Evolve Design for Testability to the Next Level

Peter Zimmerer, Siemens AG
Evolve Design for Testability to the Next Level

EuroSTAR 2012
Amsterdam, The Netherlands
November 8, 2012

Peter Zimmerer
Principal Engineer
Siemens AG
Corporate Technology
81739 Munich, Germany
peter.zimmerer@siemens.com
http://www.siemens.com/corporate-technology/
Contents

- What is testability all about?
- What are the influencing factors and constraints of testability?

- Why is testability so important?
- Who is responsible and who must be involved to realize design for testability?

- How can we improve, foster, and sustain the testability of a system?
- How can we innovate and renovate design for testability in our projects?

→ A new, comprehensive strategy on design for testability over the whole lifecycle of a system
Testability – Theoretical definitions

Testability is the degree to which, and ease with which, software can be effectively tested.


The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.


The testability of a program P is the probability that if P contains faults, P will fail under test.


The degree to which an objective and feasible test can be designed to determine whether a requirement is met.

ISO/IEC 12207

The capability of the software product to enable modified software to be validated.

ISO/IEC 9126-1

Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

ISO/IEC 25010, adapted from ISO/IEC/IEEE 24765
Testability – Factors (1)

Control(lability)
- *The better we can control the system, the more and better testing can be done, automated, and optimized*
- Ability to apply and control the inputs to the system under test or place it in specified states (for example reset to start state, roll back)
- Interaction with the system under test (SUT) through control points

Visibility / observability
- *What you see is what can be tested*
- Ability to observe the inputs, outputs, states, internals, error conditions, resource utilization, and other side effects of the system under test
- Interaction with the system under test (SUT) through observation points
Testability – Factors (2)

Availability, operability
- The better it works, the more efficiently it can be tested
- Bugs add overhead for analysis and reporting to testing
- No bugs block the execution of the tests

Simplicity, consistency, and decomposability
- The less there is to test, the more quickly we can test it
- Standards, (code / design) guidelines, naming conventions
- Modules can be tested independently, loose coupling
- Internal software quality (architecture, design, code), technical debt

Stability
- The fewer the changes, the fewer the disruptions to testing
- Changes are infrequent, controlled and do not invalidate existing tests
- Software recovers well from failures

Understandability, knowledge (of expected results)
- The more (and better) information we have, the smarter we will test
- Design is well understood, good and accurate technical documentation

@Microsoft: SOCK – Simplicity, Observability, Control, Knowledge
Testability – Factors (3)

- Control(lability)
- Question
- Availability
  - Operability
  - Stability
- Answer
- Visibility / Observability
- Oracle
  - Simplicity
  - Understandability
  - Knowledge
Testability – Constraints

How we design and build the system affects testability

Testability is the key to cost-effective test automation
Testability is often a better investment than automation

- Test environment: stubs, mocks, fakes, dummies, spies
- Test oracle: assertions (design by contract)
  → Implemented either inside the SUT or inside the test automation

Obstacles for testability

- Size, complexity, structure, variants, 3rd party components, legacy code
- You cannot log every interface (huge amount of data)
- Conflicts with other non-functional requirements like encapsulation, performance or security
- Non-determinism: concurrency, threading, race conditions, timeouts, message latency, shared and unprotected data
Design for Testability (DfT) – Why?

Testability ≈ How easy / effective / efficient / expensive is it to test? Can it be tested at all?

Increase depth and quality of tests – Reduce cost, effort, time of tests
- More bugs are detected (earlier) with less effort
- Support testing of error handling and error conditions, for example to test how a component reacts to corrupted data
- Testing of non-functional requirements, test automation, regression testing
- Cloud testing, testing in production (TiP)

Reduce cost, effort, and time for debugging, diagnosis, maintenance
- Typically underestimated by managers, not easy to measure honestly

Provide building blocks for self-diagnosing / self-correcting software

Possible savings ~10% of total development budget
(Stefan Jungmayr, http://www.testbarkeit.de/)

Simplify, accelerate, and power up the testing
Focus the testing on the real stuff ... and have more fun in testing ...
Design for Testability (DfT) – Who?

Joint venture between architects (+ developers) and testers
- Collaboration between different stakeholders
- Requires accountability and clear ownership

Testability must be built-in by architects (+ developers)
- Pro-active strategy:
  design the system with testability as a key design criterion
- Testers (test automation engineers) must define testability requirements to enable effective and efficient test automation

Contractual agreement between design and test

Balancing production code and test code
- Production code and test code must fit together
- Manage the design of production code and test code as codependent assets

Educate stakeholders on benefit of DfT
Design for Testability (DfT) – How?

Suitable testing architecture, good design principles

Interaction with the system under test through well-defined **control** points and **observation** points

Additional (scriptable) interfaces, hooks, mocks, interceptors for testing purposes (setup, configuration, simulation, modification)

Coding guidelines, naming conventions

Internal software quality (architecture, code)

Built-in self-test (BIST), built-in test (BIT)

Consistency checks (assertions, design by contract)

Logging and tracing

Diagnosis and dump utilities (internal states, resource utilization)

**Think test-first (TDD): how can I test this?**
Strategy for Design for Testability (DfT) over the lifecycle (1)

- Consistently defined and well-known in the project to drive a common understanding, awareness, and collaboration with clear accountability

- Addressed and specified as one important non-functional requirement from the beginning

- Investigated by elicitation techniques (interviews, observation, brainstorming, story writing, prototyping, personas, reuse, apprenticing)

- Included and elaborated in the product backlog

- Defined, specified, traced, and updated as a risk item in the risk-based testing strategy with well-defined mitigation tasks

- Balanced with regards to conflicting non-functional requirements

- Implemented dependent on the specific domain, system, and (software) technologies that are available

Educate stakeholders on benefit of DfT
Strategy for Design for Testability (DfT) over the lifecycle (2)

- Specified and elaborated in a testability guideline for architects, developers, testers (\( \rightarrow \) DfT – How?)
  - Testability view*/profile in the architectural design approach (ADA)
    - Description how to recognize characteristics of the ADA in an artifact to support an analysis and check of the claimed ADA
    - A set of controls (controllables) associated with using the ADA
    - A set of observables associated with using the ADA
    - A set of testing firewalls within the architecture used for regression testing (firewalls enclose the set of parts that must be retested, they are imaginary boundaries that limits retesting of impacted parts for modified software)
    - A set of preventive principles (e.g. memory / time partitioning): specific system behavior and properties guaranteed by design (e.g. scheduling, interrupt handling, concurrency, exception handling, error management)
    - A fault model for the ADA: failures that can occur, failures that cannot occur
    - Analysis (architecture-based or code-based) corresponding to the fault model to detect ADA-related faults in the system (e.g. model checking)
    - Tests made redundant or de-prioritized based on fault model and analysis

*Additional view to the 4+1 View Model of Software Architecture by Philippe Kruchten
Strategy for Design for Testability (DfT) over the lifecycle (3)

- Included as one important criteria in milestones and quality gates, e.g.:
  - Check **control** and **observations** points in the architecture
  - Check testability requirements for sustaining test automation
  - Check logging and tracing concept to provide valuable information
  - Check testability support in coding guidelines
  - Check testability needs (especially **visibility**) for regression testing

- Investigated and explored in static testing:
  architecture interrogation (interviews, interactive workshops),
  architecture reviews, QAW, ATAM*, code reviews

- Investigated and explored in dynamic testing:
  special testability tour in scenario testing (application touring)

➔ *Neglecting testability means increasing technical debt … $$ …*

*Architecture-centric methods from SEI, see [http://www.sei.cmu.edu/](http://www.sei.cmu.edu/)*
Summary – Testability and Design for Testability (DfT)

- What?
- Why?
- Who?
- How?

→ Strategy for Design for Testability (DfT) over the lifecycle