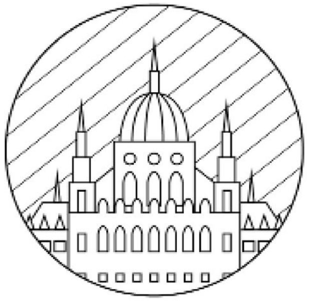
Reflections on the Design of Business Applications

Christian Schneider



APACHECON EUROPE

CORINTHIA HOTEL
BUDAPEST, HUNGARY
NOVEMBER 17-21, 2014

Introduction

Christian Schneider

Open Source Architect at Talend

Active in several Apache projects (Karaf, Camel, CXF, Aries)

Focusing on architecture and OSGi

Agenda

- Good and bad high level structuring
- Separating concerns
- Dependencies
- OSGi benefits and perils
- Camel driven architecture?

Do your packages look like this?

Technical package structure

Package structure of a backend built with camel

```
com.mycompany...

  .services
      .routes
      .converters
      .exceptions
```

So your business is all about apache camel... ?

Technical package structure

Or does this look familiar?

```
com.mycompany....

  .frontend
      .views
      .model
      .controller

  .backend
      .dto
      .dao
      .exceptions
```

Typical for a spring based application. Is it better?

Business oriented package structure

Communicate your business functions

com.mycompany.shop

- .cart
-     .ui
-     .model (API → Service iface, model classes, exceptions)
-     .model.impl (Private)
- .articles
-     ...

.checkout

classes for a business function together → higher cohesion

- less deps between packages → lower coupling
- different packages when packaged separately

Separating concerns

Separating concerns

- List the concerns you want to check (Business, Technical) or (Persistence, Business Logic)
- Each concern gets a color
- Then look at classes and color them according to the concerns they cover
- Each class should only have one color (basically the single responsibility principle)
- Ideally the same can apply for whole packages

Beware of util packages

Util package

Contains everything that does not match other packages

- Tends to contain classes with very low cohesion
- Indirectly coupling most of your code
- Very difficult to version

➡ Try to avoid util packages
Instead put the classes in more specific packages

Util bundle

➡ In OSGi rather embed needed classes than installing util bundles

External dependencies

Dependencies tend to spread

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[d] \arrow[dr] & \\
B & C \arrow[dl] \arrow[d] \arrow[dr] & D \\
E & F & G
\end{tikzcd}$$

- Dependency numbers tend to multiply for each layer of libraries
- In the end maven downloads the internet ...
- We as library developers can improve that

External dependencies

- Universal dependencies

  Needed in almost all bundles

- Specific Dependencies

  Can be limited to layer or single bundle

Universal Dependencies

Used in almost all of the code

Examples:
- Blueprint / Spring
- APIs like for Transaction support
- Commons lang, collections

→ Have as few as possible
Disallow single developer to add universal dependencies

Specific Dependencies

- Can be limited to single layer or single bundle

  - Template engine / mail sending
    Only in single bundle, access using service
  - CXF
    Only in rest service layer
  - JPA / Hibernate
    Only in persistence layer

- Make sure the dependency does not leak outside
- decouple using plain java API bundle
- Works especially well in OSGi

Example

separation of concerns
limiting use of
dependencies

Example template engine for mailings

Caller

```
Input input = new TemplateEngineInput();
input.getTemplateObjectMap().put(“user”, userEntity);
producerTemplate.sendBody(“direct:mailEndpoint”, input);
```

Route

```
from(“direct:userMailEndpoint”)
    .bean(extractCustomerDataFromSAP)
    .bean(applyTemplate)
    .to(“smtp://mailer@localhost:25?contentType=text/html”)
```

What could be wrong with that?

Example template engine for mailings. Better design

Caller

```
MailData input = new MailData();
input.setEMail(userEntity.getUserEmail());
input.getMap().put("name", userEntity.getUserName());
mailSender.send(input);
```

Service

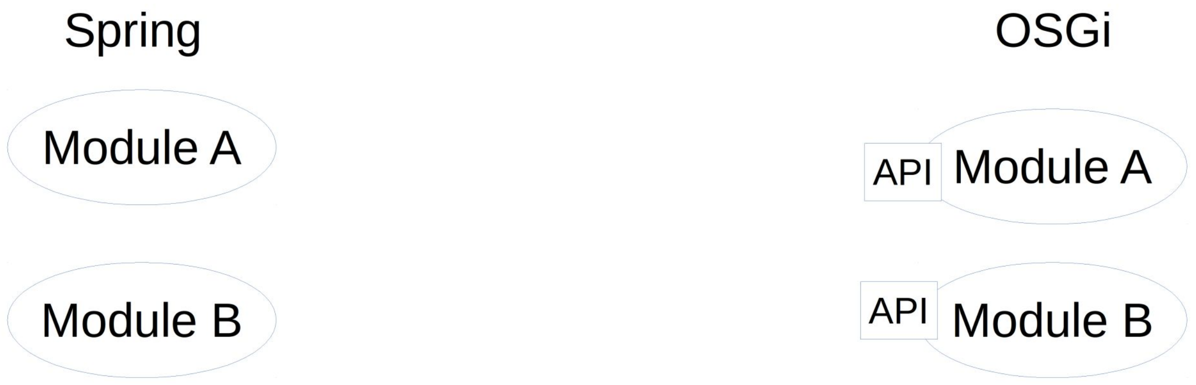
Whatever .. just make sure you do not leak the details it out

Use service with plain java interface

Promises and perils of OSGi

What does OSGi offer?

- Well defined and versioned dependencies on package level
- Loose coupling based on versioned APIs
- Self contained modules providing services



Complexity C(A) * C(B) C(APIA) * C(B) + C(APIB) * C(A)

→ lowers the complexity of a well designed application considerably

What are the perils?

- Too many small modules

  → complexity of wiring and managing the modules

- API not minimal or simply all classes public

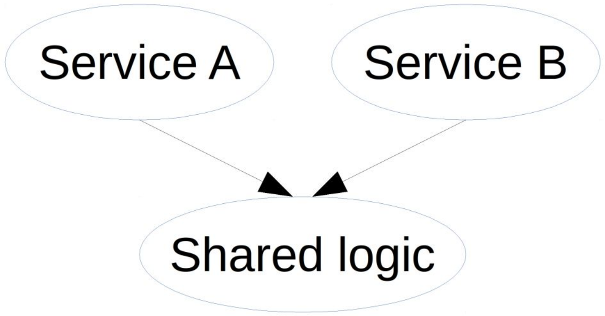
  → Complexity not reduced

- All modules released independently and mixed at deployment time

  → Only works with well managed APIs,
  even then difficult to get right

➡ Can make the OSGIfied version more complex than the original

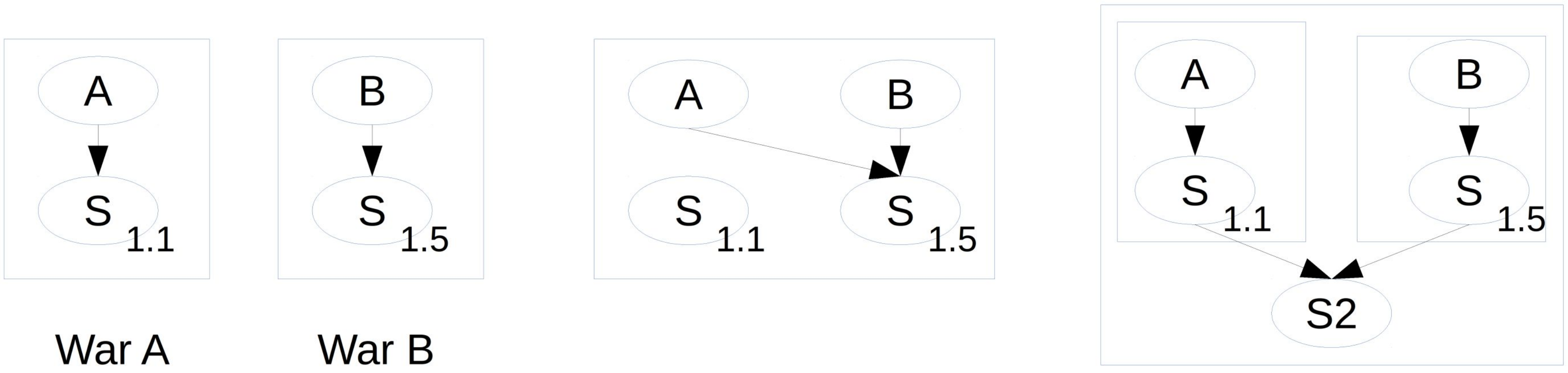
Microservices with OSGi ?



Servlet Container

Plain OSGi

OSGi with subsystems



→ Separate Classpaths
No conflicts
but lots of duplication

→ Shared classpath
Only works if managed
carefully

→ Configurable boundaries
No conflicts
Not yet mature (2014)

Semantic versioning the OSGi way

- Each API package versioned independently (using packageinfo file)
- Automatically check vs base line API using e.g. maven bundle plugin
- Depending on change package version increased at minor or major version

➡ Only sane way to do API versioning in OSGi

Camel is great, so let's do everything with it … really?

Camel based design?

```
@Autowire
DetermineBigCustomer determineBigCustomer;
...

from(REST-ENDPOINT-CHECKOUT)
     .to(determineBigCustomer)
     .when(header("isBigCustomer"))
          .to(new BigCustomerDiscount())
     .end
     .to("direct:checkpriceAndAvailabiltiy")
    .bean(sapConverter)
    .bean(sapSender)
```

What type of data is transmitted ?
Would it be harder to do this logic in plain java?

Why not plain Java?

```
void checkout(CheckoutData checkoutData) {
        Customer customer = checkoutData.getCustomer();
        if (isBigCustomer(customer)) {
                applyBigCustomerDiscount(checkoutData)
        }
        SapCheckoutData sapCheckoutData
sapConverter.convertCheckoutData(checkoutData);
        sapSender.checkout(sapCheckoutData);
}
```

- Typesafe interfaces
- Clearly shows what data is processed
- As concise as the camel route
- Easy to debug

→

use Camel for integration,
Java for business logic,
look for high cohesion

What to remember

What to remember ?

- Structure your code according to business functions not technology
- Separate business code from technical code
- Communicate using well defined APIs
- Use semantic versioning for APIs
- Make most of the classes private to the bundle
- More often use plain java instead of external libraries
- Limit influence of external libraries to very few bundles

Some references

- My homepage: http://www.liquid-reality.de
- How OSGi can solve some complexity problems
  http://njbartlett.name/2013/02/04/no-solution-for-complexity.html
- Critical view on Neil's article and some discussions
  http://milen.commsen.com/2013/02/about-complexity-modularity-and-osgi.html
- How to do semantic versioning

  http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf