software studio

modularity & dependences

Daniel Jackson

what makes a system "modular"?

in traditional engineering

› components can be built, tested & replaced independently

in software engineering

› components can also be reasoned about independently

› changes to components are "localized" or "contained"

containing failures?

› may not follow from modularity

› maybe the opposite (since modularity encourages sharing)

> "The current configuration of electronics on the Dreamliner puts passenger electronic entertainment on the same computer network as the flight control system."
>
> http://www.wired.com/politics/security/news/2008/01/dreamliner_security

when does modularity fail?

client-service binding

- when service changes, client must too
- eg: old apps fail on new release of OS

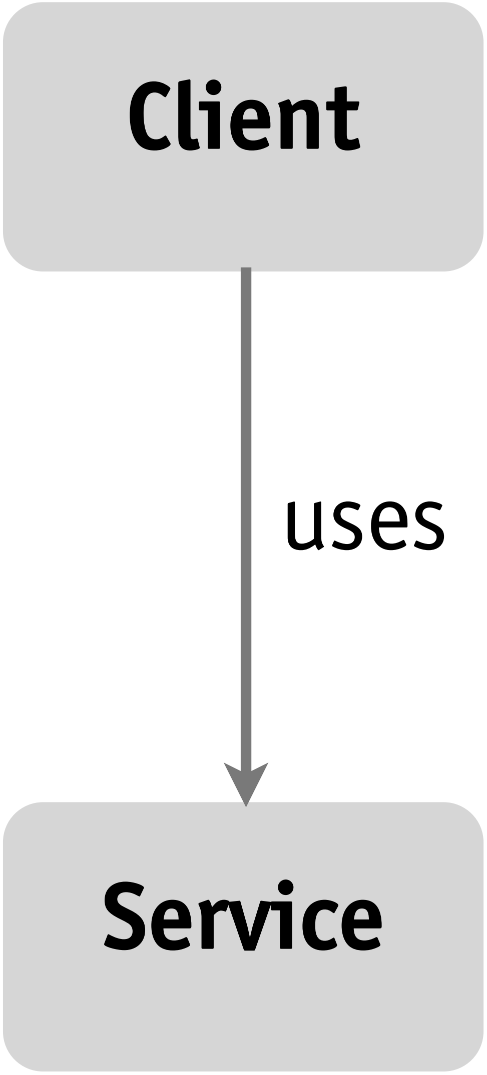
abstraction violation

- service doesn’t change, but client must anyway
- eg: representation of datatype is leaked

module-problem binding

- one piece of the problem in two modules
- eg: document is paragraph-structured, in Word

Parnas's uses relation

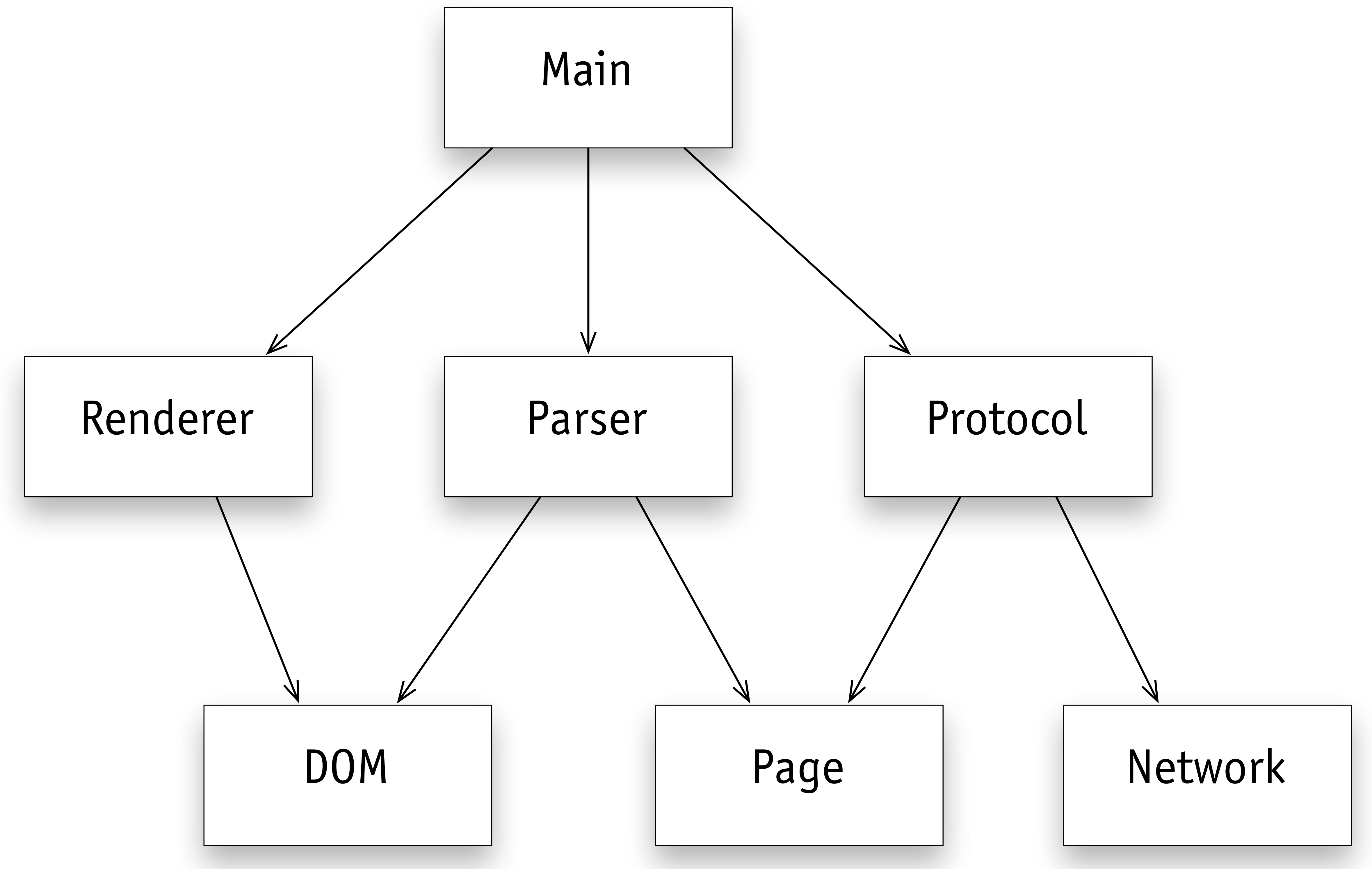


Client

uses

Service

example: a browser



minimal subsets

a common problem

- suppose you want module M
- what other modules do you need?

solution

- you need all the modules M uses
- and the ones they use...

examples

- minimal subset for Renderer? for Parser?

other uses of uses

you change module M

- which modules might break?

you want to test M

- which modules must be complete?

you want to reason about M

- which module's specs do you need?

software subtlety

X may use Y without knowing about it

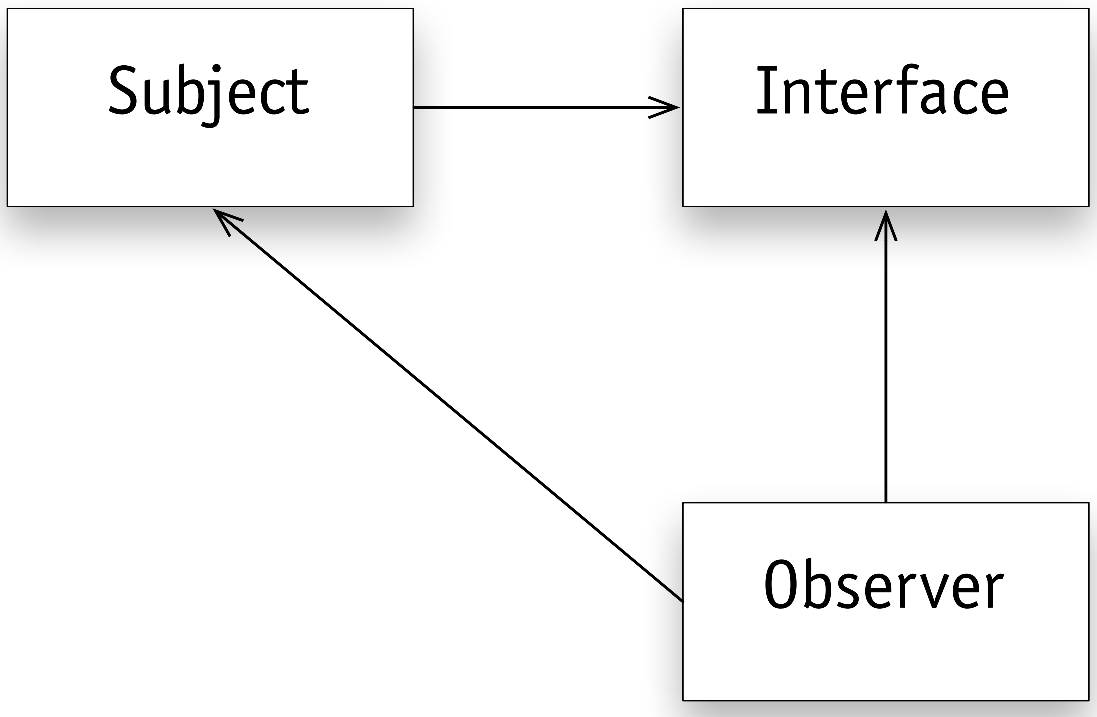
› eg, because Y is configured dynamically

› X only knows interface of Y

example: observer pattern

› interface I interposed between subject S and observer O

› now S depends on I, but not on O

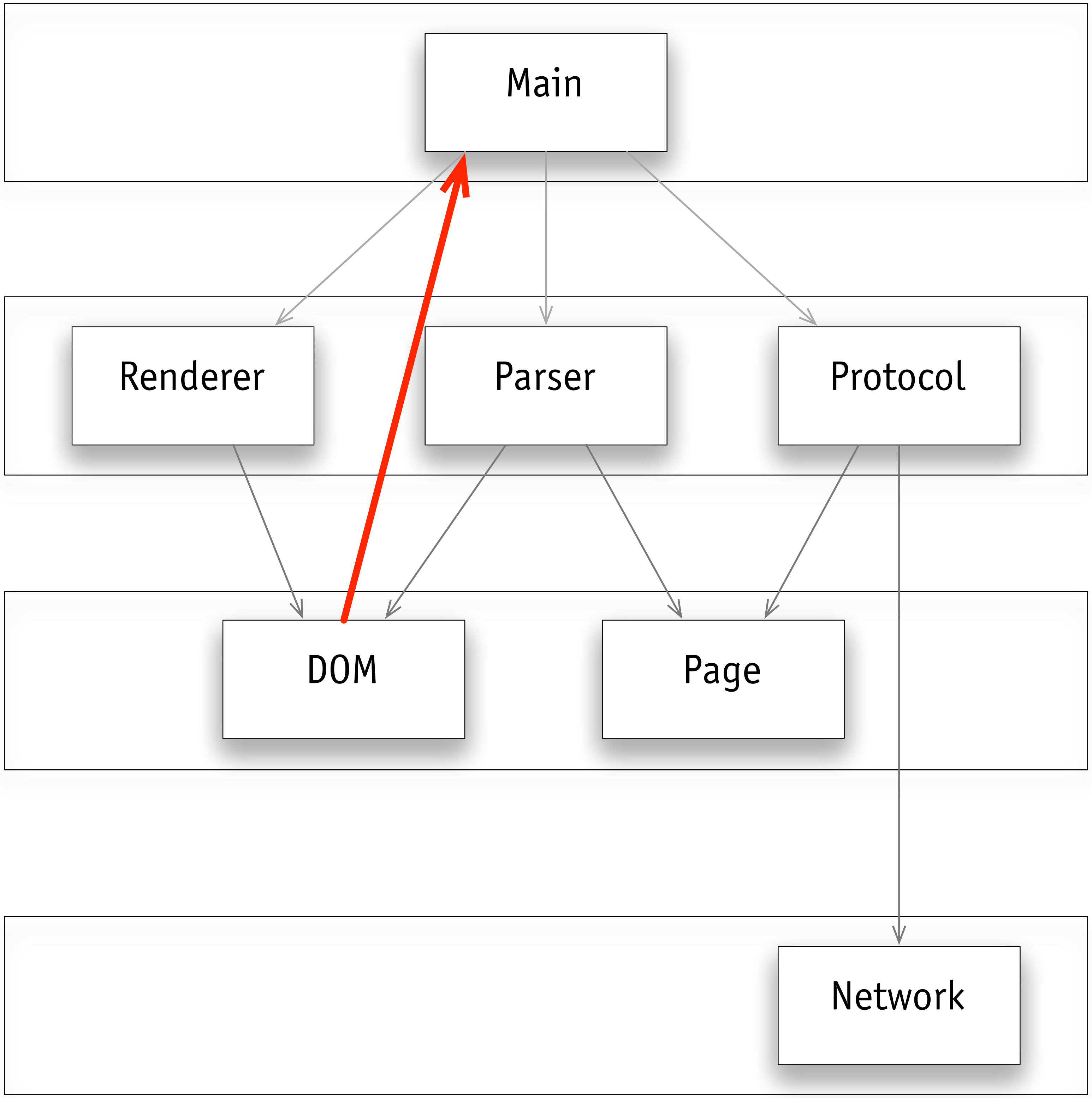


layering: a common pattern

Diagram of Android's multi-layered operating system architecture (in "The Embedded Beat") removed due to copyright restrictions.

Android architecture from https://community.freescale.com/community/the-embedded-beat/blog/2010/05/24/android-makes-the-move-to-power-architecture-technology

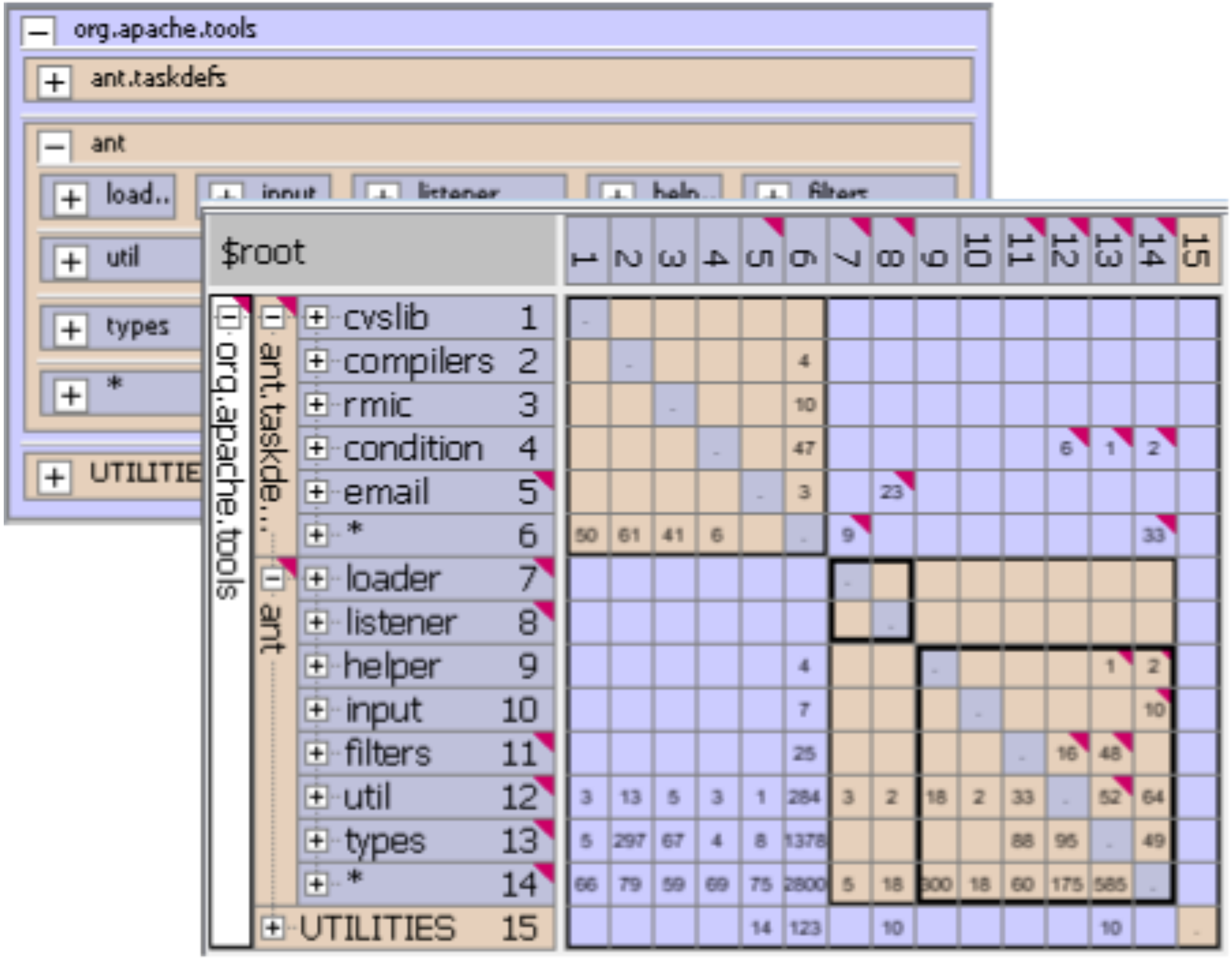
back edges



design structure matrix

Matrix of classes of Spring framework (in "Dependency Structure Matrix")
removed due to copyright restrictions.

highlighting back edges



$root	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
cvslib 1	-														
compilers 2		-				4									
rmic 3			-			10									
condition 4				-		47						6	1	2	
email 5					.	3		23							
* 6	50	61	41	6		.	9							33	
loader 7							.								
listener 8								.							
helper 9						4			.				1	2	
input 10						7				.				10	
filters 11						25					.	16	48		
util 12	3	13	5	3	1	284	3	2	18	2	33	.	52	64	
types 13	5	297	67	4	8	1378					88	95	.	49	
* 14	66	79	59	69	75	2800	5	18	300	18	60	175	585	.	
UTILITIES 15					14	123		10					10		.

Courtesy of Lattix, Inc. Used with permission.

from: http://www.lattix.com/products/modules/java

how to avoid modularity failures

client-service binding

- control dependences, especially back edges

abstraction violation

- make sure clients only rely on specs
- use language abstraction constructs

module-problem binding

- encapsulate design decisions
- this is “information hiding”

DRY

a rule of thumb

- “Don’t Repeat Yourself”

can you explain this rule?

- how does it relate to *uses*? information hiding?
- what are its limitations?