

**Design for Testability (Enabler for Effective test automation)**  
door **Bryan Bakker en Con Bracke**

---

**11 februari 2010**

voordracht georganiseerd door het



**TECHNOLOGISCH INSTITUUT**  
*Discussiegroep Software Testing*

met de steun van



**cronos**  
*e-business integrator*



a Clear2Pay company



systematically delivering success

**quasus**



tesco  
testing your software

**logica**

**QApri-IT**  
SOFTWARE TESTING

**ps\_testWare**  
Your devil's advocate

---

Ingenieurshuis - K VIV, Antwerpen



# Design for Testability

TI-KVIV SWT



Bryan Bakker

February 11<sup>st</sup> 2010



## Contents

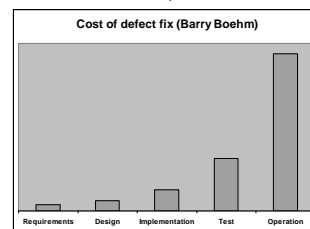
- Sioux
- Intro
- What is Design for Testability (DfT)?
- Test Automation
- Design Rules
- Pre-requisites
- Watch out
- Conclusion
  
- Practical experience by Con Bracke
  
- Scope: embedded/technical domain



- Device including HDD
- During test phase no serious HDD issues
- After release: HDD failures in field
  - Customers return units (NFF)
  - False alarms!
- SW not robust against HDD imperfections
- Firmware upgrade needed to prevent more returns
- Could this have been prevented?
  
- Simulate HDD imperfections
  - find defects during development/test
  - more robust SW/System

## What is Design for Testability (DfT)?

- **Definition:**  
*Take testing into account during design/architecture definition*
  
- **Main goals:**
  - More efficient testing (find defects earlier, automation)
  - Increase coverage of testing (manual and automatic, make it possible to detect other problems)
  - Enable automatic testing



© Sioux Embedded Systems | Confidential | 5

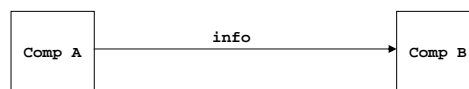
## Examples

- **Think of:**
  - Testing without HW (not finished or expensive)
  - Simulate environment (for automatic testing or unfeasible environment)
  - Replace mechanical switches/buttons (test automation)
  - Support for test automation
  - Negative testing (failures from HW or environment)
  - Support for test/sw engineers (diagnosis)
  - Logging/Tracing
  - Test components in isolation (modular architecture)
  - Support for integration testing (test for messages)
  - Test without UI
  - Reliability/Profile testing: record user actions and replay
- **By**
  - Visibility
  - Control

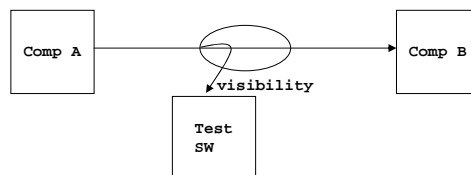
© Sioux Embedded Systems | Confidential | 6

- Visibility
  - Usually: subset of system information is shown to end-user
  - DfT: interface defined to extract info from system
  - Also for “hidden” info

- Normal transfer of information

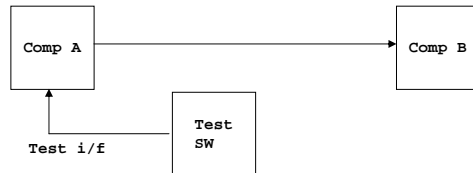


- Offer information to test software:



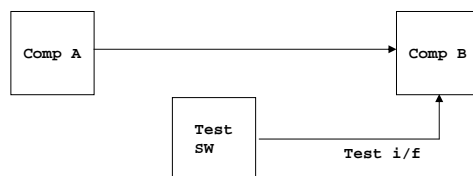
- Define test interface (test hook) to inspect info from Comp A
- On Comp A or Comp B or in between?

- Test interface on Comp A:



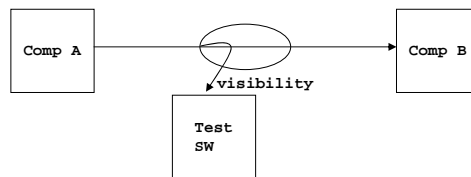
- Comp A is aware of interface

- Test interface on Comp B:



- Comp B is aware of interface

- Use wrapper or message queue inspector (e.g. VxWorks)

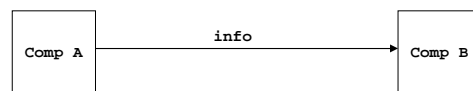


- Comp A and B are unaware of interface
- But not everything is sent to other components...
- Where to interface is design decision

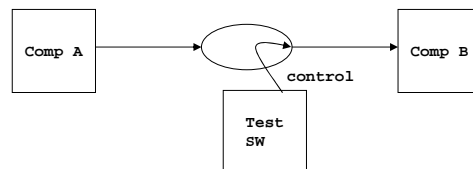
- Extract all kinds of system information
  - Temperature
  - #Images passing through image chain
  - Recording speed of recorder
  - Mechanical movements verification
  - Inspect messages (for integration tests)
  - State information (of system or components)
  - Logging (better inspection/analysis, tool support)
  - Resource usage (cpu, memory, network)
  - ...

- Control
  - Usually: system controlled by system interfaces like user, environment, network, etc.
  - DfT: interface defined to control the system

- Normal transfer of information



- Information altered by test software:



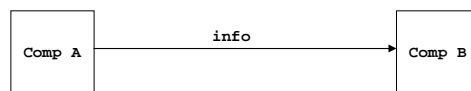
- Define test interface to control Comp B
  - set information
  - ignore control from Comp A (optionally)

## Control examples

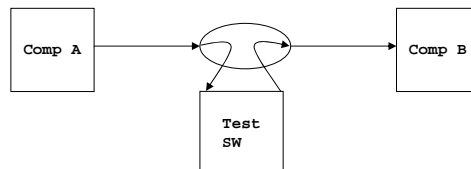
- Trigger all kinds of system actions
  - Push buttons (UI, mechanical)
  - Set configurations
  - Simulate events (motion events, alarms, hot temps)
  - Mechanical movements
  - Simulate HW failures/imperfections
  - ...

## Combined

- Normal transfer of information



- Information retrieved and altered by test software:



- Define get and set test interfaces

- Control used to trigger actions
  - Best practice: as “low” as possible in the architecture
    - close to hardware
    - as much coverage as possible
    - trade-off between costs and coverage
  - Possible to test below the UI
    - UI is volatile (except “mechanical UI”)
- Visibility used to verify expected result
  - Best practice: use logfile (also evidence) or internal system information
    - Avoid UI information (volatile)

- State visibility:
  - Every component stores state information
  - In one dedicated component
  - Testcases can get this information
  - Possibility: with one key-press → dump the complete system information (for defect analysis)
  - Not to be used internally by system (no information hiding)
- State machines trace/log state transitions
  - “easily” test the state machines with state-transition testing
  - Determine coverage of testcases (n-switch coverage)

- Communication between each set of components visible via interfaces (in tracing)
  - Default functionality in VxWorks
  - Communication can also be altered
  - Used for integration testing
- All user actions are logged, and can “replayed”
  - Input for profile tests (software reliability engineering)
  - Records error-guessing/exploratory tests for reproducibility
- Failures in HW to be simulated via (test i/f in) drivers
- Most projects start with: logging conventions

- Early involvement of test discipline
- Influence on architecture/design
  - By (test) architect
  - Architecture must support effective testing
- Test requirements
  - Functionality needed in the product to support testing
  - Real requirements, need priority
  - Implementation available on time
- Test interfaces
  - Are deliverables of project
  - Supported interfaces, thus maintained
  - Used for automatic tests
- Test req/interfaces become part of the product
  - Test functionality grows into supported functionality of the product (Excel, XRays)
- Management commitment (DfT is an investment)

1. Disable test functionality in release versions?
  - Like logging, tracing, test functions
  - Different version, will behave differently
    - Performance
    - Issues in release version not reproducible in development version
  - Test functionality may still be needed
    - Service/diagnostics/factory
    - Problem analysis in the field
2. Testing via test interfaces → not the real thing
  - Customer/environment uses different interfaces
  - Decide where to interface (coverage ↔ cost)

3. Beware: **Probe Effect**
  - “unintended alteration in system behavior caused by measuring that system” (wikipedia).

**Be ware of these effects!**

- Design for Testability
  - More efficient testing
  - Increase coverage of testing
  - Enable automatic testing
- Visibility & Control
- Part of design/architecture
- Nothing new! But hardly practiced structurally
- Beware: different in real world!





Source of your development.



[www.siox.eu](http://www.siox.eu)



[bryan.bakker@siox.eu](mailto:bryan.bakker@siox.eu)



+31 (0)40 26 77 100 / +32 (0)14 848 718

# PHILIPS

sense **and** simplicity

## Bringing your ideas to life

Philips Applied Technologies

July 2009

Visit us at [www.apptech.philips.com](http://www.apptech.philips.com)

## Contract innovation services

Innovation through turnkey solutions and specialist support



### Healthcare

- Medical devices & implants
- Patient monitoring & connected care
- Medical imaging & therapy systems
- Molecular healthcare devices

### Lifestyle

- Multimedia Experience
- Personal Care, Wellness & Beauty
- Robotics
- Information, Storage & Streaming & Retrieval

### Technology

- Home and Building Automation & Security
- High Precision Systems
- Opto-electronic modules
- Energy

## At the heart of leading innovations

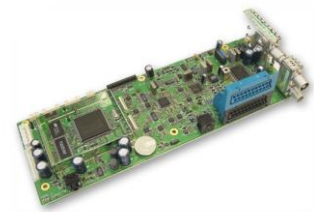


# PHILIPS

sense and simplicity

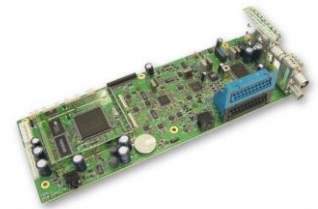
## Design for testability Lessons learned

Con Bracke  
Philips Applied Technologies  
February, 2010



# Design for testability real-life experiences

- Embedded software projects
- First of a kind products
  - New hardware
  - New software
  - Software and hardware developed in parallel
  - Almost no debug tools available for this (new) environment



# Design for testability real-life experiences

- Building a product

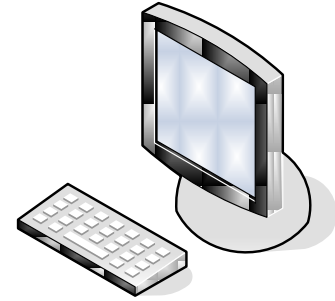
Developers and testers need visibility and control

- Component testing
- Integration testing
- System testing

Factory and service also want visibility and control

- Producing the product
- Servicing the product

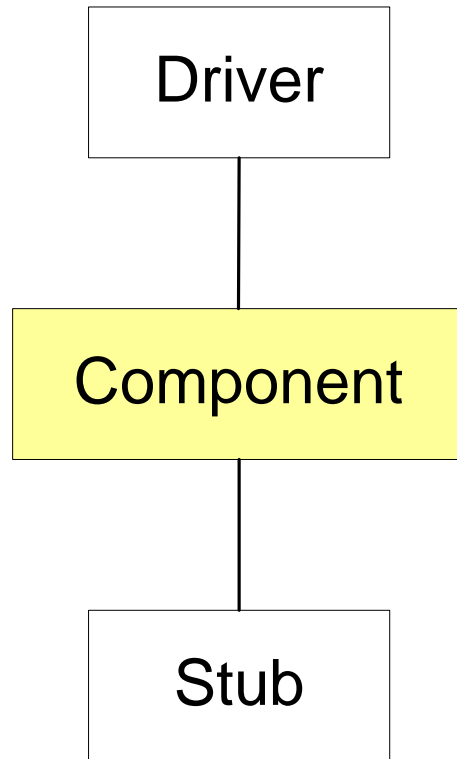
# Design for testability: considerations



- Can parts be tested on a PC?
  - Easier to debug
  - Lot of supporting software available
- Pros
  - Timing is different
  - Final tests should be done on a real system



# Design for testability: considerations



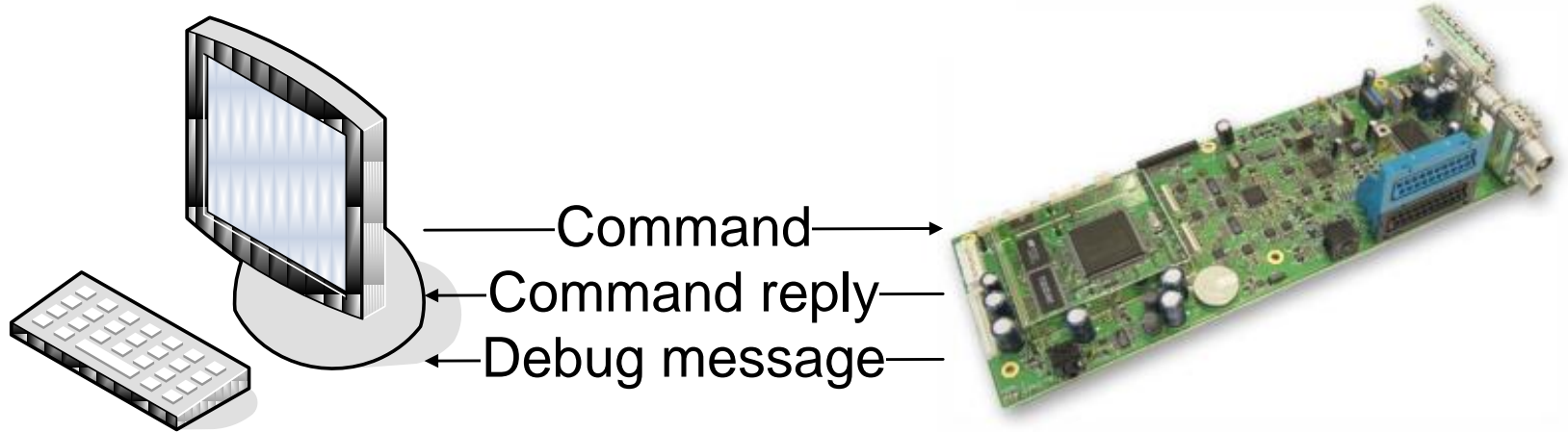
 Component under test

- Decisions influenced by:
  - Complexity of stubs / drivers
  - Simulation of hardware
  - Complexity of test data
  - Return On Investment (ROI)

# Testability techniques

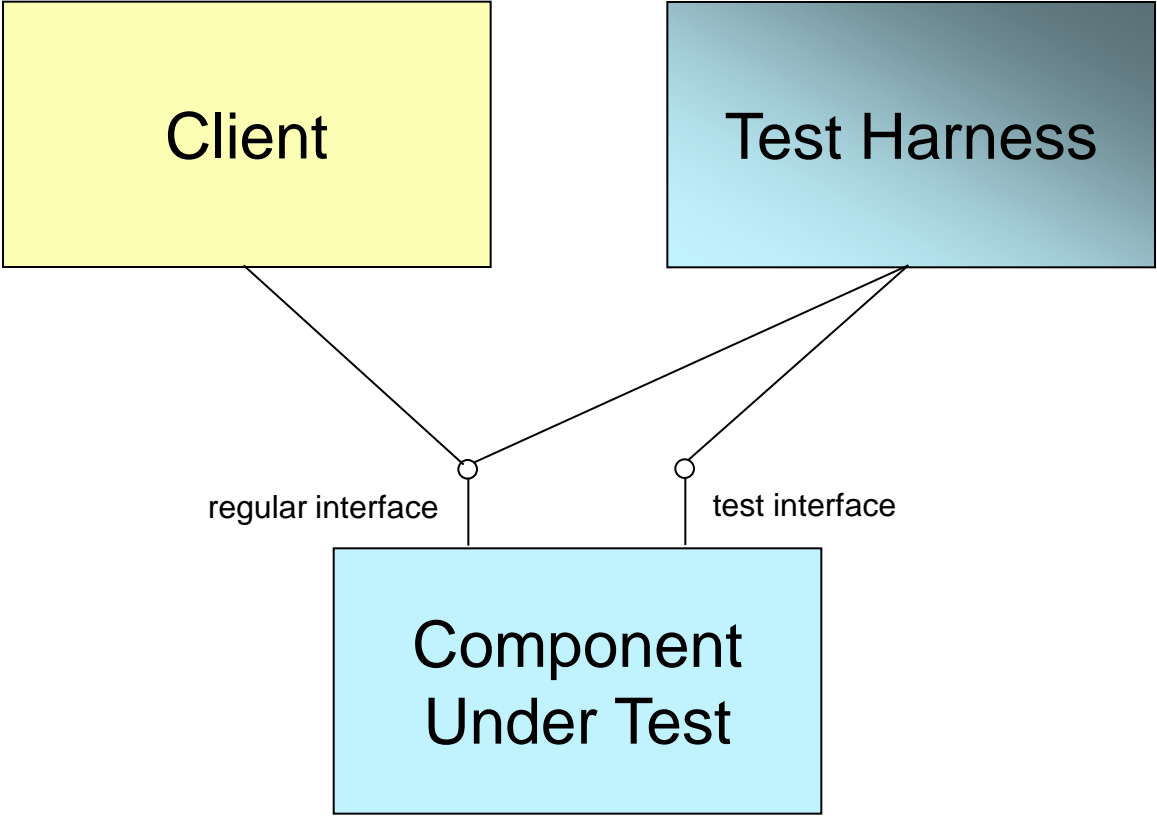
- Logging
- Assertions
- Diagnostics
- Resource monitoring
- Test points
- Fault injection hooks

# Test infrastructure



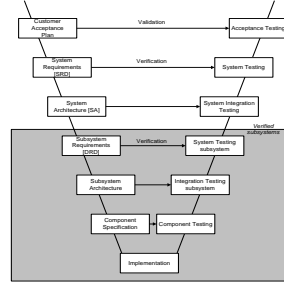
Keyboard or script file

# Test infrastructure



# Logging

- Log important / unusual events
- Logs not only needed to record defects, but also to see what happened before
- On all test levels: component-, integration-, system test



# Events to log

- Defects
- Warnings
- Interface calls (public calls)
- Component calls (private calls)
- Exceptions, unusual events
- States

# Events to log

- Installation and Configuration settings
- Start-up and shutdown of components and connections
- Completion of significant execution of algorithms
- Completion of transactions
- User Inputs

# What to do with the logging information?

- Detect 'internal' errors before they appear at system level
- Additional information simplifies reproducing failures
- Study timing sequences; traceability.
- Study load patterns
- Learn from end user usage on behalf of Operational Profiles
  - mostly used features
  - kinds of mistakes
  - most common configurations
  - usability problems; where did the user gave up

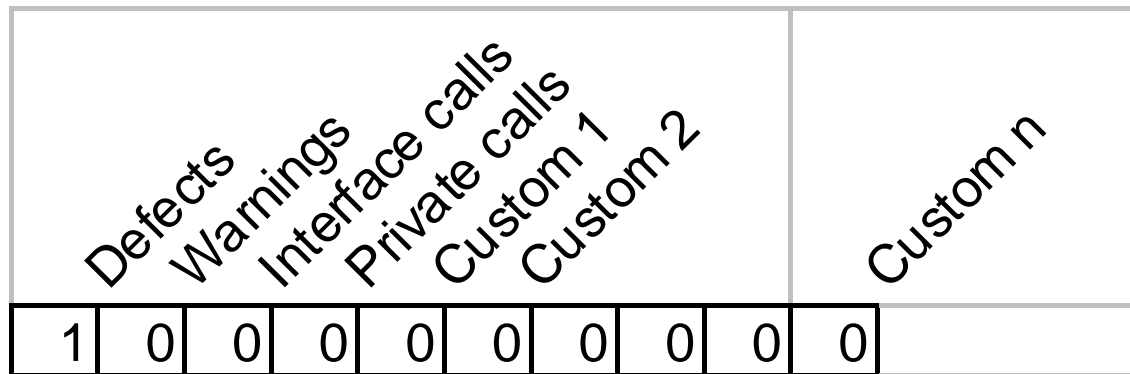
# Logging: lessons learned

- Log records must include timestamps and the component or subsystem identification.
- Limit size of log records: send tokens instead of the real messages.
- Tokens also make test scripts less vulnerable for changes in message contents.
- Test harness should provide room for extensions.



# Logging: lessons learned

- Define variable logging levels
  - Definition of debug masks
  - Output for logging and tracing can be switched on and off per component or subsystem



# Assertions

- Assertions in the code can make faults visible
- Assertions allow faults easier to detect and remove
- A FALSE assertion will raise an exception
  - E.g., code for calling a square root could include an assertion to check that the input was non-negative  
`ASSERT(input >= 0)`

## Diagnostics



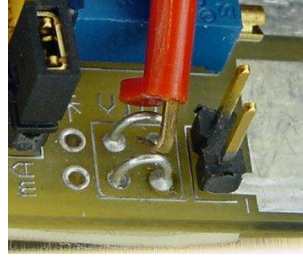
- Helps to detect faults that do not immediately lead to failures
  - Example chip access:
    - Number of retries
    - Number of times a chip reports errors
  - Useful for service engineer
  - Useful for test engineer
- Suspicious results can trigger:
- change in focus for testing
  - give an indication for possible software/hardware problems

# Resource monitoring



- Provides visibility of the internal workings of the code
- Examples:
  - Amount of free memory: a decreasing number could be a sign for memory leaks
  - Stack usage of different tasks

## Test points



- Allows data to be inserted and examined at various points
- Monitoring
- Fault injection, especially for data streams
  
- Test points often required by:
  - Factory (for producing the product)
  - Service



# Fault injection hooks

- Special functionality enforce the software to deal with unexpected situations
- Examples:
  - Simulate low on memory or memory full
  - Force exceptions
  - Influence timing (critical sections, slow down tasks)
  - Control state behavior

# End of the project: lessons learned

- System tests should be done on the same system as delivered: do not remove the test code
- Consider protecting direct access to test functionality
- Assertions need special attention: What to do when an assert is triggered? (ignore, recover, something else?)

# Design for testability

- Should be part of requirement process
- Follows normal development process
- Increases visibility of internal behavior
- Results in a better understanding of the system
- Dependencies between internal, external and shared components are well understood
- Scriptable interface makes testing easier
- Better understanding how the system is used by the end-user
- More control by the test engineer results in improved testing

## Benefits



- Better quality
- Good fault location and isolation
- Less verification & validation time: faster defect analysis and reporting
- Lower cost



☐☐ *consulteer onze Website* ➡ <http://www.ti.kviv.be>



**Technologisch Instituut VZW**

INGENIEURSHUIS - K VIV

Desguinlei 214, B-2018 Antwerpen 1, Tel. (03) 260.08.40, Fax (03) 216 06 89, E-mail [INFO.TI@TI.KVIV.BE](mailto:INFO.TI@TI.KVIV.BE)

GK 068-2142216-95 BBL 320-0843321-73 BTW BE 435 567 909

<http://www.ti.kviv.be>