Typical Java Problems in the Wild

Eberhard Wolff

SpringSource

SpringSource Solution

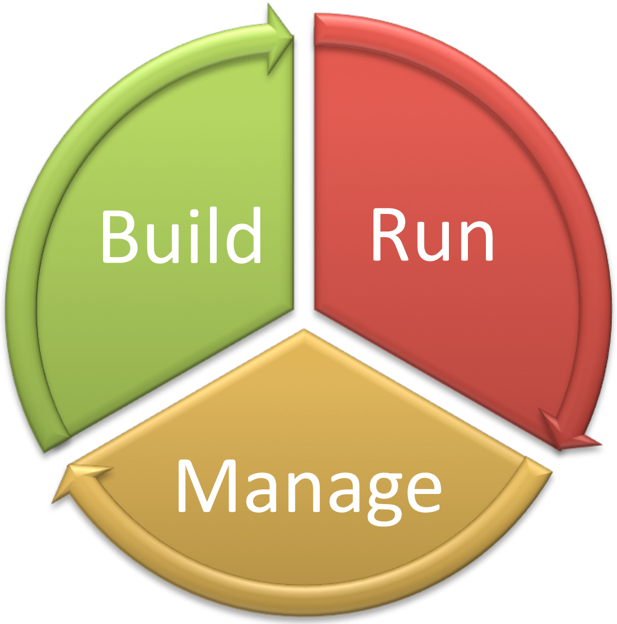
Unifying the Application Lifecycle: from Developer to Datacenter

High Productivity Tools

Spring Enterprise

Groovy and Grails

SpringSource Tool Suite



Lean Powerful Runtimes

SpringSource tc Server

SpringSource dm Server

SpringSource http Server

Application Infrastructure Management

SpringSource Hyperic HQ

SpringSource Hyperic IQ

About me

- Regional Director German speaking region and Principal Consultant
- Author of several articles and books
- First German Spring book
- Speaker at national and international conferences
- Eberhard.Wolff@springsource.com

Why this talk?

- I do a lot of reviews
- There are some common problems you see over and over again

- So: Here are 10
  - …not necessarily the most common
  - ...but certainly with severe effects

#1

```java
public class Service {

  private CustomerDao customerDao;
  private PlatformTransactionManager transactionManager;

  public void performSomeService() {
    TransactionStatus transactionStatus = transactionManager
     .getTransaction(new DefaultTransactionDefinition());
    customerDao.doSomething();
    customerDao.doSomethingElse();
    transactionManager.commit(transactionStatus);
  }

}
```

#1 Weak Transaction Handling

```
public class Service {

  private CustomerDao customerDao;
  private PlatformTransactionManager transactionManager;

  public void performSomeService() {
    TransactionStatus transactionStatus = transactionManager
     .getTransaction(new DefaultTransactionDefinition());
    customerDao.doSomething();
    customerDao.doSomethingElse();
    transactionManager.commit(transactionStatus);
  }

}
```

- What happens to the transaction if the DAO throws an exception?
- We might never learn...
- ...or learn the hard way

Weak Transaction Handling: Impact

- Hard to detect, has effects only if exception is thrown
- …but then it can lead to wired behavior and data loss etc.
- That is why you are using transactions in the first place

Solution

- Declarative transactions

```
public class Service {

  private CustomerDao customerDao;

  @Transactional
  public void performSomeService() {
    customerDao.doSomething();
    customerDao.doSomethingElse();
  }

}
```

- Exception is caught, transaction is rolled back (if it is a RuntimeException)
- Exception handling can be customized

A different solution…

```
public void performSomeService() {
  TransactionTemplate template = new TransactionTemplate(
   transactionManager);
  template.execute(new TransactionCallback() {

    public Object doInTransaction(TransactionStatus status) {
      customerDao.doSomething();
      customerDao.doSomethingElse();
      return null;
    }

   });
}
```

- Allows for multiple transactions in one method
- More code – more control
- Rather seldom really needed

#2 Exception Design

- Get all the details from a system exception!
- Each layer must only use its own exceptions!
- Exceptions have to be checked – then they must be handled and the code is more secure.

- Sounds reasonably, doesn't it?

```
public class OrderDao {
  public void createOrder(Order order) throws SQLException {
    try {
     jdbcTemplate.update("INSERT INTO ORDER ...");
    } catch (DataAccessException ex) {
      throw (SQLException) ex.getCause();
    }
  }
}
```

Get all the details!

Use checked exceptions!

```
public class SomeService {
  public void performService()
    throws ServiceException {
    try {
      orderDao.createOrder(new Order());
    } catch (SQLException e) {
      throw new ServiceException(e);
    }
  }
}
```

Service must only throw ServiceException!

```
public class SomeController {
  public void handleWebRequest() {
    try {
      someService.performService();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

What am I supposed to do now?

No real logging

And I don’t care about the specific ServiceException

Impact

- Lots of useless exception handling code
- Lots of exception types without specific handling of that type
- In the end all you get is a log entry and lots of code

- And what should the developer do?
  - All he knows "Something went wrong"
  - Does not really care and can not really handle it

Why is this commonplace?

- Very few languages have checked exceptions (Java - CLU and Modula-3 had similar concepts)
- Checked exception force developers to handle an exception – very rigid
- How common is it that you can really handle an exception?
- Checked exceptions are perceived to be more secure
- Checked exceptions are overused – also in Java APIs

- In many cases there are even no exception concepts in projects

Solution

- Use more unchecked exceptions aka RuntimeExceptions
- Remember: A lot of languages offer only unchecked exceptions

- Avoid wrap-and-rethrow – it does not add value
- Don't write too many exception classes – they often don't add value
- A specific exception classes is only useful if that exception should be handled differently

Solution

```
public class OrderDao {
  public void createOrder(Order order) {
    jdbcTemplate.update("INSERT INTO ORDER ...");
  }
}
```

Where is the exception handling?

```
public class SomeService {
  public void performService() {
    orderDao.createOrder(new Order());
  }
}
```

```
public class SomeController {
  public void handleWebRequest() {
    someService.performService();
  }
}
```

AOP in one Slide

```
@Aspect
public class AnAspect {

  // do something before the method hello
  // is executed
  @Before("execution(void hello())")
  public void doSomething() {
  }

 // in a specific class
  // that ends in Service in any package or subpackage
  @Before("execution(*  com.springsource.MyService.hello())")
  public void doSomethingElse2() {
  }

  // do something before any method in a class
  // that ends in Service in any package or subpackage
  @Before("execution(*  *..*Service.*(..))")
  public void doSomethingElse2() {
  }
```

Aspect for Logging

```
@Aspect
public class ExceptionLogging {

  @AfterThrowing(value="execution(* *..Service*.*(..))",
   throwing="ex")
  public void logRuntimeException(RuntimeException ex) {
    System.out.println(ex);
  }

}
```

- Logs every exception – 100% guaranteed!

Handle only cases you really want to handle

```java
public class SomeService {
  public void performService() {
    try {
      orderDao.createOrder(new Order());
    } catch (OptimisticLockingFailureException ex) {
      orderDao.createOrder(new Order());
    }
  }
}
```

- Everything else will be handled somewhere else
- Can handle specific error conditions using catch with specific types

Generic Exception Handling

```
public class MyHandlerExceptionResolver
 implements HandlerExceptionResolver {

  public ModelAndView resolveException(
   HttpServletRequest request,
   HttpServletResponse response, Object handler, Exception ex) {
    return new ModelAndView("exceptionView", "exception", ex);
  }

}
```

- In the web layer
- Handle all the (Runtime)Exceptions not handled elsewhere

#3 Exception Handling

```
public void someMethod() {
  try {

  } catch (Exception ex) {
    ex.printStackTrace();
  }
  try {

  } catch (Exception ex) {
    // should never happen
  }
}
```

Exception is not logged
just written to stdout
operations might not notice

Exception is swallowed

Impact

- Related to #2: If you have excessive checked exceptions this will occur more often
- …as developers are forced to handle exceptions they can't really handle
- In the end you just get a message on the console and the application continues.
- All kinds of wired behavior
- i.e. exception is swallowed
- You will have a hard time finding problems in the code
- Potentially a huge problem – so worth its own explanation

Solution

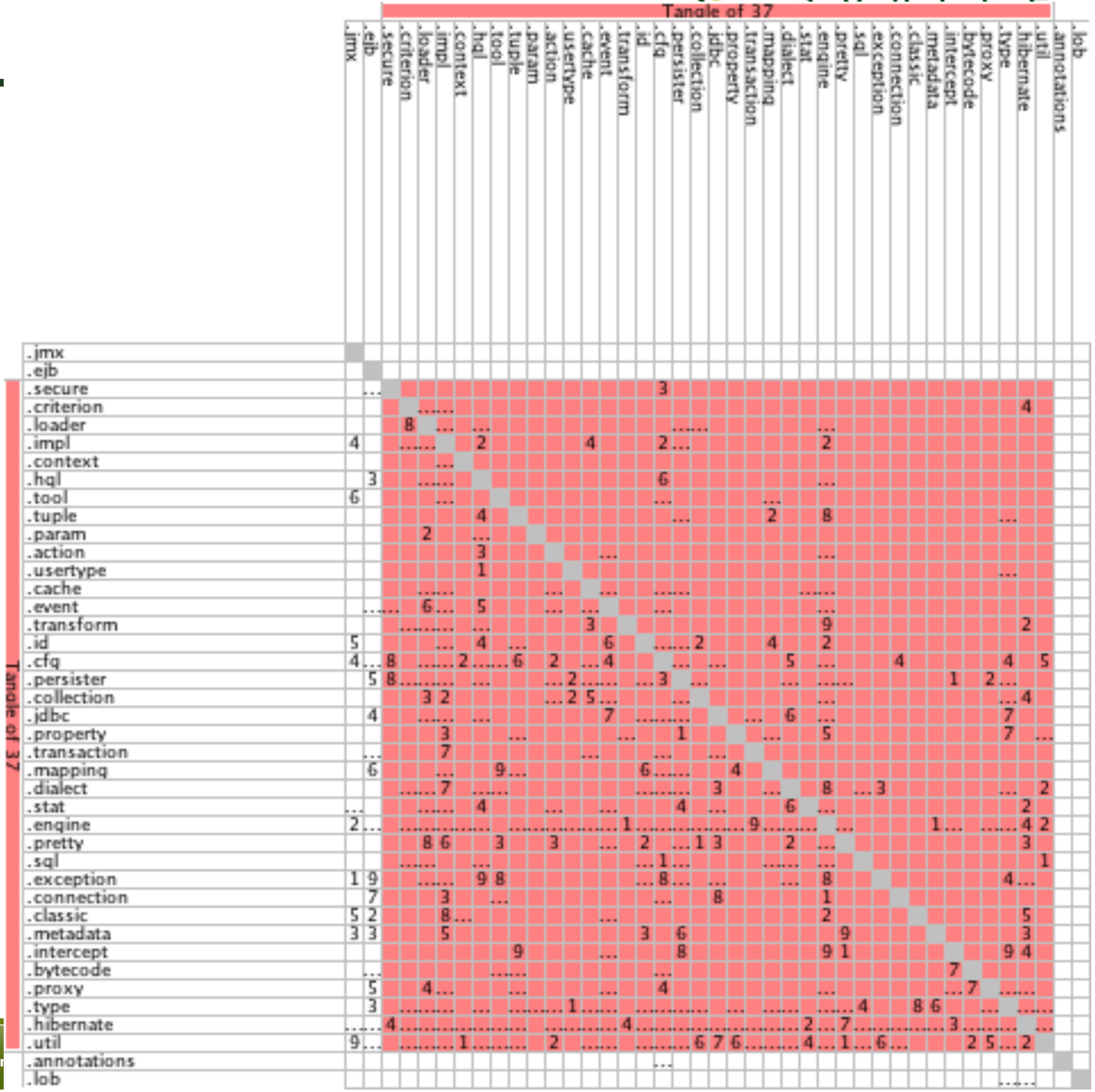
- At least log exceptions
- Rethink: Is it really OK to continue in this situation? If not - don't handle the exception. Might be better to let a generic handler handle it.
- Introduce generic handling at least for RuntimeException (AOP, web front end, etc)
- Enforce the logging using Findbugs, PMD etc.
- And: Improve the exception design (#2)

```java
public void someMethod() {
  try {

  } catch (Exception ex) {
    log.error(ex);
  }
}
```

#4

- Table of packages and the relations between them
- Everything in red is part of a cycle
- This is actual code from an Open Source project

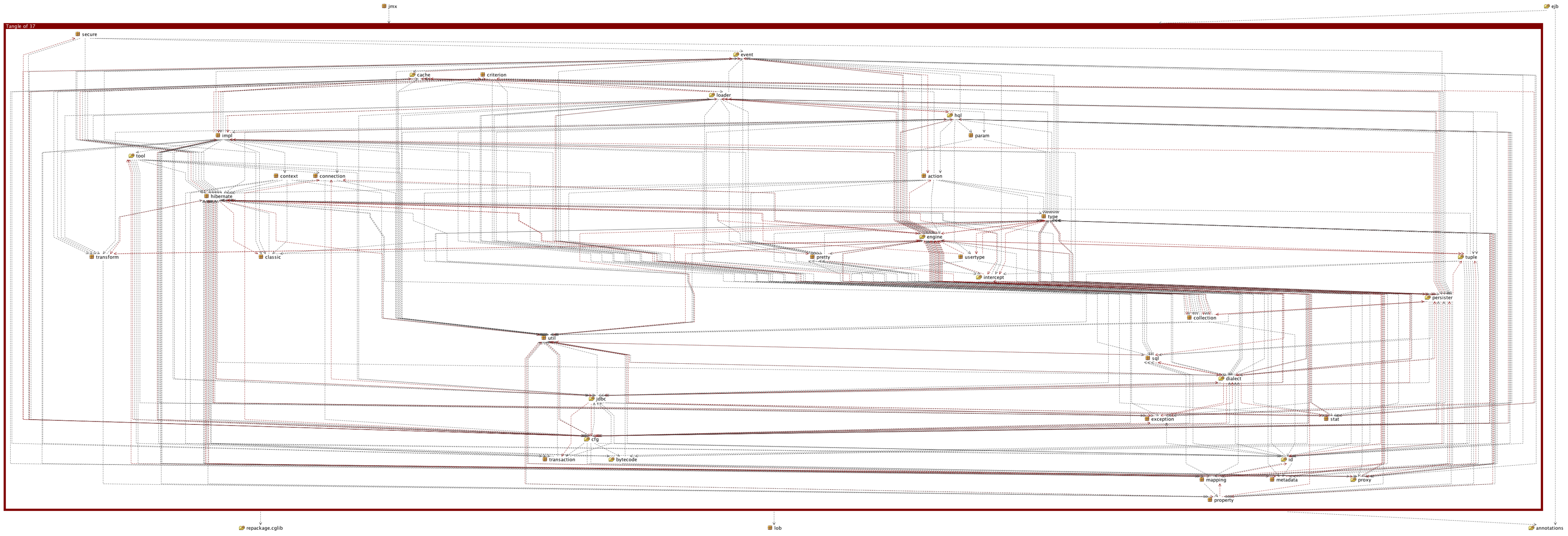


Tangle of 37

	.jmx	.ejb	.secure	.criterion	.loader	.impl	.context	.hql	.tool	.tuple	.param	.action	.usertype	.cache	.event	.transform	.id	.cfg	.persister	.collection	.jdbc	.property	.transaction	.mapping	.dialect	.stat	.engine	.pretty	.sql	.exception	.connection	.classic	.metadata	.intercept	.bytecode	.proxy	.type	.hibernate	.util	.annotations	.lob
.jmx																																									
.ejb																																									
.secure		...																3																							
.criterion					...	...																																4			
.loader				8		...		...											...	...							...														
.impl	4			...	...			2						4				2	...								2														
.context						...																																			
.hql		3			...	...												6									...														
.tool	6					...												...						...																	
.tuple								4											...					2			8										...				
.param					2			...																																	
.action								3							...												...														
.usertype								1																													...				
.cache					...	...						...			...			...	...							...	...														
.event		...	...		6	...		5				...		...				...									...														
.transform				...	...	...		...						3													9											2			
.id	5					...		4		...					6			...	...	2				4			2														
.cfg	4	...	8		...	...	2	...	...	6		2		...	4				...		...				5		...				4						4		5		
.persister		5	8	...	...	...		...				...	2	...	...		...	3		...					...		...	...						1		2	...				
.collection					3	2						...	2	5	...				...								...										...	4			
.jdbc		4			...	...		...							7		...	...	...				...		6		...										7				
.property						3				...						...			1					...			5										7		...		
.transaction		...				7								...				...			...																				
.mapping		6				...			9	...							6	...	...			4																			
.dialect				...	...	7		...	...								...	...	...		3			...			8		...	3							...		2		
.stat	...				...	...		4				...			...				4		...				6		...											2			
.engine	2	...		...	...	...	...	...		...	...	...	...	...	...	1	...	...	...	...	...	...	9	...	...	...		...					1	...		...	...	4	2		
.pretty					8	6			3			3			...		2		...	1	3				2		...											3			
.sql				...	...			...									...	1	...					...	...		...												1		
.exception	1	9			...	...		9	8								...	8	...		...				...		8										4	...			
.connection		7				3			...									...			8						1														
.classic	5	2				8	...								...												2											5			
.metadata	3	3				5											3		6									9										3			
.intercept										9					...				8								9	1									9	4			
.bytecode		...							...	...								...																7							
.proxy		5			4	...				...					...			4									...							...	7		...	...			
.type		3		...	...	...		...		...	...	...	1	...	...		...	...	...	...		...		...	...		...	...	4			8	6			...		...	...		
.hibernate	...	...	4	...	...	...	...	...	...	...		...	...	...	...	4	...	...	...	...	...	...	...	...	...	2	...	7	...	...	...	...	...	3	...	...	...		...		
.util	9	...		...	...	...	1	...	...	...		2		...	...		...	...	...	6	7	6	...	...	...	4	...	1	...	6	...				2	5	...	2			
.annotations																		...																							
.lob																																					...	...			

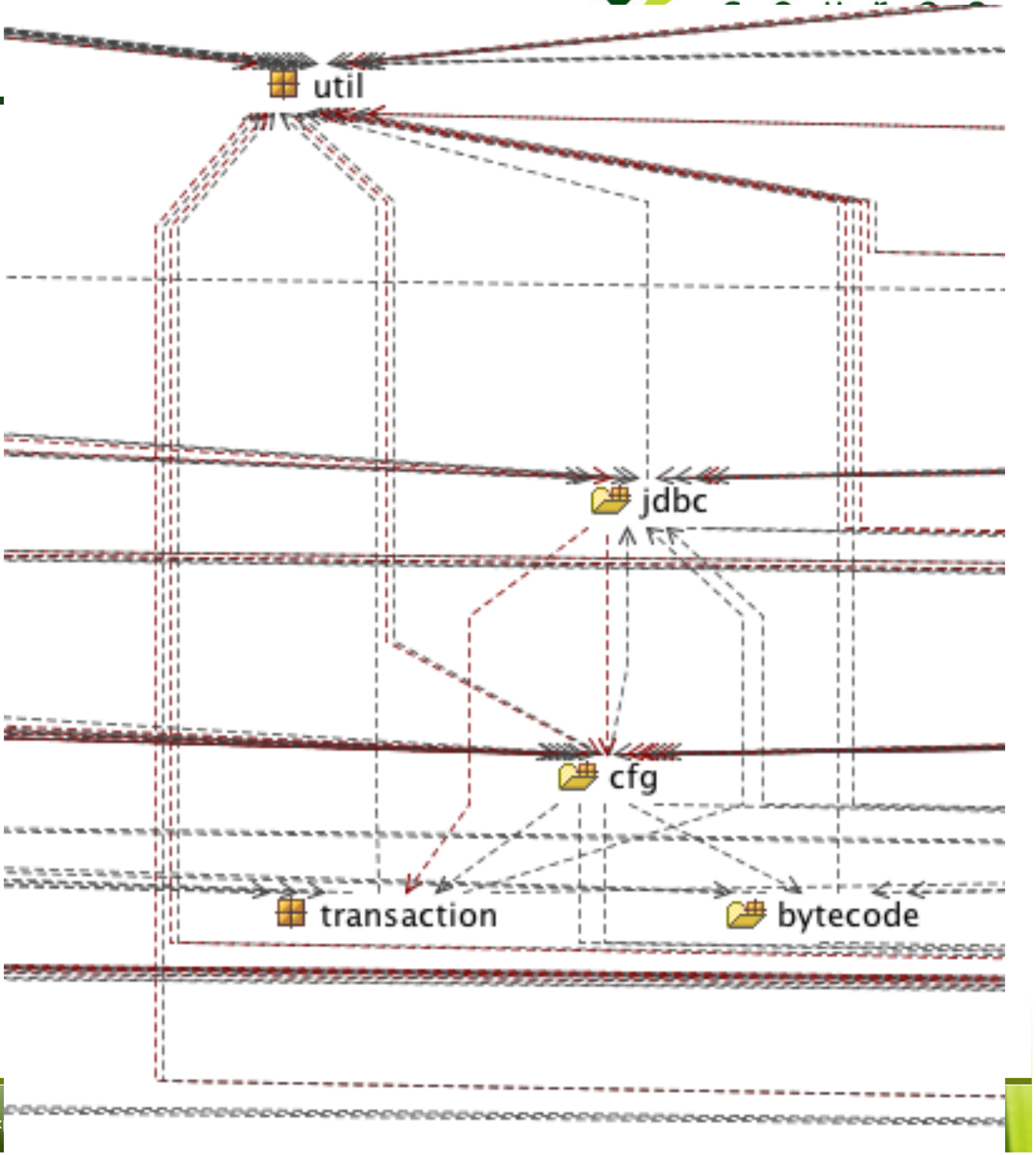
Dependency Graph

- Overview



Dependency Graph

- Just a small part
- Red line show circular references



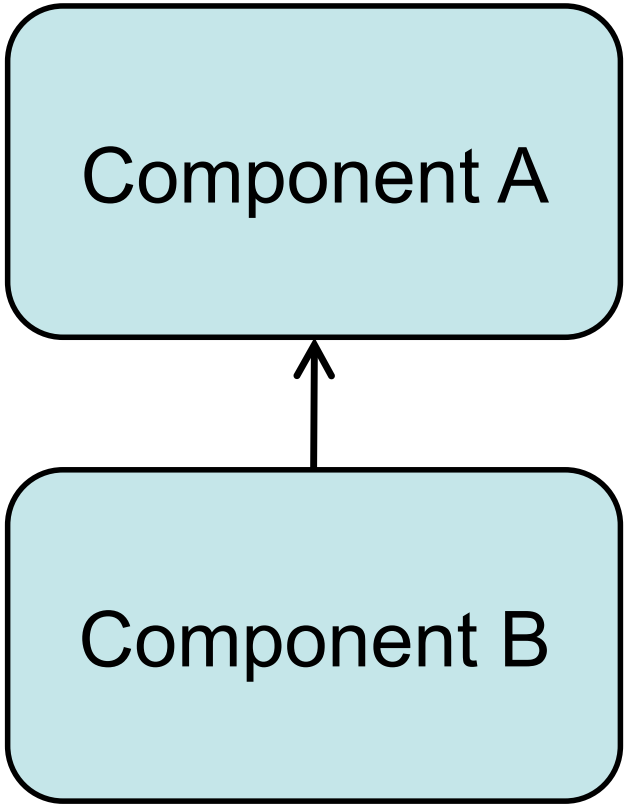
What is Architecture?

- Architecture is the decomposition of systems in parts

- No large or complex parts
- No cyclic dependencies

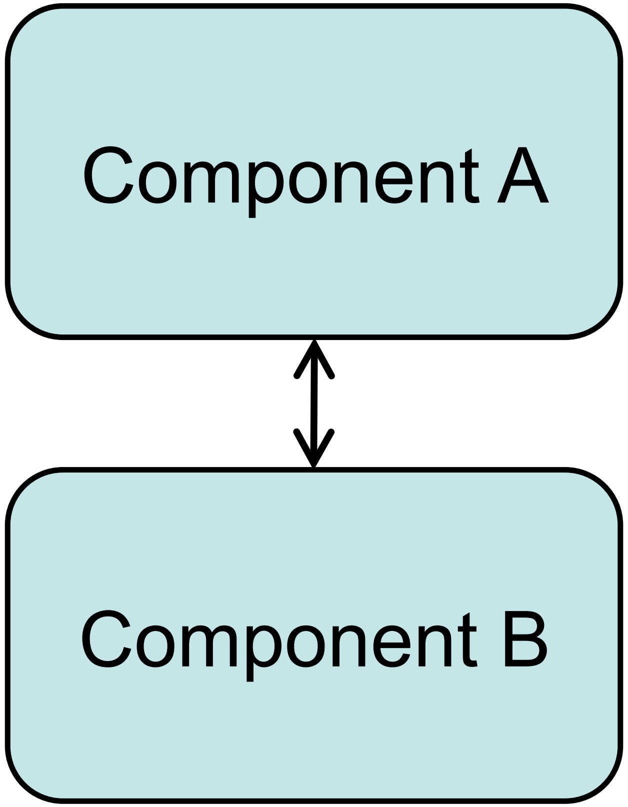
Normal Dependencies

- B dependes on A, i.e. it uses classe, methods etc.
- Changes in A impact B
- Changes in B do not impact A

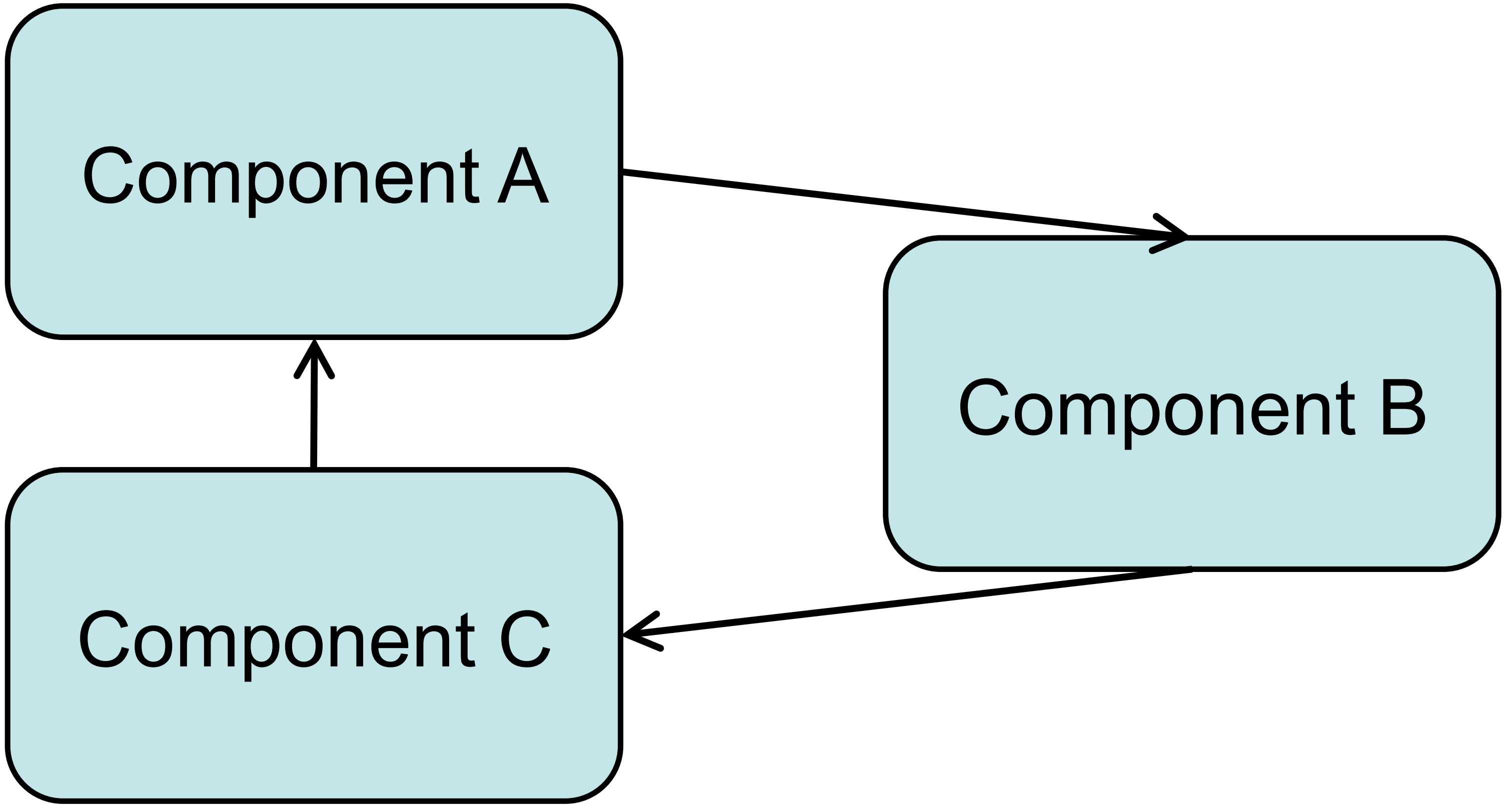


Cyclic Dependency

- B depends on A and A on B
- Changes in A impact B
- Changes in B impact A
- A and B can only be changed as one unit
- …even though they should be two separate units



Bigger cyclic dependencies



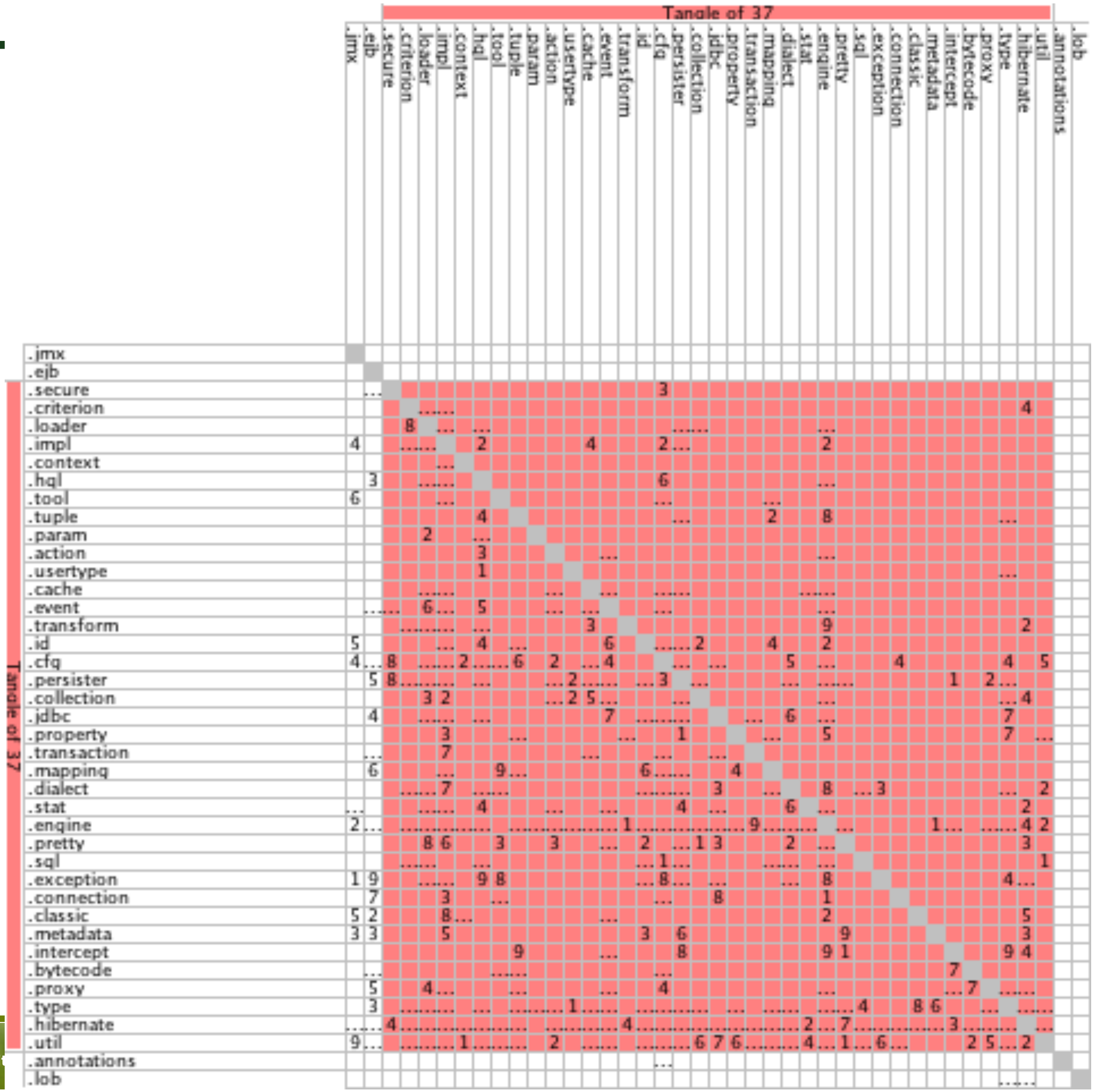
Component A

Component B

Component C

#4: Architecture Mess

- This is effectively just one big unstructured pile of mud
- Maintenance will be hard
- Concurrent development will be hard
- Changes will have unforeseeable results



Tangle of 37

Tangle of 37

	.jmx	.ejb	.secure	.criterion	.loader	.impl	.context	.hql	.tool	.tuple	.param	.action	.usertype	.cache	.event	.transform	.id	.cfg	.persister	.collection	.jdbc	.property	.transaction	.mapping	.dialect	.stat	.engine	.pretty	.sql	.exception	.connection	.classic	.metadata	.intercept	.bytecode	.proxy	.type	.hibernate	.util	.annotations	.lob
.jmx																																									
.ejb																																									
.secure		...																3																							
.criterion					...	...																																4			
.loader				8		...		...											...	...							...														
.impl	4			...	...			2						4				2	...								2														
.context						...																																			
.hql		3			...	...												6									...														
.tool	6					...												...						...																	
.tuple								4											...					2			8										...				
.param					2			...																																	
.action								3							...												...														
.usertype								1																													...				
.cache					...	...						...			...			...	...							...	...														
.event		...	...		6	...		5				...		...				...									...														
.transform				...	...	...		...						3													9											2			
.id	5					...		4		...					6			...	...	2				4			2														
.cfg	4	...	8		...	...	2	...	...	6		2		...	4				...		...				5		...				4						4		5		
.persister		5	8	...	...	...		...				...	2	...	...		...	3		...					...		...	...						1		2	...				
.collection					3	2						...	2	5	...				...								...										...	4			
.jdbc		4			...	...		...							7		...	...	...				...		6		...										7				
.property						3				...						...			1					...			5										7		...		
.transaction		...				7								...				...			...																				
.mapping		6				...			9	...							6	...	...			4																			
.dialect				...	...	7		...	...								...	...	...		3			...			8		...	3							...		2		
.stat	...				...	...		4				...			...				4		...				6		...											2			
.engine	2	...		...	...	...	...	...		...	...	...	...	...	...	1	...	...	...	...	...	...	9	...	...	...		...					1	...		...	...	4	2		
.pretty					8	6			3			3			...		2		...	1	3				2		...											3			
.sql				...	...			...									...	1	...					...	...		...												1		
.exception	1	9			...	...		9	8								...	8	...		...				...		8										4	...			
.connection		7				3			...									...			8						1														
.classic	5	2				8	...								...												2											5			
.metadata	3	3				5											3		6									9										3			
.intercept										9					...				8								9	1									9	4			
.bytecode		...							...	...								...																7							
.proxy		5		4		...				...					...			4									...							...	7		...	...			
.type		3		...	...	...		...		...	...	...	1	...	...		...	...	...	...		...		...	...		...	...	4			8	6			...		...	...		
.hibernate	...	...	4	...	...	...	...	...	...	...		...	...	...	...	4	...	...	...	...	...	...	...	...	...	2	...	7	...	...	...	...	...	3	...	...	...		...		
.util	9	...		...	...	...	1	...	...	...		2		...	...		...	...	...	6	7	6	...	...	...	4	...	1	...	6	...				2	5	...	2			
.annotations																		...																							
.lob																																					...	...			

Solution

- Very hard if you have this state
- Therefore: Manage dependencies from the start
- Otherwise you are looking at a major restructuring of your application
- …which might not be worth it
- Effort for restructuring pays off by lower effort for maintenance
- …might take a long time to amortize

- Throwing away + redevelopment means that you have to migrate to a new solution -> complex and risky

#5

```
public class ServiceAdaptor {
  public void performService(OrderDTO orderDTO)  {
    logger.trace("Entering performService");
    try {
      if (orderDTO == null) {
        throw new NullPointerException("order must not be null");
      }
      if (youAreNotAllowedToDoThis()) {
        throw new IllegalStateException(
         "You are not allowed to call this!");
      }
      OrderEntity order = new OrderEntity();
      order.setCustomer(orderDTO.getCustomer()); // ...
      service.performService(order);
      commandLog.add(new Command("performService",
       service,order));
    } finally {
      logger.trace("Leaving performanceService");
    }
  }
}
```

#5: Adaptor Layer

- Adds to a service:
  - Security
  - Tracing
  - Check for null arguments
  - Log for all commands (auditing, replay…)
  - Conversion from DTO to internal representation
- Lots of boilerplate for each service
- Changes to tracing etc. hard: lots of methods to change

Solution: Tracing with AOP

- …or use Spring's predefined TraceInterceptor, DebugInterceptor etc.

```
@Aspect
public class TraceAspect {

  @Before("execution(* *..*Service.*(..))")
  public void traceBegin(JoinPoint joinPoint) {
    System.out.println("entering method "
     + joinPoint.getSignature().getName());
   }

   @After("execution(* *..*Service.*(..))")
   public void traceEnd(JoinPoint joinPoint) {
     System.out.println("leaving method "
      + joinPoint.getSignature().getName());
   }
}
```

Solution:
Null Checks with AOP

```
@Aspect
public class NullChecker {

  @Before("execution(* *..*Service.*(..))")
  public void checkForNull(JoinPoint joinPoint) {
    for (Object arg : joinPoint.getArgs()) {
      if (arg==null) {
        throw new NullPointerException("Argument was null!");
      }
    }
  }

}
```

- Security can be handled with Spring Security or AOP
- Command log also possible

What is left…

```
public class ServiceAdaptor {

  public void performService(OrderDTO orderDTO) {
    OrderEntity order = new OrderEntity();
    order.setCustomer(orderDTO.getCustomer()); // ...
    service.performService(order);
  }
}
```

- You should probably switch to Dozer
- http://dozer.sf.net
- Can externalize mapping rules
- i.e. the layer can be more or less eliminated
- Everything (mapping, security, tracing…) is now implemented in one place (DRY)
- Often services just delegate to DAOs – same issue

#6: No DAO

```
public class SomeService {

  @PersistenceContext
  private EntityManager entityManager;

  public void performSomeService() {
    List<Order> list = entityManager.
     createQuery("select o from Order").getResultList();
    for (Order o : list) {
      // ...
      if (o.shouldBeProcessed()) {
        o.process();
      }
    }
  }
}
```

- We don't need to abstract away from JPA – it's a standard, right?

#6: Even worse

```
public class SomeServiceJdbc {

private OrderDao someDoa;

  public void performSomeService() throws SQLException {
    ResultSet rs = someDoa.getOrders();
    while (rs.next()) {
      //...
    }
  }

}
```

- Service depends on JDBC
- …and throws SQLException
- Persistence visible in the service layer and beyond

Impact

- Code is impossible to test without a database
- …so no real unit tests possible

- Service depends on persistence – cannot be ported

- How do you add data dependent security?

- No structure

Solution

- Use a DAO (Data Access Object)
  - Separate persistence layer
  - Technical motivation

- …or a Repository
  - Interface to existing objects
  - Non technical motivation: Domain Driven Design, Eric Evans

- Basically the same thing

Solution

```
public class SomeServiceDAO {

 public void performSomeService() {
  List<Order> list = orderDao.getAllOrders();
  for (Order o : list) {
   // ...
   if (o.shouldBeProcessed()) {
    o.process();
   }
  }
 }
}
```

- Clear separation
- Tests easy

Solution: Test

```
public class ServiceTest {
  @Test
  public void testService() {
    SomeService someService = new SomeService();
    someService.setOrderDao(new OrderDao() {

      public List<Order> getAllOrders() {
        List<Order> result = new ArrayList<Order>();
        return result;
      }
    });
    someService.performSomeService();
    Assert.assertEquals~~(expected, result);~~
  }

}
```

#7

- No Tests

#7 Or bad tests

- No asserts
- System.out: results are checked manually
- Tests commented out: They did not run any more and were not fixed
- No mocks, so no real Unit Tests
- No negative cases

```
public class MyUnitTest {
  private Service service = new Service();

  @Test
  public void testService() {
    Order order = new Order();
    service.performService(order);
    System.out.print(order.isProcessed());
  }

  // @Test
  // public void testOrderCreated() {
  // Order order = new Order();
  // service.createOrder(order);
  // }

}
```

Impact

- Code is not properly tested
- Probably low quality – testable code is usually better designed
- Code is hard to change: How can you know the change broke nothing?
- Design might be bad: Testable usually mean better quality

Solution

- Write proper Unit Tests!

```
public class MyProperUnitTest {
  private Service service = new Service();

  @Test
  public void testService() {
    Order order = new Order();
    service.performService(order);
    Assert.assertTrue(order.isProcessed());
  }

  @Test(expected=IllegalArgumentException.class)
  public void testServiceException() {
    Order order = new BuggyOrder();
    service.performService(order);
  }

}
```

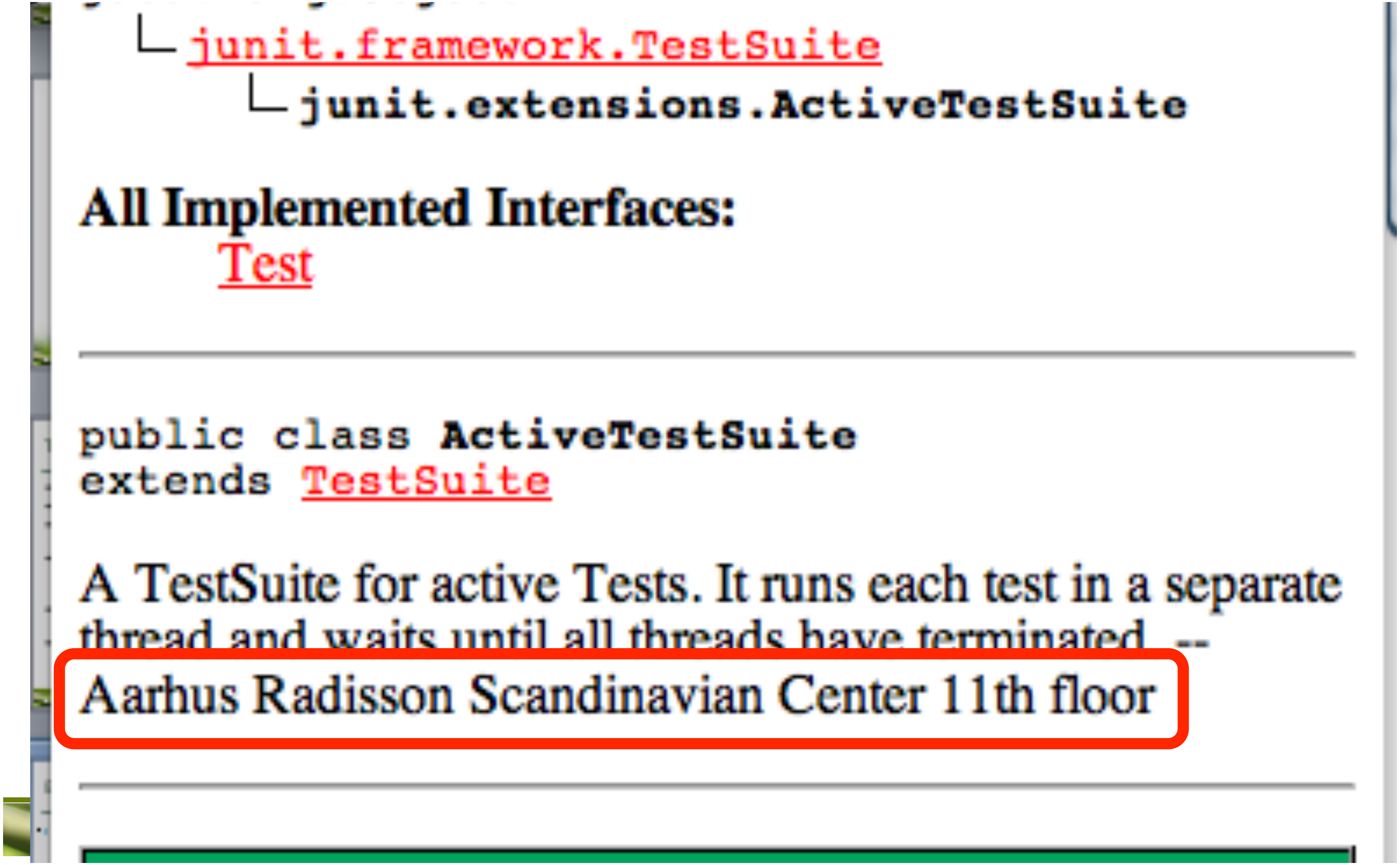
Wow, that was easy!

The real problem…

- The idea of Unit tests is over 10 years old
- Not too many programmer actually do real unit tests
- Even though it should greatly increased trust and confidence in your code
- …and make you much more relaxed and therefore improve quality of life…

- Original paper: Gamma, Beck: "Test Infected – Programmers Love Writing Tests"
- Yeah, right.

BTW



```
└ junit.framework.TestSuite
    └ junit.extensions.ActiveTestSuite
```

All Implemented Interfaces:

Test

```
public class ActiveTestSuite
extends TestSuite
```

A TestSuite for active Tests. It runs each test in a separate thread and waits until all threads have terminated. --

Aarhus Radisson Scandinavian Center 11th floor

Solution

- Educate
  - Show how to write Unit Test
  - Show how to build Mocks
  - Show aggressive Testing
  - Show Test First / Test Driven Development
- Coach / Review
- Integrate in automatic build
- Later on: Add integration testing, functional testing, FIT, Fitnesse etc.
- …or even start with these

What does not really work

- Measuring code coverage
  - Can be sabotaged

```
public class MyProperUnitTest {
  private Service service = new Service();

  @Test
  public void testService() {
    Order order = new Order();
    service.performService(order);
  }
}
```

- Let developers just write tests without education
  - How should they know how to test properly?
  - Test driven development is not obvious

#8: Creating SQL statements

```
public class OrderDAO {

  private SimpleJdbcTemplate simpleJdbcTemplate;

  public List<Order> findOrderByCustomer(String customer) {
    return simpleJdbcTemplate.query(
     "SELECT * FROM T_ORDER WHERE name='"
     + customer + "'", new OrderRowMapper());
  }

}
```

Impact

- Performance is bad:
  - Statement is parsed every time
  - Execution plan is re created etc.

Impact

- Even worse: SQL injection
- Pass in a' or 't'='t'
- Better yet: a'; DROP TABLE T_ORDER; SELECT * FROM ANOTHER_TABLE

```
public class OrderDAO {

  private SimpleJdbcTemplate simpleJdbcTemplate;

  public List<Order> findOrderByCustomer(String customer) {
    return simpleJdbcTemplate.query(
     "SELECT * FROM T_ORDER WHERE name='"
     + customer + "'", new OrderRowMapper());
  }

}
```

Solution

```
public class OrderDAO {

  private SimpleJdbcTemplate simpleJdbcTemplate;

  public List<Order> findOrderByCustomer(String customer) {
    return simpleJdbcTemplate.query(
     "SELECT * FROM T_ORDER WHERE name=?",
     new OrderRowMapper(), customer);
  }

}
```

- … and white list the allowed characters in name

#9

- "What about Performance?"
- "Well, we figured the response time should be 2s."
- "How many request do you expect?"
- "…"
- "What kind of requests do you expect?"
- "..."

#9

- The software is happily in the final functional test
- Then the performance test start
- Performance is too bad to be accepted
- You can hardly do anything:
  - Changes might introduce functional errors
  - Too late for bigger changes anyway
- The results might be wrong if the performance test is on different hardware than production.
- You can't test on production hardware: Too expensive.

Impact

- You have to get bigger hardware
  - Prerequisite: The software is scalable
- Worse: You can't go into production

Solution

- Get information about the number of requests, expected types of requests, acceptable response times
- Pro active performance management:
  - Estimate the performance before implementation
  - …by estimating the slow operations (access to other systems, to the database etc)
  - Measure performance of these operation in production
- Practice performance measurements and optimizations before performance test

#10

```
public class SomeService {

private Map cache = new HashMap();
private Customer customer;

 public Order performService(int i) {
  if (cache.containsKey(i)) {
   return cache.get(i);
  }
  Order result;
  customer = null;
  cache.put(i, result);
  return result;
 }

}
```

#10 Multiple threads, memory leaks

```
public class SomeService {

  private Map<Integer,Order> cache =
   new HashMap<Integer, Order>();
  private Customer customer;

  public Order performService(int i) {
    if (cache.containsKey(i)) {
      return (Ordercache.get(i);
    }
    Order result;
    customer = null;
    ...
    cache.put(i, result);
    return result;
  }

}
```

The cache is filled – is it ever emptied?

HashMap is not threadsafe

customer is an instance variable – multi threading will be a problem

Impact

- System working in small tests
- In particular Unit tests work

- But production fails
- …probably hard to analyze / fix
- Almost only by code reviews
- …or extensive debugging using thread dumps

Solution

- Use WeakHashMap to avoid memory leaks
- Synchronize
- Prefer local variables
- Usually services can store most things in local variables

```java
public class SomeServiceSolution {

  private Map<Integer, Order> cache =
   new WeakHashMap<Integer, Order>();

  public Order performService(int i) {
    synchronized (cache) {
      if (cache.containsKey(i)) {
        return cache.get(i);
      }
    }
    Order result = null;
    Customer customer = null;
    synchronized (cache) {
      cache.put(i, result);
    }
    return result;
  }
}
```

Solution

- Also consider ConcurrentHashMap
- or http://sourceforge.net/projects/high-scale-lib

Sum Up

- #1 Weak Transaction Handling
- #2 Exception Design
- #3 Exception Handling
- #4 Architecture Mess
- #5 Adaptor Layer
- #6 No DAO
- #7 No or bad tests
- #8 Creating SQL queries using String concatenation
- #9 No performance management
- #10 Multiple threads / memory leaks