Qualifying Exam

Nicholas Chen
nchen@uiuc.edu

My Criteria for Tool Evaluation

- The tools should not distract the developer with information overload.
- The tools must be adaptive and work as the software being developed on evolves.
- The tools should be non-intrusive; the developer should be able to use the tools with minimal changes to his existing software artifacts.
- The tools should not force the developer to use a new unfamiliar environment but should work with existing tools that the developer is familiar with.

Automatic Test Factoring for Java

David Saff, Shay Artzi, Jeff H. Perkins
and Michael D. Ernst

ASE '05

What's the paper about?

- System tests help check that system requirements are met **but** they take a *long time* to run.

- So they aren't run frequently and developers don't get the benefit of *rapid feedback* when something goes wrong.

- Mock objects are used to reduce the time of running tests; this paper presents a way to *automatically factor* focused tests by introducing mock objects.

- By promoting *rapid feedback*, the developer is able quickly fix detected errors before they grow in seriousness.

The Importance of Rapid Feedback

- *Reducing Wasted Development Time via Continuous Testing*
  - Early detection of errors saves time overall
  - Problems left unfixed tend to become more obscure

Found that reducing time between error *introduction* and *discovery* improves overall development time

Agenda

1. **Motivation and Basics of Mocking**
2. Test Factoring Technique
3. Test Factoring Implementation
4. Improving Test Factoring

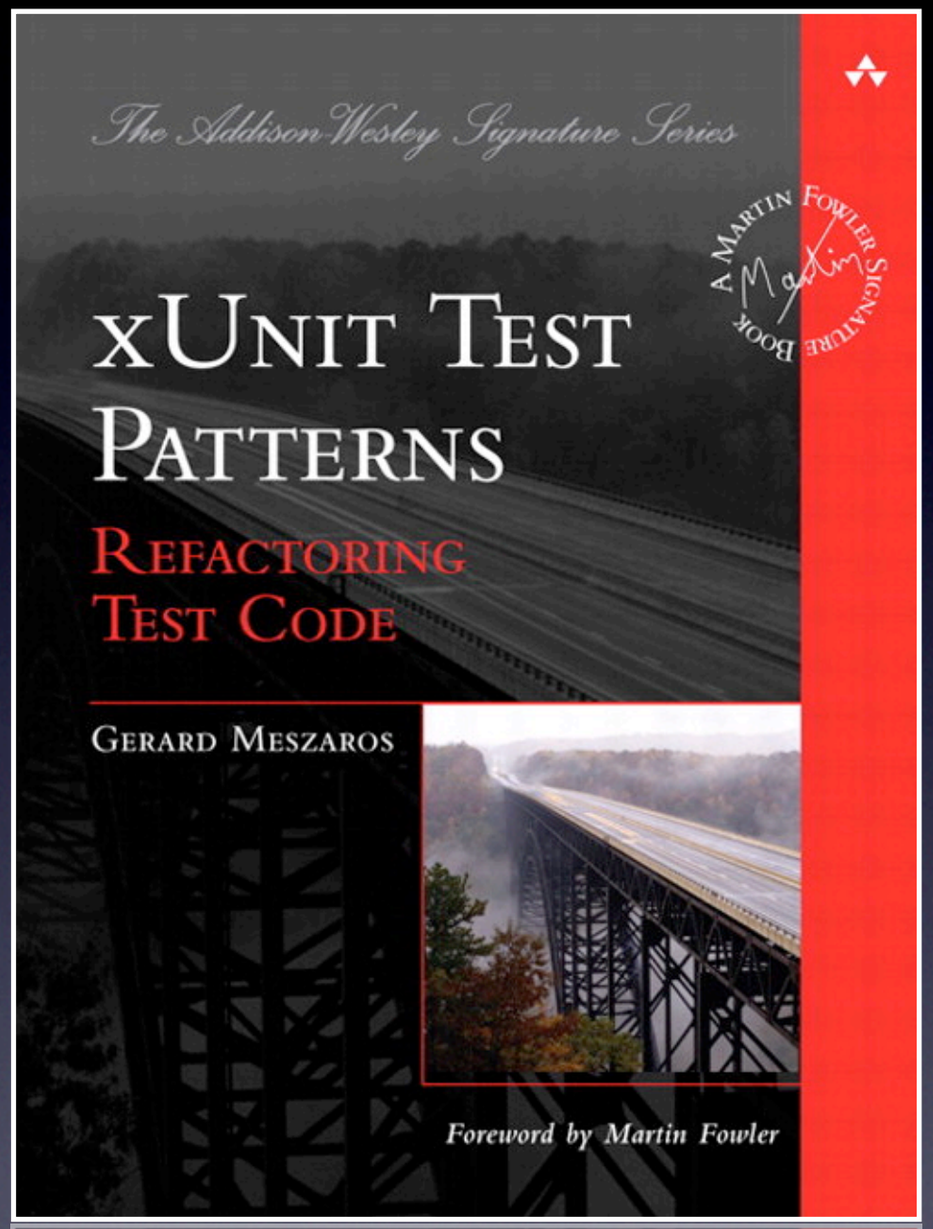
Benefits of Test-Driven Development

- Red / Green / Refactor mantra
- Effective for *unit-tests* that are focused on *small* parts of the entire system
- Still need system tests
- "System tests [however] are easier to create and understand"

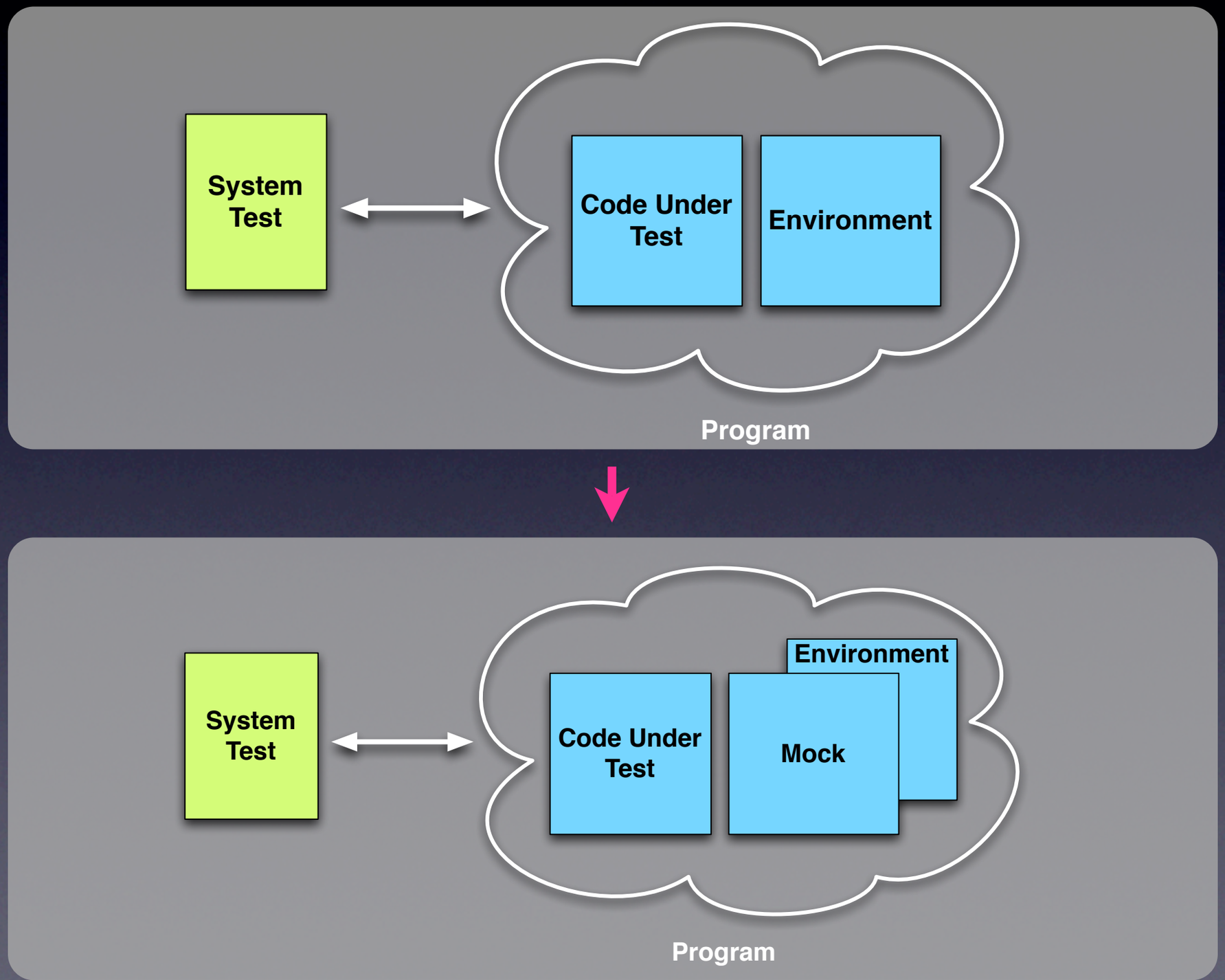
Want the same benefit of focused tests from system tests

"Slow tests" is *one* Test Smell

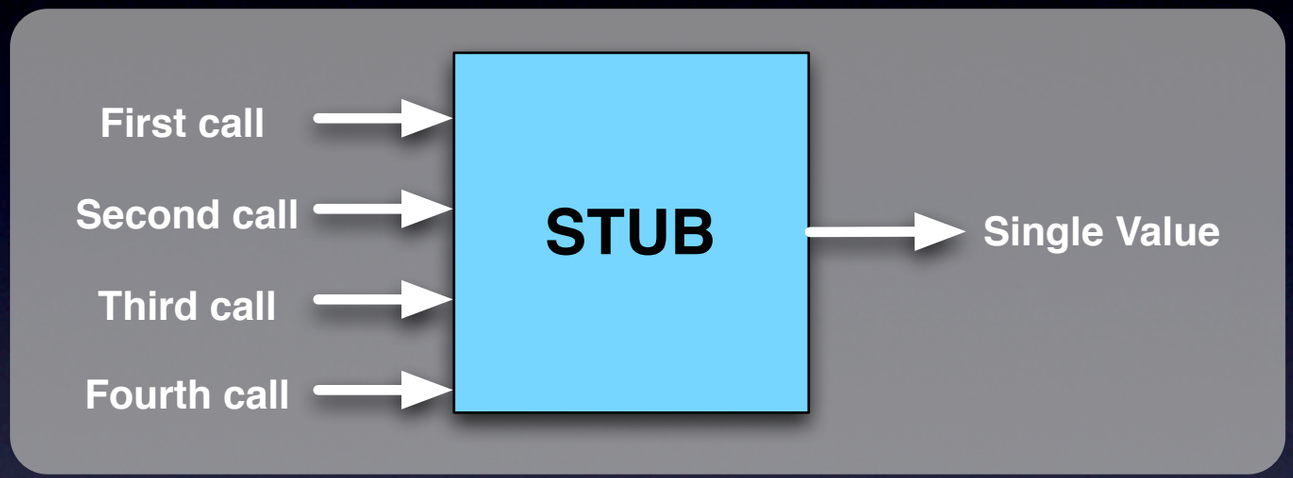
- If tests take too long, developers don't run them frequently
- If tests take too long, developers waste time waiting
- Slow tests are an *integration bottleneck*



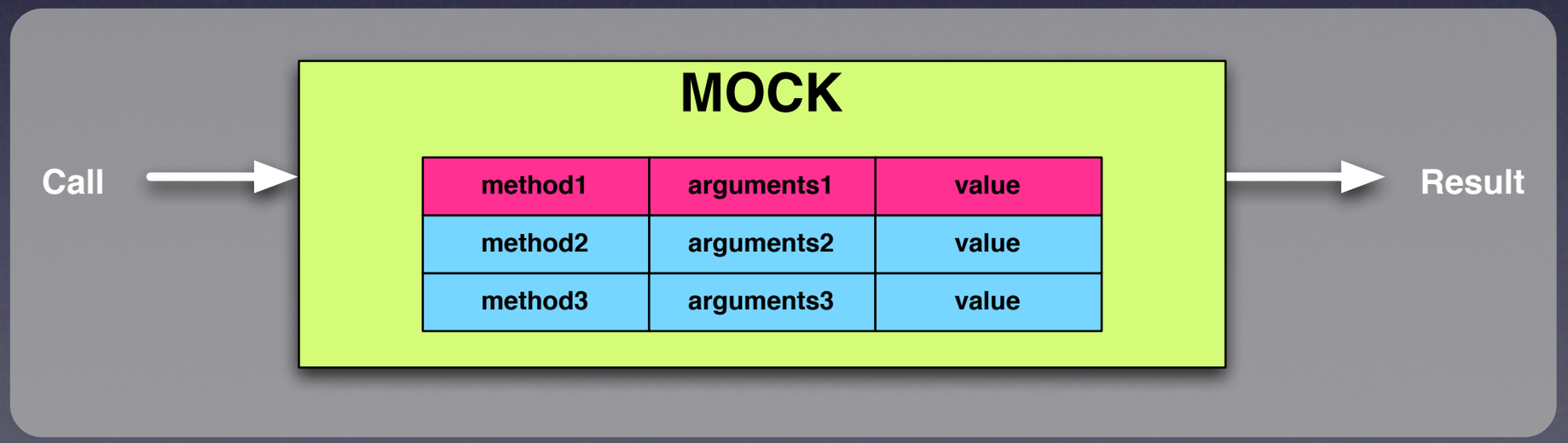
Mock Example



Difference between *stubs* and *mocks*



A stub produces a value given calls



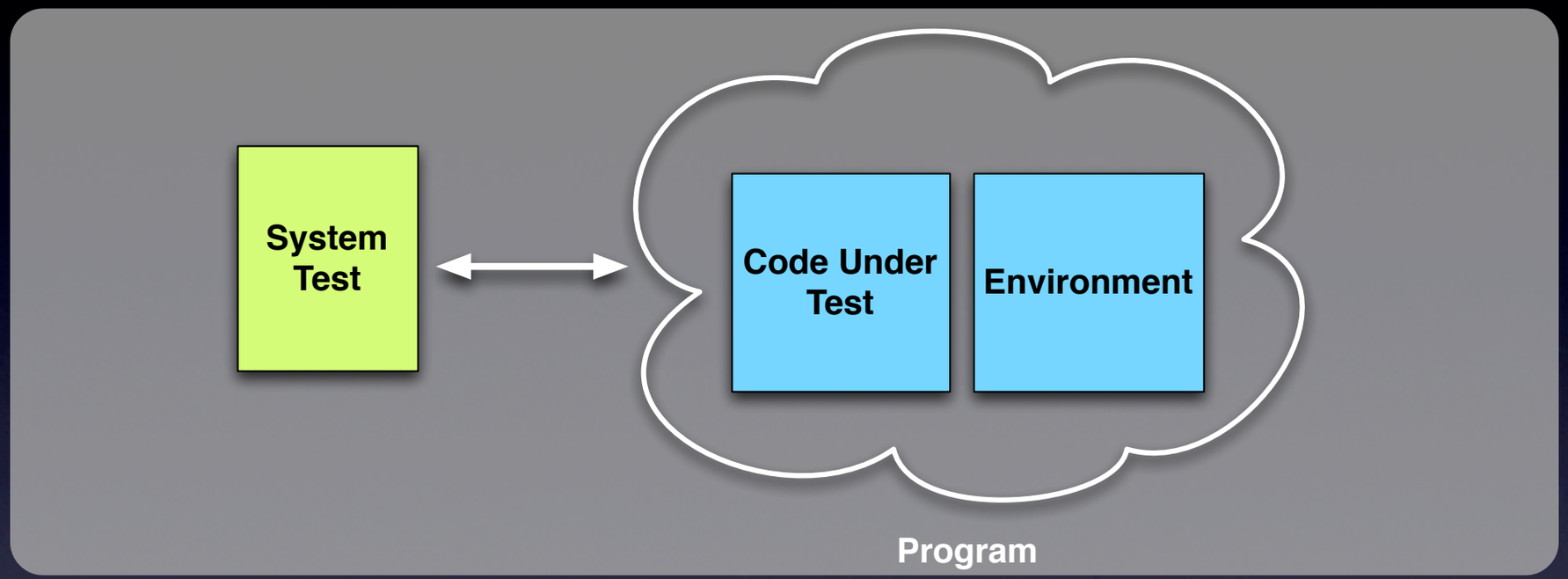
A mock produces a value given calls *after* checking its internal state

Manual Mocking

```
public void testReloadsCachedObjectAfterTimeout() {
    // Notice how this actually resembles the MockExpectation table
    // It's like filling the table entries manually
    mockClock.expects(times(3)).method("getCurrentTime").withNoArguments()
            .will(returnValues(loadTime, fetchTime, reloadTime));
    mockLoader.expects(times(2))
            .method("load").with(eq(KEY))
            .will(returnValues(VALUE, NEW_VALUE));
    mockReloadPolicy.expects(atLeastOnce())
            .method("shouldReload").with(eq(loadTime), eq(fetchTime))
            .will(returnValue(true));
    // Here we "replay" the values from our "table"
    assertSame("should be loaded object", VALUE, cache.lookup(KEY));
    assertSame("should be reloaded object", NEW_VALUE, cache.lookup(KEY));
}
```

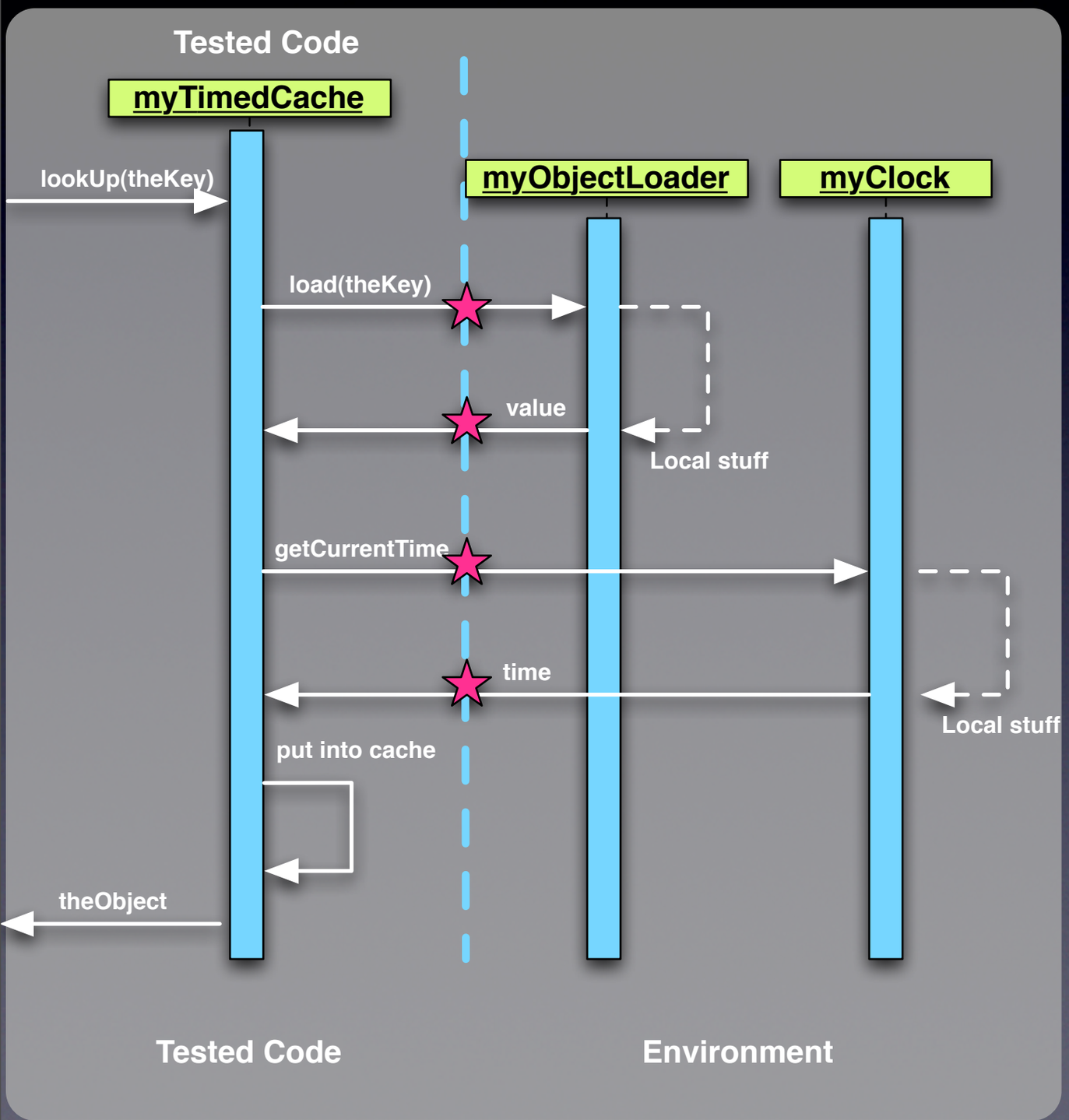
1. Motivation and Basics of Mocking
2. **Test Factoring Technique**
3. Test Factoring Approaches
4. Improving Test Factoring

The formula



Capture, Factor and Replay

Capturing

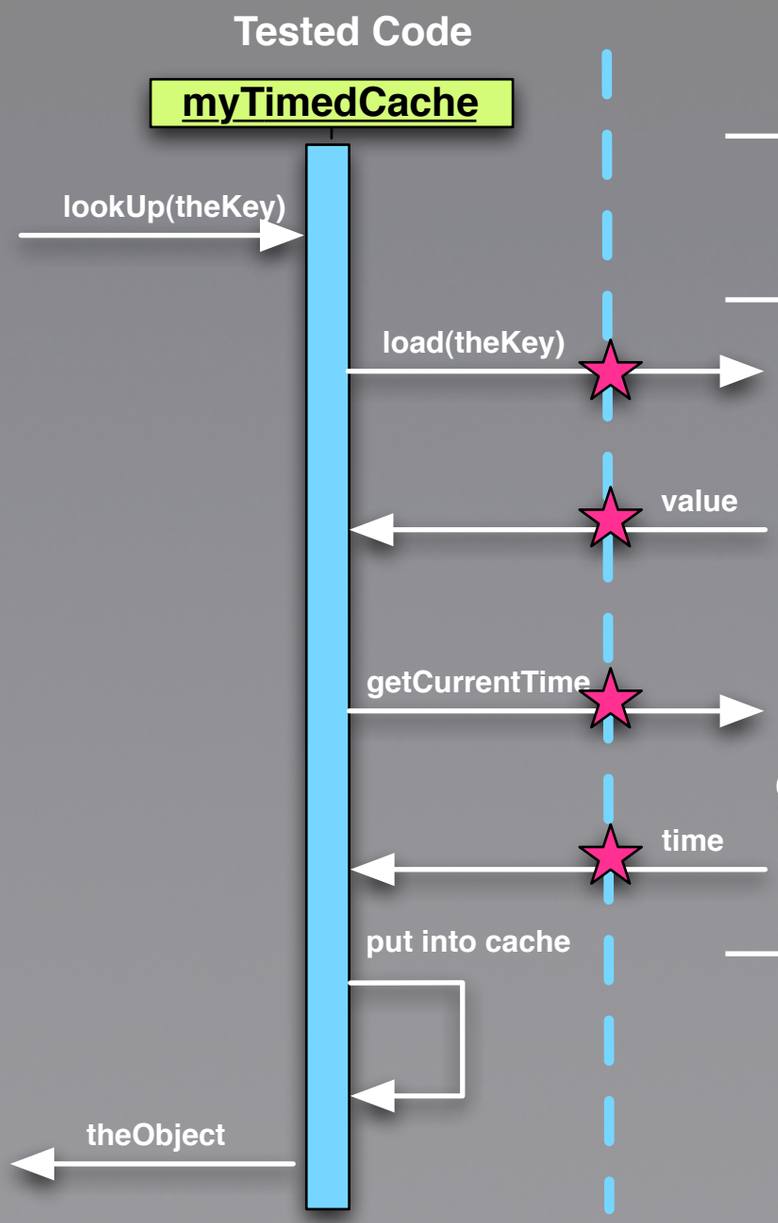



```
myTimedCache.lookup("key1")
```

MockExpectations Table

Method	Arguments	Value
load	theKey	value
getCurrentTime	void	time

Replay



MockExpectations Table

Method	Arguments	Value
load	theKey	value
getCurrentTime	void	time

1. Motivation and Basics of Mocking
2. Test Factoring Technique
3. **Test Factoring Approaches**
4. Improving Test Factoring

Custom Instrumentation

Instrumentation is the addition of byte-codes to methods for the purpose of gathering data to be utilized by tools

- Create a proxy that will intercept the relevant calls
- Transform field access to method calls for uniformity
- Introduce a new interface for class; use references to these new interfaces whenever possible
  - JDK classes instrumented beforehand; JVM modified
- The actual work is done by a *delegate* and the interaction is captured into a table

Why Custom Instrumentation?

- Most Mock-ing frameworks use java.lang.reflect.Proxy for dynamic proxies
  - can only mock interfaces and *not* classes
  - cannot handle *static* method calls
  - cannot handle *final* classes and *private* methods

Why Custom Instrumentation?

- So, *unfortunately*, normal Mock-ing frameworks
  - cannot handle legacy code
  - require you to design for testing in the first place
  - doesn't work for all cases (reflection, native methods, etc)

Do we need to support *everything* to make test factoring useful?

Why not AOP?

- Complicated process for tracing
- Had to use *non-standard* JVM so why don't just use AOP? AOP's poster child is *tracing*
- AOP and tracing JDK classes
- AOP and performance
- Instrumentation experience on the team

Twin Class Hierarchy Comparison

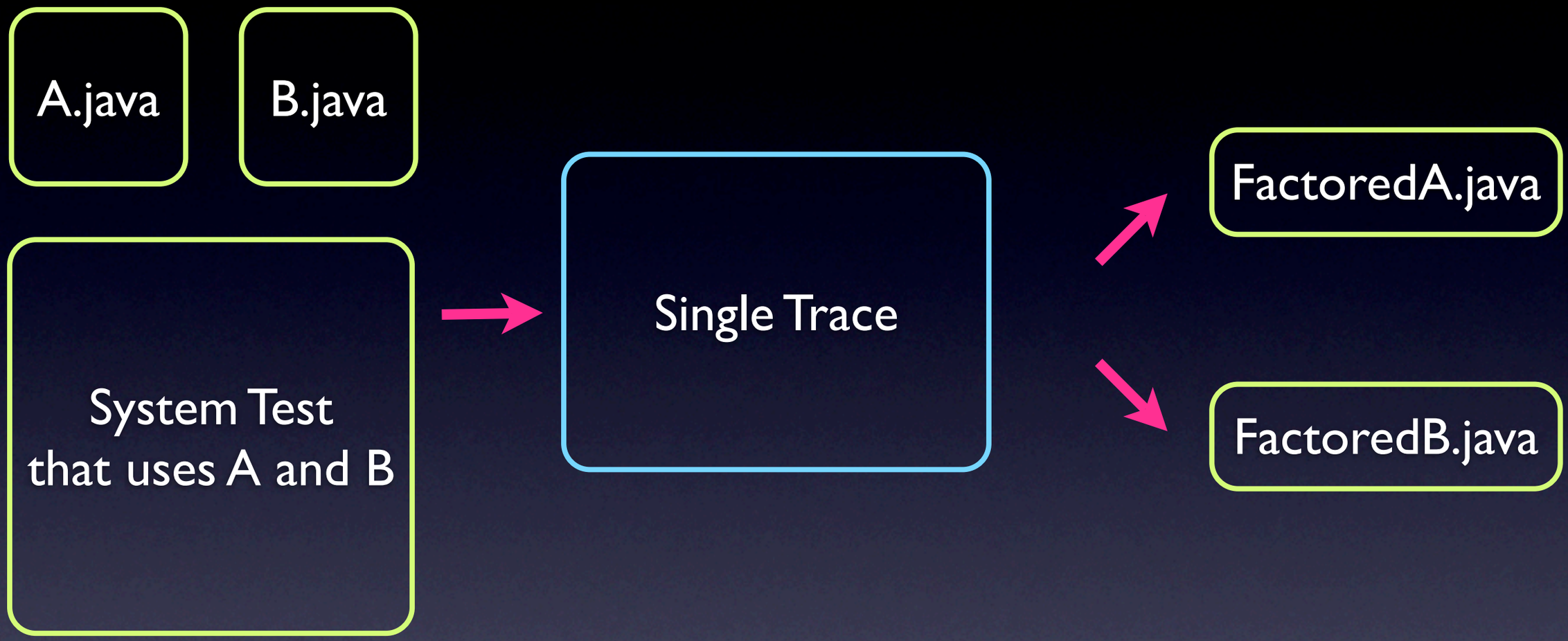
- "Wrappers must be written by hand for each native method..."
  - They can be instrumented the same way as the capturing version of classes
- "Of which there are a great many..."
  - 3% of the system classes in Java are native

Twin Class Hierarchy strategy has *multiple* benchmarks which shows that performance was fine

Partition Problems

- Heuristic: "*choose the **class** containing main routine as environment, the changed **classes** as code under test and all other **classes** as the common libraries*"
- For the same system test, need different runs for different classes
- Every call to a class is captured – "*typical run processes 1 GB of trace data....*"
- Capturing occurs only once at *night* but transcripts expected to be useful *all day*

amock



- Doesn't handle JNI, instrumenting JDK, full reflection
- Generates human readable/editable tests with JMock

Partition Problems

- Heuristic: "*choose the **class** containing main routine as environment, the changed **classes** as code under test and all other **classes** as the common libraries*"
- For the same system test, need different runs for different classes
- Every call to a class is captured – "*typical run processes 1 GB of trace data....*"
- Capturing occurs only once at *night* but transcripts expected to be useful *all day*

Real Results

- Experiment done with *one* project and *two* developers
- Time to failure actually *increased*
- Was the result reproducible in other systems? No implementation was released so hard to experiment
- Was this system specially tuned to handle Daikon?

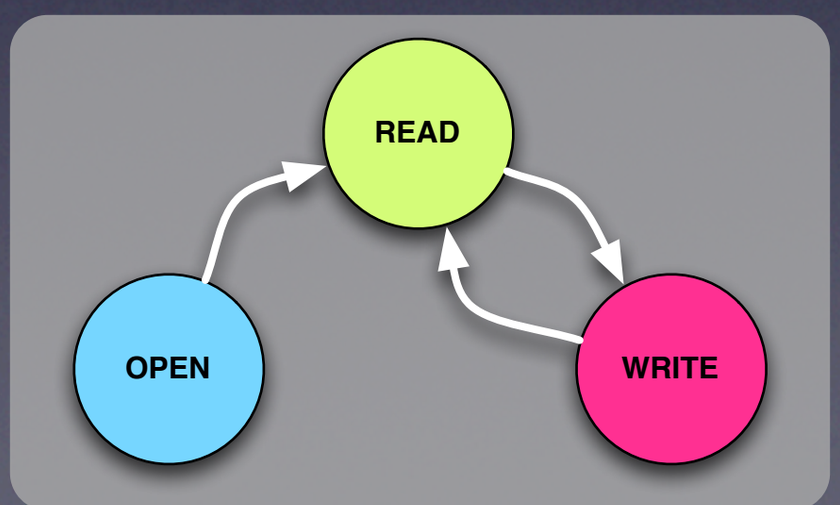
1. Motivation and Basics of Mocking
2. Test Factoring Technique
3. Novelties of Test Factoring
4. **Improving Test Factoring**

MockExpectations Table Longevity

- Capture *intent* of changes in a *change language*
- Permit reordering of calls to independent objects *possibly with* ***human intervention***

Using the MockExpectations table

- Since the calls to/from the environment are already captured, we could use the MockExpectations table to check that certain calls are in order
- This is called *behavior verification*



Integration with Mylyn

- *Test prioritization* and *test selection* are desired improvements
- Eclipse already has *Mylyn* that tracks user focus on current *tasks* and stores them in a *context*
- Correlate edited classes and tests for test selection and prioritization

My Evaluation of Test Factoring

- **The tools should not distract the developer with information overload.**
- **The tools must be adaptive and work as the software being developed on evolves.**
- *The tools should be non-intrusive; the developer should be able to use the tools with minimal changes to his existing software artifacts.*
- The tools should not force the developer to use a new unfamiliar environment but should work with existing tools that the developer is familiar with.

Conclusion

Automatic Test Factoring takes advantage of existing system tests by factoring the main parts into *less expensive* tests that developers can run frequently to verify functionality.

Questions

Qualifying Exam

Nicholas Chen
nchen@uiuc.edu

Appendices

Mock Example - Code Under Test

```
public class TimedCache {
    // ObjectLoader, Clock and ReloadPolicy are INTERFACES
    private ObjectLoader loader;
    private Clock        clock;
    private ReloadPolicy reloadPolicy;
    private HashMap      cachedValues = new HashMap();

    private class TimestampedValue {
        // TimeStamp is an INTERFACE
        public final Timestamp loadTime;
        public final Object    value;

        public TimestampedValue(final Object value, final Timestamp timestamp)
    }

    public TimedCache(ObjectLoader loader, Clock clock, ReloadPolicy reloadPolicy)

    public Object lookup(Object theKey)

    private TimestampedValue loadObject(Object theKey)

    public void putValue(Object key, Object value, Timestamp loadTime)
}
```

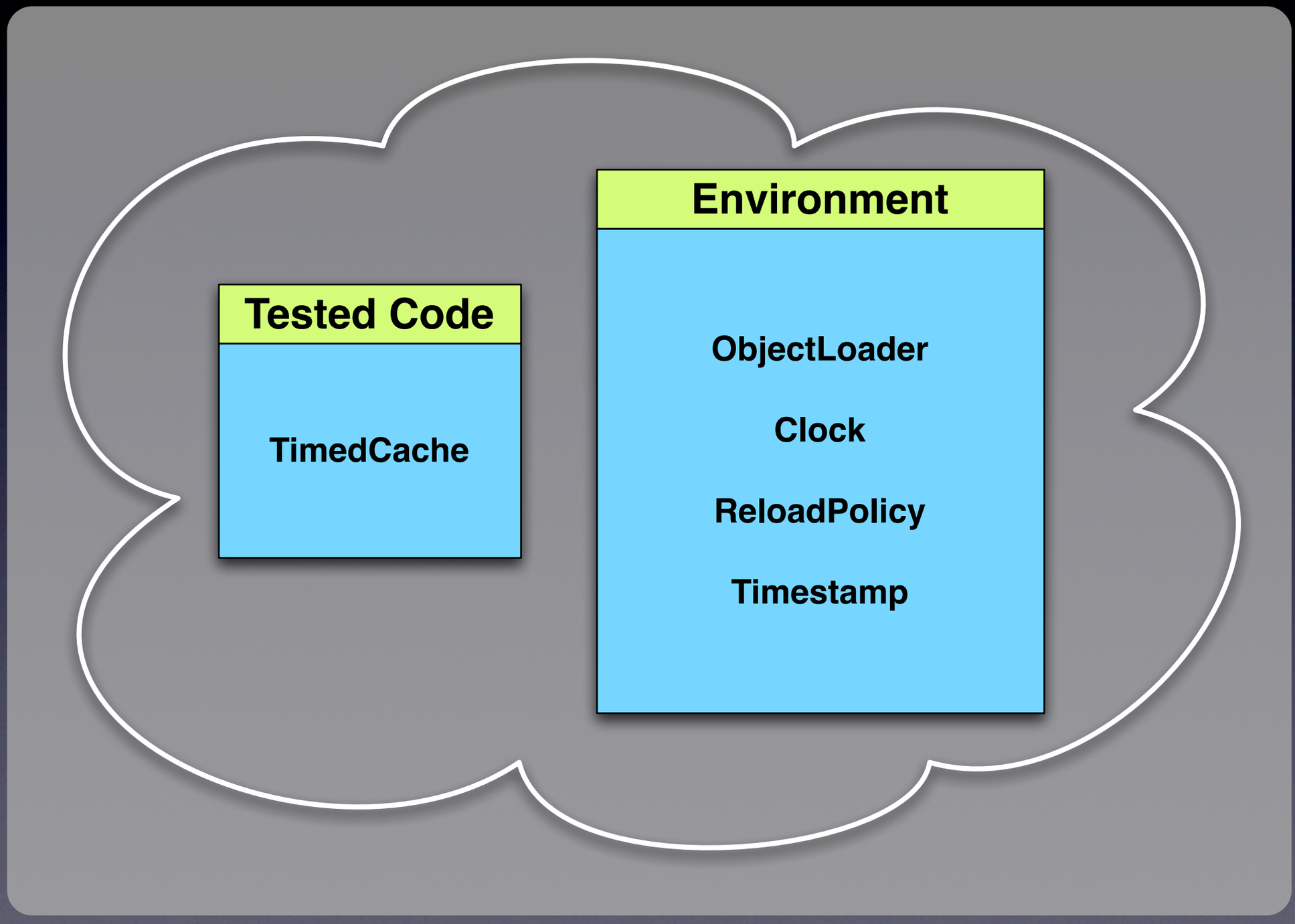
Mock Example - Setting up mocks

```
// Create MOCK OBJECTS
private Mock mockClock = mock(Clock.class);
private Mock mockLoader = mock(ObjectLoader.class);
private Mock mockReloadPolicy = mock(ReloadPolicy.class);
private TimedCache cache =
        new TimedCache((ObjectLoader)mockLoader.proxy(),
                       (Clock)mockClock.proxy(),
                       (ReloadPolicy)mockReloadPolicy.proxy());

// Create DUMMY OBJECTS
final private Object KEY = newDummy("key");
final private Object VALUE = newDummy("value");
final private Object NEW_VALUE = newDummy("newValue");

private Timestamp loadTime =
(Timestamp)newDummy(Timestamp.class, "loadTime");
private Timestamp fetchTime =
(Timestamp)newDummy(Timestamp.class, "fetchTime");
private Timestamp reloadTime =
(Timestamp)newDummy(Timestamp.class, "reloadTime");
```

Mock Example - Testing with Mocks

```
public void testLoadsObjectThatIsNotCached() {
    // Notice how this actually resembles the MockExpectation table
    // It's like filling the table entries manually
    mockLoader.expects(once()).method("load").with(eq("key1"))
            .will(returnValue("value1"));
    mockLoader.expects(once()).method("load").with(eq("key2"))
            .will(returnValue("value2"));
    mockClock.expects(atLeastOnce()).method("getCurrentTime")
            .withNoArguments().will(returnValue(loadTime));

    // Here we "replay" the values from our "table"
    assertSame("first object", "value1", cache.lookup("key1"));
    assertSame("second object", "value2", cache.lookup("key2"));
}
```

Partitioning



Dynamic Proxies in Java

```
// Foo is an interface;
// handler is an InvocationHandler
Foo f = (Foo) Proxy.newProxyInstance(
                    Foo.class.getClassLoader(),
                    new Class[] { Foo.class },
                    handler);
```

- Proxy classes are public, final, and not abstract.
- "$Proxy" is prepended to dynamic proxies
- The handler has an invoke(...) method that is called in the proxy

Twin Class Hierarchy