Favorite Picks in Software Testing

Arie van Deursen
TU Delft
@avandeursen

Kickoff presentation Test Week Stabiplan, 2011
Outline

1. Background

2. Test Strategies

3. Test Week Objectives

4. Selected Strategies
   *Combinations, states, contracts, exceptions*

5. Testing Research at TU Delft

6. Open Q & A
The TU Delft
Software Engineering Research Group

Education
• Programming, software engineering
• MSc, BSc projects

Research
• Software architecture
• Software testing
• Repository mining
• Collaboration
• Services
• Model-driven engineering
• End-user programming
Software Testing

• An activity to assess the quality of a system

• Using simple scenarios that can be understood
• Each test case is an executable example of system behavior

• Each example can help in stakeholder communication

• Throughout the full development cycle
The Set of Examples is Incomplete

- Too much data
- Too many combinations
- Too many paths

- Properties of interest fundamentally undecidable
Testing can be Manual or Automatic

Manual Testing
• Clever test case design
• Interaction with system inspiration for new tests
• Human oracle
• Single test case execution
• Limited data

Automated Testing
• Clever test case design
• Specs, models, & code used to derive test cases
• Automated oracle needed
• Test execution easily repeatable
• Massive input data possible

Clearly, we need both
Manual Testing: 
*Exploratory Software Testing*

- Human tester,
  - using brain, fingers, & wit
  - to create scenarios that
  - will cause software either to fail or to fulfill its mission.

- Take advantage of human cleverness
  - *No scripts: Exploratory Testing*
  - Record test findings as you go
Exploratory software testing

- is a style of software testing
- that emphasizes the personal freedom and responsibility of the individual tester
- to continually optimize the value of her work
- by treating
  - test-related learning,
  - test design,
  - test execution, and
  - test result interpretation
- as mutually supportive activities
- that run in parallel throughout the project.
Testing is about Varying Things: Input, State, Paths, Data, Environment

Input:
- Atomic versus abstract
- Input combinations
- Order of inputs
- Legal versus illegal
- Input filters & checks
- Normal versus special
- Default / user supplied

State
- History of stimuli

Paths
- Routes through system

Data
- Evolution over months

Environment
- Simulate the real world
The “Tour” Metaphor

Touring

• Guidebook tour
• Money tour
• Landmark tour
• After hours tour
• Museum tour
• Rained-out tour
• Couch potato tour
• Antisocial tour

Testing

• Use the manual
• The money-generating features
• Key features
• Batch functionality
• Legacy features
• Start and then cancel operations
• Do as little as possible (all defaults)
• Known bad inputs

http://blogs.msdn.com/b/james_whittaker/
Can we change a mosaic, tile by tile, in a team of 25+ developers and keep its structure?

An evolving product needs automated testing.
Test Strategy

• System model
  – Abstraction of system to reason about it

• Fault model
  – Likely faults that can be related to system model

• Test procedure
  – Systematic steps targeting likely faults via system model

• Adequacy criterion
  – Measure to tell to what extent strategy has been applied
Test Adequacy

Functional
- All use cases
- All decision paths
- All states
- All boundary values
- ...

Structural
- All methods
- All statements
- All code branches
- All def-use pairs
- ...

Test adequacy criterion gives rise to test obligations.

Test coverage: percentage of obligations that are met.
Test Week Objectives?

• Write test cases
• Achieve coverage goals

• Create / update models
• Identify relevant test strategies
• Analyze failures to identify *root cause*
• Rethink embedding of testing in dev. cycle
• Rethink design to improve testability
Combinatorial Testing

• What to do about the combinatorial explosion?


Tests done (2.5 bill. y)
### Input Handling / Combinational Testing

**Geavanceerd zoeken**

- **Zoekresultaten**
  - met alle woorden
  - met de exacte woordcombinatie
  - met een van deze woorden
  - zonder de woorden

- **Taal**
  - Alleen pagina's geschreven in het

- **Regio**
  - Pagina's zoeken in:

- **Bestandsformaat**
  - Alleen resultaten weergeven in het bestandsformaat

- **Datum**
  - Pagina's weergeven die zijn bekeken in de

- **Waar**
  - Toon resultaten als mijn zoekopdracht voorkomt

- **Domein**
  - Alleen resultaten weergeven van de site of het domein

- **Gebruiksrechten**
  - Resultaten weergeven:

- **SafeSearch**
  - Geen filters
  - Gebruik SafeSearch

**Pagina-specifiek zoeken**

- **Soortgelijk**
  - Zoek pagina's die lijken op de pagina

**Links**

- Zoek pagina's met link naar de pagina

**Specifiek zoeken per onderwerp**

- [nieuw Google Code Search](http://www.google.com/code_search) - Zoeken in openbare broncode
3-Way vs 2-Way Interaction

Server

IIS
Windows
Linux
Oracle
MySQL
Apache

OS

Database

full combi’s: $2^3 = 8$ triangles
### Possible Pairwise Selection

<table>
<thead>
<tr>
<th>OS</th>
<th>Server</th>
<th>DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win</td>
<td>IIS</td>
<td>Oracle</td>
</tr>
<tr>
<td>Win</td>
<td>Apache</td>
<td>MySQL</td>
</tr>
<tr>
<td>Linux</td>
<td>Apache</td>
<td>Oracle</td>
</tr>
<tr>
<td>Win</td>
<td>IIS</td>
<td>MySQL</td>
</tr>
<tr>
<td>Linux</td>
<td>Apache</td>
<td>MySQL</td>
</tr>
</tbody>
</table>
Pairwise Test Strategy

• System model: feature diagram
• Fault model: feature interactions
• Procedure:
  – Each test case addresses N types of features
  – Ensure all 2-way possibilities are addressed
  – Unless impossible
• Adequacy:
  – Percentage of pairs covered.
2-way interactions mean every possible pair of values from different parameters are tested at least once (unless the pair was marked invalid).

Total possible: **6,522,981,580,800 tests**

Hexawise Complete Coverage in just: **37 tests**
Random versus Pairwise

• Assume you need $N$ test cases to achieve 100% pairwise coverage

• How high would pairwise coverage be for \textit{random} set of $N$ cases?

• ... around 95%!

Bach & Schroeder, 2004
Statecharts

- Harel, 1987
- Scaleable state diagrams
- OR & superstates
- AND & concurrency
- (Partly) incorporated in UML

STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*

David Harel
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel

Communicated by A. Pnueli
Received December 1984
Revised July 1986

Abstract. We present a broad extension of the conventional formalism of state machines and state diagrams, that is relevant to the specification and design of complex discrete-event systems, such as multi-computer real-time systems, communication protocols and digital control units. Our diagrams, which we call statecharts, extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication. These transform the language of state diagrams into a highly structured and economical description language. Statecharts are thus compact and expressive—small diagrams can express complex behavior—as well as compositional and modular. When coupled with the capabilities of computerized graphics, statecharts enable viewing the description at different levels of detail, and make even very large specifications manageable and comprehensible. In fact, we intend to demonstrate here that statecharts counter many of the objections raised against conventional state diagrams, and thus appear to render specification by diagrams an attractive and plausible approach. Statecharts can be used either as a stand-alone behavioral description or as part of a more general design methodology that deals also with the system's other aspects, such as functional decomposition and data-flow specification. We also discuss some practical experience that was gained over the last three years in applying the statechart formalism to the specification of a particularly complex system.

1. Introduction

The literature on software and systems engineering is almost unanimous in recognizing the existence of a major problem in the specification and design of large and complex reactive systems. A reactive system (see [14]), in contrast with a transformational system, is characterized by being, to a large extent, event-driven, continuously having to react to external and internal stimuli. Examples include telephones, automobiles, communication networks, computer operating systems, missile and avionics systems, and the man-machine interface of many kinds of ordinary software. The problem is rooted in the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and

* The initial part of this research was carried out while the author was consulting for the Research and Development Division of the Israel Aircraft Industries (IAI), Lod, Israel. Later stages were supported in part by grants from IAI and AD CAD, Ltd.

0167-6423/87/33.50 © 1987, Elsevier Science Publishers B.V. (North-Holland)
(UML) State Diagrams

- State
- Transition
- Event
- [Condition]
- /Response
- Initial state

- State
- Transition
- Event
- [Condition]
- /Response
- Initial state
State-Based Testing Strategy

- System model: Statechart
- Fault model:
  - incorrect transitions or responses
- Procedure:
  - Unfold loops into “transition tree”
  - Cover all paths from root to leaf
- Adequacy:
  - All roundtrip-paths
## Built-in Testing

- **During design**, explicitly consider:
  - Preconditions
  - Postconditions
  - Class and other structural invariants

- Use assertions to enforce these at run time:
  - Increase *fault sensitivity* and *redundancy*
  - Serve as *oracle* for (automated) testing
Design By Contract

• Contract metaphor:
  – **Contract** is an explicit statement of the rights and obligations between a **client** and a **server**

• Server perspective:
  – If you call me and meet my **precondition**, I ensure that after returning I deliver a state in which my **postcondition** holds
  – If not, you’re on your own.

Pre, post & invariants in subclass:

Weaker precond: \( P \Rightarrow P' \)  
"Require no more": \( P' = P \) or \( P'' \)

Stronger postcond: \( Q' \Rightarrow Q \)  
"Ensure no less": \( Q' = Q \) and \( Q'' \)

Stronger invariant: \( I' \Rightarrow I \)
‘Polymorphic Server’ Test Strategy

• System model: class diagram
• Fault model
  – Subtype breaks behavior
• Procedure:
  – Parallel class hierarchy for testing
  – With specialized factory methods
• Adequacy:
  – Every subclass tested in context of superclass expectations
Example: Persistence in JHotDraw

- Drawing framework in Java
- Play garden for design patterns
- Each figure / drawing knows how to store / resurrect itself
- www.jhotdraw.org
```java
@Test void tstWriteRead{
    Storable s1 = make();
    Storable s2 = make();
    File f = new File(..);
    s1.write(output(f));
    s2.read(input(f));
    assertTrue(
        equalsTo(s1, s2));
}
```
SRP: Single Responsibility
  • One reason for change

OCP: Open-Closed
  • Implementation open
  • Interface closed

LSP: Liskov Substitution
  • Behavioral subtyping

ISP: Interface Segregation
  • Multiple interfaces

DIP: Dependency Inversion
  • Hollywood
Contract Styles

**Tolerant** (= Defensive)
- Server doesn’t trust its clients!
- Contract includes well-defined exceptions upon non-standard behavior
- Client can’t break server
- Extensive server-side checking
- Usually: duplicate checking

**Demanding** (for client)
- Server assumes it can trust its clients
- Lean, minimal contract
- Non-standard behavior not in contract
- Contract breach leads to undefined response
- No duplicate checking

Testing for Exceptions

• **Only if they’re part of the contract**

• **Fault model:**
  – Incorrectly raised / thrown
  – Incorrectly handled / caught

• **System model:** *exception flow*

• **Procedure must address controllability:**
  – Stub interfaces to trigger exceptional conditions
Design for Testability

• Interfaces
• Tell, don’t ask
• A “mock” keeps track of how it is used
  – Test cases can set `expectations’ on, e.g. calls

• Dependency injection

Mocking framework: http://jmock.org
Fighting Poor Testability

• “I Can’t Get This Class into a Test Harness.”

• “This Class Is Too Big and I Don’t Want It to Get Any Bigger”

• “I Need To Make a Change but I Don’t Know What Tests To Write.”
Finale: Testing Research at TU Delft

[ Selection ]
Crawljax: Testing Ajax Apps
Monitoring The Testing Process

Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining.

Understanding Testing Processes in Eclipse
ETSE: Eclipse Test Suite Explorer
Summary

- Test case design = selecting examples

- Reflect on your strategies: Fault & system model, procedure, adequacy.

- Can you test interfaces, exceptions, combinations, state machines?
A Test Week Ahead of You!

- What testing do you want to do?
- What challenges do you expect?
- Which successes do you want to celebrate?