

Unit Testing and Test Driven Design

Björn Beskow

Callista Enterprise AB

bjorn.beskow@callista.se

<http://www.callista.se/enterprise>

keep the bar green to keep the code clean...



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 1
Copyright 2003, Callista Enterprise AB



Unit Testing and Test Driven Design

□ Target audience

- *Developers, Designers, Architects, Project Managers and Project Sponsors interested in lean and mean ways to achieve good-enough quality without paying an excessive price*

□ Objectives

- *Provide an overview of Unit Testing, and how Designing with Testability in mind changes your way of thinking*

□ Non-Objectives

- *To say anything about Functional Testing, Performance Testing, GUI Testing, ...*

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 2
Copyright 2003, Callista Enterprise AB





Agenda

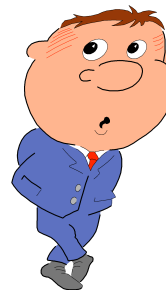
- About Tests and Testing
- What is a Unit Test?
- Inside a Unit Test
- Test-First Design: How does Testability affect your way of thinking?

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 3
Copyright 2003, Callista Enterprise AB



About Tests ...

- Everybody knows they should, but few actually do
- "Why isn't this tested before?"
 - *Because it has been too expensive, difficult, cumbersome to test*
 - *Because we have been too busy*
 - *Because things have changed*



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 4
Copyright 2003, Callista Enterprise AB





Absence of tests ... Greetings from Hell!



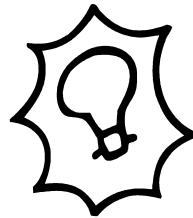
CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 5
Copyright 2003, Callista Enterprise AB



The Diabolical Challenge of Modern Software Development

To rapidly complete large projects that are both research-like and mission-critical in a turbulent business and technology environment.

- *Exciting Features*
- *Rapid delivery*
- *High quality*
- *High change*
- *Low cost*



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 6
Copyright 2003, Callista Enterprise AB





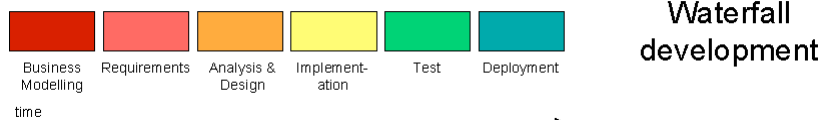
OMG's Six 'Best Practices' for Software Engineering

1. **Develop software iteratively**
2. Manage requirements
3. **Use component-based architectures**
4. Model software visually
5. **Verify software quality continuously**
6. Control changes

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 7
 Copyright 2003, Callista Enterprise AB

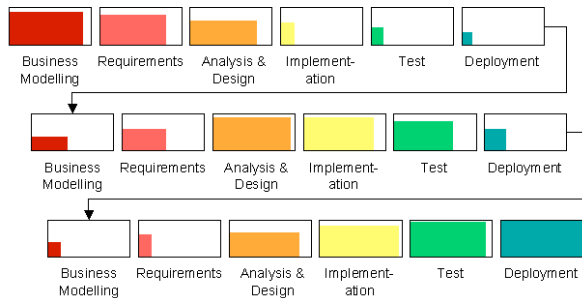
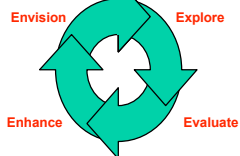


1. Develop software iteratively



VS

Iterative development



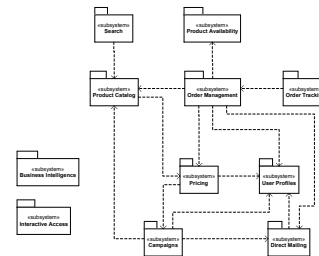
CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 8
 Copyright 2003, Callista Enterprise AB





3. Use component-based architectures

- Using well-defined interfaces makes a system much more resilient to change
- A component-based architecture enables projects to run more efficiently and with lesser risk

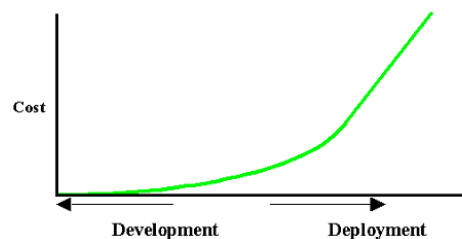


CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 9
Copyright 2003, Callista Enterprise AB



5. Verify software quality continuously

- Quality: The characteristic identified by the following:
 - *satisfies or exceeds an agreed upon set of requirements, and*
 - *assessed using agreed upon measures and criteria, and*
 - *produced using an agreed upon process*
- Test every iteration – automate test!



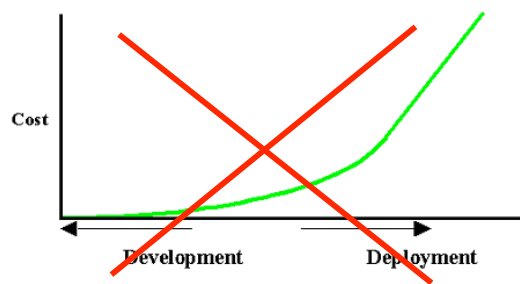
CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 10
Copyright 2003, Callista Enterprise AB





Refactoring challenges the Software Quality Entropy!

- The device 'Do it right the first time' sends the wrong message to an iterative project – make sure you do it right the last time!
- Refactoring is a systematic approach to improve the design and quality of an existing system, without changing its external behaviour.

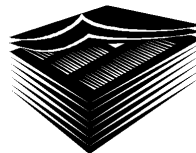


CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 11
Copyright 2003, Callista Enterprise AB



Full Lifecycle Object Oriented Testing

- Requirements Testing
 - *Use-case scenario testing*
 - *Prototype walkthroughs*
 - *User Requirements reviews*
- Analysis & Design Testing
 - *Model walkthroughs*
 - *Prototype walkthroughs*
 - *Peer reviews*
- Code Testing
 - *Black-box testing*
 - *White-box testing*
 - *Boundary-value testing*
 - *Class-integration testing*
 - *Class testing*
 - *Code reviews*
 - *Coverage testing*
 - *Regression testing*
- System Testing
 - *Function testing*
 - *Installation testing*
 - *Stress testing*
 - *Operations testing*
 - *Support testing*
- User Testing
 - *Alpha testing*
 - *Beta testing*
 - *Pilot testing*
 - *User acceptance testing*



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 12
Copyright 2003, Callista Enterprise AB





Gee, that sounds both difficult, boring and expensive!

- Yes, all testing comes with a price.
 - 0% defect rate is impossible, and perhaps not even desirable?
- but ...
- If it can be built, it can also be tested!
 - If it's not worth testing, maybe it's not even worth building?

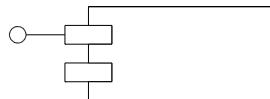
Lesson: **Test cheap, test early, test often!**

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 13
Copyright 2003, Callista Enterprise AB



Unit Tests

- Black-box or White-box test of a *logical unit*, which verifies that the logical unit behaves correctly – *honors its contract*.



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 14
Copyright 2003, Callista Enterprise AB





Smoke Tests

- A set of Unit Tests (which tests a set of logical units) executed as a whole provides a way to perform a *Smoke Test*: Turn it on, and make sure that it doesn't come smoke out of it!
- A relatively cheap way to see that the units "seems to be working and fit together", even though there are no guarantees for its overall function (which requires functional testing)

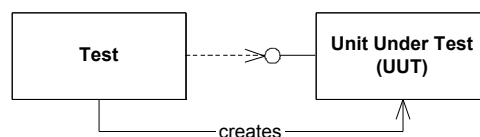


CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 15
Copyright 2003, Callista Enterprise AB



What exactly is a Unit Test?

- A self-contained software module (typically a Class) containing one or more test scenarios which tests a Unit Under Test *in isolation*.
- Each test scenario is autonomous, and tests a separate aspect of the Unit Under Test.



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 16
Copyright 2003, Callista Enterprise AB





Unit Test Example

```
public interface Account {  
    public void withdraw(double amount);  
    public void deposit(double amount);  
    public double balance();  
    ...  
}  
  
public class AccountTest extends TestCase {  
    public void testWithdraw() {  
        AccountImpl account = new AccountImpl("1234-9999", 2000);  
        account.withdraw(300);  
        assertEquals(account.balance(), 1700);  
    }  
    public void testWithdrawTooMuch() { ... }  
    ...  
}
```

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 17
Copyright 2003, Callista Enterprise AB



Desiderata for Unit Tests

- Easy to write a test class
- Easy to find test classes
- Easy to test different aspects of a contract
- Easy to maintain tests
- Easy to run tests



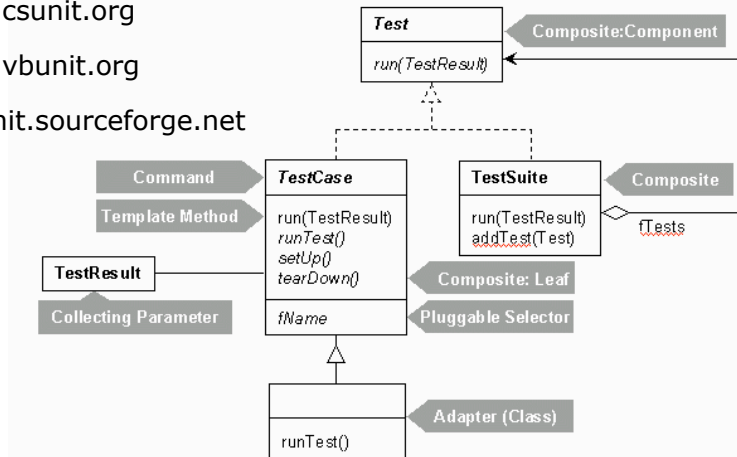
CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 18
Copyright 2003, Callista Enterprise AB





XUnit: A Framework for Unit Tests

- www.junit.org
- www.csunit.org
- www.vbunit.org
- cppunit.sourceforge.net



CADEC 2003-01-:
Copyright 2003, Callista Enterprise AB



Test-Driven Design

Unit Tests may be written very early. In fact, they may even be written *before any production code exists*:

1. Write a test that specifies a tiny bit of functionality
2. Ensure the test fails (you haven't built the functionality yet!)
3. Write the code necessary to make the test pass

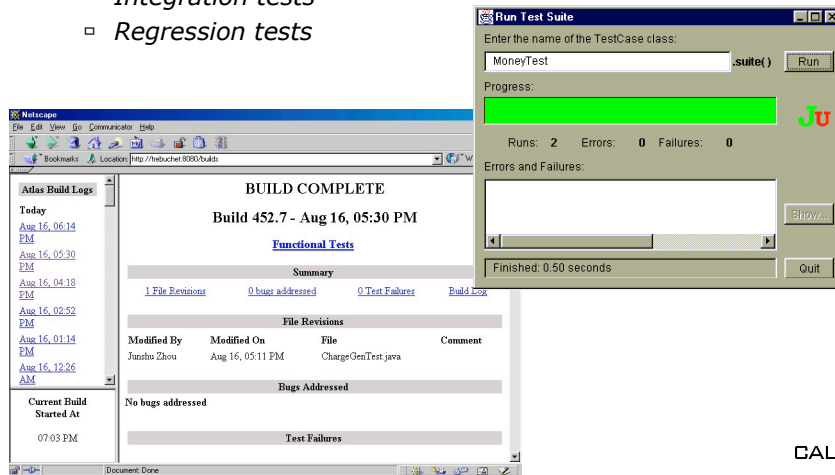
There is a certain rhythm to it: Design a little – test a little – code a little – design a little – test a little – code a little – ...

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 20
Copyright 2003, Callista Enterprise AB



Obvious Effects of Test-Driven Design

- Already automated tests, immediately useful for
 - *Integration tests*
 - *Regression tests*



The screenshot shows a Netscape browser window displaying a build log for 'Atlas Build Logs'. The main content area shows 'BUILD COMPLETE' for 'Build 452.7 - Aug 16, 05:30 PM'. Below this, there are sections for 'Functional Tests', 'Summary', 'File Revisions', 'Bugs Addressed', and 'Test Failures'. The 'Summary' section shows '1 File Revisions', '0 bugs addressed', and '0 Test Failures'. The 'File Revisions' table has one entry: 'Modified By: Jianhui Zhou, Modified On: Aug 16, 05:11 PM, File: ChargeOrnTest.java, Comment:'. The 'Bugs Addressed' section shows 'No bugs addressed'. The 'Test Failures' section is empty. Overlaid on the browser is a 'Run Test Suite' dialog box. It has a text field containing 'MoneyTest' and a '.suite()' suffix. There is a 'Run' button. Below the text field is a progress bar that is green. It shows 'Runs: 2', 'Errors: 0', and 'Failures: 0'. There is a 'Show...' button and a 'Finished: 0.50 seconds' label. There is also an 'Out' button.

CALLISTA

Not-so-obvious Effects of Test-Driven Design

- Intentional Design of Interfaces
 - *Since the code in question is not written yet, we are free to choose the interface that is most usable.*
- Non-speculative Interfaces
 - *Interfaces provide the functionality which is just enough for right now*
- Documented requirements and intended usage
 - *The tests themselves provide immediately useful documentation of the Interfaces*
- Good OO Design: High Cohesion and Low Coupling
 - *If you have to write tests first, you'll devise ways of minimizing dependencies in your system in order to write your tests.*

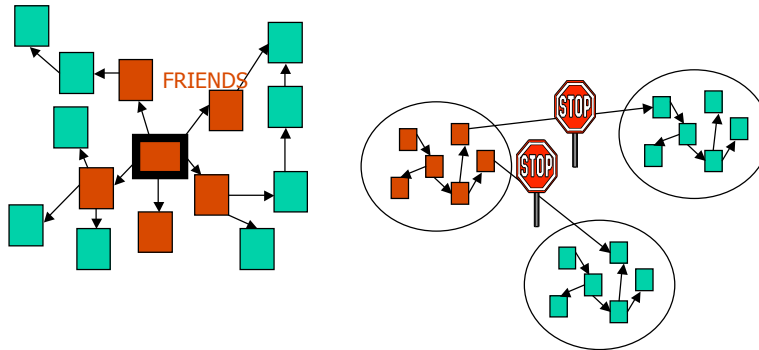
CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 22
Copyright 2003, Callista Enterprise AB

CALLISTA



Designing for Testability: *Low Coupling*

- Minimize dependencies between classes
- Only allow "closely related" classes to interact directly

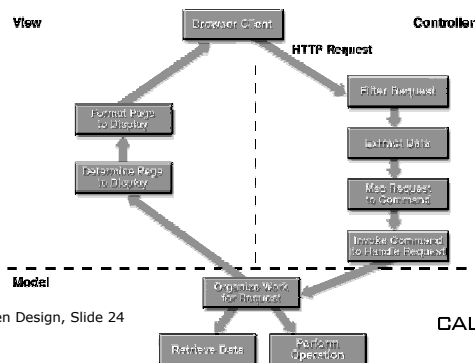
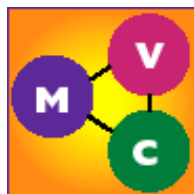


CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 23
Copyright 2003, Callista Enterprise AB



Designing for Testability: *Model-View-Control*

- User Interfaces are notoriously difficult to test
- Splitting a complex application into separate, cohesive parts which separates presentation from application logic allows testing the application logic in isolation



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 24
Copyright 2003, Callista Enterprise AB





So what exactly is a Unit?

- Class?
- Interface?
- Component?
- Sub-system?
- Whole system?

Pragmatics!



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 25
Copyright 2003, Callista Enterprise AB



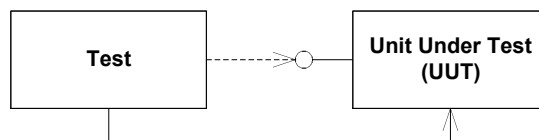
Design properties and Design goals

For Units:

- Modularity
- High cohesion
- Low coupling

For Tests:

- Modularity
- Locality

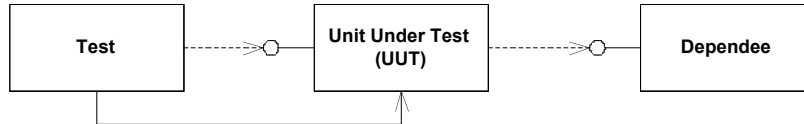


CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 26
Copyright 2003, Callista Enterprise AB





But what about units that depend on other units (with potential side effects)?



□ Different strategies possible:

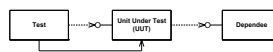
- Create a real context
- Run in it's natural context (In Container in the case of J2EE or .NET)
- Create a synthetic context

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 27
Copyright 2003, Callista Enterprise AB

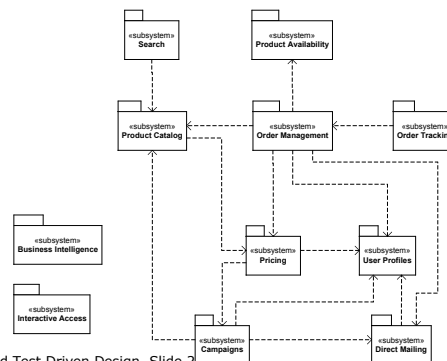


Create a real context

□ Ok



□ ?



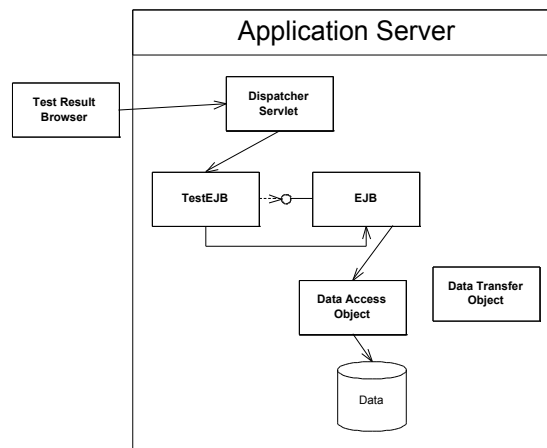
CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 28
Copyright 2003, Callista Enterprise AB





Run in context - Cactus

□ jakarta.apache.org/cactus/

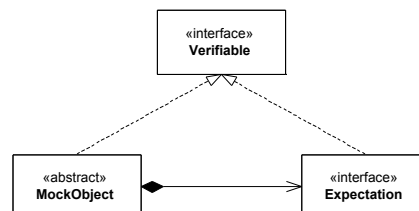


CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 29
Copyright 2003, Callista Enterprise AB



Synthetic context – MockObjects

- Implements the same interface as the resource that it represents
- Enables configuration of its behavior from outside (i.e. from the test class, in order to achieve locality)
- Enables registering and verifying *expectations* on how the resource is used



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 30
Copyright 2003, Callista Enterprise AB





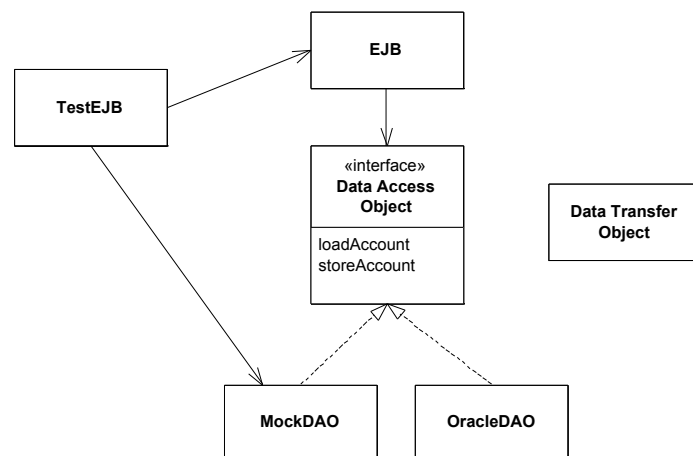
Typical usage scenario for Mock Objects in a TestCase

1. Instantiate mockobjects
2. Set up state in mockobjects, which govern their behavior
3. Set up expectations on mock objects
4. Execute the method(s) on the Unit Under Test, using the mockobjects as resources
5. Verify the results
6. Verify the expectations

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 31
Copyright 2003, Callista Enterprise AB



Example: A Mock database resource



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 32
Copyright 2003, Callista Enterprise AB





Example (contd.)

```
public void testWithdraw() {  
    EJB ejb = new EJB();  
    MockDAO mockDAO = new MockDAO();  
    mockDAO.setupLoadAccount(new ValueObject(...));  
    mockDAO.setExpectedStoreAccount(new ValueObject(...));  
    mockDAO.setExpectedLoadAccountCalls(1);  
    mockDAO.setExpectedStoreAccountCalls(1);  
    ejb.setDAO(mockDAO);  
    int result = ejb.withdraw(...);  
    assertEquals(result, expectedResult);  
    mockDAO.verify();  
}
```

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 33
Copyright 2003, Callista Enterprise AB

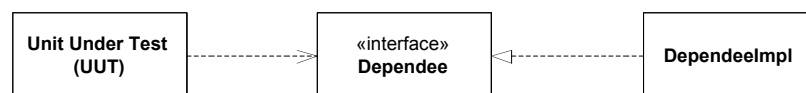


Designing for Testability : *Don't Talk To Strangers*

- If there are no strong reasons why two classes should talk to each other directly, *they shouldn't!*



becomes



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 34
Copyright 2003, Callista Enterprise AB





Designing for Testability : *Law of Demeter*

Any method should have limited knowledge about an object structure.

```
public EJBBean() {  
    ...  
    DAO dao = new DAO();  
    ...  
}
```

becomes

```
public void setDAO(DAO dao) {  
    this.dao = dao;  
}
```

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 35
Copyright 2003, Callista Enterprise AB



Bottom Line: Unit Testing and Test-First Design is Infectious!

It's always a bit painful to change your habits, but once you've been there, you're stuck!

- Enables truly iterative projects
- Improves your design
- Doesn't cost your project a fortune
- Is even fun!



Enables you to test cheap, to test early, and to test often!

CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 36
Copyright 2003, Callista Enterprise AB





Questions?



CADEC 2003-01-29, Unit Testing and Test Driven Design, Slide 37
Copyright 2003, Callista Enterprise AB

