Mock Objects and Distributed Testing

Making a Mockery of your Software

Brian Gilstrap

"Once," said the Mock Turtle at last, with a deep sigh,
"I was a real Turtle."

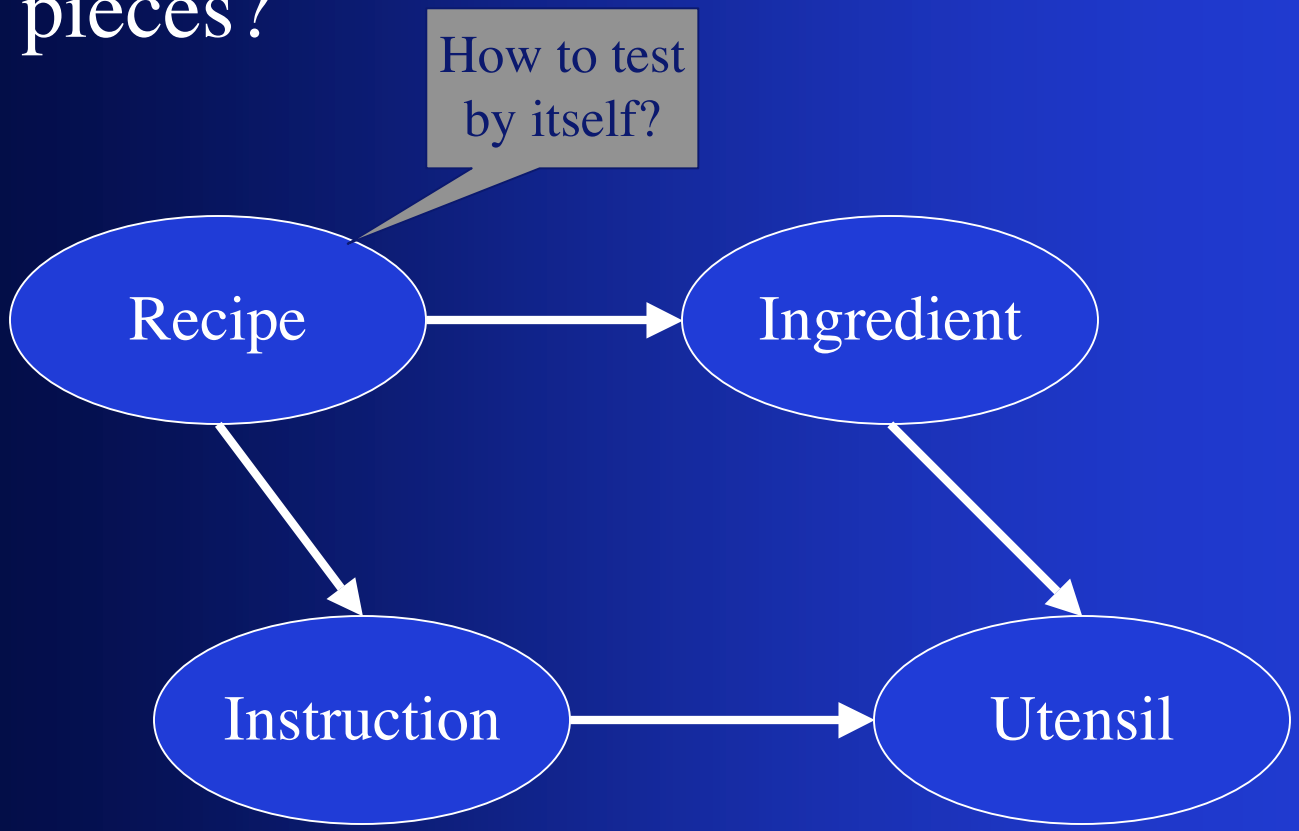
(Alice In Wonderland, Lewis Carroll)

The Assumptions

- You need to test your code
- You don't want to test by hand every time
- Therefore: You need automated tests
- But:
  - Your software is too large to test effectively as a single unit
  - Your software has classes/components which collaborate with each other
- Ergo: You need mock objects

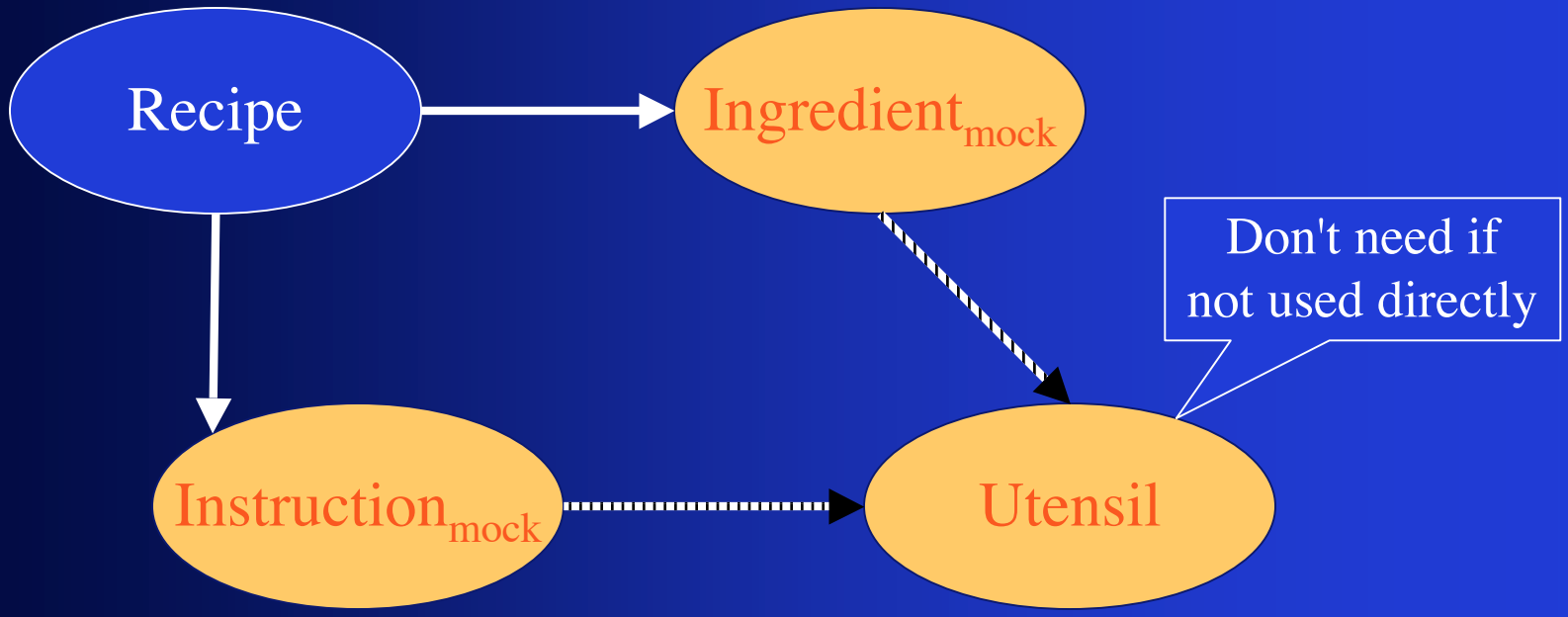
The Testing Problem

- In complex systems, how do you test the pieces?



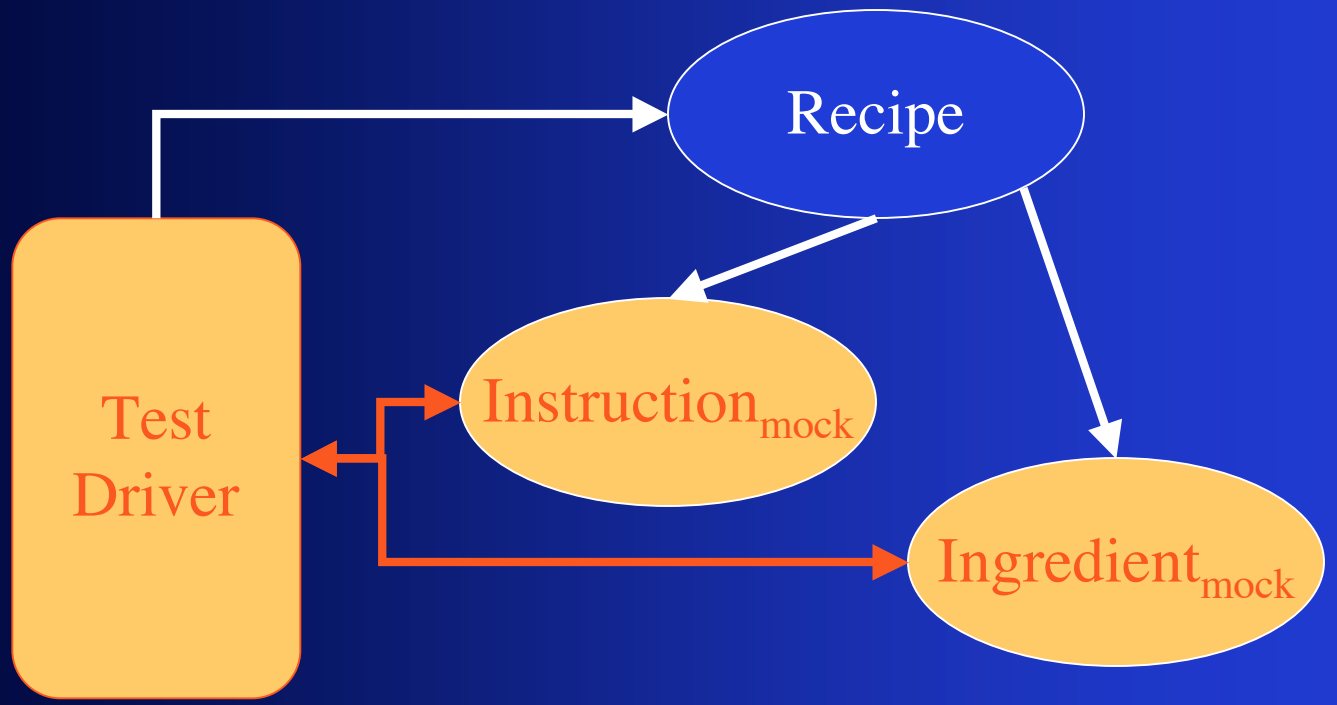
Mock Objects: The Idea

- Replace the real objects with substitutes, called Mock Objects
  - Not specific to Java



Using Mock Objects to Test

- Test code 'conspires' with Mock objects to make sure behavior of tested code meets requirements

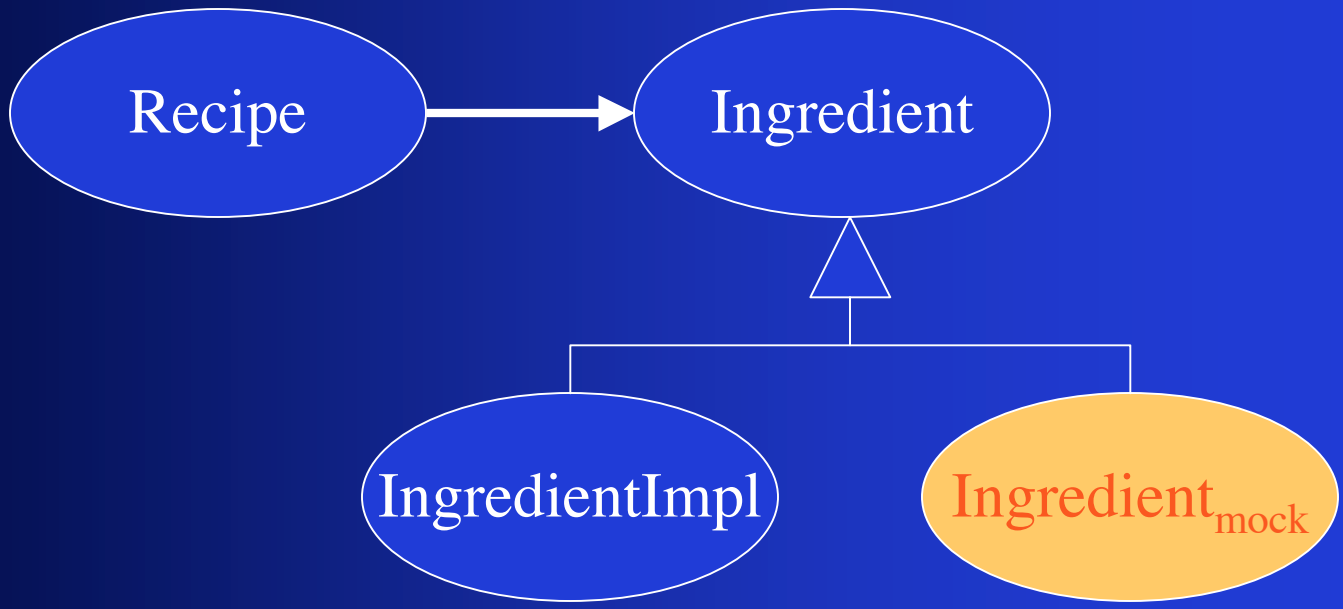


Principles of mock objects

- Start small
  - Start by testing smallest possible units
- Build upon previous tests
- Make tests automated
  - Otherwise, you just won't run them
  - Automated build systems can run them
- Tested object doesn't know there are mocks

Interfaces are Your Friend

- Easily replace normal implementation with mock
- Tested object can be unaware talking to a mock object



Dependency Injection is your Friend

- Prevents problems substituting the mock object for the real one
- Must be careful to avoid hard-coding types or directly instantiating dependencies
  - Avoid:

```
Ingredient ingredient = new SimpleIngredient(…);
```

Issue: Code doesn't use Interfaces

- Implementing Mock objects is awkward
  - Have to sub-class and override methods
  - Hard to get right
  - Brittle
  - Sometimes impossible
    - Generally indicates bad (or at least fragile) design of the object to be Mocked out

Issue: Dependency Resolution is Hard-Coded

- No real solution without changing the code
- Refactor code to use Dependency Injection
- Singletons aren't the answer, they are part of the problem

Mock Objects in the large

- Scaling up to test larger collections of code
- Component, Sub-system, module, service, etc.
- It's a 'normal' unit test unless your code makes remote calls
  - mostly (more later)

Mock Objects for Distributed Systems: Mockeries

- Needed when testing distributed systems
- Required to isolate & test a component
- Normal testing techniques fail when testing distributed apps
  - Local mock object doesn't test transport
    - Common cause of subtle or software/version-specific bugs
    - Can change semantics
  - Would like to skip testing but vendor's don't do all they should
    - So the developer has to do it instead

Mockeries (cont)

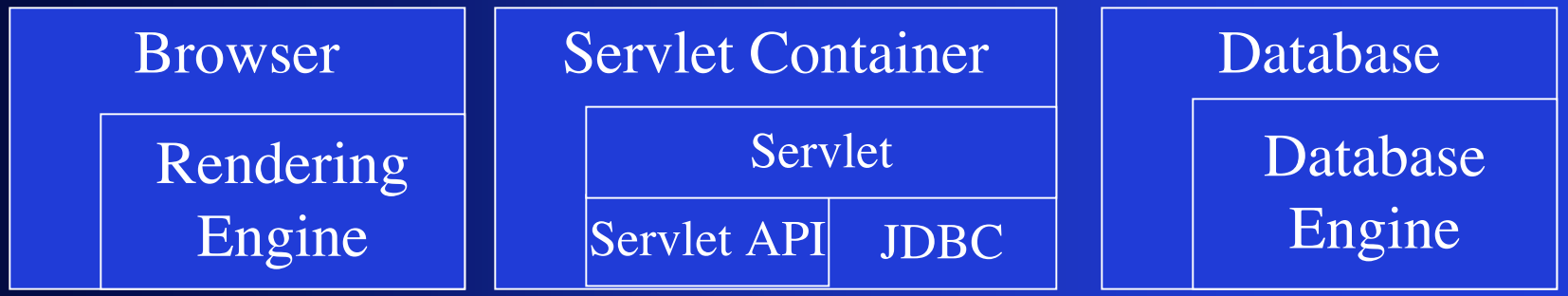
- Many real-world testing situations need them:
  - Databases
  - EJBs
  - Servlets
  - CORBA
  - Java Connector Architecture (JCA)
  - Sockets
  - Etc. (e.g. these are just Java examples)
- Assumes you've already tested individual classes/components

Example: WebApp Mockery

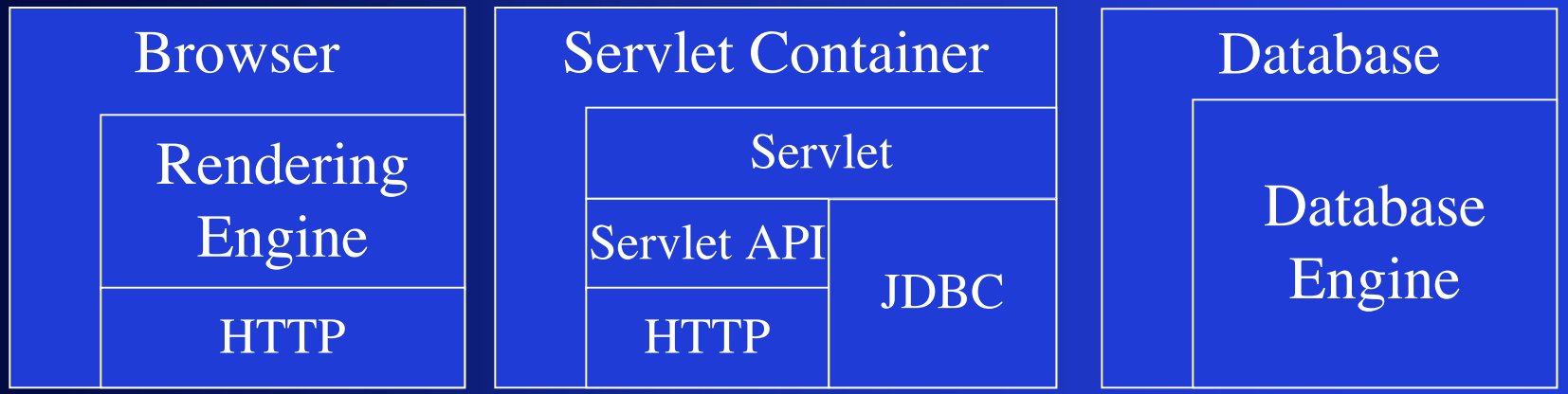
- Is really a bit more complicated

WebApp Mockery

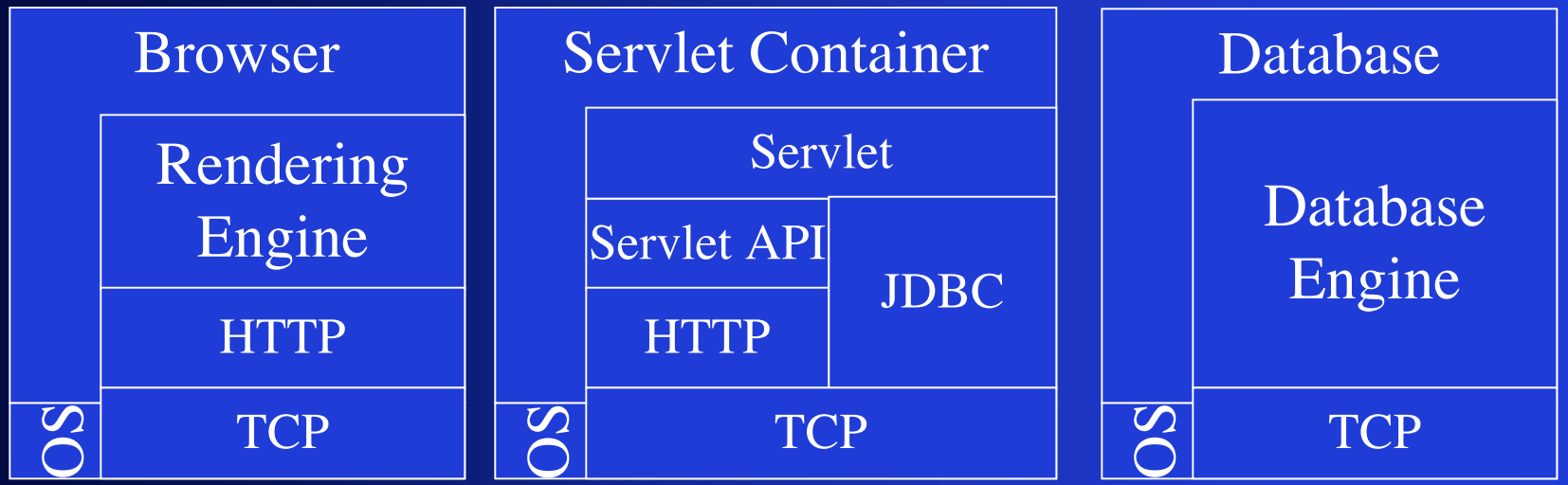
WebApp Mockery



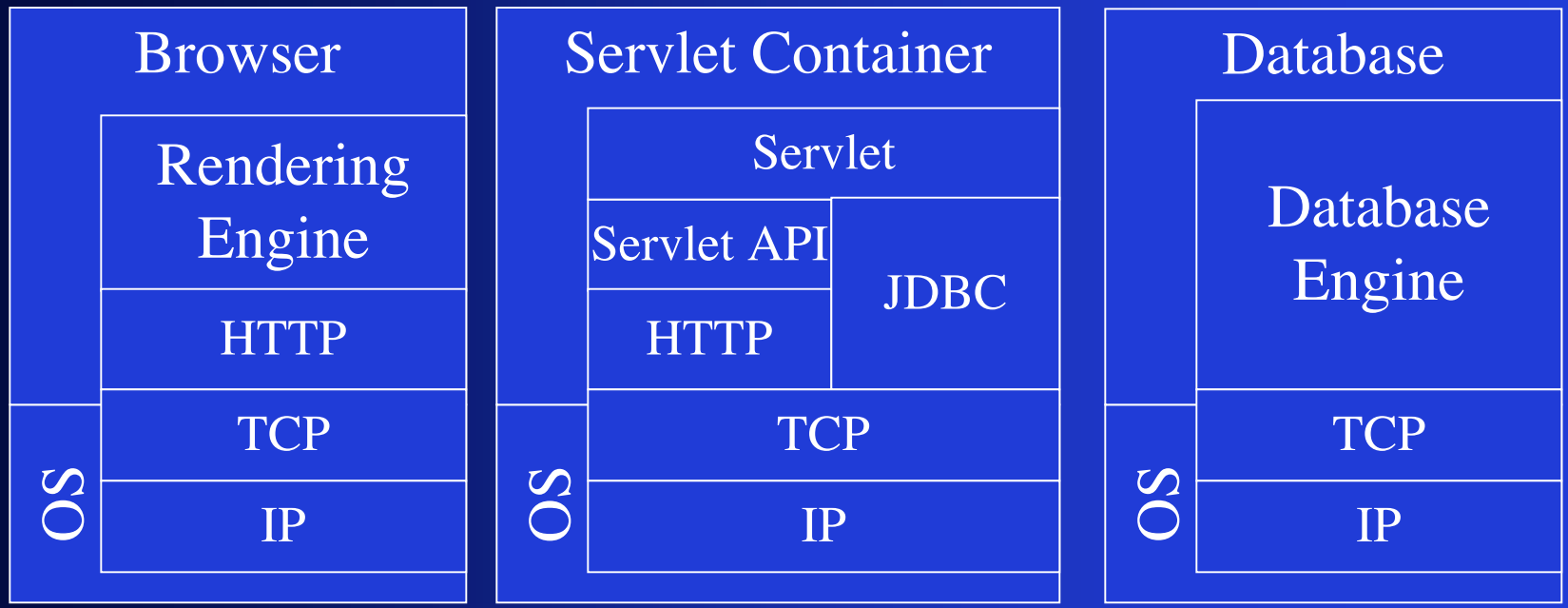
WebApp Mockery



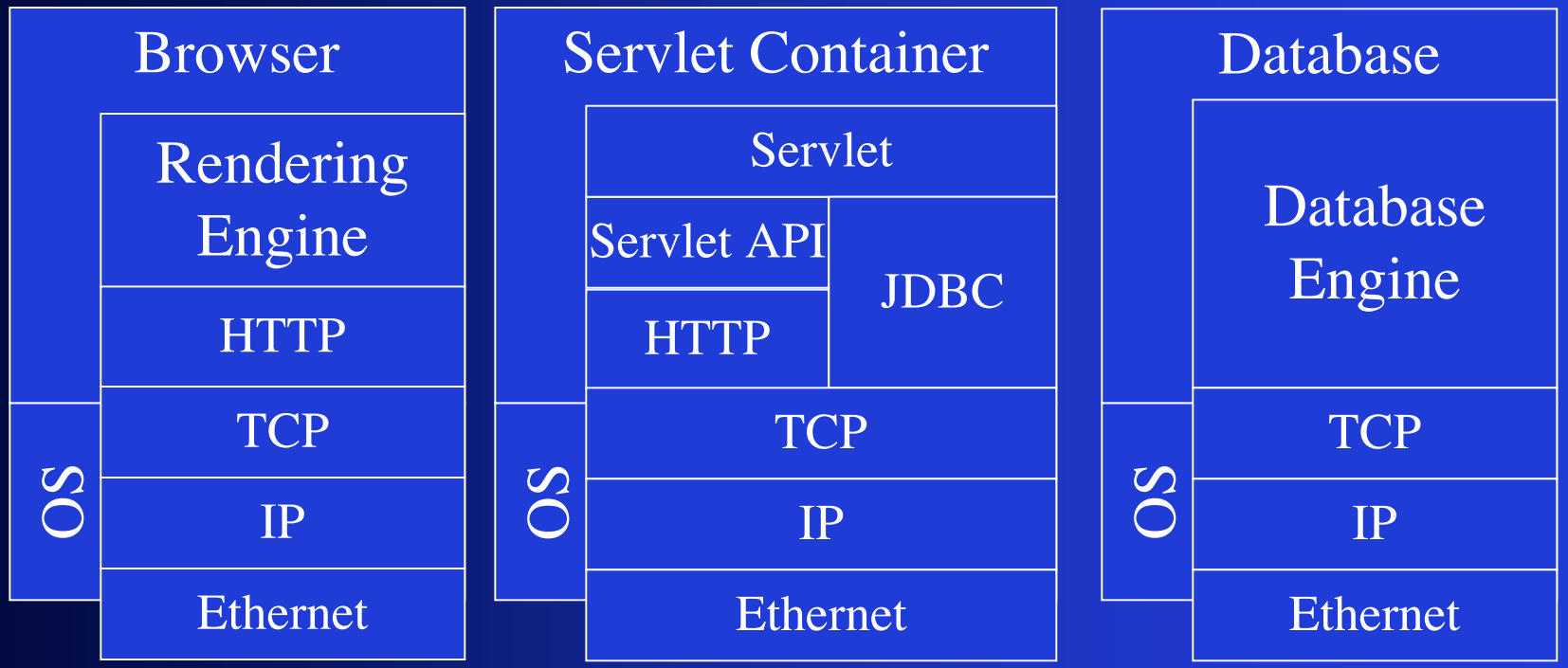
WebApp Mockery



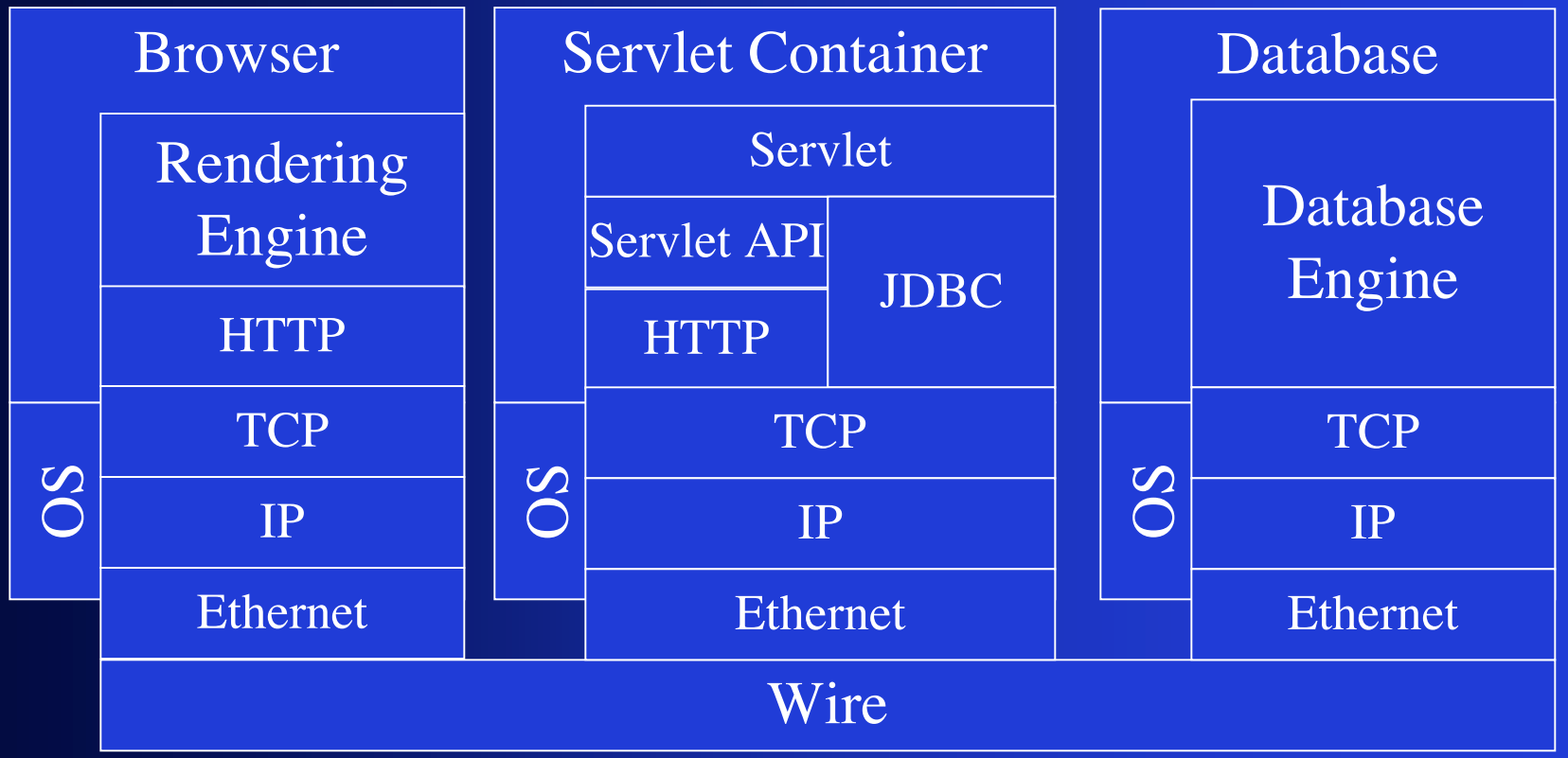
WebApp Mockery



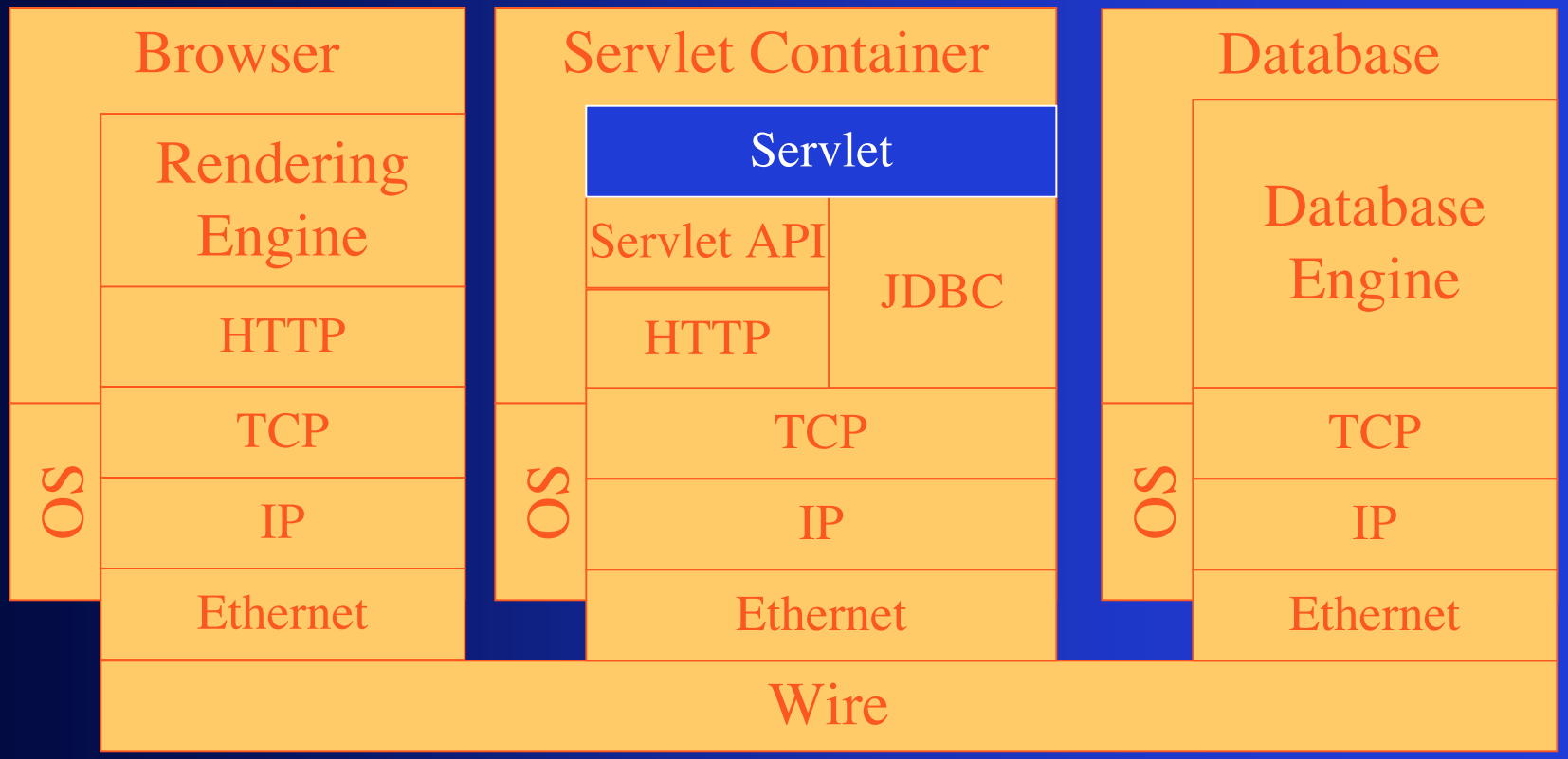
WebApp Mockery



WebApp Mockery



WebApp Mockery: Testing Opportunities



WebApp Mockery: Breaking it Down

- Lots of things we could simulate
- Start Simple
- Expand tests as
  - Time allows
  - Bugs require
- Hard to test entire path with simple mocks

Mockeries versus normal Mocks: Distance Matters

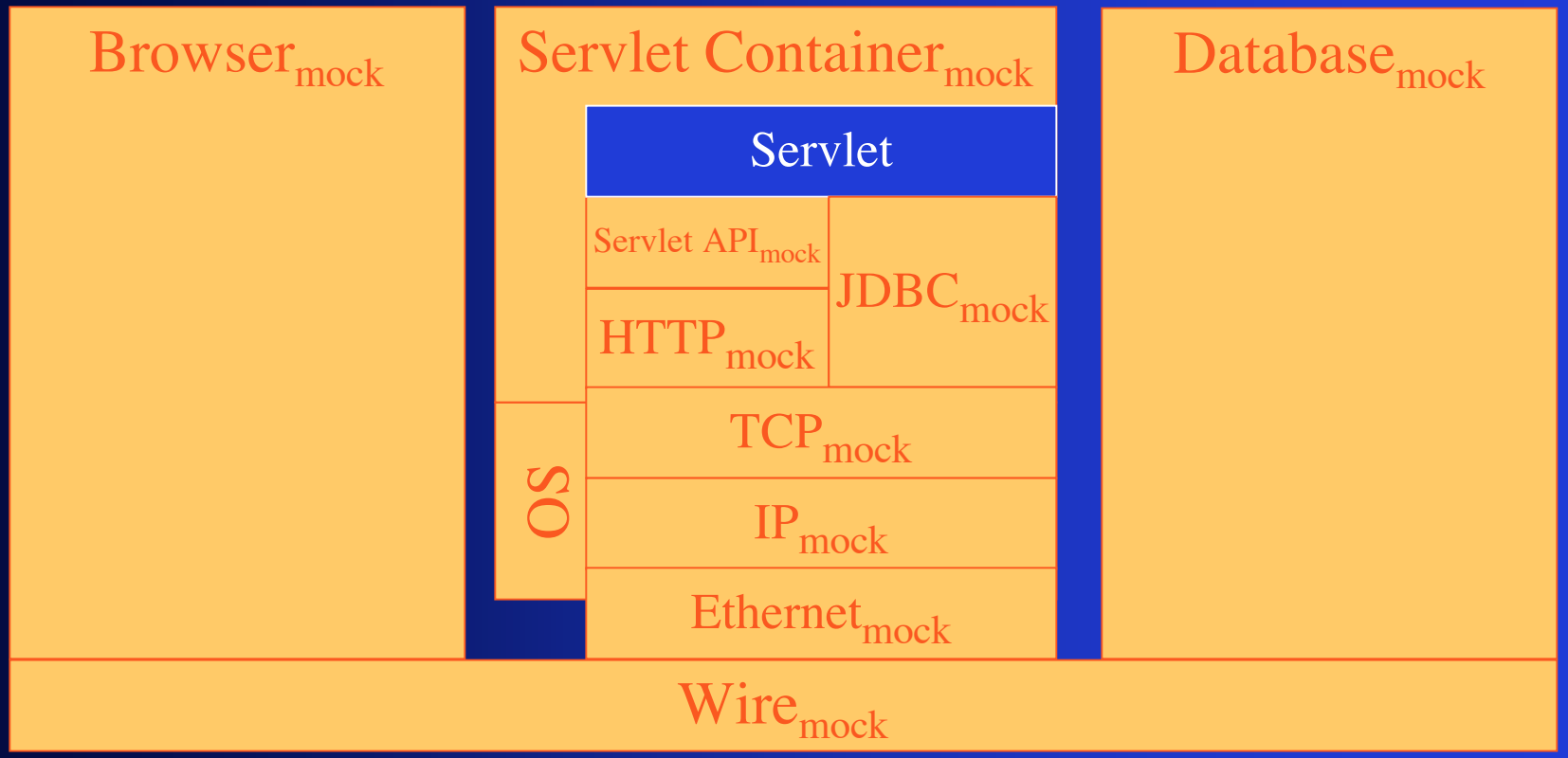
- New failure modes
  - Communication failure
  - Service/component failure
  - Transport errors
- New sources of existing failure modes
  - Race conditions
    - More likely
  - Deadlock
    - Much harder to diagnose
  - Bad dependency resolution (talking to a wrong/missing object)
    - Much more likely

Distributed Testing Tradeoffs

- Eliminate distribution to test correctness
  - Make sure to preserve pass-by-value/reference semantics of transport
- Use pre-planned requests and results to test distribution/timing issues
- Use random tests to search for subtle bugs

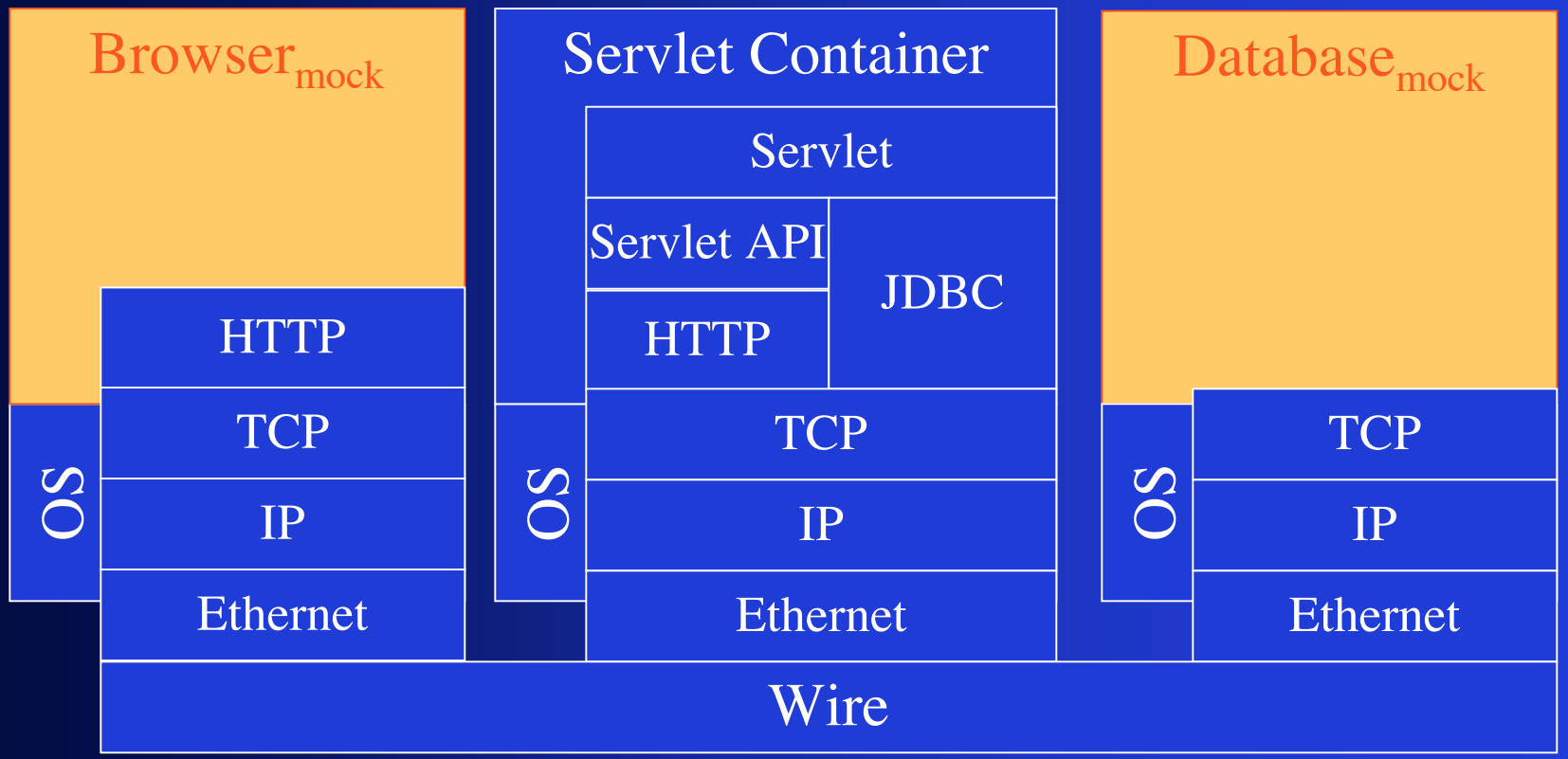
Distributed Testing Tradeoffs

- Eliminate distribution to test correctness
  - Make sure to preserve pass-by-value/reference semantics of transport



Distributed Testing Tradeoffs

- Use pre-planned requests and results to test distribution/timing issues



Distributed Testing Tradeoffs

- Use random tests to search for subtle bugs
  - Especially good for deadlock/race conditions
  - Worth their weight in 'gold' (development time) in complex systems
  - Requires you know how to recognize errors
    - Request/response is predetermined
    - Logging indicates errors
    - Crash or corrupted data leaves no doubt
  - Hardest to analyze

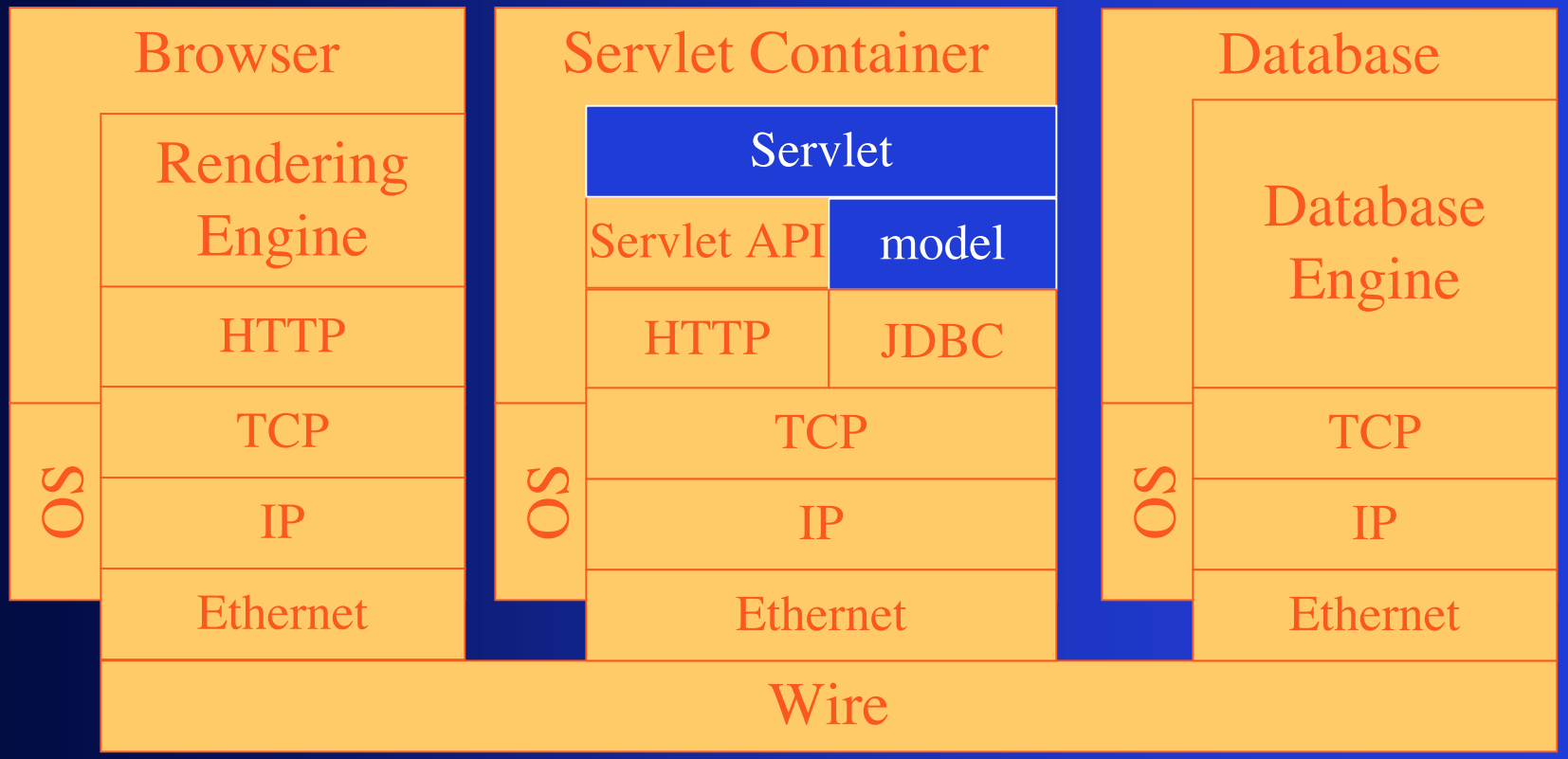
Divide and conquer

- Build up confidence with layers of tests
- Basic Correctness - unit tests
  - Remove distribution
  - One component at a time
- Timing
  - Simulated and real distribution
  - Force multi-threaded situations
- Resilience - test specifically for handling of failure modes
  - Network trouble
  - Remote component goes away
  - Incorrect results from called services
- Discovery
  - Doesn’t generally occur in standalone programs

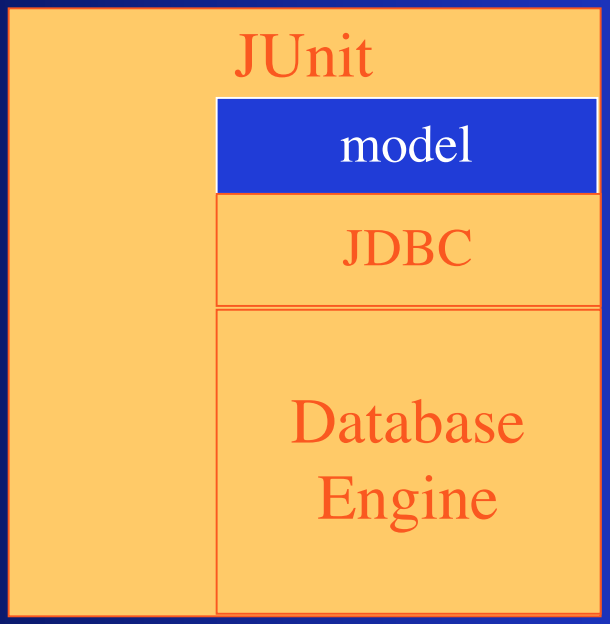
An Example: Music Database

- Artists
  - Name
  - Albums produced
- Albums
  - Tracks
- Tracks
  - Title
  - Track number

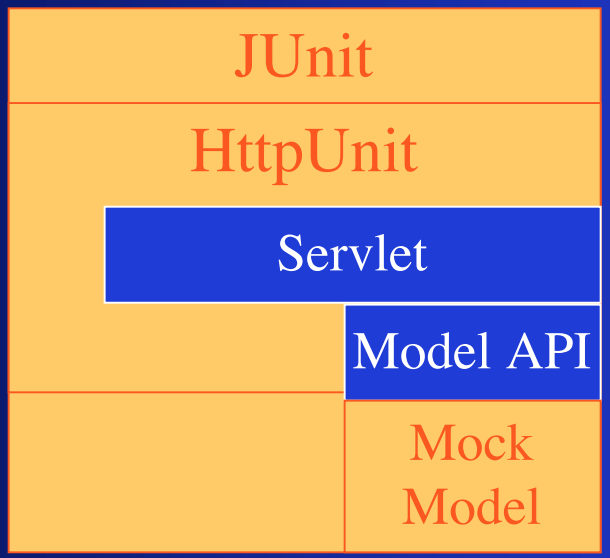
Music Database: Interfaces



Music Database: Test the model



Music Database: Test the servlet



Music Database: Where next?

- Multi-threaded test (concurrency)
  - No servlet, just model
  - Mock model, just servlet
- Resource management test
  - JDBC connection management
  - Mock driver that tracks open connections, statements, etc.
    - Fails if any left open after executing test
- Etc.

Making a Mockery

- Determine what you want to test:
  - Correctness of component logic
  - Correctness of transport/marshalling
  - Timing
  - Deadlock
- Re-use existing mock frameworks if possible
- Build mocks if you have to
- Strive to simulate the real world
  - e.g. don’t try to test for timing issues on a single-CPU system

Mockeries: Conclusion

- "I found it in the last place I looked"
  - Bugs are in the code/situations we haven't tested
- Doesn't this go beyond ‘unit’ testing?
  - Yes
  - If you've implemented unit tests, remaining bugs aren't caught through normal unit tests
- Not always easy to implement
  - Much easier than thrashing about with one-off tests or code inspection
- Don't throw away effort required to reproduce the bug!
  - Capture it in a Mockery

References

- Original Paper
  - http://www.connextra.com/aboutUs/mockobjects.pdf
- General Sites
  - http://www.mockobjects.com
- Articles
  - http://www.ociweb.com/jnb/jnbJun2003.html
  - http://www-106.ibm.com/developerworks/library/j-mocktest.html

More References

- Frameworks
  - https://www.mockobjects.com/
  - https://www.easymock.org
  - http://httpunit.sourceforge.net/
  - http://www.hsqldb.org/
  - http://jdbcunit.sourceforge.net/
  - Etc.
- Books
  - Pragmatic Unit Testing (Pragmatic Programmers)
  - Unit Testing in Java: How Tests Drive the Code
    - <Soapbox>Test-driven development is good; not so sure about test-first development</Soapbox>
- Lots more