SIEMENS

Ingenuity for life

PLM and Innovation Excellence

Learning Campus

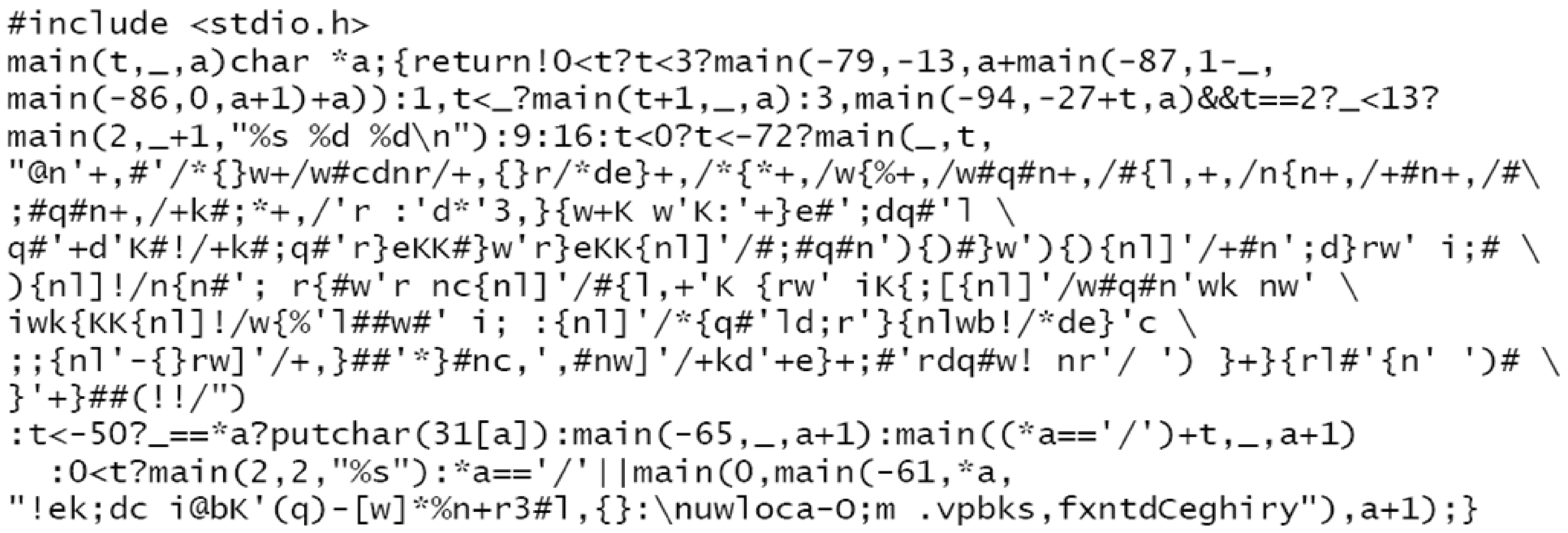
Your partner for Business Learning

Siemens Core Learning Program

Design for Testability (DfT)

Author:
Peter Zimmerer, CT

Example code – Twelve Days of Christmas V1



```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Reference: Paul Anderson, Gramma Tech, StarEAST Conference, 2008

Example code – Twelve Days of Christmas V2

```
int main()
{
    char *int_to_word[] = { "",
                            "first", "second", "third", "fourth",
                            "fifth", "sixth", "seventh", "eighth",
                            "ninth", "tenth", "eleventh","twelfth" };
    int i;
    for (i=1 ; i<=12 ; i++)
    {
        printf("On the %s day of Christmas my true love gave to me\n",
               int_to_word[i]);

        switch (i)
        {
          case 12: printf("twelve drummers drumming, ");
          case 11: printf("eleven pipers piping, ");
          case 10: printf("ten lords a-leaping,\n");
          case 9: printf("nine ladies dancing, ");
          case 8: printf("eight maids a-milking, ");
          case 7: printf("seven swans a-swimming, \n");
          case 6: printf("six geese a-laying, ");
          case 5: printf("five gold rings;\n");
          case 4: printf("four calling birds, ");
          case 3: printf("three french hens, ");
          case 2: printf("two turtle doves\nand ");
          case 1: printf("a partridge in a pear tree.\n\n");
        }
    }
    return 0;
}
```

A story on missing testability …

The **Therac-25** was a radiation therapy machine produced by Atomic Energy of Canada Limited (AECL) after the Therac-6 and Therac-20 units. It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation, approximately 100 times the intended dose. These accidents highlighted the dangers of software control of safety-critical systems, and they have become a standard case study in health informatics and software engineering.

A commission concluded that the primary reason should be attributed to the bad software design and development practices, and not explicitly to several coding errors that were found. ***In particular, the software was designed so that it was realistically impossible to test it in a clean automated way.***

http://en.wikipedia.org/wiki/Therac-25

Design for testability

Learning objectives

- Understand design for testability: what, why, who, how?
- Learn how to drive design for testability over the whole lifecycle

SIEMENS
Ingenuity for life

Design for Testability

Agenda

SIEMENS
Ingenuity for life

Testability – Theoretical definitions (1)

Testability is the ease of validation, that the software meets the requirements. Testability is broken down into: accountability, accessibility, communicativeness, self-descriptiveness, structuredness.

Barry W. Boehm: Software Engineering Economics, Prentice Hall,1981

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

Donald G. Firesmith: *Testing Object-Oriented Software*, Proceedings of the Object EXPO Europe, London (UK), July 1993

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.

IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY, 1990

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

Dick Hamlet, Jeffrey Voas: *Faults on its Sleeve. Amplifying Software Reliability Testing*, in Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA), Cambridge, Massachusetts, USA, June 28–30, 1993

Testability – Theoretical definitions (2)

Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing.

Len Bass, Paul Clement, Rick Kazmann: *Software Architecture in Practice*

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.

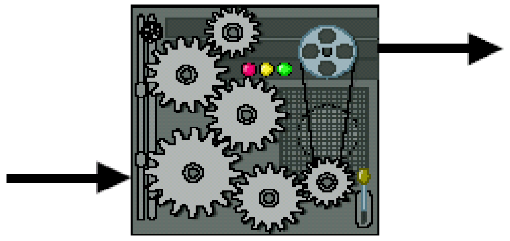
ISO/IEC 9126-1

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

SIEMENS
Ingenuity for life

Testability – Factors (1)



Control(lability)

- *The better we can control the system (in isolation), 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

Control(lability)

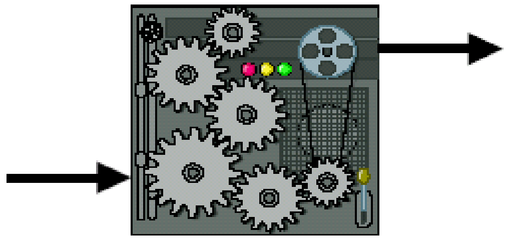
Visibility / observability

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

SIEMENS
Ingenuity for life

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
- Layering, modularization, isolation, loose coupling, separation of concerns, SOLID
- Internal software quality (architecture, design, code), technical debt

Stability

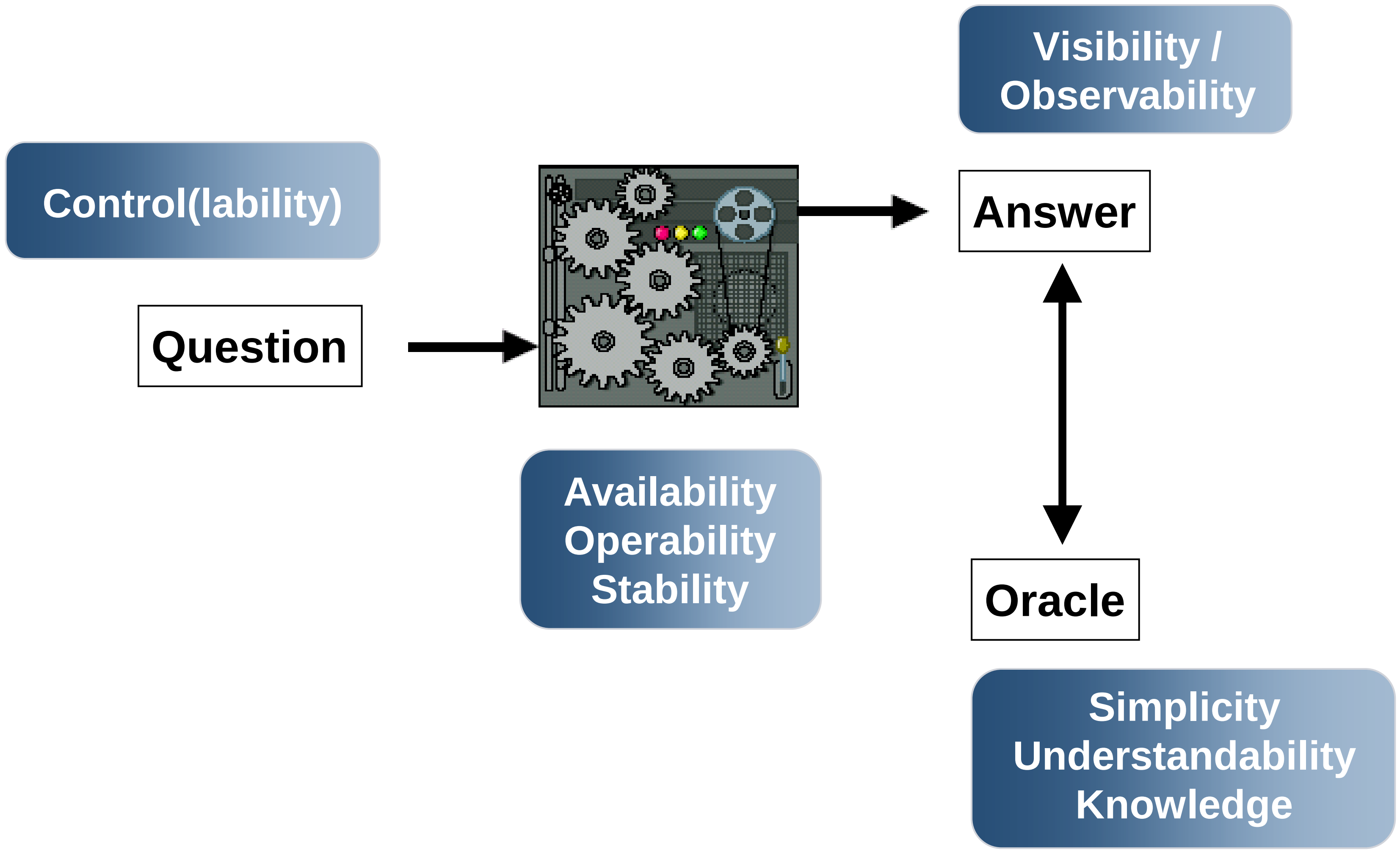
- *The fewer the changes, the fewer the disruptions to testing*
- Changes are infrequent, controlled and do not invalidate existing tests
- Software recovers well from failures

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, **O**bservability, **C**ontrol, **K**nowledge

Testability – Factors (3)



SIEMENS
Ingenuity for life

Heuristics of Testability

Practical Testability

Understanding of product (status)

Epistemic Testability

- Prior Knowledge of Quality
- Tolerance for Failure

How narrow is the gap between what we know and what we need to know about the status of the product under test?

Understanding of project

Project-Related Testability

- Change Control
- Information Availability
- Tool Availability
- Test Item Availability
- Sandboxing
- Environment Controllability
- Time

Testability influenced by changing the conditions under which we test

Understanding of users & usage

Value-Related Testability

- Oracle Availability
- Oracle Authority
- Oracle Reliability
- Oracle Precision
- Oracle Inexpensiveness
- User Stability & Unity
- User Familiarity
- User Availability
- User Data Availability
- User Environment Availability
- User Environment Stability & Unity

Testability influenced by changing the quality standard or our knowledge of it

Control & observe the product

Subjective Testability

- Product Knowledge
- Technical Knowledge
- Domain Knowledge
- Testing Skill
- Engagement
- Helpers
- Test Strategy

Testability influenced by changing the tester or the test process

Intrinsic Testability

- Observability
- Controllability
- Algorithmic Simplicity
- Unbugginess
- Smallness
- Decomposability
- Similarity (to known and trusted technology)

Testability influenced by changing the product itself

Reference: James Bach
http://www.satisfice.com/tools/testability.pdf

Testability – Constraints

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*

Challenges for testability

- Size, complexity, structure, variants, 3rd party components, legacy code
- Regulatory aspects
- *You cannot log every interface* (huge amount of data)
- Conflicts with other non-functional requirements like encapsulation, performance or security (privacy), and impacts system behavior
- Non-determinism: concurrency, threading, race conditions, timeouts, message latency, shared and unprotected data
- Memory / Time partitioning in embedded, safety-critical systems
- Adaptive, **SO** systems (self-* properties: **S**elf-**O**rganizing, self-configuring, self-optimizing)

Testability is affected / required by new technologies
Example: Cloud computing

SIEMENS
Ingenuity for life

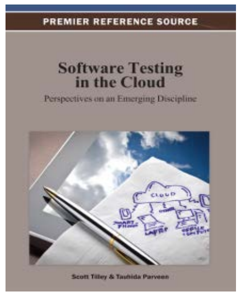
Cloud testability – the negative

- Information hiding and remoteness
- Complexity and statefulness
- Autonomy and adaptiveness
- Dependability and performance
- Paradigm infancy (temporary)

Cloud testability – the positive

+ Computational power
+ Storage
+ Virtualization

Testing requires lots of resources
and the cloud is certainly powerful enough to handle it


PREMIER REFERENCE SOURCE
Software Testing
in the Cloud
Perspectives on an Emerging Discipline
Scott Tilley & Tauhida Parveen

Reference: Tariq M. King, Annaji S. Ganti, David Froslie:
Towards Improving the Testability of Cloud Application Services, 2012

Design for Testability

Agenda

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 detected (earlier) and better root cause analysis with less effort
- Support testing of error handling and error conditions, for example to test how a component reacts to corrupted data
- Provide information on various kinds of coverage
- Better and more realistic estimation of testing efforts
- 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 …

SIEMENS

Ingenuity for life

Design for Testability

Agenda

What? Factors and Constraints

Why?

Who?

How?

Strategy for Design for Testability over the Lifecycle

Summary

Design for testability (DfT) – Who?

Educate stakeholders on benefit of DfT

Joint venture between Software / System Architects and Test Architects
- Collaboration between different stakeholders
- Requires accountability and clear ownership

Testability must be built-in by Software / System Architects (+ developers)
- Pro-active strategy in tactical design:
  design the system with testability as a key design criterion
- Test Architects must define testability requirements
  to enable effective and efficient test automation

Contractual agreement between design and test

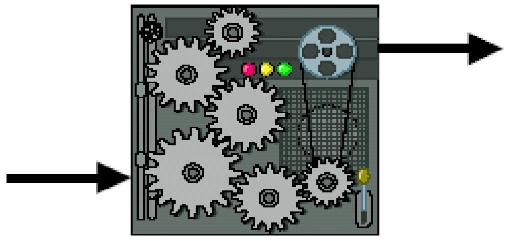
Balancing production code and test code
- Production code and test code must fit together
- Manage the design of production code and test code
  as codependent assets

Design for Testability

Agenda

SIEMENS
Ingenuity for life

Design for testability (DfT) – How?



Suitable test architecture, good design principles
Choice of technologies, libraries, frameworks, services

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

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

Control(lability)

Visibility / observability

Coding guidelines, naming conventions
Internal software quality (architecture, code)

Built-in self-test (BIST), built-in test (BIT)
Consistency checks (assertions, design by contract, deviations)

Logging and tracing (AOP, counters, monitoring, probing, profiling)

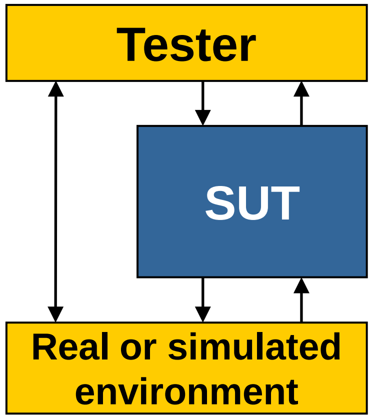
Diagnosis and dump utilities, black box (internal states, resource utilization, anomalies at runtime, post-mortem failure analysis)

Think test-first (xTDD): how can I test this?

Educate stakeholders on benefit of DfT

SIEMENS
Ingenuity for life

Architectural and design patterns (1)



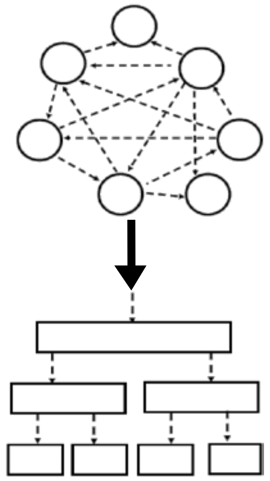
Use layered architectures, reduce number of dependencies

Use good design principles

- High cohesion, loose coupling, separation of concerns

Component orientation and adapters ease integration testing

- Components are the units of testing
- Component interface methods are subject to black-box testing



Avoid / resolve cyclic dependencies between components

- Combine or split components
- Dependency inversion via callback interface (observer-observable design pattern)

Example: MSDN Testability guidance

- *Using the Model-View-Presenter (MVP) Design Pattern to enable Presentational Interoperability and Increased Testability*

http://blogs.msdn.com/b/jowardel/archive/2008/09/09/using-the-model-view-presenter-mvp-design-pattern-to-enable-presentational-interoperability-and-increased-testability.aspx

Architectural and design patterns (2)

Provide appropriate test hooks and factor your design in a way that lets test code interrogate and control the running system

- Isolate and encapsulate dependencies on the external environment
- Use patterns like dependency injection, interceptors, introspective
- Use configurable factories to retrieve service providers
- Declare and pass along parameters instead of hardwire references to service providers
- Declare interfaces that can be implemented by test classes
- Declare methods as overridable by test methods
- Avoid references to literal values
- Shorten lengthy methods by making calls to replaceable helper methods

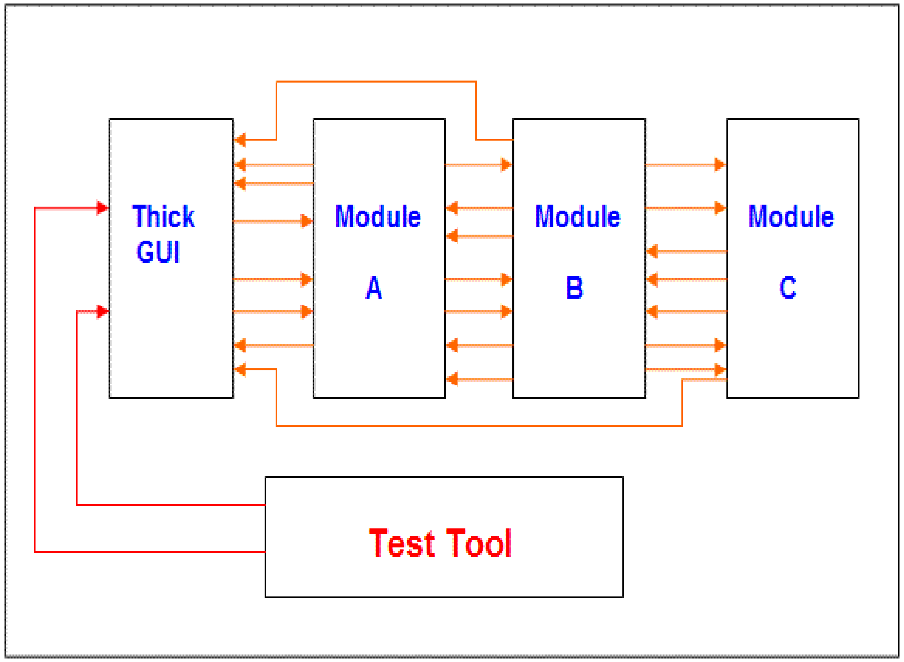
Promote repeatable, reproducible behavior

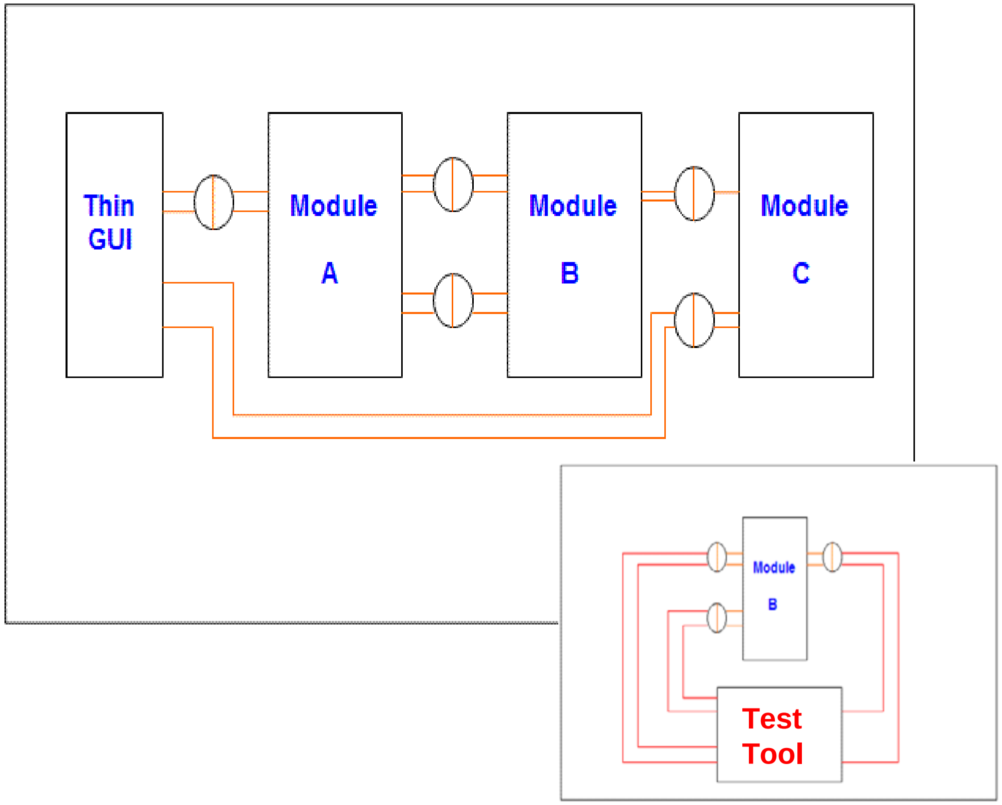
- Provide utilities to support deterministic behavior (e.g. use seed values)

→ That’s just good design practice!

SIEMENS
Ingenuity for life

Bad architecture vs. Good architecture

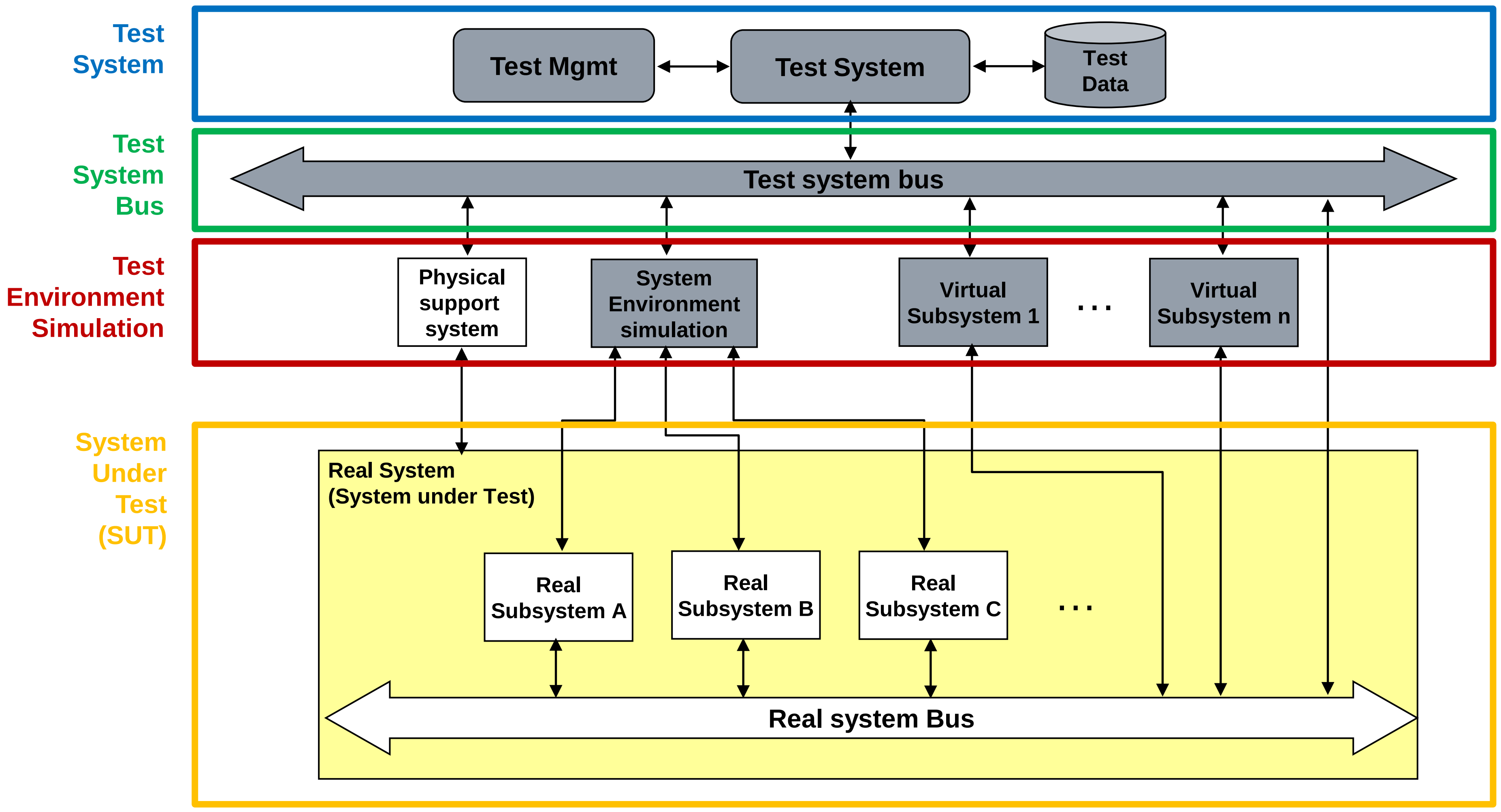




- Thick GUI that has program logic
- Many interfaces between the modules that are not clearly defined
- Testing of specific functions cannot be isolated (unit testing)
- Testing has to be done through the GUI

- Thin GUI that has no program logic other than dealing with presentation
- Interfaces between the modules are well defined
- Each module can be tested independently from the other modules

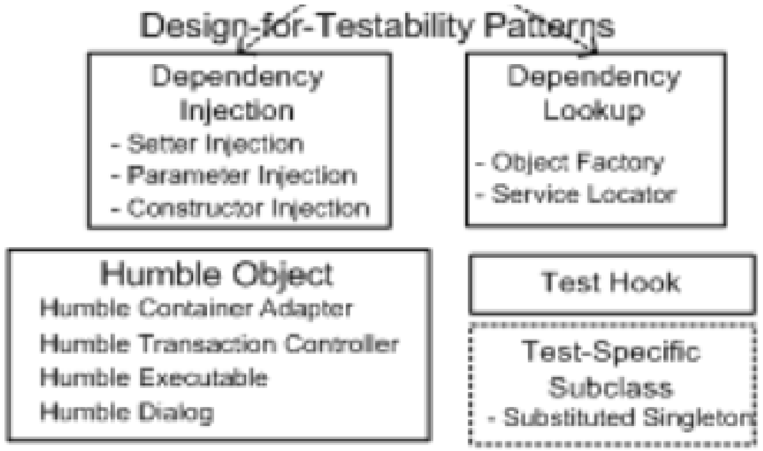
Test architecture for system integration testing



Design for testability (DfT) patterns

Dependency injection
The client provides the depended-on object to the SUT

Dependency lookup
The SUT asks another object to return the depended-on object before it uses it



Humble object
We extract the logic into a separate easy-to-test component that is decoupled from its environment

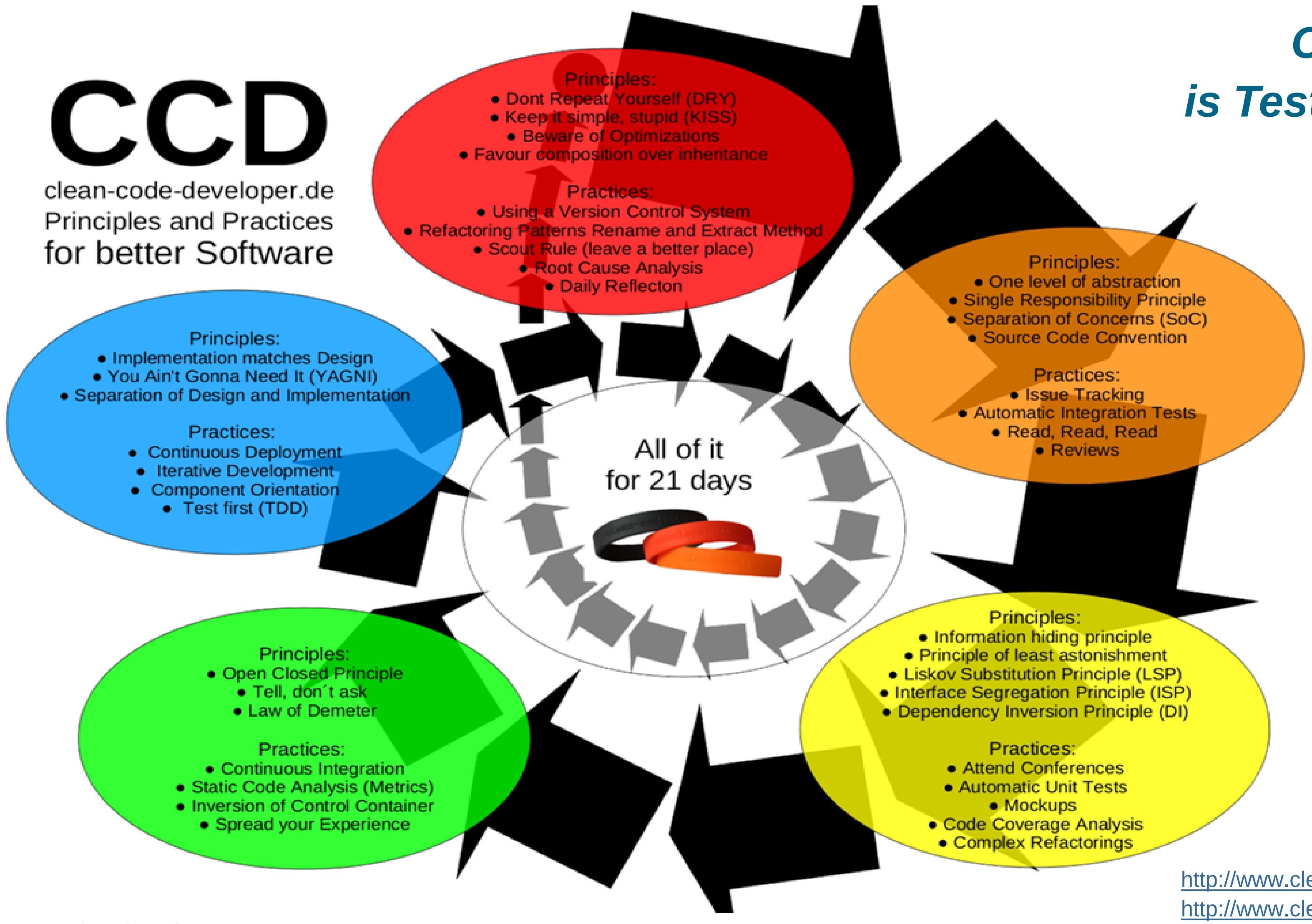
Test hook
We modify the SUT to behave differently during the test

Reference: Gerard Meszaros: xUnit Test Patterns: Refactoring Test Code, Addison-Wesley, 2007
http://xunitpatterns.com/

Use principles and practices from clean code developer (CCD) (1)

SIEMENS
Ingenuity for life

Clean Code
is Testable Code



Principles:
- Dont Repeat Yourself (DRY)
- Keep it simple, stupid (KISS)
- Beware of Optimizations
- Favour composition over inheritance

Practices:
- Using a Version Control System
- Refactoring Patterns Rename and Extract Method
- Scout Rule (leave a better place)
- Root Cause Analysis
- Daily Reflecton

Principles:
- One level of abstraction
- Single Responsibility Principle
- Separation of Concerns (SoC)
- Source Code Convention

Practices:
- Issue Tracking
- Automatic Integration Tests
- Read, Read, Read
- Reviews

Principles:
- Information hiding principle
- Principle of least astonishment
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DI)

Practices:
- Attend Conferences
- Automatic Unit Tests
- Mockups
- Code Coverage Analysis
- Complex Refactorings

Principles:
- Open Closed Principle
- Tell, don´t ask
- Law of Demeter

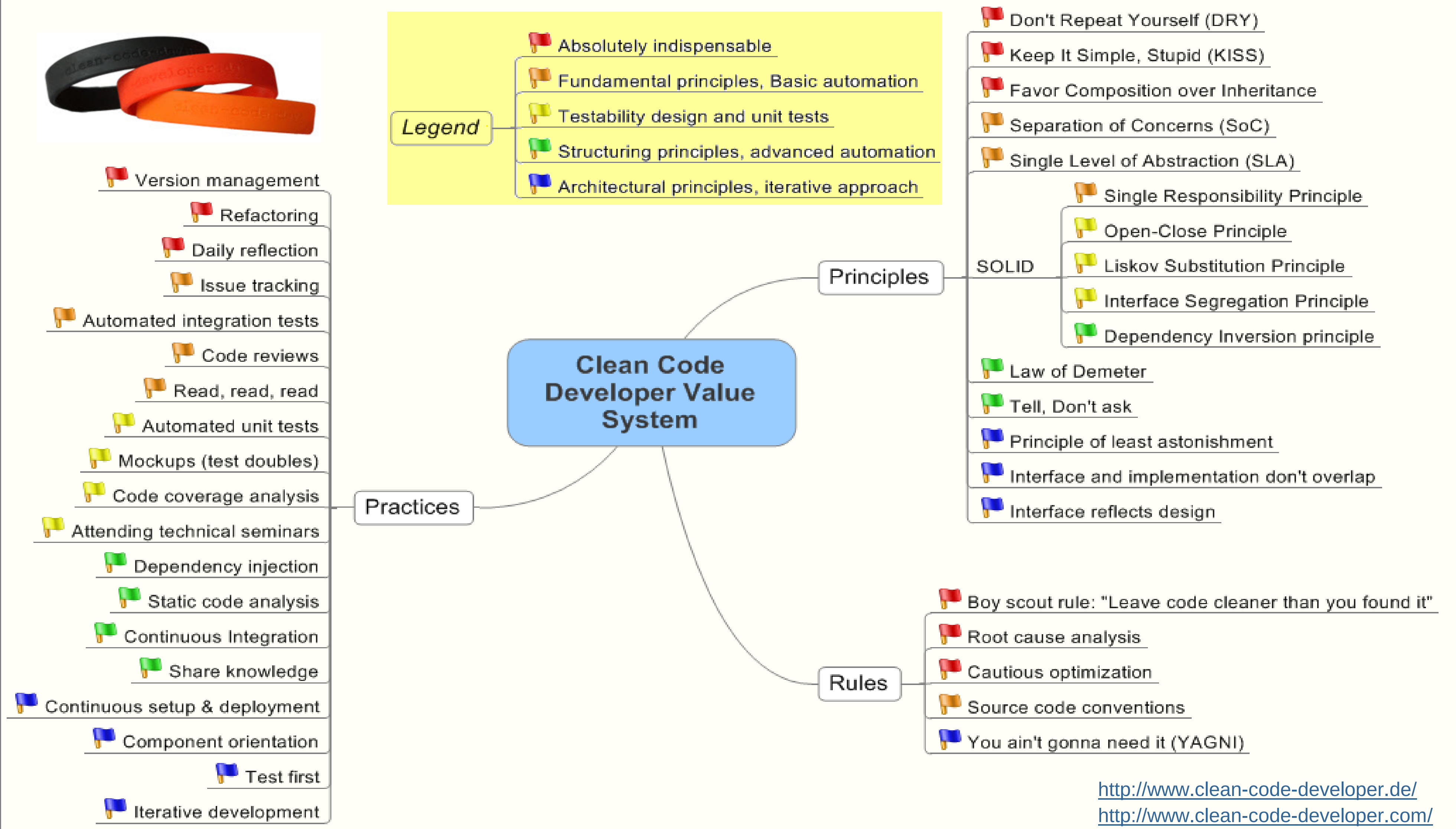
Practices:
- Continuous Integration
- Static Code Analysis (Metrics)
- Inversion of Control Container
- Spread your Experience

Principles:
- Implementation matches Design
- You Ain't Gonna Need It (YAGNI)
- Separation of Design and Implementation

Practices:
- Continuous Deployment
- Iterative Development
- Component Orientation
- Test first (TDD)

http://www.clean-code-developer.de/
http://www.clean-code-developer.com/

Use principles and practices from clean code developer (CCD) (2)



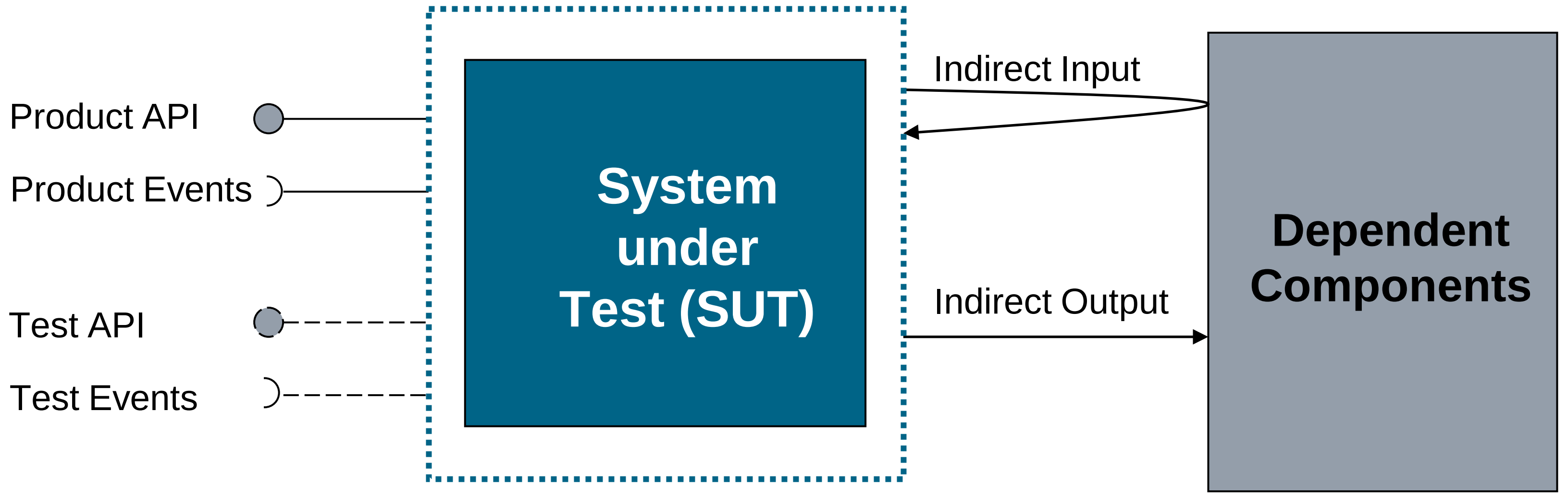
Legend
🟥 Absolutely indispensable
🟧 Fundamental principles, Basic automation
🟨 Testability design and unit tests
🟩 Structuring principles, advanced automation
🟦 Architectural principles, iterative approach
Clean Code Developer Value System
Practices
🟥 Version management
🟥 Refactoring
🟥 Daily reflection
🟧 Issue tracking
🟧 Automated integration tests
🟧 Code reviews
🟧 Read, read, read
🟨 Automated unit tests
🟨 Mockups (test doubles)
🟨 Code coverage analysis
🟨 Attending technical seminars
🟩 Dependency injection
🟩 Static code analysis
🟩 Continuous Integration
🟩 Share knowledge
🟦 Continuous setup & deployment
🟦 Component orientation
🟦 Test first
🟦 Iterative development
Principles
🟥 Don't Repeat Yourself (DRY)
🟥 Keep It Simple, Stupid (KISS)
🟥 Favor Composition over Inheritance
🟧 Separation of Concerns (SoC)
🟧 Single Level of Abstraction (SLA)
SOLID
🟧 Single Responsibility Principle
🟨 Open-Close Principle
🟨 Liskov Substitution Principle
🟨 Interface Segregation Principle
🟩 Dependency Inversion principle
🟩 Law of Demeter
🟩 Tell, Don't ask
🟦 Principle of least astonishment
🟦 Interface and implementation don't overlap
🟦 Interface reflects design
Rules
🟥 Boy scout rule: "Leave code cleaner than you found it"
🟥 Root cause analysis
🟥 Cautious optimization
🟧 Source code conventions
🟦 You ain't gonna need it (YAGNI)

http://www.clean-code-developer.de/
http://www.clean-code-developer.com/

Attributes of testability interfaces

Interfaces can act as
- control points
- observation points

SIEMENS
Ingenuity for life



- Lightweight vs. intrusive
- Realistic vs. artificial
- Control-only vs. visibility-only vs. both
- Information-only vs. knowledge
- Debug-only (logs / asserts) vs. release build
- Internal (embedded) vs. external vs. shipping (documented) feature
- NFR (e.g. security) harmless vs. NFR conflict (e.g. security hole)
- Manual vs. automatable
- Tester owned vs. developer owned

SIEMENS

Ingenuity for life

Testability interfaces – Generic examples

All components / subsystems must provide a test interface

- for any required input / output hardware
- to set / reset its GUI and internal states to the initial state
- to signal whether it is in a stable initial state, e.g. after startup
- to retrieve the current configuration
- to change the current configuration without need to restart the SUT
- to retrieve the current deployment description, e.g. in XML
- to retrieve status information from remote components
- to set error conditions from outside (error seeding) in order to simulate extreme conditions and enter exotic code paths

It shall be possible for a test automation tool to detect the current state of the SUT, e.g. waiting for a resource, ongoing workflow step

→ For legacy systems use refactoring patterns to improve testability

Logging and tracing

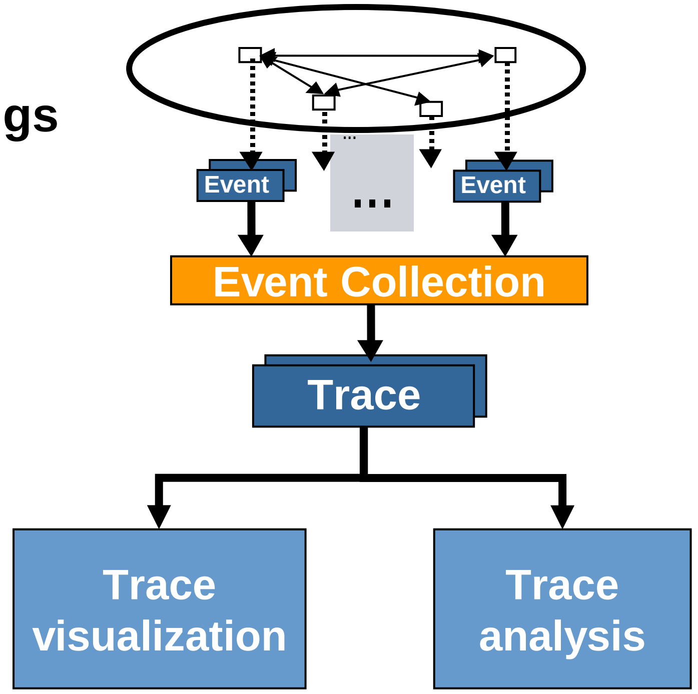
Check the architecture and design.
Check the dynamic behavior (i.e. communication) of the system. Detect bugs, including sporadic bugs like race conditions.

Analyze execution, data flow, and system state.
Follow error propagation and reproduce faults.

For this, tracing is essential.
Tracing is important for unit testing, integration testing, system testing, system monitoring, diagnosis, …

Start early with a logging and tracing concept!

Distributed System



Relationship between trace extraction, trace visualization, and trace analysis

Further benefits of good logging and tracing practices

SIEMENS
Ingenuity for life

Avoid *analysis paralysis* – a situation where the test team spends as much time investigating test failures as they do testing

Support failure matching and (automated) failure analysis

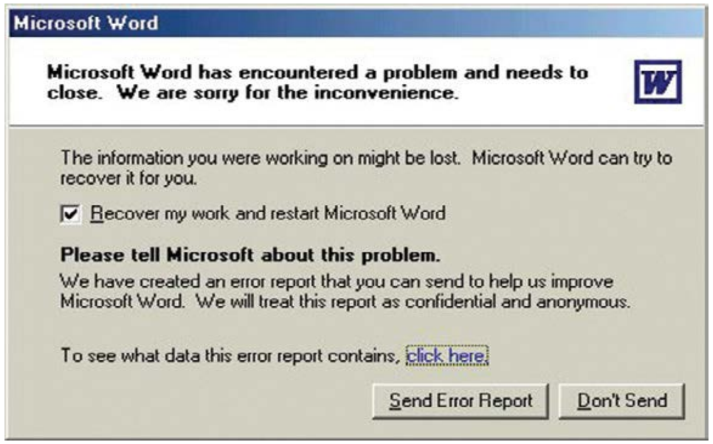
Avoid duplicates in the bug database by using rules of analysis when deciding if a failure has been previously observed

Backbone of reliable failure analysis

Example: Windows Error Reporting (WER)

- *Small, well defined set of parameters*
- *Automated classification of issues*
- *Similar crash reports are grouped together*

→ *Reduced amount of data collected from customers: **bucketing***



Microsoft Word has encountered a problem and needs to close. We are sorry for the inconvenience.

The information you were working on might be lost. Microsoft Word can try to recover it for you.

☑ Recover my work and restart Microsoft Word

Please tell Microsoft about this problem.
We have created an error report that you can send to help us improve Microsoft Word. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Diagnosis and dump utilities, black box
**(internal states, resource utilization,
anomalies at runtime, post-mortem failure analysis)**

SIEMENS
Ingenuity for life

**Detect, identify, analyze, isolate, diagnose, and reproduce
a bug (location, factors, triggers, constraints, root cause)
in different contexts as effective and as efficient as possible**

CommunicationDiagnosisExample

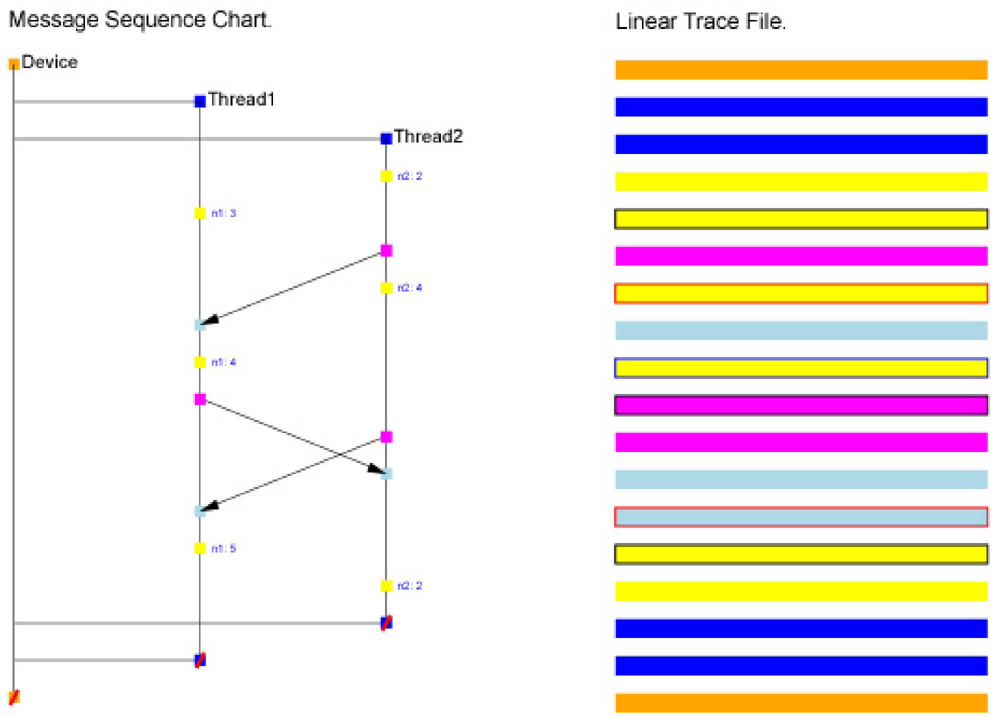
- Unit / component testing:
  bug in my unit/component or external unit/component or environment?
- Integration testing of several subsystems:
  bug in my subsystem or external subsystem or environment?
- System of systems testing:
  bug in which part of the whole system of systems?
- Testing of product lines and ecosystems:
  bug in domain engineering part or application engineering part?
  bug in my part of the system or in supplier's part of the system?
- Bug in system under test (SUT) or test system?
  → quality of test system is important as well ...
- SUT in operation:
  maintenance, serviceability at the customer site

Test evaluation in regression testing

Typically regression test suites are growing and are becoming huge
→ Testability is very important

- Especially interesting is the characteristic "visibility / observability" here to evaluate the test oracle and to discover any deviation and anomaly

Special challenge:
test evaluation
in regression testing
for concurrent systems



Strategy for test evaluation in regression testing for concurrent systems

SIEMENS

Ingenuity for life

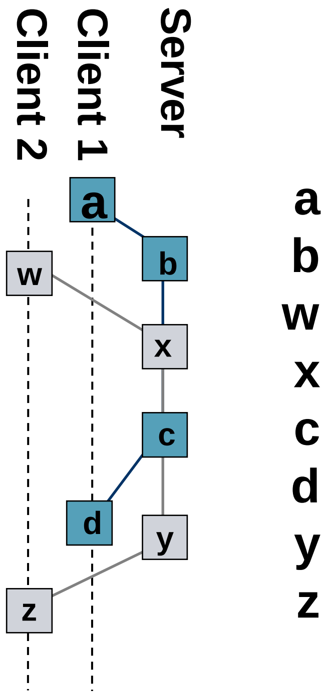
Explicitly specify the expected dynamic behavior
Build special comparators to automatically evaluate different test runs during regression testing

Use the *partial* order of events defined by the *happened-before relation (causal ordering)**

- If two events occur at the same process, then they occur in the order they were observed and executed by the process
- If a message-passing interaction occurs between two processes, the event of sending occurs before the event of receiving
- Is transitive and irreflexive

Events that are not ordered by the *happened-before* relation are *concurrent*

*Leslie Lamport: *Time, Clocks and the Ordering of Events in a Distributed System*, Communications of the ACM, Vol. 21, July 1978, pp. 558-565.



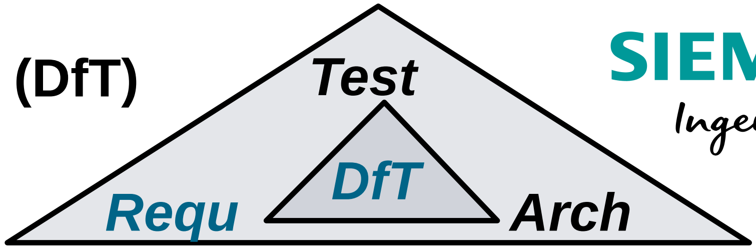
happened-before relation:
a → b, b → x, w → x, x → c,
c → d, c → y, y → z, etc.

Concurrent events:
a || w, b || w, d || y, d || z

Design for Testability

Agenda

Strategy for Design for Testability (DfT) over the lifecycle (1)

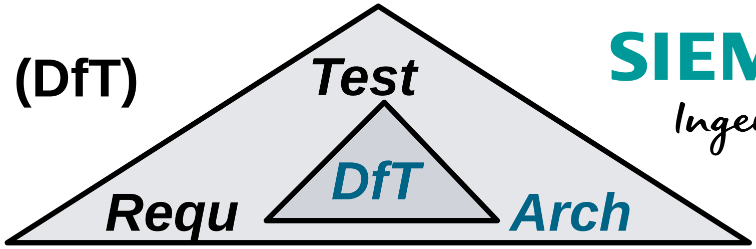


SIEMENS
Ingenuity for life

Educate stakeholders on benefit of DfT

- 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 test 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 and used

Strategy for Design for Testability (DfT) over the lifecycle (2)



SIEMENS
Ingenuity for life

Educate stakeholders on benefit of DfT

- Specified and elaborated in testability guidelines for architects, developers, testers (→ 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

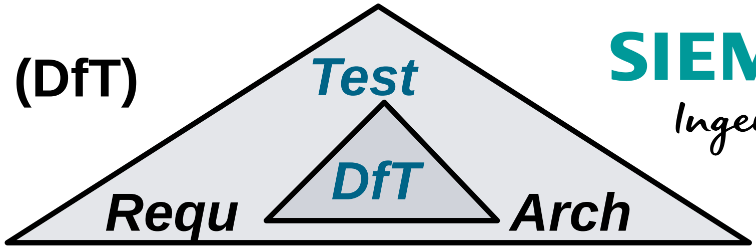
PhilippeKruchten
nal_4+1ViewMode

Control+Observatio
Points@Architectur

ZoneConcept@Ar
chitecture

*Additional view to the 4+1 View Model of Software Architecture by Philippe Kruchten

Strategy for Design for Testability (DfT) over the lifecycle (3)



SIEMENS
Ingenuity for life

Educate stakeholders on benefit of DfT

- Included as one important criteria in milestones and quality gates, e.g.:
  → Check **control** and **observation** 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 and in code reviews
  → Check testability guidelines and documentation of testability artifacts
  → 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)
- Realized by executing governance for testability guidelines

→ Neglecting testability means increasing technical debt … $$ …

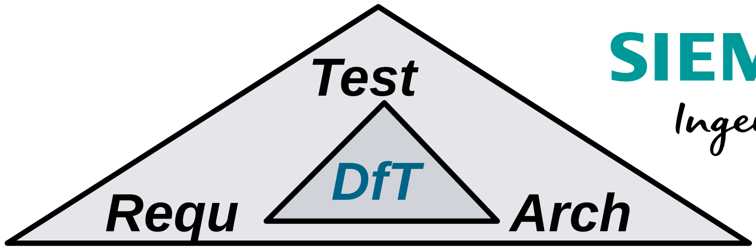
*Architecture-centric methods from SEI, see http://www.sei.cmu.edu/

SIEMENS
Ingenuity for life

Design for Testability

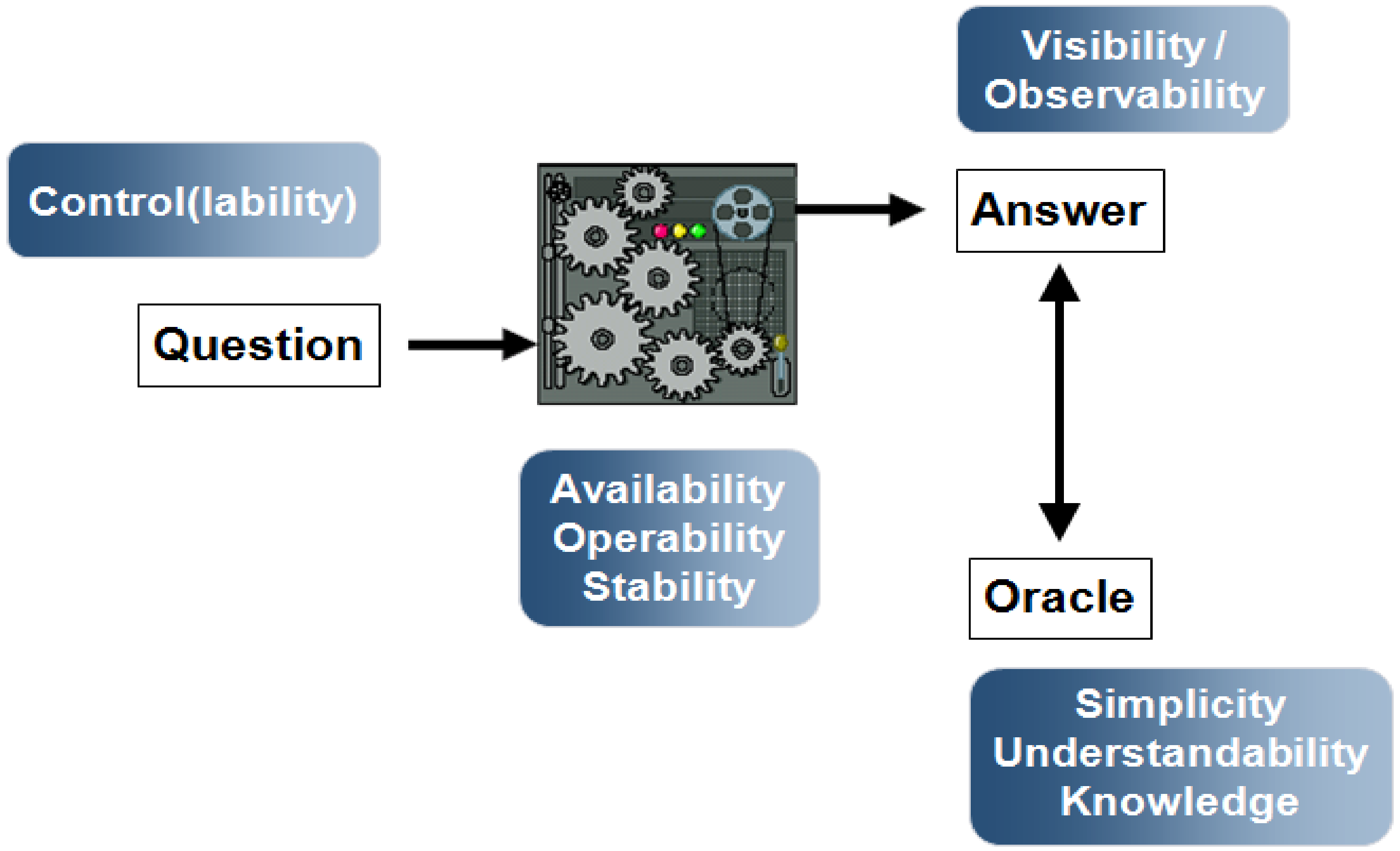
Agenda

Summary: Testability and Design for testability (DfT)



SIEMENS
Ingenuity for life

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



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

What we have learned

Testability is driven by a set of factors (most important are controllability and observability) that must be balanced to optimize their impact.

Testability is the key to cost-effective test automation.

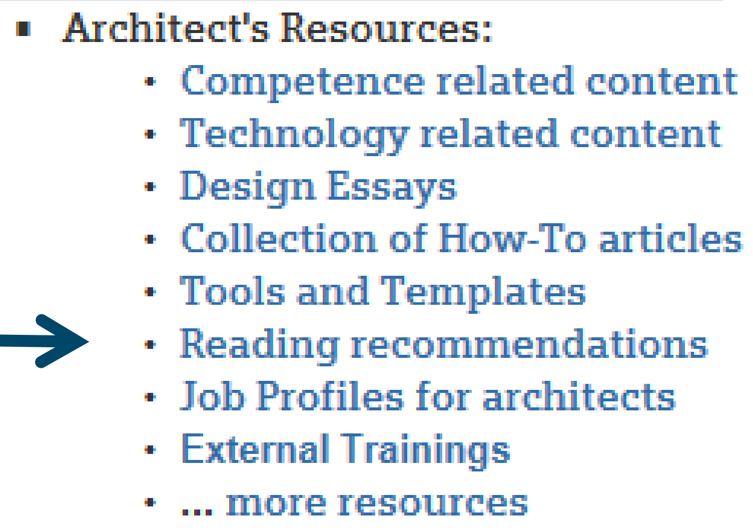
Bad testability is a major risk to project progress and product quality.

Good testability does not happen by accident it must be built-in (architectural/design patterns, clean code principles) by a joint venture between
Software / System Architects and Test Architects.

Further readings

Use the SSA Wiki :
https://wiki.ct.siemens.de/x/fReTBQ

and check the “Reading recommendations”:
https://wiki.ct.siemens.de/x/-pRgBg



- Architect's Resources:
  - Competence related content
  - Technology related content
  - Design Essays
  - Collection of How-To articles
  - Tools and Templates
  - Reading recommendations
  - Job Profiles for architects
  - External Trainings
  - ... more resources