invesdwin

Software Produktlinien Plattform

von Edwin Stang

Agenda

1. Einleitung
2. Software Produktlinie
3. Entwicklungsumgebung
4. Konfigurationsmanagement
5. Assets
6. Bindung der Variabilität
7. DDD und MDA
8. Fazit & Diskussion

Fragen gerne zwischendurch!

1. Einleitung

Entstehungsgeschichte

Motivation

1. Einleitung
- Hintergründe -

- Bachelor Thesis -> Bisherige Plattform und SPL?
  - Stattdessen Entscheidung zu neuen Business Components
  - Blaupause für Projekte statt SPL
- Meine Entscheidung SPL zuhause zu schaffen
  - Domäne Börseninvestments, SPL als Grundlage
    - Forschungsprojekt
    - 1-Mann-Entwicklung
    - niedrige/keine laufenden Kosten
  - OpenSource **Leverage**
  - **Wiederverwendung** an erster Stelle
  - F&E macht **Flexibilität** notwendig
  - **Entwicklungskomfort** hoch priorisiert
  - Es soll **Spaß** machen!

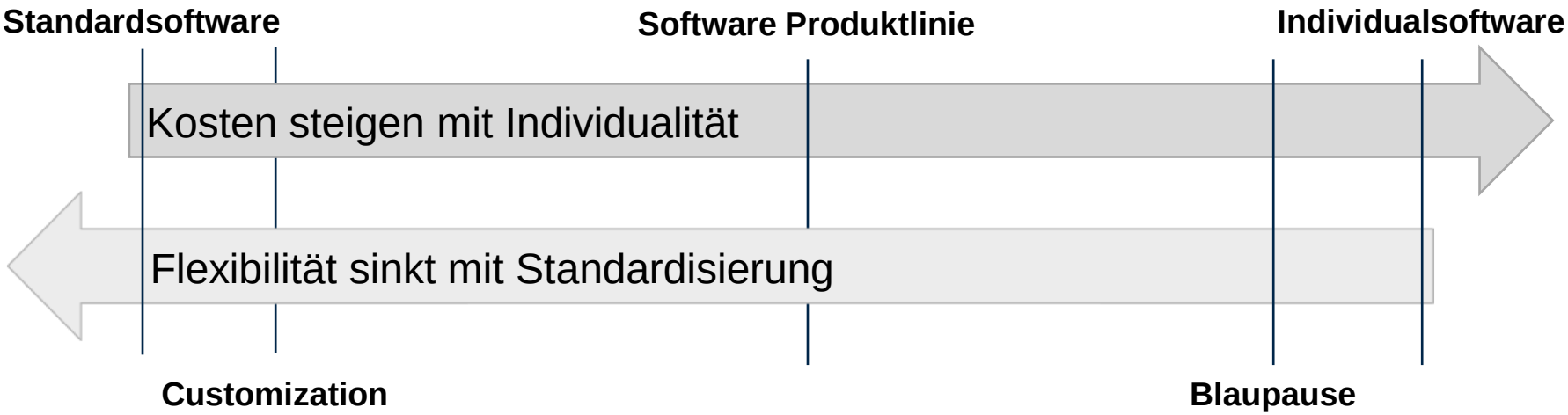
In Entwicklung seit Oktober 2009

mittlerweile recht stabil

2. Software Produktlinie

Wissenschaftlicher Hintergrund

2. Software Produktlinie
- Einordnung -

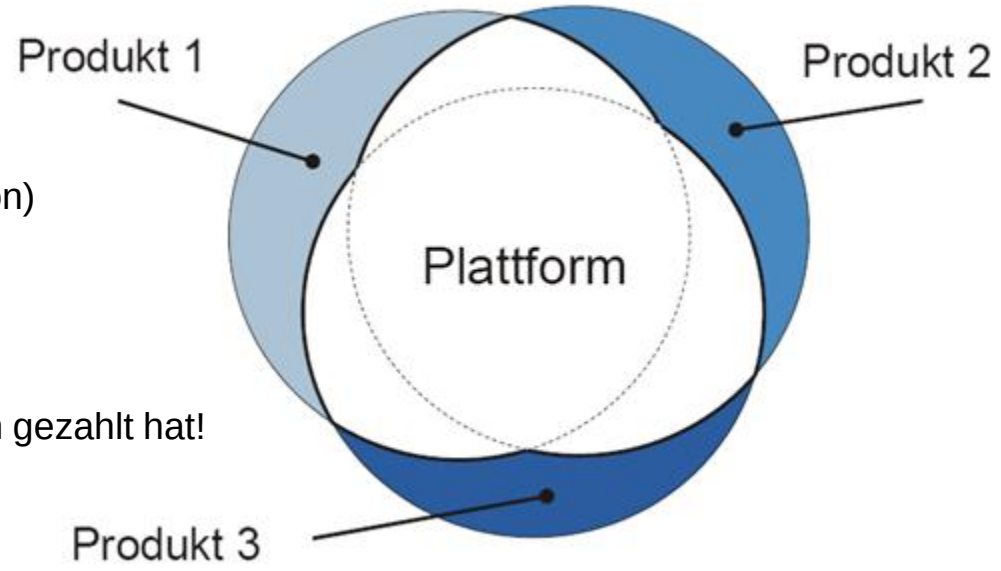


Lösung:

- ✓ Reduktion der Kosten durch **Wiederverwendung**
- ✓ Wahrung der Flexibilität durch **Variabilität**

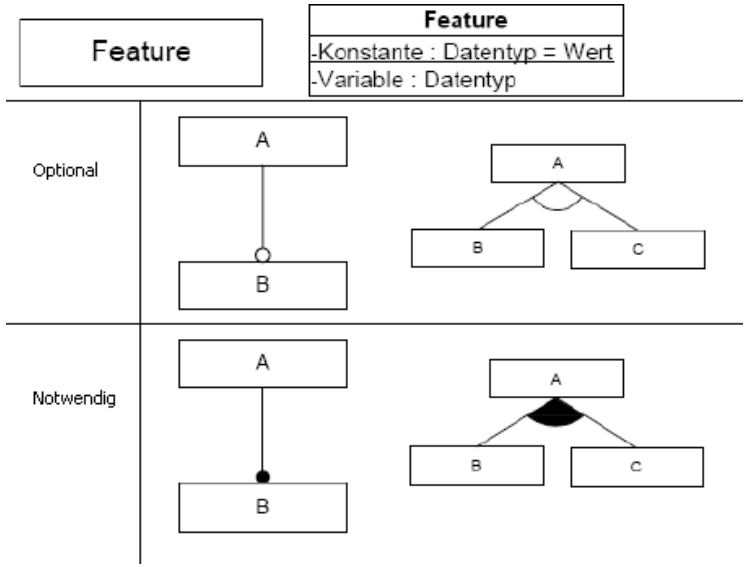
2. Software Produktlinie
- Assets -

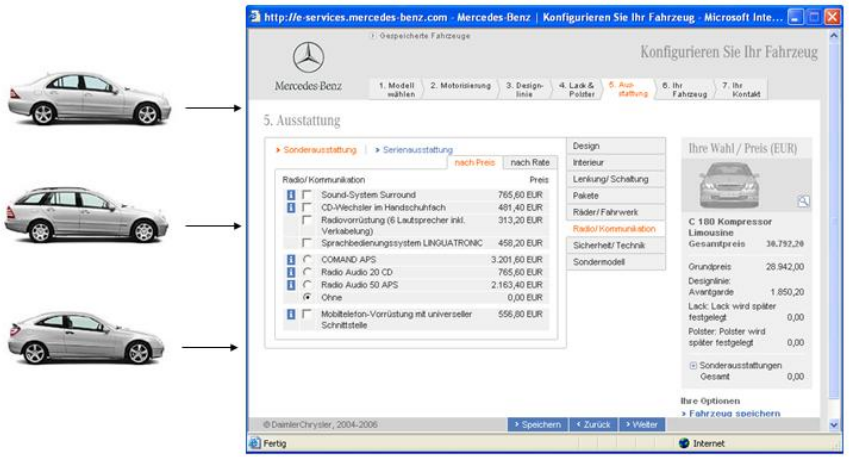
- Assets
  - Artefakte wie **Dokumentation, Code, Libs, Konfiguration, Produkte, Plattform**
  - SPL besteht aus - und managed Assets
- Wiederverwendung
  - Artefakte werden designed, um sie Wiederverwenden zu können
  - Schaffung von **Synergien**
  - Reduktion der Entwicklungskosten
  - Schnellere Time-To-Market
  - Copy/Paste vermeiden (Artefaktduplikation)
- Modularität
  - **Artefakte bündeln**
  - Nur das ausliefern, wofür der Kunde auch gezahlt hat!



2. Software Produktlinie
- Variabilität -

- Variabilität
  - Änderbarkeit der zu nutzenden Implementation
  - Selektion von **Features**
  - Preis variabel machen und am Kunden orientieren
- Variabilitätspunkte
  - Entscheidung „**wo**“ Varianten gewählt werden
  - Geplante Flexibilisierungsmöglichkeit
  - Konfigurierbar oder Wählbar
  - Kunde trifft hier Entscheidungen
- Varianten
  - Features kapseln und optional wählbar machen
  - Vgl. Autobau:
    - Anhängerkupplung ja/nein
    - Sportfahrwerk vs Komfortfahrwerk
    - Kombi vs Limousine





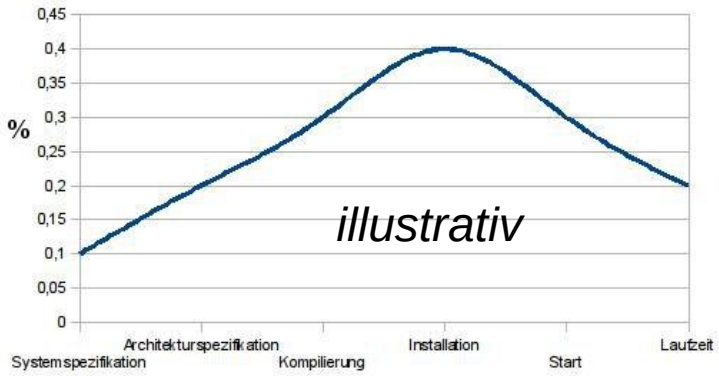
2. Software Produktlinie
- Bindung -

- Bindung
  - Variabilitätspunkt wird mit Variante besetzt
- Bindungspunkte
  - Entscheidung „**wie**“ im Code/Deployment eine Variante gewählt wird
  - Patterns: Dependencies, Properties, Runtime-Button
- Bindungszeitpunkte
  - Entscheidung „**wann“** eine Variante gewählt werden kann
  - Vgl. fertiges Auto:
    - Ausstattungspakete wählen (beim Kauf)
    - Navi-Karten aktualisieren (vor Autostart)
    - Sport-Schaltung aktivieren (während Fahrt)
    - ESP deaktivieren (während Fahrt)

Übersicht Bindungszeitpunkte:

	flexibility	performance	code size	complexity
source time	-	+	+	-
compile time	+	+	+	-
link time	+	+	+	-
load time	++	+	+	+
run time	+++	-	-	+

Anteile genutzter Bindungszeitpunkte:



- Früh - - Spät -

3. Entwicklungsumgebung

Worin ist die SPL realisiert?

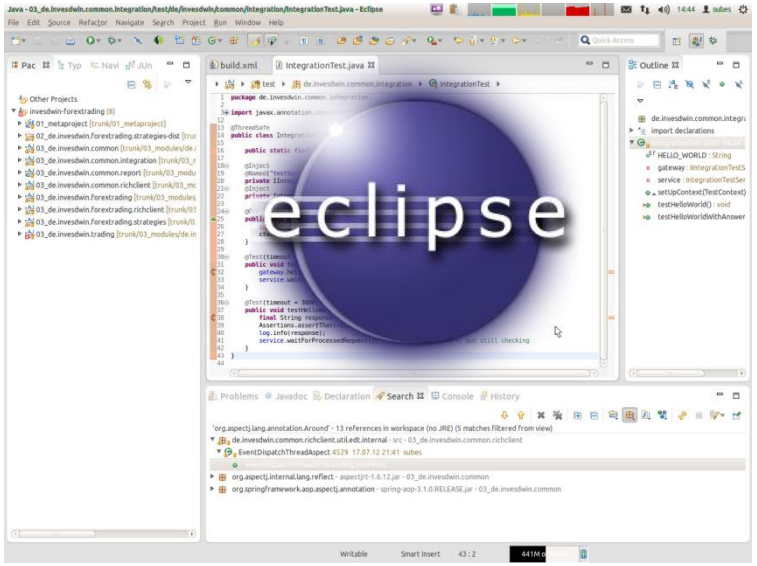
Standardzeugs…

3. Entwicklungsumgebung
- Low Impact -

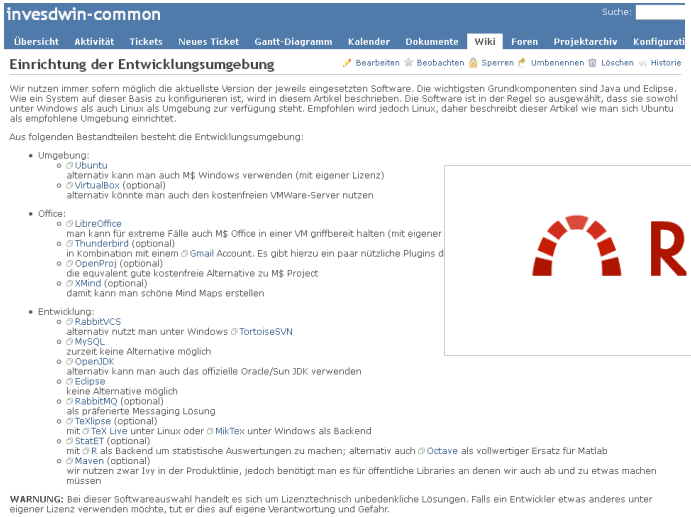
- **OS:** Windows/Linux
- **Versionsverwaltung:** SVN
- **Sprache:** Java 6 (Java 7 kompatibel)
- **IDE:** Eclipse + Plugins
- **Build & Dependency Mgmt:** Ant + Ivy (+ Groovy)

  Wieso kein Maven?

  → SPL erfordert abweichende Konventionen



- **Dokumentiert:**



3. Entwicklungsumgebung
- Voraussetzungen -

- **Rechner:**
  - **CPU:** Dual Core
  - **RAM:** 4 GB
  - **Festplatte:** nicht zu langsam
  - **Speicherplatz:** ~3GB frei
  - **Build-Zeiten**: Anhand Plattformprojekt

    ~ 8 Minuten für 18 Module

    => ~ 30 Sekunden pro Modul

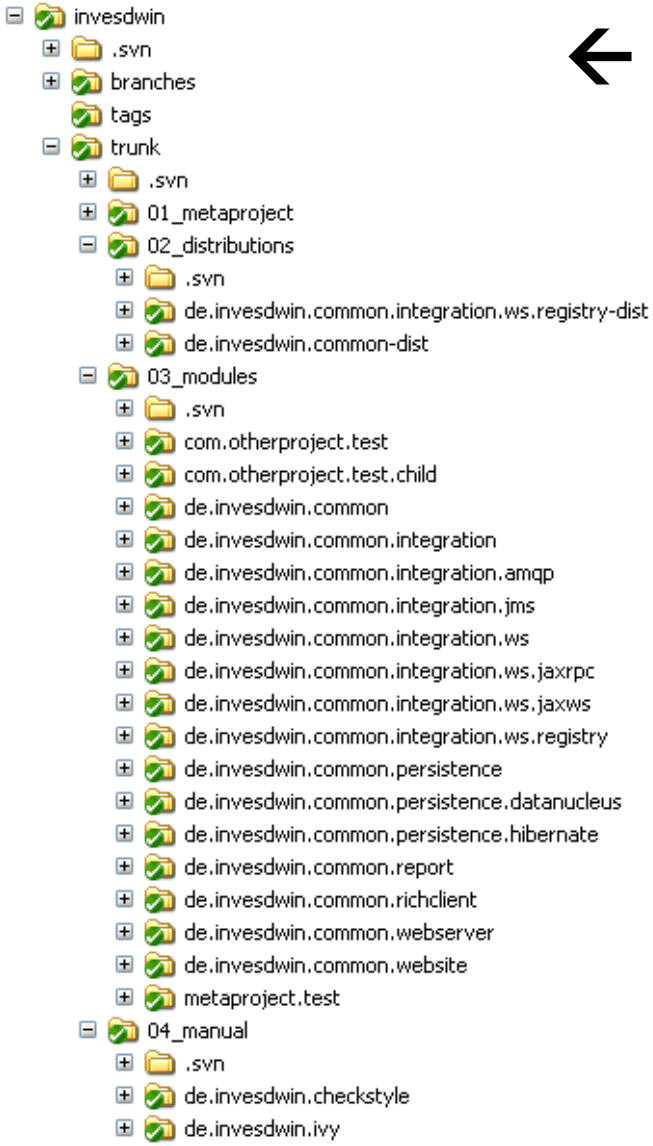
    (Halbierung durch bessere Hardware möglich)

- **Know-How:**
  - JEE-Erfahrung erforderlich (z.B. bisherige Plattform)
  - Grundlegende Kenntnisse zu Spring wünschenswert
  - Nach 1-2 Wochen Entwicklung damit sollte jemand in der Plattform gut klarkommen

4. Konfigurationsmanagement

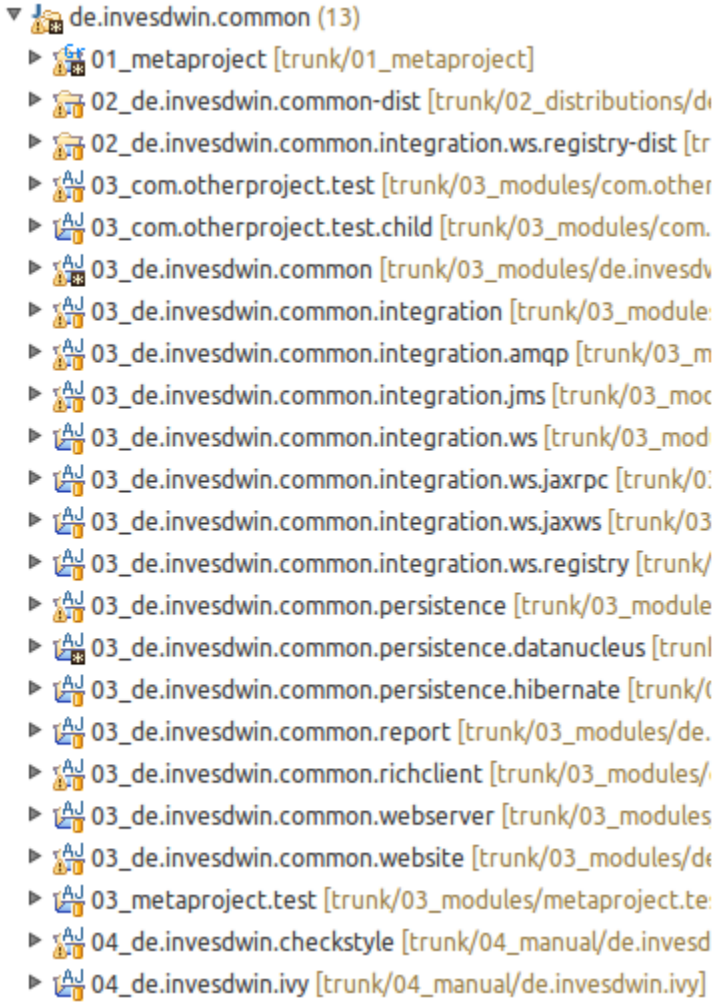
Ein wissenschaftlich neuer Ansatz

4. Konfigurationsmanagement
- Übersicht -



← Explorer

Eclipse →



Produkt:

- 01 Metaprojekt
- 02 Distribution
- 03 Modul
- (04 manuelles Modul)

4. Konfigurationsmanagement
- Modul I -

- **Wiederverwendbarer Baustein für Produkte**
  - **Technisch:** Best Practices, Patterns, Frameworks, Utils, Tools,
  - **Fachlich:** Services, Entitäten, Logisches, Algorithmen

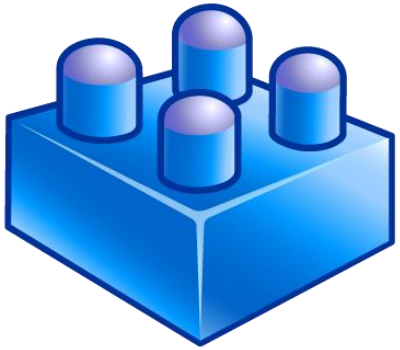
→ Jeweils ein eigenes Eclipse-Projekt, um Modularität zu gewährleisten

Komponenten

Anbieter, Vertrag, Kunde, …

Zusammenfassen &

Wiederverwendbar machen



Modul

Vertragsverwaltung

4. Konfigurationsmanagement
- Modul II -

- Vergleich zur bisherigen Plattform:
  - **Modul > Container**
  - **Upgrade-Pfade** statt Copy-Paste
    - → Lieber Variabilität zu Modul hinzufügen, statt neues Modul anlegen
    - → Ziel: Effektiv weniger Wartungsaufwand

mehrmals unterschiedlich Fixen	**VS**	1 mal Fixen + mehrmals Version aktualisieren

- Lessons Learned aus alten Business Components anwenden:
  - **YAGNI & KISS**
  - Nicht auf der **grünen Wiese** entwickeln und **Over-Engineering** vermeiden
    - → Variabilität erst hinzufügen, wenn sie benötigt wird
  - **Lose Kopplung** forcieren
    - → Submodule nutzen, um Funktionalität/Alternativen zu kapseln

4. Konfigurationsmanagement
- Modul III -

- **Wieso kein OSGI?**
  - Dynamisches Load/Unload unnötig
  - OSGI-Deskriptoren schlecht gepflegt in Open Source Projekten
  - Classpath-Probleme vermeiden (ein gemeinsamer Classpath ist einfacher)
  - Jar-Hell und Versionskonflikte bereits durch Dependency Management gelöst
  - Lose Kopplung durch „internal“ Packages mit Checkstyle-Prüfung gewährleistet

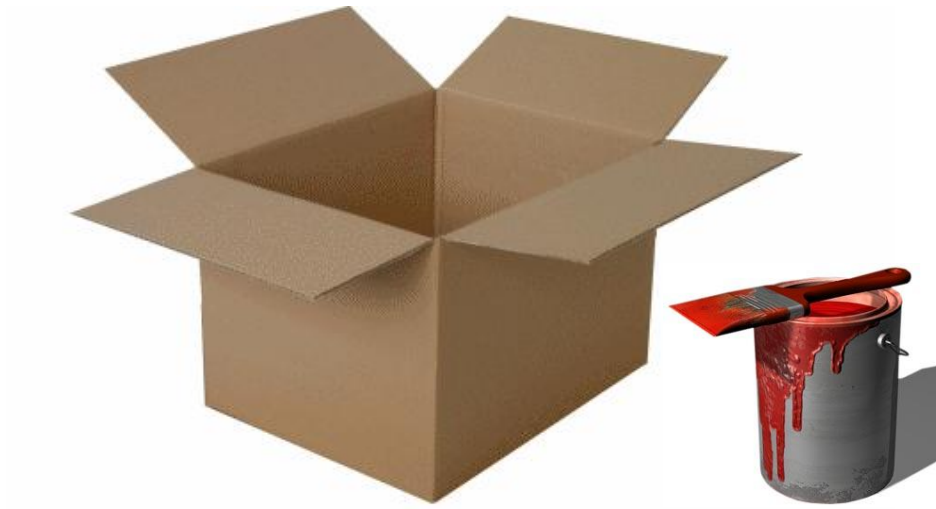
    → begründete Abweichung der „internal“ Package Regel möglich

    → ohne technische Herausforderungen

nur
OSGI
Kompatibel

4. Konfigurationsmanagement
- Distribution -

- **Ein konfiguriertes, deploybares Produkt**
  - Zusammenfassung mehrerer Module
  - **Konfiguration der Variabilität** über
    - Dependencies
    - Properties
    - Zusätzliche Ressourcen
  - Zielkundenspezifisch
  - Zielumgebungsspezifisch

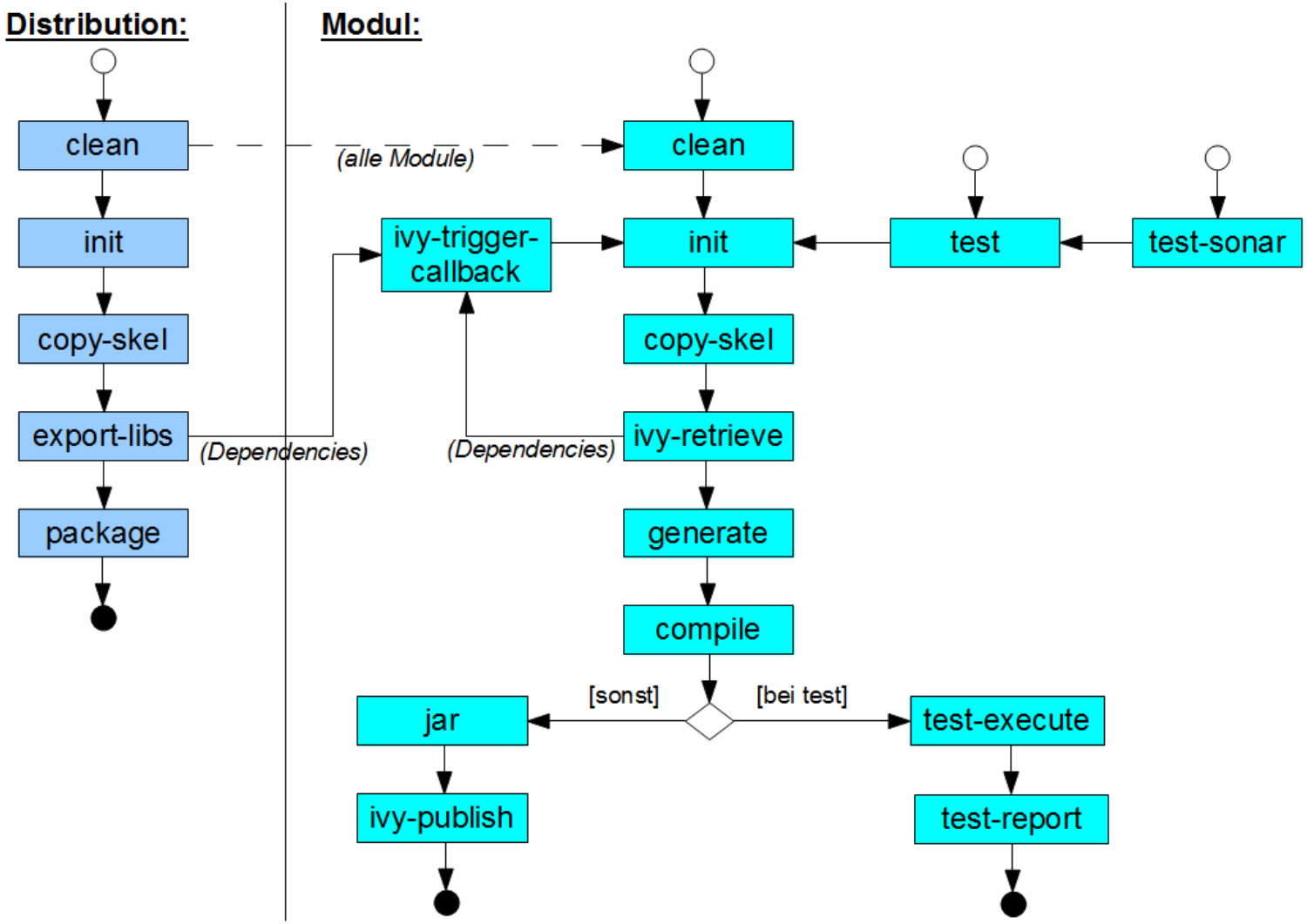


  - Entscheidung über **Pakettyp** erst hier
    - Ausführbares **Fat-Jar**
    - **Zip** mit Startskripten
    - **War** als Container

→ Beispiel Webanwendung: Fat-Jar mit embedded Jetty **oder** War für Tomcat?

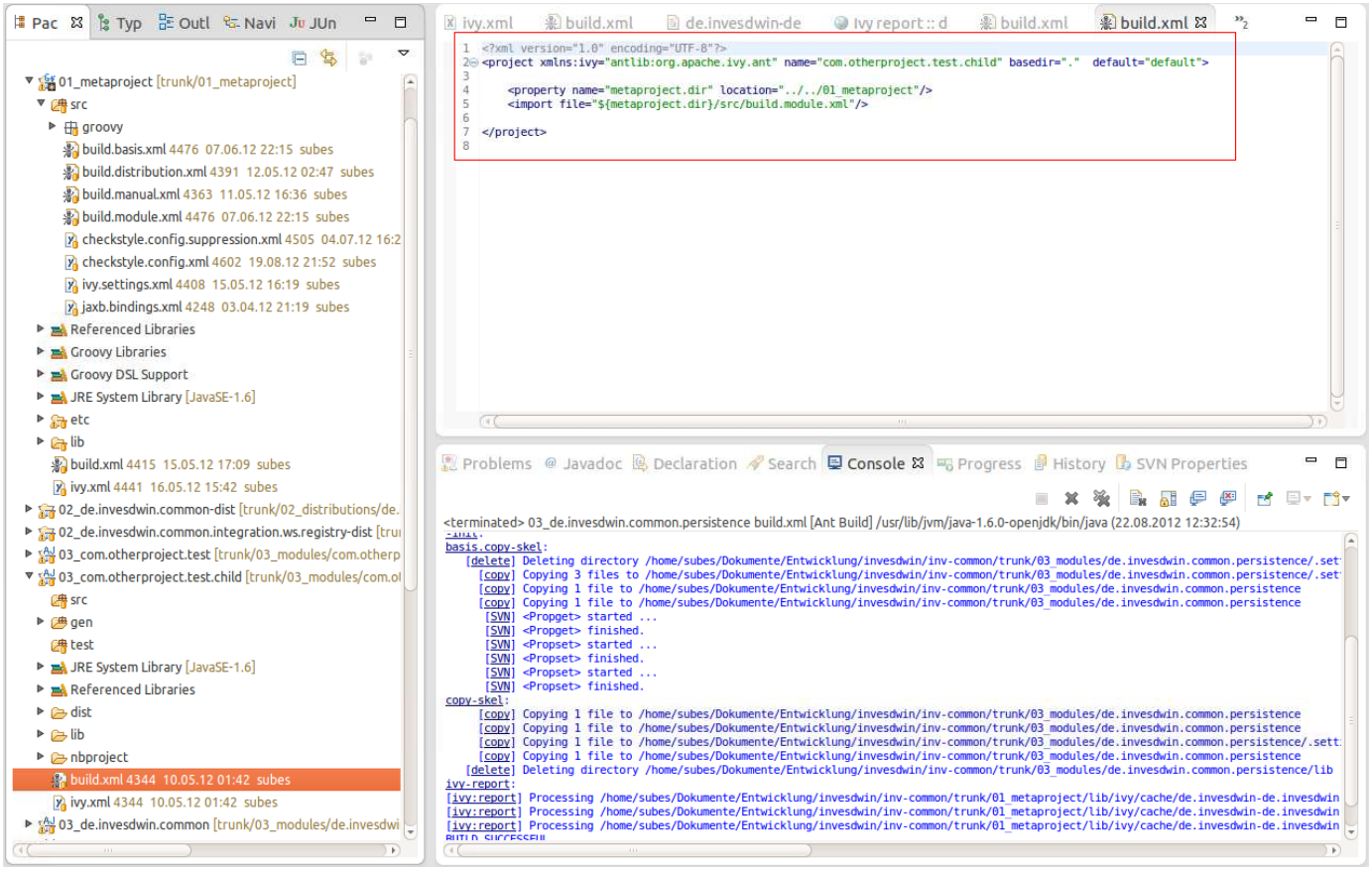
4. Konfigurationsmanagement
- Metaprojekt I -

- **Definition des einheitlichen Build-Prozesses:**



4. Konfigurationsmanagement
- Metaprojekt II -

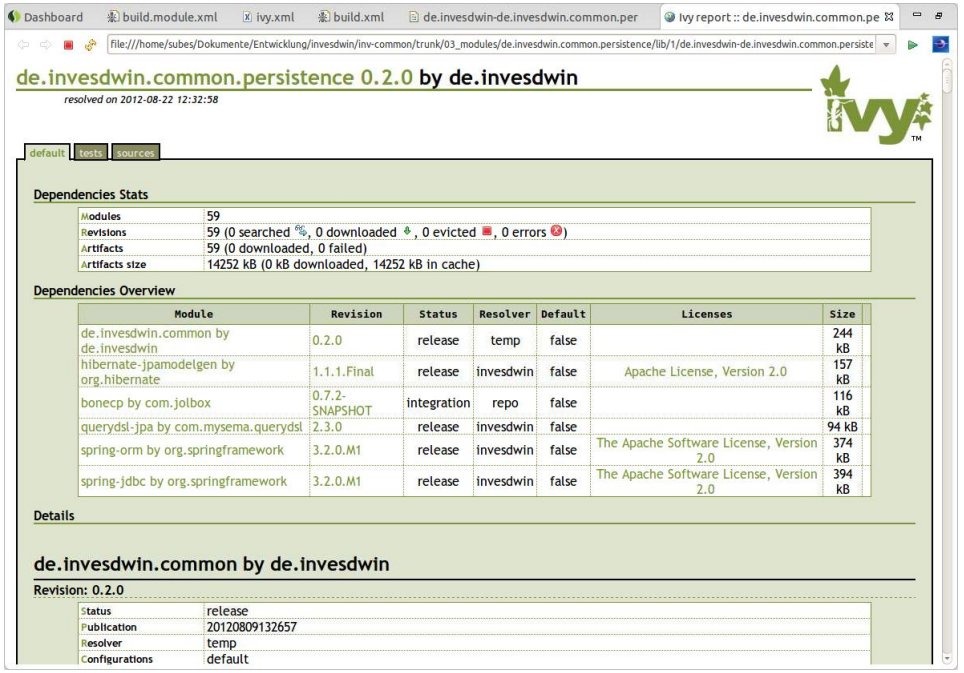
- **Wiederverwendbare standardisierte Ant-Skripte** → nur an einer Stelle pflegen
- **Projektvorlagen für Module und Distributionen** → à la Maven Archetype
- **Andere Assets erben Eclipse-Projekt-Konfiguration** → nur einmal Konfigurieren



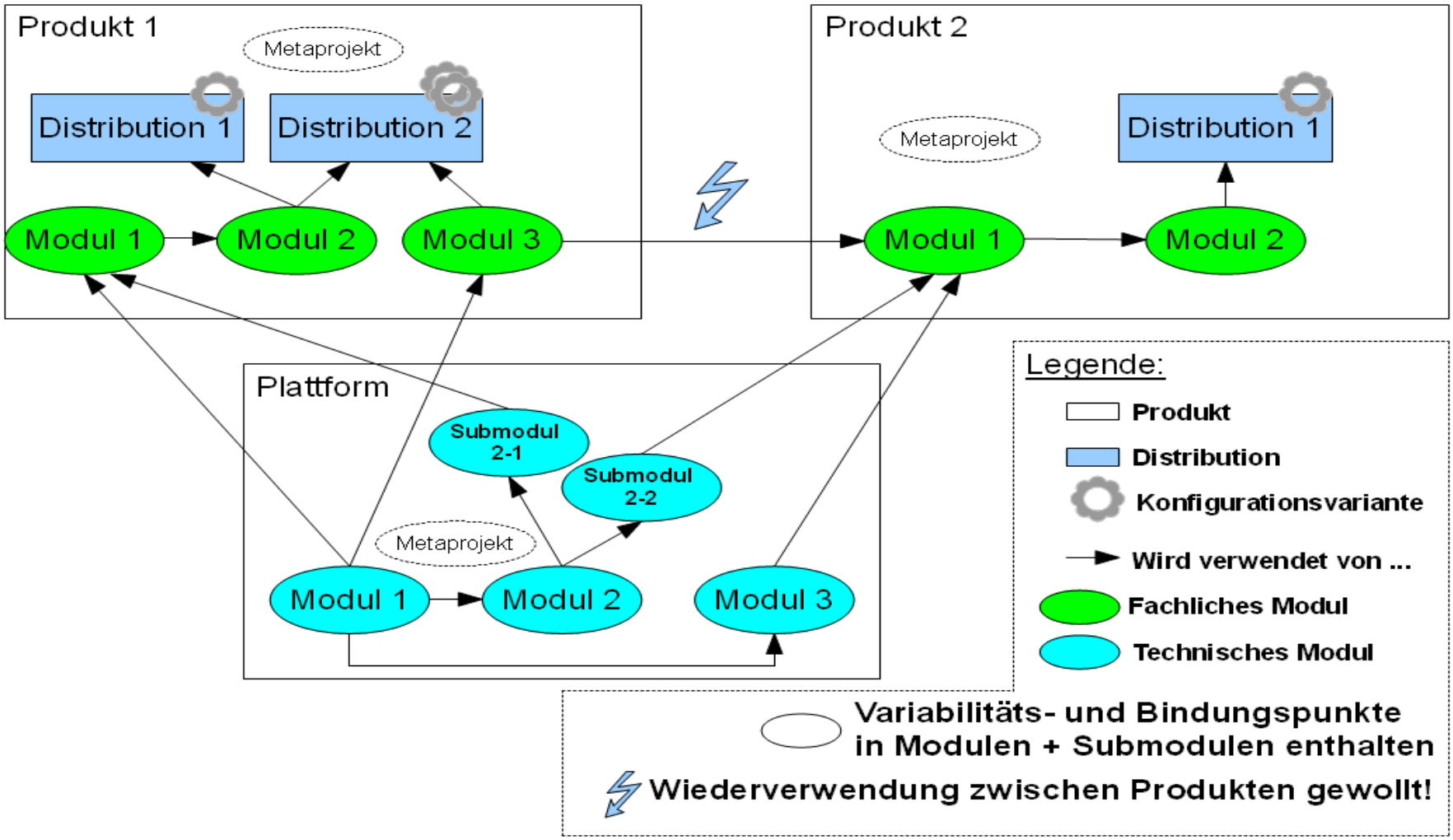
4. Konfigurationsmanagement
- Metaprojekt III -

- **Automatisch generierter Eclipse-Classpath für Dependencies:**
  - Projekt-Referenzen für modulübergreifende Refactorings
  - Einheitliche Benamung der Dependencies (<Modul>-<Version>.jar)
  - Sources zu allen Jars automatisch angehängt
  - AspectJ und weitere Eclipse-Plugins automatisch konfiguriert

→ Reduktion des Pflegeaufwandes für Module, lediglich Konfiguration in **ivy.xml** notwendig!



4. Konfigurationsmanagement
- Multiproduktmanagement -



5. Assets

Was gibt es schon?

Plattform, Produkte, Bestandteile

5. Assets
- Übersicht -

( ) = wird hier nicht betrachtet

5. Assets
- Plattform I -

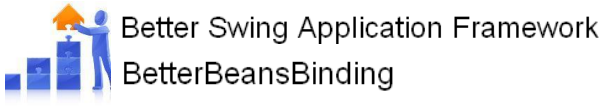
- **metaproject**

- **invesdwin-common**

- **invesdwin-common-persistence**

- **invesdwin-common-report**

- **invesdwin-common-richclient**



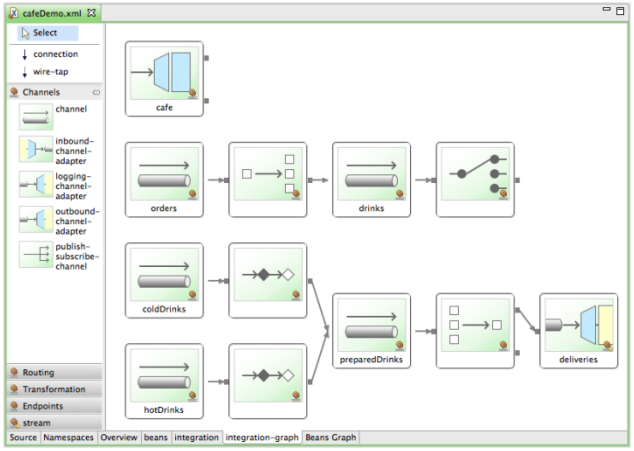
- **Infrastruktur**

- **Dependency Repos**

5. Assets
- Plattform II -

- **invesdwin-common-integration**
  - invesdwin-common-integration-amqp
  - invesdwin-common-integration-jms
  - invesdwin-common-integration-ws

SPRING INTEGRATION



JAXB

<?xml?>

google-gson

JSON
JavaScript Object Notation

-jaxrpc

-jaxws

-registry

JAXR → UDDI

- **invesdwin-common-website**

- **invesdwin-common-webserver**

Powered by

Deployment in Tomcat als WAR

konfigurierbar in Distribution

5. Assets
- Produkt webproxy -

- ínvesdwin-webproxy(-...)

HtmlUnit

GUI-Less browser for Java programs

JAKARTA COMMONS
HTTPCLIENT

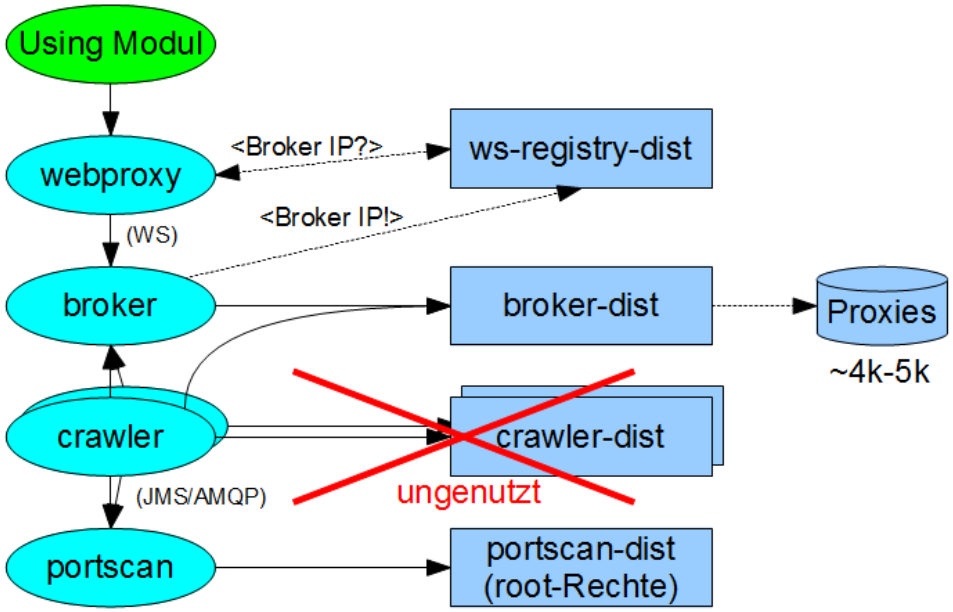
→Proxies

win/lib/j - Pcap

GeoNames Geo IP

→Location

- Vereinfachtes Kontextdiagramm:



6. Bindung der Variabilität

Bootstrap schafft Flexibilität

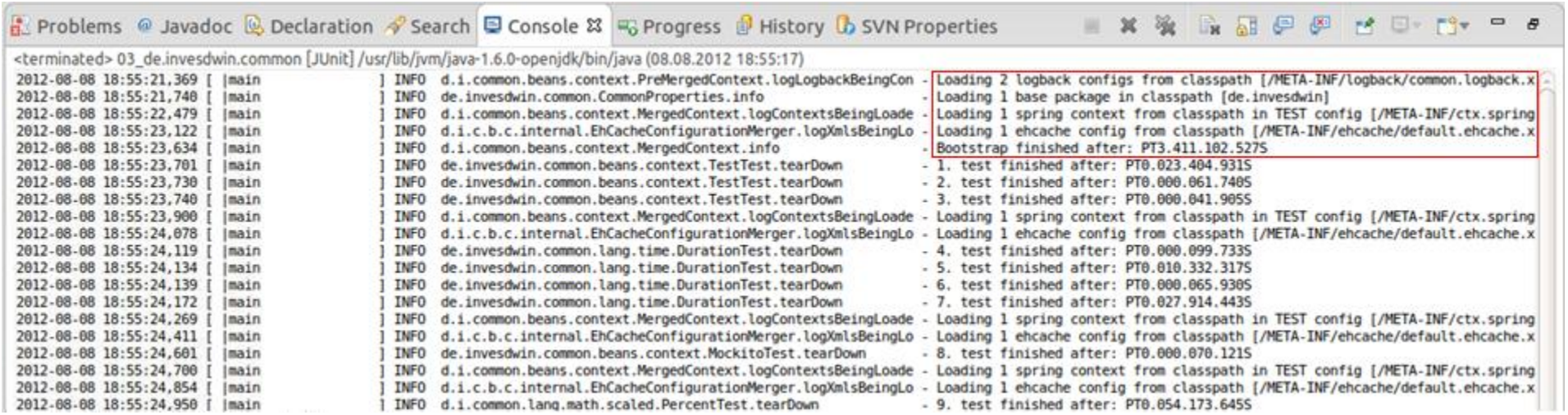
Pattern-Beispiele

6. Bindung der Variabilität
- Application Bootstrap -

- Konfiguration verteilt über verschiedene Module
- Gemeinsamer Classpath (kein OSGI)
- Zwei Spring ApplicationContexte
  - Premerged → sammeln und konfigurieren
  - Merged → Anwendung laufen lassen

- Pro JUnit Testklasse ein neuer Bootstrap (schnell)

  für Konfigurationsänderungen:
  - Auswahl der Spring-XMLs mittels IContextLocation
  - Im Test mittels setUpContext()
  - Durch Mocks mittels ITestLifecycleHook



6. Bindung der Variabilität
- Properties -

- **Alle Properties sind Systemproperties**
- Damit **verfügbar** in:
  - anderen Properties Dateien à la Ant ${property}
  - XML (Spring, Frameworks mit Commons-Configuration)
  - Java (System.getProperty(„property“))
  - VisualVM (Monitoring, Vorsicht bzgl. Sicherheit!)

*PortscanProperties.java

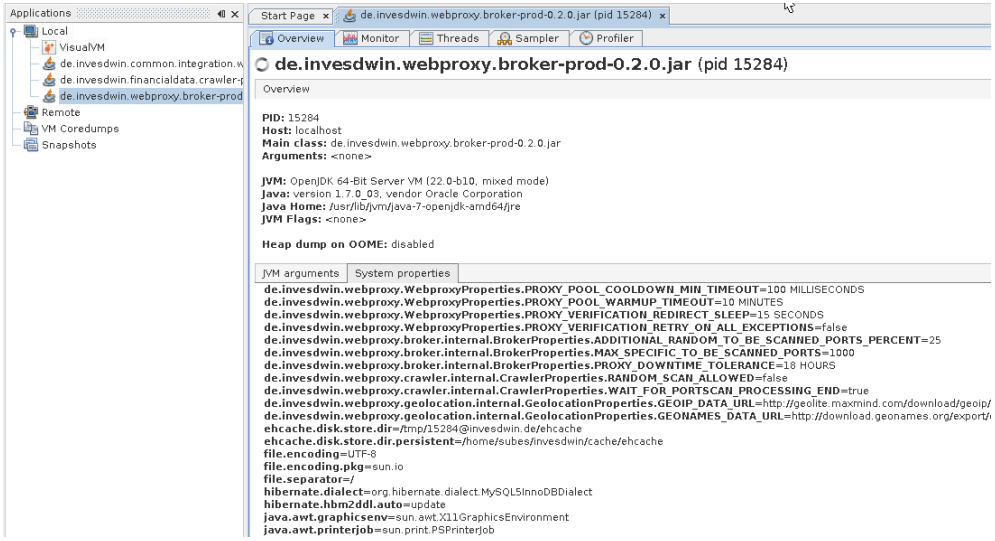
```
1  package de.invesdwin.webproxy.portscan.internal;
2
3  import java.net.InetAddress;
13
14 @Immutable
15 public final class PortscanProperties {
16
17     public static final InetAddress CHECK_HOST;
18     public static final int CHECK_PORT = 80;
19     public static final int LOCAL_BIND_PORT;
20     public static final Duration ICMP_RESPONSE_TIMEOUT;
21
22     private static final SystemProperties SYSTEM_PROPERTIES = new SystemProperties(PortscanProperties.class);
23
24     static {
25         CHECK_HOST = NetworkUtil.toAddress(SYSTEM_PROPERTIES.getString("CHECK_HOST"));
26         LOCAL_BIND_PORT = readLocalBindPort();
27         ICMP_RESPONSE_TIMEOUT = SYSTEM_PROPERTIES.getDuration("ICMP_RESPONSE_TIMEOUT");
28     }
29
30     private PortscanProperties() {}
31
32     private static int readLocalBindPort() {
33         final String key = "LOCAL_BIND_PORT";
34         final Integer value = SYSTEM_PROPERTIES.getInt(key);
35         Assertions.assertThat(NetworkUtil.isPort(value))
36                 .as(SYSTEM_PROPERTIES.getErrorMessage(key, value, null, "Value must be inclusively between "
37                         + NetworkUtil.PORT_MIN + " and " + NetworkUtil.PORT_MAX + "."))
38                 .isTrue();
39         return value;
40     }
41 }
```

de.invesdwin.webproxy.portscan.properties

```
1 #On the host port 80 must be open and a service has to be running on it. The host also has to answer pings so that the check
2 de.invesdwin.webproxy.portscan.internal.PortscanProperties.CHECK_HOST=google.de
3 de.invesdwin.webproxy.portscan.internal.PortscanProperties.LOCAL_BIND_PORT=44125
4 de.invesdwin.webproxy.portscan.internal.PortscanProperties.ICMP_RESPONSE_TIMEOUT=3 SECONDS
5 #For timings see: http://www.networkuptime.com/nmap/page09-09.shtml
6 de.invesdwin.webproxy.portscan.internal.PortscanProperties.UPLOAD_PAUSE_BETWEEN_PACKETS=0 MILLISECONDS
7 de.invesdwin.webproxy.portscan.internal.PortscanProperties.UPLOAD_PAUSE_BETWEEN_PACKETS_PER_HOST=0 MILLISECONDS
8 de.invesdwin.webproxy.portscan.internal.PortscanProperties.RESPONSE_TIMEOUT_BETWEEN_SYN_PACKETS_PER_HOST=500 MILLISECONDS
9 de.invesdwin.webproxy.portscan.internal.PortscanProperties.MAX_OPEN_ICMP_REQUESTS=25
10 de.invesdwin.webproxy.portscan.internal.PortscanProperties.MAX_OPEN_SYN_REQUESTS=10
```

ctx.persistence.test.memory.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xmlns:context="http://www.springframework.org/schema/context"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5          http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7      <bean id="persistenceProperties" class="de.invesdwin.common.system.properties.SystemPropertiesDefinition">
8          <property name="systemProperties">
9              <map>
10                 <entry key="javax.persistence.jdbc.driver" value="org.h2.Driver" />
11                 <entry key="javax.persistence.jdbc.url" value="jdbc:h2:mem:invesdwin;DB_CLOSE_ON_EXIT=FALSE" />
12                 <entry key="javax.persistence.jdbc.user" value="sa" />
13                 <entry key="javax.persistence.jdbc.password" value="sa" />
14                 <entry key="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />
15                 <entry key="hibernate.hbm2ddl.auto" value="create" />
16             </map>
17         </property>
18     </bean>
19
20     <import resource="actx.persistence.hibernate.xml" />
21
22 </beans>
```



6. Bindung der Variabilität
- Properties -

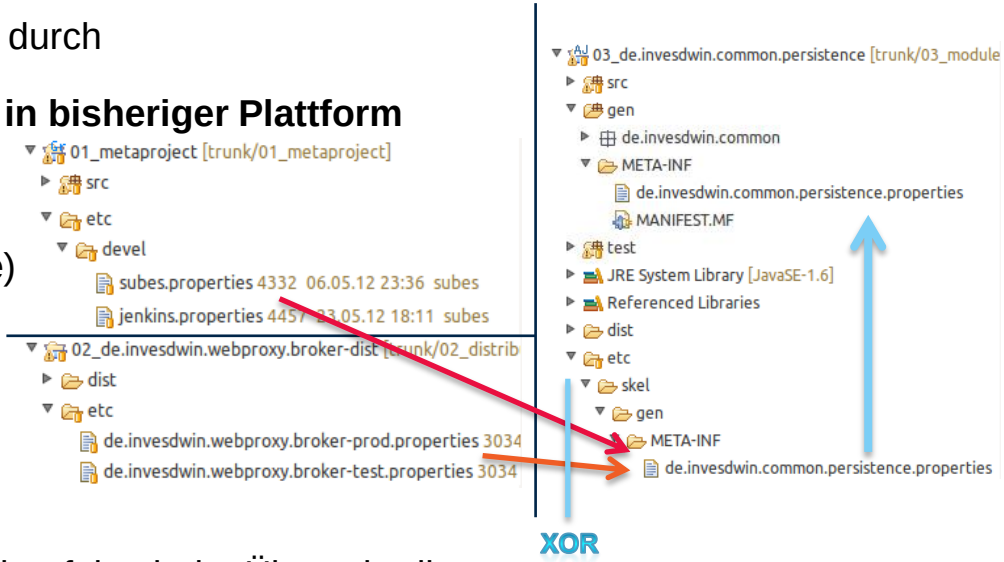
- Default-Werte in Modul-Properties und Ersetzen durch

  Entwickler- und Distributions-Properties wie in bisheriger Plattform

- Build-Prozess **ersetzt** die Properties in

  referenzierten Modul-Projekten (**im gen-Source**)

  und geholten invesdwin-Libs (**im Jar**)



- **Variabler Bindungszeitpunkt** mit folgender Reihenfolge beim Überschreiben:

1. Source-Time: Default-Werte in Properties-Datei

   oder Spring-XML
2. Build-Time: Entwickler- oder Distributions-Properties
3. Load-Time: Spring-XML

   oder Java-Parameter via –Dproperty=value
4. Runtime: Java via Properties.setProperty(„value“)

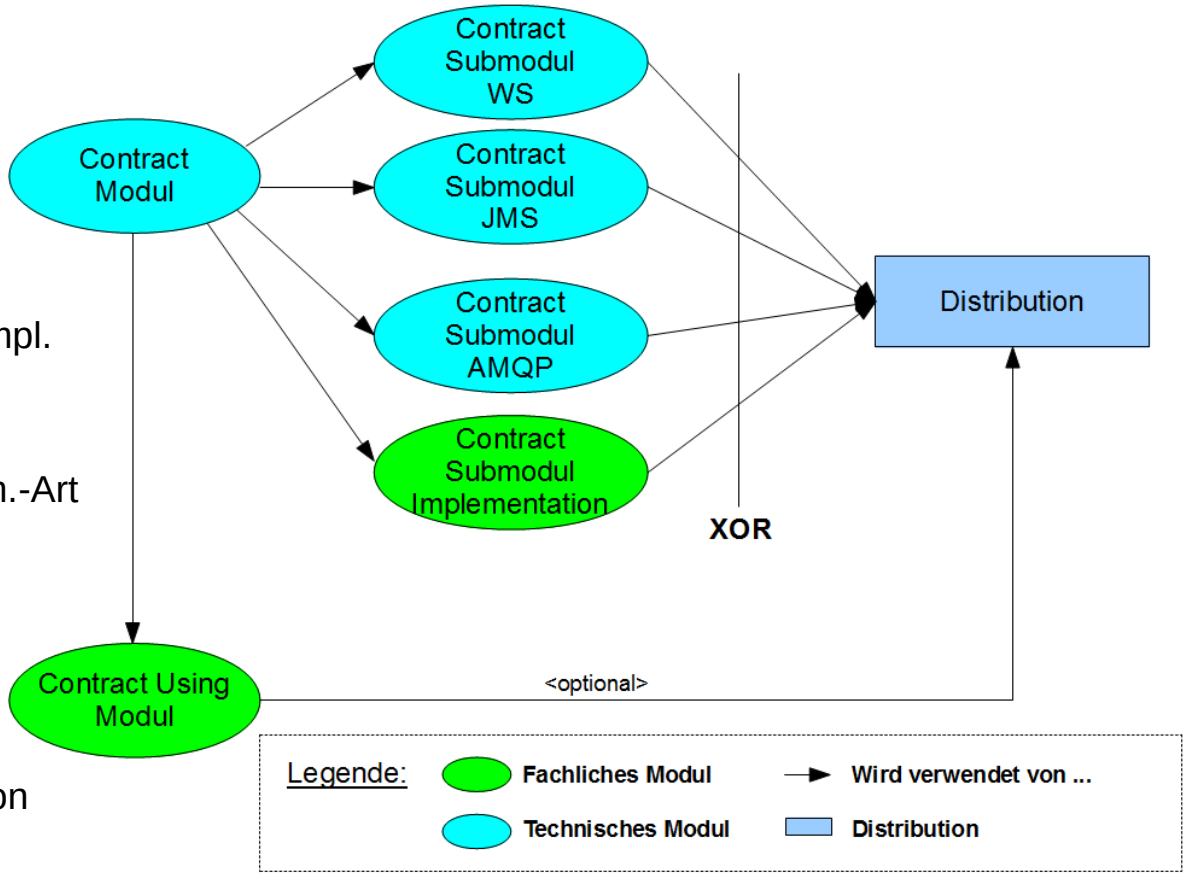
	flexibility	performance	code size	complexity
source time	-	+	+	-
compile time	+	+	+	-
link time	+	+	+	-
load time	++	+	+	+
run time	+++	-	-	+

6. Bindung der Variabilität
- Dependency Management via Contract Module -

- **Buch:** Enterprise Integration Patterns
- **Framework:** Spring Integration (SI)

- **SPL-Nutzung:**
  - Trennung von Schnittstelle und Impl.
  - Forcierte lose Kopplung
  - Techn. Unabhängigkeit zur Komm.-Art

- **Contract:** Interfaces
- **Contract Submodul:** SI-Konfiguration
- **Contract Impl:** Interface-Implementation



- **Möglichkeiten:**
  - Komm-Overhead reduzieren indem Impl in Distribution mitgeliefert wird
  - Mehrere Instanzen der Impl mit Fail-Over oder Load-Balancing

7. DDD und MDA

DDD!

Wo könnte man MDA ansetzen?

7. DDD und MDA
- Konzepte -

- **DDD – Domain Driven Design/Development**
  - Technische Module in Englisch
  - Fachliche Module in Domänensprache
  - Frontend internationalisiert → keine Modulkopie die übersetzt wird
- **MDA – Model Driven Architecture**
  - Bisher kein MDA:
    - Lieber Framework statt Generator → einfachere Wartung, simplerer Build-Prozess
    - Forschungsprojekt:
      - Kaum wiederkehrende Konzepte
      - Anfangsinvestition für Generator würde sich nicht auszahlen
  - MDA machbar:
    - Code-Generierung auf Modulebene, nicht übergreifend
    - Ein Modell pro Modul
    - Erweiterung des „generate“-Build-Schrittes
    - Viel SPL-Produktivitäts-Protenzial

8. Fazit & Diskussion

Vorteile

Nachteile

8. Fazit & Diskussion
- Eure Meinung ist sehr willkommen! -

Vorteile	Nachteile
+ SPL nicht mehr nur Theorie, sondern nutzbar	- Einarbeitungsaufwand
+ Hoher Entwicklungskomfort	- Neue Plattform
+ Hohe Flexibilität	- Neue Technologien
+ Günstigere Wartung bei komplexen Projekten und im Multiprojektumfeld	- Organisationsstrukturen ändern?
+ Crawler-Modul mit verbauter Tricktüte	- Mehr Diversität zu Managen
+ Deckt viele Features/Ansprüche von der bisherigen Plattform ab	
+ Standardisierte Eclipse-Projekte und Build-Skripte	
+ Offene, verbreitete Technologien	
+ Wiederverwendbarkeit als oberstes Ziel	

Vielen Dank für Ihre Aufmerksamkeit