See it as you need it

http://24.media.tumblr.com/tumblr_lp9ydv5Wgp1qg8x7mo1_500.jpg

Code / Slides / ME @

- https://github.com/schuchert/spring_aop
  - Clone the repo:
    git clone git://github.com/schuchert/spring_aop.git
  - Uses Maven (get over it)

- Me:
  @schuchert
  schuchert@yahoo.com
  shoe@thoughtworks.com

Pragmatics on Principles

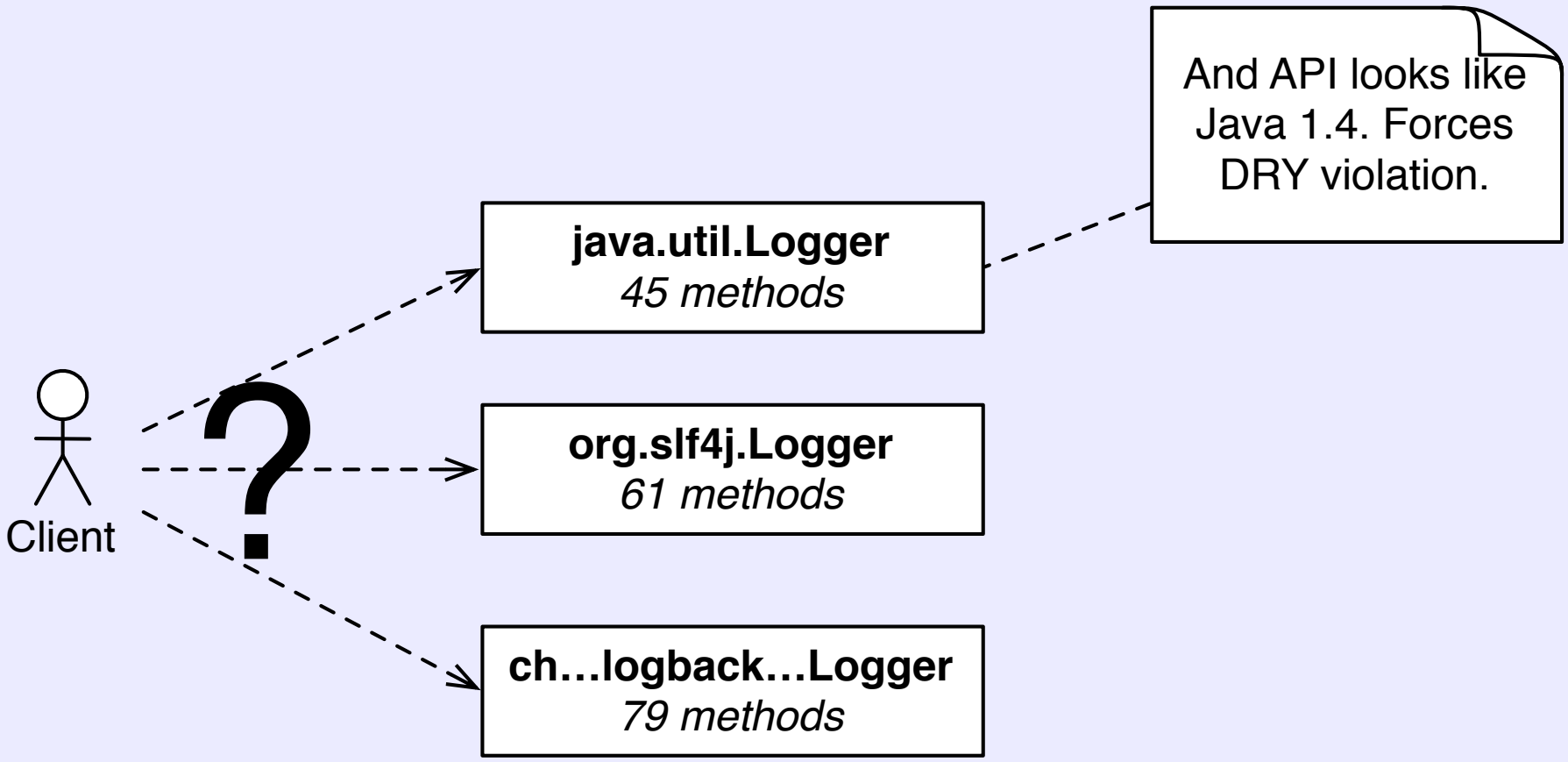
- Not personally a fan of "best practices"
  - I do like good ideas for a given context
- These are all examples from real projects
  - But I've not uses ***all*** of these on any ***one*** project
- Design principles
  - Should be "violated" sometimes
  - Are often at odds with each other
- There are no free lunches
  - All abstractions have a cost

Dependency Inversion Principle

- Abstractions should not depend on details, details should depend on abstractions
  http://c2.com/cgi/wiki?DependencyInversionPrinciple

- Abstraction could be
  - Interface
  - Abstract Base Class
  - Idea
  - Design level, e.g. depending on a low-level interface can still violate DIP

Logger


And API looks like Java 1.4. Forces DRY violation.
java.util.Logger
45 methods
org.slf4j.Logger
61 methods
ch…logback…Logger
79 methods
Client
?

Logger

- Too many methods
  - Discipline required for consistency
  - JDK Logger forces DRY violation
  - Different OSS libs use different versions
- Reducing flexibility
  - Less discipline required
  - Avoid DRY violation through API design

Efficiency vs. DRY

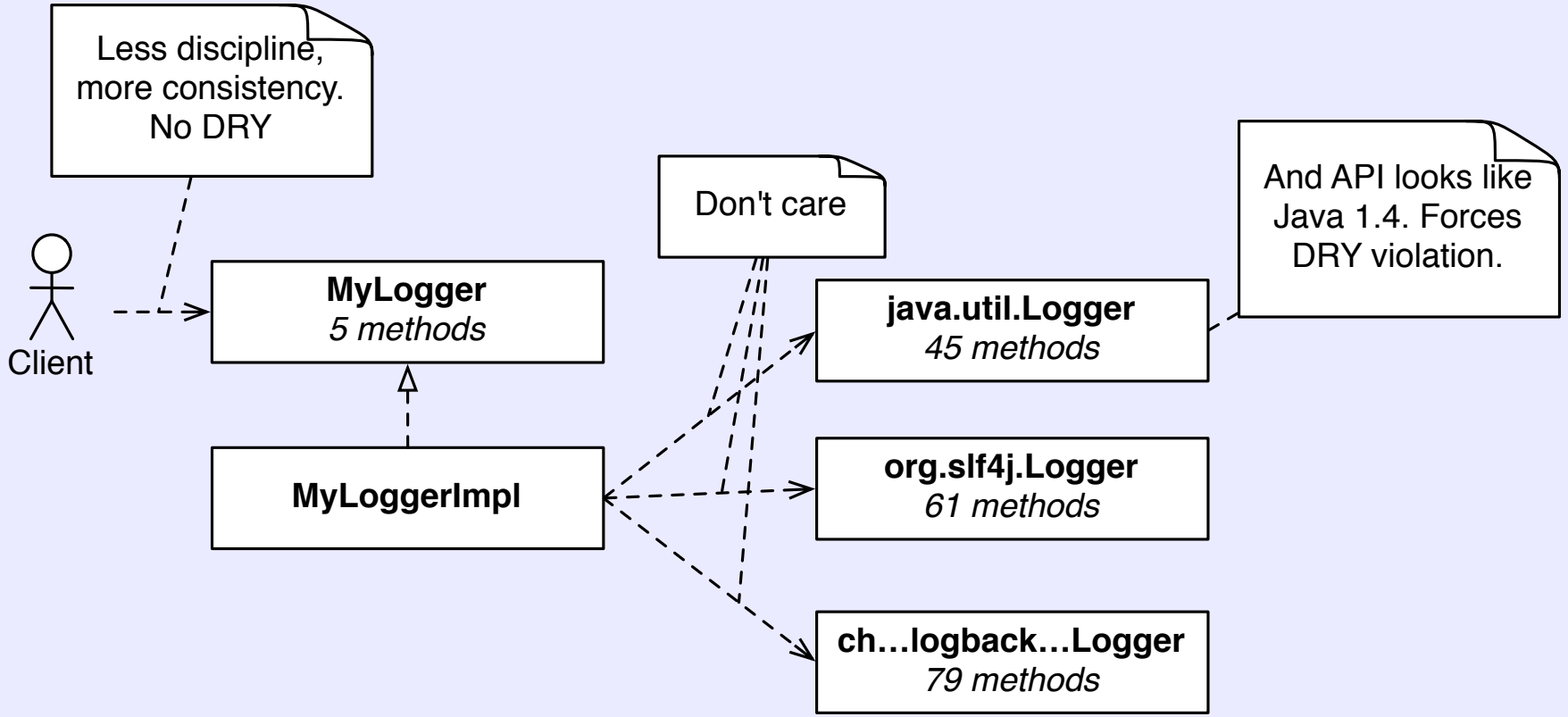
- Consider this...

```
Logger logger = Logger.getLogger(getClass().getName());
String message =
  String.format("%s-%s-%s", "part1", "part2", "part3");
logger.log(Level.INFO, message);
```

- Vs.

```
Logger logger = Logger.getLogger(getClass().getName());
  if (logger.isLoggable(Level.INFO)) {
    String message =
      String.format("%s-%s-%s", "part1", "part2", "part3");
    logger.log(Level.INFO, message);
  }
}
```

Logger


Less discipline,
more consistency.
No DRY
Client
MyLogger
5 methods
MyLoggerImpl
Don't care
java.util.Logger
45 methods
And API looks like
Java 1.4. Forces
DRY violation.
org.slf4j.Logger
61 methods
ch…logback…Logger
79 methods

Logger

- Vs.

```
SystemLogger logger = SystemLoggerFactory.get(getClass());
logger.info("%s-%s-%s", "part1", "part2", "part3");
```

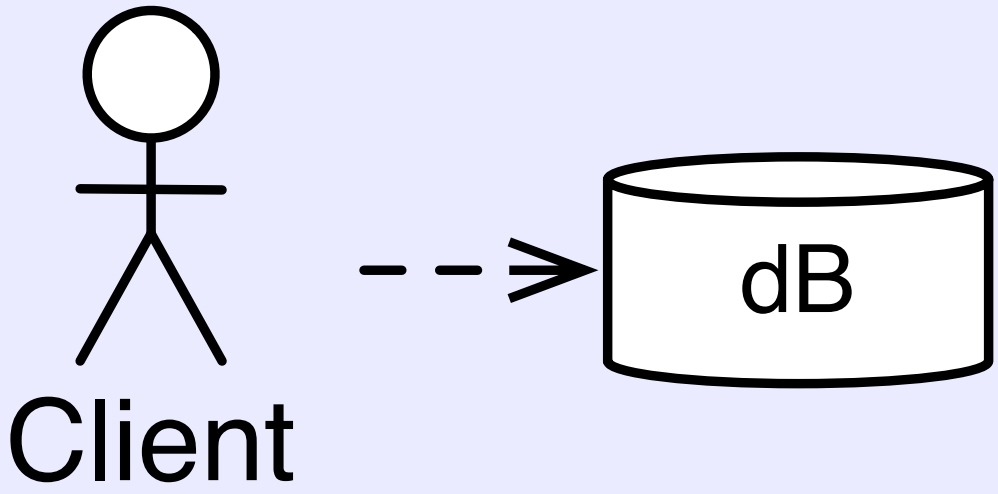
- And one such implementation...

```
public void info(String message, Object... args) {
  if (logger.isInfoEnabled()) {
    logger.info(String.format(message, args));
  }
}
```

- Still passing parameters...but
  - No unnecessary String concatenation
  - Reduces object creation & gc

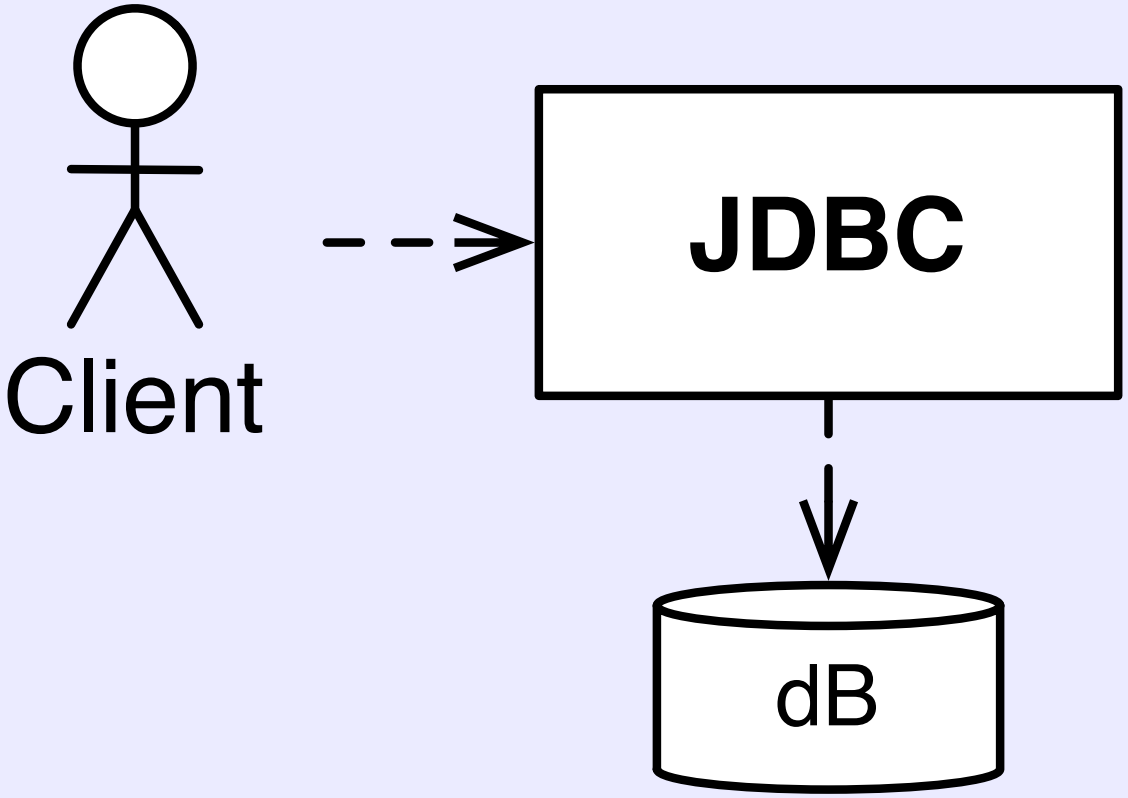
Depending on Interfaces

- Starting with



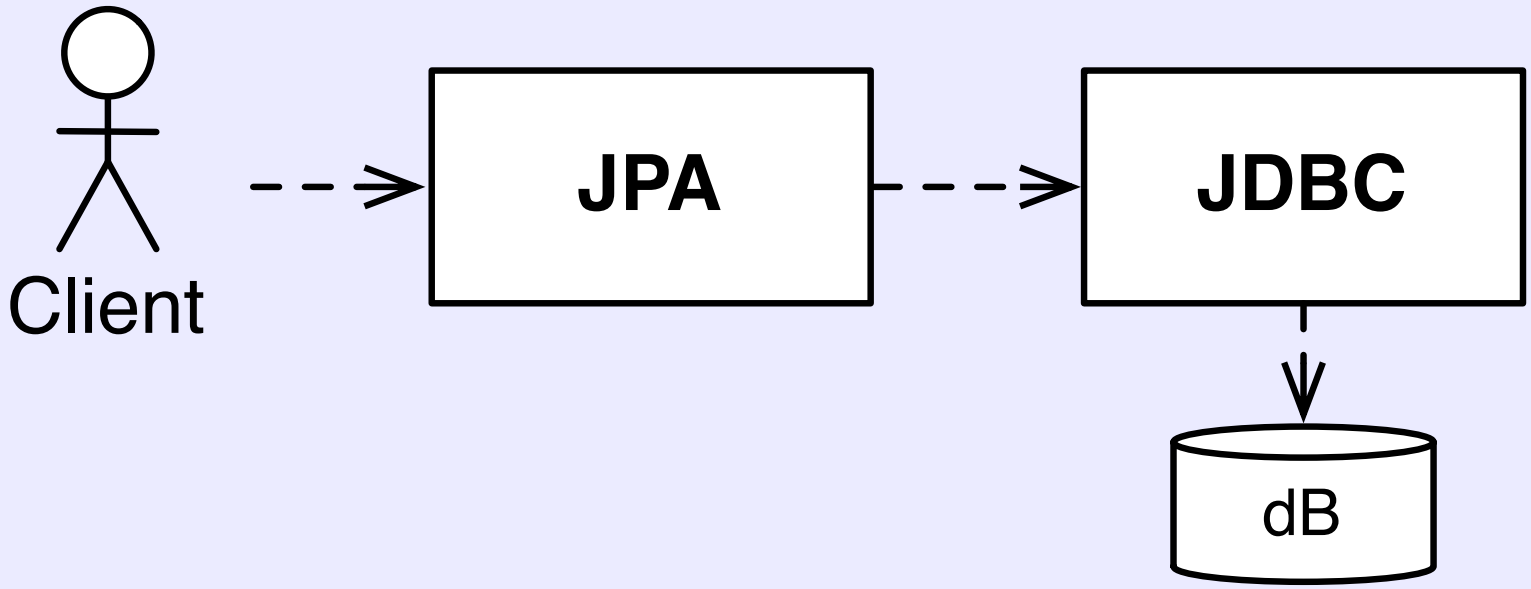
Depending on Interfaces

- Last century we had...Better?



Depending on Interfaces

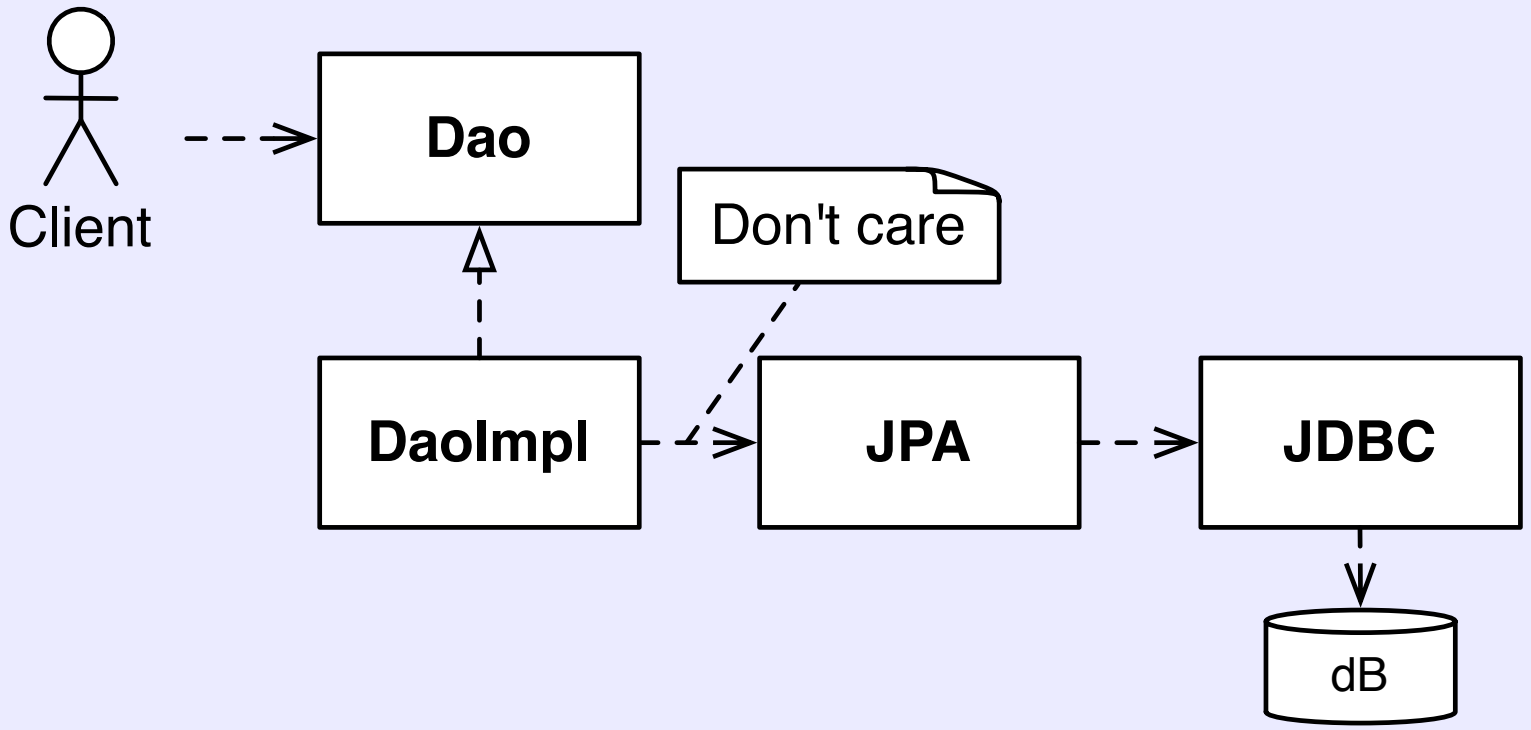
- 2005ish we had...Better?



- Hibernate came before, not essentially different.

Depending on Interfaces

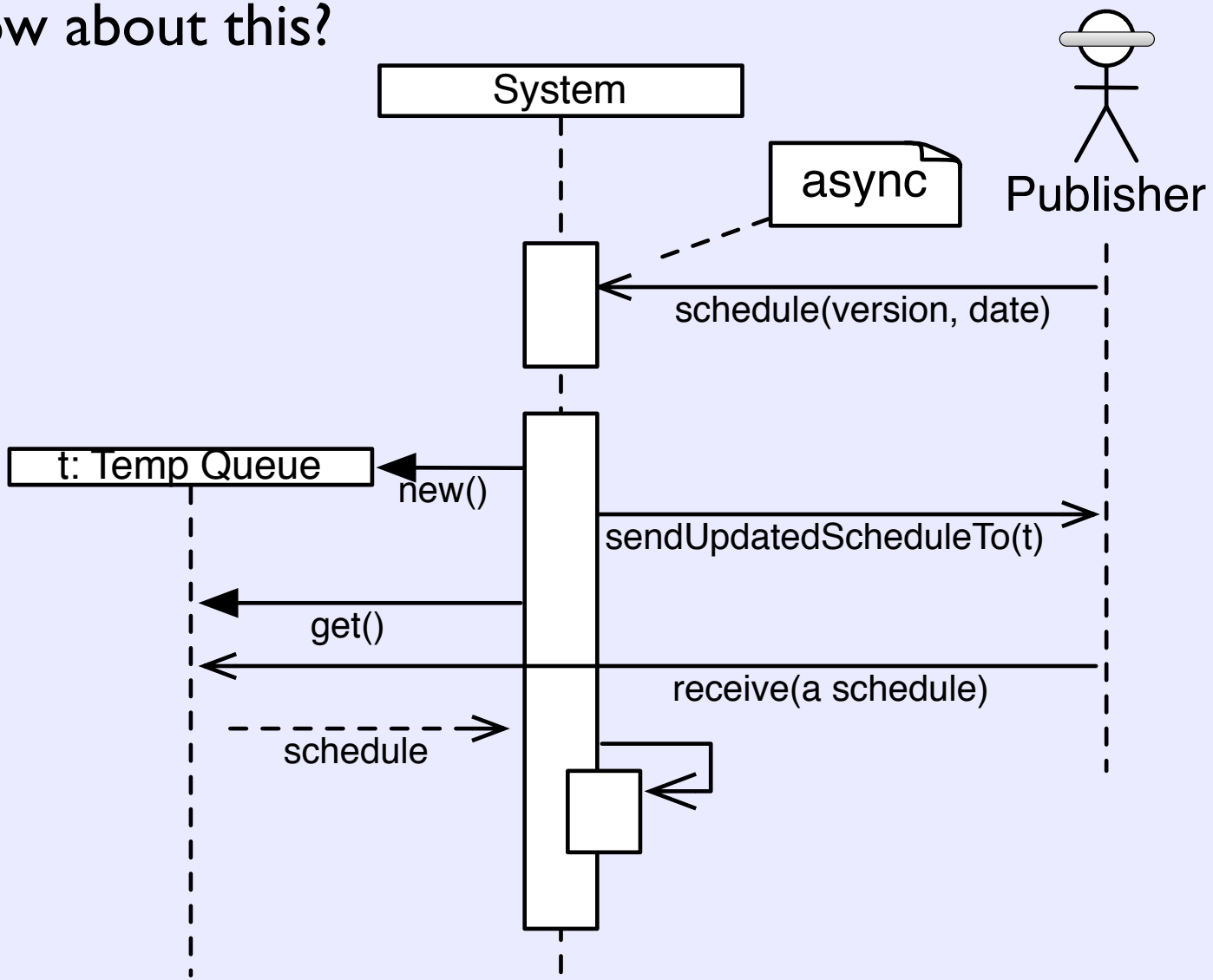
- Better?



- Modern approaches avoid DaoImpl for "normal" stuff

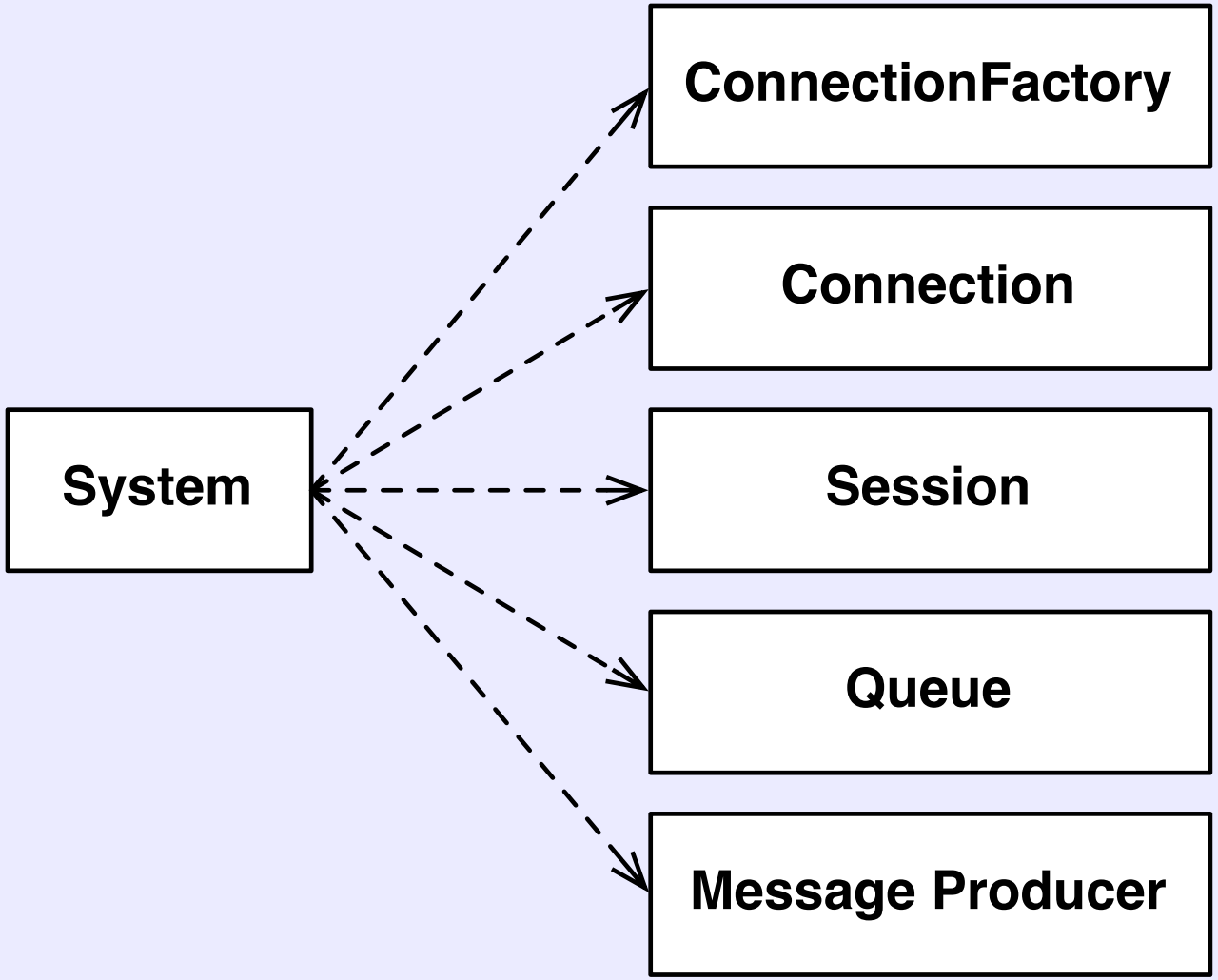
Message Oriented Middleware

- How about this?



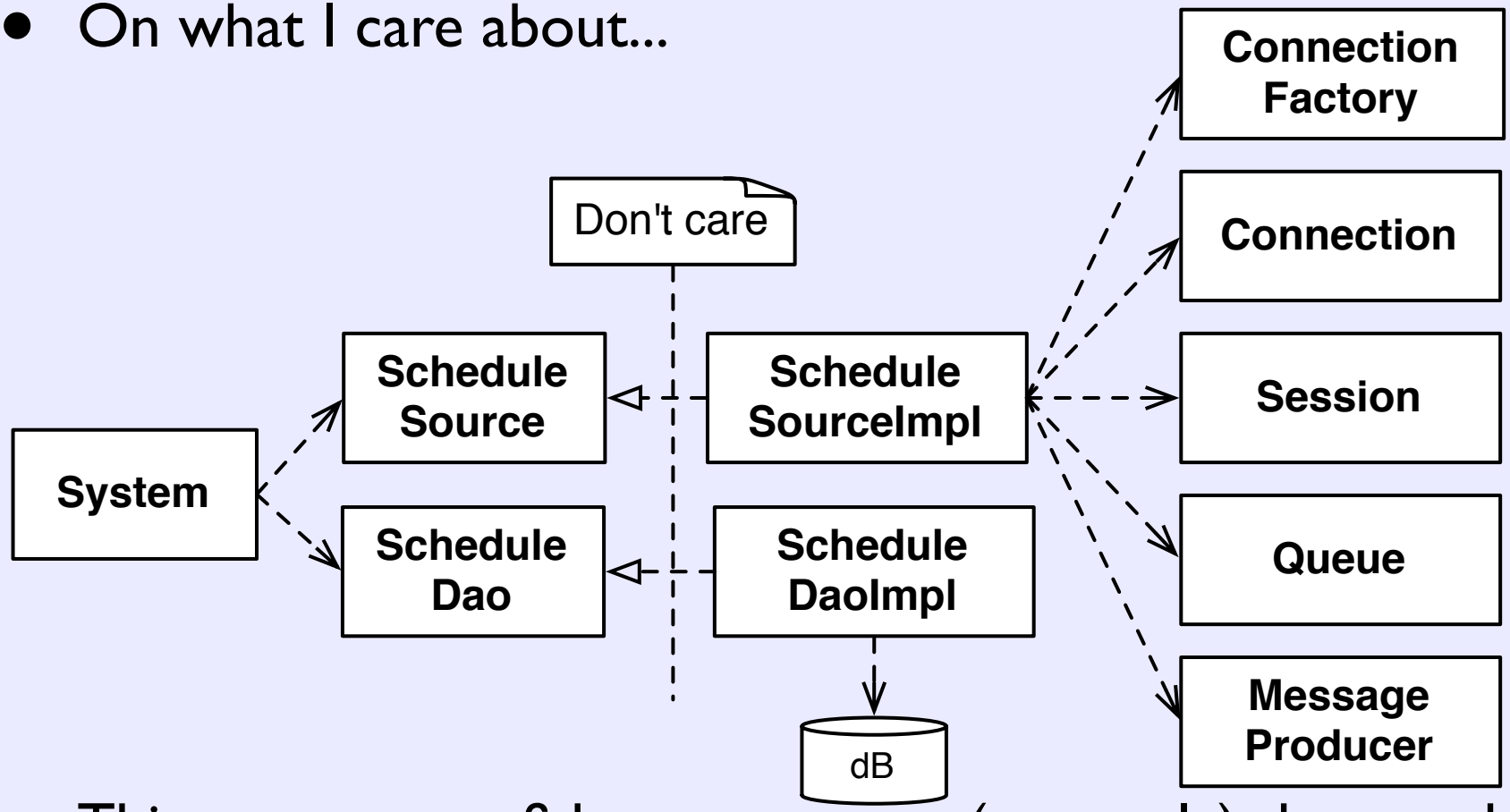
As a system...

- I can depend on a bunch of JMS interfaces...



Or I Can Depend

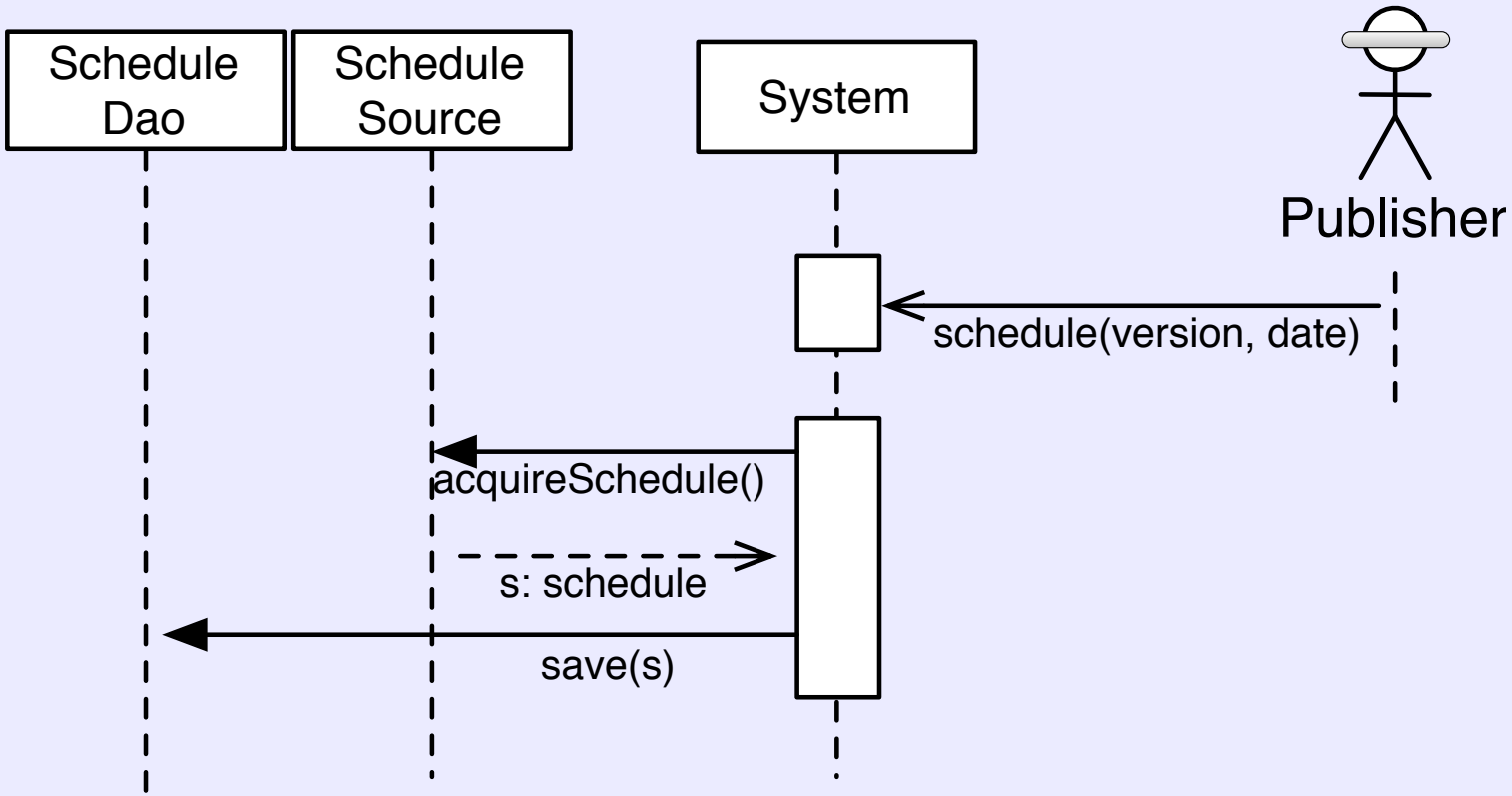
- On what I care about...



- This gave me confidence to assert (correctly) that prod and QA system queues were inconsistently configured.

Divide & Conquer

- Single Responsibility + Dependency Inversion ++++



Given...

- We need to
  - Capture metrics on execution times across:
    services, repositories and components
  - We want to optionally log this information
  - When logged, correlate related log entires with a unique ID
- These are sanitized from a real project, there were more constraints, but this gives major factors

What would you do with...

```
01:private Object executeShell(ProceedingJoinPoint jp) throws Throwable {
02:  CorrelationId.enter();
03:  String className = jp.getSignature().getDeclaringTypeName();
04:  String methodName = jp.getSignature().getName();
05:  MetricName name = new MetricName(group(), className, methodName);
06:  SystemLogger targetLogger = SystemLoggerFactory.get(className);
07:  targetLogger.info("start :%s-%s", methodName, CorrelationId.get());
08:  Timer responses = Metrics.newTimer(name, MILLISECONDS, SECONDS);
09:  TimerContext context = responses.time();
10:  long start = System.currentTimeMillis();
11:  try {
12:    Object result = jp.proceed();
13:    targetLogger.info("finish :%s-%s(%dms)", methodName, CorrelationId.get(), System.currentTimeMillis() - start);
14:    return result;
15:  } catch (Throwable t) {
16:    targetLogger.info("failing:%s-%s(%dms)", methodName, CorrelationId.get(), System.currentTimeMillis() - start);
17:    throw t;
18:  } finally {
19:    context.stop();
20:    CorrelationId.exit();
```

Additional Resources

Design, Design, Design

- Here's a staring list to help with OOD

GRASP	Craig Larman
SOLID	Robert Martin
CODE SMELLS	Martin Fowler
WELC	Michael Feathers
TEST DOUBLES	Several
CODING KATAS	Several
DESIGN PATTERNS	Gang of 4

- http://schuchert.wikispaces.com/TddIsNotEnough

GRASP

Information Expert	Assign responsibility to the thing that has the information.
Controller	Assign system operations (events) to a non-UI class. May be system-wide, use case driven or for a layer.
Low Coupling	Try to keep the number of connections small. Prefer coupling to stable abstractions.
High Cohesion	Keep focus. The behaviors of a thing should be related. Alternatively, clients should use all or most parts of an API.
Polymorphism	Where there are variations in type, assign responsibility to the types (hierarchy) rather than determine behavior externally,
Pure Fabrication	Create a class that does not come from the domain to assist in maintaining high cohesion and low coupling.
Protected Variations	Protect things by finding the change points and wrapping them behind an interface. Use polymorphism to introduce variance.

SOLID Principles

- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

S	**Single Responsibility**	Single Reason to Change
O	**Open/Closed**	Open for extension closed to change
L	**Liskov Substitution**	Derived types substitutable for base types
I	**Interface Segregation**	Interfaces should be focused (small) & client specific
D	**Dependency Inversion**	Dependencies should go from concrete to abstract

Package Cohesion/Coupling

- Guidelines for package cohesion

REP	Release/Reuse Equivalency	What you release is what you reuse.
CCP	Common Closure	Classes that change together should be packaged together
CRP	Common Reuse	Classes that are used together should be packaged together

- Guidelines for package coupling

ADP	Acyclic Dependencies	No cycles in your dependencies
SDP	Stable Dependencies	Dependencies should go from less to more stable. Depend on stable things
SAP	Stable Abstractions	Abstraction increase with stability

A Few Code Smells

- A few of Martin's code smells:

Poor Names	Name suggests wrong intent
Long methods	More than 1 thing/multiple levels of abstraction
Large classes	More than one concept/multiple levels of abstraction
Long parameter list	Too many arguments to keep straight (> 3)
Duplicated code	Same or similar code appears in more than one place
Divergent change	The class/method changes for dissimilar reasons
Shotgun Surgery	Single change affects multiple classes/methods
Feature Envy	One class uses another class' members
Switch Statements	Duplicated switches/if-else's over same criterion

- http://c2.com/cgi/wiki?CodeSmell

Some Legacy Refactorings

- From Working Effectively with Legacy Code

Adapt Parameter	326	Change parameter to an adapter when you cannot use extract interface
Break Out Method Object	330	Convert method using instance data into a class with a ctor and single method
Encapsulate Global References	339	Move access to global data into access via a class to allow for variations during test
Extract and Override Call	348	Turn chunk of code into overridable method and then subclass in test
Extract and Override Getter	352	Turn references into hard-coded object into call to getter and then subclass
Extract Interface	362	Extract interface for concrete class, then use interface. Override in test.
Introduce Instance Delegator	317	Add instance methods calling static methods. Call through instance, which test subclasses.
Parameterize Constructor **Parameterize Method**	379 383	Examples of Inversion of Control (IoC)
Subclass and override method	401	Test creates subclass & passes it in/requires some IoC
Sprout Method **Sprout Class**	59 63	Create a method or class out of existing code.

Test Doubles

- Gerard Meszaros
  http://xunitpatterns.com/Test%20Double%20Patterns.html

DUMMY	Empty implementation. Not called or don't care if it is
STUB	Canned replies – "snapshot in time"
SPY	Watches and Records
FAKE	Partial Simulator
MOCK	Has & Validates expectations
SABOTEUR	Designed to always fail, e.g., always throws an exception.

F.I.R.S.T.

- http://pragprog.com/magazines/2012-01/unit-tests-are-first

F	**FAST**	Tests should be fast. So fast that you won't hesitate to run them. Unit tests, 1000's per second. Acceptance tests, we'll discuss.
I	**ISOLATED INDEPENDENT**	A test should fail because the production code is wrong. If it fails because of an uncontrolled external dependency make that dependency configurable. A test affects no other tests.
R	**RELIABLE REPEATABLE**	A test should run every time and fail/succeed the same way. Two people should be able to run the same test at exactly the same time on the same machine.
S	**SMALL**	Focused. The smaller the test, the more detailed the check. The larger the test, the less it should check. Too many checks leads to ambiguous failures.
T	**TIMELY**	Should be written about the same time as the production code. If you don't design for testability, it'll probably be hard to test. The longer you wait, the more it costs.

Design Patterns

- From: Design Patterns: Elements of Reusable Object-Oriented Software

STRATEGY	Define a function or algorithm as a class. Form a wide but shallow hierarchy of different algorithms.
TEMPLATE METHOD	Write an algorithm in a base class with extension points represented as abstract methods. Subclass and override.
ABSTRACT FACTORY	A base interface for creating one or a family of objects through a standard API. Create implementations for each family of objects that need to be created.
COMPOSITE	A class that implements some other interface and also holds onto zero or more instances of that same interface.
STATE	Similar to strategy, though the states are interdependent. States can cause a so-called context to change from one state to another during its lifetime.

Additional Resources

- Video Series

C++	Dice Game	http://vimeo.com/album/254486
C#	Shunting Yard	http://vimeo.com/album/210446
JAVA	Rpn Calculator	http://vimeo.com/album/205252
IPHONE	iPhone & TDD	http://vimeo.com/album/1472322

- Mocking

JAVA	Mockito	http://schuchert.wikispaces.com/Mockito.LoginServiceExample
C#	Moq	http://schuchert.wikispaces.com/Moq.Logging+In+Example+Implemented

- Other

JAVA	FitNesse	http://schuchert.wikispaces.com/FitNesse.Tutorials
RUBY	Several	http://schuchert.wikispaces.com/ruby.Tutorials
JAVA	UI	http://schuchert.wikispaces.com/tdd.Refactoring.UiExample