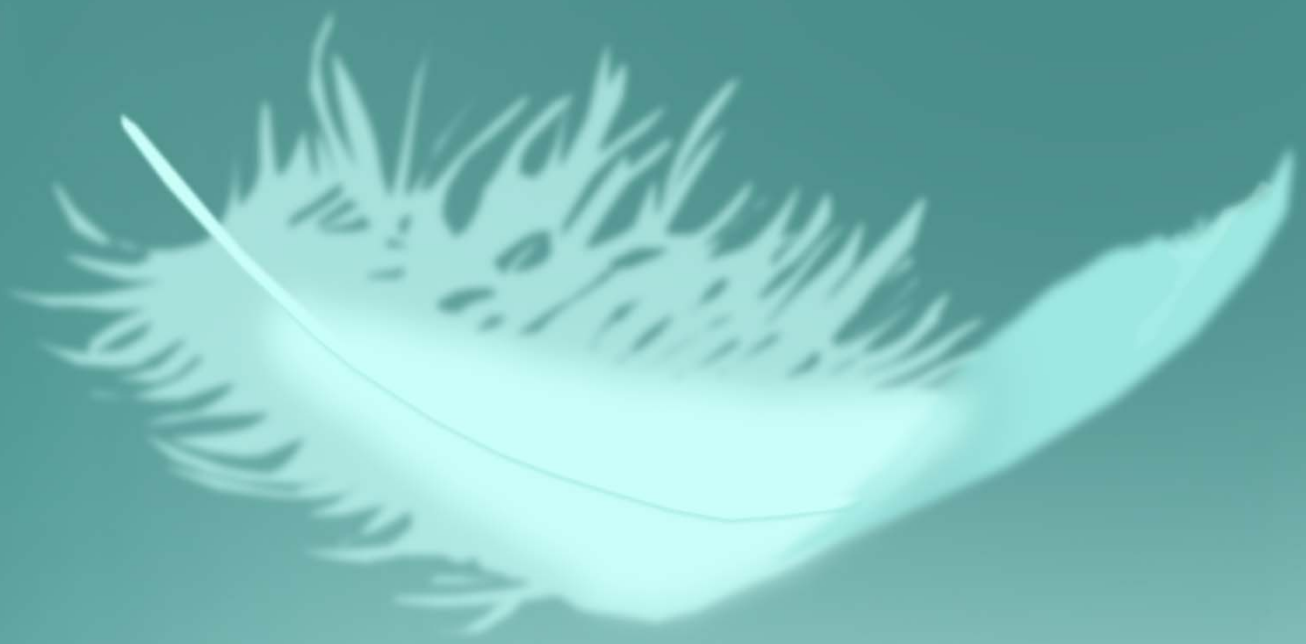
BREDEX
Software-Entwicklung und Beratung



Handling failures in automated acceptance tests

Alexandra Imrie, BREDEX GmbH

 Version 2.3

Introductions

- **BREDEX GmbH, Braunschweig (Germany)**
  - Software development and consulting since 1987
  - Java-focus since 1995
  - Quality assurance
  - Agile processes
- **Alex Imrie**
  - Linguistics graduate (York, UK)
  - Customer support, demonstrations, training, perspective
  - Test coach and tester

The story

- **What tests should be telling us**

  And what stops them doing this
- **Where events come from**

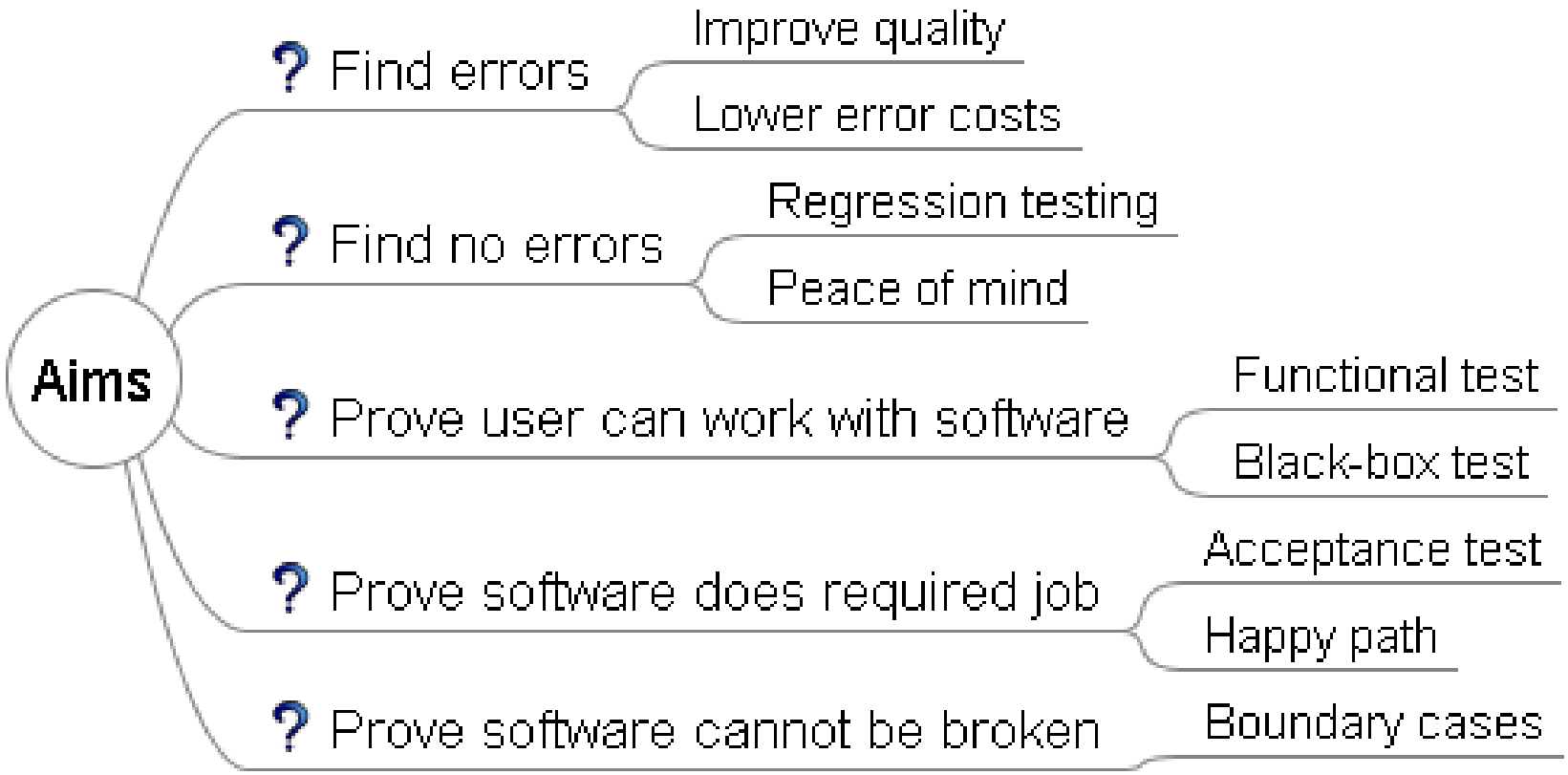
  And how to avoid them where possible
- **Catching events before they happen**
- **Dealing with events**

  Test design

  Event handling strategies

  Within the project

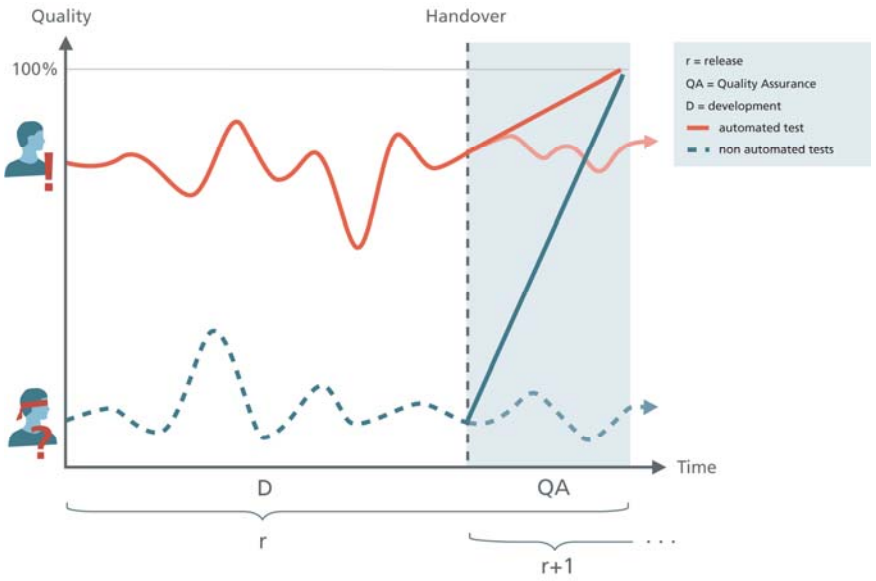
Aims of acceptance testing



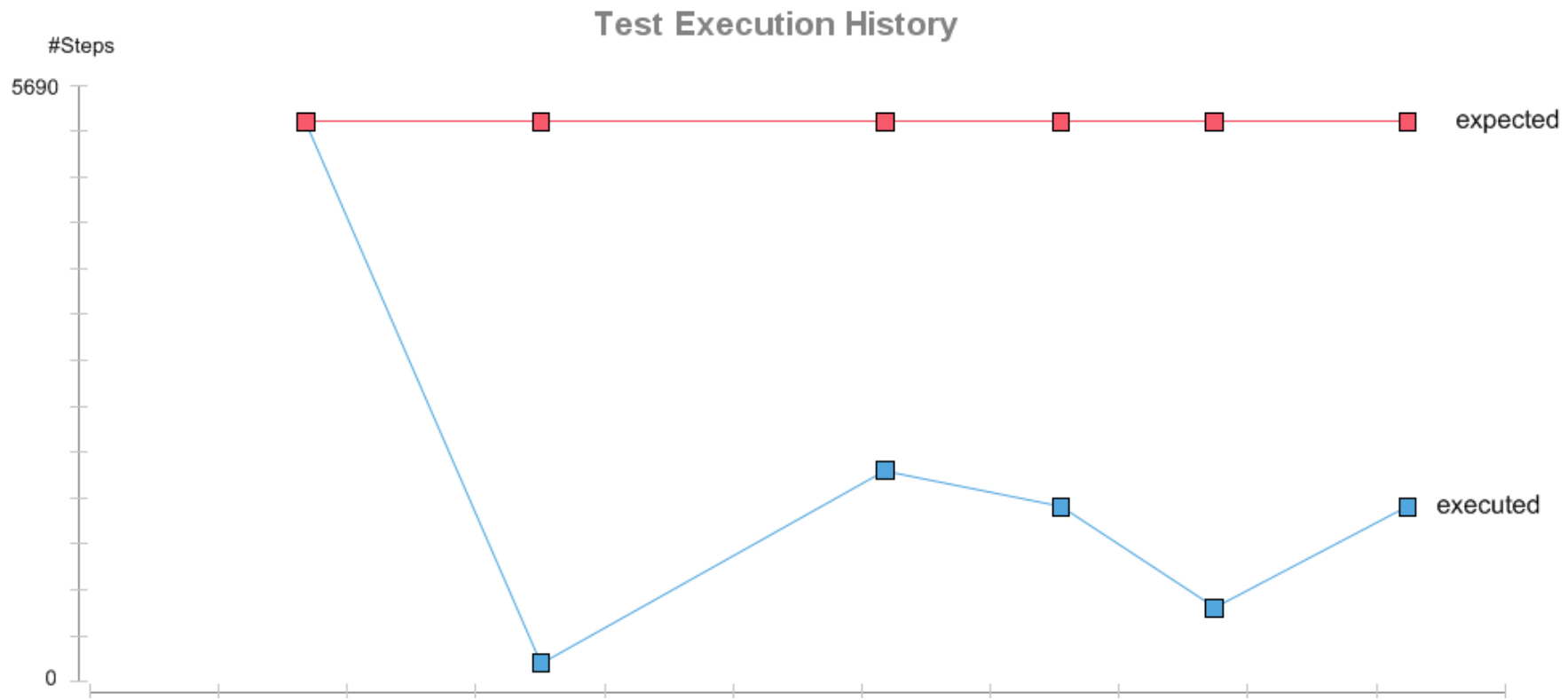
Whatever tests are telling us…

- **Has to be constant!**
- **Only with knowledge can we**

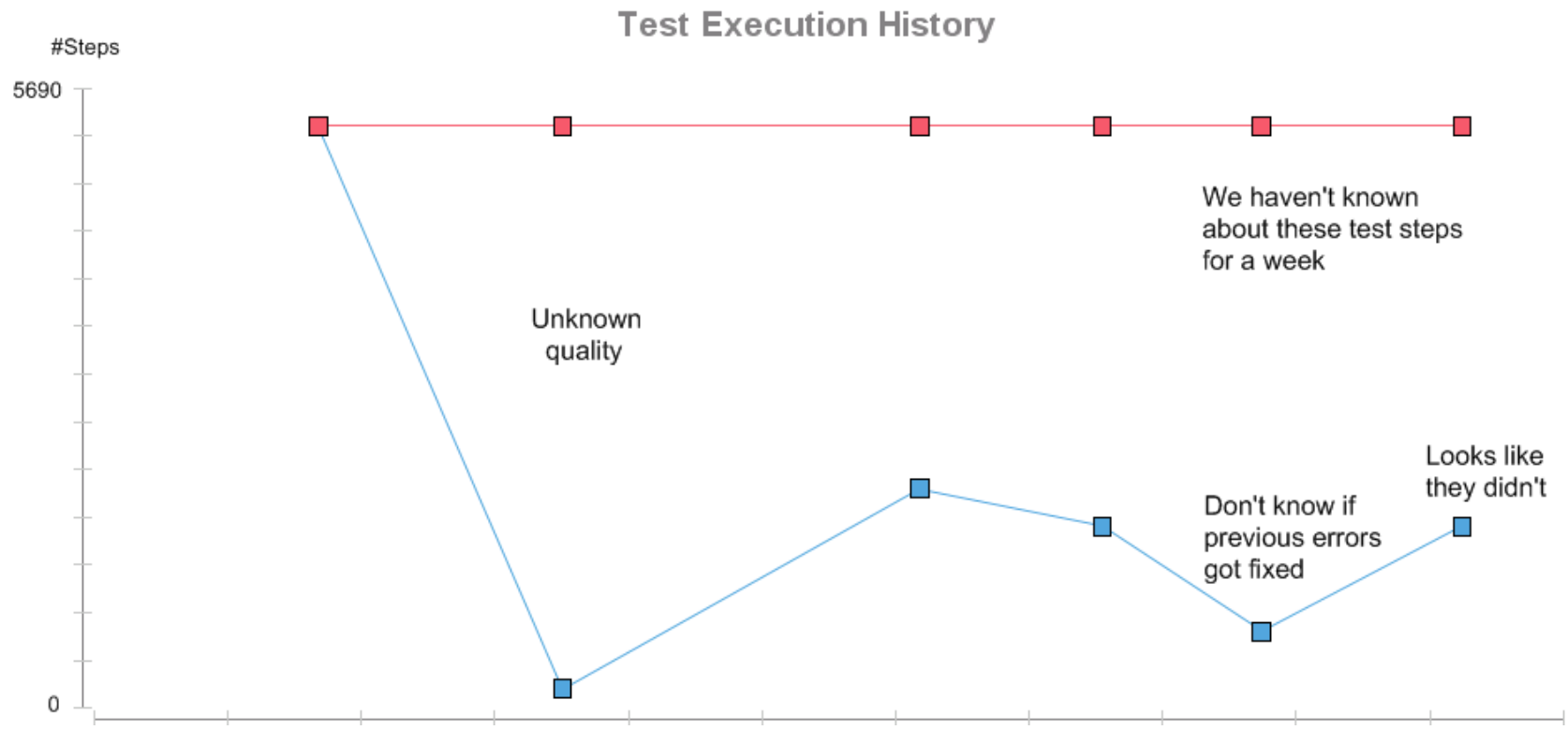
  Monitor & improve quality
- **→ Automation for continuous feedback**



Dealing with errors is important



Dealing with errors is important



Let's talk about errors: definitions

- **EVENT**

Causes test to deviate from expected execution: a discrepancy between expected state/behaviour and actual state/behaviour

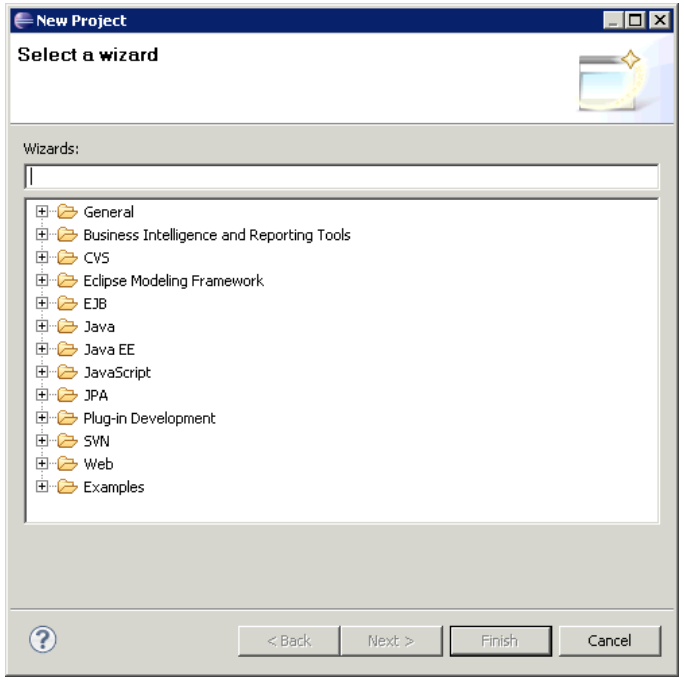
Definitions

- **EXCEPTION**

  A failed assertion (expected & actual value / status not same)

  An event handler can intervene & test can continue if status can be changed

- **→ check finish button active**



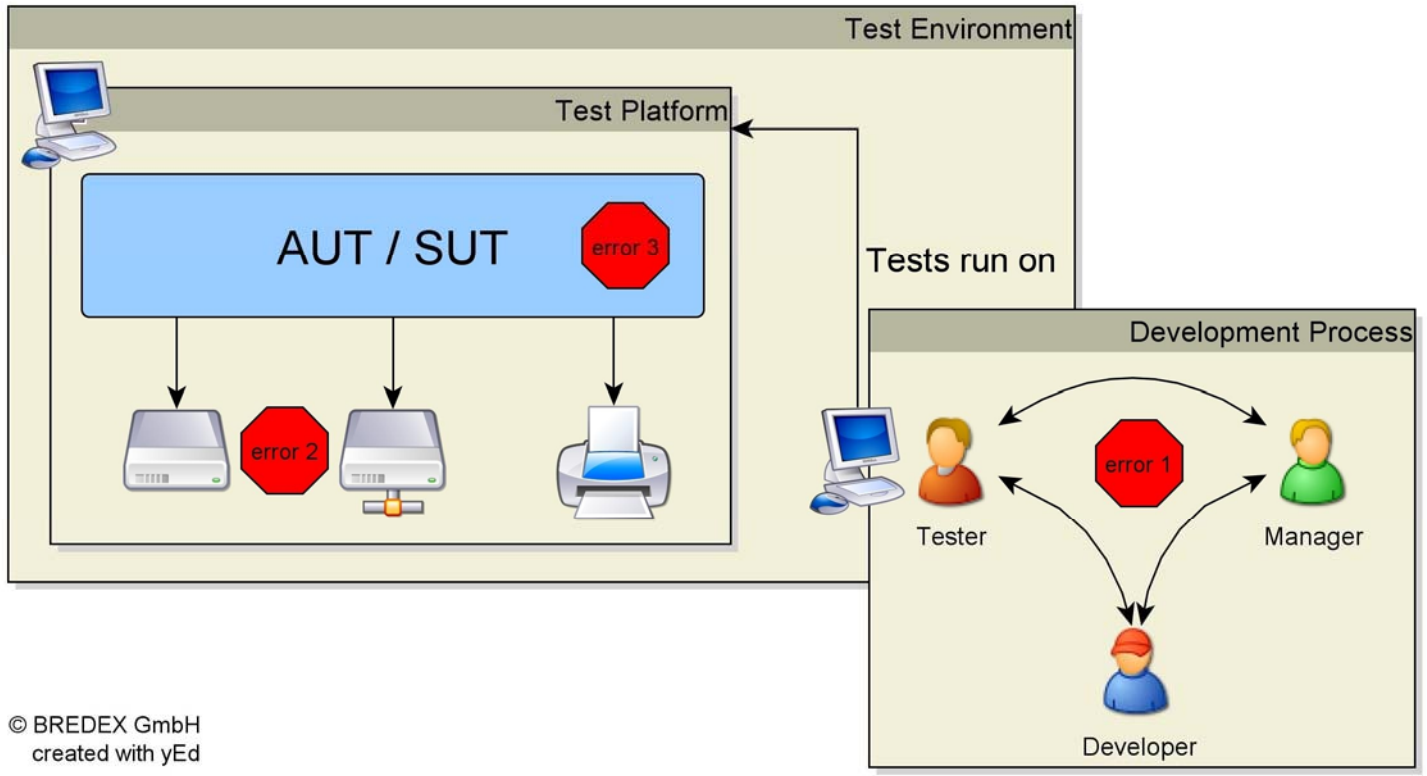
Definitions

- **ERROR**

  An event that can't be dealt with in the test execution.

  State or value cannot be changed, or unknown state of application

Where do errors come from?



Exterminating events?

- **Errors in test can be minimized**
  - Better communication reduces misunderstandings
  - Tester presence in meetings
- **Errors in environment**
  - Dedicated, well-defined test environment
  - Data preparation strategy
  - Deterministic tests

Exterminating development errors?

- **Dev errors can't be totally avoided**

  → must be dealt with in the test

- **But test design can help**

  Robust synchronisation

  Manual tester intelligence built in

- **Explicit understanding of how app works**

  Logic, behaviour, interface information display

Be prepared! (and slightly paranoid…)

- **Different states**

  Execution

  Verification

- **Different behaviours**

  Execution

  Verification

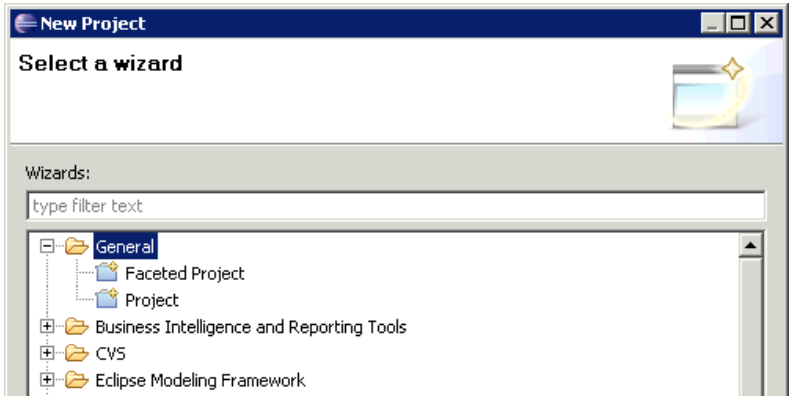
  Synchronization

Different states

Execution

Invalid data:

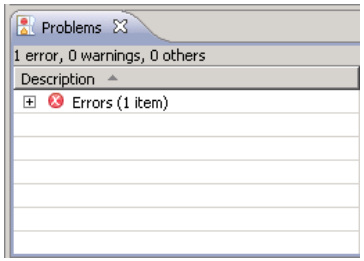
General / Test Project



Verification

Data

Check label: 0 items



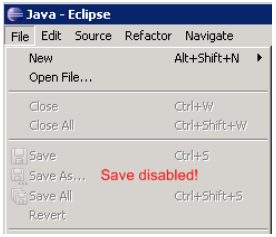
Enablement

Check save button disabled

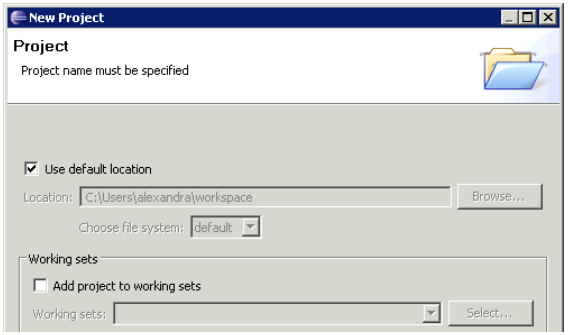
Different behaviours

Execution

Menu entry can't be selected

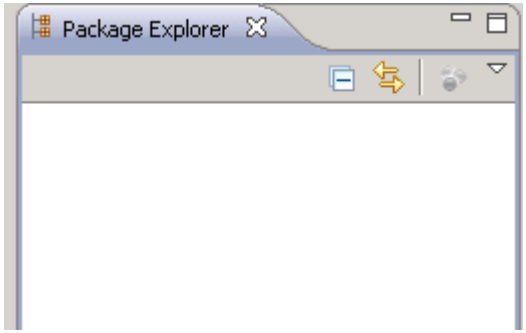


Component not found



Verification

Project not created



Synchronisation

Progress dialog still open after 10 seconds

Using verification to avoid / catch errors

- **Easiest to react to failed state verifications**

  React to real issue, not inherited problem

  Easier to analyse and deal with

  If state can be corrected, test can continue

  (with success or not)

Using verification to avoid / catch errors

- **Easiest to react to failed state verifications**

  Wherever possible, include state verification

- **Before action:**

  Check existence / enablement of component

  Check existence of data before selection

- **After action:**

  Check all results (enablement)

  Also include behaviour verification (effects in application)

Structuring tests for verification

Verify and click any button		
<BUTTON>	Verify existence	true
<BUTTON>	Verify enablement	true
<BUTTON>	Click	

Structuring tests for verification

Select any path from any tree		
<TREE>	Verify existence	true
<TREE>	Verify enabled	true
<TREE>	Verify <PATH> existence	true
<TREE>	Select <PATH>	
<TREE>	Verify <PATH> selected	true

Checking challenges and advantages

- **Good structure**

  reusable, flexible

- **Discipline**

- **“Unhelpful“ GUI**

  Design for testability

- **Failed checks can often be handled as exceptions**

  React to actual error, not inherited

  Easier to analyse

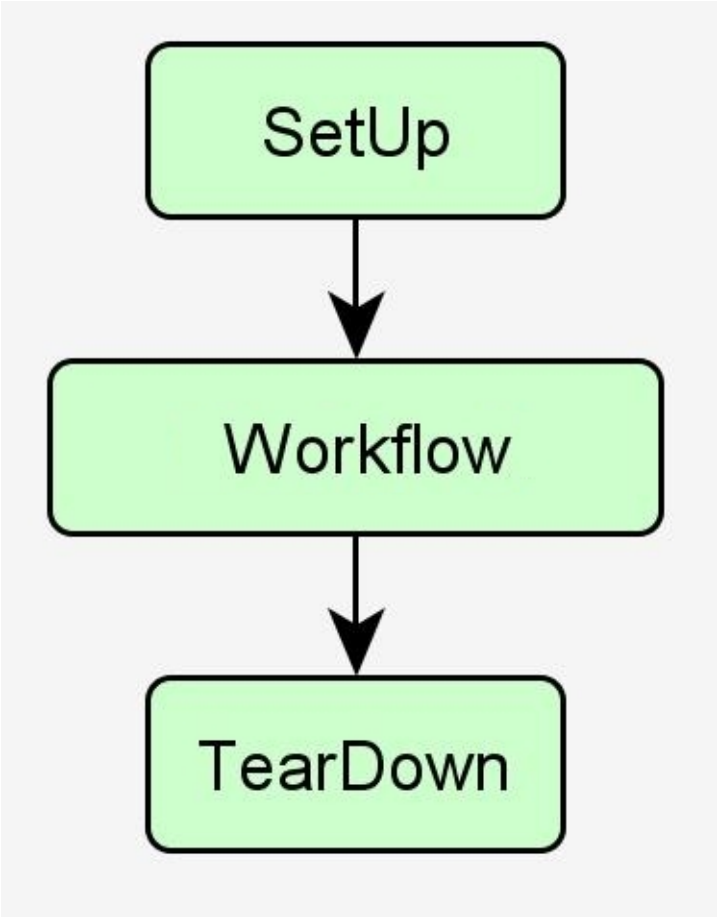
  Easier to handle

- **If state can be corrected, test can continue**

  Either +ve or -ve

Test design to support event handling

- **Especially for errors**
  - Can‘t be handled
  - Test can‘t continue
- **Independent use cases**
  - Specific structure



Some thoughts on set up / tear down

- **Managing data and preconditions**

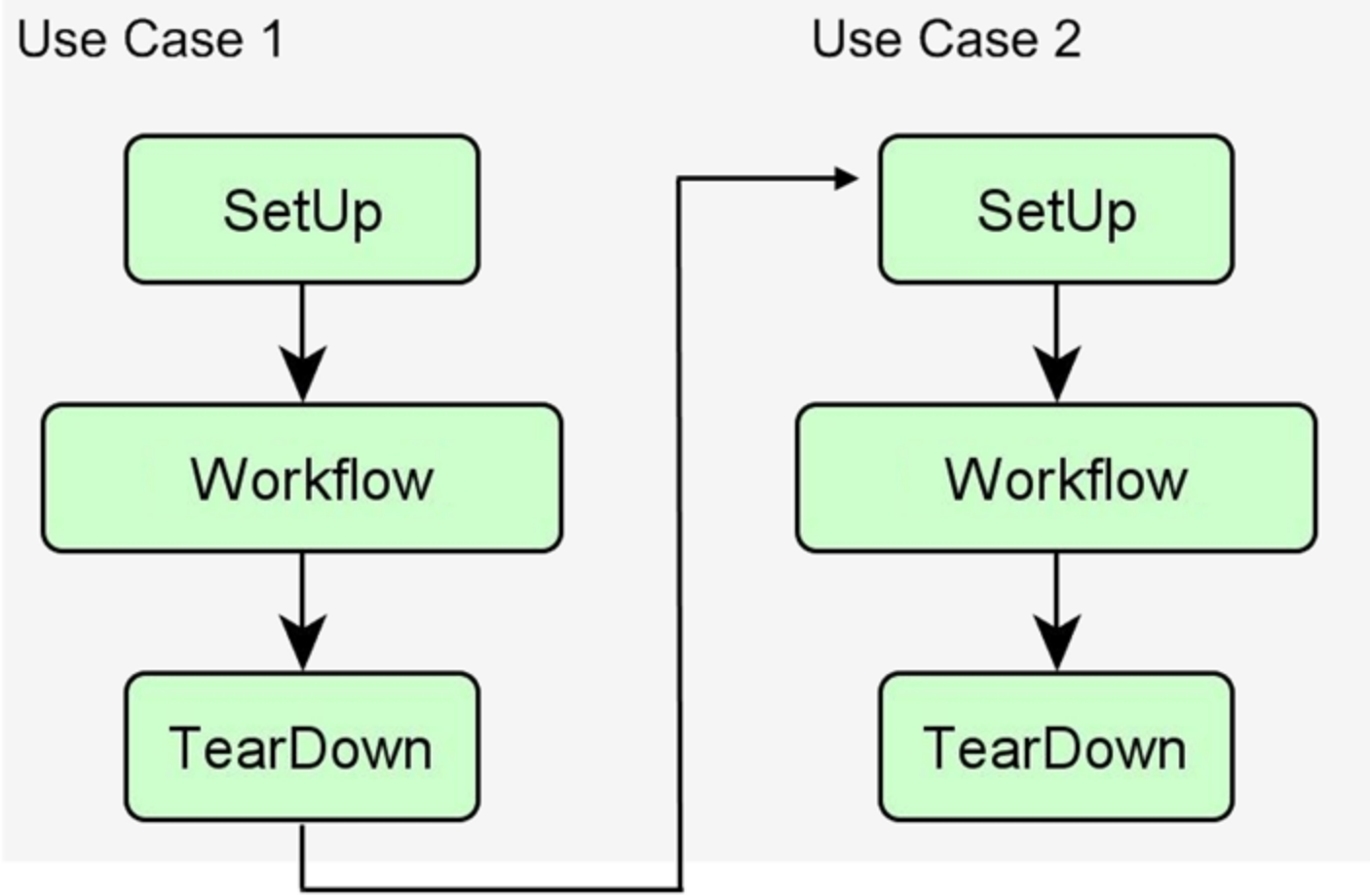
  Startup test case

  Data made available

- **Minimizing dependencies**

  Or at least be aware of them!

Handshake


Use Case 1
SetUp
Workflow
TearDown
Use Case 2
SetUp
Workflow
TearDown

Two event handling strageties

- **Local handling**

Exceptions

Expected or known states

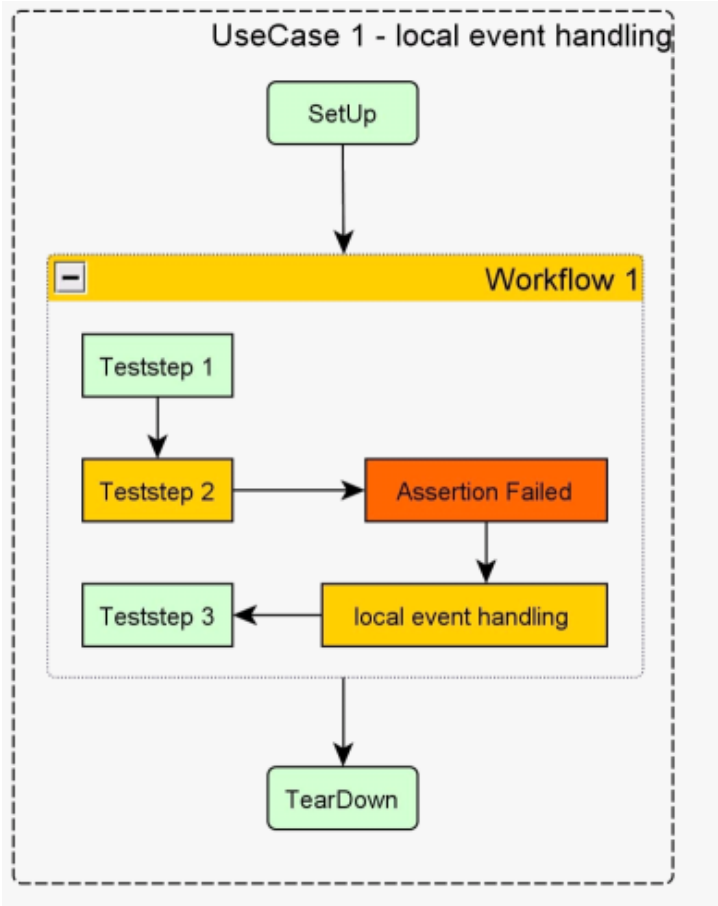
Small problems / inconsistencies

- **Can be resolved in test**

→ *test successful*

- **Or can be ignored**

→ *test unsuccessful, but complete*



Two event handling strategies

- **Global handling**

Errors

Unexpected

Unknown state of app results

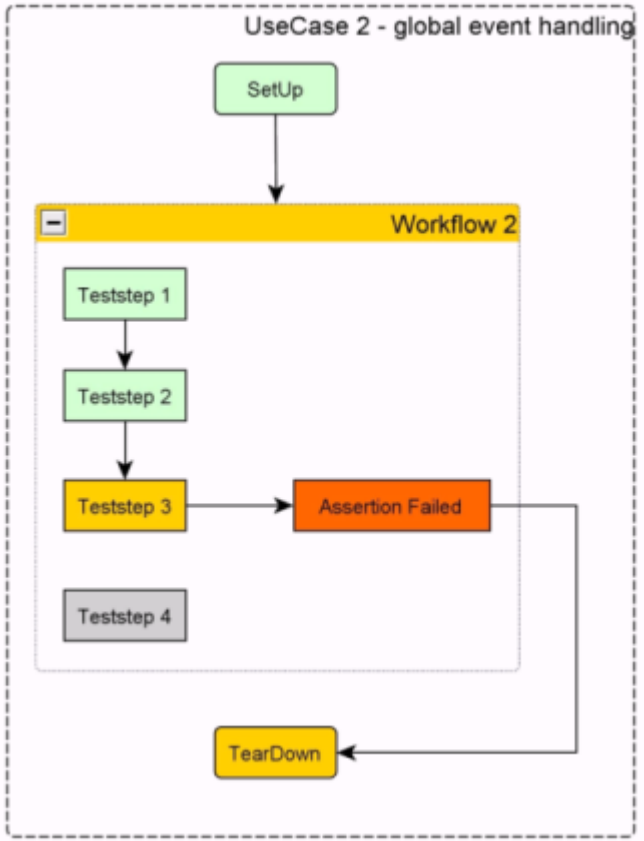
→ inherited errors

- **Strategy**

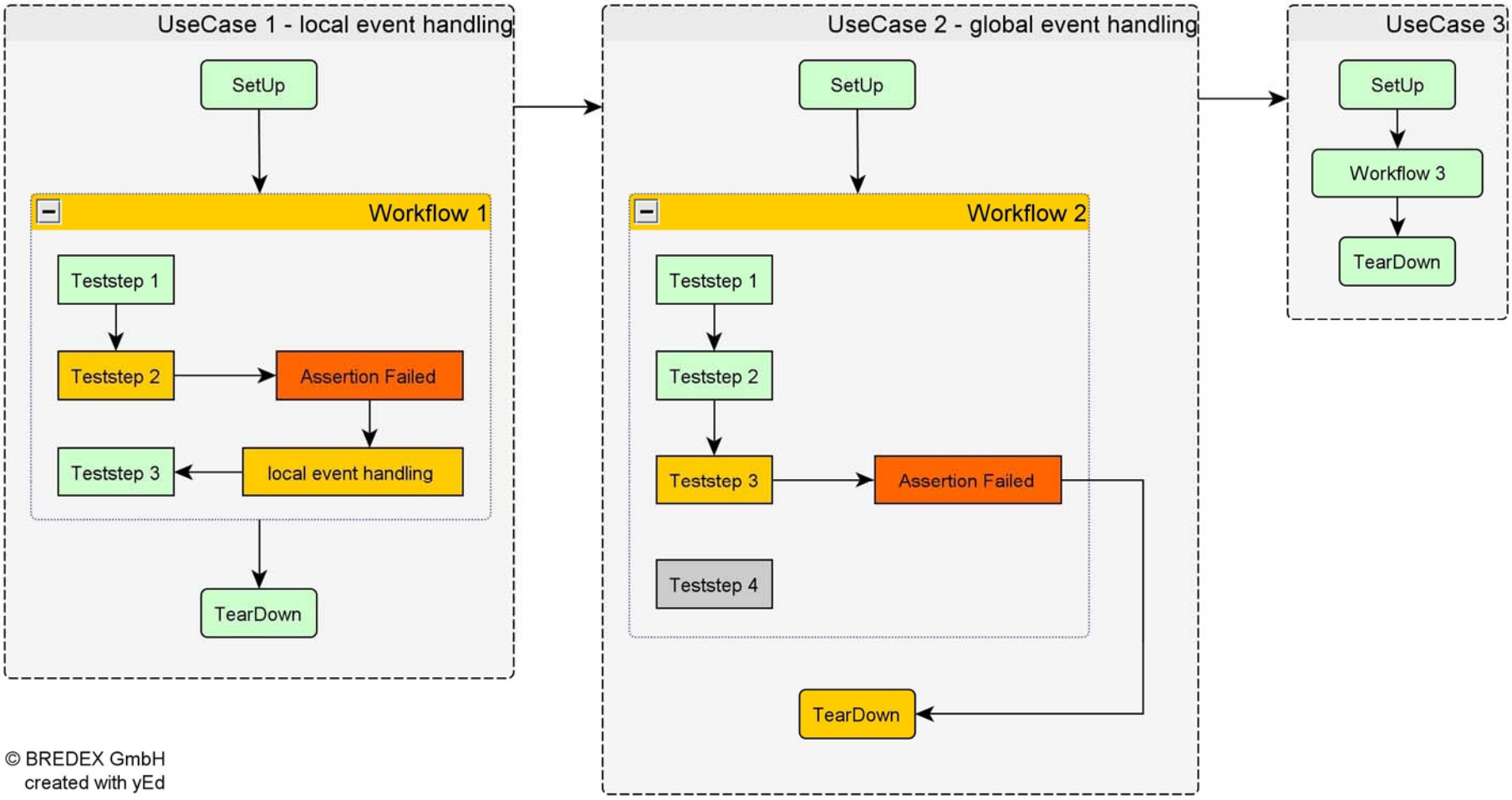
Document

Leave use case

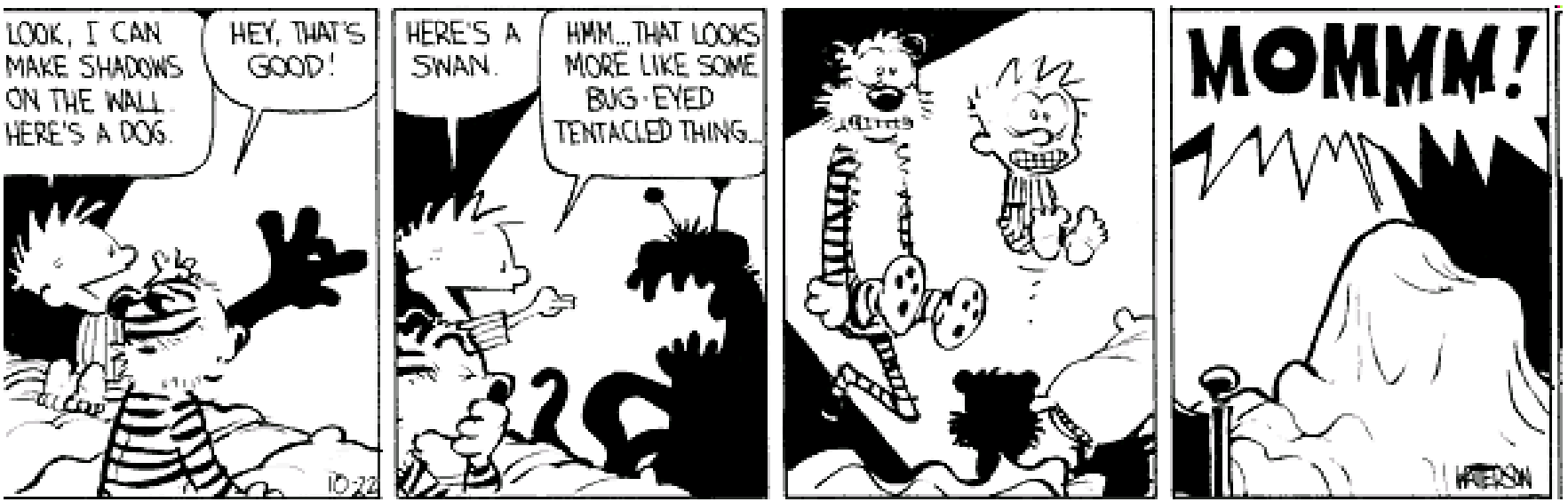
Prepare app for next use case



Test continues after exception or error



Seeing bugs under the bed?



The Essential Calvin and Hobbes, Bill Watterson

Bugs under the bed…

- Start with global event handling
- Add local as required

Dealing with errors – process

- **Fix it instantly**

  Close to development – time / money saved

  Dev probably still working on code

  Affected test up and running quickly

- **Watch out for error driven development!**

Dealing with errors – process

- **Write a ticket**

  Better resolution planning (complex errors, higher risk)

  Follow progress

  Remove use case (independent!) from productive tests

  → “Broken“ tests

- **Watch out for accumulation of tickets!**

  And more tests in broken than in productive

  Whole areas of test not running at all

Don’t test around / ignore the error!

- **Pressure from dev team**

  More errors = more stress

- **Pressure from management**

  All tests green

Conclusion

Always know your quality!

- **Global event handling & independent use cases**

  For maximum use case completion

- **Local event handling**

  For maximum test coverage within use case

Thank you!

- **Any questions?**