

# See it as you need it



# Slides and Code at...

- [https://github.com/schuchert/spring\\_aop](https://github.com/schuchert/spring_aop)
- Clone the repo:  
`git clone git://github.com/schuchert/spring_aop.git`

# 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

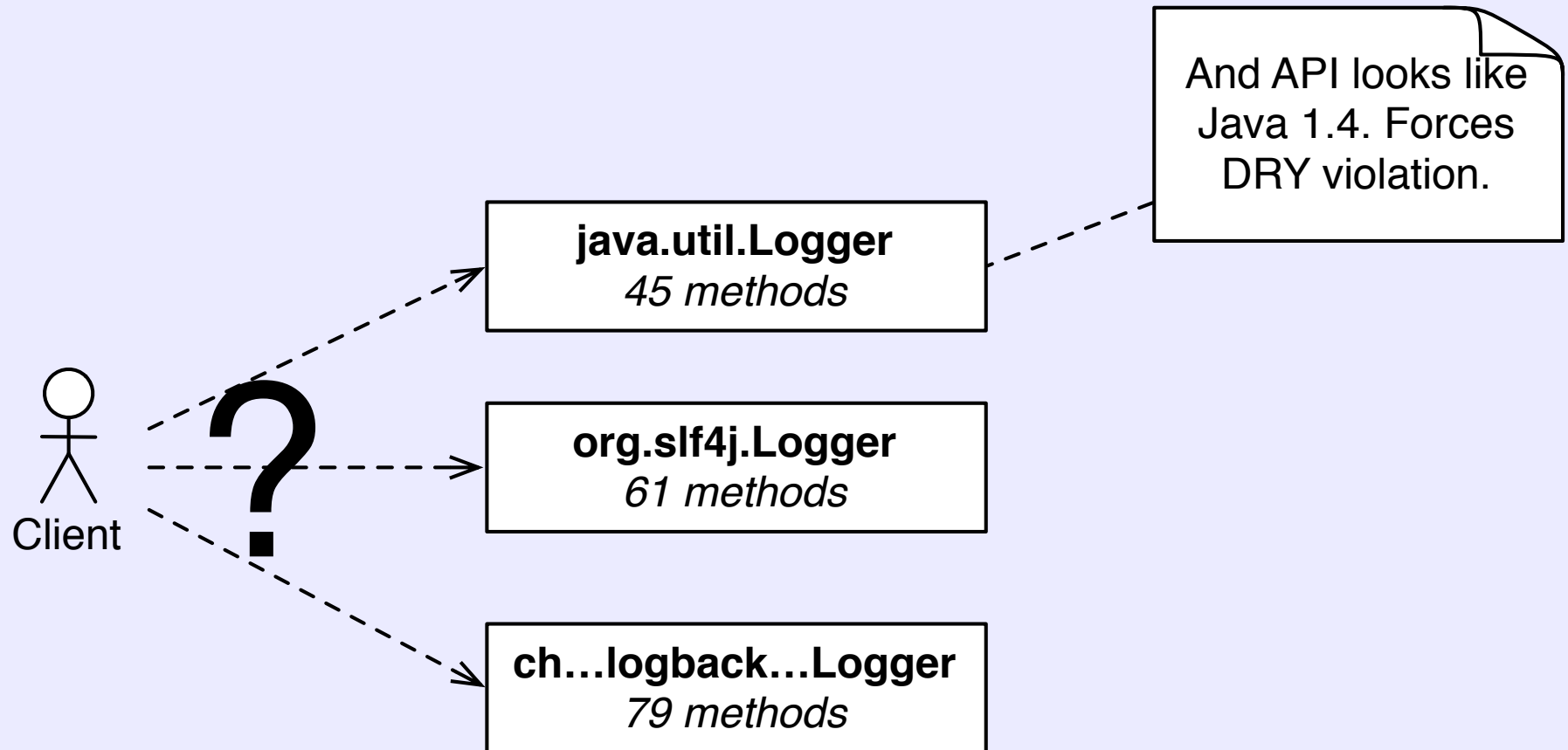
# Dependency Inversion Principle

- 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



# 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

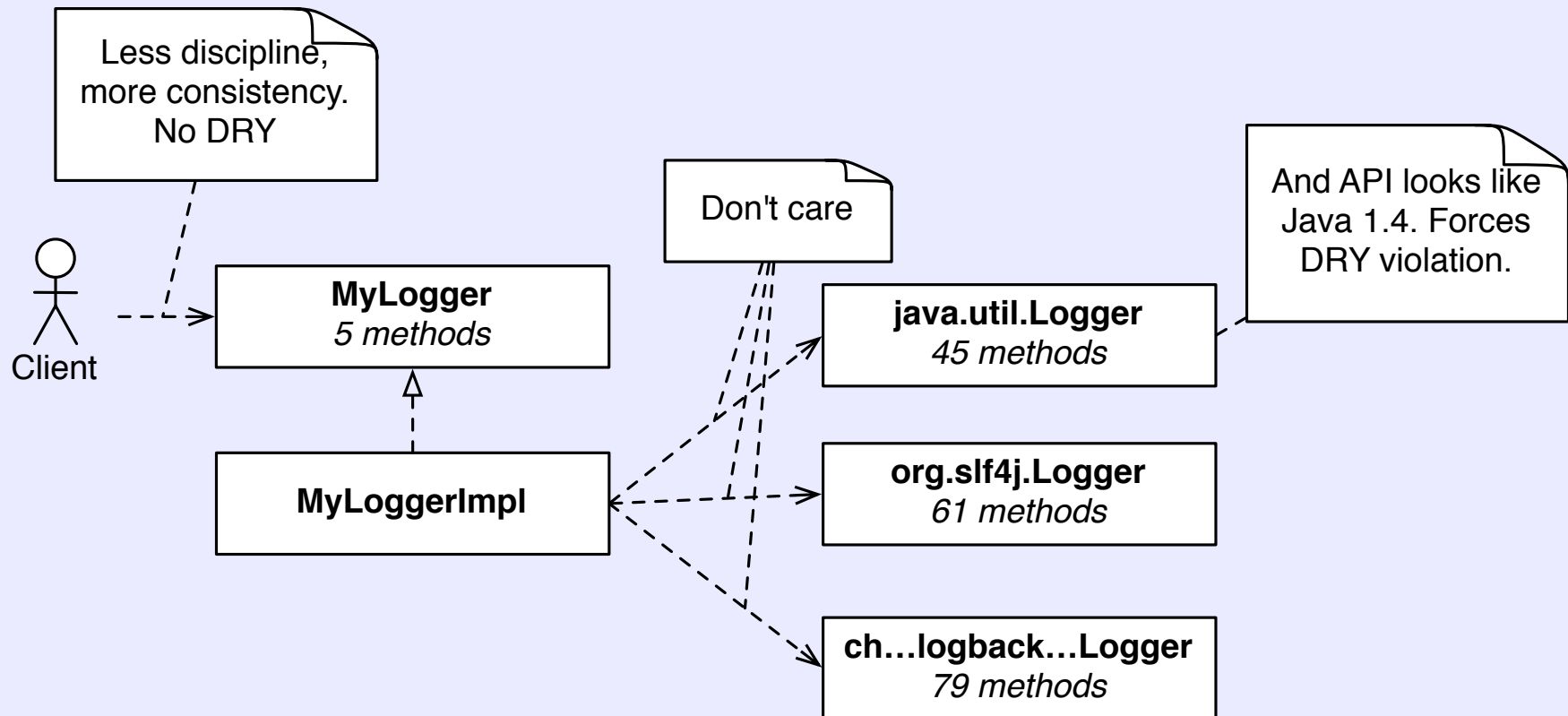
- Consider this...

```
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





# Logger

- Vs.

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

- And one such implementation...

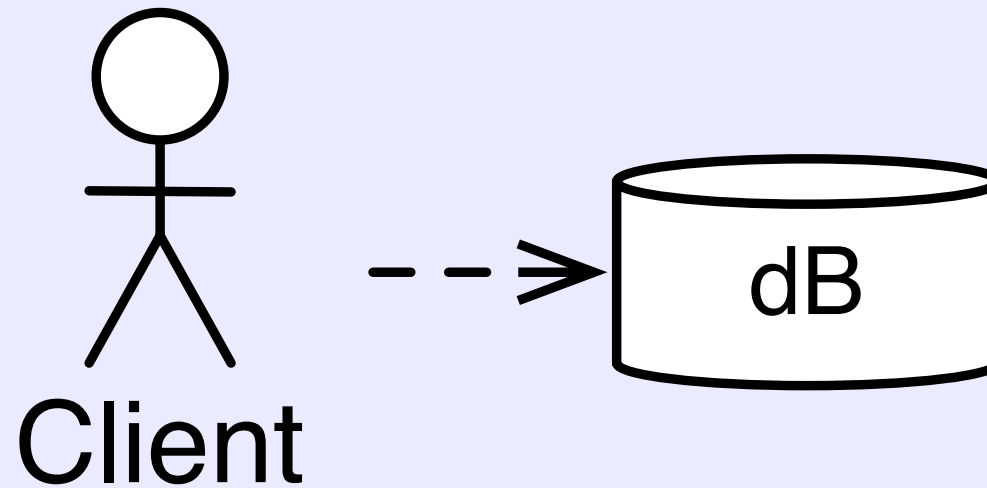
```
public void info(String message, Object... args) {  
    if (logger.isInfoEnabled()) {  
        logger.info(String.format(message, args));  
    }  
}
```

- Still passing parameters...but

- No unnecessary String concatenation
- Reduces object creation & gc

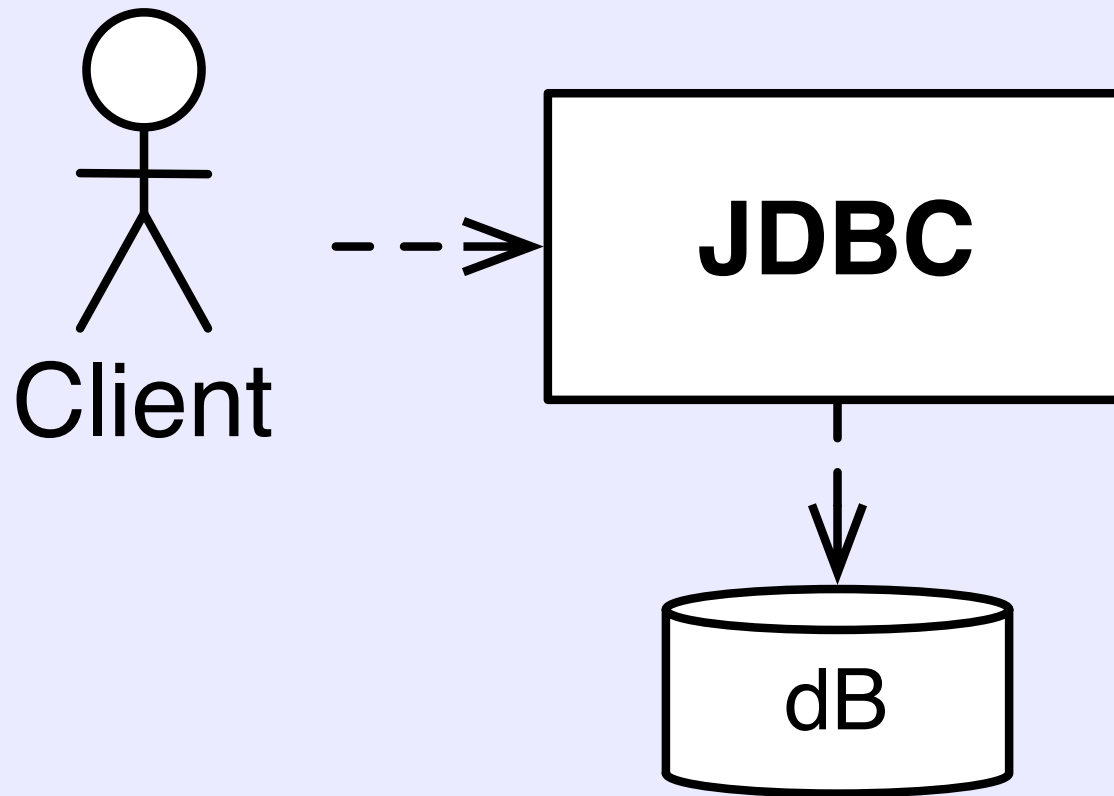
# Depending on Interfaces

- Starting with



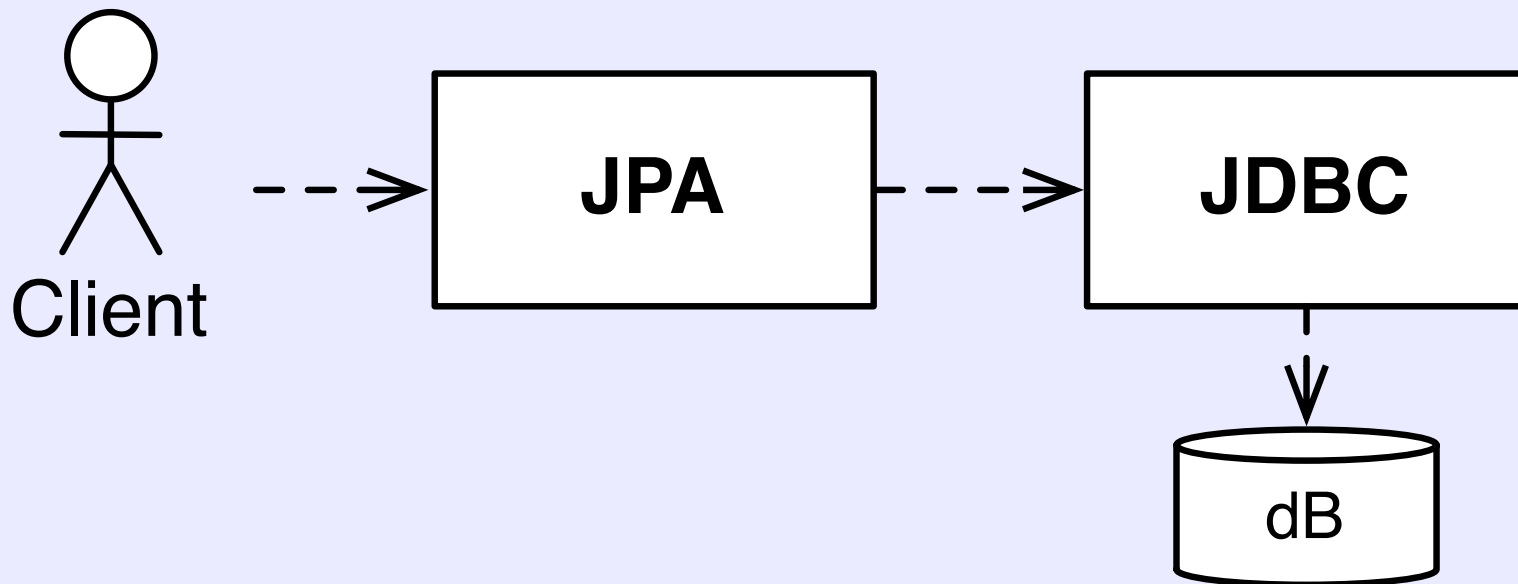
# Depending on Interfaces

- Last century we had...Better?



# Depending on Interfaces

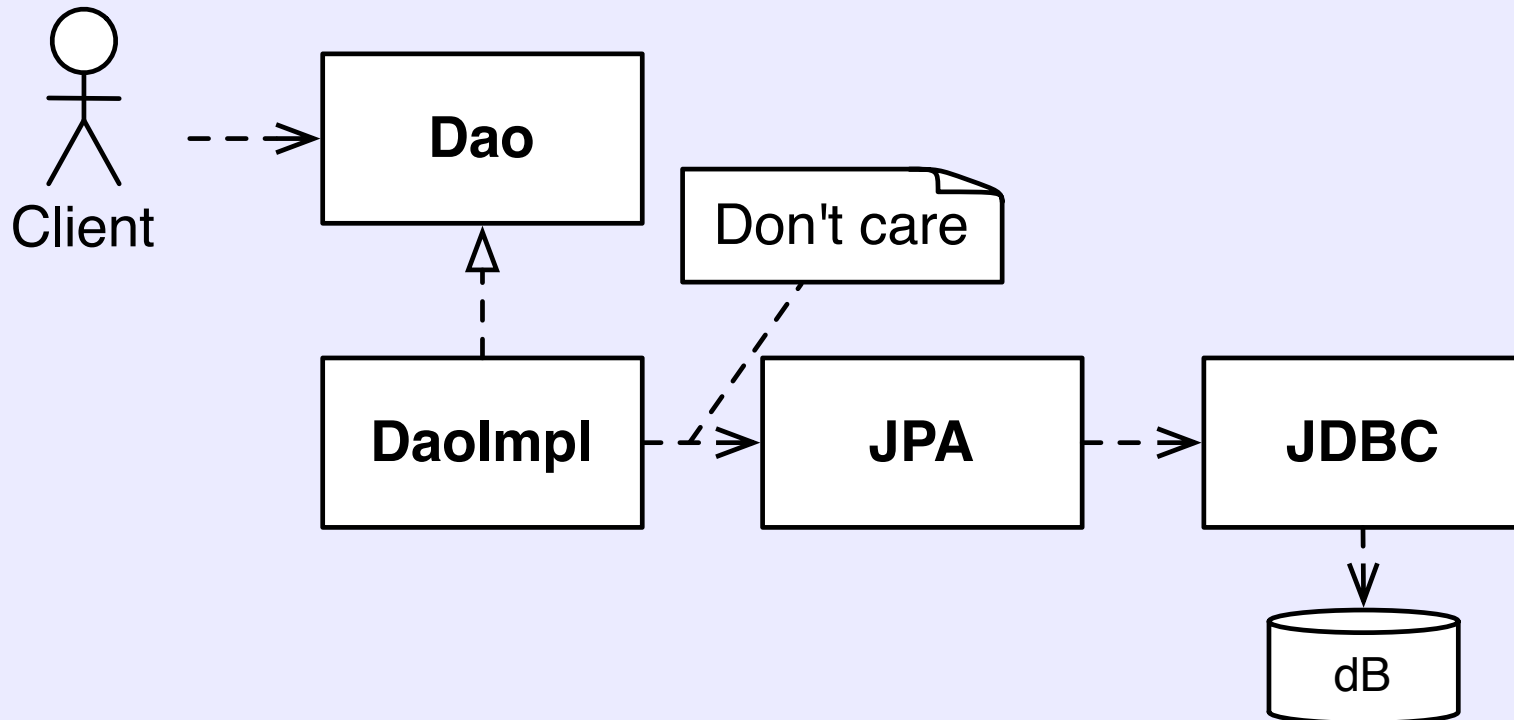
- 2005ish we had...Better?



- Hibernate came before, not essentially different.

# Depending on Interfaces

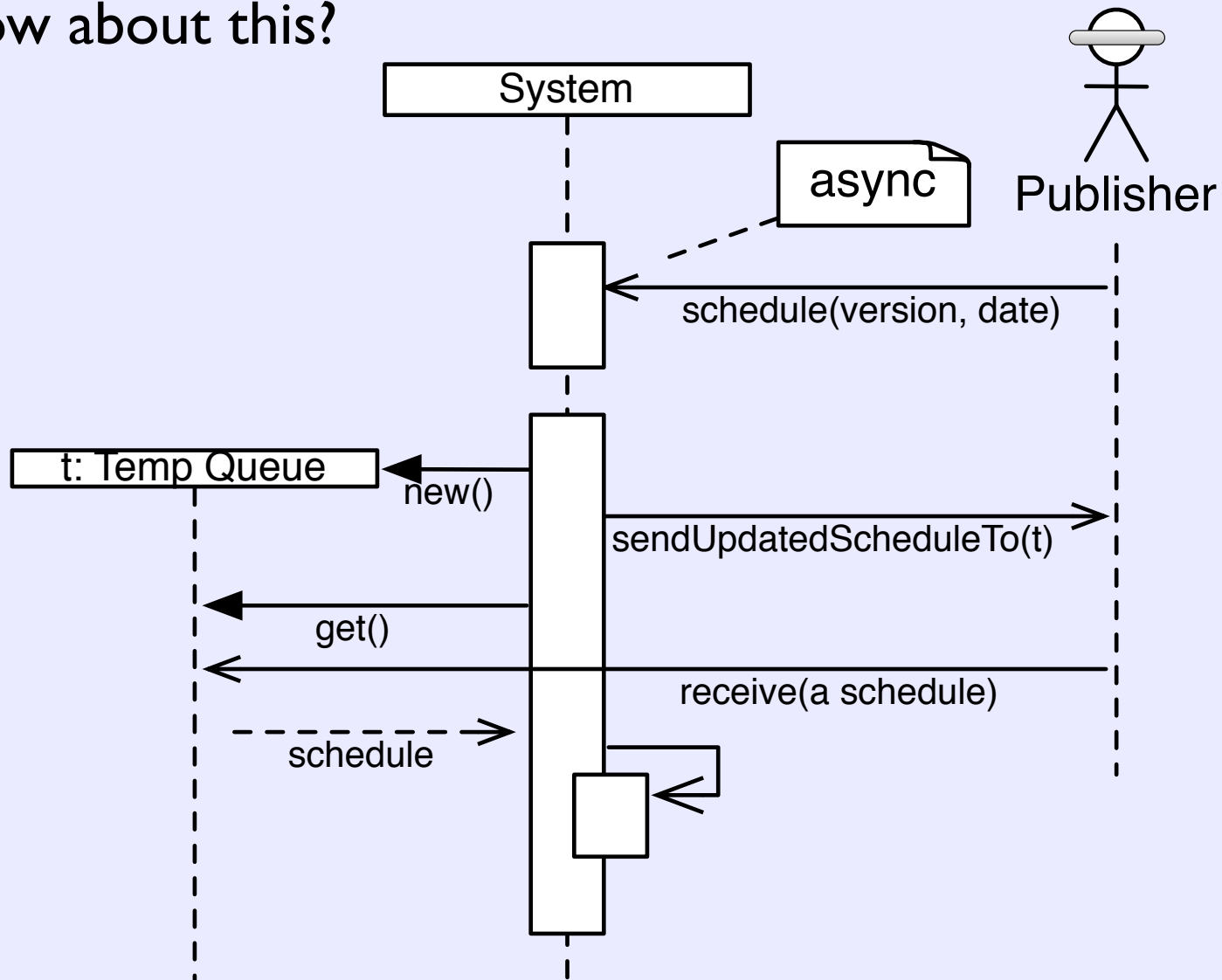
- Better?



- Modern approaches avoid DaoImpl for “normal” stuff

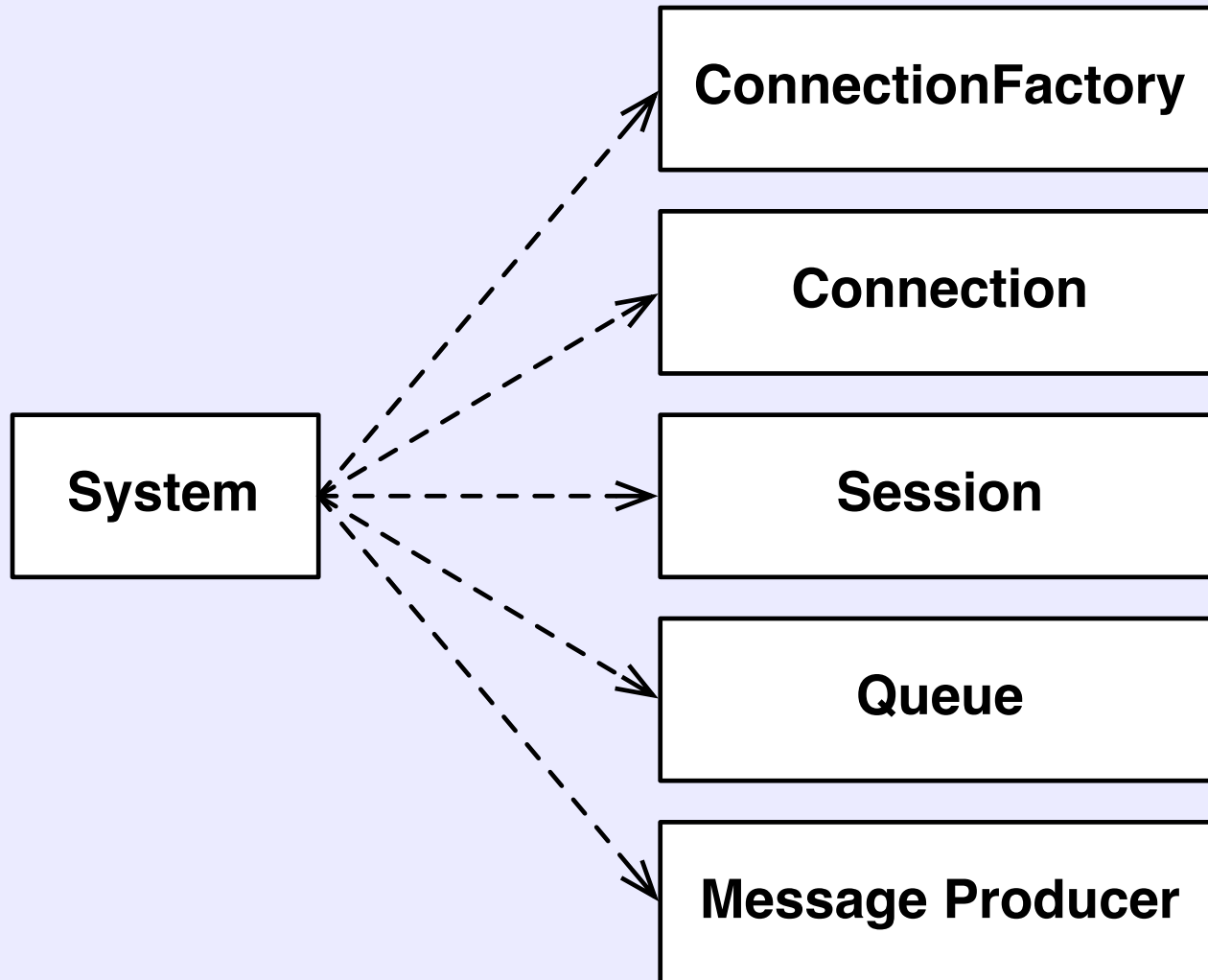
# Message Oriented Middleware

- How about this?



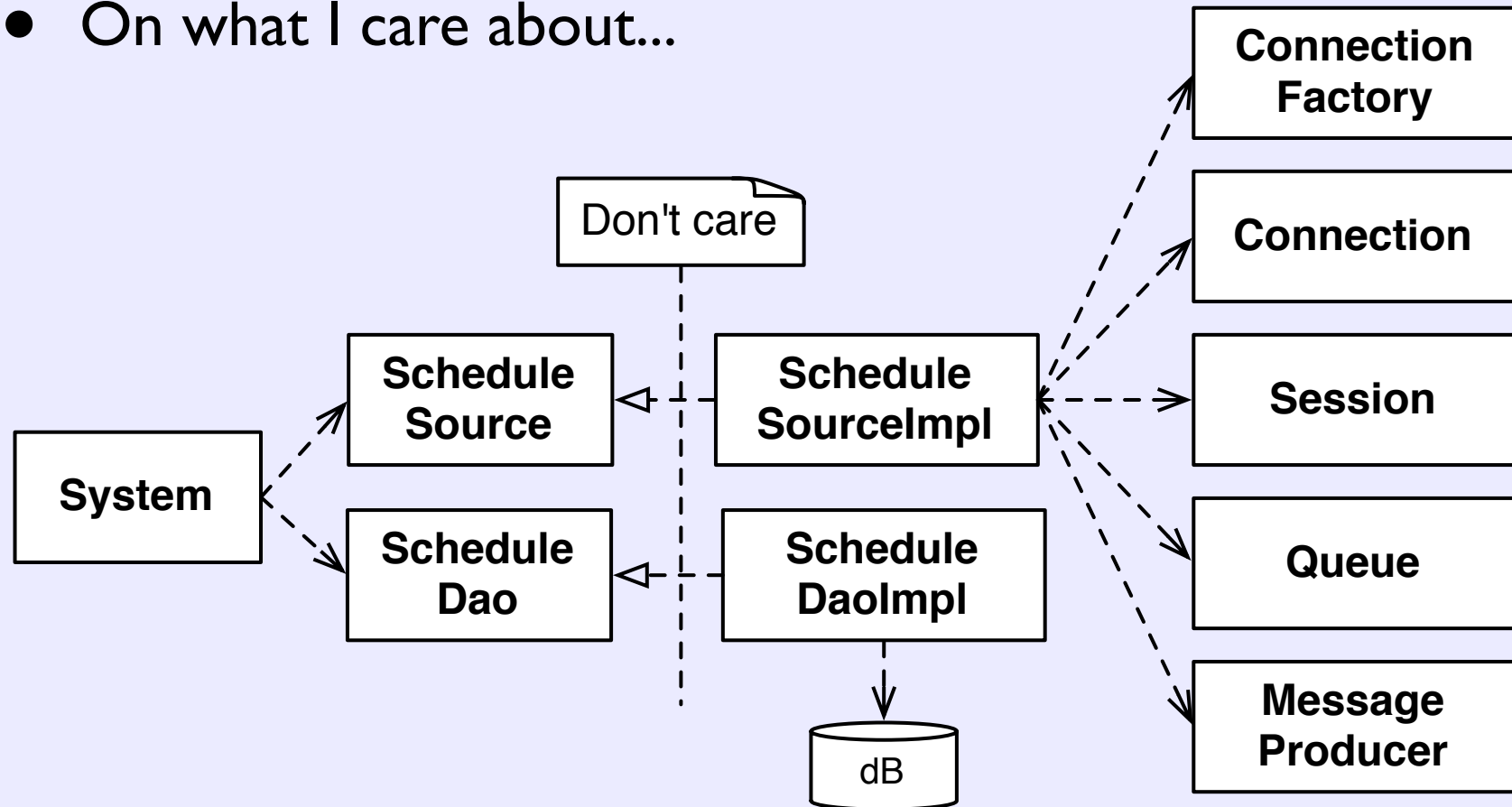
# As a system...

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



# Or I Can Depend

- On what I care about...

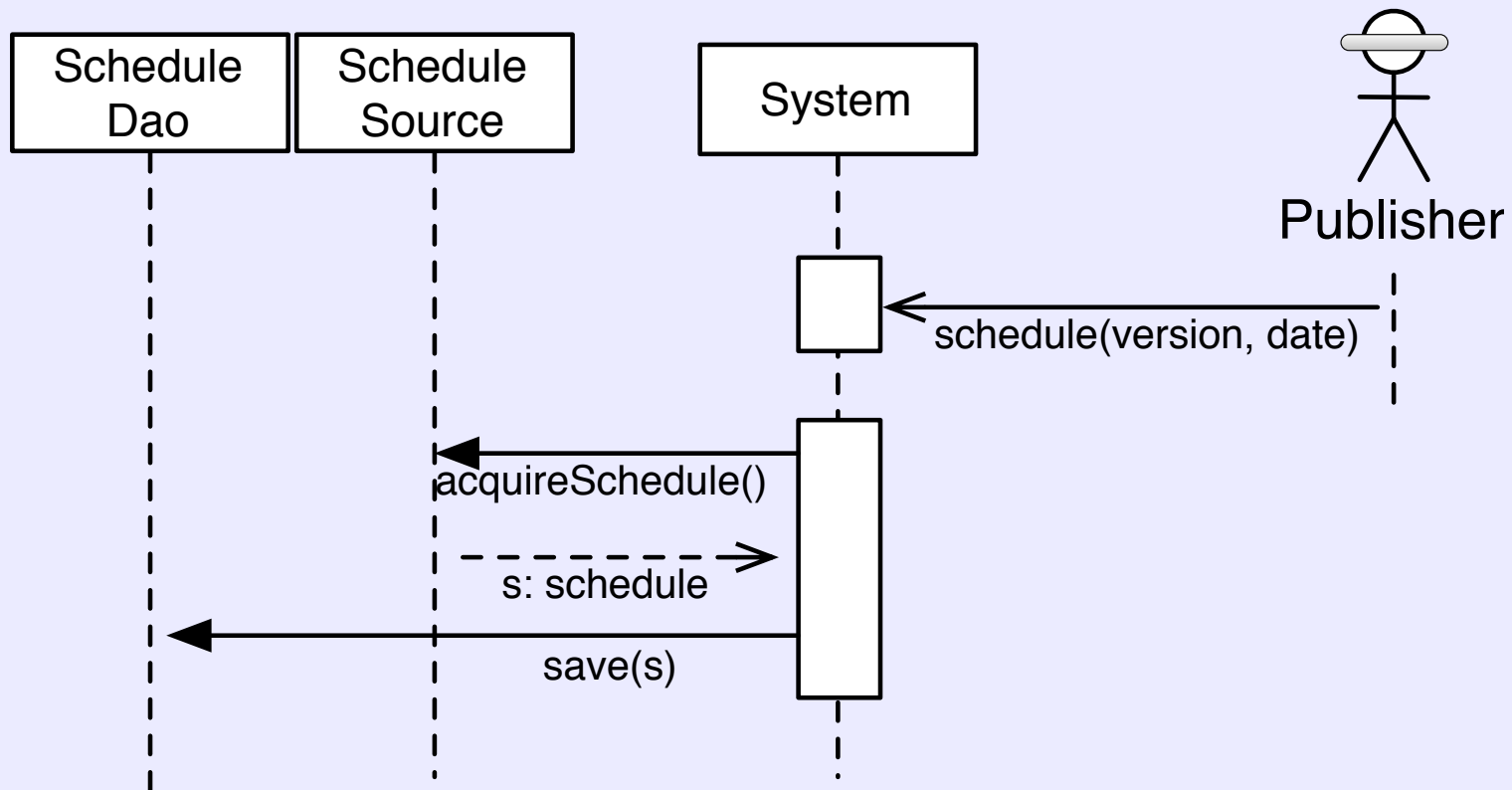


- This gave me confidence to assert (correctly) that prod and QA system queues were inconsistently configured.



# 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 entries 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 starting list to help with OOD

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

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

# GRASP



<b>INFORMATION EXPERT</b>	Assign responsibility to the thing that has the information.
<b>CONTROLLER</b>	Assign system operations (events) to a non-UI class. May be system-wide, use case driven or for a layer.
<b>LOW COUPLING</b>	Try to keep the number of connections small. Prefer coupling to stable abstractions.
<b>HIGH COHESION</b>	Keep focus. The behaviors of a thing should be related. Alternatively, clients should use all or most parts of an API.
<b>POLYMORPHISM</b>	Where there are variations in type, assign responsibility to the types (hierarchy) rather than determine behavior externally,
<b>PURE FABRICATION</b>	Create a class that does not come from the domain to assist in maintaining high cohesion and low coupling.
<b>PROTECTED VARIATIONS</b>	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>

<b>S</b>	<b>SINGLE RESPONSIBILITY</b>	Single Reason to Change
<b>O</b>	<b>OPEN/CLOSED</b>	Open for extension closed to change
<b>L</b>	<b>LISKOV SUBSTITUTION</b>	Derived types substitutable for base types
<b>I</b>	<b>INTERFACE SEGREGATION</b>	Interfaces should be focused (small) & client specific
<b>D</b>	<b>DEPENDENCY INVERSION</b>	Dependencies should go from concrete to abstract

# Package Cohesion/Coupling



- Guidelines for package cohesion

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

- Guidelines for package coupling

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



# A Few Code Smells



- A few of Martin's code smells:

<b>POOR NAMES</b>	Name suggests wrong intent
<b>LONG METHODS</b>	More than 1 thing/multiple levels of abstraction
<b>LARGE CLASSES</b>	More than one concept/multiple levels of abstraction
<b>LONG PARAMETER LIST</b>	Too many arguments to keep straight (> 3)
<b>DUPLICATED CODE</b>	Same or similar code appears in more than one place
<b>DIVERGENT CHANGE</b>	The class/method changes for dissimilar reasons
<b>SHOTGUN SURGERY</b>	Single change affects multiple classes/methods
<b>FEATURE ENVY</b>	One class uses another class' members
<b>SWITCH STATEMENTS</b>	Duplicated switches/if-else's over same criterion

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

# Some Legacy Refactorings



- From Working Effectively with Legacy Code

<b>ADAPT PARAMETER</b>	326	Change parameter to an adapter when you cannot use extract interface
<b>BREAK OUT METHOD OBJECT</b>	330	Convert method using instance data into a class with a ctor and single method
<b>ENCAPSULATE GLOBAL REFERENCES</b>	339	Move access to global data into access via a class to allow for variations during test
<b>EXTRACT AND OVERRIDE CALL</b>	348	Turn chunk of code into overridable method and then subclass in test
<b>EXTRACT AND OVERRIDE GETTER</b>	352	Turn references into hard-coded object into call to getter and then subclass
<b>EXTRACT INTERFACE</b>	362	Extract interface for concrete class, then use interface. Override in test.
<b>INTRODUCE INSTANCE DELEGATOR</b>	317	Add instance methods calling static methods. Call through instance, which test subclasses.
<b>PARAMETERIZE CONSTRUCTOR PARAMETERIZE METHOD</b>	379 383	Examples of Inversion of Control (IoC)
<b>SUBCLASS AND OVERRIDE METHOD</b>	401	Test creates subclass & passes it in/requires some IoC
<b>SPROUT METHOD SPROUT CLASS</b>	59 63	Create a method or class out of existing code.

# Test Doubles



- Gerard Meszaros

<http://xunitpatterns.com/Test%20Double%20Patterns.html>

<b>DUMMY</b>	Empty implementation. Not called or don't care if it is
<b>STUB</b>	Canned replies – “snapshot in time”
<b>SPY</b>	Watches and Records
<b>FAKE</b>	Partial Simulator
<b>MOCK</b>	Has & Validates expectations
<b>SABOTEUR</b>	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>

<b>F</b>	<b>FAST</b>	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.
<b>I</b>	<b>ISOLATED INDEPENDENT</b>	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.
<b>R</b>	<b>RELIABLE REPEATABLE</b>	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.
<b>S</b>	<b>SMALL</b>	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.
<b>T</b>	<b>TIMELY</b>	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

<b>STRATEGY</b>	Define a function or algorithm as a class. Form a wide but shallow hierarchy of different algorithms.
<b>TEMPLATE METHOD</b>	Write an algorithm in a base class with extension points represented as abstract methods. Subclass and override.
<b>ABSTRACT FACTORY</b>	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.
<b>COMPOSITE</b>	A class that implements some other interface and also holds onto zero or more instances of that same interface.
<b>STATE</b>	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

<b>C++</b>	Dice Game	<a href="http://vimeo.com/album/254486">http://vimeo.com/album/254486</a>
<b>C#</b>	Shunting Yard	<a href="http://vimeo.com/album/210446">http://vimeo.com/album/210446</a>
<b>JAVA</b>	Rpn Calculator	<a href="http://vimeo.com/album/205252">http://vimeo.com/album/205252</a>
<b>IPHONE</b>	iPhone & TDD	<a href="http://vimeo.com/album/1472322">http://vimeo.com/album/1472322</a>

- Mocking

<b>JAVA</b>	Mockito	<a href="http://schuchert.wikispaces.com/Mockito.LoginServiceExample">http://schuchert.wikispaces.com/Mockito.LoginServiceExample</a>
<b>C#</b>	Moq	<a href="http://schuchert.wikispaces.com/Moq.Logging+In+Example+Implemented">http://schuchert.wikispaces.com/Moq.Logging+In+Example+Implemented</a>

- Other

<b>JAVA</b>	FitNesse	<a href="http://schuchert.wikispaces.com/FitNesse.Tutorials">http://schuchert.wikispaces.com/FitNesse.Tutorials</a>
<b>RUBY</b>	Several	<a href="http://schuchert.wikispaces.com/ruby.Tutorials">http://schuchert.wikispaces.com/ruby.Tutorials</a>
<b>JAVA</b>	UI	<a href="http://schuchert.wikispaces.com/tdd.Refactoring.UiExample">http://schuchert.wikispaces.com/tdd.Refactoring.UiExample</a>