

# **VOLVO**

## **TDD**

Introduction to Test-Driven Development

# Test-Driven Development

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

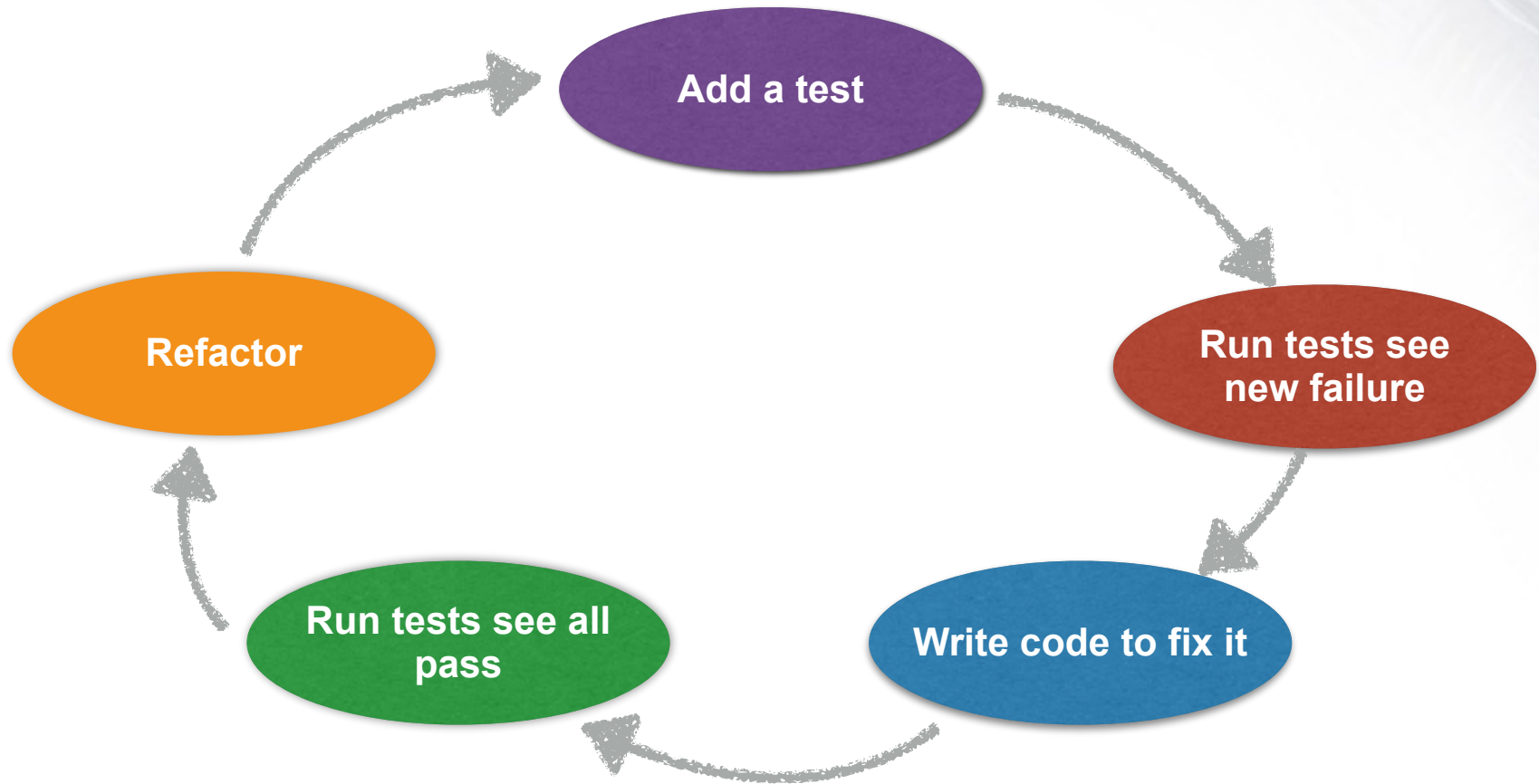
- Write a test that specifies a tiny bit of functionality
- Ensure the test fails (you haven't built the functionality yet!)
- Write the code necessary to make the test pass
- Refactoring the code to remove redundancy

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 – ...

# Test Driven Development process

1. Think about what you want to do.
2. Think about how to test it.
3. Write a small test. Think about the desired API.
4. Write just enough code to fail the test.
5. Run and watch the test fail (and you'll get the "Red Bar").
6. Write just enough code to pass the test (and pass all your previous tests).
7. Run and watch all of the tests pass (and you'll get the "Green Bar").
8. If you have any duplicate logic, or inexpressive code, **refactor** to remove duplication and increase expressiveness.
9. Run the tests again (you should still have the "Green Bar").
10. Repeat the steps above until you can't find any more tests that drive writing new code.

# Test Driven Development process (TDD process)



# Simple Design

- “Simplicity is more complicated than you think. But it’s well worth it.”  
– Ron Jeffries
- Satisfy Requirements
  - No Less
  - No More

You can use your  
developer intuition  
to find best choice

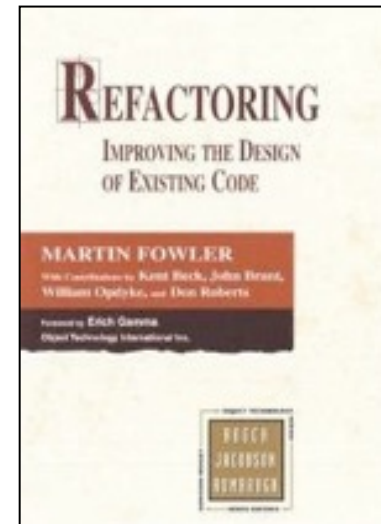


# Simple Design Criteria

- In Priority Order
  - The code is appropriate for the intended audience
  - The code passes all the tests
  - The code communicates everything it needs to
  - The code has the smallest number of classes
  - The code has the smallest number of methods

# Refactoring

- **Definition:** Improve the code without changing its functionality
- Code needs to be refined as additional requirements (tests) are added
- For more information see  
*Refactoring: Improve the Design of Existing Code* – Martin Fowler



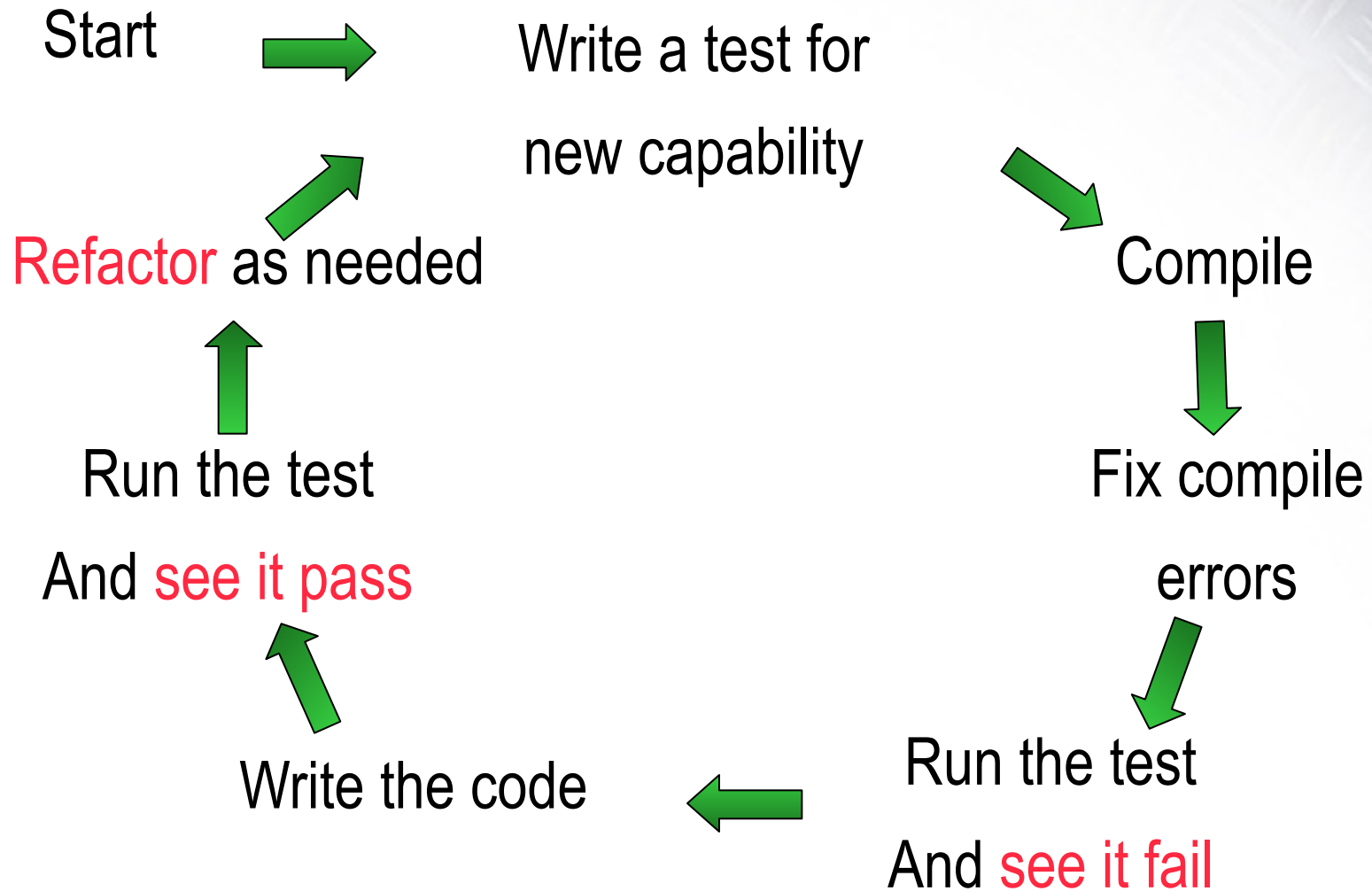
# Working Breadth First - Using a Test List

- Task Based
  - 4-8 hour duration (maximum)
- Brainstorm a list of developer tests
- Do not get hung up on completeness... you can always add more later
- Describes completion requirements





# Red/Green/Refactor

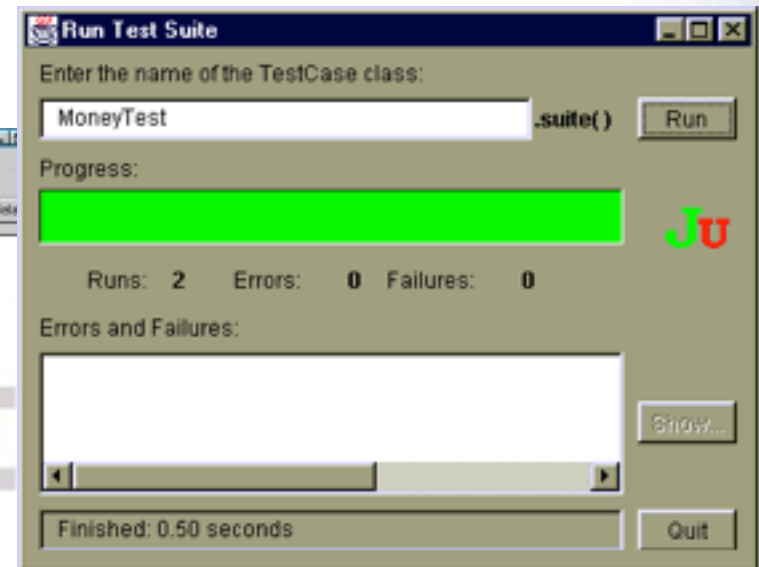
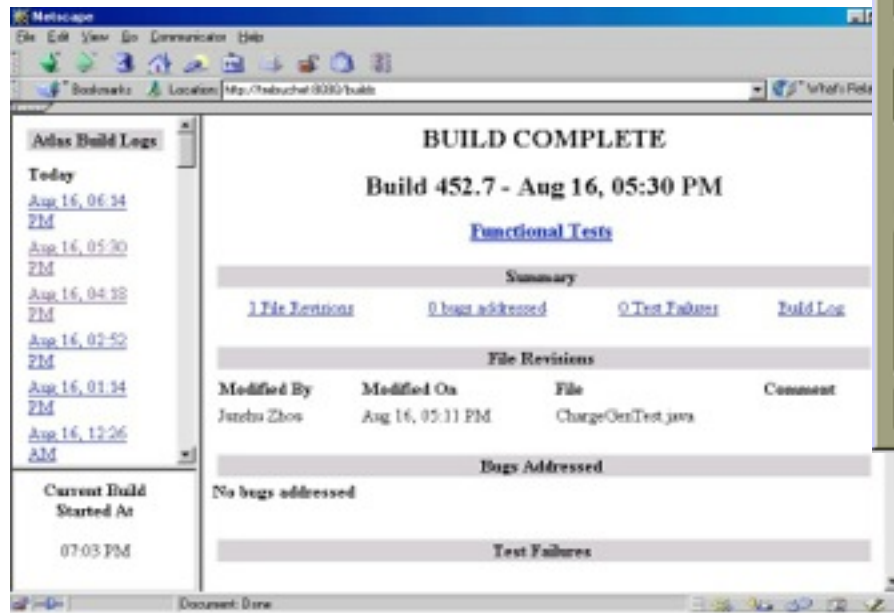


# Exercise 5

- Separate presentation, or demo

# Obvious Effects of Test-Driven Design

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



# Not-so-obvious Effects of Test-Driven Design

- Testing as we write means we spend **less time debugging**. We get our programs done faster.
- Testing as we write means that we don't have those **long testing cycles** at the end of our projects. We like working without that death march thing.
- Our tests are the first users of our code. We experience what it is like to use our code very quickly. The **design turns out better**.
- **Testing before coding** is more interesting than testing after we code. Because it's interesting, we find it easier to maintain what we know is a good practice.

# Not-so-obvious Effects of TDD (Contd.)

- 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.

# Possible week points of TDD ?

- When test code is using very intensively production API then it can **impact the ability to refactor**.
- **Buggy tests** – tests that failing because of bugs in themselves.
- This will simply not be worth to bother when **cost for maintenance** of the tests will be higher than benefits.



## Hey there!

We are developers and should strive to mitigate these week points, shouldn't we?

# Types of tests

- **Unit test:** Specify and test one point of the contract of single method of a class. This should have a very narrow and well defined scope. Complex dependencies and interactions to the outside world are stubbed or mocked.
- **Integration test:** Test the correct inter-operation of multiple subsystems. There is whole spectrum there, from testing integration between two classes, to testing integration with the production environment.
- **Smoke test:** A simple integration test where we just check that when the system under test is invoked it returns normally and does not blow up. It is an analogy with electronics, where the first test occurs when powering up a circuit: if it smokes, it's bad.
- **Regression test:** A test that was written when a bug was fixed. It ensure that this specific bug will not occur again. The full name is "non-regression test".
- **Acceptance test:** Test that a feature or use case is correctly implemented. It is similar to an integration test, but with a focus on the use case to provide rather than on the components involved.
- A **Canary test** is an automated, non-destructive test that is run on a regular basis in a LIVE environment, such that if it ever fails, something really bad has happened.
  - Examples might be:
    - Has data that should only ever be available in DEV/TEST appeared in LIVE.
    - Has a background process failed to run
    - Can a user logon
    - the concept of a canary in a coal mine.

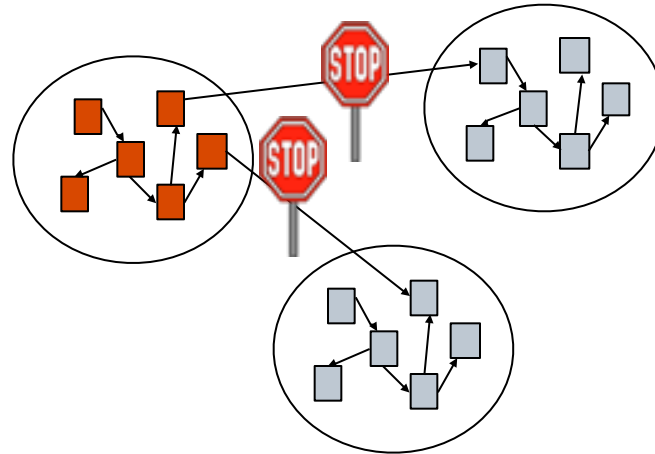
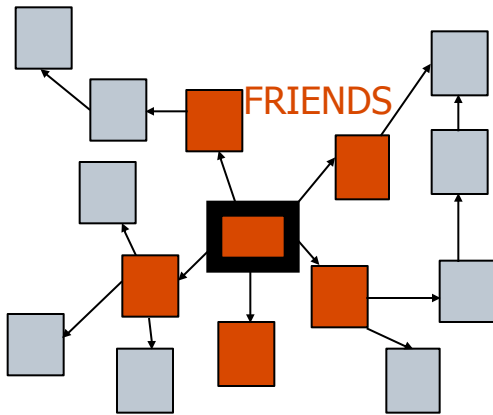
# Canary Test

- In software testing, a canary (also called a canary test) is a push of programming code changes to a small number of end users who have not volunteered to test anything. The goal of a canary test is to make sure code changes are transparent and work in a real world environment.
- Canary tests, which are often automated, are run after testing in a sandbox environment has been completed. Because the canary is only pushed to a small number of users, its impact is relatively small should the new code prove to be buggy and changes can be reversed quickly.
- The word canary was selected to describe the code push because just like canaries that were once used in coal mining to alert miners when toxic gases reached dangerous levels, end users selected for testing are unaware they are being used to provide an early warning.



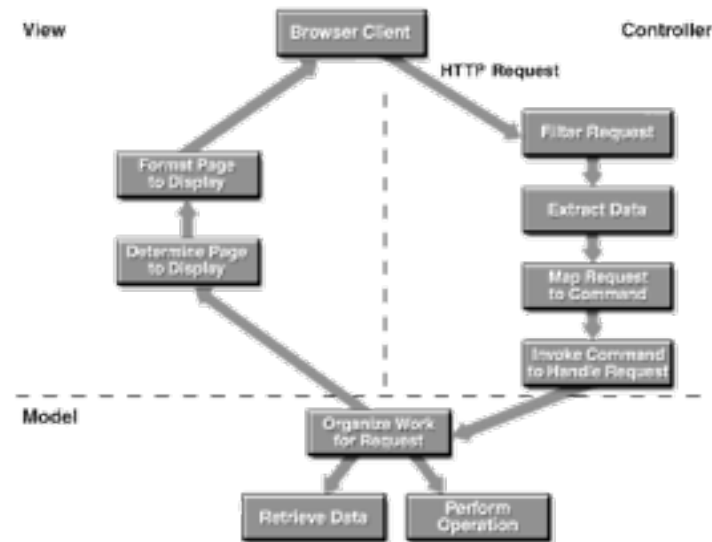
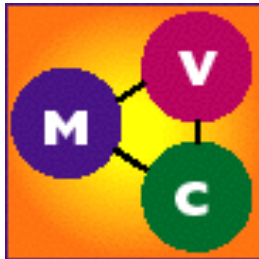
# Designing for Testability: Low Coupling

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



# 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**

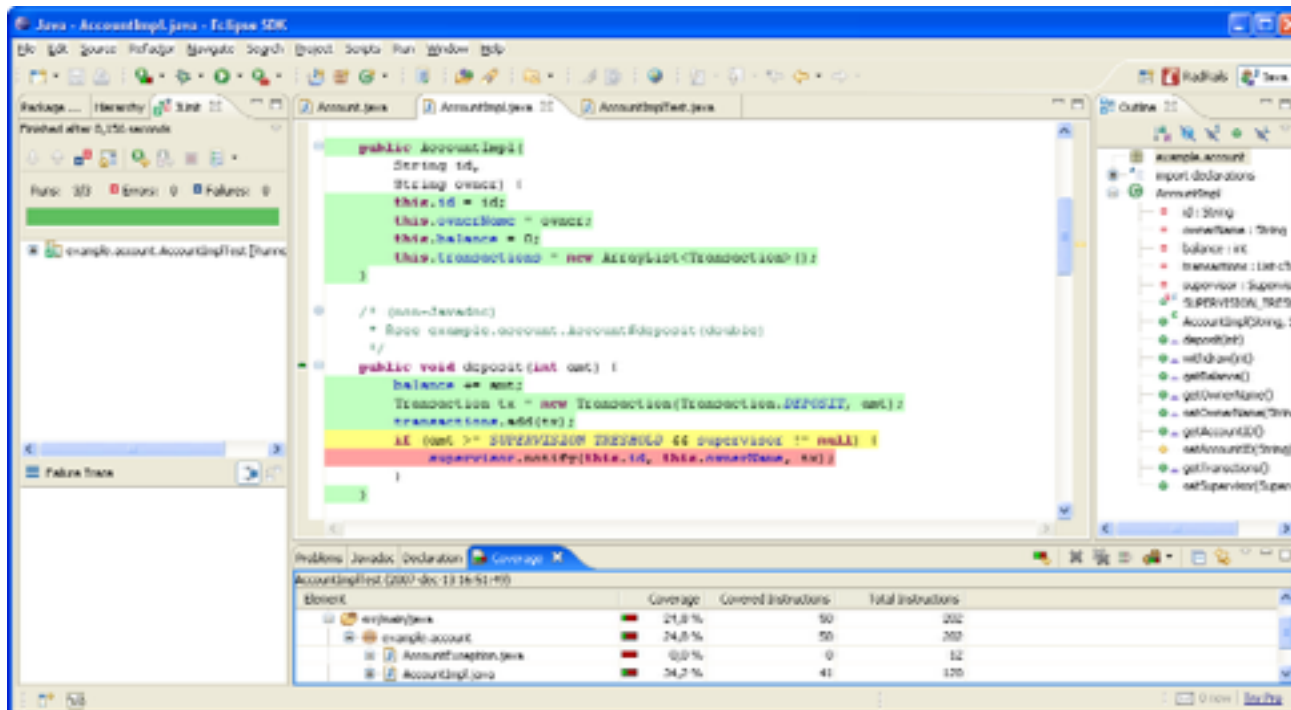


# Some quotes on Test Driven Development

- *“Test-Driven Development is a powerful way to produce well designed code with fewer defects”* – Martin Fowler
- *“The best way that I know to write code is to shape it from the beginning with tests”* – Ron Jeffries
- *“Fewer defects, less debugging, more confidence, better design, and higher productivity in my programming practice”* – Kent Beck

# Code Coverage (Java)

- Which statements of my application are being executed?
- Useful to identify incomplete testing
  - [Option 1: Install from Eclipse Marketplace Client](#)



# But ...

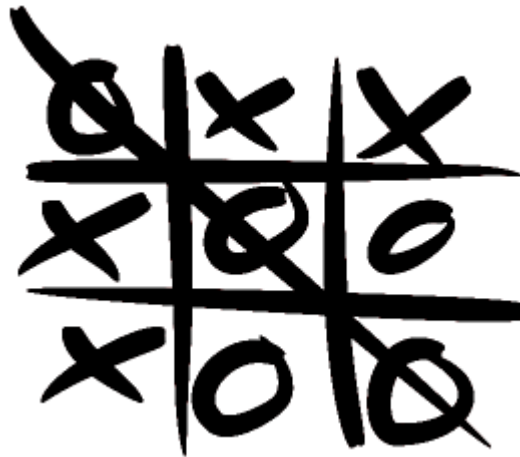
- Focusing only on coverage is not sufficient, you may miss:
  - Missing code
  - Incorrect handling of boundary conditions
  - Timing problems
  - Memory Leaks
- Use coverage sensibly
  - Objective, but incomplete
  - Too often distorts sensible action

# Exercise 6

- Run your previous tests from Exercise 5, to make sure you have adequate Code Coverage!
- Use ECLemma (bundled in the JVS Developer Tools)

## Exercise 7

Use TDD to test and implement **Naughts and Crosses** game board interface.



Details about game rules can be found here: <http://en.wikipedia.org/wiki/Tic-tac-toe>

# **VOLVO**

**TDD**

Integration Test Introduction



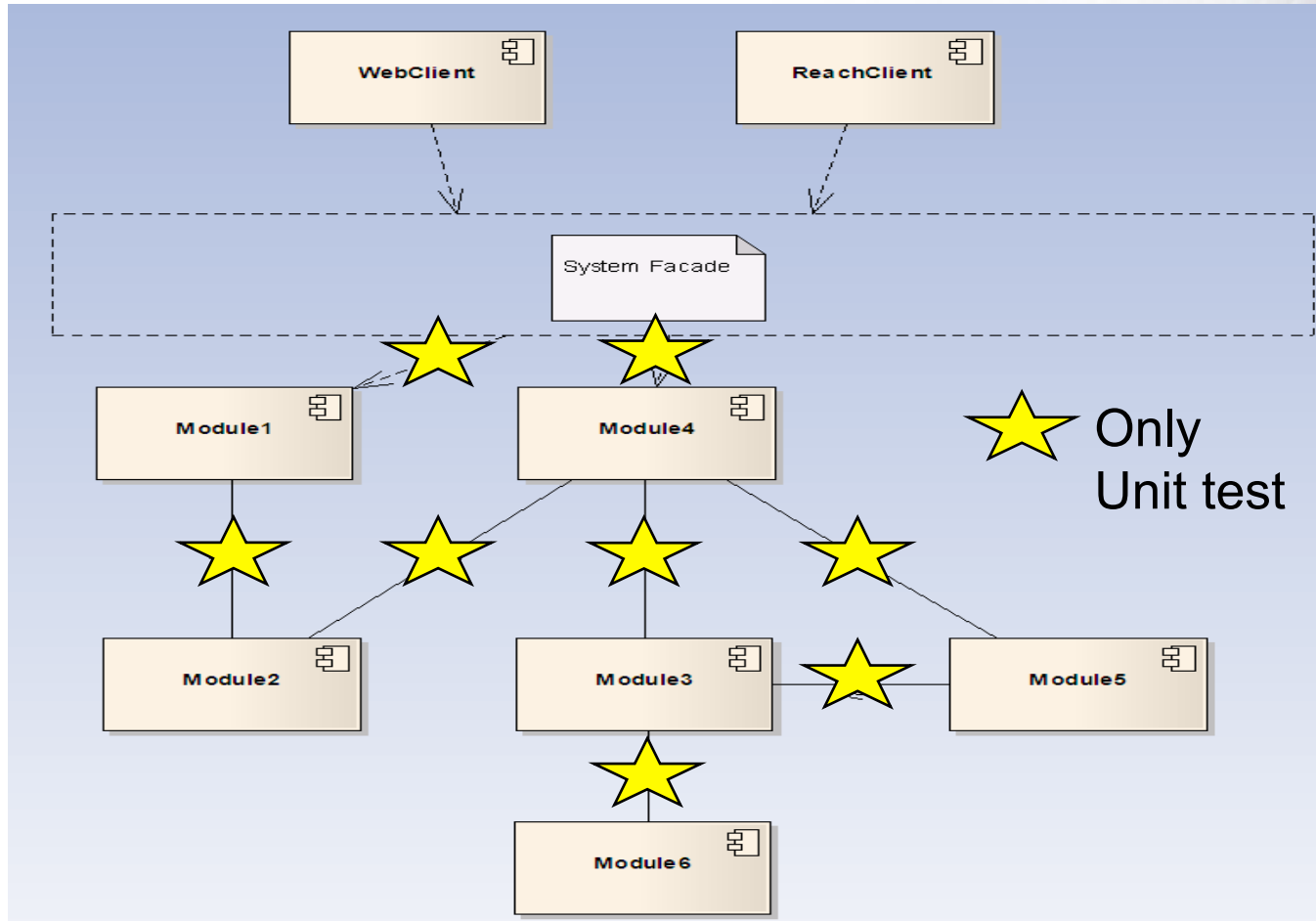
# Integration Tests

- An Integration Test is any test which tests a logical unit *together with other units that it depends on*, such as other software units but more frequently external resources such as Databases or Message Queues.
- Thus the integration tests share many of the characteristics of Unit Tests, but the *granularity* is much bigger.
- Due to the performance costs in accessing external resources, the integration tests usually takes *much longer time to execute*.

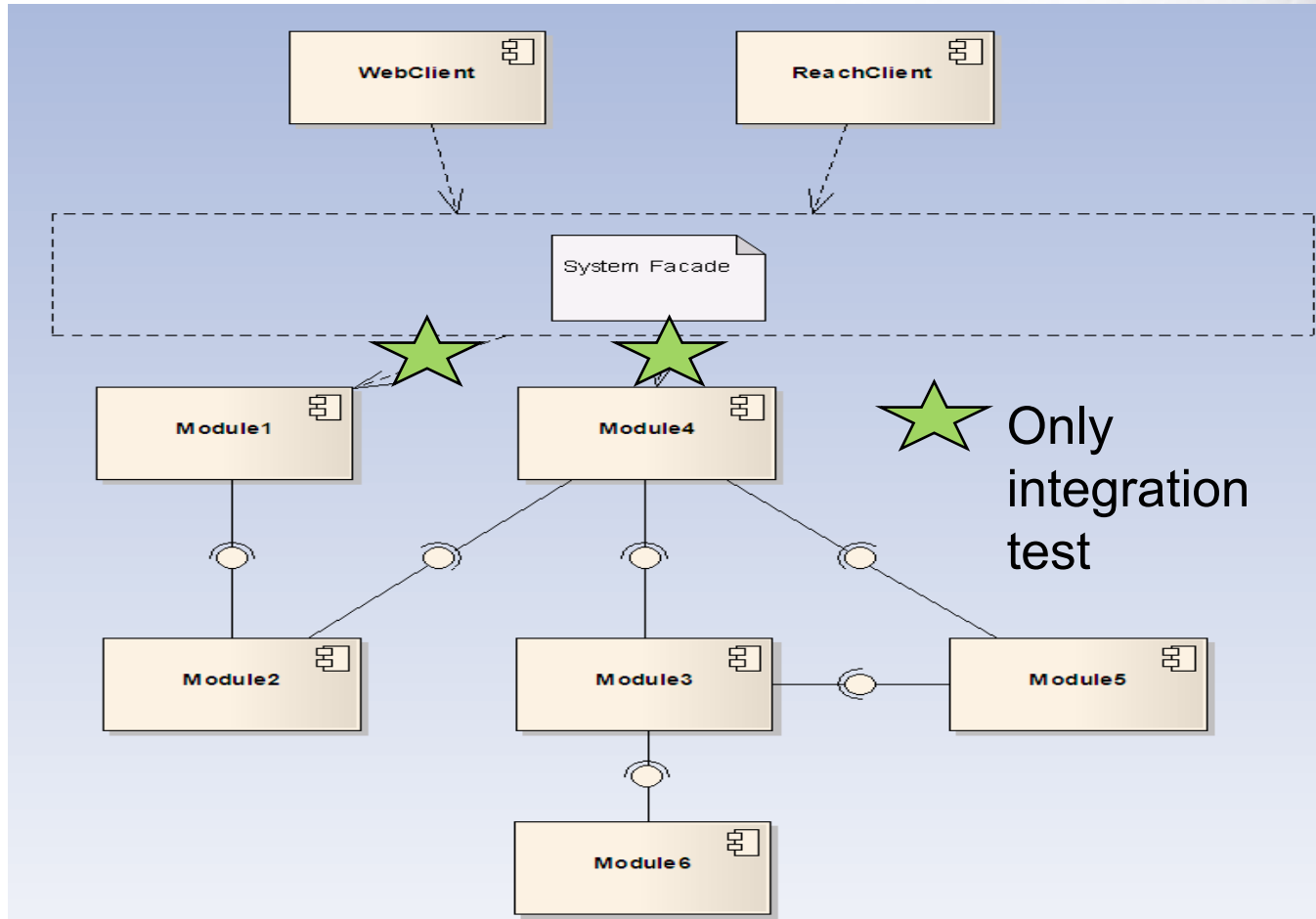
# Ratio between Unit and Integration Tests

- Unit tests are naturally the most efficient way of catching defects on the Unit level.
  - If a defect can be caught using a Unit Test, it should therefore be preferred instead of catching it using an Integration test.
  - Hence there will typically be many more Unit tests than integration tests.
- Integration tests should be used for testing interaction between units and between units and external resources.

# Ratio between Unit and Integration Tests

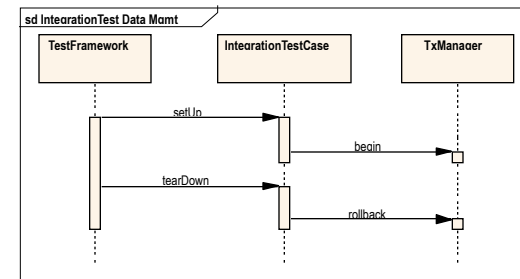


# Ratio between Unit and Integration Tests

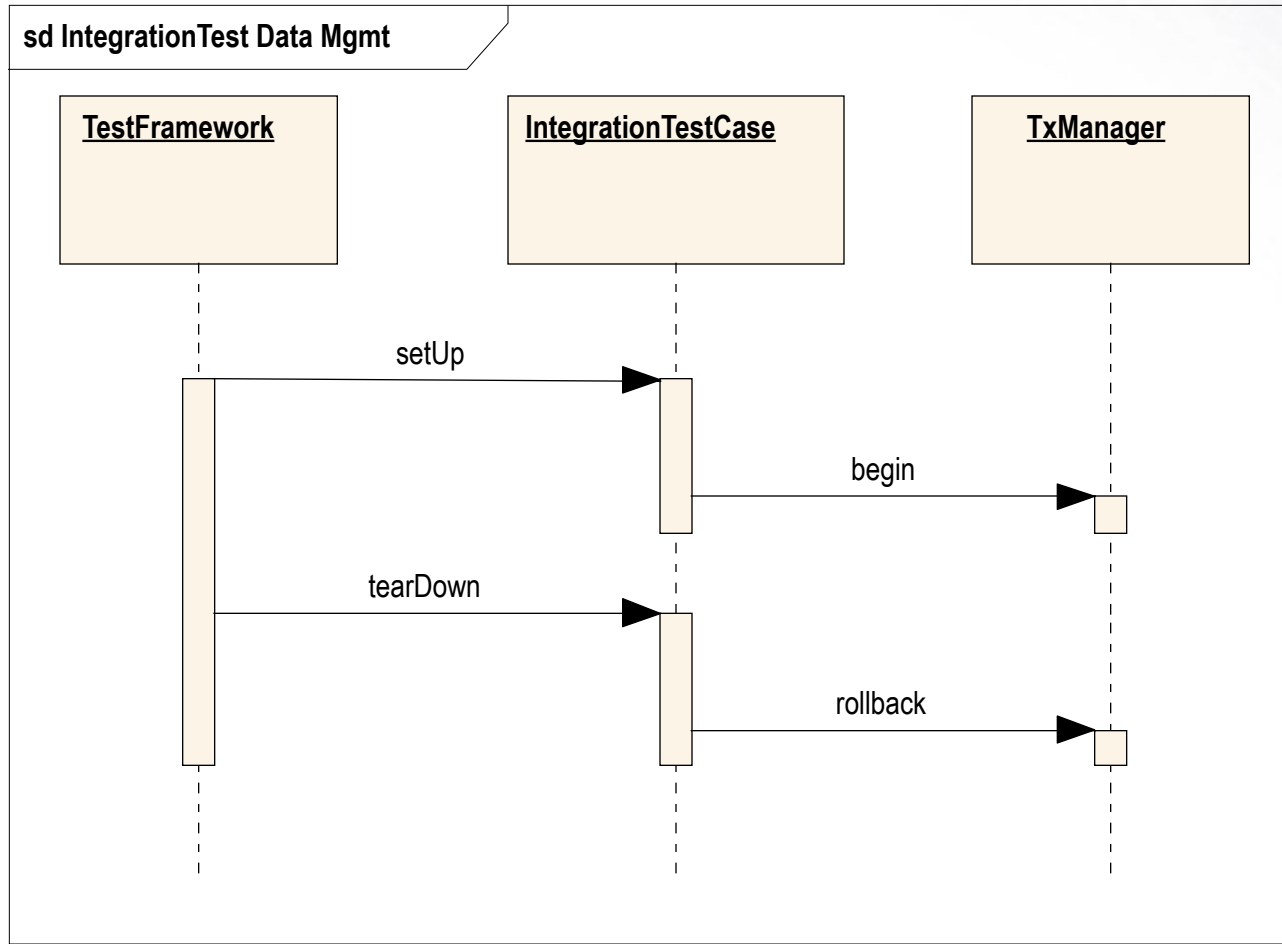


# Test Data, concurrency and repeatability

- Integration tests which have side effects (i.e. which affects persistent data) are problematic:
  - Modifying data which other tests may depend on, may cause subsequent test failures
  - Several instances of tests which uses the same data may run concurrently, which may cause test failures
- Transaction demarcation is a common idiom to protect test data from modification:
  - Start a transaction in [SetUp]
  - Rollback the transaction in [TearDown]



# Test Data, concurrency and repeatability



# Test Data strategies

- Integration tests
  - Local test data, owned and managed by test
  - Global, common test data, pre-populated via SQL scripts
- System tests
  - All data owned by test script
- Separate Databases
  - Primary keys/IDS and Test Data

# Managing External Test Data Files

- Some test data is most **easily kept in a file** format (e.g. XML files) which are read and manipulated by the tests. In order to make the tests insulated from how and where they are executed, the tests should not refer to external files via the file system. Both absolute and relative file names may differ depending on the execution environment.
- Instead, the tests may keep data files as “**Embedded Resources**” within the test assembly itself (src/test/resources).



# DbUnit



- DbUnit is a JUnit extension targeted at database-driven applications.
- Puts your database into a known state between test runs.
- DbUnit has the ability to export and import your database data to and from XML datasets.
- DbUnit is used by JVS.

# Bottom Line: Automated Testing and Test Driven 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!**