Embedded Functional

A FitNesse Way

Version 1.0

Code and slides at...

- https://github.com/schuchert/sdc_2013_cpp

- Clone the repo:
  git clone git://github.com/schuchert/sdc_2013_cpp.git

- Review README.md (or that website)

Functional Testing

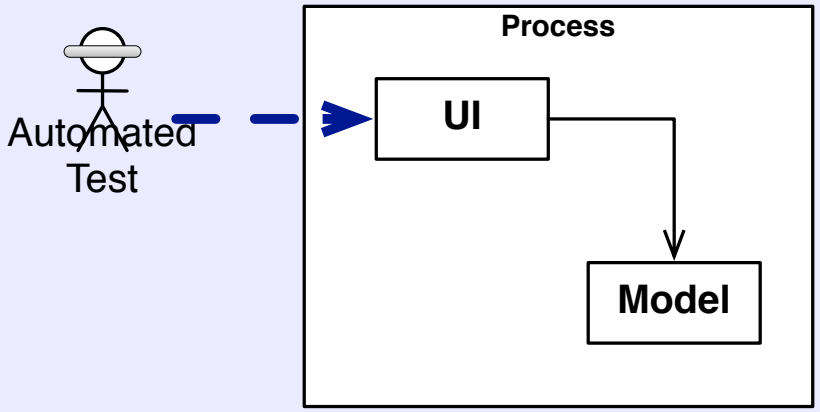
- So we are on the same page, here's one definition:
  ... a type of black box testing that bases its test cases on the specifications ... Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered
  - http://en.wikipedia.org/wiki/Functional_testing
- Key Points
  - Black Box
  - Feed input
  - Get output
  - Internal Program Structure Rarely Considered

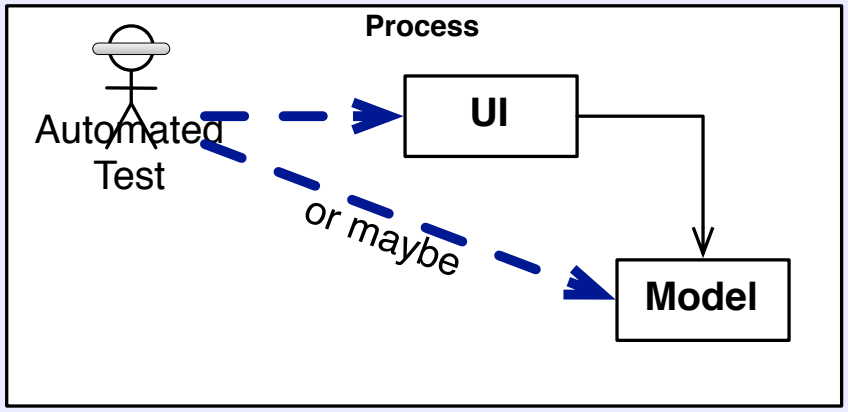
Assertion

- Test is not attempting to prove correctness ...

- Rather, it is attempting to reduce likelihood of releasing a defect into the wild

Embedded?

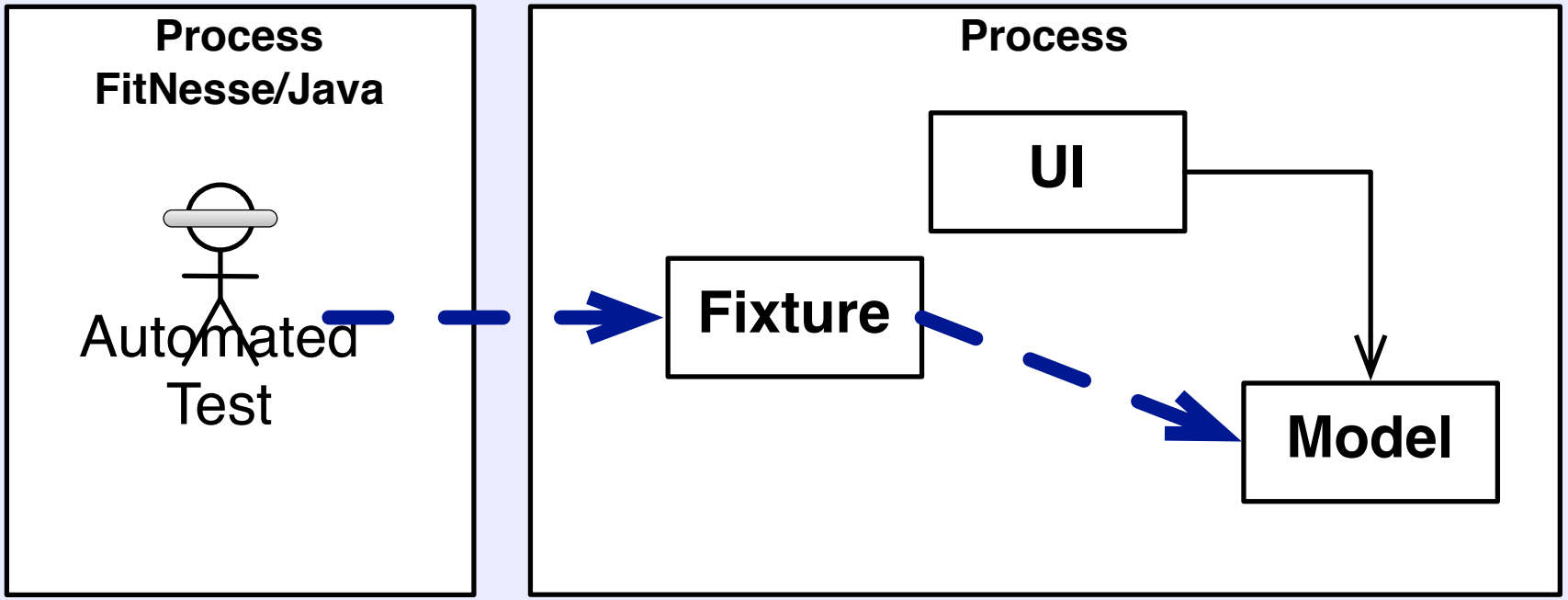

Process
Automated Test
UI
Model


Process
Automated Test
UI
or maybe
Model

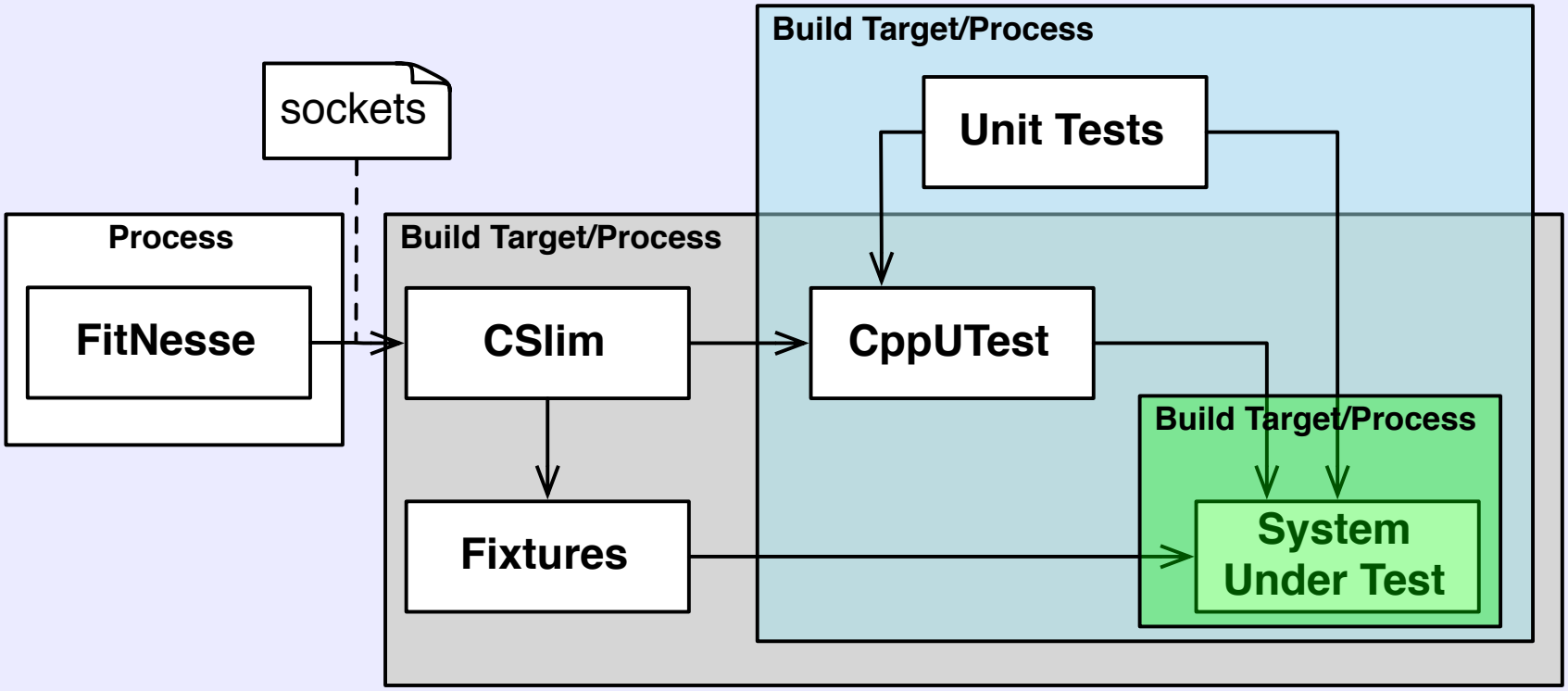
Considerations

- Con
  - May not fully test functionality
  - Wiring
  - Probably write more, smaller tests
  - Build
- Pro
  - Potential better separation of concerns
    - UI tech. independent / Potential less impact to UI changes
    - Hypothetical syllogism - http://krypton.mnsu.edu/~tony/courses/609/Logic/Logic2.html
  - Can get more direct access to results
  - Write more, smaller tests

What we'll look at

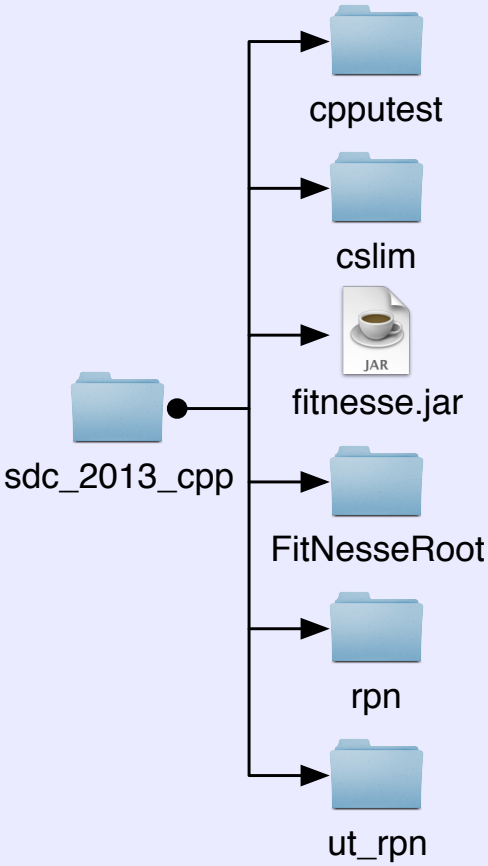


The Moving Parts



A few notes

- Originally developed under gcc 4.6
  - The current version is using gcc 4.8
  - Uses some features of C++ 11
  - -std=c++11



  - `git://github.com/schuchert/sdc_2013_cpp.git`

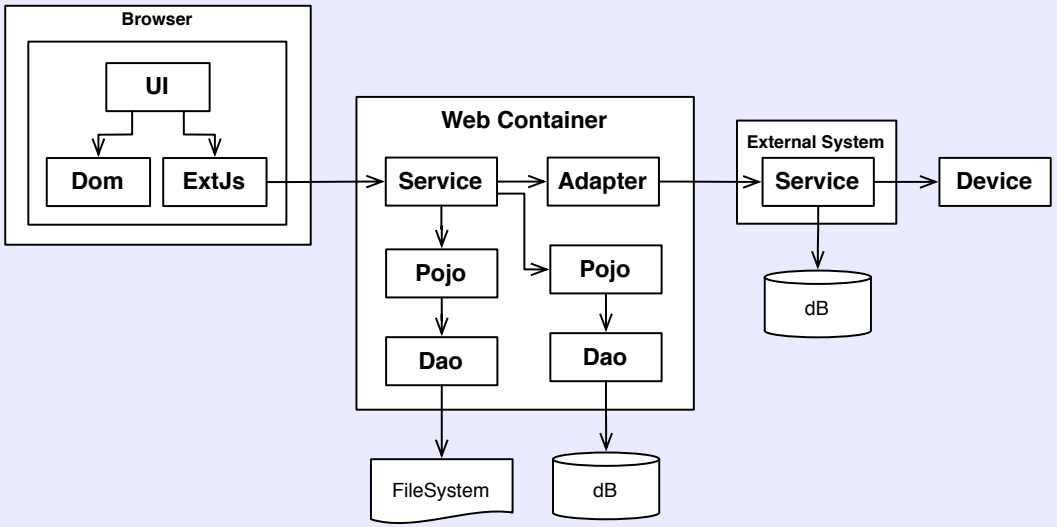
Code

- Start FitNesse
- Run some tests
- Look at the moving parts
- More at:
  http://schuchert.wikispaces.com/cpptraining.GettingStartedWithFitNesseInCpp
  http://schuchert.wikispaces.com/cpptraining.ExamlesOfEachFixtureType

Test Thickness

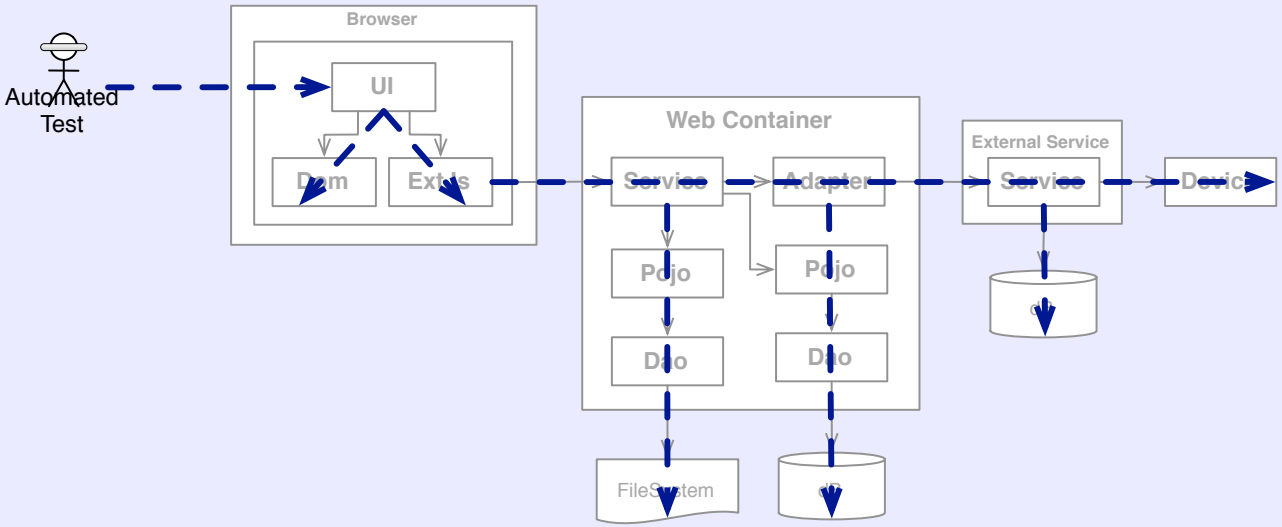
Version 1.0

A System


Browser
UI
Dom
ExtJs
Web Container
Service
Adapter
Pojo
Pojo
Dao
Dao
FileSystem
dB
External System
Service
Device
dB

Fully Integrated, Black Box

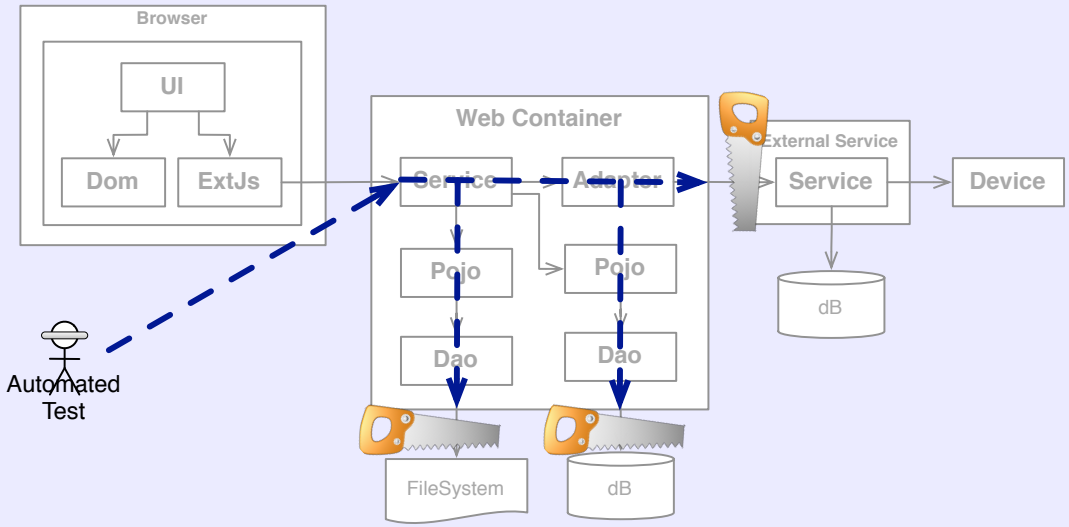
- Typical first cut



- Tests written against "known" instance. Might be configurable.
- Test will fail if external systems not in place: web server, database, file system, etc.
- Tests do not stand alone...

Service Test

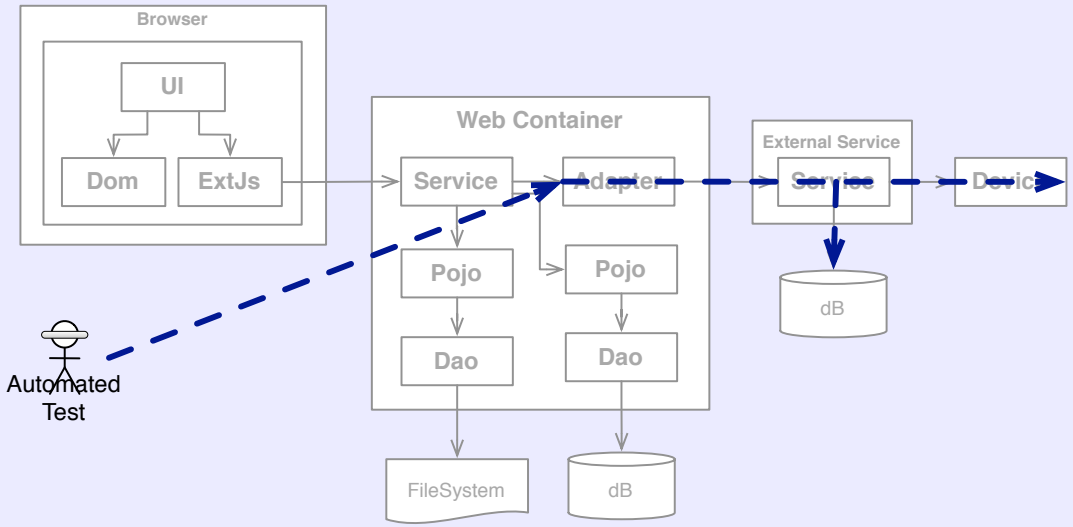
- Make sure service logic works



- Automated tests starts just enough of the system
- Decouples from external stuff like file system, dB, etc. when possible (this is a design decision that should have already been made)

Adapter Integration Test

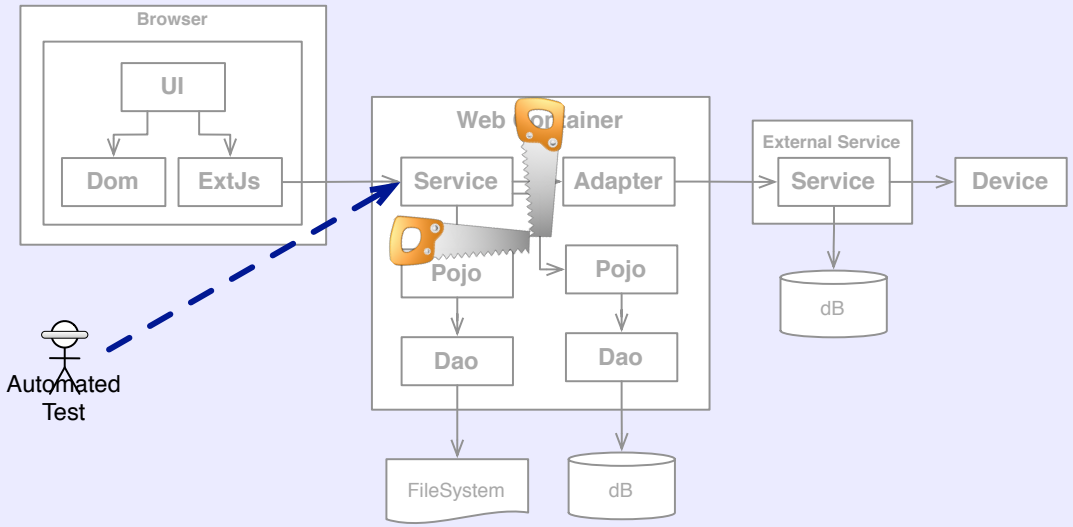
- If service does not check integration points, or even if it does, still need to make sure integration with external services work



- We could write this test against the service, and may even need to. But avoid if we can. Why?

Unit Test

- When targeting a single object, control all external dependencies, making for a fast, focused, unit test



- This could easily target anything in the system.
- What about cutting out the Pojo?

Unit Test - a definition

- ***Given*** an object to be tested
  When verifying a use of that object
  Then no method invocations leave that object to something that is not under the test's direct control
  - What about the String class?
  - What about using a Java Bean, e.g. a DTO?

Integration Test - a definition

- ***Given*** a number of interacting objects
  When verifying a use of those objects
  Then no method invocations leave those objects to something that is not in the test's control

Integration Test - alternative def

- ***Given*** a system with configurable sub-systems/ components/parts
  When verifying its general use of each of those sub-systems/components/parts
  Then check that the plumbing does not leak (the system still has its smoke)

- The cost of introducing flexibility for configuration ***creates*** a whole category of test cases that would otherwise not necessarily exist

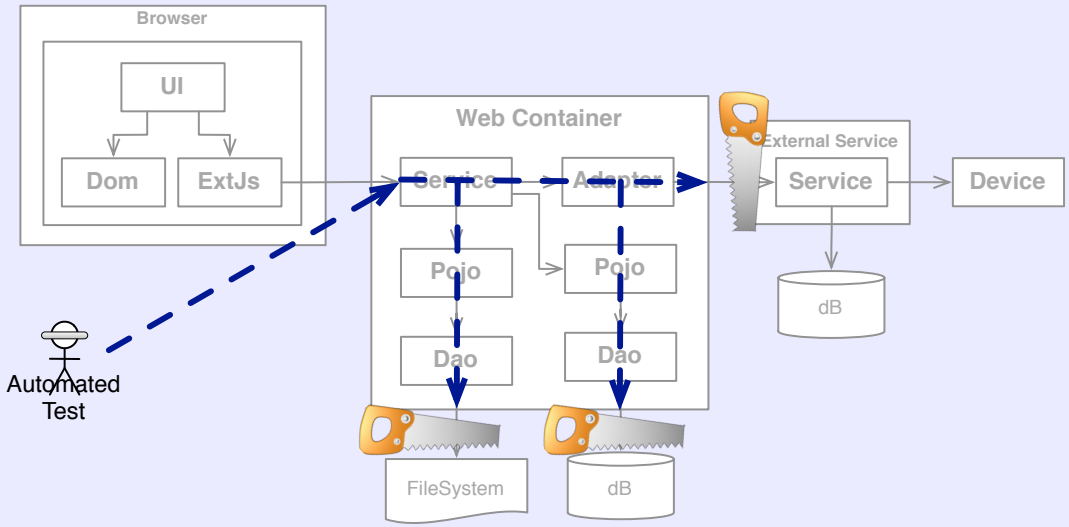
Unit versus Integration

- Two perspectives
  - It's a matter of thickness
  - Integrated tests are a scam: http://blog.thecodewhisperer.com/2010/10/16/integrated-tests-are-a-scam/
- Why do we care about thickness?
  - The complexity to solve any problem grows at least as fast as the square of the number of things you are trying to solve simultaneously - Weinberg, *An Introduction to General Systems Thinking*
  - Corollary: the likelihood that a test will break grows at least as fast as the square of the number of layers of interaction (internal and external) -> thick test - fragile test

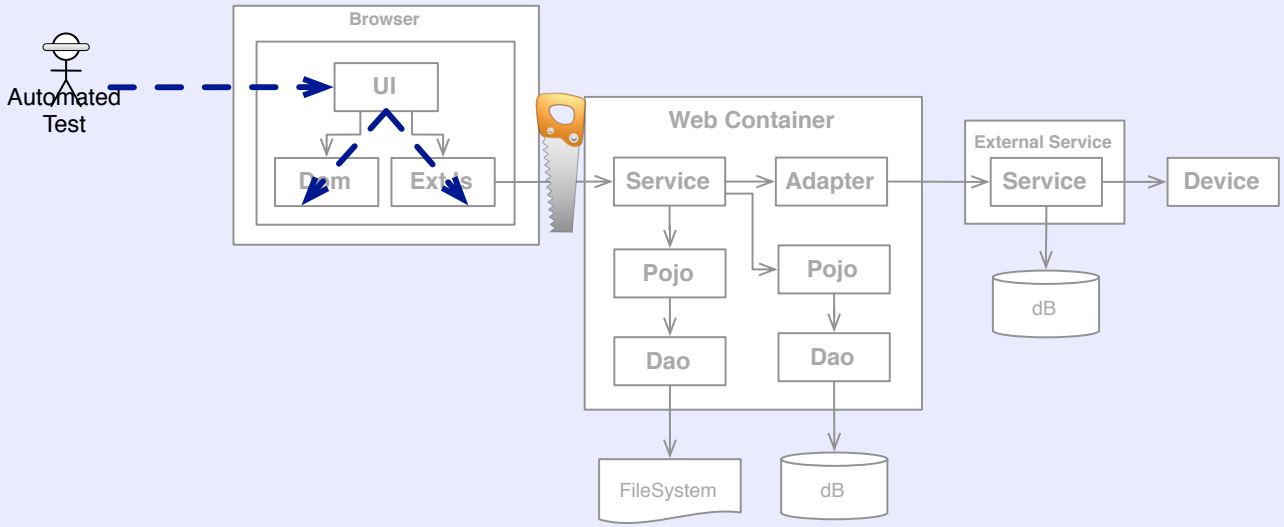
Acceptance Tests vs. Integration

- When should an acceptance test be fully integrated?
- When not?
- What about headless testing?
- What about body-less testing?

Headless Testing



Body-less Testing



Automated Test Design

- 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.

Saws: 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.

Why do we even care?

Jeopardy Style...

Version 1.0

Maintenance & Evolution

> 90%

Maintenance & Evolution

> 90%

- What is the approximate % of the budget devoted to maintenance & evolution?
  - Source: http://users.jyu.fi/~koskinen/smcosts.htm

Maintenance & Evolution

> 10X

Maintenance & Evolution

> 10X

- Ratio of reading code to writing code
  - Early on
  - Longer-lived legacy systems, both higher and lower

Regression

66%

Regression

66%

- What is the chance that a one-line defect fix will introduce another defect?
  - Source: Weinberg (QSM series & confirmed via personal email)

Change

1287

Change

1287

- What is the largest number of files in a single check-in Brett's changed without introducing a defect?
  - 50ish developers, 1.7 MLOC, Java

How Is That?

How Is That?

Test Automation

What Helps Automation?

What Helps Automation?

Design

So just what is TDD?

So just what is TDD?

- TDD
  - is a design practice,
  - it uses tests as a mechanism for ***discovery*** and ***feedback***
  - it is one end of a spectrum

  - it is about releasing waste water upstream
  - it is about regression
  - it is not always the right thing to do...

How Old Is The Idea?

How Old Is The Idea?

- Late 50's
  - Original Mercury Rocket Project
  - Computers expensive
  - People relatively inexpensive
  - Produce scenarios
  - Hand calculate
  - Write programs (card deck)
  - Execute against scenarios

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