware is the glue between the different application tiers.
- Middleware helps with cross-platform portability issues, load balancing, automatic fail-over, message delivery & queuing, and more…
- Middleware is indispensable for "Enterprise Application Integration" (EAI) projects, which allow multiple applications to share data.

Traditional Two-Tiered Architecture

Client:

- User Interface “presentation layer” (GUI).
- Most, if not all, of the business logic - “FAT”.
- Code to deal with the OS, Network & Server APIs.

Server:

- Shared resources.
- Access to persistent data.
- Centralized security & administration.
- Commonly a Database Management System (DBMS).
- Some business logic if the server is “active”.

- Inflexible - Hard to add new application to share legacy data and repeated business logic... Load balancing? Automatic failover?

Tiered Architectures

First Tier:

- Thin client.
- User Interface “presentation layer.”
- Little or no business logic.
- Multiple types of front ends share the second-tier services.

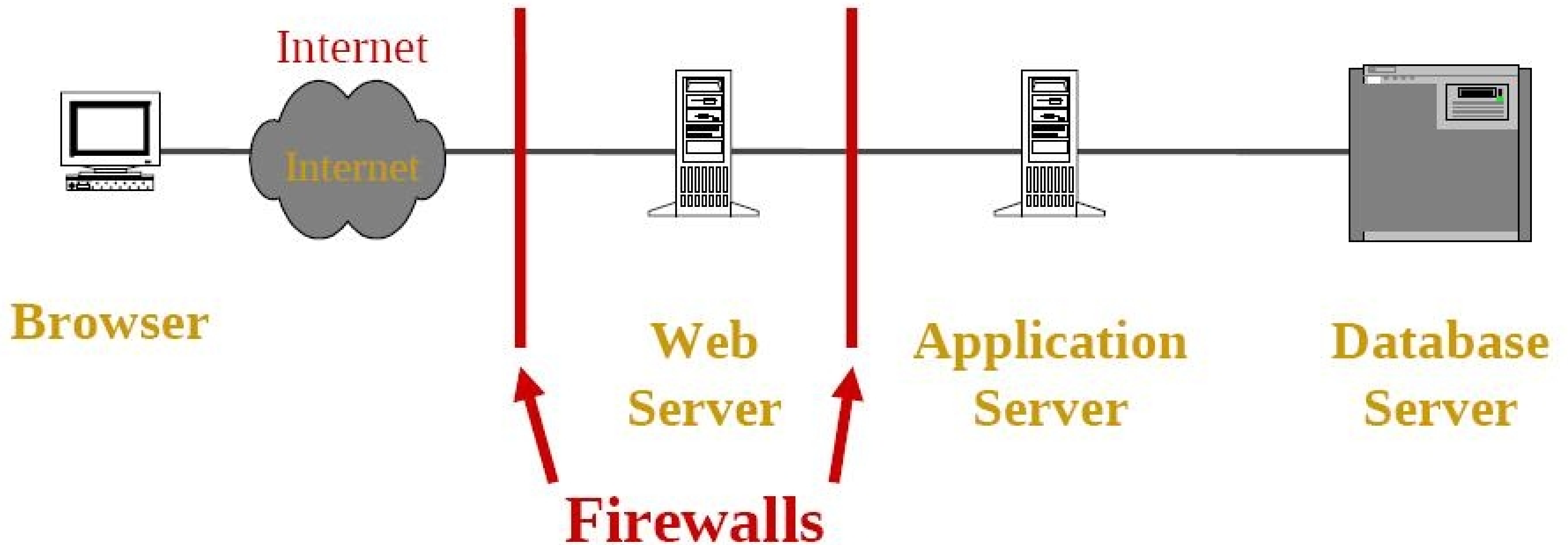
Second Tier:

- Application Server or “active” Database Management System.
- Shared business objects.
- Centralized security & administration.
- Encapsulates the third tier from the first tier.

Third+ Tier:

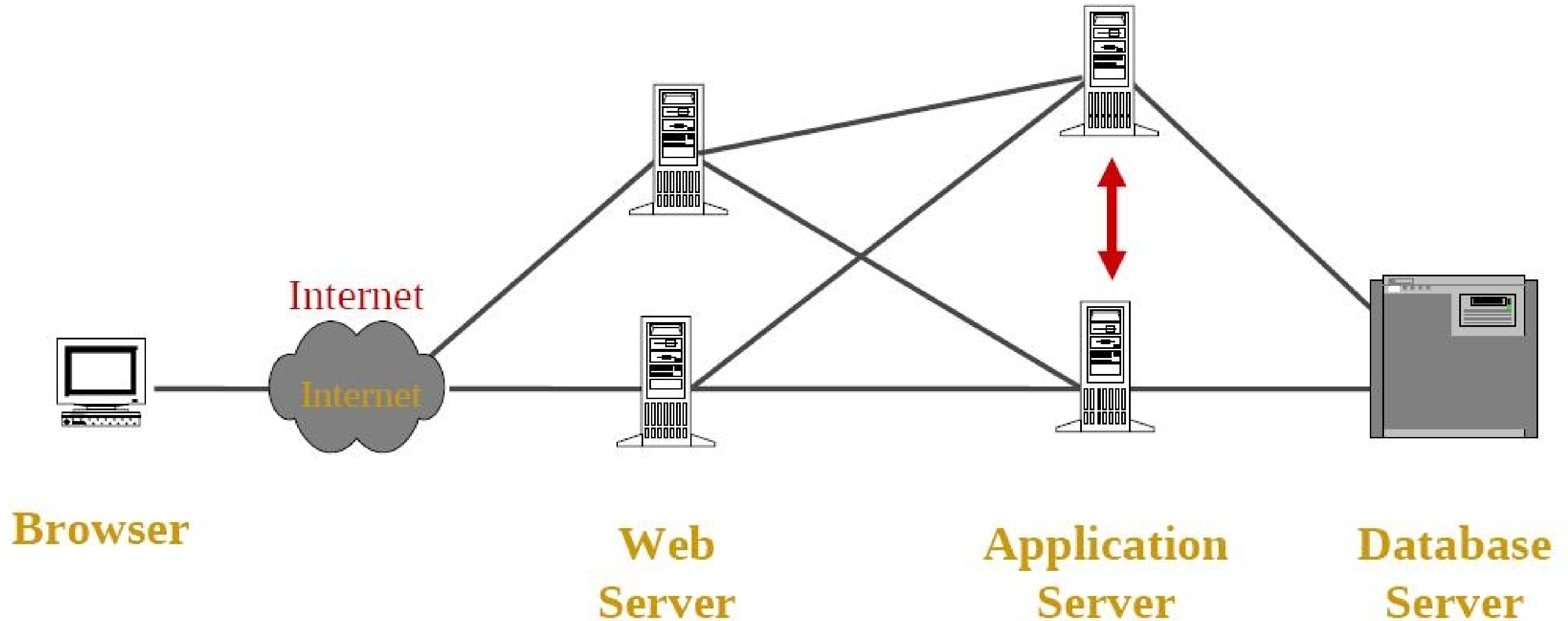
- Data Servers.
- Miscellaneous legacy systems.

Example: Web Server in the "DMZ"



Scalability

Enterprise load-balancing network components are not shown.



Distributed Objects

A ***Distributed Object*** can live anywhere on the net and can be accessed remotely. There are many issues (load balancing, scalability, disaster failover, security, message queuing)...

- The computers involved are autonomous and connected by a network.
- The system must be fault tolerant (highly available?). Crashes happen.
- Communication is done via interfaces (APIs).
- Objects often outlive the programs that created them, and they can survive system crashes (persistence).
- Multiple processes execute in parallel on different machines; this may require synchronization and/or transactions.
- Study ***Cloud Design Patterns*** to start learning advanced topics.

Message-Oriented Middleware

MOM – Message-Oriented Middleware

- Service to carry, route and deliver messages (analogous to email)
- Persistent message queues provide transactional boundary
- Promotes loose coupling and improved testability
- Scalable, reliable, persistent
- Fast growing market for tools and technologies
- Beware marketing hype; consider design before expensive tools
- Enterprise Service Bus (ESB) – design pattern or tool?
- Java Messaging Service (JMS)
- Generally reliable in the presence of network and system crashes
- Every DAD needs a MOM

Enterprise Application Integration

Enterprise Application Integration (EAI) is challenging.

- What is the ***best*** way for a company to integrate their systems?
- This is a huge topic, beyond the scope of this course.

Message-Oriented Middleware can help

- Beware of tool-driven architecture (“marketecture”).
- The principles of good design still apply.

Service-Oriented Architecture (SOA) and ***Micro-services*** are growing quickly.

- A service should be independently testable and deployable.
- Strive to develop a simple and generalizable API.

Web Services

Provide a **Service**.

- ***SOAP*** is a protocol for exchanging XML messages over HTTP (HTTPS). SOAP uses the Envelope / Letter pattern. The SOAP header is used for transport (the envelope). The bulk of the message is the letter (in XML).
- ***WSDL*** (Web Service Description Language) is a standard XML-based language for SOAP messages.
- Cross platform portability is achieved thanks to the ubiquitous use of ***XML*** in plain text. Tools exist to help with Java to XML mapping (marshalling).
- A ***Message Queue*** may be added to the architecture to provide persistence and a transactional boundary.
- Consider also: ***REST*** (a good way to provide an API over HTTP(S))
- See also: ***GraphQL***
- Consider ***Scalability*** as part of your architecture.

The Broker Architecture Pattern

The “Broker Architecture Pattern” provides:

- A “single system image” achieved with a local *Proxy* for remote services.
- Message forwarding, data marshaling, exception propagation, etc...
- Facilitates the implementation of a middle tier in a N-tier architecture as a central site for business rules and/or common data processing with a language/platform/OS independent interface.
- Encapsulation of many complex implementation details.
- Run time metadata describing every server interface.
- Other services (naming, transactions, encryption, …)

Examples:

- Java’s RMI (Remote Method Invocation)
- CORBA – an early standard for interoperability across platforms
- Microsoft’s .NET

Fine vs. *Coarse* grained interfaces

Fine-grained distributed interfaces:

- The client must understand many server-side classes to use the service.
- Small changes to an interface will ripple throughout multiple applications.
- Possible performance problem with a proliferation of distributed object references.

Vs. ***Coarse-grained:***

- Use of the *Façade* design pattern.
- Service oriented.
- Design the interface(s) to batch multiple operations (fewer distributed calls).
- Ideally, the server manages its own objects (no distributed reference counting).

Synch. **vs.** ***Asynch.*** **messaging**

Synchronous method calls:

- Like a phone call (blocks until the method returns).
- Simple and fast... but what if the server is down?

Vs. ***Asynchronous:***

- Like sending email (an event or a message).
- Supports publish/subscribe & asynchronous callback design.
- Use a persistent message queue for reliability.

Stateful vs. *Stateless* Servers

Stateful servers:

- The server maintains contextual information (state) about the client's on-going operation, even across system crashes if so designed.
- It is not necessary to repeat security checking for each distributed call.
- Often used with browser cookies that maintain session IDs.
- Decreased reliability and harder to scale.
- Java EE frameworks such as Spring MVC offer great support.

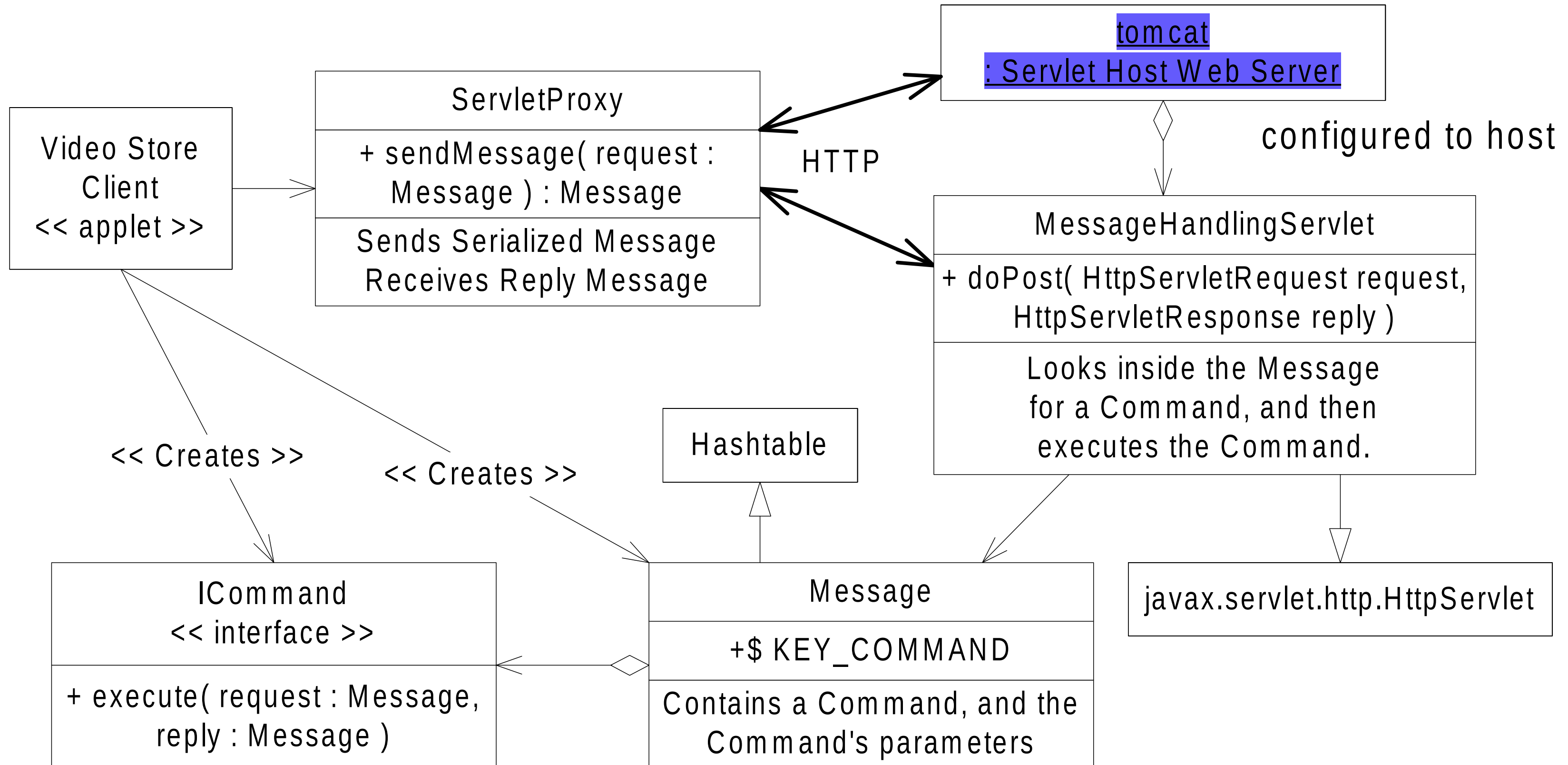
Vs. ***Stateless:***

- All of the necessary data for the operation must be supplied on every call.
- Often used with course-grained distributed interfaces (one transaction per call).
- The server does not maintain client-specific information, easier to scale.

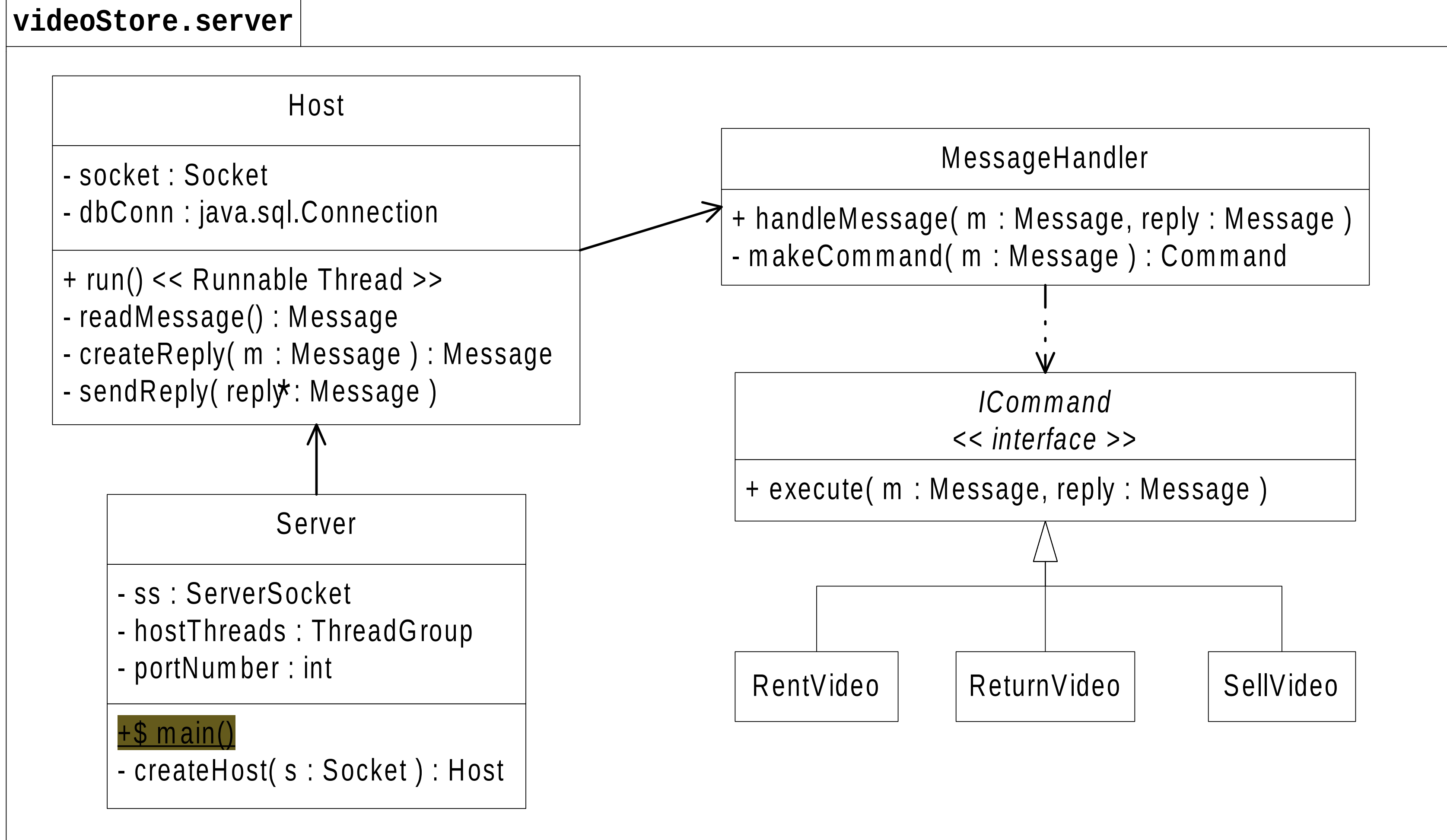
The Object Web

- Move behavior not just data.
- Content distribution networks.
- Platform transparency.
- Components galore.
- Services such as persistence, transactions, encryption, lifecycle, ...
- SOAP and REST
- Service-Oriented Architecture (SOA)
- Use your imagination and make it so.
- Now with the Cloud!!
- ***Cloud Design Patterns*** are beyond the scope of this course.

Applet - Servlet Messaging



Video Store Server



Roles, Experience, Expertise

An ***architect*** is someone who can design the entire solution for a complex, distributed application. This person is an expert in the latest technologies and vendor-supplied tools. The architect works with managers and analysts to determine the hardware, software, and third-party components that meet the business requirements.

A ***designer*** is someone who is expert in object-oriented design and a programming language such as Java. This person will work within the defined architecture to create software components.

A ***programmer*** codes the designer's design.

- These 3 roles can all be held by 1 person.
- Further discussion of Architecture is beyond the scope of this course.