



Lessons Learned: 5 Years of Building Enterprise OSGi Applications

Martin Lippert (akquinet it-agile GmbH)





Overview

- Background
- Structure matters
- Extensions & Services
- Dynamics
- Integration
- Build & Provisioning



Background

- Enterprise Business Applications
 - ♦ On top of OSGi
 - ♦ Developed since (2004, Eclipse 3.0)
 - ♦ No classical RCP stuff... ;-)
- Client apps using:
 - ♦ Swing, Hibernate, JDO, JDBC, JNI, SOAP, a lot of Apache stuff, JUnit, FIT, Spring DM, Jetty, CICS-Adaptor, ...
- Server apps using:
 - ♦ JDO, Hibernate, SOAP, REST, Tomcat, Spring DM, CICS-Adaptor, HTTP, a lot of custom libs, Memcached, ...



Characteristics

- Highly integrated systems
 - ◆ Broad variety of backend systems
 - ◆ All kinds of technologies used for integration
- Different deliverables
 - ◆ Different rich client configurations
 - ◆ Different standalone configurations
 - ◆ Different server container setups



Structure matters





Dependencies

***Managing dependencies* within large systems is one of the most critical success factors for healthy object-oriented business applications**



What kind of dependencies?

- Dependencies between:
 - ◆ Individual classes and interfaces
 - ◆ Packages
 - ◆ Subsystems/Modules
- Dependencies of what kind?
 - ◆ Uses
 - ◆ Inherits
 - ◆ Implements



Experiences

**“Less coupling, high cohesion”
is no theoretical blah**

**OSGi makes you think
about dependencies**



Observations when using OSGi

- Design flaws and structural problems often have a limited scope
 - ◆ Problems remain within single bundles
 - ◆ No wide-spreading flaws



Import-Package vs. Require-Bundle

- We used Require-Bundle a lot
- **That was a very bad decision**
- Why?



What is the difference?

- **Require-Bundle**
 - ◆ Imports all packages of the bundle, including re-exported bundle packages
- **Import-Package**
 - ◆ Import just the package you need



What does it mean?

- **Require-Bundle**
 - ◆ Defines a dependency on the producer
 - ◆ Broad scope of visibility
- **Import-Package**
 - ◆ Defines a dependency on what you need
 - ◆ Doesn't matter where it comes from!



When to use what?

- **Prefer using Import-Package**

- ◆ Lighter coupling between bundles
- ◆ Less visibilities
- ◆ Eases refactoring

- **Require-Bundle only when necessary:**

- ◆ Higher coupling between bundles
- ◆ Use only for very specific situations:
 - split packages



Keep Things Private





Bundle API

- What should you export from a bundle?
- The easy (stupid) way:
 - ♦ Export everything
- That is a really bad idea:
 - ♦ If everything is visible, everything will be used by someone
 - ♦ Broad visibility
 - ♦ High coupling between components



Instead: Think about your APIs

- Export only the public API of a bundle
 - ◆ Less is more
 - ◆ Think about what is the API of a component
 - ◆ API design is not easy
- Don't export anything until there is a good reason for it
 - ◆ Its cheap to change non-API code
 - ◆ Its expensive to change API code



Your Buddies are Your Enemies

**Don't use buddy loading to solve
all your dependency problems**

mostly it is your fault

(structural problem, design flaws)

**Use with care to workaround library
classloading problems**



Composing





Structuring Bundles

Just having bundles is not enough

You still need an architectural view

You still need additional structures



Your Bundles shouldn't end up like this



Go! Get some structure!



What we do

- Bundle rules in the small
 - ◆ Separate UI and core
 - ◆ Separate service implementations and interfaces
 - ◆ Isolate backend connectors
- Bundle rules in the mid-size
 - ◆ Access to resources via services only
 - ◆ Access to backend systems via services only
 - ◆ Technology-free domain model



What we do

- Bundle rules in the large
 - ◆ Separate between domain features
 - ◆ Separate between applications / deliverables
 - ◆ Separate between platform and app-specific bundles
- Don't be afraid of having a large number of bundles
 - ◆ Mylyn
 - ◆ Working Sets
 - ◆ Platforms



Shippable units

- Bundle sets form different products
 - ♦ Different clients
 - ♦ Different server-side apps
- Easy to deploy different apps, but not for free
- You need:
 - ♦ Less bundle dependencies
 - ♦ Pluggable units (adding stuff from outside)
- Configuration code is a bad smell



Refactoring Bundles

“A good design today might be a bad one tomorrow”

Refactor early, refactor often



Don't forget to test

- JUnit-Tests wherever you can
 - ♦ (TDD preferred, of course)
- Don't rely on the OSGi runtime
- Test bundle-internals?
 - ♦ No, just the public API is good (black-box)
 - ♦ Yes, I would like more tests (while-box)
 - x-friends
 - Fragments
 - Separate source folders
 - ♦ Having both is a good idea



Extensions and/or OSGi Services



(borrowed from Peter Kriens)



Experiences

- Extension Points are really useful and powerful
 - ◆ Allows you to implement pluggable apps
 - ◆ Decouples your system
 - ◆ Forces Dependency Inversion
 - ◆ Eases scaling up
- You can easily misuse them
 - ◆ We used extension points for all kinds of things
 - ◆ We used them statically
 - ◆ We used them for N-to-one relationships



OSGi Services vs. Extension-Points

- Some things are like extensions
- Some things are like services
- **Use the appropriate mechanism**



Dynamics





Dynamics are hard

**Its hard to build a really dynamic system,
you need to change your mindset**

Think about **dependencies**

Think about **services**

Think about **everything** as of being **dynamic**



Dynamics are hard

**It's even harder to turn a static system
into a dynamic one**



Integration





Integration is easy

**Integrating an OSGi system into an
existing environment is easy**

OSGi runtimes are easy to start and to embed
Clear separation between inside and outside world



Experiences

- Integrate existing rich client app into proprietary client container
 - ◆ Ugly boot-classpath additions like XML parser stuff
 - ◆ Self-implemented extension model using classloaders in a strange way
 - ◆ Used a large number of libs that where not necessarily compatible with the existing rich client app
- **Integration went smoothly**
 - ◆ **just launch your OSGi framework and you are (mostly) done**



Integration can be hard

- Using existing libraries can be hard
 - ◆ Sometimes they do strange classloader stuff
 - ◆ Start to love `ClassNotFoundException`, it will be your best friend for some time
- The Context-Classloader hell
 - ◆ Some libs are using context-classloader
 - ◆ OSGi has no meaning for context-classloader
 - ◆ Arbitrary problems



Experiences

- We got every (!) library we wanted to use to work within our OSGi environment
 - ♦ Rich-client on top of Equinox
 - ♦ Server-app on Equinox
 - ♦ Server-app embedded into Tomcat and Jetty using Servlet-Bridge
- But it can cause some headaches at the beginning



Build & Provisioning





Build

- An automated server-side build is priceless
 - ♦ PDE-Build
 - ♦ Custom ANT-Build
 - ♦ CruiseControl
 - ♦ Hudson
 - ♦ Unit-Tests
 - ♦ PMD, Checkstyle, FindBugs, etc.



Experiences

- Even though its server-side - it should be fast
 - ♦ Long-running builds are a bad smell
- Produce ready-to-use packages
 - ♦ Additional installation work is tedious and errorprone



Provisioning

- We use p2, of course... - Just kidding...
- Used zipped install package
- Simple solutions to simple problems... ;-)
- Adopted existing self-implemented server-side update mechanism
 - ♦ But avoid tedious publishing steps of new builds



Conclusions





Looking back

- Large OO system, grown over years
- **Its still easy and fast to add/change features**
- I think OSGi is a major reason...
- But why?



OSGi led us to...

- Thinking about structure all the time
 - ♦ Avoids mistakes early (before the ugly beast grows)
 - ♦ Less and defined dependencies
 - ♦ No broken windows
- Good separation of concerns
- Dependency inversion & pluggable architecture
 - ♦ easy to add features without changing existing parts
- Many small frameworks
 - ♦ better than few overall ones



Conclusions

Never again without OSGi

You will love it

You will hate it

And in the end its your best friend





Thank you for your attention!

- Questions and feedback welcome!
- Let us know if you need assistance!!!
- Visit us at our booth!!!



Martin Lippert: martin.lippert@it-agile.de