Java: History and Outlook

Eberhard Wolff
Principal Consultant & Regional Director
SpringSource

Disclaimer

- These are my personal opinions.
- We will do a lot history, little outlook.
- You will learn a lot about high level architecture in Java by its history.
- So in the beginning….

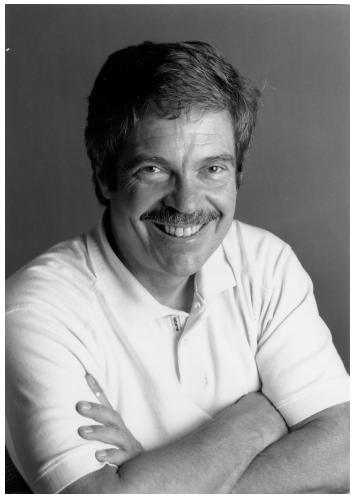
Before Java

- C++ predominant
- Easy adoption from C

- Better but less successful alternatives:
- Eiffel: Statically typed language with advanced type system
- Smalltalk: Clean dynamically typed object oriented language with a VM

“The best way to predict the future is to invent it.”

Alan Kay, Smalltalk Co-Developer



The Beginning…

- Java was originally created for small consumer devices (and later set top boxes etc.)
- Was called Oak

- Simple/ simplistic / good enough language
- Weaker than Eiffel (type system)
- …and probably less powerful than Smalltalk
- Hardware independent

Why was Java successful?

- Easy language with clear migration path from C/C++
- Project shifted focus
- Applets & the Rise of the Internet

- SUN as an alternative to Microsoft
- I meant to say: standards

What remains?
Focus on small devices

- Mobile, embedded
- Especially: phones

- Jini tried this idea again

- And lately: SUN Spots
- For even smaller devices
- i.e just a few chips
- What can you do with that?

What mains?
Set Top boxes

- Blu-Ray is the DVD successor
- Blu Ray disc can be enhanced using Java
- Currently supported on Sony's Playstation 3
- Finally Java is on Set Top boxes…

What remains? Standards

- Standards are still used as a tool for innovation in Java
- Standard and innovation is otherwise considered a contradiction
- …but this is how it started

- Community still looks at standards
- …and does not always judge technologies purely by its value

What remains? Simple Language

- JDK 1.5 / Java 5 introduced some work arounds
- Original Java: Primitive data types (int, long, float, double) are no objects
- …but there are object wrappers
- ...to handle everything uniform

- Workaround: Autoboxing (i.e. a primitive data type becomes an object if need to)

Fun with Autoboxing

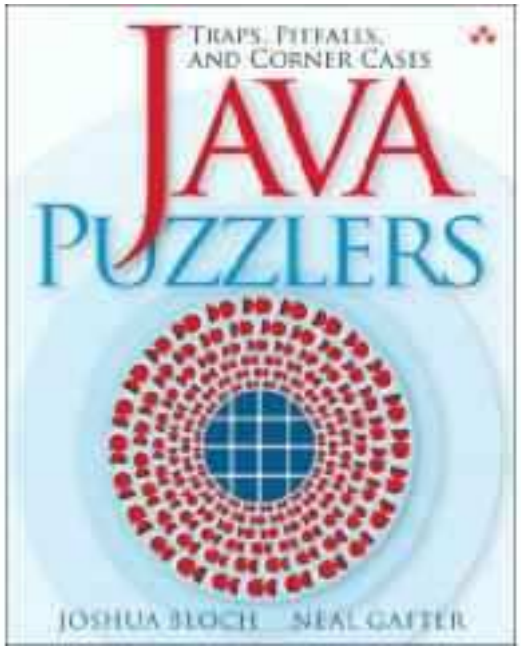
```
publicstaticvoidmain(String[] args) {
  List list = newArrayList();
  Set set = newLinkedHashSet();

for (int x = 0; x < 10; x++) {
list.add(x);
set.add(x);
  }

for (int x = 1; x < 6; x++) {
    list.remove(x-1);
set.remove(x);
  }

System.out.println(list);
System.out.println(set);
}
```

[1, 3, 5, 7, 9]
[0, 6, 7, 8, 9]



Java Puzzlersby
Joshua Bloch, Neal Gafter

What remains? Simple Language

- No parametric polymorphism (type parameter / templates)
- JDK 1.5 Workaround: Changed compiler but same byte code
- So just type checking
- Problem: Limitations for reflection at runtime
- Otherwise a good approach and extension!

What remains? Simple Language

- Several other enhancements in JDK 1.5 (methods with variable number of arguments)
- Workarounds for initial design decisions
- Java is not simplistic any more
- Is it still simple?
- Design goal: Bytecode should remain stable (.NET decided differently)

What remains? Simple Language

- Checked Exceptions

- Default: Exceptions have to be caught or declared to be thrown
- ...except for RuntimeExceptions
- Seemed OK: It forces you to think about error conditions
- But: Java is the only popular language with this concept
- What do you do with a checked Exception?

Checked Exceptions: Output the error somewhere

- Probably hard to find – not in the log file
- Error makes the application just continue

```
try {
   ...
} catch (JMSExceptionjmsex) {
jmsex.printStackTrace();
}
```

Checked Exceptions: Ignore the error

- Error makes the application just ignore & continue
- …and can never be detected

```
try {
   ...
} catch (JMSExceptionjmsex) {
}
```

Checked Exceptions: Wrap the exception

- Lots of pointless code (wrapping)
- Every method will throw `MyException`
- Same effect as a `RuntimeException`: Every method can throw it
- …but you do a lot of wrapping and `throws`...

```
try {
   ...
} catch (JMSExceptionjmsex) {
thrownewMyException(jmsex);
}
```

Take away

- Checked exceptions are (almost) unique to Java
- …and should only be used if really needed

First applications: Applets

- Vision: From an HTML/HTTP based Web to networked servers with rich clients
- Distributed Objects Everywhere!

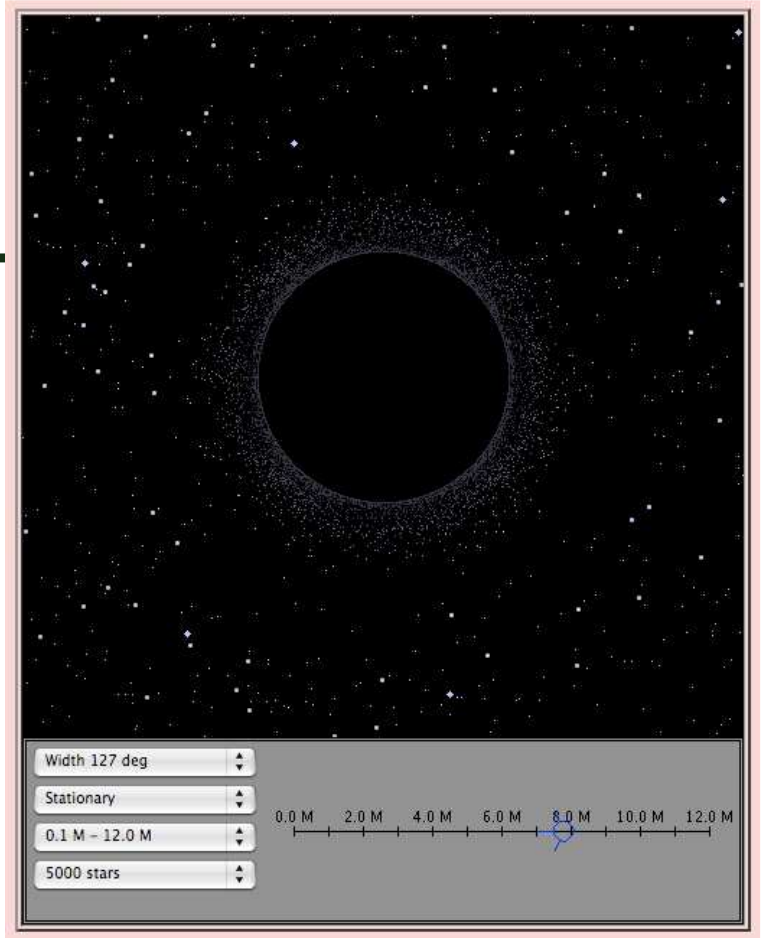
- But:
  - Slow startup and bad performance
  - Incompatible JDKs
  - Often HTML is enough
  - Lots of animation players etc.

Applets: No success

- The vision was right, but…
- …even online Office applications use HTML + AJAX nowadays…
- It gets not more GUI focused
- Much better JavaScript support in browsers
- Java would have been an easier / more sound solution in many cases

Example: Black Hole Applet

- It is hard to even find an Applet nowadays…
- Niche: Graphics with calculation on the client
- But: The *idea* is not dead yet…



http://gregegan.customer.netspace.net.au/PLANCK/Tour/TourApplet.html

What remains? Applets

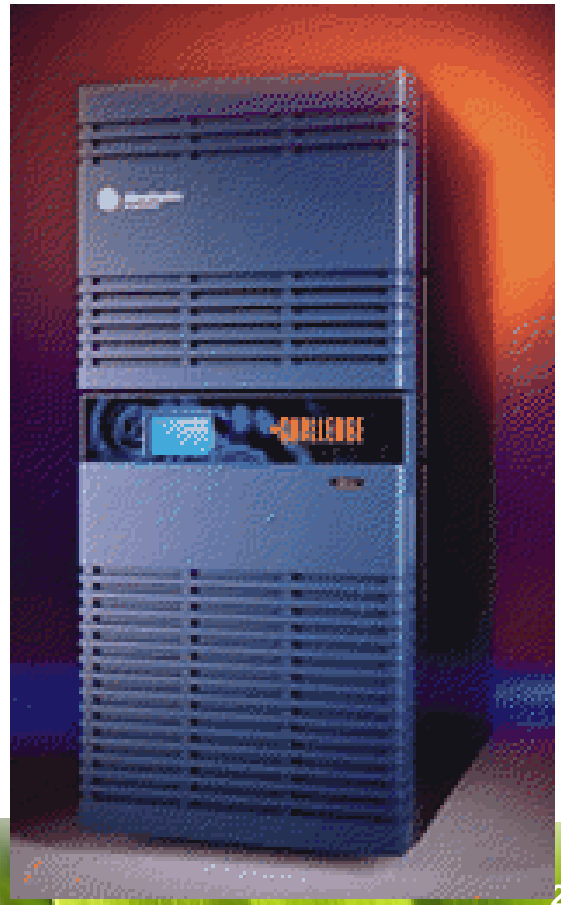
- JavaFX tries to resurrect this idea
- Small JVM + easy development for rich GUIs

- But: Fierce competition
  - Adobe Flash – better adoption than Windows
  - Microsoft Silverlight

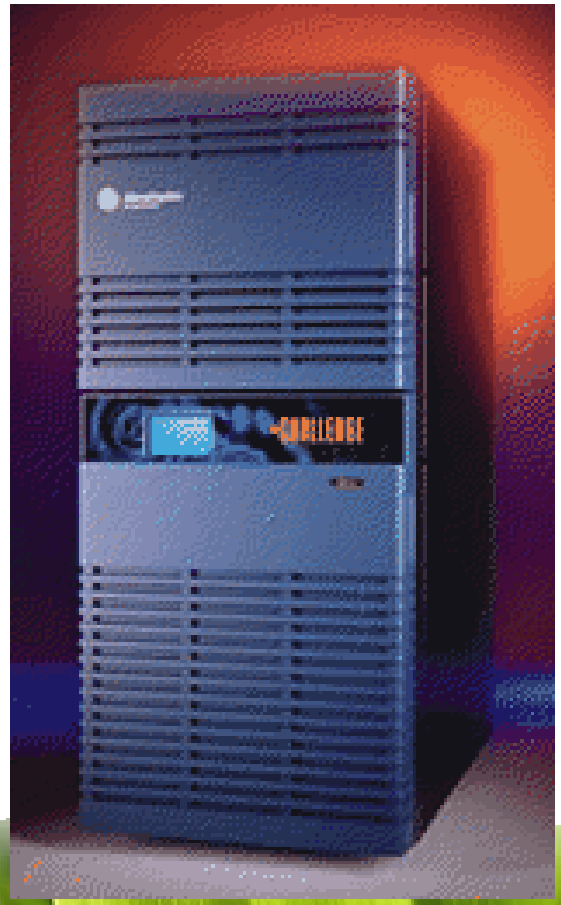
- And: Rather slow development

Server

- So the client side Java did not work out.
- What now?
- Let's try the server!
- Evolution from shared database to shared logic



Server



Before Java

- CORBA/C++ was predominant and *very* complicated
  - No garbage collection
  - Manual thread and instance handling
  - Lots of technology, not too much focus on business logic
  - Many standards for services, some very hard to implement and use
  - Death by committee
- So an alternative was needed

J2EE: Java 2 Enterprise Edition

- Web focus
  - Servlets/JSPs are even older
- But also distributed 2PC transactions, Message Orientied Middleware (JMS)
- Technical solution
- Other approaches also defined core Business Objects and not just technology
- Anyone remember IBM San Francisco?

1999
J2EE 1.2 =
EJB 1.1+
JSP 1.1+
Servlets 2.2+
JDBC 2.0+
JNDI 1.2+
JMS 1.0.2+
JTA 1.0.1+
JTS 0.95+
JavaMail 1.1+
JAF 1.0

J2EE vs. C++/CORBA

- Garbage Collection
- Automatic thread / instance pooling (EJB)
- Components define
  - Thread pooling
  - Instance handling
  - Default decisions as opposed to roll-your-own on CORBA

J2EE

- Tremendous success
- SAP, IBM, SUN, Oracle, BEA
- Original vendors (Orion, WebLogic, …) were bought
- Microsoft had to create .NET

But…

- This is before Hotspot, JIT, optimized Garbage Collection etc.
- Bad performance
- Unstable middleware etc.

- In retrospect the success is surprising

Why did J2EE succeed? User

- Thinking in the Java-Community: "J2EE is a standard by experts, is has to be suited for enterprise problems."
- …and Java is much simpler than what we use at the moment
- Open, standards based
- …and then the middleware company followed
- The next big thing after CORBA

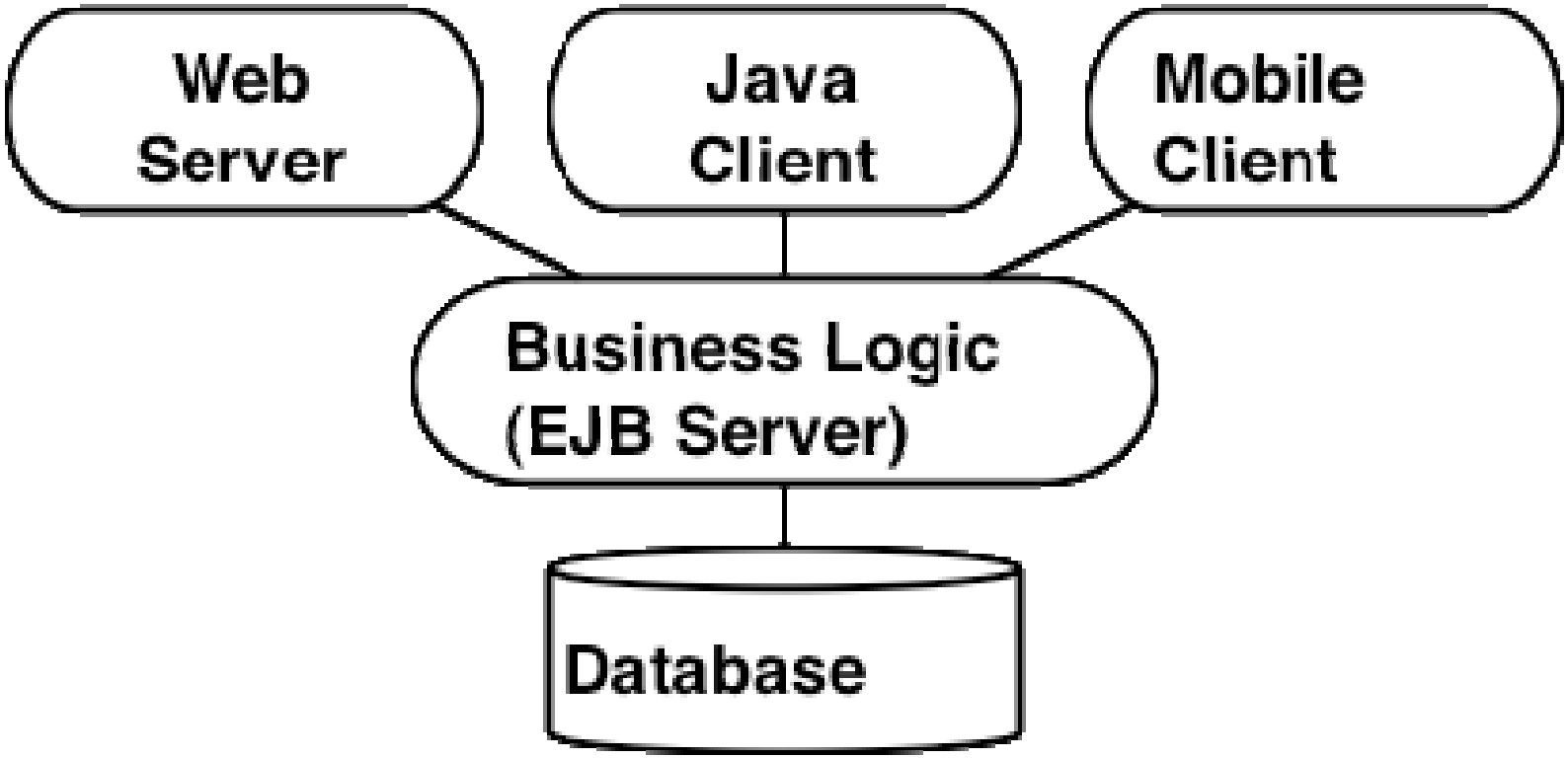
Back to the J2EE disaster

- "Up to when I left [a large consultancy] in 2002, not one of our J2EE projects had succeeded."
  anonymous

- Translate that to the € / $ lost
- (Of course you can blame it on the customers and the process)
- Why did it fail?

Reasons for Failure: Distribution

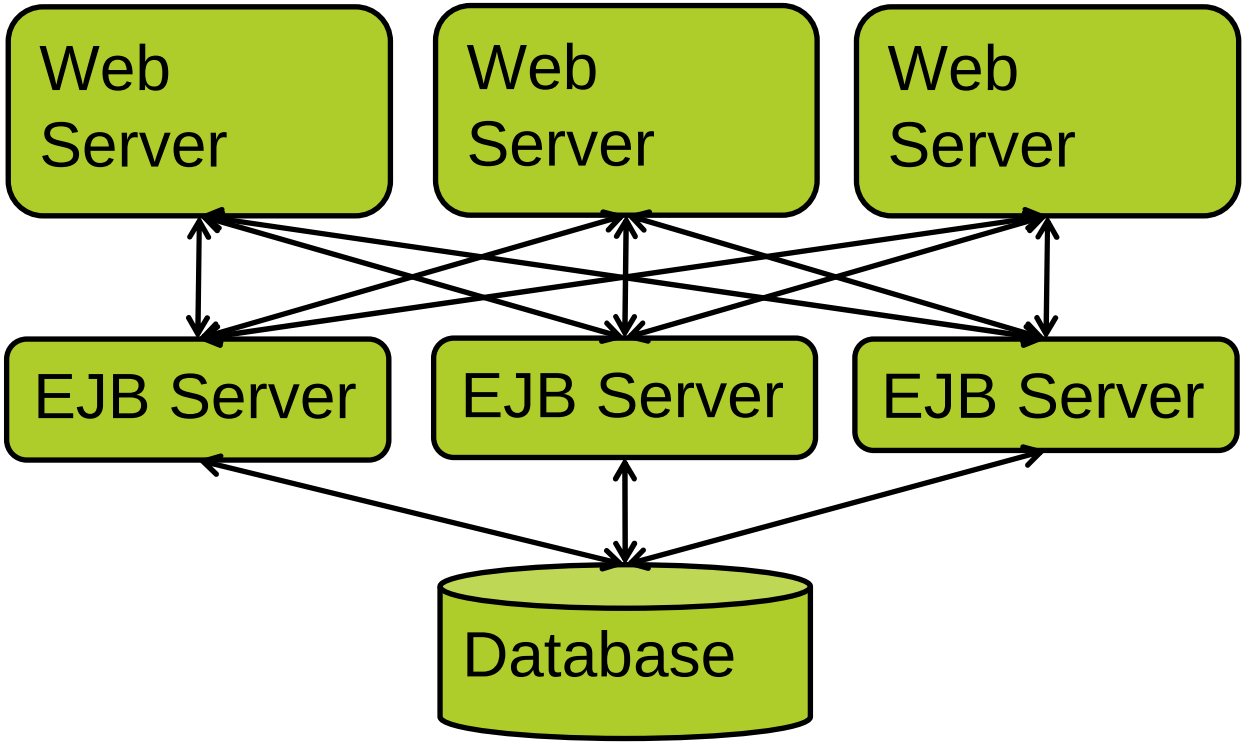
- To access the same logic from different front ends it has to be distributed



The disaster: Distribution

- "You have to be able scale the business logic independently from the web to handle high load."
- So you must have a web server and an separate EJB server
- Note that the logic is just used from the web server, so no different front ends
- All EJBs (1.0/1.1) are remotely accessible

Distribution



"I turn your software scalability problem in a hardware scalability problem."

anonymous J2EE consultant

The disaster: Distribution

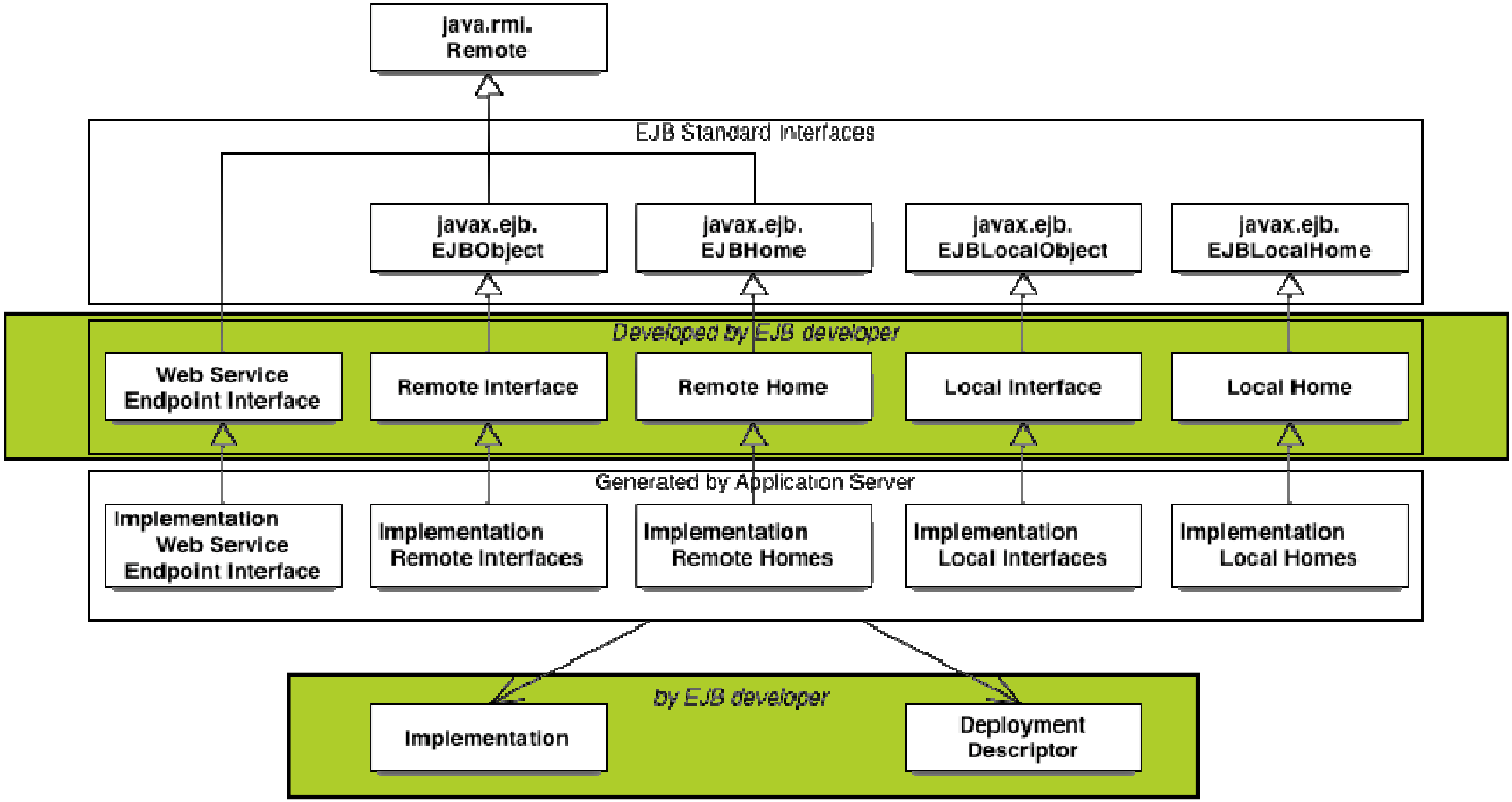
- Complete and dangerous nonsense
- (almost always)
- A distributed call from the web server to the EJB server is orders of magnitude slower than a local call
- The marshalling and demarshalling might use more performance than the logic itself

- "Don't distribute your objects!"
  Martin Fowler

The disaster: Productivity

- An Entity Bean (EJB 1.0-2.1) consists of:
  - Three Java classes / interfaces
  - A deployment descriptor (or part of it)
  - …which is so hard to write that most generate it using XDoclet
- Also hard to understand

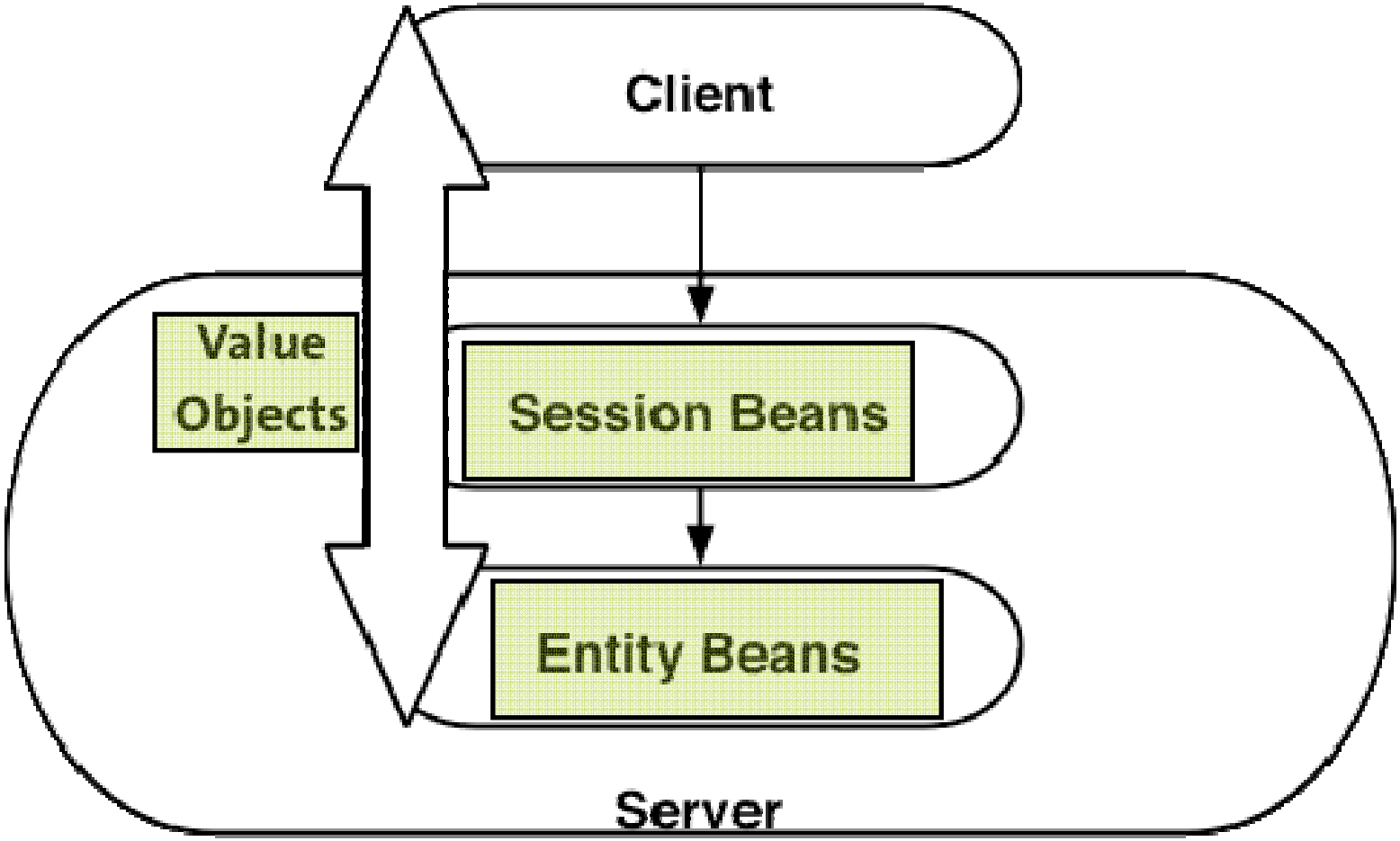
How to write an EJB…



The disaster: Productivity

- Usually also a DTO (Data Transfer Object) is needed
  - to transfer the data to the client / web tier
  - …and of course the data has to be copied into it
  - …so you need also a Stateless Session Bean (three Java classes/ interfaces + Deployment Descriptor)

Productivity



The disaster: Productivity

- All EJBs cannot really be tested without a container
- So people started to wrap POJOs (Plain Old Java Objects) into EJBs
- Even more code
- Not object oriented any more
- Workaround: Generators
- This hurt MDA
- …was often perceived as an EJB work around

The disaster: Persistence

- Persistent objects
- Can be remotely accessed
- Synchronize state with database on each call
- So with a naïve implementation every get/set is a remote call (and a SQL query)
- Now imagine you want to edit some data…
- Reading n objects uses n+1 SQL queries
- 1 Entity Bean = 3 classes (each technology dependent) + XML configuration

- Total usability and performance disaster

But it has to be good for something…

- So you would need coarse grained persistent object
- With concurrent access by multiple users

- Why don't we have that in our applications?
- The standard cannot be wrong, right?
- This concept is still around

It could have been avoided…

- TopLink was around before that (even for Smalltalk)
- NeXT did the Enterprise Object Framework before that

- Problem persists: Standardization committees do not look at prior art

What remains?
J2EE

- J2EE's success is not based on excellent engineering
- J2EE has had very deep trouble and was still a success

- It is unlikely that Java Enterprise will go away
- …as it survived this when it was much weaker.

- People still think that a technology has to work because it is a standard
- …often they don't even question it.

What remains? J2EE

- J2EE has been extended and renamed
- But some concepts are still around
  - EJB very much resembles how CORBA servers are created and "optimizes" for bad garbage collection

What remains? J2EE

- Examples:
  - instance pooling obsolete because of thread safe Singletons and Hotspot VMs
  - two phase commit often unnecessary and slow
  - Still no good security concept: How do you express that a customer can only see his accounts?
  - EJB Security is largely unchanged and can't handle a lot of common issues (instance based security, Access Control Lists, integration with common security technologies…)

Fundamental problems J2EE

- Concept is based on small cluster and client/server
- What about cloud / virtualization / grid?
- Still not solved

- J2EE is invasive i.e. your code depends on the technology
- Hard to test, develop and port
- Object-oriented programming becomes hard: The decline of OO

Early J2EE: Issues

- Performance and productivity problems
- Decline of OO

- But: The predominant platform in Enterprise
- So: What now?

- Fix the issues!

Summary

- Small Devices
- Standards
- Simple Language
- Applets
- Server / J2EE Issues

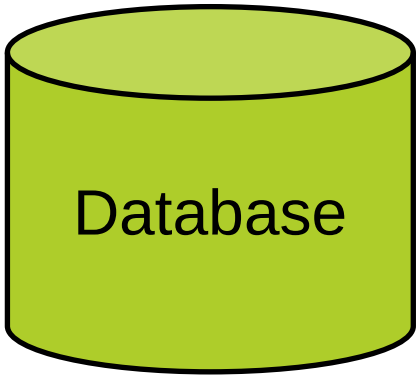
To come…

- Persistence
- Open Source
- Web
- Client
- Outlook

J2EE issues: Persistence

Persistence

- The need for fast persistence solutions is obvious for all enterprise applications
- Work around: Use JDBC
- Complex and error prone
- With Entity Beans even harder and a performance disaster



Persistence: Really solve the problem!

- O/R mappers were already well known (as mentioned)
- …but not used in the Java community

- Choosing the wrong persistence technology will lead to many problems
  - Complex code
  - Bad performance

Persistence: Solving the Problem

- So: The usual approach: Create a standard
- JDO: Java Data Objects
- Technically sound
- Lots of implementations
- Some JDO vendors based their business on JDO

- But not adopted by any big vendor
- …and did not become part of J2EE

Persistence

- People chose Hibernate as an Open Source product instead
- …and other projects like iBATIS and OJB were created

- Open Source victory over two standards: JDO *and* Entity Beans!

The stories continues…

- JSR 220 (EJB 3) updated the Entity Bean model
- JPA – Java Persistence API
- However:
  - New model was also usable outside a Java EE server
  - Technically good, but why is it standardized as part of EJB then?
  - Why are there two standards? JDO and JPA?

Persistence War

- JPA won
- Backing by large vendors (Hibernate, Oracle, …)
- Basically no JDO implementations around any more

- So the persistence problem is solved
- (sort of)

What remains? Persistence Wars

- Just a few years back JDBC was the (only) persistence technology
- Now O/R Mapper are the default
- But: They are complex pieces of software
- Caching, lazy loading, complex schema mappings …
- …and therefore a trade off
- …but often a better one
- Alternatives: Direct SQL with iBATIS, JDBC
- BTW: On .NET the default is ADO.NET not an O/R mapper
- Make your choice!
- And: Open Source is a viable alternative.

Open Source

The rise of Open Source

- Struts: First Web Framework in 2000
- (In retrospect it had a huge potential)
- (for optimization that is)

- …but much better than no framework at all
- Especially for the Java-in-JSP problems

- No JCP standard for web frameworks

Open Source

- Struts offers no solution to the general productivity problem
- Hibernate solves at least persistence issues

- Spring appeared

What Spring offers

- Dependency Injection to structure applications
- AOP to add transactions, security ...
- Simplified APIs for JDBC, JMS …
- Foundation for Spring Web Services, Spring Web Flow ...

Why Spring was successful

- Solved the productivity challenges: Much simpler model
- Very flexible: Support for EJB, JMS, JDBC, Hibernate, JNDI, …
- So you can use it in almost any context
- …and you can use just the parts you need
- Code technology independent: Applications are more portable

- Promotes best practices and makes them easy: Testing, local access to objects…
- The rerise of OO

Spring's impact

- Helped to solve the productivity problem of Java EE
- Made parts of J2EE (especially EJB) obsolete
- A lot of projects used Tomcat + Spring instead of J2EE
- No EJB, so no need for a full blown app server
- Transaction management still a point for app servers

Open Source: Tomcat

spring source

- Started as Servlet reference implementation
- Became an Apache project
- Very fast and efficient Servlet container
- Today: Infrastructure for most Java projects

- Another example of Open Source
- Let's talk about Web for a while

Web

The start for Web

- Servlets: A standardized way to handle HTTP with Java
- Mostly like CGI scripts
- …but in process

- Lots of Java code that outputs HTML
- Can we do better?

JSP

- Java Server Pages
- Coincidence: ASP by Microsoft is called similar
- HTML + embedded Java
- + special tags
- + your own tags

- Easier than Servlets: No need to output HTML

What remains: JSP

- Still predominant
- Embedding Java in JSPs is considered a Worst Practice™ nowadays
  - No separation between logic and implementation
- …but JSP are compiled into Servlets / Java to enable exactly this
- JSPs cannot easily be tested (Servlet / Server dependencies)

- You should consider other template engines

Next step: Struts

- Web MVC framework
  - Model: The data
  - View: Renders data
  - Controller: Logic
- Clear separation and much better maintainability
- Good fit for request-response based processing

Struts: The ugly

- Not very flexible
- Superclasses instead of Interfaces
- Dependencies on the ActionServlet

- Evolutionary dead end
  - No new versions for a long while
  - Struts 2 is completely incompatible
  - Spring MVC is a worthy successor and much more flexible

A quick look at other approaches…

- ASP.NET had a completely different concept:
- Components (buttons, text fields etc.) used to set up web pages
- Components send events
- Developer can handle events
- Much like traditional GUI development

- Web Objects (NeXT/Apple) has a similar approach (and was the first attempt in this field)

Co evolution applied

- JSF uses this approach
- …and is a standard

- Note: MVC or component-based is a trade off, no superiority
- Rich GUIs or request processing?
- JavaScript allowed?

And…

- Microsoft is working on MVC frameworks (ASP.NET MVC)
- So while Java travels from MVC to component based
- … .NET goes from component based to MVC

Web: What remains

- There are a lot of competing Open Source web frameworks
- This drives innovation
- …and fragmentation

- This competition sets Java apart from .NET
- ...and makes it valuable

New issues...

- Several pages are part of a business process
- Data should be shared during the process
- Can not be put in the HTTP session: Otherwise just one process can be executed in parallel
- Process should be explicitly handled

Conversations

- Allow to model processes
- Data can be stored with a process scope
- Also benefits for O/R mappers: Caches / Unit of work can be scoped in the conversation

- For example Spring Web Flow

- But enough about Web…

Client

Client

- Remember: The origin (Applets)
- With a predominant client platform portability (Windows) is often not so important here
- …so I believe it is not the most important area for Java
- But: One language on client and server

Client in the beginning

- Basic infrastructure: AWT
  - Abstract Windowing Toolkit
  - Every GUI element is shadowed by a native GUI element
  - Better performance, better integration in the platform
  - First version was done quickly
- Portability issues for application and AWT itself

Swing

- Idea: Do every in Java
- Much easier portability
- Less integration with the underlying platform

- Became a viable solution after Java performance issues were solved
- Much preferred over AWT nowadays

Eclipse RCP

- Eclipse created SWT framework
- Architecture comparable to AWT
- Ironic: IBM originally objected AWT because they had the most platforms to support
- Eclipse also has a Plug In system
  - later OSGi
  - Now also on the server
- This created RCP
- Widely used

RIA

- Rich Internet Applications
- Provide a non-HTML based rich GUI
- Silverlight, Flex / Flash, JavaFX

- I believe this will not be won by JavaFX

- …but it will make thin clients interesting for a lot more applications
- …and the server is the strong point for Java

Outlook

Java will stay

- A lot of long running buy ins from vendors…
  - IBM, SAP, SUN, Oracle / BEA, SpringSource ☺
- ...and customers
  - "We are using Java / will be using it for 10+ years."
  - i.e. for ever

Java will stay

- I think success of Java is unparalleled in the history
  - Seemingly comparable previous success: COBOL, C++
  - So much buy in
  - Binary compatibility
  - Standards (Java EE, Servlets…)
- …so comparisons are hard

Open Source will become more important

- Better capabilities to innovate
- Low risk: Commercial support, can be used freely
- Vivid Open Source Java community

- Everything is Open Source now, even Java itself.
- …so even the platform can be changed.

Java EE will become less important

- Often already replaced by Tomcat + Spring
- In some areas (Investement Banking) Java EE was never really used
- Alternative approaches for cloud, grid etc. are needed and exist

- Java EE 6 defines profiles
  - A: Web Server
  - B: Web Server + JTA + Component model
  - C: Full blown

Middleware will become modular

- One-size-fits-all is officially ended by Java EE 6
- Very diverse requirements
  - Web: Just Servlets + some framework
  - SOA: JMS + management
  - SOA: JMS + JDBC + transaction
  - Batch
- Steps beyond Java EE profiles are needed

OSGi will become interesting on the server as well

- Basic infrastructure for many embedded systems
- …and Eclipse
- Strong modularity
- …with coarse grained components
- Better architecture

OSGi will become interesting on the server as well

- Modular middleware can be customized depending on usage context
- Java EE 6 is also modularbut more is needed
- Individual updates for bundles
- So for a fix in the customer module just this module has to be replaced
- Less testing, less complex build process, …

- Heavy interest in the Enterprise
- Come to Adrian Colyer's SpringSource dm Server talk!

New languages will be interesting

- Java (the language) is not evolving fast enough
- …and was not such a good language from the start (only better)
- …and only work-arounds were added
- …and dynamically typed languages are becoming fashionable now

- You should use Java only with AOP (Spring / AspectJ)

Some examples for new languages

- Java influence:
  - Scala: Better statically typed language
  - Groovy: Dynamically typed language with clear migration path from Java
- Ports to the JVM:
  - And of course JRuby with heavy investment from SUN
  - Jython

The JVM might be more important than Java

- Lots of engineering and research
- Highly optimized
- Ubiquitous
- Often already the predominant platform
- Lots of application server and other infrastructure

- Other languages need such a platform

- However, Google Android chose the language but not the JVM