invesdwin

Software-Product-Line Platform

Agenda

1. Introduction
2. Software-Product-Line
3. Development Environment
4. Configuration Management
5. Assets
6. Binding Variability
7. DDD and MDA
8. Conclusion

Feel free to ask questions!

1. Introduction

Story

Motivation

1. Introduction
- Background -

- Bachelor Thesis -> Companies platform vs SPL (Software-Product-Line)?
  - Company decided to stick to their business components
  - Blueprint for projects instead of SPL
- Decided to research SPL further as a side project
  - Domain of Algorithmic Trading, SPL as foundation
    - Research Project
    - Suitable for any Team Size
    - low/no running costs
  - OpenSource **Leverage**
  - **Reuse** has highest priority
  - R&D requires **Flexibility**
  - **Development Comfort** very important
  - It has to be **FUN**!

In development since October 2009

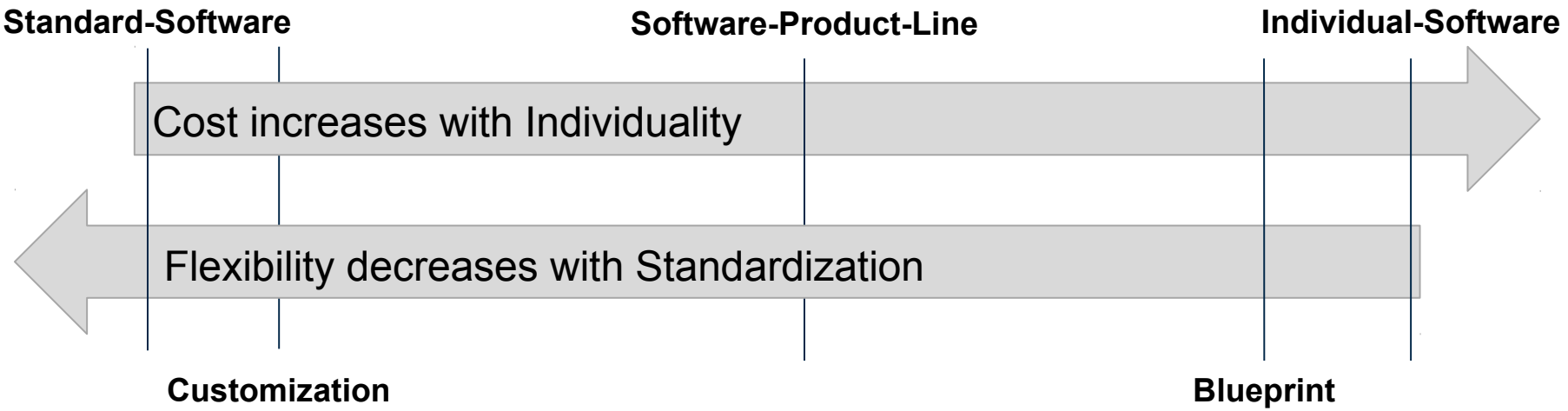
relatively stable since 2012

Open Source since 2016

2. Software-Product-Line

Scientific Background

2. Software-Product-Line
- Classification -

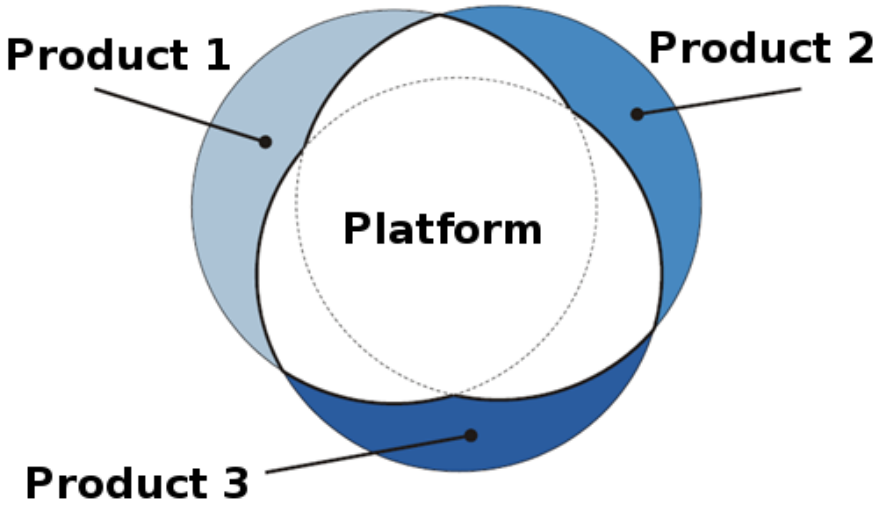


Solution:

- ✓ Reduce costs through **Reuse**
- ✓ Keep flexibility through **Variability**

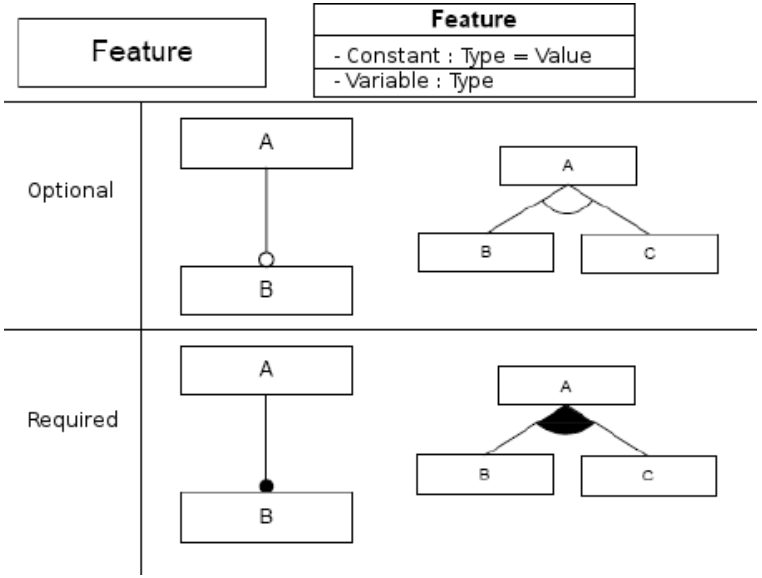
2. Software-Product-Line
- Assets -

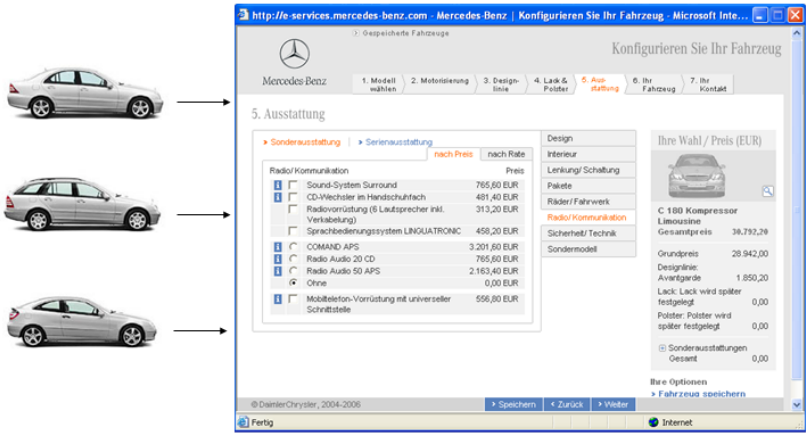
- Assets
  - Artifacts such as **Documentation, Code, Libs, Configuration, Products, Platform**
  - SPL is made of - and manages Assets
- Reuse
  - Artifacts are designed to be reused
  - Creation of **Synergies**
  - Reduction of Development Cost
  - Faster Time-To-Market
  - Avoid Copy/Paste (Artifact duplication)
- Modularity
  - **Bundle Artifacts**
  - Only deliver what the customer paid for!



2. Software-Product-Line
- Variability -

- Variability
  - Interchangeability of used Implementation
  - Selection of **Features**
  - Make price variable and align it to the customer
- Variability Points
  - Decision „**where**“ Variants can be chosen
  - Planned Flexibility Options
  - Configurable or Chooseable
  - Customer makes the Decision here
- Variants
  - Encapsulate Features to be chosen optionally
  - Compare with Car Manufacturing:
    - Coupling Device yes/no
    - Sport Suspension vs Comfort Suspension
    - Combi-Van vs Limousine





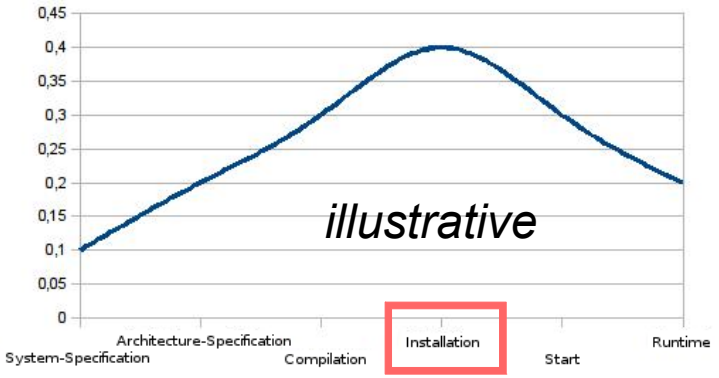
2. Software-Product-Lines
- Binding -

- Binding
  - Variability Point gets placed with a Variant
- Binding Points
  - Decision „**how**“ in Code/Deployment a Variant gets chosen
  - Patterns: Dependencies, Properties, Runtime-Button
- Binding Times
  - Decision „**when**“ a Variant can be chosen
  - Compare with a finished Car:
    - Choose Equipment (when Buying)
    - Update Navigation-Maps (before Engine Start)
    - Activate Sportgear (while Driving)
    - Deactivate Electronic Stability Control (while Driving)

Overview of Binding Times:

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

Usage of Binding Times:



- Early - - Late -

3. Development Environment

What is the SPL realized in?

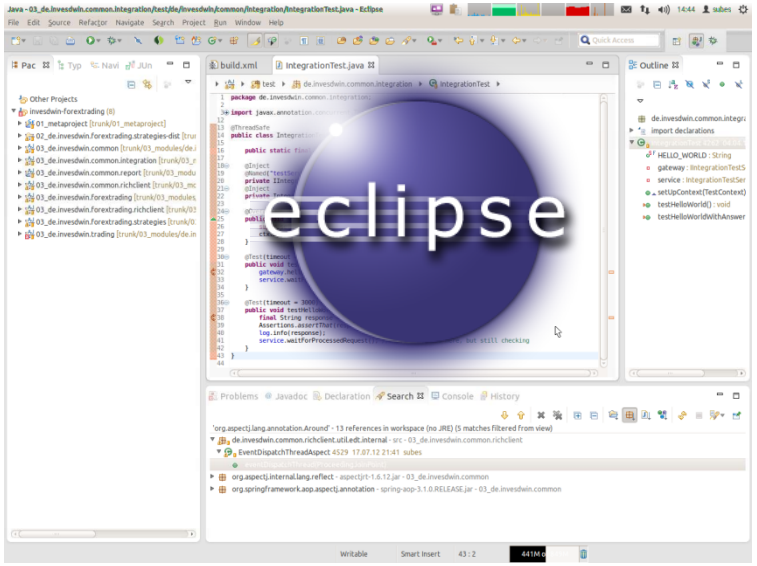
Standard Stuff…

3. Development Environment
- Low Impact -

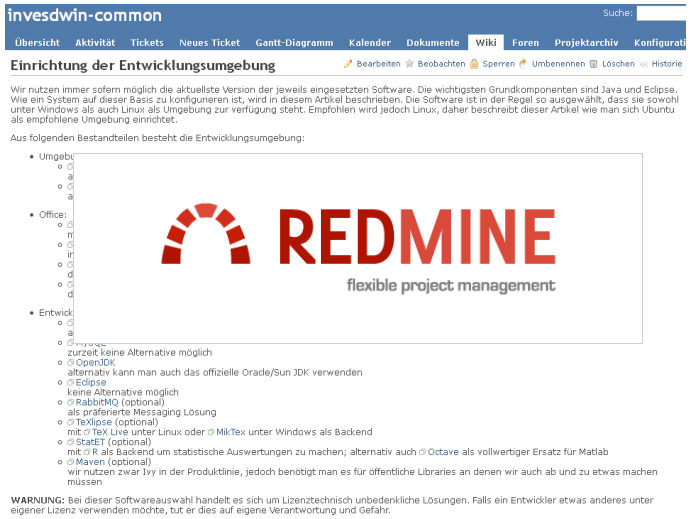
- **OS:** Windows/Linux
- **Versioning:** SVN/Git
- **Language:** Java 8 (started with Java 6)
- **IDE:** Eclipse + Plugins
- **Build & Dependency Mgmt:** Maven

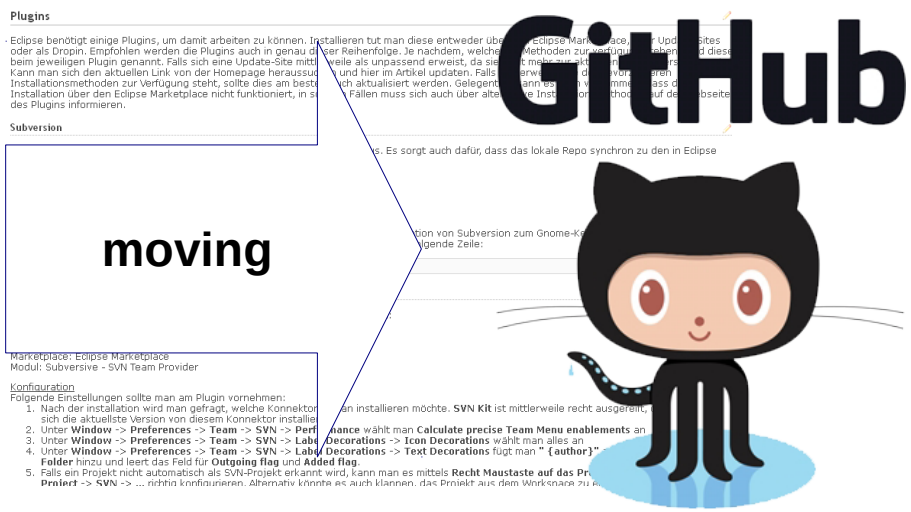
  Earlier Ant+Ivy+Groovy was used (basis for this concept),

  later reimplemented with Maven to improve build times.



- **Documentation:**



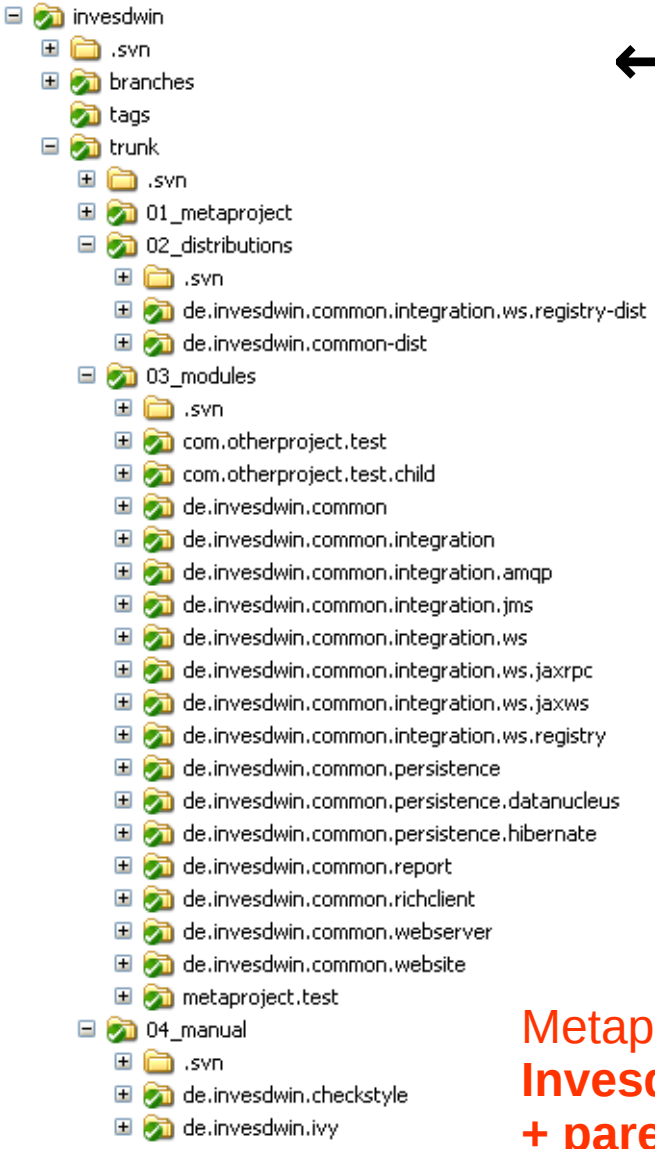


4. Configuration Management

New Approach

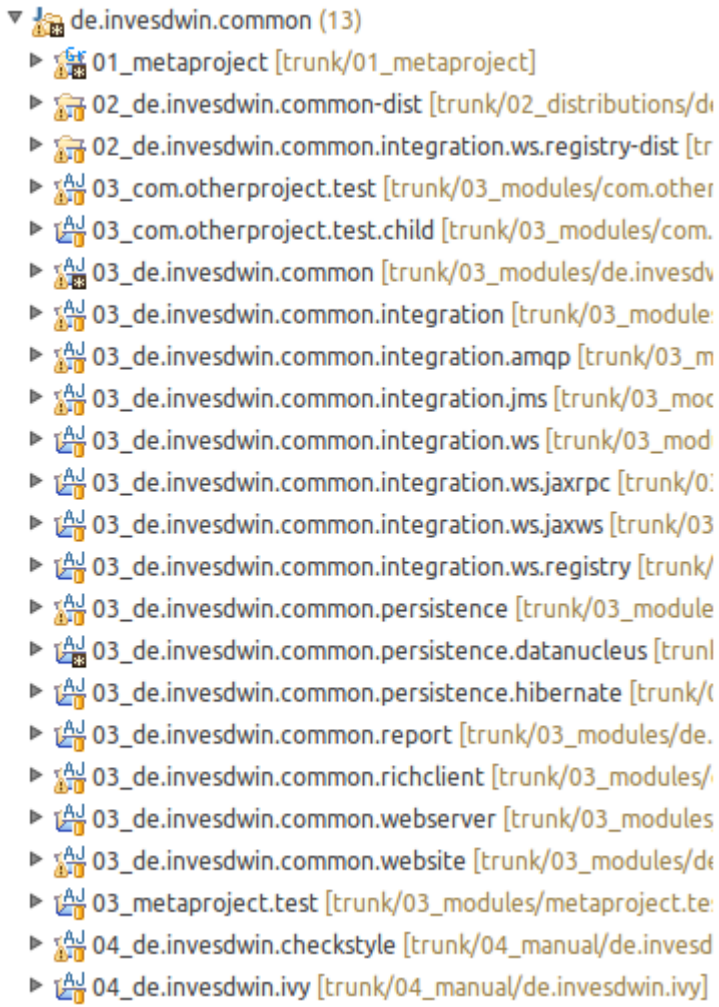
4. Configuration Management
- Overview -

REPLACED BY MAVEN
with a better implementation of this concept



← Explorer

Eclipse →



Product:

- 01 Metaproject
- 02 Distribution
- 03 Module
- (04 manual Module)

Metaproject got turned into:
Invesdwin-maven-plugin
+ parent pom.xmls

4. Configuration Management
- Module I -

- **Reusable Building Blocks for Products**
  - **Technology:** Best Practices, Patterns, Frameworks, Utils, Tools
  - **Domain:** Services, Entities, Logic, Algorithm

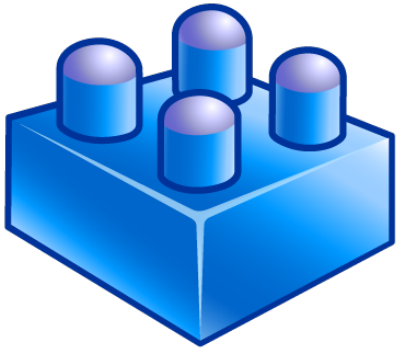
→ Each has its own Eclipse-Project to ensure Modularity

Components

Insurer, Contract, Customer, …

Combine &

make Reusable



Module

Contract Management

4. Configuration Management
- Module II -

- Comparison to previous Platform:
  - **Module > Container**
  - **Upgrade-Path** instead of Copy-Paste
    - → Rather add Variability to a Module, instead of creating another one
    - → Goal: Effectively less Maintenance Cost

Previous Platform had a monolithic architecture where services/beans/components were bundled into „containers“ that had tight coupling among each other.
A module in this sense can bundle multiple containers, thus is more than a container.

Fix Bug multiple times differently	**VS**	Fix Bug once + update Version multiple times

- Apply Lessons Learned from old Business Components:
  - **YAGNI & KISS**
  - Do not develop on **Green Field** and avoid **Over-Engineering**

    → Only add Variability when it is actually needed
  - Enforce **Loose Coupling**

    → Utilize submodules to encapsulate Functionality/Alternatives

4. Configuration Management
- Module III -

- **Why no OSGI?**
  - Dynamic Load/Unload not needed
  - OSGI-Descriptors not maintained well by Open Source Projects
  - Avoid Classpath-Problems (one common Classpath is easier)
  - Jar-Hell and Version-Conflicts already solved by Dependency Management
  - Loose Coupling via „internal“ Packages enforced by Checkstyle-Rule

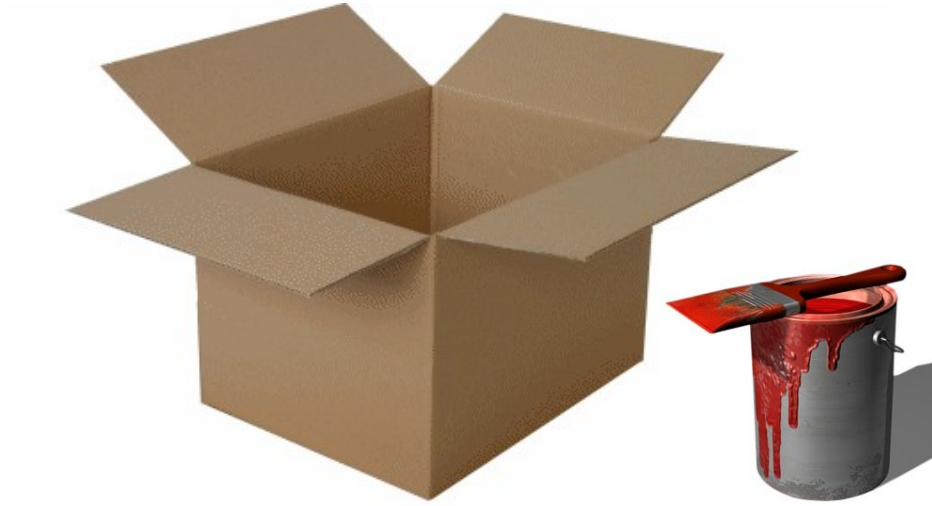
    → reasoned deviations from the „internal“ Package Rule possible

    → without technical challenges

only OSGI Compatible

4. Configuration Management
- Distribution -

- **A configurable, deployable Product**
  - Bundles multiple Modules
  - **Configuration of Variability** via
    - Dependencies
    - Properties
    - Additional Resources
  - Customer Specific
  - Target Environment Specific



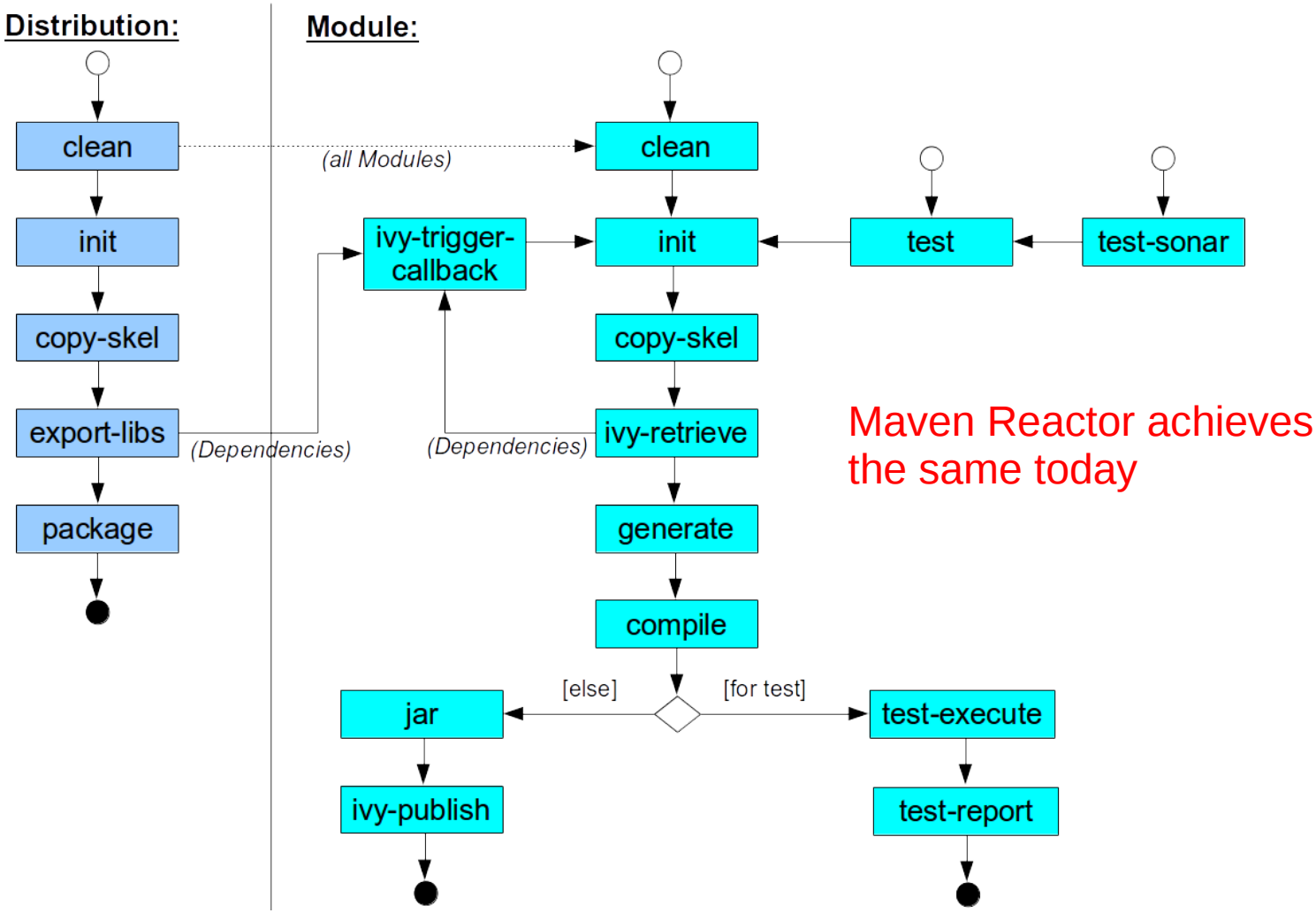
  - Decision about package type only here:
    - Executable **Fat-Jar**
    - **Zip** with launch scripts
    - **War** as Container

→ Example Web-Application: Fat-Jar with embedded Jetty **or** War for Tomcat?

4. Configuration Management - Metaproject I -

REPLACED BY MAVEN
with a better implementation of this concept

- **Definition of unified Build-Process:**

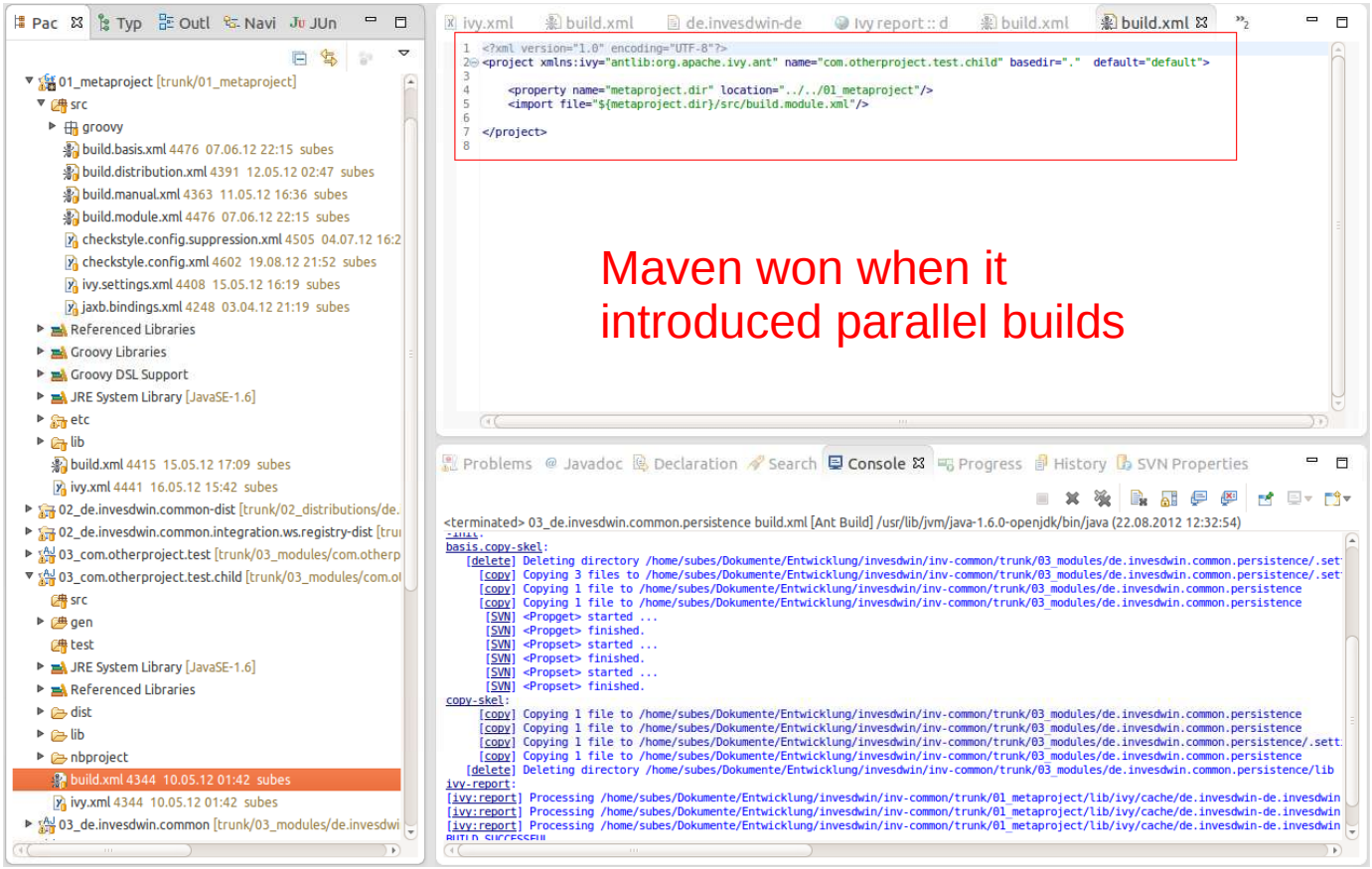


Maven Reactor achieves the same today

4. Configuration Management - Metaproject II -

REPLACED BY MAVEN
with a better implementation of this concept

- **Reusable standardized Ant-Scripts** → maintain in only one place
- **Project Templates for Modules and Distributions** → à la Maven Archetype
- **Other Assets inherit Eclipse-Project-Configuration** → configure only once



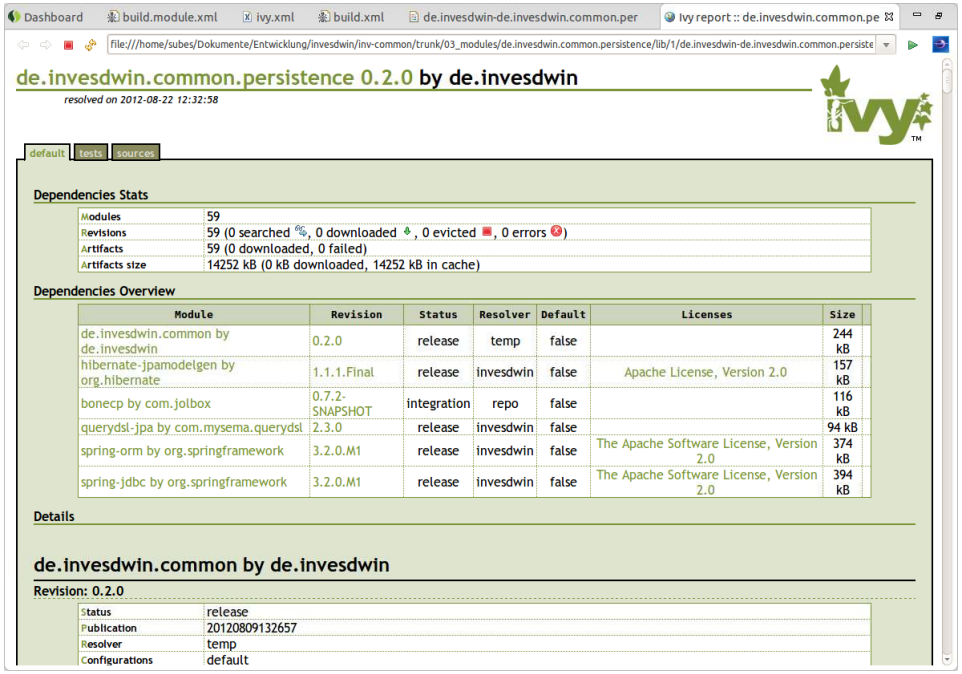
4. Configuration Management - Metaproject III -

REPLACED BY MAVEN
with a better implementation of this concept

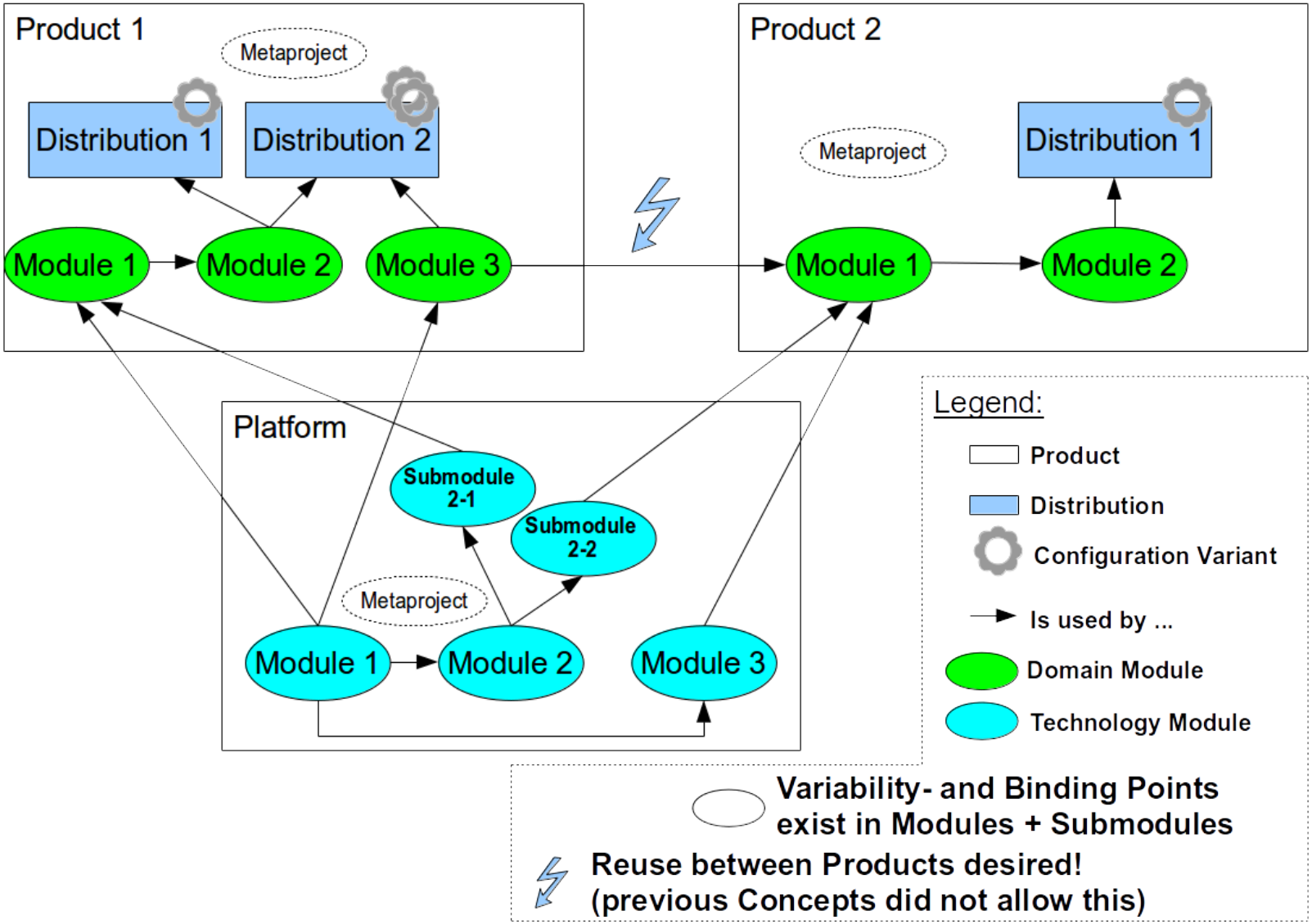
- **Automatically generated Eclipse-Classpath for Dependencies:**
  - Project-References for multiple modules spanning Refactorings
  - Unified Naming of Dependencies (<Module>-<Version>.jar)
  - Sources for all Jars automatically linked
  - AspectJ and other Eclipse-Plugins automatically configured

M2E Plugin for Eclipse achieves the same today

→ Reduction of maintenance effort for Modules, only configuration in **ivy.xml** required!



4. Configuration Management
- Multi-Product-Management -



5. Assets

What exists already?

Platform, Products, Components

5. Assets
- Platform I -

Outdated Information

some components/names have changed
also today there are more modules

- **metaproject**

- **invesdwin-common**

- **invesdwin-common-persistence**

- **invesdwin-common-report**

- **invesdwin-common-richclient**

Better Swing Application Framework
BetterBeansBinding

- **Infrastructure**

- **Dependency Repos**

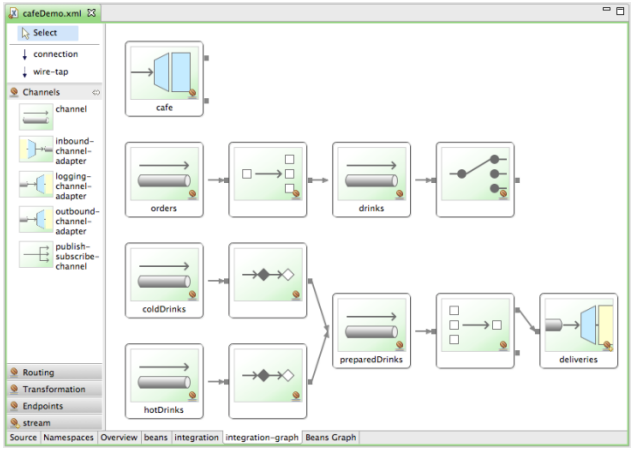
5. Assets
- Platform II -

Outdated Information

some components/names have changed
also today there are more modules

- **invesdwin-common-integration**
  - invesdwin-common-integration-amqp
    - RabbitMQ – Messaging that just works
    - AMQP – Advanced Message Queuing Protocol
  - invesdwin-common-integration-jms
    - ActiveMQ
    - JMS
  - invesdwin-common-integration-ws
    - WS-I Web Services Interoperability Organization
    - Spring Web Services
    - REST
    - -jaxrpc: axis
    - -jaxws: Apache CXF
    - -registry: JUDDI – An Apache Project; JAXR ⊔ UDDI

SPRING INTEGRATION



JAXB

<?xml?>

google-gson
JSON
JavaScript Object Notation

- **invesdwin-common-website**
  - APACHE WICKET
  - λambdaj

- **invesdwin-common-webserver**
  - Powered by jetty://
  - Deployment in Tomcat as WAR
  - configurable in Distribution
  - Apache Tomcat

5. Assets
- Example Product webproxy -

- ínvesdwin-webproxy(-...)

HtmlUnit

GUI-Less browser for Java programs

JAVA SCRIPT

JAKARTA COMMONS
HTTPCLIENT

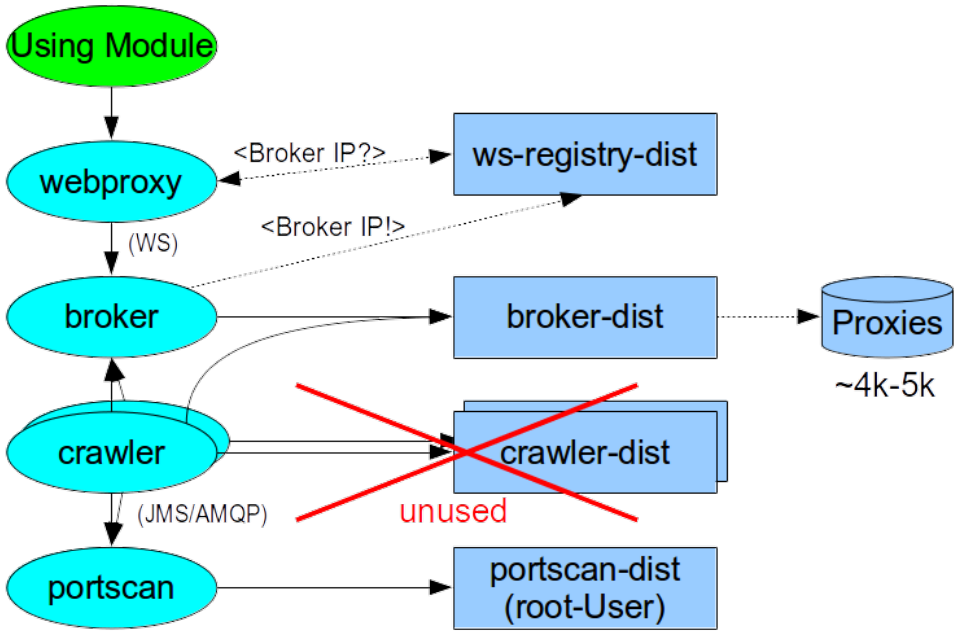
→ Proxies

win/lib/j - Pcap

GeoNames Geo IP

→ Location

- Simplified Context Diagram:



6. Binding Variability

Bootstrap creates Flexibility

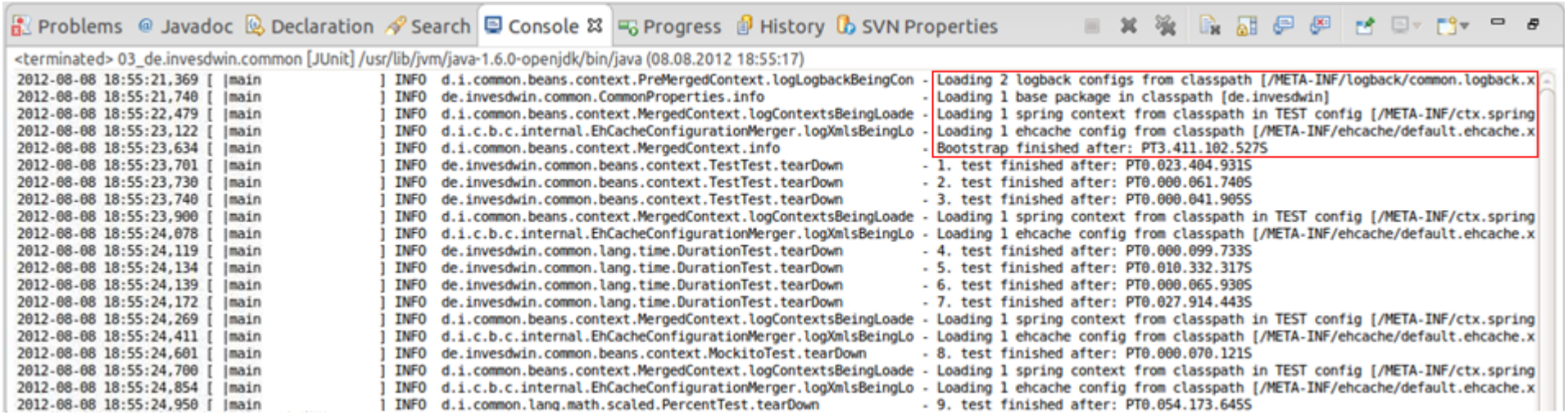
Pattern-Examples

6. Binding Variability
- Application Bootstrap -

- Configuration spread among various Modules
- Common Classpath (no OSGI)
- Two Spring ApplicationContexts
  - Premerged → collect and configure
  - Merged → running Application

- Per JUnit Test-Class a new Bootstrap (fast)

  for Configuration Changes:
  - Selecting Spring-XMLs via IContextLocation
  - In Test via setUpContext()
  - In Mocks/Stubs via IStub



6. Binding Variability
- Properties I -

- **All Properties are System-Properties**
- Thus **available** in:
  - other Properties Files à la Ant ${property}
  - XML (Spring, Frameworks with Commons-Configuration)
  - Java (System.getProperty(„property“))
  - VisualVM (Monitoring, be aware of security!)

*PortscanProperties.java

```
package de.invesdwin.webproxy.portscan.internal;

import java.net.InetAddress;

@Immutable
public final class PortscanProperties {

    public static final InetAddress CHECK_HOST;
    public static final int CHECK_PORT = 80;
    public static final int LOCAL_BIND_PORT;
    public static final Duration ICMP_RESPONSE_TIMEOUT;

    private static final SystemProperties SYSTEM_PROPERTIES = new SystemProperties(PortscanProperties.class);

    static {
        CHECK_HOST = NetworkUtil.toAddress(SYSTEM_PROPERTIES.getString("CHECK_HOST"));
        LOCAL_BIND_PORT = readLocalBindPort();
        ICMP_RESPONSE_TIMEOUT = SYSTEM_PROPERTIES.getDuration("ICMP_RESPONSE_TIMEOUT");
    }

    private PortscanProperties() {}

    private static int readLocalBindPort() {
        final String key = "LOCAL_BIND_PORT";
        final Integer value = SYSTEM_PROPERTIES.getInt(key);
        Assertions.assertThat(NetworkUtil.isPort(value))
                .as(SYSTEM_PROPERTIES.getErrorMessage(key, value, null, "Value must be inclusively between "
                        + NetworkUtil.PORT_MIN + " and " + NetworkUtil.PORT_MAX + "."))
                .isTrue();
        return value;
    }
}
```

de.invesdwin.webproxy.portscan.properties

```
#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
de.invesdwin.webproxy.portscan.internal.PortscanProperties.CHECK_HOST=google.de
de.invesdwin.webproxy.portscan.internal.PortscanProperties.LOCAL_BIND_PORT=44125
de.invesdwin.webproxy.portscan.internal.PortscanProperties.ICMP_RESPONSE_TIMEOUT=3 SECONDS
#For timings see: http://www.networkuptime.com/nmap/page09-09.shtml
de.invesdwin.webproxy.portscan.internal.PortscanProperties.UPLOAD_PAUSE_BETWEEN_PACKETS=0 MILLISECONDS
de.invesdwin.webproxy.portscan.internal.PortscanProperties.UPLOAD_PAUSE_BETWEEN_PACKETS_PER_HOST=0 MILLISECONDS
de.invesdwin.webproxy.portscan.internal.PortscanProperties.RESPONSE_TIMEOUT_BETWEEN_SYN_PACKETS_PER_HOST=500 MILLISECONDS
de.invesdwin.webproxy.portscan.internal.PortscanProperties.MAX_OPEN_ICMP_REQUESTS=25
de.invesdwin.webproxy.portscan.internal.PortscanProperties.MAX_OPEN_SYN_REQUESTS=10
```

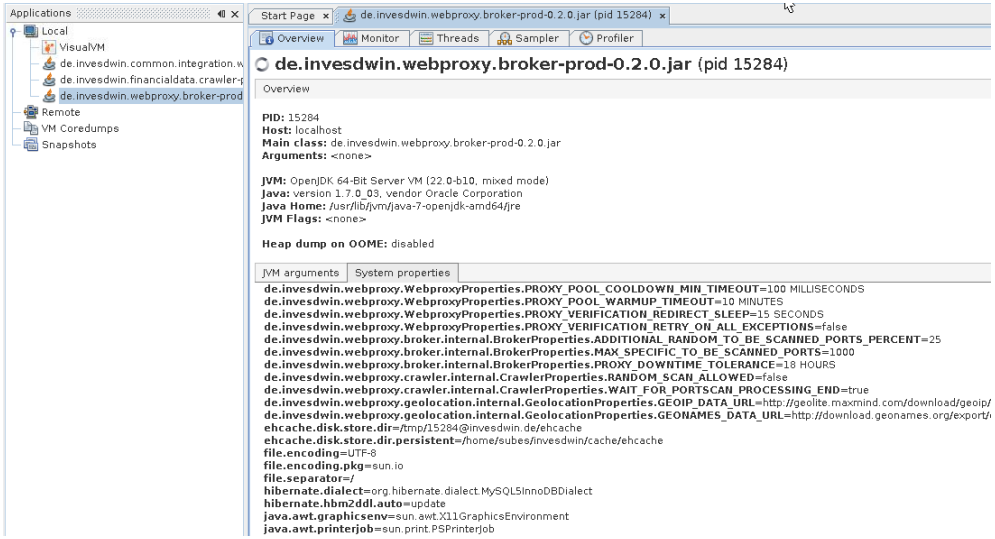
ctx.persistence.test.memory.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="persistenceProperties" class="de.invesdwin.common.system.properties.SystemPropertiesDefinition">
        <property name="systemProperties">
            <map>
                <entry key="javax.persistence.jdbc.driver" value="org.h2.Driver" />
                <entry key="javax.persistence.jdbc.url" value="jdbc:h2:mem:invesdwin;DB_CLOSE_ON_EXIT=FALSE" />
                <entry key="javax.persistence.jdbc.user" value="sa" />
                <entry key="javax.persistence.jdbc.password" value="sa" />
                <entry key="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />
                <entry key="hibernate.hbm2ddl.auto" value="create" />
            </map>
        </property>
    </bean>

    <import resource="actx.persistence.hibernate.xml" />

</beans>
```



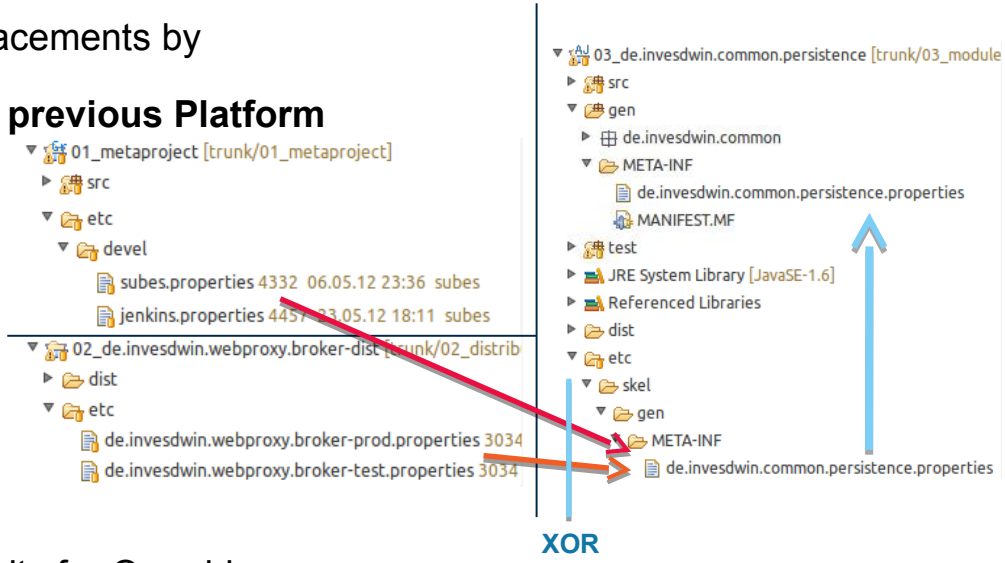
6. Binding Variability - Properties II -

Now per Module Overrides with Maven Implementation

- Default-Values in Module-Properties and Replacements by

Developer- and Distribution-Properties as in previous Platform

- Build-Process **replaces the** Properties in references Module-Projects (in **gen-Source**) and retrieved invesdwin-Libs (in **Jar**)



- **Variable Binding-Time** with the following Priority for Overrides:

1. Source-Time: Default-Values in Properties-File
   or Spring-XML
2. Build-Time: Developer- or Distribution-Properties
3. Load-Time: Spring-XML
   or Java-Parameter via –Dproperty=value
4. Runtime: Java via Properties.setProperty(„value“)

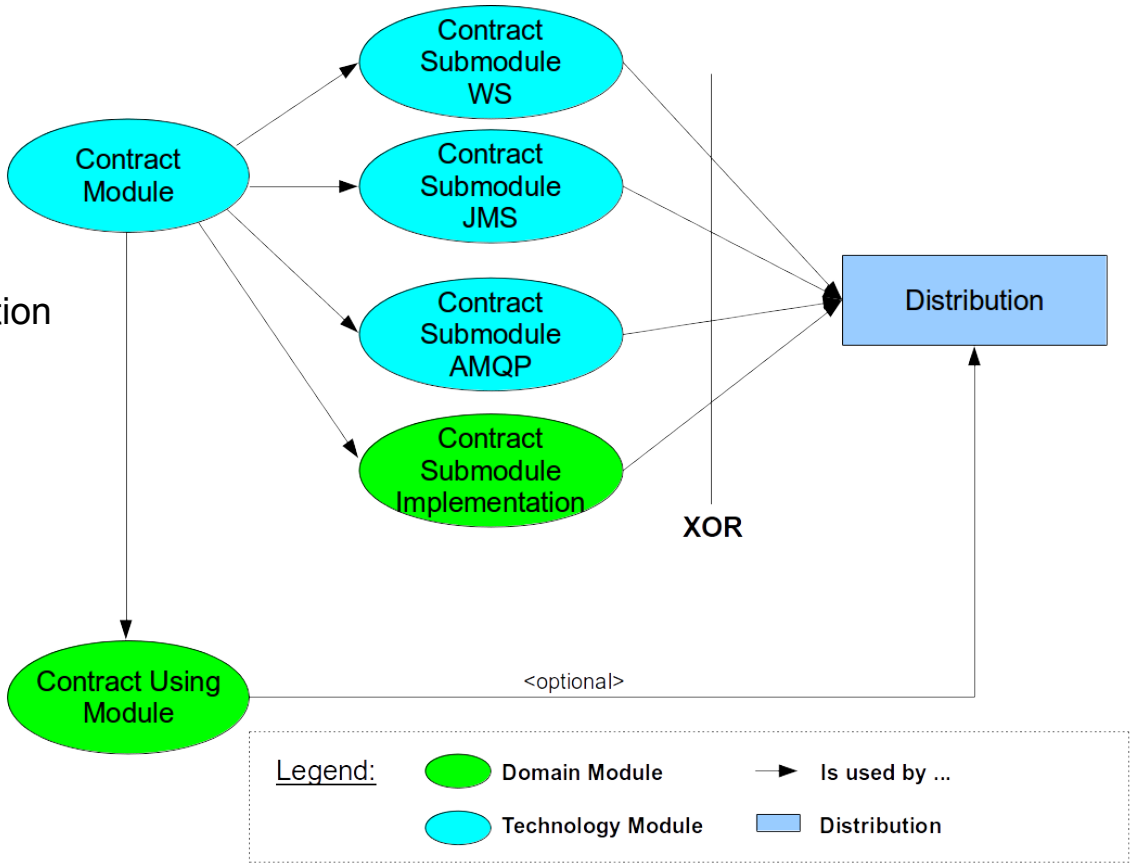
	flexibility	performance	code size	complexity
source time	-	+	+	-
compile time	+	+	+	-
link time	+	+	+	-
load time	++	+	+	+
run time	+++	-	-	+

6. Binding Variability
- Contract Modules -

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

- **SPL-Usage:**
  - Separate Interface and Implementation
  - Enforce loose coupling
  - Technological indifference

    to Communication-Type
- **Contract:** Interfaces
- **Contract Submodule:** SI-Configuration
- **Contract Impl:** Interface-Implementation



- **Possibilities:**
  - Communication-Overhead reduced by putting Implementation in Distribution directly
  - Multiple Instances of Impl with Fail-Over or Load-Balancing

7. DDD and MDA

DDD!

Where could MDA be used?

7. DDD und MDA
- Concepts -

- **DDD – Domain Driven Design/Development**
  - Technology Modules in Englisch
  - Domain Modules in Domain-Language
  - Frontend internationalized → no Module copies that get translated
- **MDA – Model Driven Architecture**
  - As of now no MDA:
    - Prefer Framework over Generator → easier Maintenance, simpler Build-Process
    - Research Project:
      - Less recurring Concepts
      - Initial invest for Generator would not provide ROI fast enough
  - MDA possible:
    - Code-Generation at Module-Level, not all-encompassing
    - One Model per Module
    - Extension of „generate“-Build-Step
    - Lots of SPL-Productivity-Potential

8. Conclusion

Benefits

Drawbacks

8. Conclusion
- Your opinion is very welcome! -

Benefits	Drawbacks
+ SPL not just theory, but usable	- Learning Curve
+ High Development Comfort	- New Platform
+ High Flexibility	- New Technologies
+ Cheaper Maintenance for complex Projects and in Multi-Project-Environment	- Change of Organizational Structure needed?
+ Matches most Features/Requirements of previous platform	- More Diversity to manage
+ Standardized Eclipse-Projects and Build-Scripts	
+ Open, widely used technologies	
+ Reuse as highest goal	

Thanks a lot for your attention!