A stylized, blue-tinted image of the Golden Gate Bridge, showing the suspension towers and cables against a light background.

Where Did My Architecture Go?

*Preserving and recovering the design
of software in its implementation*

*QCON London
March 2011*

Eoin Woods
www.eoinwoods.info

Content

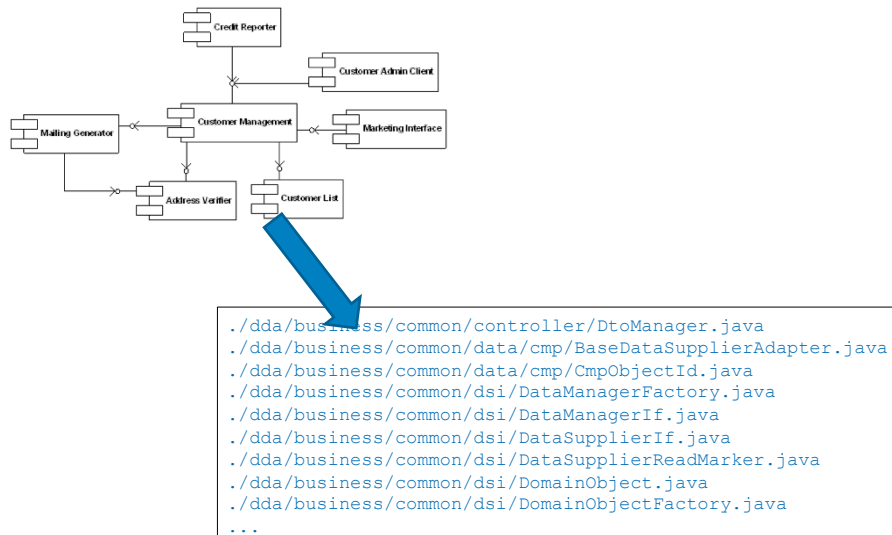
- Losing Design in the Code
- Key Design Constructs
- My Example Application
- Finding and Keeping Design in Code
 - Conventions
 - Dependency analysis
 - Module systems
 - Augmenting the code & checking rules
 - Language extensions
- Summary

Losing Design in the Code

Losing the design in the code

- Good designers talk about many structures that aren't part of mainstream programming languages
 - layers, adapters, proxies, components, brokers, ...
- Yet the code is all that exists at runtime
 - "the code doesn't lie"
- But if we've only got code where does all that design information go?

Losing the design in the code



Problems with Code as Design

- Only language structures are present
 - language structures are fine grained
- Fine grained nature is very detailed
 - difficult to discern structure from the details
- Design structures don't exist
 - need to discern design structure from the code
 - a lot of mapping and assumption required
 - relies on well structured and named code

Strengths of Code as Design

- Interpretation by machine
 - searching
 - checking
 - querying
- Certainty with respect to runtime
 - “the code doesn’t lie”
- Integration of different abstraction levels
 - see where a line fits in the overall design

Using Code as our Design Notation

- Need to extend our notion of “code”
 - Extend languages to allow design elements
 - Sapir-Whorf – extend language, influence thinking
- We can create or use new languages
 - e.g. aspect based languages, ArchJava,
- Or extend the ones we have
 - comments, annotations, packages, aspects, ...
- Or use tools to visualise and infer design
 - dependency visualisers, rules checkers, ...

Key Design Constructs

What Do We Mean by Design Information?

- Many possible types of “design” information but two broad groups: *static* and *dynamic*
- *Static* information describes structures visible at *design time*
 - packages, classes, components, connectors, ...
- *Dynamic* information describes structures visible at *runtime*
 - invocations, data flow, sequencing, ...

Static Design Information

- Layers
- Modules (or “subsystems”)
- Components (and component types)
- Pattern implementations
- Dependencies
 - e.g. between components or on resources
 - particularly difficult across tiers/technologies
- Hierarchies and containment (e.g. plugins)

Dynamic Design Information

- Run-time invocations & control flow
 - i.e. **do** I call X, not **can** I call X (and when)
- Size and speed
 - how many times do I do something?
 - how many items are accessed / created / ...
- Design validation
 - does it really do what the design says it should

This session's focus is static information

An Example Application

An Example Application

- A very, very basic CRM system server + integration test
- Based on code supplied as a vendor example
- 6 functional modules + a generic framework
 - CRM Framework
 - Contacts
 - Customers
 - Distribution Partners
 - Enterprise Service Integration (ESI – links to other systems)
 - Requests
 - Users
- Code is organised first by module, then by layer
- Small, not trivial: 6.5 KLOC, 20K byte codes, 162 types

Simple CRM System Layers

Controller

Domain

Service

Data Service Interface

Persistence

Simple CRM Modules

Customers

Contacts

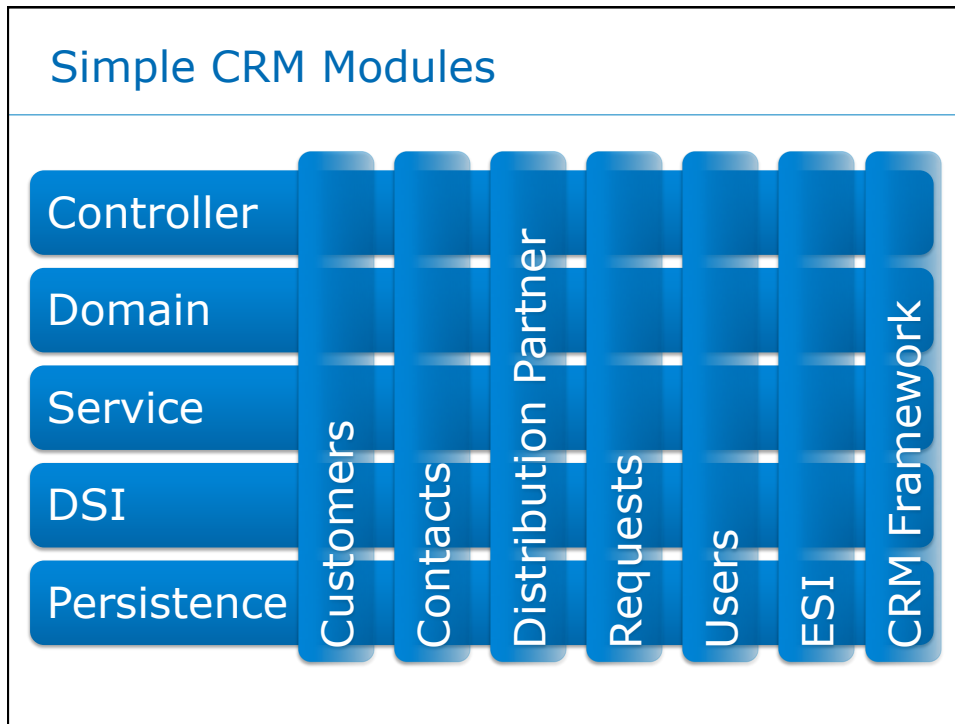
Distribution Partner

Requests

Users

ESI

CRM Framework



Finding & Keeping Design in Code

Techniques

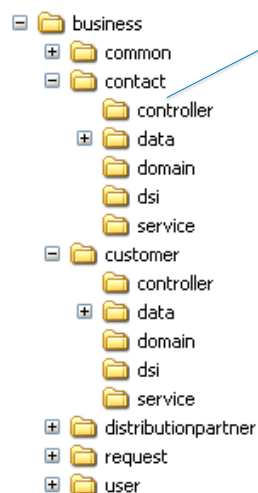
- Naming conventions with code & build structure
- Dependency analysis tools
- Module systems (Spring, Guice, OSGi, ...)
- Augmenting the code (annotations, rules)
 - checking design rules (Architecture Rules, Macker)
- Aspects
 - useful for design constructs and checking
- Special purpose languages (e.g. ArchJava)

Naming and Conventions

Naming and Conventions

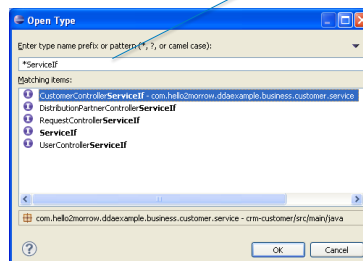
- Code structure
 - classes, packages, ...
- Build system
 - Maven's module based build
 - dependencies

Naming and Conventions



Functional areas & layers
shown by package naming

Identify code element
types by name matching



"Simple" stuff but often done badly or not at all

Dependency Analysis

Dependency Analysis

- A key element of design is modularisation
- A key part of modularisation is defining dependencies
- Poorly designed dependencies make software almost un-maintainable
 - well designed dependencies enable change with a minimum of risk and cost
- Dependencies are difficult to visualise, analyse and understand from source code
- A good place to start understanding design and managing design information is dependency analysis

Tools 1 – Structure & Analysers

- Understanding Java code using
 - Maven
 - jDepend
 - ClassCycle
 - Structure 101

Tools1 – Maven and Design

- Maven is a Java project automation system
 - build, tool integration, doc generation, ...
 - a key philosophy is enforcing a set of conventions
 - forces you to consider modularisation & dependencies
- Maven “modules” (code trees) have explicit versioned dependencies on each other
- The structure of a Maven project is almost always comprehensible if you’ve seen one before

Tools1 - Maven

```
+ pom.xml
+ module1
  - pom.xml
  - src/main/java/ ...
  - src/test/java/ ...
  - target/module1-1.0-SNAPSHOT.jar
+ module2
  - pom.xml
  - src/main/java/ ...
  - src/test/java/ ...
  - target/module2-1.0-SNAPSHOT.jar
```

pom.xml declares structure and dependencies at each level

each module has the same structure

1 module = 1 built target (JAR)

Tools1 - Maven

```
<project ...>
  ...
  <groupId>com.artechra.simplecrm</groupId>
  <artifactId>crm-request</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>CRM User Request Module</name>
  <dependencies>
    <dependency>
      <groupId>com.artechra.simplecrm</groupId>
      <artifactId>crm-framework</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    ...
  </dependencies>
</project>
```

An example pom.xml module definition file

Explicit dependencies in the build system help to preserve design information

Tools 1 - Maven Dependency Analysis

```
$> mvn dependency:tree -Dverbose=true
com.artechra.simplecrm:crm-itest:jar:1.0-SNAPSHOT
+- com.artechra.simplecrm:crm-contact:jar:1.0-SNAPSHOT:compile
| +- (com.artechra.simplecrm:crm-framework:jar:1.0-SNAPSHOT:compile - ...
| +- (com.artechra.simplecrm:crm-esi:jar:1.0-SNAPSHOT:compile - omitted for duplicate)
| \- (log4j:log4j:jar:1.2.9:compile - omitted for duplicate)
+- com.artechra.simplecrm:crm-customer:jar:1.0-SNAPSHOT:compile
| +- (com.artechra.simplecrm:crm-framework:jar:1.0-SNAPSHOT:compile - ...
| +- (com.artechra.simplecrm:crm-contact:jar:1.0-SNAPSHOT:compile - ...
| +- (com.artechra.simplecrm:crm-user:jar:1.0-SNAPSHOT:compile - omitted for duplicate)
| \- (log4j:log4j:jar:1.2.9:compile - omitted for duplicate)
+- com.artechra.simplecrm:crm-distributionpartner:jar:1.0-SNAPSHOT:compile
| +- (com.artechra.simplecrm:crm-framework:jar:1.0-SNAPSHOT:compile - ...
| +- (com.artechra.simplecrm:crm-request:jar:1.0-SNAPSHOT:compile - ...
[trimmed]
```

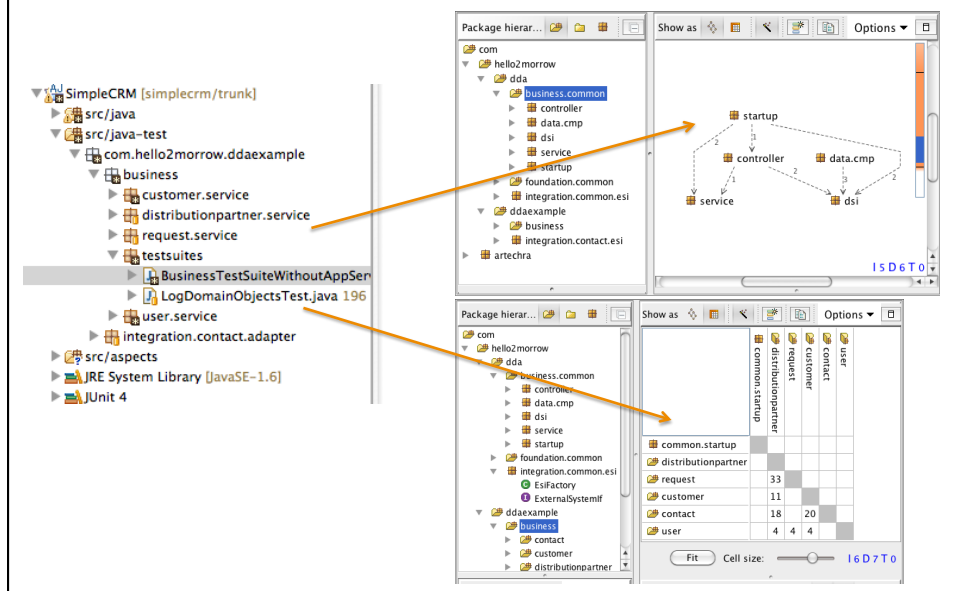
Generates a dependency tree for the project modules

The Maven "site report" web sites also have dependency reports

Tools 2 – Dependency Analysers

- **Static dependency checkers**
 - Structure 101, Lattix, CppDepend, Ndepend
 - jDepend, ClassCycle, Dependency Finder
- **Reveal real structures via dependency analysis**
 - often with checking for simple user defined rules
- **Capabilities and focus vary by tool**
 - one common limitation is use of package structure

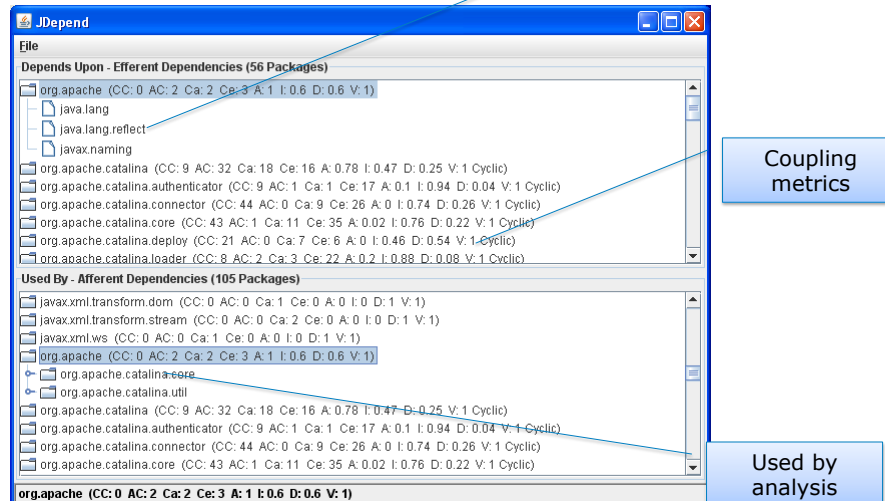
Tools 2 - Dependency Analysers



Tools 2 - JDepend

- **Probably the original Java dependency analyzer**
 - byte code analysis of coupling and dependencies
 - now extremely stable
 - limited to package level analysis
- **Limited GUI feature set**
 - afferent and efferent coupling report with metrics
- **Primarily a library**
 - callable from JUnit, other tools, FITness, Ant, ...
 - comprehensive XML or text report

Tools 2 - JDepend



Tools 2 - ClassCycle

- **Similar tool to JDepend**
 - extends it with class level dependency analysis
 - adds dependency checking language and engine
 - well documented algorithms
 - a couple of limitations removed
- **XML reporting of dependencies**
 - command line or Ant plugin
 - Eclipse plugin is available
- **Nice dependency checking language**

Tools 2 - ClassCycle Example Rules

```
{base-pkg} = com.artechra

[util] = ${base-pkg}.util.*
[non-util] = ${package}.* excluding [util]

check [util] independentOf [non-util]

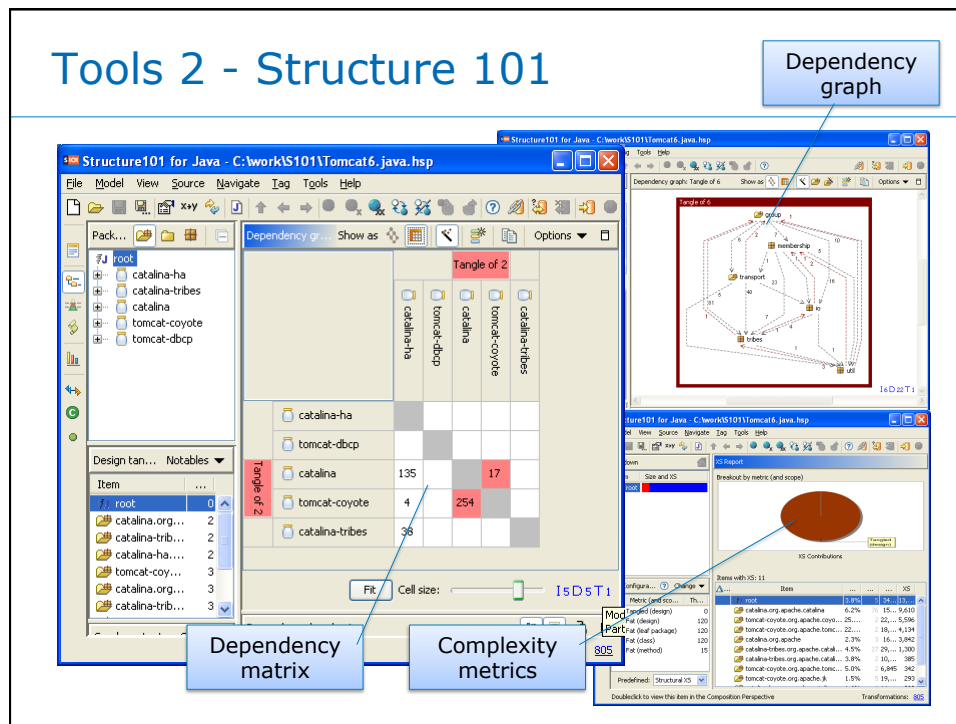
check absenceOfPackageCycles > 1 in ${package}.*

layer infra = [util] ${base-pkg}.type.*
layer persistence = ${base-pkg}.dao.*
layer domain-logic = ${package}.domain.*
check layeringOf basic persistence domain-logic
```

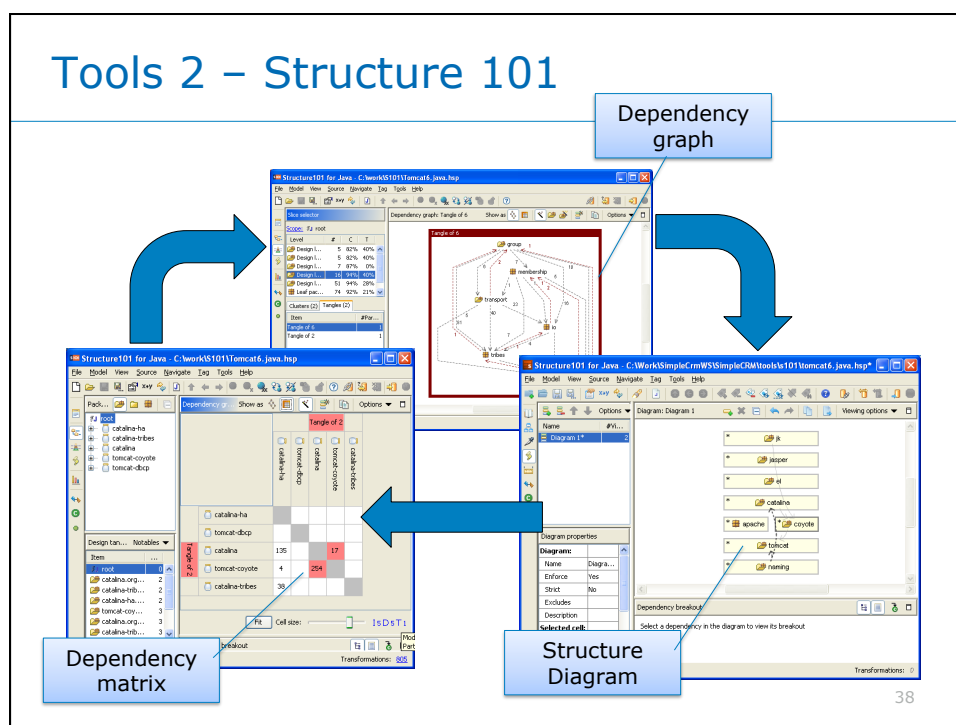
Tools 2 - Structure 101

- **Commercial dependency analyser**
 - Java, .NET, C/C++ and “generic” versions
- **Desktop tool and optional “headless” tools with webapp for history and build integration**
- **Dependencies, collaborations, metrics**
 - basic violation checking via “architecture” view
- **Rich UI for navigation and analysis**
 - separate IDE integration for IntelliJ and Eclipse

Tools 2 - Structure 101



Tools 2 - Structure 101



Module Systems

Module Systems & Structuring

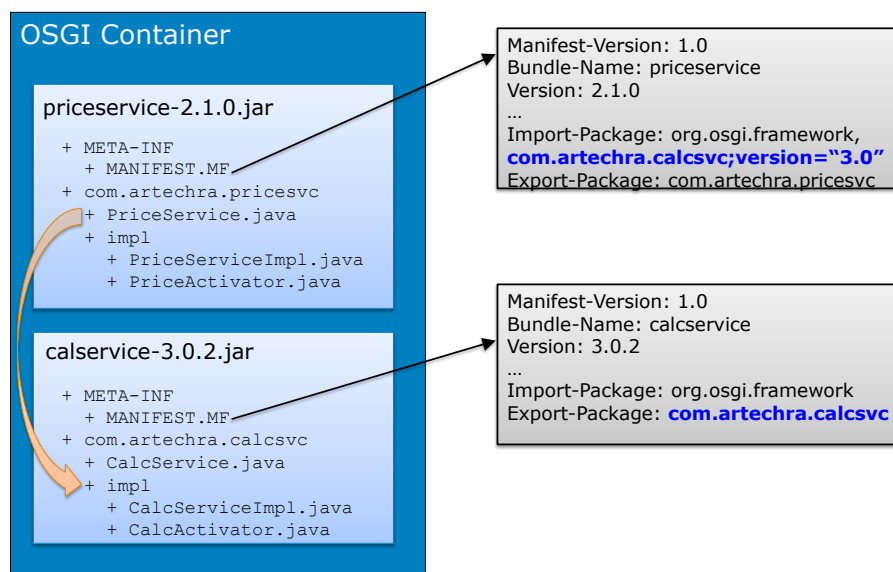
- **Dependency Injection (IoC) containers**
 - Java: Spring, Guice, ...
 - .NET: AutoFac, Spring.NET, ...
 - Containers instantiate components & “wire” together
 - Tendency to end up with very large configurations
 - Tendency to end up with very fine grained “components”
 - Do allow some degree of system structure to be visible
- **Full blown module systems like OSGi**
 - provides a module system for Java based on JAR files
 - a reasonably high commitment technology to adopt
 - can be complicated in a JEE environment
 - explicit declaration of module’s exports and imports

Java Modules with OSGi

- OSGi is an example of a full-blown module system
 - defines model of modules, their lifecycle and services
 - specifies a runtime container of standard services
 - allows (functional) structure to be clearly seen
 - makes inter module dependencies very clear
- Open standard developed by the OSGi Alliance
- Evolves existing Java technologies
 - JAR files used as the basis of components ("bundles")
 - Manifest files extended to provide meta-data
 - Bundle services are simply Java interfaces
 - Imports and exports specified by Java packages

41

Java Modules with OSGi



Augmenting the Code

Augmenting the Code

- Design information meta-data
 - annotations
 - external meta-data (e.g. declarative design rules)
- Rules based tools for validation
 - commercial: Structure 101, SonarJ, ...
 - open source: Macker, Architecture Rules, ...
 - aspect oriented languages: AspectJ, ...

Meta Data in the Code

- A number of meta-data systems exist
 - Java annotations & .NET attributes
 - Doxygen comments
- These can be used to “mark” design elements
 - @Layer(“Persistence”), @Facade
- Current limitation is the lack of tool support
 - except aspects, most tools ignore annotations etc
 - can write own utilities using reflection type APIs
 - feed outputs of proprietary analysis to generic tools

Java Metadata - Annotations

- Annotations allow meta data to be attached to code
 - fields, methods, parameters, classes, interfaces, packages
 - useful information for people, also machine processable
- Annotations are little classes
 - defined as special interfaces using keyword “@interface”
 - can contain own data, allowing storage of design information
 - result in objects attached to Java code elements
- Can define annotations for design level information
 - Component types: @Service, @DAO, @DomainObject, ...
 - Containers: @Component, @Layer
- Include in the code as you write it
 - for packages, remember they’re in package-info.java ! ⁴⁶

Java Metadata - Annotations

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PACKAGE)
public @interface Layer {
    enum LAYERS {PERSISTENCE, WEBUI, DOMAIN,
                INTEGRATION, SERVICE} ;

    LAYERS layer() ;
}
```

Package layer
annotation with a
"layer" attribute

```
@Layer(layer=Layer.LAYERS.DOMAIN)
package com.artechra.simplecrm.business ;
import com.artechra.Layer ;
```

package-info.java

Applies the annotation to
the package meta-data

```
Package p = Package.getPackage(pkgName) ;
if (p.isAnnotationPresent(Layer.class)) {
    Layer l = p.getAnnotation(Layer.class) ;
    if (l.layer() == Layer.LAYERS.SERVICE) {
        // ...
    }
}
```

Reflection allows you to
write utilities that see this
design information
*(you can also use aspects or
add annotation processors
to your build via javac)*

47

Checking Design Rules

Rules Based tools

- Earlier we saw dependency analysers
 - primarily for extracting dependencies
 - some with interactive analysis
 - most provide basic checking
- Another approach are the rules checking tools
 - provide a rules language and checking engine
- Examples for Java are
 - Architecture Rules and Macker (open source)
 - Semmlé ODASA and SonarJ (commercial)

Tools 3 - Macker

- Macker is a rules-based structural analyser
 - open source GPL project, analyses Java byte code
- Rules are described in an XML based language
 - wildcards, inclusion/exclusion, quantifiers, ...
- Macker is a command / Ant target / library to check the rules against a set of classes
 - output to XML or HTML reports or the console
- Real strength is flexibility and the rule language

Tools 3 – Macker Enforcing Rules

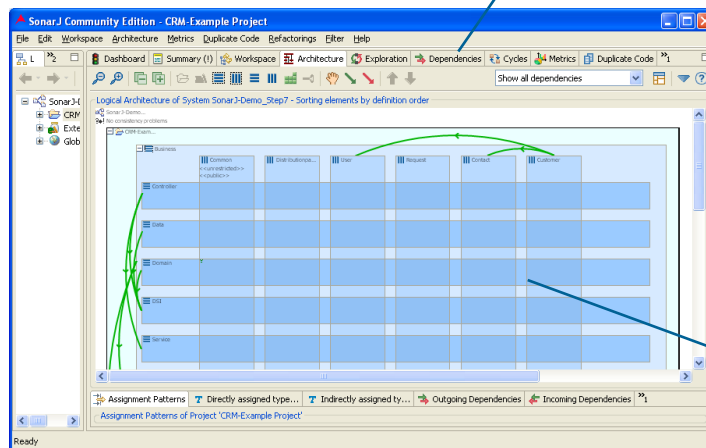
```
<ruleset>
  <var name="base-pkg" value="com.artechra.simplecrm" />
  <foreach var="module"
    class="{base-pkg}.module.*" />
    <pattern name="api" class="{module}.*" />
    <pattern name="inside" class="{module}.*" />
    <pattern name="outside">
      <exclude pattern="inside" />
    </pattern>
    <access-rule>
      <message>{from} must access {module} via its API</message>
      <deny>
        <from pattern="outside"/><to pattern="inside" />
      </deny>
      <allow><to pattern="api" /></allow>
    </access-rule>
  </foreach>
</ruleset>
```

Tools 3 - SonarJ

- Commercial code analysis tool
 - dependency analysis, metrics, refactoring
 - reporting against a quality model
 - processes Java source code and byte code
- GUI, command line, Ant and Maven options
 - database option for storing results
- Allows fine grained dependency rules
 - provision for layering *and* partitioning

Tools 3 - SonarJ

Metrics, dependency analysis,
code duplication checks, ...



Architecture
slices and layers

Extending Programming Languages

Extending the Language

- **Sapir-Whorf hypothesis: language affects thought**
 - we do see this with software developers
 - implementation language drives vocabulary
 - Design level language == design level thinking?
- **Aspects**
 - split code into design oriented “slices”
 - generate errors and warnings based on structure
- **Special purpose languages**
 - Arch Java

Aspect Orientation

- **AOP provides two possibilities**
 - change the way the code is structured
 - check the code using warning/error advice
- **AOP separates code into modules that are applied across the codebase**
 - “advice” is the code to apply
 - “point cuts” specify where to put it
 - special AspectJ advice just creates warnings when point cuts match

Aspect Orientation

- A common code pattern:

```
public Person getEmployee(String empId) {
    log.entering("HRService", "getEmployee", empId) ;
    if (!SecMgr.checkAccess(Ctx.getCaller(),
        new Resource(PERSON, empId), READ)) {
        throw new AuthorizationException(...) ;
    }
    Person ret = empRepo.getPersonByRole(empId, Role.EMPLOYEE);
    log.exiting("HRService", "getEmployee", ret) ;
    return ret ;
}
```

The one line
of business
logic !

It's difficult to see your design when so many concerns
are tangled together

Aspect Orientation

AOP allows us to untangle the concerns ...

```
public aspect LoggingAspect {
    Logger _log = Logger.getLogger("MyAppLogger") ;

    pointcut loggedCall() : execution(@LogPoint * *.*(..) && !within(LoggingAspect));
    before() : loggedCall() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        Object arg = (thisJoinPoint.getArgs().length > 0 ?
            thisJoinPoint.getArgs()[0] : null) ;
        _log.entering(sig.getDeclaringType().getName(), sig.getName(), arg);
    }
    after() returning (Object ret): loggedCall() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        _log.exiting(sig.getDeclaringType().getName(), sig.getName(), ret) ;
    }
}
```

... this can then be applied to code where needed ...

Aspect Orientation

- The result of using aspects:

```
@LogPoint
@AuthorizationCheck(type=PERSON, access=READ)
public Person getEmployee2(String empId) {
    Person ret = empRepo.getPersonByRole(empId,
                                         Role.EMPLOYEE) ;

    return ret ;
}
```

Cross cutting code
replaced with
annotations

- Non-functional code factored out to be dealt with separately
 - two aspects, one for logging one for security
 - worth noting that the security one is quite complicated

Checking Rules with Aspects

- As well as applying code, aspects can be used to check code structures and dependencies
- AspectJ's "warning" and "error" advice keywords


```
declare error <pointcut> : "error message"
declare warning <pointcut> : "warning msg"
```
- Create a library of pointcuts that allow the error and warning declarations to be read easily
- Better suited to some sorts of checks than others
 - e.g. "don't call X from Y" is easy to do
 - e.g. "If I have an X I should have a Y" is difficult to express

Example Rule Checking Aspect

- We don't want any classes calling JDBC unless it's part of the DAO layer
- We do this by creating pointcuts to define the JDBC layer and JDBC calls and combining them

```
public aspect StructureCheckingAspect {
    pointcut inDao() : within(com.myorg.myapp.dao.*) ;

    pointcut callsJdbc() : call(* java.sql.*.*(..)) ||
                          call(* javax.sql.*.*(..)) ;

    declare error : !inDao() && callsJdbc() :
        "Only call JDBC from DAOs" ;
}
```

Arch Java

- Arch Java is a research project from CMU
 - Created as part of Jonathan Aldrich's PhD in 2003
 - Development continued to 2005, but largely dormant now
 - Not a practical proposition, but a glimpse at the future
- Seamless extension to Java to add design ideas
 - first class components and connectors
 - provides a compiler to compile to vanilla .class files
- While not practical for project use, Arch Java does illustrate extending code for design concepts
 - can only hope that an open source project does something similar!

62

Arch Java Syntax

New keywords
added to Java

Vanilla Java syntax
still used

```
public component class MasterSlave {
    private final Master master = new Master();

    connect pattern Master.Request, Slave.Service
        with AsynchronousConnector {

        // connection constructor called to connect to a slave
        connect(Master sender, int id) {

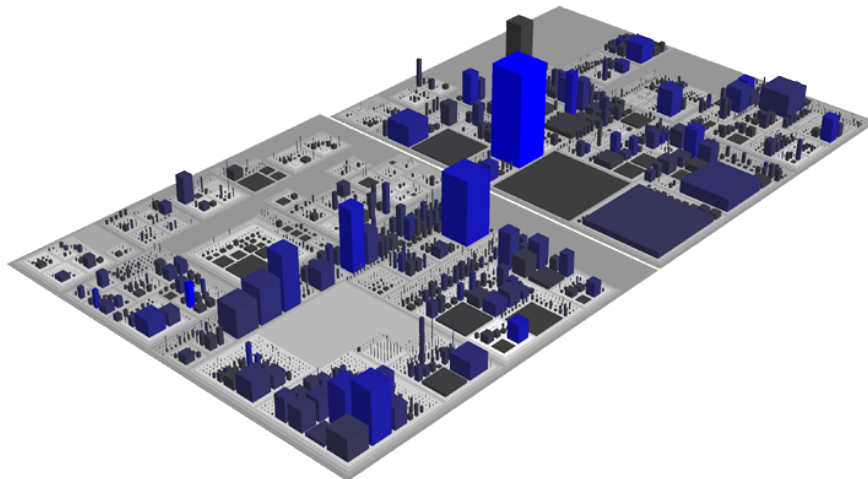
            // create a new slave component
            Slave slave = new Slave(id);

            // create and return an async connection master to slave
            return connect(sender.Request, slave.Service)
                with new AsynchronousConnector(connection);

        }
    }
}
```

Design concepts
introduced into the
code as executable
statements

A Final Tool – Code City Visualisation



This is Code City primarily showing metrics rather than structure
<http://www.inf.usi.ch/phd/wettel/codecity.html>

Summary

Summary

- A lot of valuable design gets lost when you move to code
- Most mainstream tools don't help to prevent this
- We *can* keep, check & recover design
 - careful structuring, modularisation and naming
 - dependency analysis and rules checking
 - external design meta-data
 - new code structuring technologies (e.g. aspects)
- Much of this is new or niche
 - how many projects to do you know with messy design?!
- Use a combination of tools to maintain, check and retrieve design information in/from code
 - integrate tools into the build as well using them interactively

A final Thought ...

**It's late ...
do you know where
your design is?? 😊**

Questions and Comments?

Eoin Woods
www.eoinwoods.info
contact@eoinwoods.info

68