SIEMENS

Corporate Technology

Advanced Test-Driven Development

Quality Engineered Software and Testing Conference
QUEST 2009
Chicago, IL, USA

Peter Zimmerer
Principal Engineer
Siemens AG, CT SE 1
Corporate Technology
Corporate Research and Technologies
Software & Engineering, Development Techniques
81739 Munich, Germany
peter.zimmerer@siemens.com
http://www.siemens.com/research-and-development/
http://www.ct.siemens.com/

SIEMENS

Contents

SIEMENS

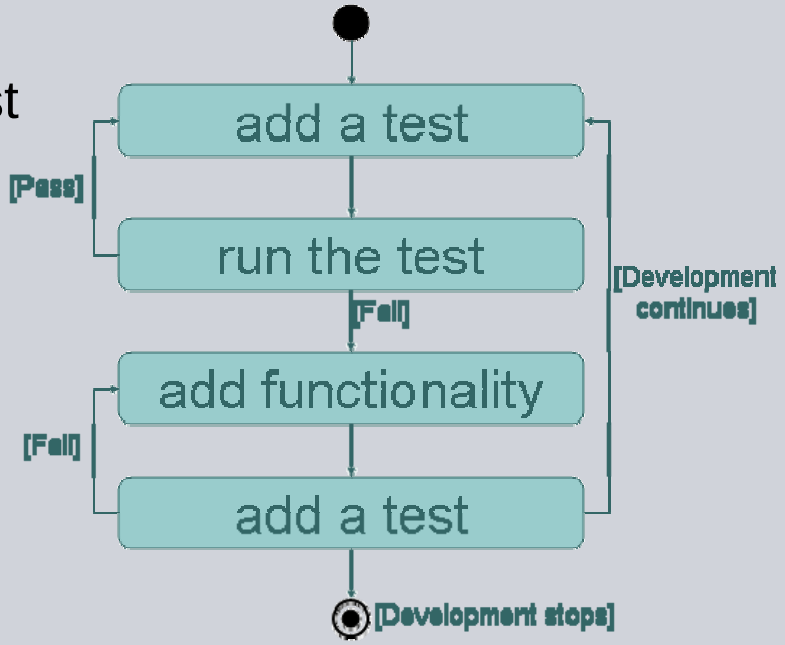
Introduction *Test-driven Development (TDD)*

Write a test | **Write the code** | **Refactor**

I'M TEST-DRIVEN

Kent Beck:

- Never write a single line of code unless you have a failing automated test
- Eliminate duplication



3 steps:

- Write a test that fails
- Write necessary code to pass the test
- Refactor the code

Related terms: test-first development, test-first programming

- TDD = test-first development ⊕ refactoring (see http://www.agiledata.org/essays/tdd.html)

SIEMENS

Typical usage of TDD

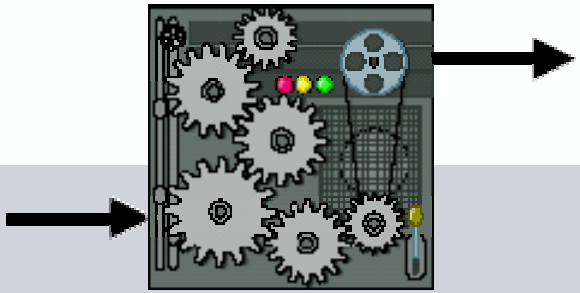
Unit testing by the programmers – very popular

- xUnit tools (http://www.xprogramming.com/software.htm)
- E.g. JUnit (http://www.junit.org/) (new version 4.0 available since 02/16/06)

Acceptance testing to enhance customer involvement

- Fit, acceptance testing framework by Ward Cunningham and others (**F**ramework for **I**ntegrated **T**est, http://fit.c2.com/)
  - Tests are specified as HTML tables (created by Excel, Word, etc.)
  - Fixtures act as the glue between the written tests and the application's code
  - Ideal for data-centric tests where each test does the same kind of thing to different kinds of data
  - Available in different languages (Java, C++, C#, Python, Perl, Ruby, …)
- FitNesse, a fully integrated standalone wiki and acceptance testing framework (based on Fit) by Robert C. Martin and Micah D. Martin (http://fitnesse.org/)
- FitLibrary, a set of fixtures and runners that extend Fit (http://sourceforge.net/projects/fitlibrary)
- Others: see http://www.xprogramming.com/software.htm (main focus is on web testing over HTTP)

Design for testability



SIEMENS

Visibility / observability

- Ability to observe the outputs, states, internals, resource usage, and other side effects of the software under test

Control(lability)

- Ability to apply inputs to the software under test or place it in specified states (for example reset to start state)

How?

- Suitable testing architecture
- Additional (scriptable) interfaces for testing purposes
- Interaction with the system under test through well-defined observation points and control points
- Coding guidelines, naming conventions
- Built-in self-test
- Consistency checks (assertions, design by contract)
- Logging and tracing, diagnosis and dump utilities (internal states)
- ***Think test-first: how can I test this?***

SIEMENS

TDD ≈ Preventive Testing (1)

Preventive testing is built upon the observation that one of the most effective ways of specifying something is to describe (in detail) how you would accept (test) it if someone gave it to you.

David Gelperin, Bill Hetzel (<1990)

Given any kind of specification for a product, the first thing to develop isn't the product, but how you'd test the product. Don’t start to build a product till you know how to test it.

Tom Gilb

The act of designing tests is one of the most effective bug preventers known.

Boris Beizer, 1983

SIEMENS

TDD ≈ Preventive Testing (2)

Use testing to discover, identify, create, specify, influence, and control requirements, architecture, design, implementation, deployment, and maintenance artifacts:
→ *Testware development leads software development*

A requirement / architecture / design / implementation / deployment cannot be accepted unless it also specifies exactly what tests will be run to check it

The idea of TDD is

- *Nothing actually new*
- *Nothing new brought to us by XP or agile methods and the hype around it*
- *Rather an old idea …*

SIEMENS

Some references

- D. Gelperin, B. Hetzel, **The Growth of Software Testing**, Communications of the ACM, Vol. 31, Issue 6, June 1988, pp. 687-695.
- Research 1: B. George and L. Williams, **A Structured Experiment of Test-Driven Development**, Information and Software Technology, Vol. 46, No. 5, 2003, pp. 337-342.
- Research 2: E. Maximilien and L. Williams, **Assessing Test-Driven Development at IBM**, Proceedings of the 25th International Conference on Software Engineering (ICSE 03), IEEE CS Press, 2003, pp. 564-569.
- Research 3: L. Williams, E. Maximilien and M. Vouk, **Test-Driven Development as a Defect-Reduction Practice**, Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 03), IEEE CS Press, 2003, pp. 34-45.
- D. Janzen and H. Saiedian, **Test-Driven Development: Concepts, Taxonomy, and Future Direction**, IEEE Computer, Vol. 38, No. 9, September 2005, pp. 43-50.
- D. Janzen and H. Saiedian, **Does Test-Driven Development Really Improve Software Design Quality?** IEEE Software, Vol. 25, No. 2, March 2008, pp. 77-84.

SIEMENS

Data on TDD research in the industry

Research (see references on previous slide)

1. controlled experiment, 3 companies, 24 programmers
2. case study, 1 company, 9 programmers
3. case study, 1 company, 9 programmers

Results: Quality – Productivity

1. 18% more tests passed – TDD took 16% longer
2. 50% reduction in defect density – minimal impact
3. 40% reduction in defect density – no change

In two studies, programmers were new to TDD.
TDD seems to improve quality without damage for productivity.
The productivity gap for the first experiment is explained by the fact that the other group wrote far fewer tests than the TDD group.

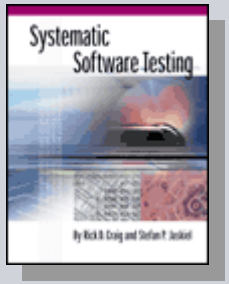
See also summary of selected empirical studies of test-driven development in IEEE Software, Vol. 24, No. 3, May / June 2007, pp. 24-30.

SIEMENS

TDD – Example: Requirements

Automated teller machine (ATM):

A valid user must be able to withdraw up to $200 or the maximum amount in the account.



R. Craig, S.P. Jaskiel
Systematic Software Testing

The first two high level test cases

TC1: Withdraw $200 from an account with $165 in it. Result ???

TC2: Withdraw $168.46 from an account with $200 in it. Result ???

already help us to discover two ambiguities in the requirements:

Some people will interpret it to mean that the ATM user can withdraw the lesser of the two values ($165), while other people will interpret it to mean they can withdraw the greater of the two values ($200).

Does the bank want the ATM to dispense coins to the users???

…so already help us to identify / create essential additional requirements!

SIEMENS

Example: Non-functional requirements (NFR)

Often it seems that NFR are difficult to specify well
→ This means that NFR are also difficult to test

We can see the cost when we get it wrong
- The costs of building in NFRs and testing NFR late in a project
- Many examples where getting NFR and/or testing NFR wrong has been disastrous

NFRs are the prime driver for the architecture
- If we decide to test the architecture, we have to test NFR

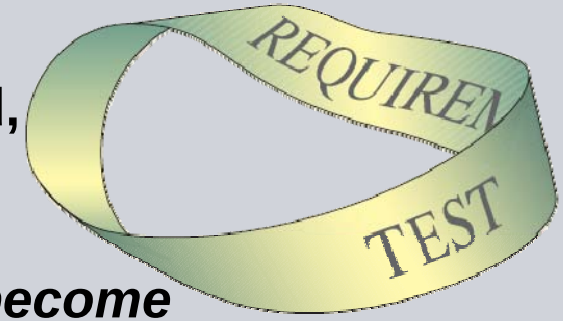
Early tests for NFR will influence and change the architecture to be developed → TDD

SIEMENS

TDD as a requirements engineering technique

Surprisingly, to some people, one of the most effective ways of testing requirements is with test cases very much like those for testing the completed system.
… write these tests when gathering, analyzing, and verifying requirements – long before those requirements are coded

Donald C. Gause and Gerald M. Weinberg, 1989



Writing requirements and testing are interrelated, much like the two sides of a Möbius strip.
Equivalence hypothesis:
As formality increases, tests and requirements become indistinguishable. At the limit, tests and requirements are equivalent.

Robert C. Martin, Grigori Melnik, 2008

IEEE Software, Vol. 25, No. 2, Jan/Feb 2008, pp. 54-59

SIEMENS

In contrast – TDD is *not* …

Step 1: Create UML class diagrams

Step 2: Generate "header" files with all interfaces (method signatures)

Step 3: Implement (unit) tests

Step 4: Implement "body" files of system under test (implementation of methods)

***Real* TDD means to let testing drive and influence your architecture and design: For example creation of test cases to identify and specify required interfaces, parameter types, etc. in the UML model.**

Lesson 1 – Preventive testing: Comprehensive view of TDD

TDD = Test-first *design* ⊕ test-first *implementation* ⊕ refactoring

- Including early creation of abstract non-executable test cases as well as detailed implemented and executable test cases

TDD is possible and strictly recommended on *every* test level, *not* only for unit and acceptance testing (→ *preventive testing*)!
Emphasizes the importance and benefits of early testing activities.

- *Building the test specification is testing*
- Tests represent a set of *executable specifications*
- Proactive design for testability

These things are NOT actually new and are already well known for a long time.
What‘s new is that these things are more and more really used in projects today. From my point of view that‘s the real big benefit of the *TDD-hype brought to us by XP and agile methods*.

Lesson 2 – Test-first implementation

Not always 100% possible in real life

- Usage can be costly and time-consuming dependent on what test environment (e.g. by creating and maintaining mock objects) is needed
- Legacy code with low testability
- GUI testing (e.g. Java Swing, capture/replay tools)
- Web applications (using ASP.NET, JSP, servlets, Ajax):
  Tests could be created to check the HTML output of the code, but that doesn't really test that the HTML code itself is properly displayed within the browser.
- Distributed objects (e.g. EJB) deployed on application servers
- Code running on different types of machines and interacting with a complex environment: e.g. communication servers, middleware servers, database servers, content management systems, web interfaces, etc.
- Event-based reactive systems, multi-threaded applications
- Embedded systems

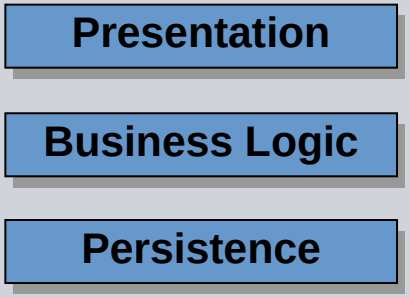
SIEMENS

TDD strategy for GUI testing

The more code you can make testable the more reliable the system will be.
Divide the code into appropriate components that can be built, tested, and deployed separately.

Build most of the functionality (business logic, services) outside the context of the user interface code using TDD.
Let the user interface code be just a very thin layer on top of rigorously tested code. I.e. build as much functionality as possible outside the GUI.

Again, this "good" basic architecture style of a clear separation between business logic and user interface is NOT new and is already well known for a long time: e.g. 3-tier architecture.



Lesson 3 – TDD and innovation (1)

Not always 100% possible in real life
Is it an enemy of innovation and invention?

- Example:
  Invention of the car
  by Carl Benz and Gottlieb Daimler
  in 1886

Only 70 years later:
Invention of the crash test
by Mercedes-Benz in the fifties.
Systematic crash tests
since 1959.

Lesson 3 – TDD and innovation (2)

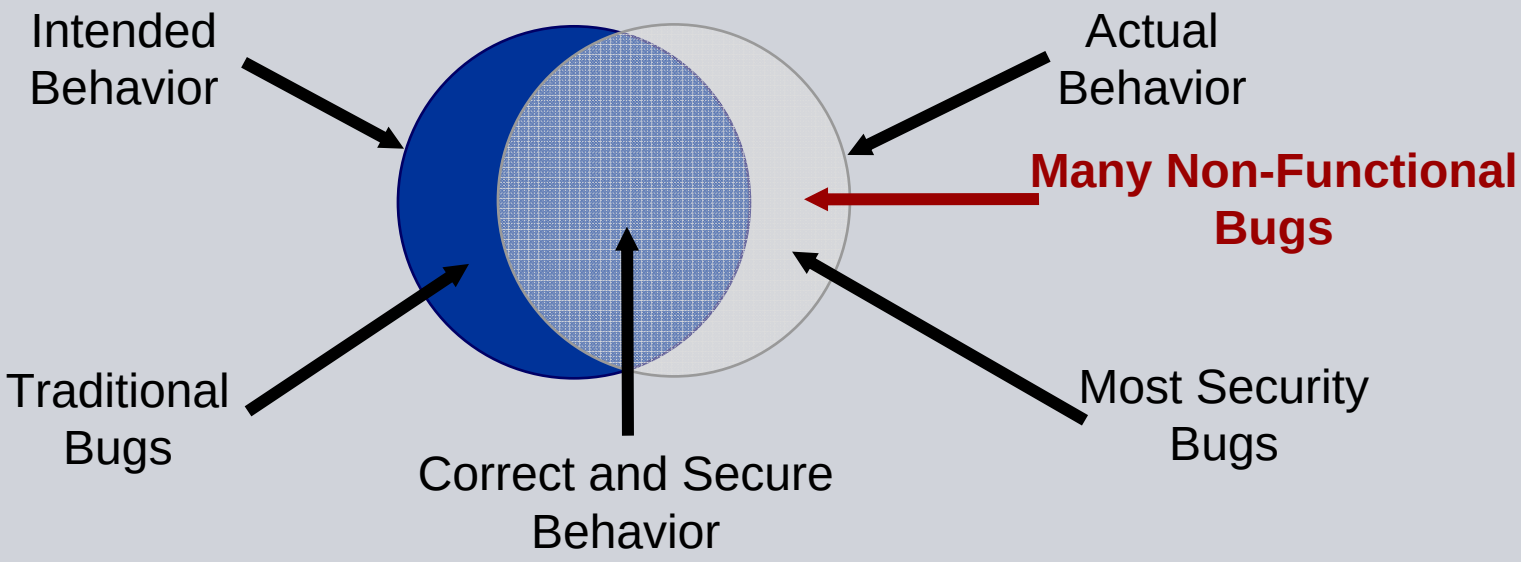
Not always 100% possible in real life

- Example **Software technologies (e.g. object-orientation (OO), web technologies, aspect-oriented programming (AOP), grid computing):**
  When looking back in history we can see that first there was the idea, vision and invention of these new software technologies, then later people thought about needed strategies, methods, and tools for testing them.
- Example **Architectural and design patterns:**
  These patterns claim to contain innovative approved best practices.
  Here again the patterns have been invented first (or some have been reinvented based on already known knowledge). Later people started to think about how to test a specific design pattern, i.e. also design patterns have not been developed in a core TDD manner.
- Example **Testing tools:**
  What's about all the innovative testing tools we get from the commercial testing tool industry? Do you think these tools are usually developed in a TDD manner? I don't …

Lesson 4 – TDD and non-functional requirements

Not always 100% possible in real life

- Non-functional requirements: performance, usability, etc.
- Example: Security (Ref. James Whittaker)



SIEMENS

Lesson 5 – Cost efficiency and predictability

Think about cost efficiency again …

- What about continuously changing early requirements and architectural prototypes → rework in testing?
- Find the right balance in your project …

Not enough in real life

- Test cases created using a test-first approach or generated from (always incomplete) requirements / design are never enough
- Developers tend to think in terms of "happy day" scenarios; typically invalid test cases are left out
- Do not miss the investigative mission of testing (deliver information)
- You cannot predict everything
  → use approaches like exploratory testing as well
- Decisions made during implementation won't be well-tested by tests exclusively created upfront

SIEMENS

Experiences (1)

TDD increases visibility and importance of testing.
TDD needs changes in development:
process, people (including management!), tooling.
TDD results in a closer cooperation of testers and developers.

Question from developers:
If doing test-first can I skip to write specifications anymore?

Sorry, no.
You need to specify the big picture in some way.
The specifications will be different if going the TDD way.
Specifications will have higher quality and will be testable at all.
Specifications will be more realistic in terms of what can be built.

SIEMENS

Experiences (2)

Question from developers:

Why can't I write my unit tests concurrently or right after my code?

From experience we know that often this is NOT done, i.e. really doing this takes even more discipline than writing the tests first

If testability is not designed into the code then some things will be considered "not testable (especially with unit tests)"

Completely ignore the evolutionary design benefits

Tests won't be as good: not writing tests that *lead the developer to the result* but only testing to show *what is there / has been done*

SIEMENS

Experiences (3)

Questions from developers:

Why do we need 99% code coverage?
Why is 60% (code) coverage not enough?
Which level of (code) coverage is required?

If I always write adequate tests first and then only the necessary piece of implementation code (using the simple design approach) then I will automatically get a very high level of (code) coverage. Otherwise it seems that the used TDD approach can be improved …

SIEMENS

Experiences (4)

Question from developers:

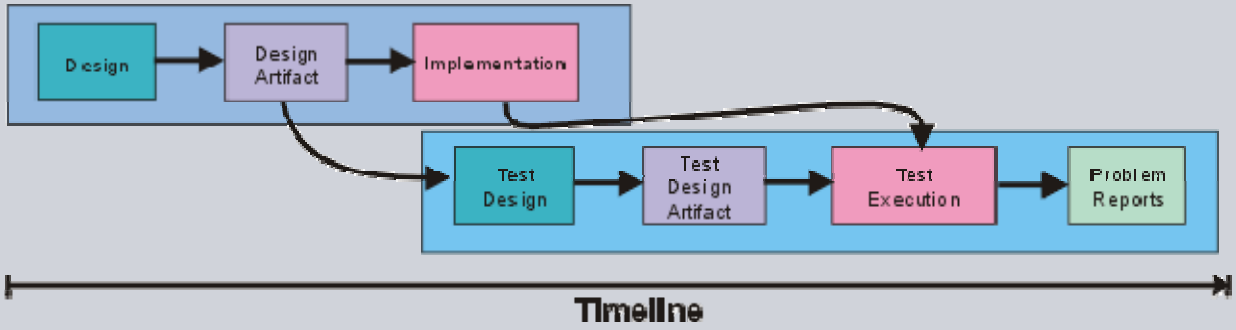
How do I test private member functions?

From a technical point there are different answers dependent on the used programming language (e.g. friends in C++, reflection in Java)

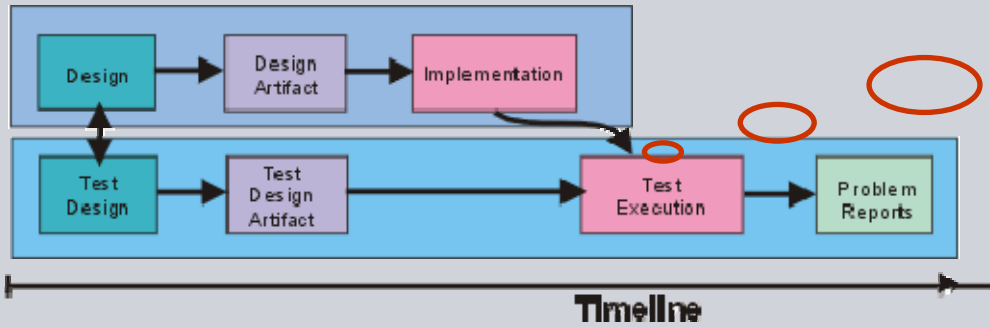
In core TDD this question is NOT allowed, i.e. it does not make sense, because in TDD we first have the designed and implemented test and then do some implementation for this test.
I.e. the details of the implementation do not matter so much …

TDD – Changes (Ref. IBM Rational Unified Process)

Traditional



Test-first design

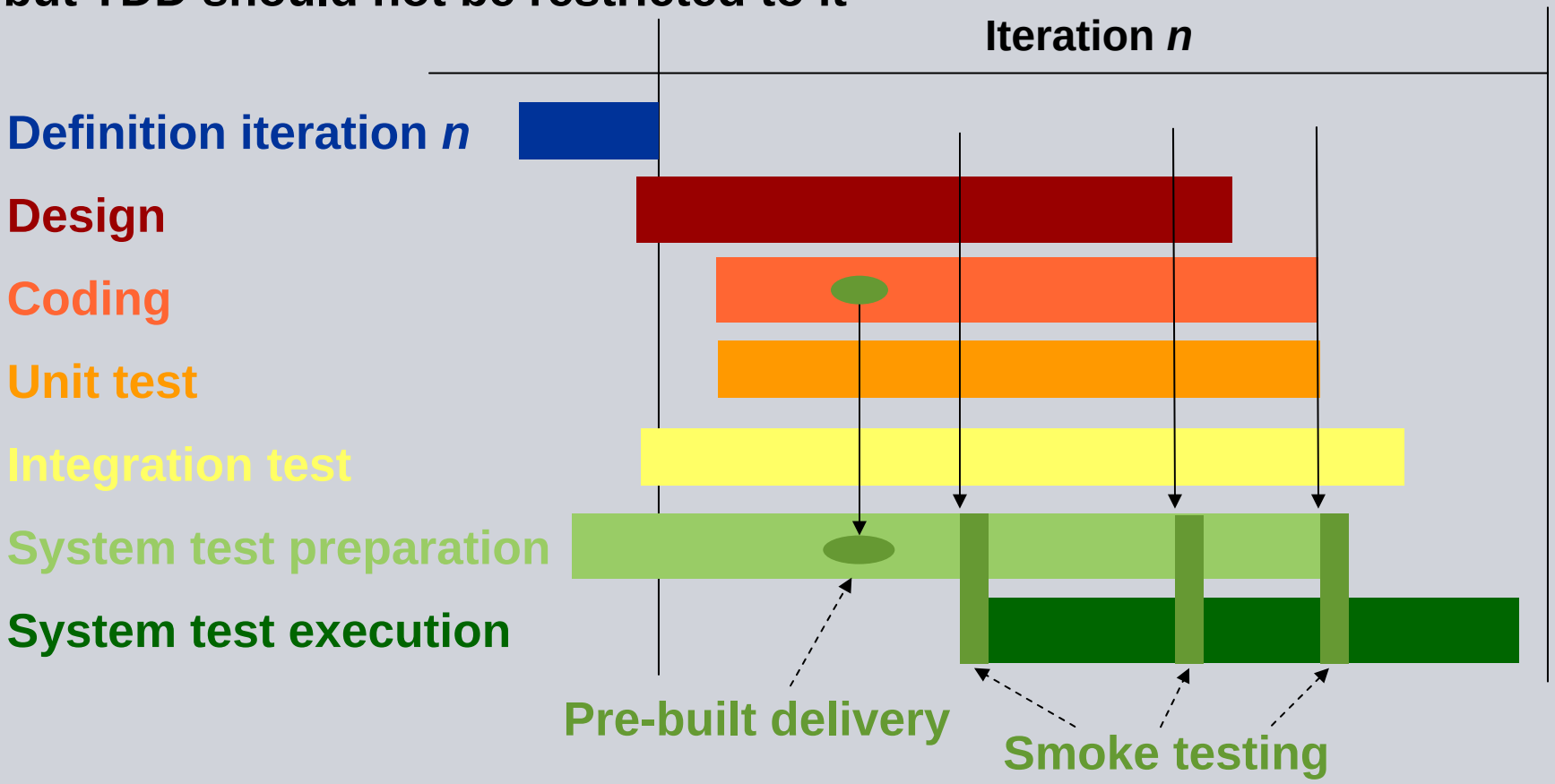


But, why does the test execution happen so late …?

SIEMENS

Example for a test workflow visualizing TDD

Test-driven development is often used in the context of an iterative / incremental, agile development process but TDD should not be restricted to it



SIEMENS

Summary

TDD = Test-first design ⊕ test-first implementation ⊕ refactoring

TDD is possible and strictly recommended on every test level, not only for unit and acceptance testing (→ *preventive testing*):
Let testing drive your development and maintenance at all!

TDD needs changes in development: process, people, tooling.
TDD results in a closer cooperation of testers and developers.

TDD is neither 100% possible nor sufficient in real-world projects.
TDD does not completely replace conventional "afterwards" software testing: so, do test-first as well as *test-second*.

The right project specific balance is the key for cost efficiency.

If not already done then start with TDD tomorrow!!!

