Object - Oriented Programming & Design

Part XVI - Distributed Architecture

Architecture

Software ***Architecture*** is a collection of important and/or long-term strategic decisions, and is the basis for managing:

- major infrastructure, interfaces & system boundaries, estimation, staffing, build or buy decisions, tools, technology, integration with legacy systems, future extensions, maintenance costs, security, error handling, logging, programming language, DBMS, fault tolerance, team structure, test & deployment planning, robustness, flexibility, reusability ...

Plan for change. Plan to develop iteratively. Consider function and form. Separate concerns. Balance economic and technical constraints.

The architecture phase is where a project's managers, designers and advisors apply their collective ***experience*** to make these important decisions.

An indication of a good architecture is that the major subsystems and interfaces remain relatively stable over time. Also, a good architecture should be relatively simple and approachable.

Architecture

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 the first delivery, on into the future, developers have to deal with a “bad” legacy system (ouch!). If managers understand this, they will often give more time to complete the project. But “time to market” is another powerful force that pushes many business managers the other direction.

We might 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 classes are abstract and 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.
- Enterprise Java Beans (EJB), for building fast, scalable, and secure servers.
- The Spring Framework integrates with EJB 3.0. It's good. Check it out.
- JBOSS – Another free, open source, JMS & J2EE implementation.
- Microsoft's .NET

Example Framework: The Java AWT

- Provides infrastructure to detect all user events and notify all registered *observers* (1.1+ event model). (The 1.02 event model works slightly differently, ala the *Chain of Responsibility* design pattern.)
- 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 UI components.
- Provides the ability to create custom components by extending existing ones.
- 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.

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.

Example Workflow Framework

A "workflow" can be defined as a user task where all of the data collected from some set of screens must all go to the database within a single transaction.

- This requires an object which understands the sets of screens that constitute workflows, and which persistent object(s) reside on each screen.
- The framework should encapsulate all interactions with the database because it alone controls transactions.
- If a transaction fails (perhaps due to deadlock) it should be retried.

A common problem is that when a GUI screen gets instantiated, it is expected to "know" certain information from the previous screen.

- Our framework should facilitate the passing of parameters as objects from one screen to another.
- It should also manage the objects' lifetime and (in C++) memory.

In Java, look at the Spring framework.

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.

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 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 a Web browser.
- A UI which can be manipulated visually by a developer at *design / deployment time*.
- Version control.
- Compatibility with one or more of the industry’s component models.
- Internationalization (I18N) and Localization (L10N).

Components

There are two leading standard component models on the market today. They use slightly different design conventions.

- Java Beans
  - Runs inside a Java virtual machine on virtually all platforms.
  - Enterprise Java Beans (EJB) is not the same thing.

- OLE (Object Linking and Embedding) ActiveX Controls
  - A.K.A. COM objects
  - A subset of Microsoft's .NET technology

Just as integrated circuit technology revolutionized the chip industry with hardware components, it is expected that components may "revolutionize" software ...

The goal is "off-the-shelf" - "Black Box" component assembly.

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, EJB, JBOSS, ESB, ...

- 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?

Three - Tiered Architectures

First Tier:

- Thin client.
- User Interface “presentation layer.”
- Java applet, Rich Internet Application (RIA), or a stand-alone application.
- Little or no business logic.
- Multiple types of front ends share the second tier services.

Second Tier:

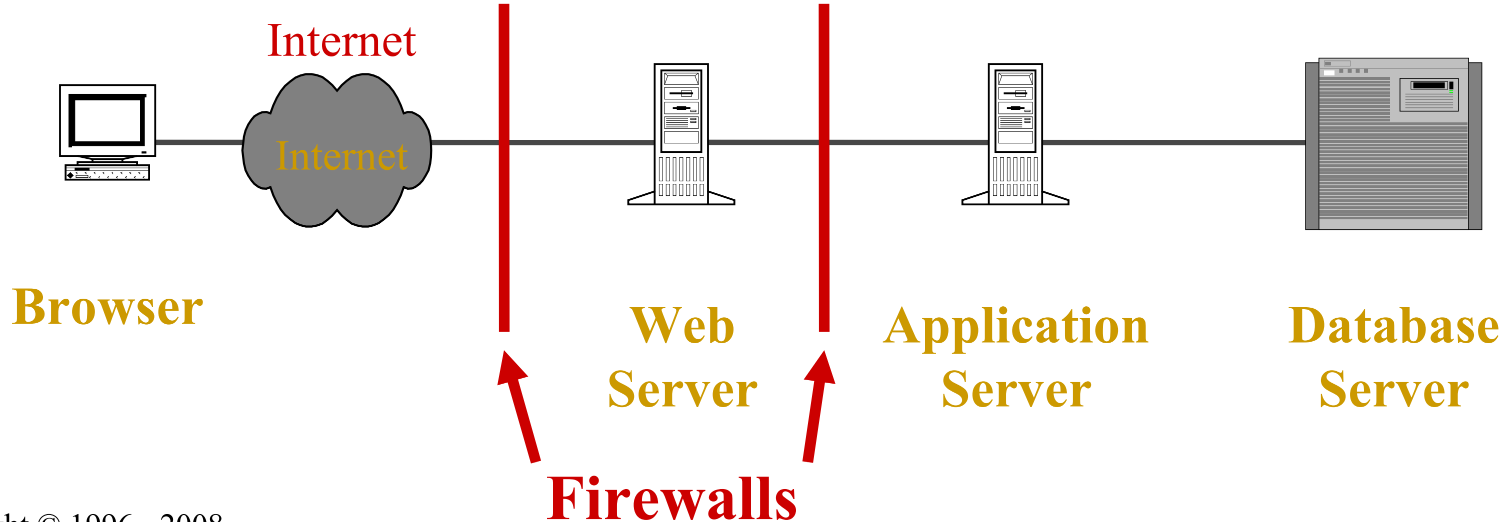
- Application Server or “active” Database Management System.
- Shared business objects.
- Centralized security & administration.
- Encapsulates the third tier from the first tier.

Third+ Tier:

- Data Servers.
- Miscellaneous legacy systems.

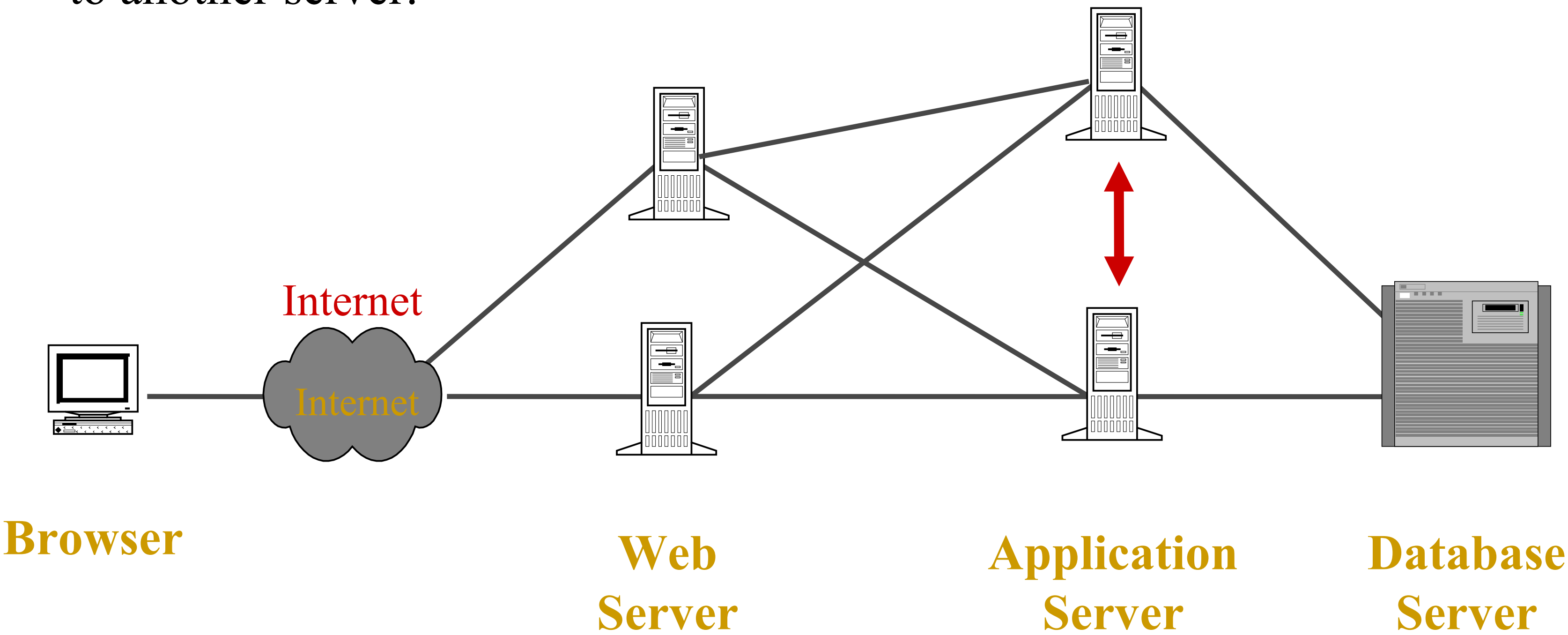
Tiered Architectures

- Browser supports “thin” UI.
- Web Server lives in a semi-secure area of network (the DMZ).
- Application Server holds the business logic & short-term data storage.
- Database Server provides long-term data storage.
- Business logic & data storage inside secure corporate network.



Tiered Architecture Scalability

- Web page and business logic services scale independently of each other.
- Load may be automatically balanced between Application Servers.
- If any server crashes, client sessions may be able to automatically switch to another server.



Distributed Objects

A ***Distributed Object*** can live anywhere on the net and can be accessed remotely. There are many issues (to name a few: load balancing, automatic failover, security, messaging and 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 with dynamic binding.
- 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.
- Use of the *Proxy* design pattern to reference objects remotely.
- Use of object replication (passing objects “by value”) to get performance gains, (fewer distributed calls) but copies get out of sync.
- Use of ORBs, MOM, Enterprise Java Beans, and other technologies.

Message Oriented Middleware

MOM - Message Oriented Middleware

- Service to carry, route and deliver messages (analogous to email)
- Asynchronous event model, based on publish-subscribe design pattern
- Persistent message queues provide transactional boundary
- Promotes loose coupling
- Scalable, reliable, persistent
- Fast growing market for tools and technologies
- Enterprise Service Bus (ESB)
- 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. The technologies continue to evolve. Multiple applications, multiple hardware platforms, multiple programming languages and operating systems.

Corporate data should be kept in synch, and always available in a timely manner. Business rules should be changed in exactly one place.

A pattern: ESB (Enterprise Service Bus).

Another pattern: Publish-Subscribe Channel.

- CORBA (older, widespread) – Provides platform, location, and programming language transparency.
- SOA (newer, growing quickly) – Service-Oriented Architectures structure enterprise applications as Web Services.

Web Services

One system on a network is a service provider.

Its accessible data and/or behaviour is known as a ***Web Service***.

It describes its services using ***WSDL*** (Web Service Description Language), a standard XML-based language.

A service requestor may use ***XML-RPC*** (Remote Procedure Call), ***SOAP*** (Simple Object Access Protocol), or other technologies to access the 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).
- Cross platform portability is achieved thanks to the ubiquitous use of ***XML*** in plain text. Tools exist to help with Java to XML mapping.
- A ***Message Queue*** may be added as part of an Enterprise Service Bus (ESB) to provide persistence and a transactional boundary.

CORBA

CORBA - Common Object Request Broker Architecture

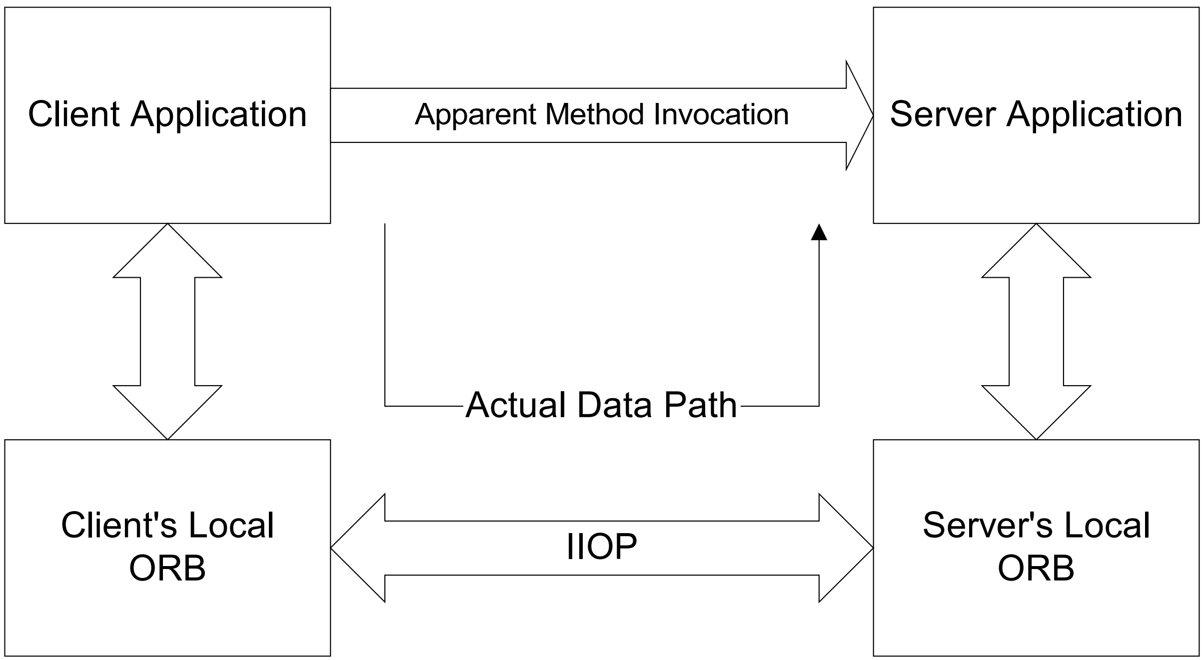
- Object Request Brokers (**ORB**s) communicate with each other using the Internet Inter-ORB Protocol (**IIOP**).
- Each ORB has an ***Interface Repository***, based on a standard Interface Definition Language (**IDL**), similar in syntax to Java interfaces.

Using a CORBA ORB, an application can access distributed objects (components) without worrying about where they are (location transparency) - ORBs also provide platform and programming language transparencies.

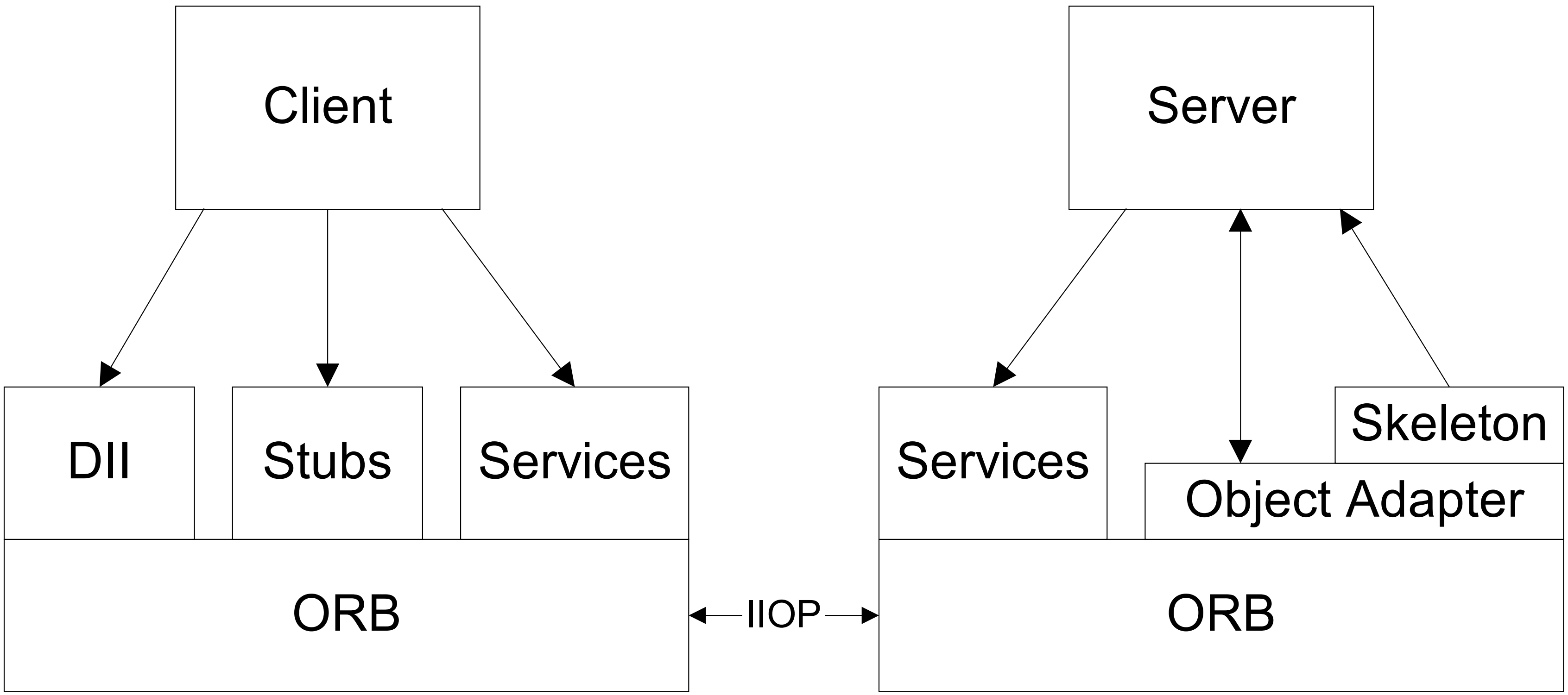
- The CORBA standard comes from the Object Management Group (**OMG**), which has 800+ companies.
  - Notable exception: Microsoft.

CORBA

- Allows objects to dynamically discover each other and interact across machines and OSs.
- There are many commercial CORBA ORBs on the market.



CORBA



Client
Server
DII
Stubs
Services
Services
Skeleton
Object Adapter
ORB
IIOP
ORB

CORBA

Interface Repository - A distributed database containing metadata that describes interfaces; all services are self-describing.

Dynamic Invocation Interface (**DII**) - Allows run-time inspection of server interface metadata, and allows the interface methods to be invoked *dynamically* (similar conceptually to Java reflection).

Stub - A local ***Proxy*** for a remote server object; stubs are machine-generated code, compiled from the IDL. This is the *static* alternative to DII.

Skeleton - An ***Adapter*** for the server-side code. The stubs and skeletons together have the code for *marshaling* data from ORB to ORB over IIOP.

Object Adapter - Responsible for receiving incoming service requests, instantiating and managing server objects. What about server-side garbage collection?

- It is common to wrap stubs and skeletons using the *Adapter* design pattern.

The Broker Architecture Pattern

The "Broker Architecture Pattern" provides:

- A "single system image" to clients, achieved with a local *Proxy* for remote components and 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, …)

Other examples:

- Microsoft's DCOM (Distributed COM)
- Java's RMI (Remote Method Invocation)

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... 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 between crashes.
- It is not necessary to repeat security checking for each distributed call.
- Often used with fine-grained distributed interfaces.
- Decreases reliability; more distributed calls; more likely to fail.

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).
- If no two transactions share state, the threading model is much simplified, and it is easier to provide scalability / load balancing.

The Object Web

Java provides:

- The ability to move behavior around.
- Simplified code distribution.
- Platform and OS transparency for behavior.
- JavaBeans and EJB components.

CORBA provides:

- Services such as naming, persistence, transactions, encryption, lifecycle, ...
- Location, Platform, OS & Programming language transparency.

Web Services provides:

- XML based standards (including SOAP)

Messages & Commands

- A common design pattern in distributed systems is for the client to send "messages" to the server, where the message contains a ***Command***.
- In this example design, **`Message`** is a subclass of **`java.util.Hashtable`**, so it can contain arbitrary information (everything of interest must implement **`Serializable`**). Messages have one required key: **`KEY_COMMAND`**, whose value indicates the name of the concrete **`Command`** subclass. Other entries in the **`Hashtable`** specify the rest of the **`Command`**'s parameters.

```
class Message extends Hashtable implements java.io.Serializable {
  public final static String KEY_COMMAND = "command";
  public final static String KEY_ERROR = "error";
  public Message( String commandName ) {
    put( KEY_COMMAND, commandName );
  }
  // "convenience methods" : getCommandName(), createReply(), . . .
}
```

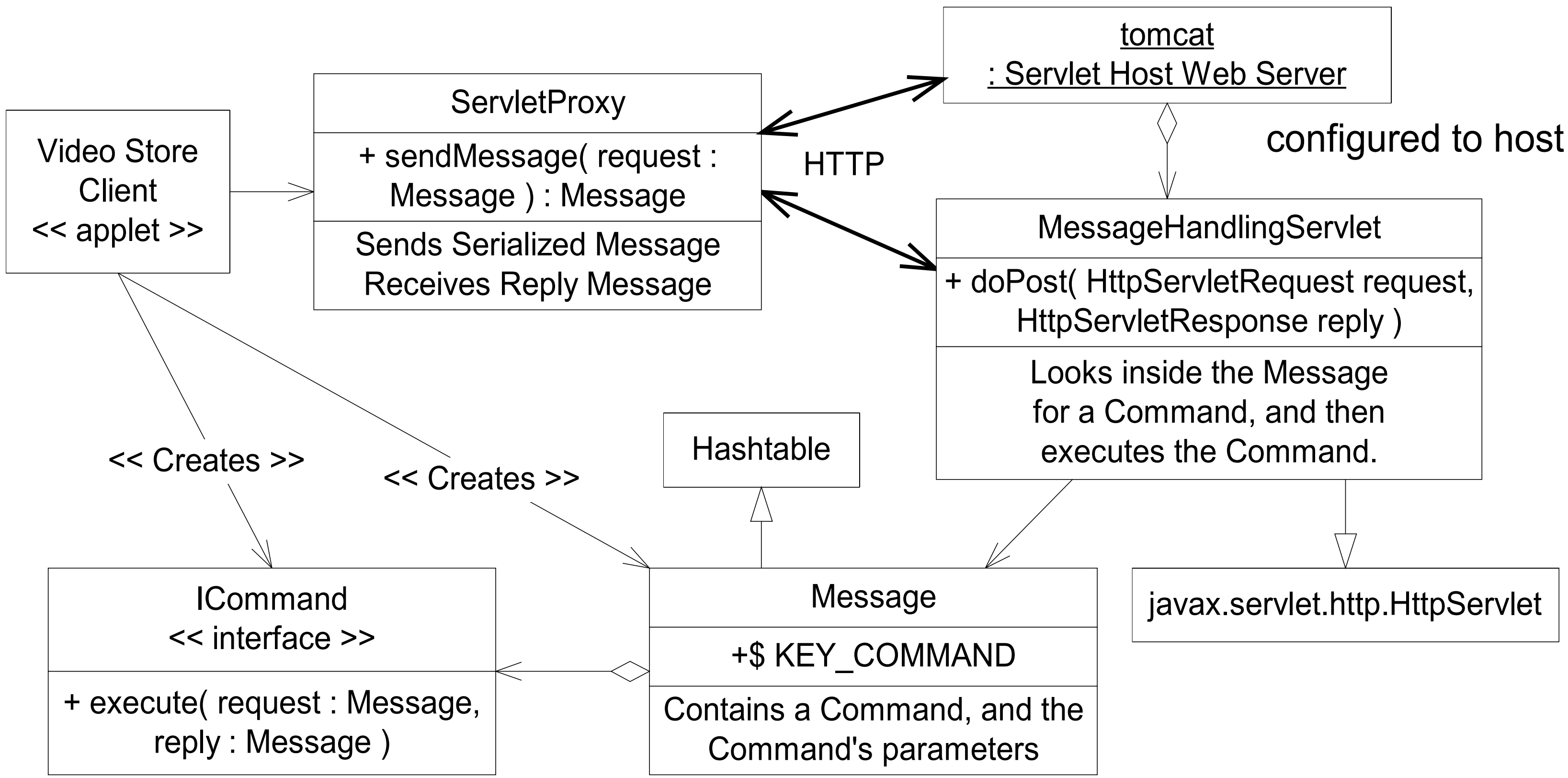
Messages & Commands

```
public interface IVideoConst {
  final static String COMMAND_RENT_VIDEO =
                      "cs4448.videoServer.RentVideoCommand";
  final static String KEY_CUSTOMER = "customer";
  final static String KEY_VIDEO_NAME = "video name";
  final static String KEY_VERIF_CODE = "verification code";
  // . . .
}

// Client-side code. . .
Message request = new Message( IVideoConst.COMMAND_RENT_VIDEO );
request.put( IVideoConst.KEY_CUSTOMER, customer );
request.put( IVideoConst.KEY_VIDEO_NAME, "Java Junkies" );
Message reply = ServletProxy.sendMessage( request );
```

Messages & Commands

```
package cs4448.server;
public class RentVideoCommand implements ICommand
{
  public void execute( Message request, Message reply ) {
    try {
      Customer c = (Customer)request.get( IVideoConst.KEY_CUSTOMER );
      String v = (String)request.get( IVideoConst.KEY_VIDEO_NAME );
      String verificationCode = rentVideo( c, v );
      reply.put( IVideoConst.KEY_VERIF_CODE, verificationCode );
    }
    catch( Throwable t ) {
      reply.put( Message.KEY_ERROR, "Video NOT rented! " + t );
  } }
  private String rentVideo( Customer c, String videoName ) { . . . }
}
```

Applet - Servlet Messaging



Tomcat's role in example design

- An untrusted applet may communicate to the server from where it originated. For this to be useful, we need some process to be running on the server. Perhaps the simplest way to do this is to use Tomcat, the servlet container used as the *Official Reference Implementation* for the **Java Servlet** and **Java Server Pages** (JSP) technologies.
- Refer to: **`http://jakarta.apache.org/tomcat/`**
- We only need one servlet: **`cs4448.server.MessageHandlingServlet`**.
- Once Tomcat is installed and configured, access the servlet through a URL: **`http://www.MyHost.com/cs4448/servlet/MessageHandlingServlet`**
- The **`javax.servlet.http.HttpServlet`** API uses a request / reply protocol, which is ideally suited for our purposes; applets make requests to the servlet and receive synchronous replies.
- The client-server communication code requires *no maintenance*. **`Hashtable`** works very well as a general purpose *container* for **`Command`** parameters.
- Server functionality may be *extended* by adding new **`Command`** subclasses.

ServletProxy

```
package cs4448.message;
public class ServletProxy {
  public static Message sendMessage( Message request ) {
    try {
      URL servletURL = new URL(
      "http://localhost:8080/cs4448/servlet/MessageHandlingServlet" );
      InputStream in = sendHttpPostMessage( request, servletURL );
      ObjectInputStream ois = new ObjectInputStream( in );
      return (Message) ois.readObject();
    }
    catch( Throwable t ) {
      Log.error( "ServletProxy: Check Tomcat logs.", t );
      Message reply = request.createReply();
      reply.put( Message.KEY_ERROR, "Server Down " + t );
      return reply;
 } }
```

ServletProxy

```
public static InputStream sendHttpPostMessage( Serializable msg,
                               URL servletURL ) throws Exception {
    URLConnection conn = servletURL.openConnection();
    conn.setDoInput( true );
    conn.setDoOutput( true );
    conn.setUseCaches( false );
    conn.setRequestProperty( "Content-Type", "java-internal/" +
                             msg.getClass().getName() );
    ObjectOutputStream out = new ObjectOutputStream(
                             conn.getOutputStream() );
    out.writeObject( msg );
    out.flush();
    out.close();
    return conn.getInputStream();
} } }
```

MessageHandlingServlet

```
package cs4448.server;
import javax.servlet.http.*;
public class MessageHandlingServlet extends HttpServlet {
  public synchronized void doPost( HttpServletRequest req,
                                   HttpServletResponse res ) {
    try {
      // The HttpServletRequest contains a Message
      ObjectInputStream ois = new ObjectInputStream(
                                 req.getInputStream() );
      Message request = (Message) ois.readObject();
      ois.close();
      // The Message contains a Command
      String cmdName = request.getCommandName();
      Class c = Class.forName( cmdName );
      ICommand command = (ICommand) c.newInstance();
```

MessageHandlingServlet

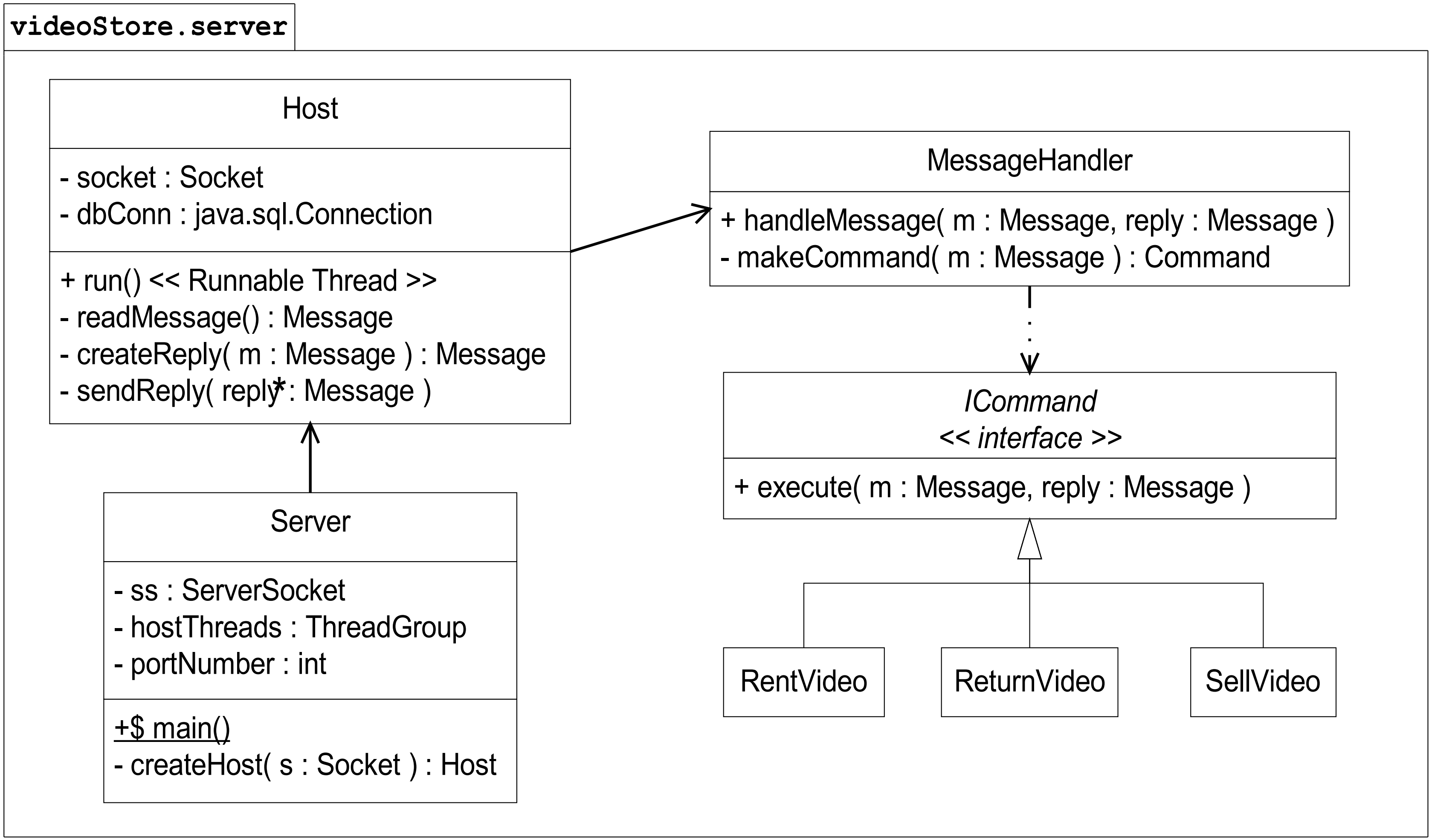
```
      Message reply = request.createReply();

      command.execute( request, reply ); // !!!

      // Serialize the reply back to the client
      ObjectOutputStream oos = new ObjectOutputStream(
                                   res.getOutputStream() );
      oos.writeObject( reply );
      oos.close();
    }
    catch( Throwable t ) {
      Log.error( "MessageHandlingServlet ERROR ", t }
} } }
```

Video Store Server

To help understand what *Tomcat* does for you, consider "rolling your own" ...

- The **`Server`**'s **`main()`** method creates a **`java.net.ServerSocket`** which listens on a given port for a client connection request. For each such request, it makes a new **`Thread`** to run a new instance of **`Host`**, initialized with a **`java.net.Socket`**.
- The **`Host`** can ask the **`Socket`** for a **`java.io.InputStream`** and then *decorate* it with a **`java.io.ObjectInputStream`** from which it can *deserialize* the **`Object`** sent from the client's **`ServerProxy`**. The **`Host`** will *downcast* the **`Object`** sent to be of type: **`Message`**. The **`Host`** then creates a reply and *delegates* further processing to **`MessageHandler`**. From here on out, everything works exactly the same as for **`MessageHandlingServlet`**.
- There are ***many failure scenarios*** that must be dealt with in order for such code to be 'production quality' - fast, robust & secure.
- Check out *Apache MINA*, a framework built using **`java.nio`** to facilitate building this kind of network application.

Video Store Server



Remote Method Invocation (RMI)

Java's RMI is an example of the Broker architecture pattern.

- In CORBA, the Broker is called the ORB; in RMI, it is called **RMIRegistry**.
- RMI also makes use of machine-generated **Stubs** and **Skeletons**, which together encapsulate the code for ***marshaling*** data (in this case using Java serialization).
- Stubs are client-side ***Proxies***; Skeletons are server-side ***Adapters***.
- RMI also uses a Naming service, for clients to find remote services.

MessageHandlingRMI service

1) Define the remote interface: **`IMessageHandlerRMI`** (which must extend the interface **`java.rmi.Remote`**).
2) Create a server-side class to implement the interface: **`MessageHandlerRMI`** (which must extend **`java.rmi.server.UnicastRemoteObject`**) .
3) **`rmic MessageHandlerRMI`** creates **`MessageHandlerRMI_Stub.class`** and **`MessageHandlerRMI_Skeleton.class`**.
4) Run **`rmiregistry`** on the remote host to start the Broker.
5) Put all the class files in the right places.
6) Run **`java MessageHandlerRMI`** on the remote host; the **`main()`** method should call **`Naming.rebind( “Message Service”, new MessageHandlingRMI() );`**
7) The client gets a local interface for communicating with the remote service:

```
IMessageHandlingRMI service = (IMessageHandlingRMI)
  Naming.lookup( RMI_HOST_NAME + “Message Service” );
```

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 to determine the hardware, home-grown 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 the home-grown software components.

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

- Study architectural patterns and best practices.
- Consider the realities of the business (personnel skills and personalities, time to market, licensing fees, other costs, technical risk), …