9^{TH} INTERNATIONAL DESIGN STRUCTURE MATRIX CONFERENCE, DSM'07
16 – 18 OCTOBER 2007, MUNICH, GERMANY

USING DSM TO TEST THE SOFTWARE ARCHITECTURE

Neil Langmead

Keywords: Architecture, DSM, Testing, Spaghetti Code, Developer

1 ABSTRACT

Testing is an extremely important part of the software lifecycle, accounting for as much as 50% of the total development costs for many projects. Formalised testing is no longer the exclusive domain of safety critical applications, as toleration of faults and bugs in software is reducing across the software world. With this in mind, rigourous tools, methodologies and solutions are needed to automate as much of the testing work as possible.

One problem often encountered in testing large, monolithic applications is the lack of clarity in the architecture of a software application. Often, a lack of process regarding the architecture leads to so called "spaghetti code", where parts of the system are characterised by inappropriate relalationships, unstructured code blocks and "blobs". Such systems are often impossible to test. Without formal architectural rules in place, and a system or tool to enforce these rules and design intent, the system often becomes unmanageable and untestable. Such systems are also prone to documentation problems, change control issues and generally poor lifecycle management.

This presentation first of all examines the problem of specifying architecture and communicating design intent formally, and then secondly, how to test the implementation of this design intent. It is aimed at software architects, designers, developers and testers. Using advanced DSM-based techniques and tools, it is now possible for the developer and architect to work together to ensure that the problem of architectural degredation and erosion of a software system, so often the biggest problem a software project faces, remains a thing of the past. Architects, developers and testers can now work together to produce better software that is also easier to extend, reuse and understand.

Many software problems are introduced when the design is converted to implementation during the coding phase of a project. The benefits of an *architecture-first* testing approach are shown through a practical demonstration. It can also be shown that a clean architecture leads to reductions in time taken for unit and integration testing phases to complete. At the developer desktop level, a clean architecture can help identify code coverage / dead code issues, and thus many defects can be found even earlier in the lifecycle by adopting this approach.

Contact: Neil Langmead
Emenda Software Limited
Elisabethstrasse 91
80797 Munich
Germany
Tel.: +49 (0) 89 59 08 - 2029
Fax: +49 (0) 89 59 08 - 1200
Mobile: +49 (0) 173 691 3617
Web: www.emenda.eu

Capitalize on Complexity

In cooperation with BMW Group

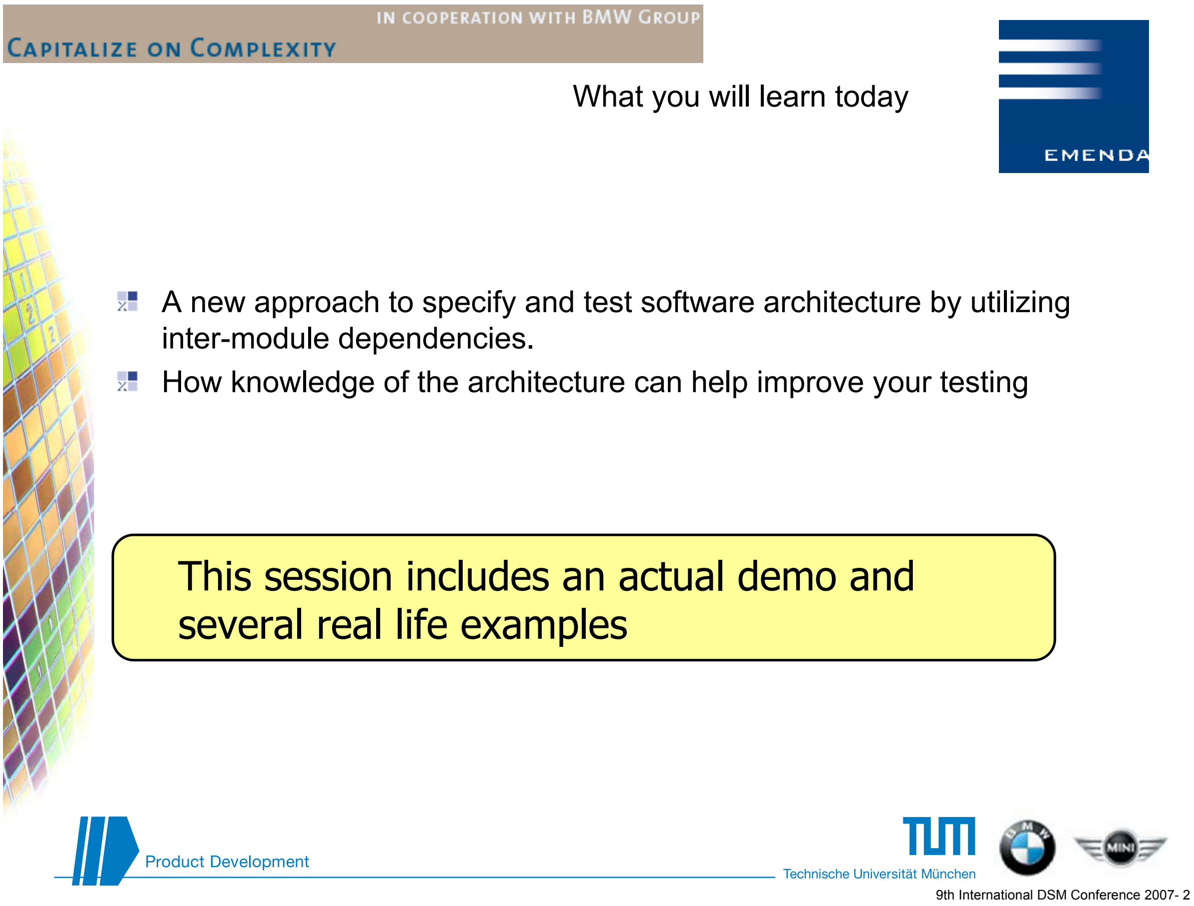
9th International DSM Conference

Testing the Software Architecture

Neil Langmead
Emenda

Product Development

Technische Universität München



Capitalize on Complexity

In cooperation with BMW Group

What you will learn today

EMENDA

- A new approach to specify and test software architecture by utilizing inter-module dependencies.
- How knowledge of the architecture can help improve your testing

This session includes an actual demo and several real life examples

Product Development

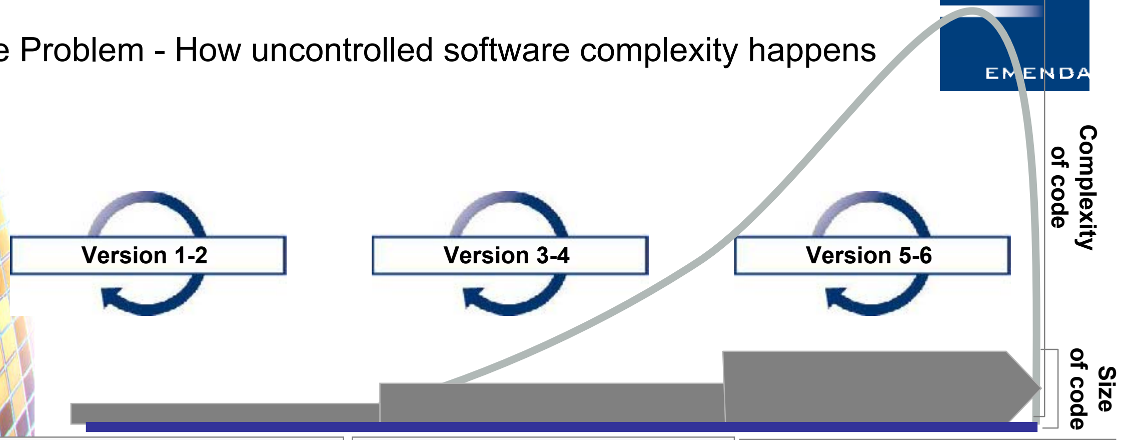
Technische Universität München

9th International DSM Conference 2007- 2

Agenda

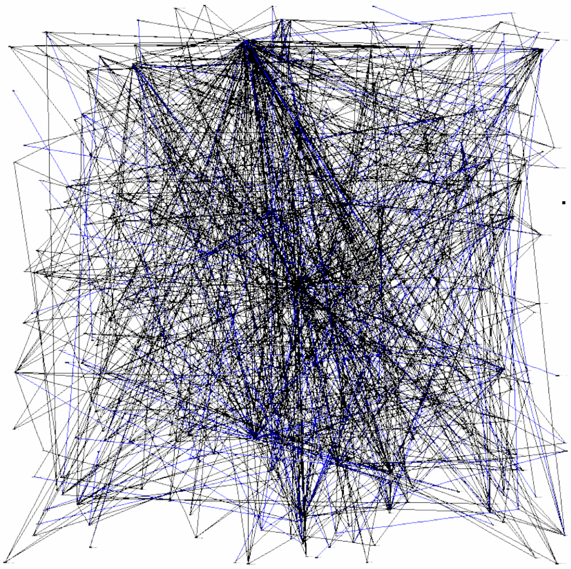
- The Problem!
- Specify architecture
  - What’s a Dependency Structure Matrix (DSM)
  - Architectural Patterns
- Test the architecture with Design Rules
- Demo with a real application
- Improve testing with knowledge of architecture
- Q & A

The Problem - How uncontrolled software complexity happens



Plan	Reality	Plan	Reality	Plan	Reality
▪Tight design spec ▪Solid architecture ▪Comprehensive Functionality	▪Reduced functionality ▪Architecture erosion ▪Limited testing to meet ship date ▪Plan to, “fix bugs in v1.5.”	▪Clean up bugs ▪Take app to “next level” ▪Respond to 1st customer fixes ▪Control Software Complexity	▪Key architects & developers gone ▪Documentation is non-existent ▪New team on learning curve ▪Security concerns	▪Must fix the architecture ▪No new defects ▪Reuse components for other applications ▪Outsource some development	▪Complexity and code is growing too quickly ▪Bug fix versus feature battle underway ▪Architecture is unknown

How do we efficiently test & add features to this software?



… what software looks like after business success!

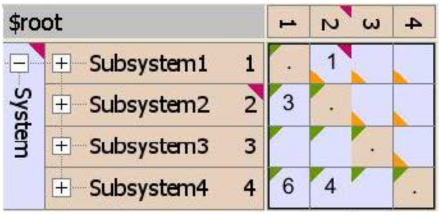
Testing the Architecture with Design Rules

- Succinct specification of acceptable and unacceptable dependencies between subsystems
- Each cell of the DSM represents design intent
- DSM offers a powerful way to visualize and specify design rules
- Design Rules enable testing of architecture

Dependency Model = DSM + Design Rules

Design Rules

EMENDA



$root			1	2	3	4
System	Subsystem1	1	.	1		
	Subsystem2	2	3	.		
	Subsystem3	3			.	
	Subsystem4	4	6	4		.

DSM with Rules View

Green Triangle – Dependency Acceptable

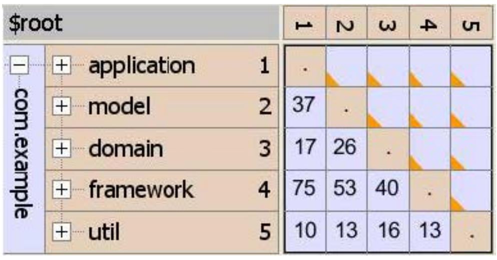
Yellow Triangle – Dependency Unacceptable

Red Triangle – Rule Violation Discovered

- External rules control library and 3rd party usage
- Internal rules are set at highest level of hierarchy and inherited down to lowest level

Design Rules

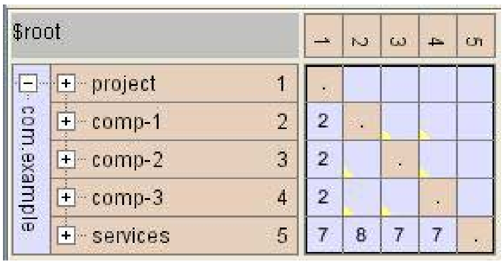
EMENDA



$root			1	2	3	4	5
com.example	application	1	.				
	model	2	37	.			
	domain	3	17	26	.		
	framework	4	75	53	40	.	
	util	5	10	13	16	13	.

Rules for Layering

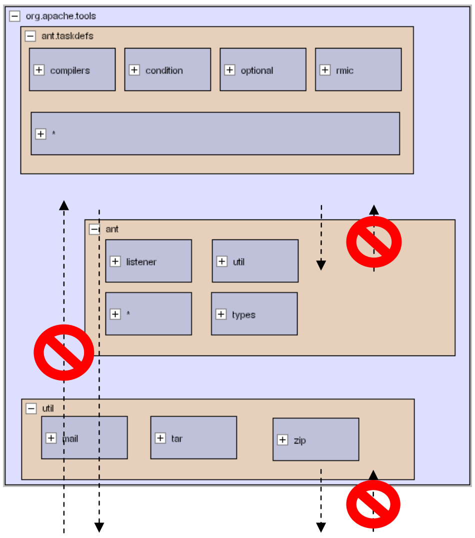
1. $root can-use $root
2. model cannot-use application
3. domain cannot-use application, model
4. framework cannot use application, model, domain
5. util cannot-use application, model, domain, framework



$root			1	2	3	4	5
com.example	project	1	.				
	comp-1	2	2	.			
	comp-2	3	2		.		
	comp-3	4	2			.	
	services	5	7	8	7	7	.

Independent Components

Example: ANT Conceptual Architecture

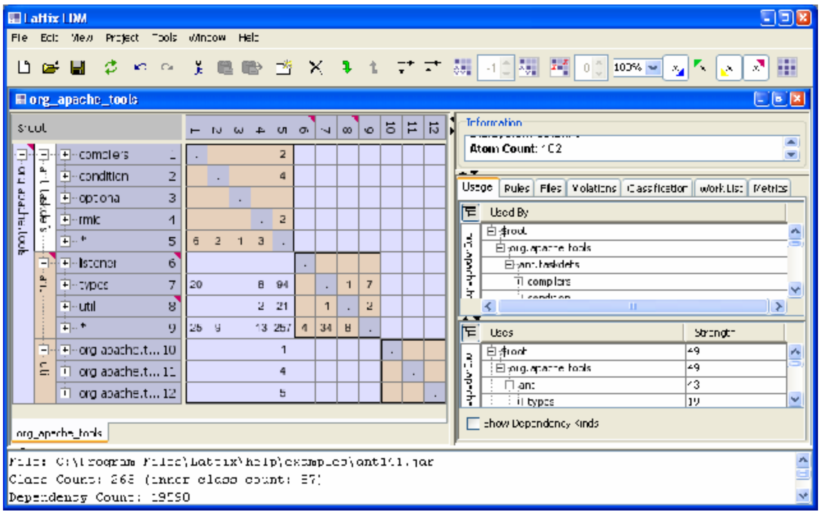


Layered Architecture with three subsystems

Tasks use common infrastructure

Key Goal: Allow independent development of tasks

Demonstration



$root		1	2	3	4	5	6	7	8	9	10	11	12
compilers	1	.				2							
condition	2		.			4							
optional	3			.									
rmic	4				.	2							
*	5	6	2	1	3	.							
listener	6						.						
types	7	20			8	94		.	1	7			
util	8				2	21		1	.	2			
*	9	25	9		13	257	4	34	8	.			
org.apache.t...	10					1					.		
org.apache.t...	11					4						.	
org.apache.t...	12					5							.

Atom Count: 102

Used By
- $root
  - org.apache.tools
    - ant.taskdefs
      - compilers
      - condition

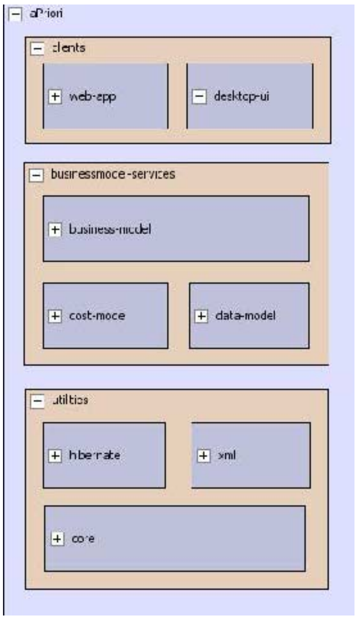
Uses	Strength
$root	49
org.apache.tools	49
ant	43
types	19

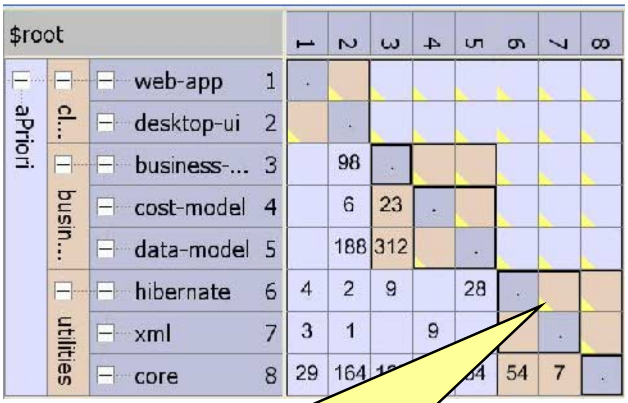
□ Show Dependency Kinds

```
File: C:\Program Files\Lattix\help\examples\ant161.jar
Class Count: 263 (inner class count: 57)
Dependency Count: 19590
```

Improve Testing with Architectural Knowledge

Architectural Visibility improves Testing

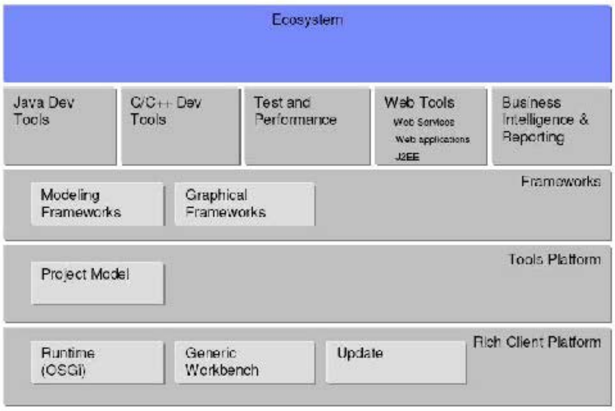




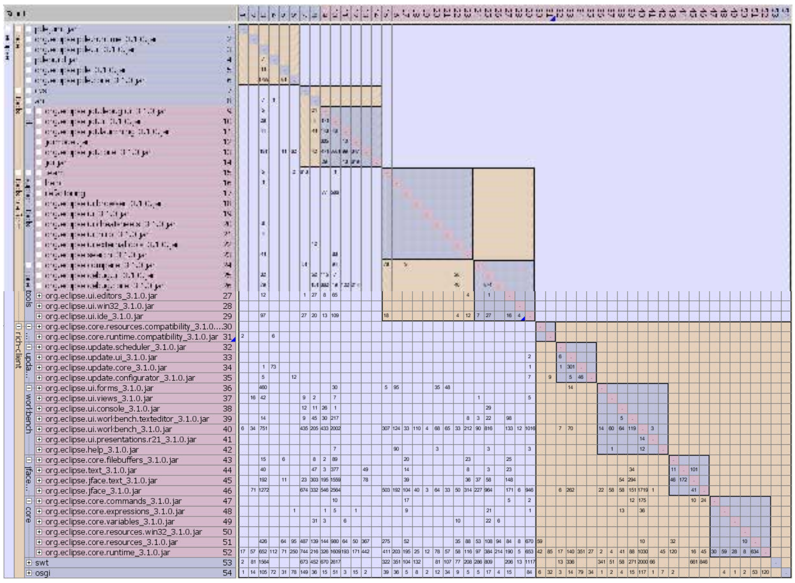
$root				1	2	3	4	5	6	7	8
aPriori	cl...	web-app	1	.							
		desktop-ui	2		.						
	busin...	business-...	3		98	.					
		cost-model	4		6	23	.				
		data-model	5		188	312		.			
	utilities	hibernate	6	4	2	9		28	.		
		xml	7	3	1		9			.	
		core	8	29	164	[illegible]	[illegible]	[illegible]	54	7	.

Design Rules communicate the intent and enable testing

Conceptual Architecture of Eclipse?

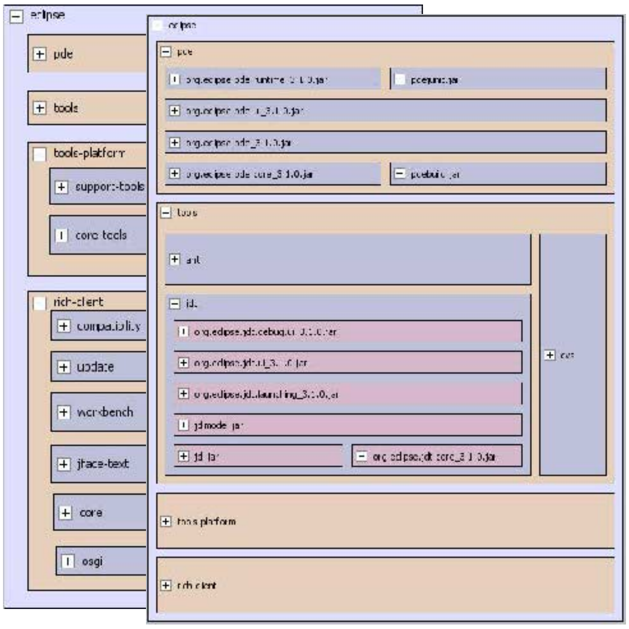


Dependency Model View of Eclipse Platform



org.eclipse.ui.editors_3.1.0.jar 27
org.eclipse.ui.win32_3.1.0.jar 28
org.eclipse.ui.ide_3.1.0.jar 29
org.eclipse.core.resources.compatibility_3.1.0....30
org.eclipse.core.runtime.compatibility_3.1.0.jar 31
org.eclipse.update.scheduler_3.1.0.jar 32
org.eclipse.update.ui_3.1.0.jar 33
org.eclipse.update.core_3.1.0.jar 34
org.eclipse.update.configurator_3.1.0.jar 35
org.eclipse.ui.forms_3.1.0.jar 36
org.eclipse.ui.views_3.1.0.jar 37
org.eclipse.ui.console_3.1.0.jar 38
org.eclipse.ui.workbench.texteditor_3.1.0.jar 39
org.eclipse.ui.workbench_3.1.0.jar 40
org.eclipse.ui.presentations.r21_3.1.0.jar 41
org.eclipse.help_3.1.0.jar 42
org.eclipse.core.filebuffers_3.1.0.jar 43
org.eclipse.text_3.1.0.jar 44
org.eclipse.jface.text_3.1.0.jar 45
org.eclipse.jface_3.1.0.jar 46
org.eclipse.core.commands_3.1.0.jar 47
org.eclipse.core.expressions_3.1.0.jar 48
org.eclipse.core.variables_3.1.0.jar 49
org.eclipse.core.resources.win32_3.1.0.jar 50
org.eclipse.core.resources_3.1.0.jar 51
org.eclipse.core.runtime_3.1.0.jar 52
swt 53
osgi 54

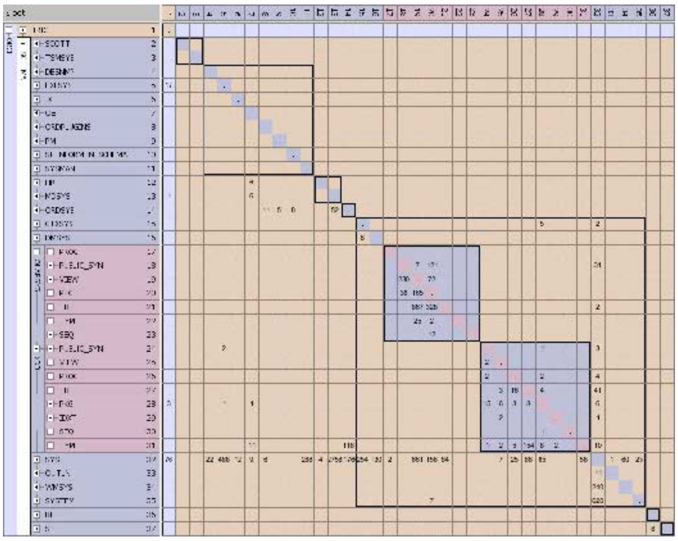
Conceptual Architecture - Eclipse Platform


eclipse
pde
tools
tools-platform
support-tools
core tools
rich-client
compatibility
update
workbench
jface-text
core
osgi
eclipse
pde
org.eclipse.pde.runtime_3.1.0.jar
pdejunit.jar
org.eclipse.pde.ui_3.1.0.jar
org.eclipse.pde_3.1.0.jar
org.eclipse.pde.core_3.1.0.jar
pdebuild.jar
tools
ant
jdt
org.eclipse.jdt.debug.ui_3.1.0.jar
org.eclipse.jdt.ui_3.1.0.jar
org.eclipse.jdt.launching_3.1.0.jar
jdimodel.jar
jdt.jar
org.eclipse.jdt.core_3.1.0.jar
cvs
tools platform
rich client

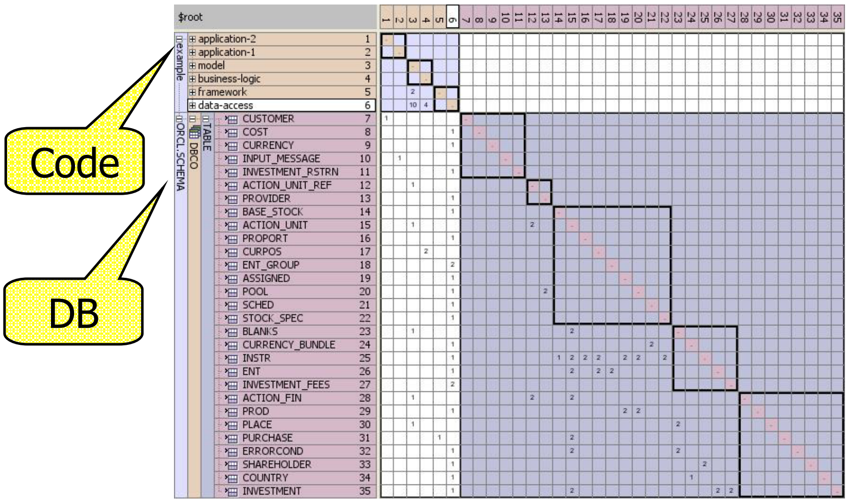
Precise big picture view derived from the Dependencies

View shows accurate layering and vertical splitting

Testing Data Architecture …



Testing the Enterprise Architecture



$root			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
example	application-2	1	-																																		
	application-1	2		-																																	
	model	3			-																																
	business-logic	4				-																															
	framework	5			2		-																														
	data-access	6			10	4		-																													
ORCL.SCHEMA DBCO TABLE	CUSTOMER	7	1						-																												
	COST	8						1		-																											
	CURRENCY	9						1			-																										
	INPUT_MESSAGE	10		1								-																									
	INVESTMENT_RSTRN	11						1					-																								
	ACTION_UNIT_REF	12			1									-																							
	PROVIDER	13						1							-																						
	BASE_STOCK	14						1								-																					
	ACTION_UNIT	15			1									2			-																				
	PROPORT	16						1										-																			
	CURPOS	17				2													-																		
	ENT_GROUP	18						2												-																	
	ASSIGNED	19						1													-																
	POOL	20						1							2							-															
	SCHED	21						1															-														
	STOCK_SPEC	22						1																-													
	BLANKS	23			1												2								-												
	CURRENCY_BUNDLE	24						1															2			-											
	INSTR	25						1								1	2	2	2		2	2		2			-										
	ENT	26						1									2		2	2								-									
	INVESTMENT_FEES	27						2																					-								
	ACTION_FIN	28			1									2			2													-							
	PROD	29						1													2	2									-						
	PLACE	30			1																				2							-					
	PURCHASE	31					1										2																-				
	ERRORCOND	32						1									2								2									-			
	SHAREHOLDER	33						1																			2								-		
	COUNTRY	34						1																		1										-	
	INVESTMENT	35						1									2											2	2								-

Summary: Testing the Big Picture View

- Specify the big picture view using DSMs - approach allows you to represent massive systems
- Formalize design intent so you can test architectural erosion
- Easy to adopt—Use it at any stage of the lifecycle
- Use the knowledge of the architecture to improve your testing