OSGi Best and Worst Practices

Martin Lippert

Context

- 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
- Eclipse platforms and frameworks including:
  - Equinox, IDE, RCP, p2 and various RT projects
- Educating and mentoring people in the real world

Don’t program OSGi

Program your application

- Use POJO
- Keep your business logic clean
- Programming practices to make gluing easy
- Dependency injection to allow composition
- Separation of concerns

- Benefits
  - Delay packaging decisions
  - Increased deployment flexibility

Solutions composed of POJOs

- Bundle POJOs as needed
- Glue together using
  - Use Declarative Services
  - iPOJO
  - BluePrint Services
  - GuicePeaberry
  - …
- Use insulating layers to keep OSGi out of your code

Structure matters

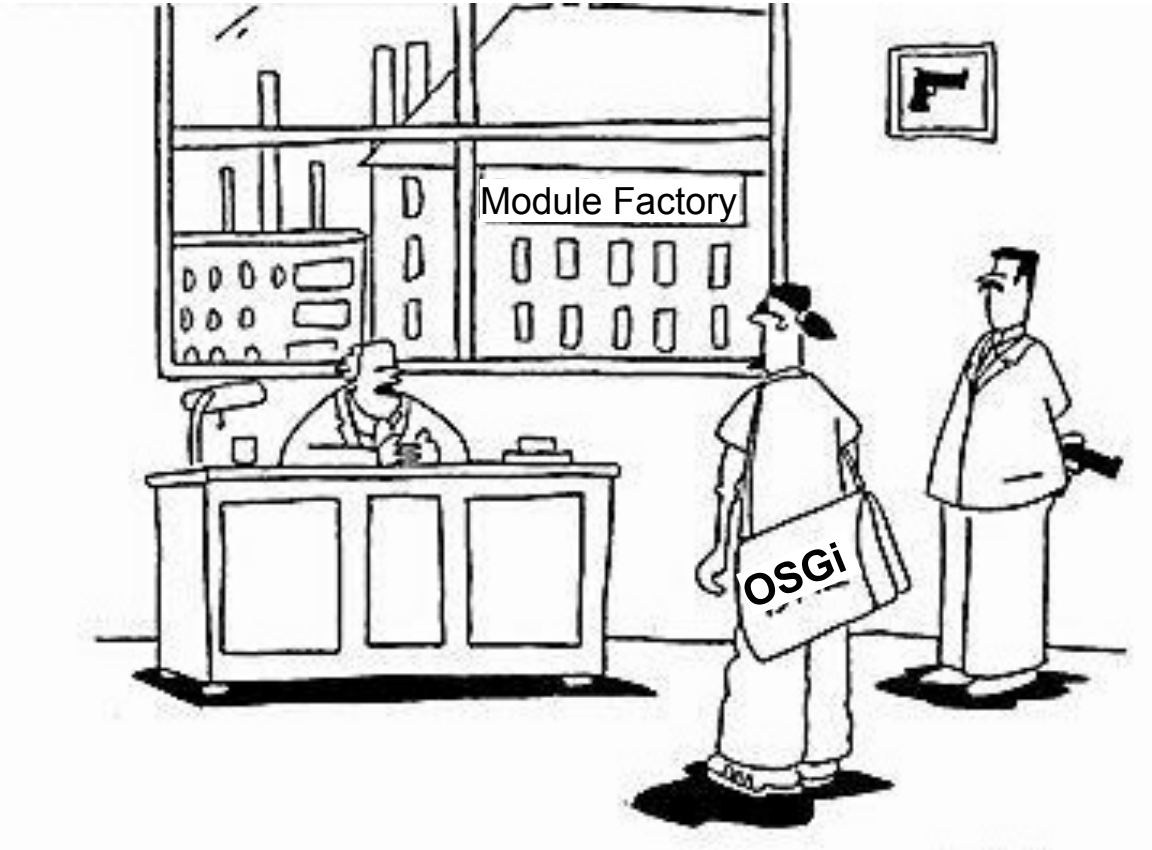
Dependencies

***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

Don’t shoot the messenger



Around here our policy is to shot the messenger!

“Low coupling, high cohesion”
Not just a nice idea

OSGi makes you think
about dependencies

It does not create them!

Observations when using OSGi

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

Take just what you need

Import-Package vs. Require-Bundle

- 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

Version management

- Version number management is essential
- Depending on a random version is pointless
- Failing to manage version numbers undermines consumers

- Import-Package ➔ package version management
- Require-Bundle ➔ bundle version management

Keep Things Private

API

- API is a contract with between producer and consumer
  - Prerequisites
  - Function
  - Consequences
  - Durability
- Key to effective modularity

Bundle API

- What should you export from a bundle?
- The easy way:
  - Blindly export everything
- That is a really bad idea:
  - No contract was defined
  - Consumers have no guidance
  - Broad visibility
  - High coupling between components

Producers: Think about your APIs

- Export only what consumers need
  - Less is more
  - Think about 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

Consumers: Think about what you're doing

- Stay in bounds
- If you can't do something, perhaps
  - Use a different component
  - Use the component differently
  - Work with the producer to cover your use-case

Informed Consent

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!

Guidelines

- Bundle rules in the small
  - Separate UI and core
  - Separate client and server and common
  - 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

Guidelines

- 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

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

Conclusions

Looking back

- Large OO systems grow over years
- **Its easy and fast to add/change features**

- OSGi is a major reason…
- But why?

OSGi leds 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 injection & pluggable architecture
  - Easy to add features without changing existing parts
- Many small frameworks
  - Better than few overall ones

Conclusions

Never without OSGi

You will love it

You will hate it


In the end its your best friend

Thank you for your attention

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

Special thanks to Jeff McAffer