

# Advanced Software Engineering

---

26.11.2015:

Five challenges you solve for every project

DI Stefan Strobl

stefan.strobl [at] inso.tuwien.ac.at



# Introduction

- Frameworks offer a lot of guidance to solve common problems (patterns)
- Crucial areas need qualified decisions to tailor the framework to your needs
- Good frameworks provide you with a series of alternatives, bad ones try to force “the right” solution on you
- Choosing suitable framework(s) and tailoring it to the scenario at hand is an integral part of any IT project



# Let us start with a regular business service

```
class MyServiceImpl implements MyService {  
  
    public ResultDTO executeBusinessMethod(InputDTO input)  
        throws MyBusinessException {  
        validateInput(input);  
        Result result = performCalculation(input);  
        if(result.failed()) {  
            throw new MyBusinessException(„error.key“);  
        }  
        updateDatabase(result);  
        return transform(result);  
    }  
    // private methods  
}
```



# Agenda

- While using (modern) application frameworks, cross cutting concerns usually need tailoring
- Handling these concerns separately from your business logic is a major factor for retaining clean, readable code
- The following five areas will be addressed today
  - Transaction Management
  - Logging & Auditing
  - Security
  - Error Handling
  - Internationalization & Localization
- Some common pitfalls will be highlighted along the way



# Transaction Management - Models

- Models describe the expected transactional behaviour
- Essentially: who is responsible for a transaction?
  - Local Transaction Model
    - Underlying database (auto commit)
    - Connection based
  - Programmatic Transaction Model
    - Logical transactions
    - Transaction manager
  - Declarative Transaction Model aka Container Managed Transactions (CMT)
    - The developer specifies the behaviour
    - The container handles the transaction



# Transaction Management - Strategies

- The following strategies can be used or customized to handle transactions
- Essentially: what is considered a unit of work?
  - Client Orchestration: for fine grained APIs
    - Web framework
    - Portal application
    - Workflow or BPM component
  - API Layer: for coarse grained methods
    - Web services
    - Message handlers
  - High Concurrency / High Speed Processing: optimizations
    - Shortest possible transaction scope



# A word on distributed transactions

- Distributed or global transactions allow atomic behavior over more than one resource (database, message queue, ... )
- Specified in the XA (eXtended Architecture) standard by “The Open Group”
- XA uses the 2-phase-commit (2PC) protocol to ensure atomic commits
- Java Transaction API (JTA, JSR-907) is based on the XA standard
- Should only be used when absolutely necessary
  - Distributed transactions can NOT cover all cases of (physical) failure
  - Many problems can be solved by fine grained, manual control of commit sequence
  - Use established Enterprise Integration Patterns (EIP) – e.g. for message queuing: Idempotent Consumer



# Declarative Transactions – Spring example

- Spring @Transactional annotation
- Transactional Interceptor
  - Begin/commit transaction
  - Join existing transaction
  - Rollback in case of (unchecked) exception
- Correct configuration of transactions is crucial

```
<<Transaction Interceptor>>
```

```
tx = getOrCreateTx().begin()
```

```
<<Bean>>
```

```
@Transactional  
void doSomething()  
    // execute code
```

```
tx.commit()  
// handle possible errors
```

# Pitfall – declarative transactions & interceptors

```
public class Bean {  
  
    @Inject Bean bean;  
  
    void a() {  
        bean.b();  
    }  
  
    @Transactional  
    void b() {  
        doSomething();  
    }  
}
```

- Calling b() works as expected
- Calling a() does not work
- WHY?
- → Solution:  
**Self reference to obtain proxy**



# Transaction Management - Summary

- Problem:

How to choose the right transaction management strategy for your project?

- Steps:

- If running inside a suitable container, declarative transactions are usually the safest bet
- Make sure you understand the implications a managed persistence context entails
- Completely managing transactions manually results in a lot of boiler-plate code and is error prone
- CMT makes automating tests hard(er) – an embedded container is necessary to provide a suitable environment
- XA Transactions only when absolutely necessary



# Apply to our business service

```
class MyServiceImpl implements MyService {  
  
    @Transactional  
    public ResultDTO executeBusinessMethod(InputDTO input)  
        throws MyBusinessException {  
        validateInput(input);  
        Result result = performCalculation(input);  
        if(result.failed()) {  
            throw new MyBusinessException(„error.key“);  
        }  
        updateDatabase(result);  
        return transform(result);  
    }  
    // private methods  
}
```



# Logging & Auditing - Definition(s)

## ■ Logging:

- Technical, text based output primarily used for detecting and debugging problems
- Amount of detail can usually be configured at runtime
- Output is not (easily) understandable for regular users of the system
- Output is not suitable for automated processing
- Short term retention (months)

## ■ Auditing

- Domain specific, fine grained and structured output for tracing user activity
- Requirements are specified by legal and/or company specific policies
- Used by (limited) end user group (e.g. internal revision)
- Frequently coupled with long term retention requirements (10+ years for regular businesses, 30+ for medical and similar)



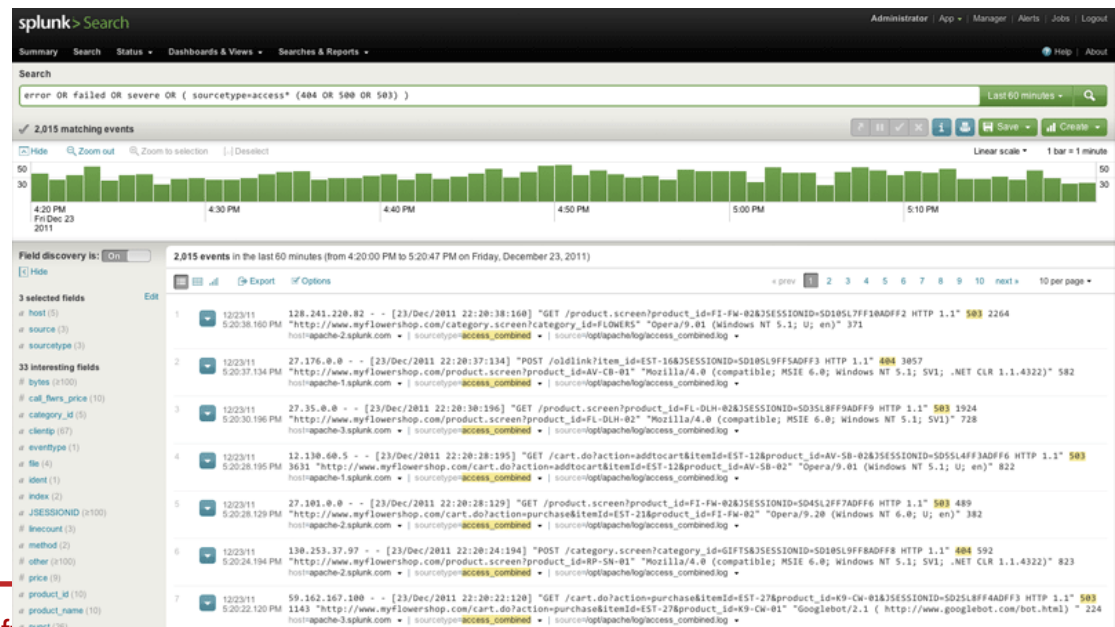
# Logging Technical

- Plenty of frameworks and meta-frameworks available
- Examples: Log4j, Logback, JUL - or meta: slf4j
- Some performance implications to consider
  - Avoid unnecessary string concatenations
  - Avoid expensive operations to generate log statements (e.g. only print id instead of loading a data set from the database)
  - Too much log output negatively affects performance
- Absolutely avoid writing directly to stdout or stderr - it takes away your flexibility
- Logging also has to be configured correctly in the frameworks and libraries you use (can be quite tricky)
  - Scenario: Framework A uses log4j, Framework B uses JUL



# Reading log files

- Clustering/load balancing/distribution can make it infinitely harder to trace an error
- Provide contextual information (userId, sessionId, transaction- or requestId, threadId) for correlation
- Make use of tooling for easy reading of large amounts of generated log files
  - Syslogd
  - Logstash
  - Splunk (commercial)



# Good Log output

- Frameworks cannot ensure "good" log output, only enable it
- Limit the amount of log levels used (usually DEBUG, INFO, WARN and ERROR are sufficient)
- Establish and enforce clear rules when to use which log category
- Make sure you know how to adjust the log level at runtime
- The possibility to activate extensive/verbose debug output can save your day in critical situations
- Automatically adjusting the log level according to the current situations can help to make problems visible earlier
- Contextual information helps you to trace the flow of execution
- Always provide a reference (“user created” vs. “user (id=1234) created”)
- Make sure to tailor the log output after gathering experience in practice



# Auditing - Technical

- No or little framework support available
- Requirements are usually too diverse/specific for "generic" solutions
- Auditing is a "domain specific" feature - requiring regular specification, quality assurance, etc
- Frequently depends on already existing (in house) product
- If using interface to external product make sure to have proper SLAs in place
- Often quite expensive to generate and deliver auditing information
- Try to work asynchronously as often as possible to reduce effect on execution time



# Logging & Auditing - Summary

- Problem:  
How to correctly configure logging (and auditing)?
- Steps:
  - Select a good logging framework with a clear and concise API and good performance
  - Configure the logging framework to give you just enough information
  - Define clear and simple rules for all developers to follow
  - Ensure the quality of your log output on a regular basis
  - Ensure the logs are easily and quickly accessible
  - Auditing, if necessary, is a regular functional requirement
  - Make sure auditing does not affect your performance



# Apply to our business service

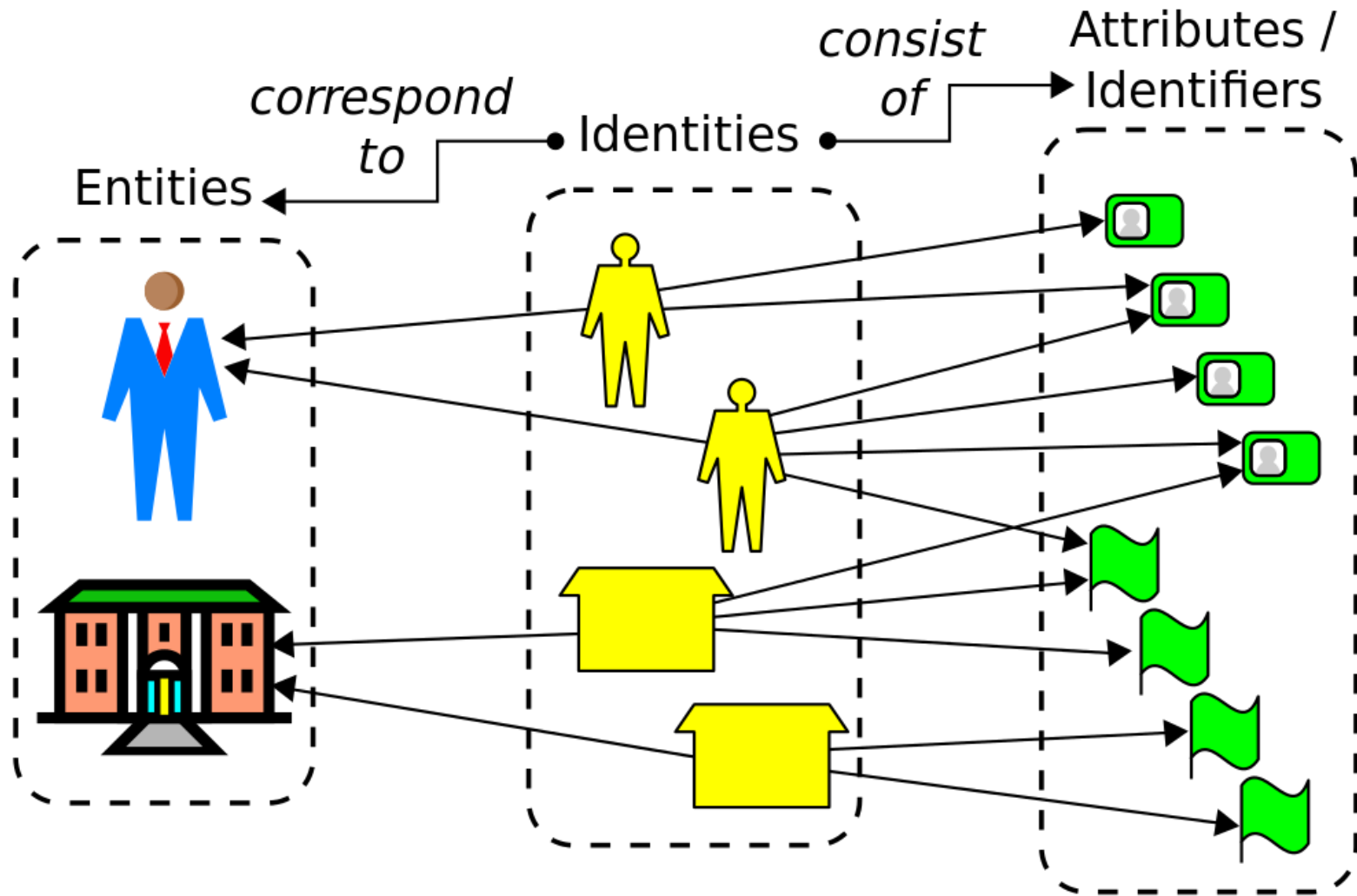
```
class MyServiceImpl implements MyService {

    @Transactional
    @TraceMethodCall(level=INFO)
    @AuditResult
    public ResultDTO executeBusinessMethod(InputDTO input)
        throws MyBusinessException {
        validateInput(input);
        Result result = performCalculation(input);
        if(result.failed()) {
            throw new MyBusinessException(„error.key“);
        }
        updateDatabase(result);
        return transform(result);
    }
    // private methods
}
```



- Two main “tasks”
  - Authentication
  - Authorization
- For each many solutions depending on your requirements
- Identity Management
  - How to handle user related data?
  - Database, Custom application
  - LDAP/Active Directory
  - Managing users and granted authorizations can become very complex with a growing number of users and actions

# Identity Management



# Authentication

- Username/Password
  - Easiest, both to implement and use
  - Still most wide-spread, though increasingly insecure
- Certificate based (client authentication)
  - Complex to roll out & manage
  - used in high security environments
- Token based / Single Sign On (SSO)
  - e.g. SAML, OpenID or proprietary (LTPA)
- Upcoming: smart card, biometric
  - “Bürgerkarte”, Fingerprint-Sensor
  - Usually needs client side support (driver, ... )



# Authorization

- Role based access control (RBAC)
  - Frequently used for resource based systems (e.g. Windows/\*nix file systems)
- Permission based
  - Simple action based
  - Complex expressions
- Access control lists (ACL)
  - Delivers fine grained control on an (object) instance level
- Rule based
  - Suitable for complicated and frequently changing business requirements



# Authorizations - what to choose

- Permission/role based: simple security requirements
  - Usually easy to govern
  - Well supported by standard technologies
  - Performance implications are minimal
- ACLs: complex, data set centric requirements
  - e.g. patient data, individual accounts with user managed grants
  - complex system for (automatically) granting access
  - can have significant performance implications



# Declarative Security

- Provided by containers out of the box
- Forces usage of provided authentication mechanisms and APIs (JAAS for Java)
- @RolesAllowed specified by JSR-250
- Spring and Deltaspike extend the mechanisms to provide more flexibility (e.g. Expression based security checks)
- Decide on a scope for your security, e.g.
  - API level
  - Client/User Interface level



# Summary - Security

- Problem

How to choose an appropriate access control mechanism?

- Steps:

- Identify available resources, especially data
- Reuse existing infrastructure or frameworks over building your own
- Decouple authentication, authorization and identity management
- Keep your business code clean of provider specific dependencies
- Make sure to adhere to organizational requirements
- Consider performance implications when using more complex authorization methods



# Apply to our business service

```
class MyServiceImpl implements MyService {

    @Transactional
    @TraceMethodCall(level=INFO)
    @AuditResult
    @RolesAllowed(SUPER_POWER_USER)
    public ResultDTO executeBusinessMethod(InputDTO input)
        throws MyBusinessException {
        validateInput(input);
        Result result = performCalculation(input);
        if(result.failed()) {
            throw new MyBusinessException(„error.key“);
        }
        updateDatabase(result);
        return transform(result);
    }
    // private methods
}
```



# Error Management

- User error vs. program error
- Program flow vs. exception
- Connected with logging and UI
- How and what to communicate to the end user
- Also connected with (client side) validation
- Three types of Exceptions
  - Checked Exceptions
  - Unchecked Exceptions
  - Errors



# Handling Lower Level Exceptions

- Ask yourself the following questions:
  - Does this method have enough information to properly handle this exception? If yes, handle it. Otherwise...
  - Does the caller have enough information to properly handle this exception? If yes, re-throw. Otherwise...
  - Does the caller need to specifically handle failures in the operations from this component? If yes, re-throw as nested within a component exception subclass. Otherwise...
  - Re-throw as unchecked.

Source: <http://stackoverflow.com/questions/5865547/java-error-handling>



# Exception Translation Pattern

- Do not expose “lower level” exceptions to upper layers of code to avoid “API bleeding”
- If exception cannot be handled at the current stage, wrap it in a module specific exception
- Easily done as (custom) Interceptor/Aspect + Annotation

```
try {  
    doSomething();  
} catch (LowerLevelException e) {  
    throw new MyBusinessException("message", e);  
}
```



# Exception Handling – Anti-Patterns

- Log and Throw
  - Do either one or the other!
- Catching or Throwing “Exception”
  - It’s like a fishers net – you do not know what you will catch
- Destructive Wrapping
  - Always pass the causing exception
- Catch and Ignore
  - This one will come back to bite YOU
- Throw from within finally
  - Will swallow any other exception



# Pitfall: Checked Exceptions and Transactions

- One of the most common data integrity problems
- EJB and Spring do not rollback on checked exceptions

```
@Transactional (rollbackFor=MyCheckedException.class)
public void blockIn(Flight flight, Time time)
    throws MyCheckedException {
    try {
        updateFlight(flight, time);
        // throws MyCheckedException
        updatePosition(flight);
    } catch (MyException e) {
        //do some error handling
        ctxt.setRollbackOnly(true);
        throw e;
    }
}
```

Spring

EJB

# Summary – Error Management

## ■ Problem

How to consistently manage user and program errors in your system?

## ■ Steps:

- Do not use exceptions to direct regular program flow
- A good exception (handling) strategy will make your code usable and maintainable
- Consistency is key for maintainability and readability
- Do not overpower your end user with incomprehensible information
- At the same time make it easy for the user to report a problem
- Watch for common pitfalls



# Apply to our business service

```
class MyServiceImpl implements MyService {

    @Transactional
    @TraceMethodCall(level=INFO)
    @AuditResult
    @RolesAllowed(SUPER_POWER_USER)
    @TransactionBarrier
    public ResultDTO executeBusinessMethod(InputDTO input)
        throws MyBusinessException {
        validateInput(input);
        Result result = performCalculation(input);
        if(result.failed()) {
            throw new MyBusinessException(„error.key“);
        }
        updateDatabase(result);
        return transform(result);
    }
    // private methods
}
```



- **Internationalization (I18n)**

The preparation of a (software) product for use in the global market, usually done only once.

- **Localization (L10n)**

Performing specific adaptations necessary to launch a product in a specific locale.

# Typical Focus Points

## ■ Language & Text

- Character encoding (UTF-8 is de facto standard)
- Orientation: Left to right vs. right to left vs. vertical
- Images, Sorting
- Pluralisation

## ■ Culture

- Names and titles
- Weights and measurements, paper sizes
- Telephone, Addresses, Postal codes

## ■ Conventions

- Currency format
- Date, Time, Time zones and Calendars
- Number format



- **Java built in (Resource Bundles)**
  - Foundation for most other frameworks
  - ResourceBundle consisting of several property files (one per supported language + one for fallback)
  - `String.format()` or `MessageFormat.format()` to properly handle parameterized messages
- **Cal10n**
  - Builds on built in Java mechanisms
  - Provides (some) type safety by using Enumerations as key
  - Configuration via annotations
  - Built in support for formatting messages
- **JavaEE/CDI/Deltaspike**
  - Builds on built in Java mechanisms
  - Provides (some) type safety by using interfaces (as keys)
  - Support for injection of message bundles

# Java Resource Bundles – Key Issues

- Property files (by Java specification) are ISO-8859-1 (Latin-1) encoded
  - Hell, if more than one platform is involved (e.g Win, Linux, Mac OS)
  - Only way to use characters not available in Latin-1 is to use Unicode escaped characters (ü -> \u00FC)
- Type Safety
  - Properties are referenced as strings
  - Missing properties can only be discovered at runtime
  - Unused properties usually remain
- Adding new and editing properties is a manual process
  - Define property in multiple property files (maybe dozens of languages)
  - Reference the property by the key
- Finding non-translated Strings inside the code is even harder
  - No-compiler checks etc



## Pluralization

- “0 Personen” vs. “1 Person” vs. “5 Personen”

```
pCount={0}{0,choice,0#Personen|1#Person|1<Personen}
```

- Supported for “easy” languages (e.g. English, German) in Java MessageFormat
- Third party library needed for complex languages (e.g. Polish, Russian) – ICU4J
- Example: Different derivations of a word for single, a few and many (e.g. 1 auto; 2, 23, 54 auta; 5, 17 aut)

```
car={0}{0,plural,one{auto}few{auta}many{aut}other{aut}}
```

## Collation

- Some languages do not have the expected 1:1 mapping of lower-case to upper-case letters
- E.g. Turkish has two lower case “ı” and “i” as well as two different uppercase: “İ” and “I”
- This results in

```
“portrait”.toUpperCase().equals(“PORTRAIT”) == false
```

**Attention:** This also happens on non localized Strings if the locale of the JVM is switched (e.g. because a Java program runs on a Windows instance with Turkish localization)

# Summary Internationalization & Localization

## ■ Problem

How to prepare you product for an increasingly global market?

## ■ Steps

- Consider Internationalization right from the beginning, especially
  - Character encoding
  - Locale & Timezone settings
- Know your target market to avoid unnecessary overhead
- I18n is not only translatable text
- Even if initially only one language is the target, investing in Internationalization can pre-empt changing requirements
- Make use of tools & frameworks



# Apply to our business service

```
class MyServiceImpl implements MyService {

    @Transactional
    @TraceMethodCall(level=INFO)
    @AuditResult
    @RolesAllowed(SUPER_POWER_USER)
    @TransactionBarrier
    @InterpolateMessages
    public ResultDTO executeBusinessMethod(InputDTO input)
        throws MyBusinessException {
        validateInput(input);
        Result result = performCalculation(input);
        if(result.failed()) {
            throw new MyBusinessException(„error.key“);
        }
        updateDatabase(result);
        return transform(result);
    }
    // private methods
}
```



# Our final business service

```
class MyServiceImpl implements MyService {  
  
    @Transactional  
    @TraceMethodCall(level=INFO)  
    @AuditResult  
    @RolePermission(permission=POWER_USER)  
    @TransactionBarrier  
    @InterpolateMessages  
    public ResultDTO executeBusinessMethod(InputDTO input)  
        throws MyBusinessException {  
        validateInput(input);  
        Result result = performCalculation(input);  
        if(result.failed()) {  
            throw new MyBusinessException(„error.key”);  
        }  
        updateDatabase(result);  
        return transform(result);  
    }  
    // private methods  
}
```

**Cross Cutting Concerns**

**Business Logic**



# Resources

- Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Joshua Bloch. 2008. *Effective Java (2nd Edition) (The Java Series) (2 ed.)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- <http://activemq.apache.org/should-i-use-xa.html>
- <https://today.java.net/article/2006/04/04/exception-handling-antipatterns>
- <http://lostechies.com/derickbailey/2011/05/24/dont-do-role-based-authorization-checks-do-activity-based-checks/>
- <http://site.icu-project.org/>
- <http://stuartgunter.wordpress.com/2011/08/14/even-better-java-i18n-pluralisation-using-icu4j/>
- <http://weblog.tetradian.com/2012/11/03/on-metaframeworks-in-ea/>
- <http://www.chrisonea.com/2012/10/24/frankenframeworks/>
- <http://blog.opengroup.org/2011/03/10/enterprise-architecture%E2%80%99s-quest-for-its-identity/>
- <http://pp.info.uni-karlsruhe.de/uploads/publikationen/constantinides04eiwas.pdf>

