Why are we here?

- To take advantage of the commonality identified in the product line feature set.
- To explore successful processes and techniques for reducing test effort, while maintaining or even improving quality, through reusable assets and strategic techniques.
- Reduce the test effort. Test effort estimates range from 50 - 200% of the effort needed to build the actual production system
- If effective use of test assets can reduce test effort by 10%, this represents substantial savings. In fact we can reduce the effort by considerably more than that depending upon the initial maturity

Who am I? – John D. McGregor

- Worked on large software systems – 500+ developer FTE
- Worked on always-available software projects
- Working on early and current product line efforts
- Domains:
  - telephony/wireless
  - cardiac surgery planning
  - insurance/financial
  - ground satellite software
  - ...

Major issues

- Managing variations
- Amortizing efforts/resources over multiple products
- Manage and sustain large numbers of product and test configurations
- Coordinating the complementary perspectives of product builders and core asset builders
- Keeping pace with product developers both in terms of throughput and cost reductions
- Achieving high levels of reuse

Outline

- Product line concepts
- Testing concepts
- Reviews and inspections
- Specific techniques
- Evaluation
- Case Study
- Conclusion

PRODUCT LINE CONCEPTS

Product Line Definition

- *A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

Product Line Testing - 1

- *sharing a common, managed set of features*
  - *System tests should be tied to features to facilitate using the tests with many products*
  - *These tests are managed in parallel to the production assets to keep them synchronized*

Product Line Testing - 2

- *particular market segment or mission*
  - *The culture of the market segment determines specific quality levels*
  - *Which in turn determines test coverage criteria*

Product Line Testing - 3

- *common set of core assets*
  - *There are test-related core assets including test plans, test cases, test harnesses, test data, and test reports, but*
  - *These assets are designed to handle the range of variability defined in the product line scope*

Product Line Testing - 4

- *a prescribed way*
  - *The test infrastructure must be compatible with the production strategy and the rest of the product line infrastructure*

Variation - 1

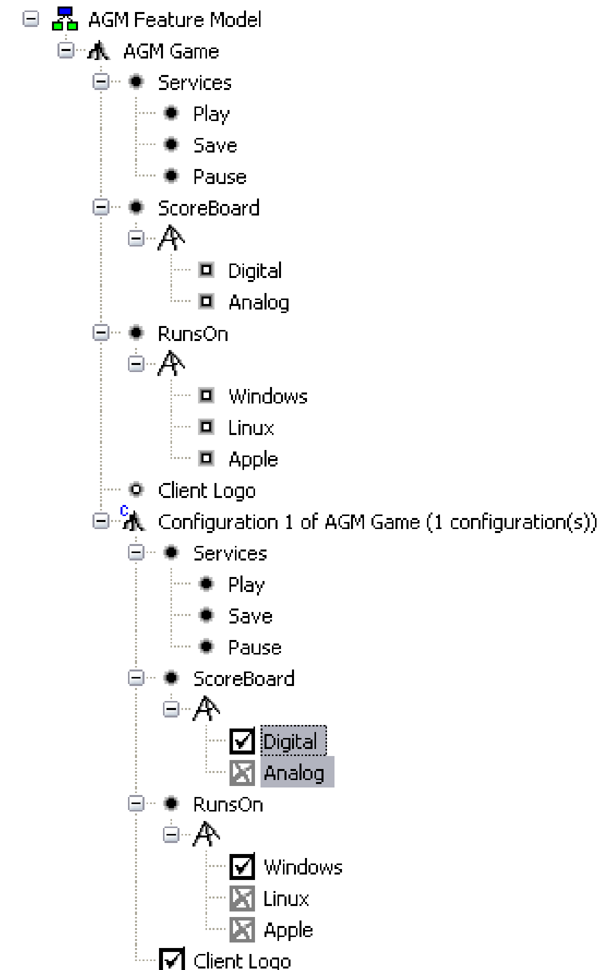
- The products vary from one another at explicit, documented variation points.
  - Some specification exists that defines the points; machine readable would be nice
- Variation among products implies variation among tests.
  - The variation points in the product will become variation points in the test cases.
- Variation between the behavior of the product software and the test software should be minimal
  - Architecture of the test case implementations should reflect the product architecture

Variation - 2

- The specific variant to be used at a variation point is selected (bound) at a specific point in time.
  - Every variation point has a specified binding time(s) and how software that contains each variation point is tested depends in part on the binding time.
- Test artifacts must be developed that exercise this binding time and evaluate the correctness at that point.
  - The test infrastructure must provide multiple facilities for testing.
- Managing variation, particularly limiting the range of variation, is as essential to testing as it is to development.
  - Each variation point has a specific set of possible values that are clearly specified.

Variation representation

- Feature models are used to represent the commonality and variability among products in a product line.
- Unfilled circles represent optional features.
- The bottom portion shows a product configuration. All variation has been resolved.



Variation representation

Variability Model

Variation Point

Variant

requires

requires

Con-
straint

VP1 Regis-
ter Type

1..2

V1.1 Simply

V1.2 Completely

VP5 Search
Options

1..1

V5.1 Hints

V5.2 Similar
Results

VP2 Bonus

0..1

V2.2 Points

V2.1 Deduction

VP3 Payment
Type

1..2

V3.1 Invoice

V3.2 Card

VP4 Card
Type

1..1

V4.1 Credit

V4.2 Debit

Customer

Register

Register simply

Register completely

Search goods

Provide search hints

Show similar results

Buy goods

Bonus points

Direct price deduction

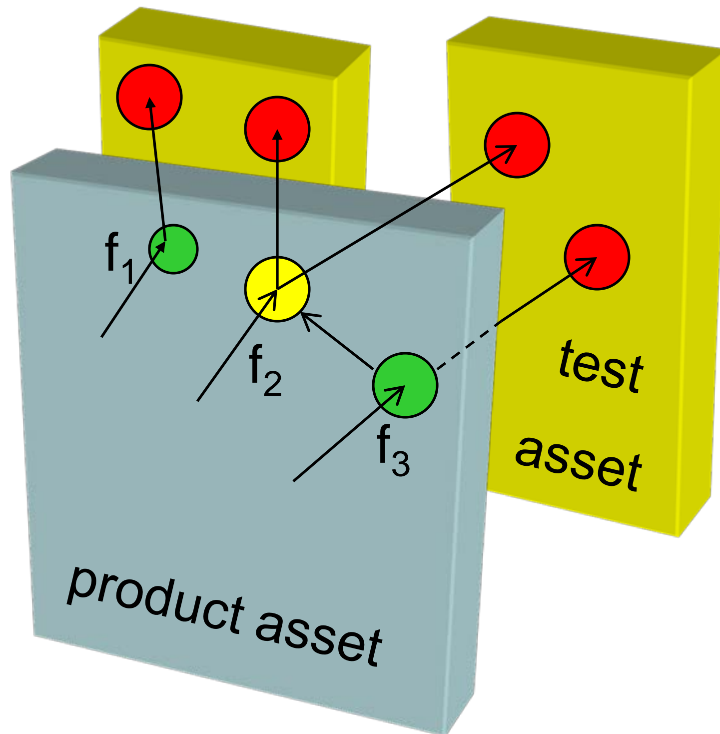
Pay by invoice

Pay by credit card

Pay by debit card

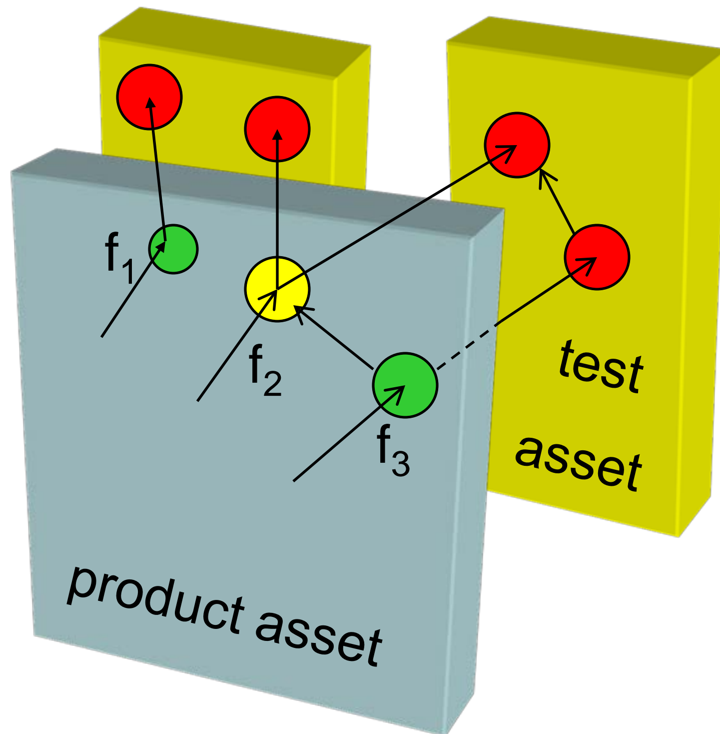
<<include>>

Feature interactions



- f_1 and f_3 are features that are unrelated. No variations associated with them interact.
- f_2 is a feature for which there are variation interactions with f_3.
- There are no interactions in the one test asset and only occasionally in the other.

Feature interactions



- f_2 and f_3 interact for some variants.
- These interactions are propagated to the test assets.

Development vs Test

work product

product line goals

Development view

Test view

Vary functionality

at variation points

Vary test cases

at variation points

Product Line vs Product

work product

product line goals

Product Line view

Product view

Higher degree of test coverage

Contradictions are resolved

Interactions

	Development	Test
Product line	Strategic reuse of product assets and test assets	Reusable across products Model-based testing
Product	Automated construction	Automated tests

Strategic reuse

- *The most powerful form of reuse is reuse of everything — not just code.* We've long known that we should be reusing designs, plans, checklists, role, etc; and McConnell reminds us that we should be reusing *processes* for developing systems.
  - "The Ten Most Important Ideas in Software Engineering" Steve McConnell

Reuse - 1

- Strategic reuse is the key to product line success
- This is as true for testing as it is for development
- One key is planning at the product line level
  - The product line test plan defines goals and strategies that are to be achieved through testing
  - This planning allows assets to be defined to handle the necessary variation at each variation point.
- A second key is structure
  - By structuring the test data, test cases, and other artifacts we can build models that are reusable

Reuse - 2

- Encapsulation
  - Use configurations so that the high-level structure is separated from the behavior
- Information hiding
  - Use good software design techniques, such as modularization, in designing test software.
- Abstraction
  - Use hierarchy to factor detail from concept – the higher in the hierarchy the more general the content

Reuse - 3

- Reuse can occur both vertically between the product line and the products and horizontally between products
  - vertical reuse propagates interfaces
  - horizontal reuse propagates opportunistic components

TESTING CONCEPTS

Testing Definitions

- Testing - the detailed examination of an item guided by specific information
- Testability – the ease with which software gives up its faults
- Test coverage – measure of the portion of the item under test that has been exercised by a set of test cases
- Defect density – measure of number of defects usually present per unit measure (e.g., 5 defects per 100 lines of code)
- Defect live range – the distance between where a defect is injected into the product and where the defect is detected

Testing Perspective - 1

- Systematic - Testing is a search for defects and an effective search must be systematic about where it looks. The tester must follow a well-defined process when they are selecting test cases so that it is clear what has been tested and what has not.
- Objective - The tester should not make assumptions about the work to be tested. Following specific algorithms for test case selection removes any of the tester's personal feelings about what is likely to be correct or incorrect.
- Thorough - The tests should reach some level of coverage of the work being examined that is "complete" by some definition. Essentially, for some classes of defects, tests should look everywhere those defects could be located.
- Skeptical - The tester should not accept any claim of correctness until it has been verified by testing.

Testing Perspective - 2

- This perspective is important because unit and some levels of integration testing are done by development people
- This perspective is important because reviews and inspections are conducted by people in a variety of roles

Basic test sequence

- Test activities
  - Test specification - business goals → test goals → test strategies
  - Test design and implementation – qualities of test software include clarity and traceability
  - Test execution – automation should be compatible with the automation of product production

View of activities each major milestone in development

Requirements

Use cases

Review

Analysis

Analysis models

Guided Inspection

Architectural Design

Architecture description

ATAM

Detailed Design

Design models

Guided Inspection

Coding

Implementation

Unit/Feature
Integration
System

Development

Testing

View of activities by levels

	Core assets	Product development
Organizational management	Shift traditional testing emphasis from backend to frontend.	Consider the impact of product sequence on development of assets
Technical management	Coordinate development of core assets with product development. What testing tools are delivered with the core assets?	Provide configuration management for test, as well as development, artifacts
Software engineering	Design for testability Make reusable assets	Use testing to guide integration

Testing strategy - 1

- Test strategy
  - determines how and how much to test

quality	tactic
quicker	automation, iteration
cheaper	automation, differentiated work force
better	more complete coverage
mass customization	combinatorial testing

Test strategy - 2

- Use a balance of functional/non-functional/structural approaches
- Invest in a balance of “product line” & “product” testing
- Devise tests for each desired quality
- Product line test coverage
  - All variant combinations
  - Pair-wise variant combinations
  - Selected variant patterns
- Use estimates of testability to give level of confidence in test results

Test risks

- The resources required for testing may increase if the testability of the asset specifications is low.
- The test coverage may be lower than is acceptable if the test case selection strategy is not adequate.
- The test results may not be useful if the correct answers are not clearly specified.

Test standards

- IEEE 829 – Test documentation
- IEEE 1008 – Unit testing
- IEEE 1012 – Verification and Validation
- IEEE 1028 – Inspections
- IEEE 730 – Software assurance

Test documentation

- **1. Preparation Of Tests**
- Test Plan: Plan how the testing will proceed.
- Test Design Specification: Decide what needs to be tested.
- Test Case Specification: Create the tests to be run.
- Test Procedure: Describe how the tests are run.
- Test Item Transmittal Report: Specify the items released for testing.
- **2. Execution of Tests**
- Test Log: Record the details of tests in time order.
- Test Incident Report: Record details of events that need to be investigated.
- **3. Completion of Testing**
- Test Summary Report: Summarize and evaluate tests.

Testing artifacts

- Plans
  - Usual levels – unit, etc
  - How test is coordinated between core assets and products
- Test cases
  - <pre-conditions, inputs, expected results>
  - Data sets
- Infrastructure
  - Must be able to execute the software in a controlled environment so that the outputs can be observed and compared to the expected results.

IEEE Test Plan - 1

- Introduction
- Test Items
- Tested Features
- Features Not Tested (per cycle)
- Testing Strategy and Approach
  - Syntax
  - Description of Functionality
  - Arguments for tests
  - Expected Output
  - Specific Exclusions
  - Dependencies
  - Test Case Success/Failure Criteria

IEEE Test Plan - 2

- Pass/Fail Criteria for the Complete Test Cycle
- Entrance Criteria/Exit Criteria
- Test Suspension Criteria and Resumption Requirements
- Test Deliverables/Status Communications Vehicles
- Testing Tasks
- Hardware and Software Requirements
- Problem Determination and Correction Responsibilities
- Staffing and Training Needs/Assignments
- Test Schedules
- Risks and Contingencies
- Approvals

Feature testing

- Depending on how development assignments are made, feature testing may be unit or interaction testing.
- “unit” is the smallest piece that will be tested. It is the work of one person.
- “integration” testing is done when the work of multiple people are joined together.
- Generally feature testing will have an integration aspect because multiple objects will be brought together to implement a feature.

Test design and variability

- Make variability explicit in the model
  - Encapsulate each variant
  - Use specific stereotypes or tags
- Make dependencies between variants and requirements explicit
  - Use the architecture
- Design fragments – the fragments correspond to variants
  - Each fragment is compete at one level

Designing for testability - 1

- The ease with which a piece of software gives up its faults
- Observability
  - Special interfaces
  - Meta-level access to state
- Controllability
  - Methods that can set state
  - Meta-level access to these state changing methods
- Product line
  - Separate test interface which can be compiled into a test version but eliminated from the production version

Designing for testability - 2

- Strategy must be based on language
  - Java – can use reflection
  - C++ - must have more static design
- Separate the interfaces for observing (get methods) and controlling (set methods)
- Compose component based on purpose; only add those interfaces needed for the purpose
- Couple the composition of tests with the composition of components

Using testability

- You have just tested your 1000 line program and found 27 defects.
- How do you feel?
- Have you finished?

REVIEWS AND INSPECTIONS

Guided Inspection

- Inspections and reviews are human-intensive examinations of artifacts that are either not executable (plans, designs,…) or that are executable but the purpose is to find problems, such as design non-conformance, that are not visible in execution.
- These examinations are biased toward finding fault with what is in the artifact. A review seldom finds what is not there.
- Guided Inspection augments typical inspection techniques with test cases that help determine whether everything that should be there is.
- The inspection is “guided” by the test cases.
- Guided inspection applies the testing perspective.

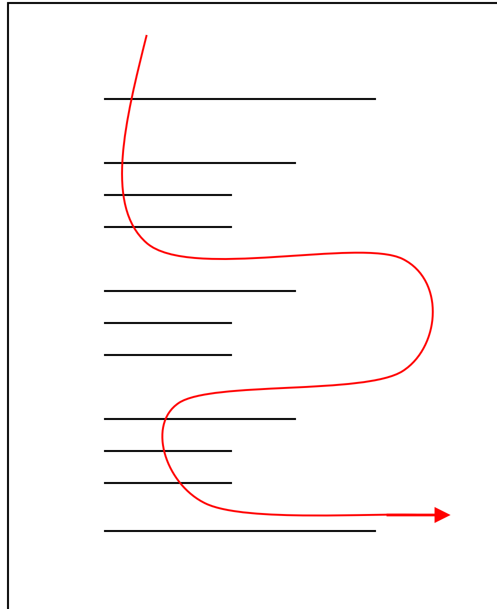
Guided Inspection

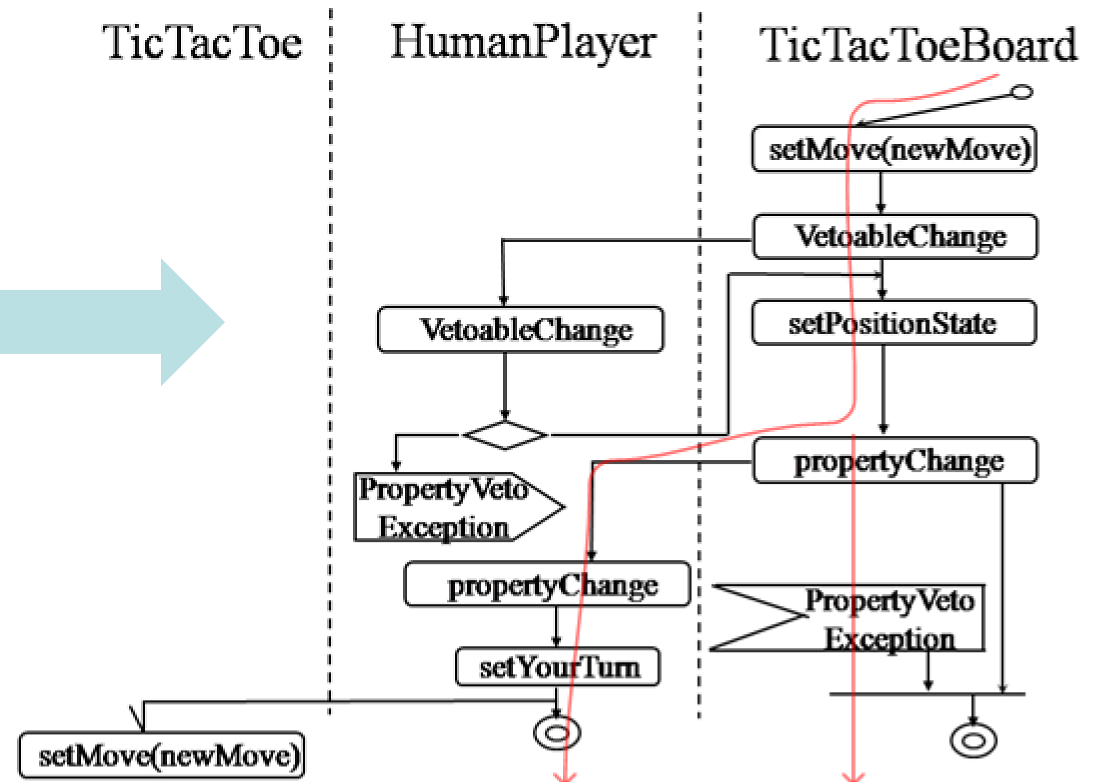
- "Guided" inspections search systematically for all necessary elements
- Process
  - The inspection team creates scenarios from use cases
  - The inspection team applies the scenarios to the artifact under test
  - For each scenario the design team traces the scenario through their design for the inspection team
  - The inspection team may ask for a more detailed scenario
  - The inspection report describes any defect discovered by tracing the scenario.

Review scenario

- The player wants to change the speed of the puck in the Brickles game. While the game is in progress, the player left clicks on the preferences menu entry, and uses the number spinner control to select the desired speed. When the OK button is left clicked, the new speed is saved and is now the speed used by the game.

Mapping from scenario to design





Architecture Tradeoff Analysis Method (ATAM)

- “Test” the architecture
- Use scenarios from the use cases
- Test for architectural qualities such as
  - extensibility
  - maintainability

Flow of activities in ATAM

Business Drivers
Quality Attributes
Scenarios
Software Architecture
Architectural Approaches
Architectural Decisions
Analysis
Tradeoffs
Sensitivity Points
Non-Risks
Risks
distilled into
Risk Themes
impacts

Architecture Analysis and Design Language

```
system ctasSoftwareLayer
   features
     mi: port group platform::platformEventsIn;
  end ctasSoftwareLayer;

  system implementation ctasSoftwareLayer.thinClient
        subcomponents
                        v: system View;
                        m: system thinModel;
  end ctasSoftwareLayer.thinClient;
```

ADeS architecture simulation

SPECIFIC TECHNIQUES

- Late one night a police officer came upon a drunk crawling around on the ground under a street light. Police officer asks, “what are you doing?” The drunk replies “I lost my keys over there and I am trying to find them.” (He points to a location over in the middle of the parking lot.) The police officer says, “well then why are you looking here?” The drunk replies “Because there is more light over here.”

Fault model

- Testers need to look where the faults are
- A fault model classifies the types of faults that have been found
- A fault model for software product lines adds the following to the fault models for the language, OS, and development model.

Fault model

- Binding time mismatches
- Instantiation errors
- Inadequate variants
- Missing variation points
- Missing/incorrect dependencies

Operational profiles – specific technique

- An operational profile describes the frequency with which certain operations of the product are invoked.
- Usually broken down by different roles (or actors in a use case driven approach).
- In a product line, one role might have different profiles for different products.
- The profile is used when computing reliability from test data.

Operational profiles

real Human Player

.1

Player selects EXIT from the menu

.8

Player selects a position on the Game Board and clicks

.2

Player selects occupied position

.6

Player selects unoccupied position

.2

Player selects when it is not his turn

.4

Player selects when it is his turn

.1

Change Case: The Player selects the SAVE menu option

Player reads the message signaling the end of a game

.03

Player has won the game

.04

Player has lost the game

.03

Player has tied the game

Composing tests – Specific technique

- Begin with unit tests
  - Where possible tie unit test to a feature
  - Design test to “observe” action rather than encapsulate it
- Compose to form integration tests
  - Compose unit tests
  - Outputs from one are input to another
- Compose to form system tests
  - Continuous integration

Composing tests - 2

- XVCL for static composition
  - Can be done in parallel to building the product code
- Aspects for static/dynamic composition
  - Aspects can be added to code that does not have any *a priori* hooks to link in the test code
  - Can be added using a weaver or can be dynamically inserted in the VM

One Approach: XVCL Generation Path

- Xml-based Variant Configuration Language uses individual frames to compose an asset.

Test plan intro

Feature list intro

Feature /test pairs

Test Specific info

Feature Specific info

Tex

Completed test plan

Completed Feature list

XVCL code

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE x-frame SYSTEM "default">
<x-frame name="main.xvcl">
<!--add variation points that are selected-->
<set-multi var="variationPoints" value="vp1.1,vp1.2"/>
<!--set actual file names for variables-->
<set var="codefile" value="c:\out.txt"/>
<set var="testfile" value="c:\testout.txt"/>
<adapt x-frame="baseSystem.xvcl"></adapt>
<adapt x-frame="registerUseCase.xvcl"></adapt>
<!--add other use cases-->
</x-frame>
```

Product variants

Generation targets

Combinatorial testing – Specific technique

- Pair-wise testing – OATS
- Multi-way test coverage – more than pair-wise, less than all possible
- Minimum test sets -

Orthogonal Array Testing System (OATS)

- Orthogonal Array Testing System (OATS)
- One factor for each variation point
- One level for each variant within a factor
- “All combinations” is usually impossible but pair-wise usually is manageable.
- Constraints identify test cases that are invalid

Factor	Level	Constraint
VP1	VP1.1	
	VP1.2	
	Both	
VP2	None	
	VP2.1	
	VP2.2	
VP3	VP3.1	Requires VP1.2
	VP3.2	Requires VP4
	Both	
VP4	VP4.1	
	VP4.2	
VP5	VP5.1	
	VP5.2	

Example test matrix

- Use standard pre-defined arrays
- This one is larger than needed but that will work
- Each of the factors has values 0,1,2
- Defined to include all pair-wise combinations

1	2	3	4	5	6	7
0	0	0	0	0	0	0
1	1	1	1	1	1	0
2	2	2	2	2	2	0
0	0	1	2	1	2	0
1	1	2	0	2	0	0
2	2	0	1	0	1	0
0	1	0	2	2	1	1
1	2	1	0	0	2	1
2	0	2	1	1	0	1
0	2	2	0	1	1	1
1	0	0	1	2	2	1
2	1	1	2	0	0	1
0	1	2	1	0	2	2
1	2	0	2	1	0	2
2	0	1	0	2	1	2
0	2	1	1	2	0	2
1	0	2	2	0	1	2
[illegible]	[illegible]	[illegible]	0	1	2	2

factor

level

Mapping

1	2	3	4	5	6	7
0	0	0	0	0	0	0
1	1	1	1	1	1	0
2	2	2	2	2	2	0
0	0	1	2	1	2	0
1	1	2	0	2	0	0
2	2	0	1	0	1	0
0	1	0	2	2	1	1
1	2	1	0	0	2	1
2	0	2	1	1	0	1
0	2	2	0	1	1	1
1	0	0	1	2	2	1
2	1	1	2	0	0	1
0	1	2	1	0	2	2
1	2	0	2	1	0	2
2	0	1	0	2	1	2
0	2	1	1	2	0	2
1	0	2	2	0	1	2
2	1	0	0	1	2	2

Factor	Level	Constraint
VP1	VP1.1	
	VP1.2	
	Both	
VP2	None	
	VP2.1	
	VP2.2	
VP3	VP3.1	Requires VP1.2
	VP3.2	Requires VP4
	Both	
VP4	VP4.1	
	VP4.2	
VP5	VP5.1	
	VP5.2	

Mapped array

VP1	VP2	VP3	VP4	VP5	
~~VP1.1~~	~~None~~	~~VP3.1~~	~~VP4.1~~	~~VP5.1~~	~~c~~
VP1.2	VP2.1	VP3.2	VP4.2	VP5.2	
Both	VP2.2	Both	2	2	x
VP1.1	None	VP3.2	2	VP5.2	x
VP1.2	VP2.1	Both	VP4.1	2	x
Both	VP2.2	VP3.1	VP4.2	VP5.1	
~~VP1.1~~	~~VP2.1~~	~~VP3.1~~	~~2~~	~~2~~	~~x,c~~
VP1.2	VP2.2	VP3.2	VP4.1	VP5.1	
Both	None	Both	VP4.2	VP5.2	
VP1.1	VP2.2	Both	VP4.1	VP5.2	
VP1.2	None	VP3.1	VP4.2	2	x
Both	VP2.1	VP3.2	2	VP5.1	x
VP1.1	VP2.1	Both	VP4.2	VP5.1	
VP1.2	VP2.2	VP3.1	2	VP5.2	x
Both	None	VP3.2	VP4.1	2	x
VP1.1	VP2.2	VP3.2	VP4.2	2	x
VP1.2	None	Both	2	VP5.1	x
Both	VP2.1	VP3.1	VP4.1	VP5.2	

- Every row is a system under test.
- 17 test products vs 108 possible combinations
- Columns 4 and 5 have more levels than are needed. Columns 6 and 7 are not needed at all.
- Where a “2” is in the column, this indicates the tester could repeat a value (one of the variants).

Traceability – Specific technique

- Start with the scenarios in the use cases
- Select scenarios to test the architecture (ATAM)
- Use these scenarios to test portions of the detailed design
- Use these scenarios as the start for system test cases

Traceability/reuse

An ATAM scenario is used for several levels of reviews and then used as the basis for system test cases

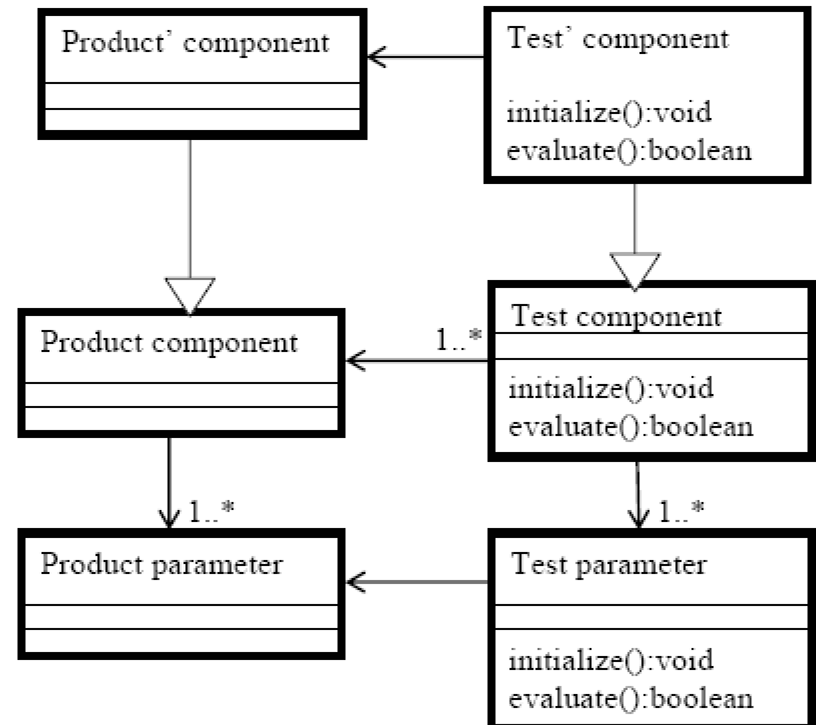
Requirement → Use case ← Requirement

Use case → Scenario, Scenario, Scenario

Scenario → Test case, Test case, Test case

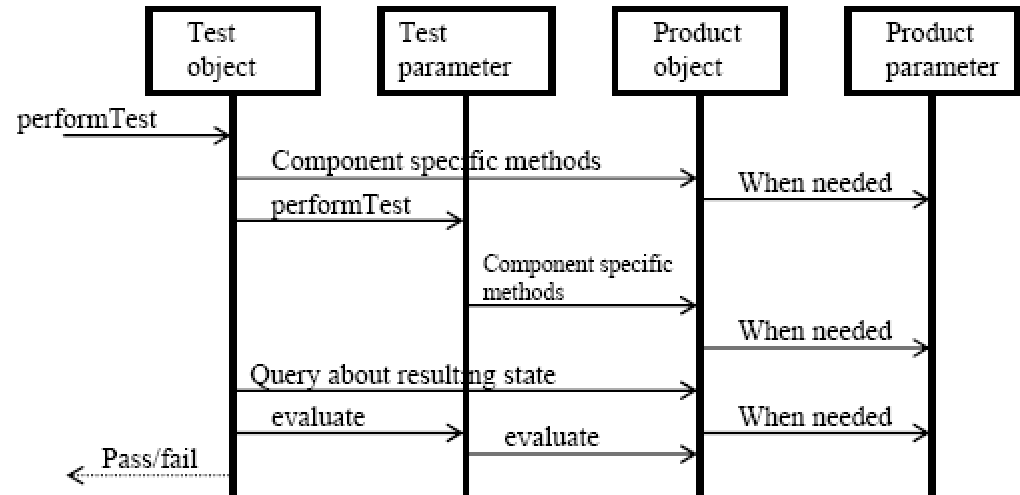
Test design - 1

- Parallel Architecture for Component Testing (PACT)
- Takes advantage of the inheritance relationship and the polymorphic handling of these pieces
- Product’ inherits from Product
- Test’ inherits from Test
- Product has dependency on Product parameter
- Test has dependency on Test parameter



Test design - 2

- Action sequence for PACT

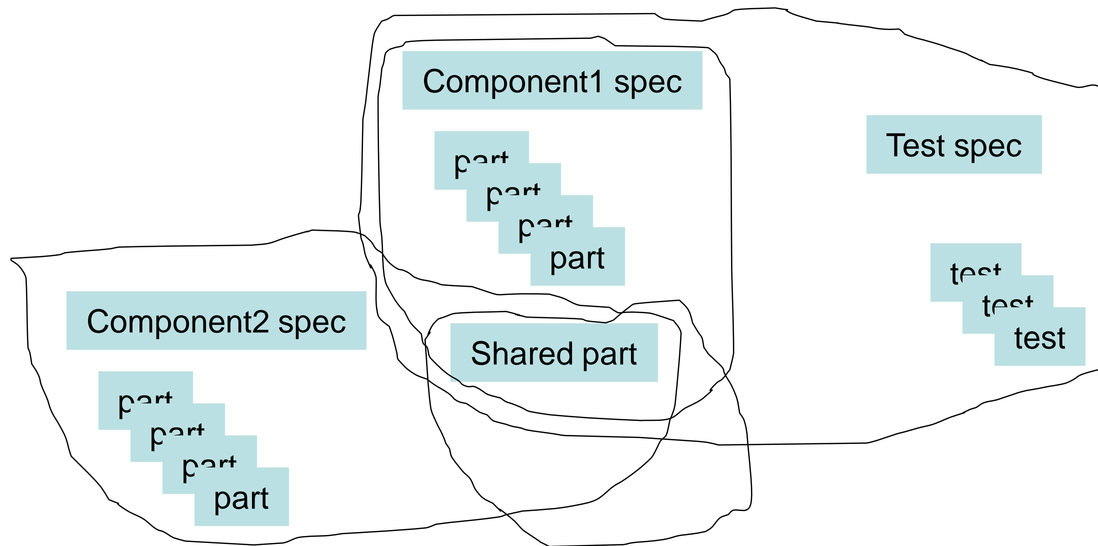


Configurations – specific technique

- Product configurations
  - Each component and core asset has a configuration
  - Each product has a configuration that identifies the configurations of assets needed to construct the product
- Test configurations
  - Each asset and product configuration should have an associated test configuration.
  - Test configuration includes the parameters needed to configure the test infrastructure, the test data, and the test procedure.

Configuration

- Component1 spec and Component2 spec are configurations for products and their contents.
- Shared part is a configuration for one piece which is shared by two or more products.
- Test spec is a configuration for a testsuite for Component1.



Configuration management for test assets

Product Line Architecture

Version 1

Version 2

Version 3

<<general structure>>

<<general structure>>

Product A

Product B

<<parameters>>

<<parameters>>

Version A.1

Version A.2

Version B.1

Version B.2

Version B.3

Test Cases

<<derived from>>

Test Cases

Configuration management for test assets

Product Line Architecture

Test Plan

Version 1

Version 2

Version 3

Test Cases

<<general structure>>

Product A

Test Plan

<<parameters>>

Version A.1

Version A.2

Test Cases

revision

Configuration management for test assets

Configuration in BuckMinster - 1

- Do a Cspec (component specification) for each item – program components, test components
  - A Cspec defines the membership of a component configuration
- For example
  - A Cspec for a product component
  - A Cspec for a test component that references a specific version of the product component (it does that by referencing the Cspec version for the component)
  - A Cspec is used when a Cquery is executed

Configuration in BuckMinster - 2

- Do a Cquery (component query) to assemble a product or a test configuration that includes an instantiation of the product
  - A Cquery describes WHERE each element that constitutes the Cspec is located and HOW it is retrieved
- For example,
  - A Cquery for the test component that is configured to instantiate the product component
  - Executing a Cquery might instantiate a project that tests the component

Core asset configuration

- Core asset configuration contains
  - Basic functional modules
  - All variant modules
  - Basic test code
  - All variant test modules
  - Any data sets needed

Code test coverage levels

- Unit
  - Based on specification and structure
- Interaction of completed units
  - Based on patterns
- Integration of completed units
  - Based on architecture
- System
  - Based on requirements
- And beyond

Simple tests in JUnit

AADL - velocityExample/SimpleTest.java - Eclipse SDK

```java
import junit.framework.*;

/**
 * Some simple tests.
 *
 */
public class SimpleTest extends TestCase {
    Velocity v;

    protected void setUp() {
            }
    public static Test suite() {


        return new TestSuite(SimpleTest.class);
    }
    public void testVelocity1() {
        v = new Velocity();
        assertTrue(v.getDirection() == 0);
    }
        public static void main (String[] args) {
        junit.textui.TestRunner.run(suite());
    }

    public void testVelocity2(){
        v = new Velocity(5,90);
        assertTrue(v.getDirection() == 90);
    }
}
```

Finished after 0.031 seconds

Runs: 2/2 Errors: 0 Failures: 0

SimpleTest [Runner: JUnit 3]

JUnit tests

- I take the __Unit (JUnit, XUnit…) approach
- Test cases are encoded in classes and executed by a test harness
- The test case representations are constructed using an architecture parallel to that of the unit
- Variations are encoded in the test cases using a mechanism similar to the one used in the asset
- Tests are based on functional specifications including state machines to certify that objects do everything they are supposed to do.
- Tests are selected to provide “adequate” coverage of the structure to certify that the objects don’t do anything they are not supposed to.

System tests

- Test sample applications
  - Combinations of choices at variation points
  - Full scale integration tests
  - Limited to what can be built at the moment
  - Involve product builders early
- Test specific application
  - Tests a specific product prior to deployment
  - Rerun some of the selected products' test cases
  - Feedback results to core asset builders

System tests

- Tie test cases to specific scenarios
- Each scenario focuses on a single use case or a single feature
- Each “extends” relationship in a use case leads to a specialization relationship between test case definitions
- Each “includes” relationship provides structural information and may lead to multiple test cases

Abbot test run

Java - Eclipse SDK

File Edit Navigate Search Project Run Window Help

Pac Hier JUni

src.AbbotTestCaseBricklesGame - testTes

Runs: 2/2 Errors: 0 Failures: 0

src.AbbotTestCaseBricklesGame [Ru
testTestMethod1_PlayGame
testTestMethod2_HitBrick

Failure Trace

Problems Javadoc Declaration Console Properties

AbbotTestCaseBricklesGame [JUnit] C:\Program Files\Java\jre1.6.0_01\bin\javaw.exe (May 25, 20

EVALUATION

Evaluating test activities

- Effectiveness
  - Defect live range
- Efficiency
  - Defects/hour
  - Changes to test assets/product

Testing effectiveness - 1

- **Testing effectiveness** is defined as the percentage of total defects found at each step in the testing process divided by the total defects
- Our goal is to find a larger percentage in the early stages and smaller percentages in later stages
- **Effectiveness** of the system testing process is defined as:
  - Faults found during system testing divided by total number of faults found by all testing activities (including the testing done by customers that we consider production testing)

Testing effectiveness - 2

- The effectiveness ratios should be calculated after the testing of each increment
- The progression of the values from one increment to the next may show trends that indicate the effectiveness of the testing process
- Expect more defects to be found in the early testing phases and fewer in the later testing phases as the increment matures
- Do the number of errors always approach zero as the component matures?

Testing efficiency

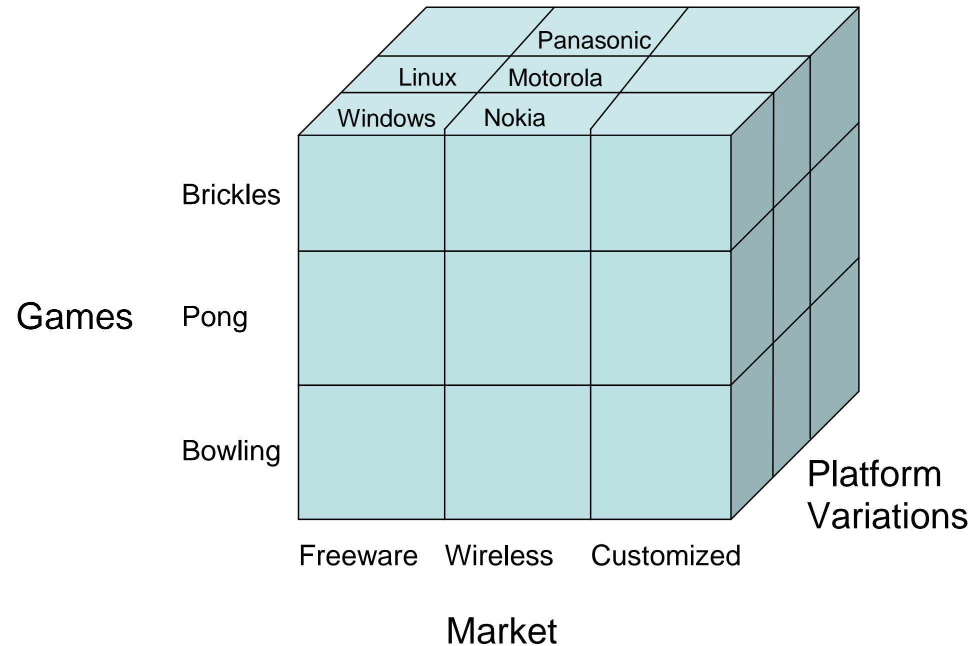
- Efficiency is productive use of resources
- Defects/hour - The more defects found per hour of effort the more efficient the test process
- As the efficiency drops, look to the risk/reward measures to see if it is worth continuing
- Changes to test assets/ product - The number of changes that need to be made because of different features indicates how well the test suite was built to begin with

CASE STUDY – ARCADE GAME MAKER

www.sei.cmu.edu/productlines/ppl

AGM

- Arcade Game Maker (AGM) product line
- Wants to use a product line approach to produce several game products for numerous platforms.
- Business plan calls for entering new markets once the product line is in operation



Test strategy

- For the product line effort, AGM decides the test strategy will be to invest in:
  - Explicit design of assets for testability
  - CM support to associate test cases with product assets
  - Automated execution of all test cases
  - Automated test generation for the third increment of the product when products are also automatically generated

Test process & plan

- Process
  - AGM is using a modified version of test-driven development in which development of software core assets is preceded by a specification of tests for the asset.
  - AGM has revised all of their development processes to have validation criteria. The production plan includes specific validation activities in each phase of product building.
- Plan
  - The product line test plan describes how product teams select test assets
  - The plan assigns responsibilities to personnel for specification, creation, maintenance, and execution of tests.

Test approaches

- The arcade games have several characteristics that influence testing.
  - The games are state-based which leads to state traversal test cases
  - The variant features are statically bound, so test cases can be defined using parameterization and inheritance
- AGM uses JUnit for unit tests and a set of in-house test harnesses that include modified core assets to support the testing of other assets.

Testability

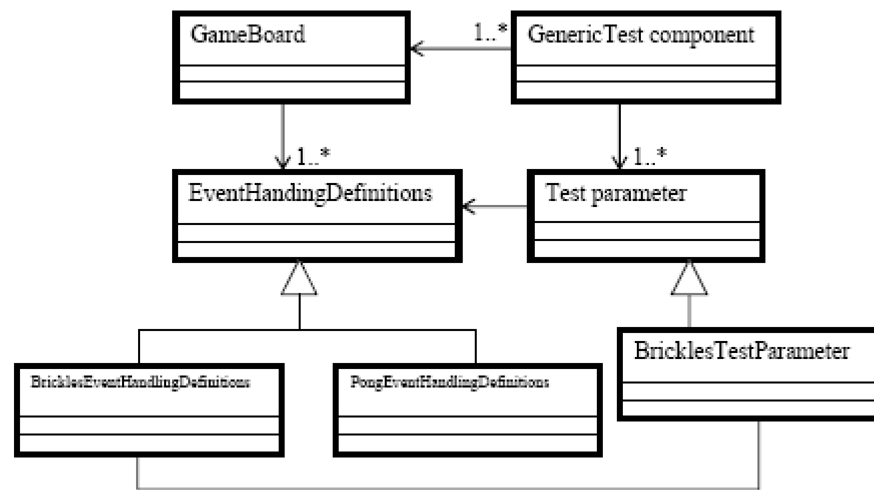
- The Java development environment supports the automatic generation of get/set methods for classes.
- The Java language provides the "implements" capability to allow a class to implement multiple interfaces. This is coordinated with the unit test tools to make the internals of the Java class visible to the testing tools without introducing dependencies with other modules in the product.

Test design

- AGM has followed the product line testing research and knows that structuring their test classes using the same architecture as their product architecture is the most efficient approach.
- The structures needed include
  - Test infrastructures
  - Test cases
- The Eclipse Test and Performance Tools Project (TPTP) provides a wide range of tools for creating and executing tests.

Test design - 2

- The dominant design structure in the game design is inheritance hierarchies.
- The corresponding test case design can be handled polymorphically with a test infrastructure that is specialized to the exact test case types being used.



Test tools

- The same tools and techniques used for product design are used for test design. Eclipse is the primary environment we use. Each person can configure their environment as they wish, choosing from a very large number of openly available plug-in tools.
- We used Junit and Abbott from TPTP project to instrument unit and product tests.

Test results & reports

- Test results and reports are used to:
  - direct the efforts of the core asset team, and
  - the architecture team.
- AGM also produces a test report that assures its wireless platform customers that AGM’s products have no external impacts on availability of the environment because of defects such as failure to release memory.

Test assumptions

- AGM games do not access OS directly so no need for “platform” testing.
- AGM games operate within a standard VM so no need for interaction testing with other products.
- There maybe interactions among features so feature testing is important.

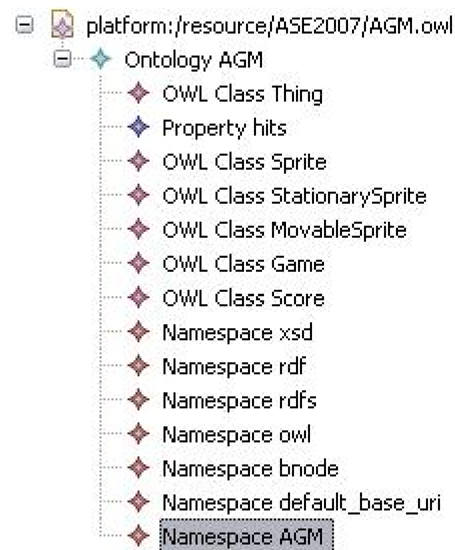
Features in AGM

- One example of a feature is the “scoreboard”
- It is an optional feature
- It introduces a new data path and a new control flow path
- Specification-based test cases use existing tests
- Structural-based test cases cover the new connections

The gaming domain

- Is relatively stable
- Has a large number of diverse offerings with a high degree of commonality
- Is a candidate for a domain specific language
- We built ours by starting with an ontology using the OWL

Partial ontology



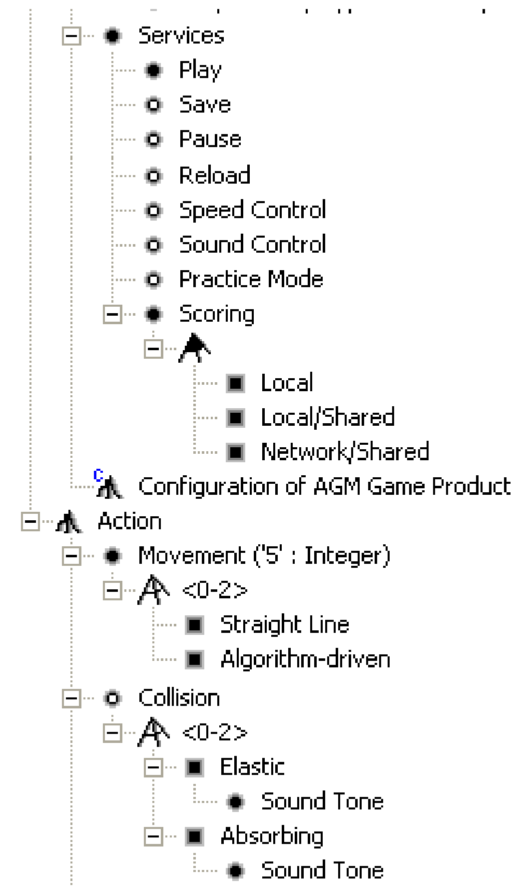
Use case

```
<usecase name="HitBrick">
  <stimulus>
    <userActions>
      <actionPerformed action="HitBrick"/>
      <attributeEffected attribute="BrickCount" value=""/>
    </userActions>
  </stimulus>
  <response>
    <reactions reactingObject="BrickCount" valueExpected=""/>
  </response>
</usecase>
```

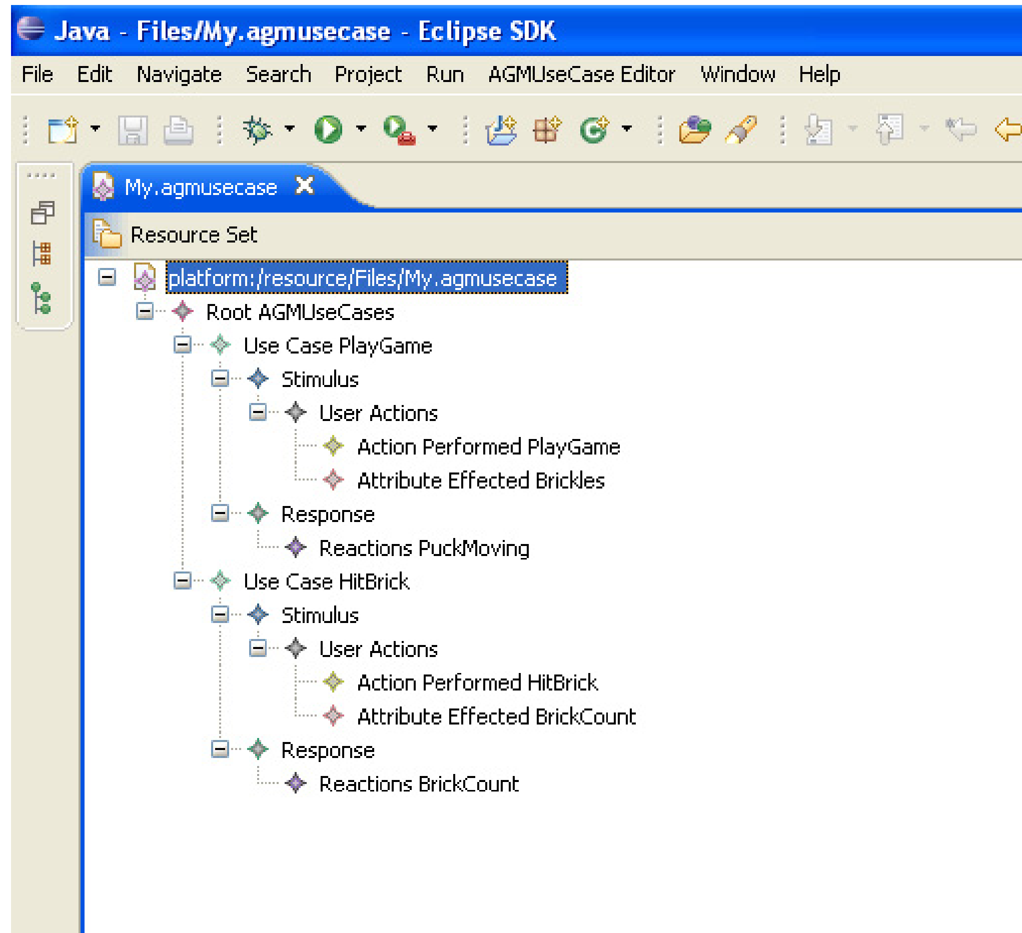
```
<usecase name="HitBrick">
  <stimulus>
    <userActions>
      <actionPerformed action="HitBrick"/>
      <attributeEffected attribute="BrickCount" value="11"/>
    </userActions>
  </stimulus>
  <response>
    <reactions reactingObject="BrickCount" valueExpected="11"/>
  </response>
</usecase>
<usecase name="HitBrick2">
  <stimulus>
    <userActions>
      <actionPerformed action="HitBrick"/>
      <attributeEffected attribute="BrickCount" value="10"/>
    </userActions>
  </stimulus>
  <response>
    <reactions reactingObject="BrickCount" valueExpected="10"/>
  </response>
</usecase>
```

AGM's test suite generation

- AGM's independent system test group associates test cases with individual features.
- Products are specified by composing feature sets
- Test suites are automatically composed in parallel to the feature set.
- Example: The Services features are modular.
- A test suite is associated with each of the services.
- Test suite composition is triggered by the feature set.



Use case editor



Test case editor

AGMUseCase Application

File Edit AGMUseCase Editor Search Run Window Help

file:/C:/Documents%20and%20Settings/SuperUser/Desktop/My.agmusecase

Resource Set

- file:/C:/Documents%20and%20Settings/SuperUser/Desktop/My.agmusecase
  - Root AGMTestCases
    - Use Case 1_PlayGame
      - Stimulus
        - User Actions
          - Action Performed PlayGame
          - Attribute Effected Brickles
      - Response
        - Reactions PuckMoving
    - Use Case 2_HitBrick
      - Stimulus
        - User Actions
          - Action Performed HitBrick
          - Attribute Effected BrickCount
      - Response
        - Reactions BrickCount
    - Use Case 2_HitBrick2
      - Stimulus
        - User Actions
          - Action Performed HitBrick
          - Attribute Effected BrickCount
      - Response
        - Reactions BrickCount

Selection | Parent | List | Tree | Table | Tree with Columns

Outline

file:/C:/Documents%20and%20Settings/SuperUser/Desktop/My.agr

Properties

Property	Value

Selected Object: file:/C:/Documents%20and%20Settings/SuperUser/Desktop/My.agmusecase

System test plan

- Use an infrastructure that can operate on a wide range of platforms
- Modularize system tests so that they can be composed as features are selected for the product

System test using Abbot

Conclusions

- Most of the differences between testing “traditional” software and software produced in a software product line is a matter of process rather than a technical difference
- Keeping the test software synchronized with the production software reduces effort
- Testing early and often, reduces the impact of variability
- Using a well designed configuration management scheme reduces the impact of reuse
- Automation is required to profit from reuse
- Organizational management, technical management, and software engineering must all work together for an effective test process.

Take aways

- Use the product structure to guide design of the test structure
- Testing the small parts reduces the effort of testing the big parts
- Test early, test often
- Plan an integrated approach to product development and test – then automate it

References

- Proceedings of SPLiT2004, 2005, 2006, 2007
- Composing Unit Tests. Markus G¨alli, Orla Greevy, and Oscar Nierstrasz, SPLiT 2005.
- J.D. McGregor: Parallel Architecture for Component Testing,
- Tomoji Kishi and Natsuko Noda, Design Testing for Product Line Development based on Test Scenarios, SPLiT 2004.
- Clements, Paul; Kazman, Rick; Klein, Mark. Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2002.