Component-Based Software Development & Testing

Hans-Gerhard Gross
Embedded Software Laboratory
Delft University of Technology

Objectives and Outline

PART I

- What are component-based systems?
- How are they developed?

PART II

- Why are component-based systems prone to error?
- How should they be tested?

Component-Based Software Development

And Software Components

What is a Software Component?

- Reusable 3rd party unit of composition ☑
- With a well defined abstraction ☑
- With well defined provided and required interfaces ☑
  - Structure, Function ☑
  - Behavior, Quality
- With well defined and openly accessible internal behavior ☒

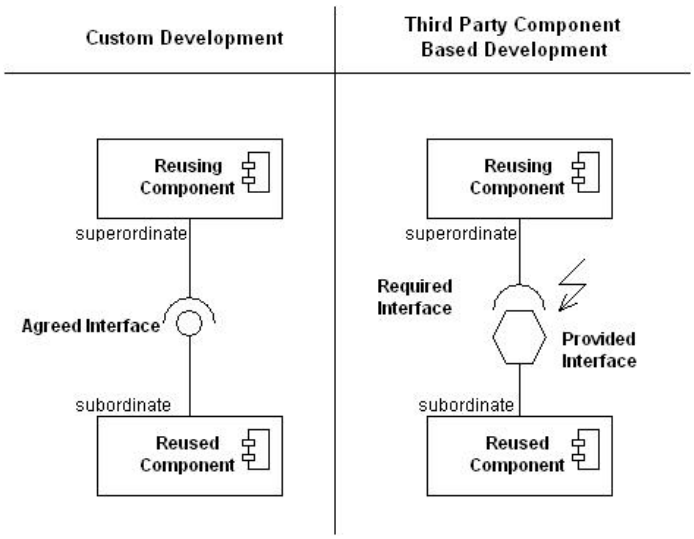
What makes them difficult to deal with?

The fact that

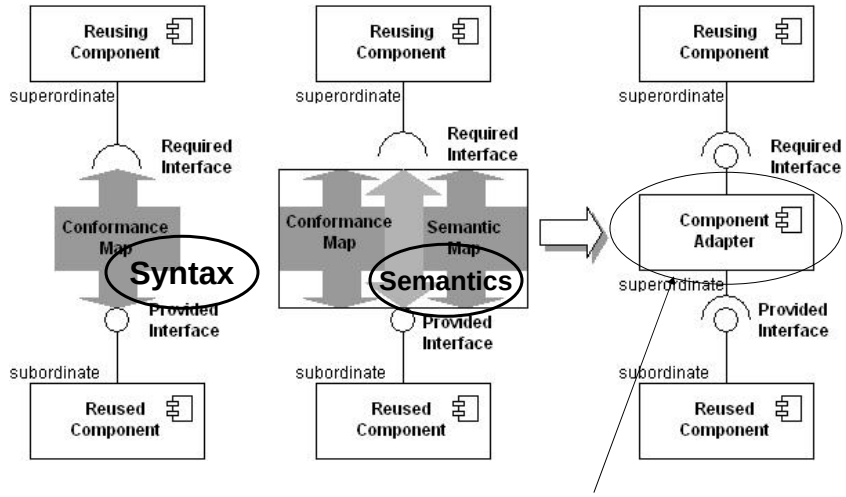
- Components are <u>independent</u> from the system context (only domain context)
- Components are replaceable and <u>deployable</u> <u>in many system contexts</u>
- Components are coming from various sources
  - In-group
  - In-house
  - Third party

Adaptation and integration are the key problems

Custom vs. Component-Based Development



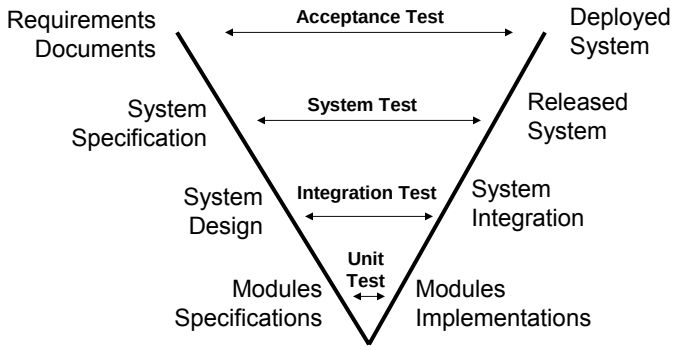
Component Integration



Additional development effort: Glue code
Plus Testing of the glue code

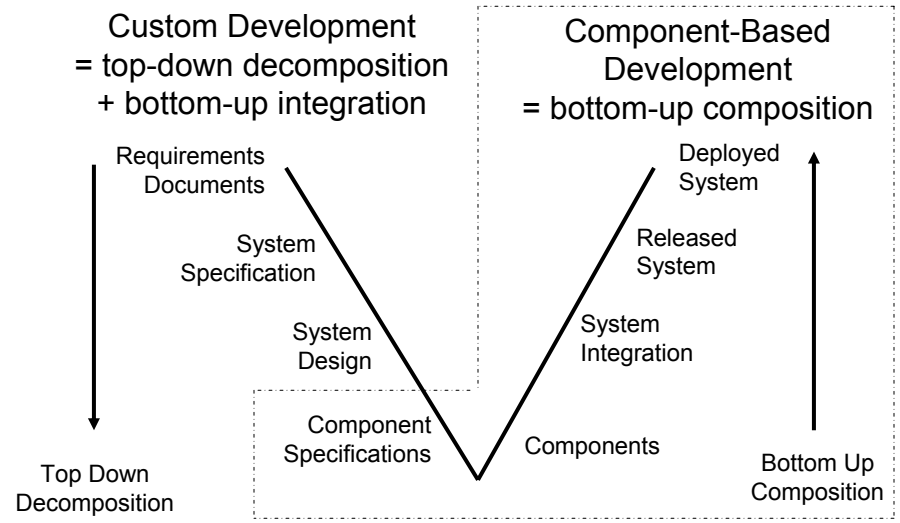
Custom Development Method

- Custom development

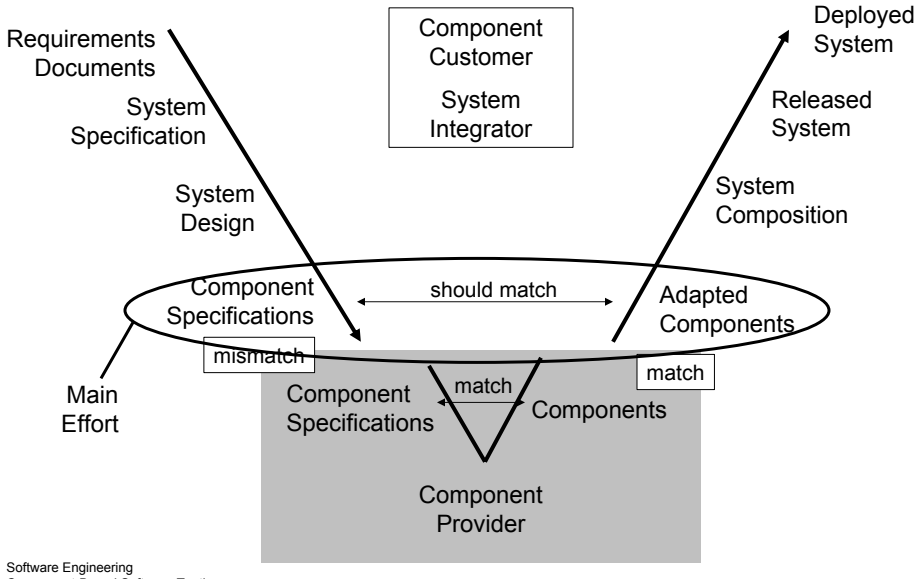


V-Model

Custom vs. Component-Based Development



Component-Based Development Process



Summary

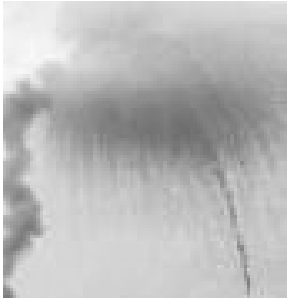
- You can now
  - give a definition for a software component
  - contrast component-based development and development from scratch
  - explain why component-based development is an integration effort
  - explain how components are integrated
  - explain how and where components fit into the V-model

Component-Based Software Testing

And Built-In Testing

Example: Ariane 5 Failure

The maiden flight of the Ariane 5 launcher, on 4.6.1996, ended in failure and a loss of 600.000.000 EUR, due to a software error.



References: http://www.niwotridge.com/Resources/DomainLinks/Ariane5Failure.htm

Example: Ariane 5 Failure

- The launcher started to disintegrate 40 sec. after lift-off in increased aerodynamic load
- This was caused through wrong adjustments of the on-board computer based on erroneous data coming from the inertial reference system
- The inertial reference system did not send correct attitude data because of an internal software exception (conversion overflow exception)

Example: Ariane 5 Failure

- The exception happened in a component of the inertial reference system that was taken from Ariane 4 and reused within Ariane 5.
- This component would only perform alignment operations before lift-off and continue to do that for another 40 sec.
- This was not a requirement of the Ariane 5.
- However, it was running and calculating for another 40 sec and eventually produced an conversion overflow.

Example: Lessons Learned

- The reused component coming from Ariane 4 was obviously alright and well tested
- ESA had used it for more than 10 years without trouble
- So they could claim rightly it was highly dependable and suitable to be reused in the new context of Ariane 5
- Could they?

Example: Ariane 5 Consequences

- Find components that match the requirements of the system exactly (find an Ariane 5 inertial reference component)?

→ no reuse, development from scratch, which costs another 2.5 millions

- Test the components better/harder?

→ they had tested it extremely well

Example: Ariane 5 Consequences

- Test the component differently?

→ Test the component according to the usage profile of the integrating (physical) context

→ Test the Ariane 4 component according to the requirements of the Ariane 5 context

→ Develop test cases that represent the usage profile of the integrating (physical) context

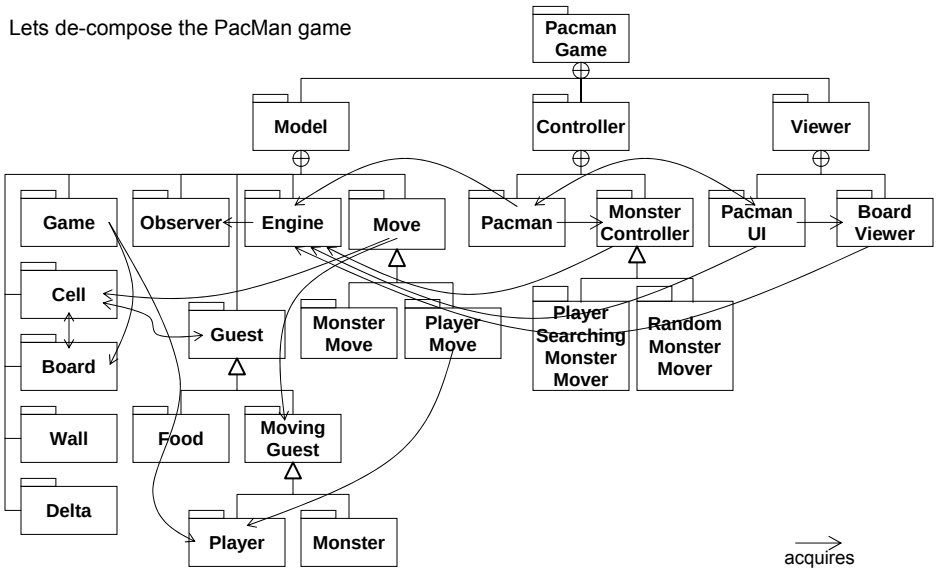
Example: Conclusions

- Components are designed and developed according to the usage profile determined by the component provider (Ariane 4 Team)
- Components are integrated and used according to the usage profile determined by the integrating system of the component customer (Ariane 5 Team)
- Both are usually different

Now, lets have a look at real component reuse and integration, before we have a look at testing it.

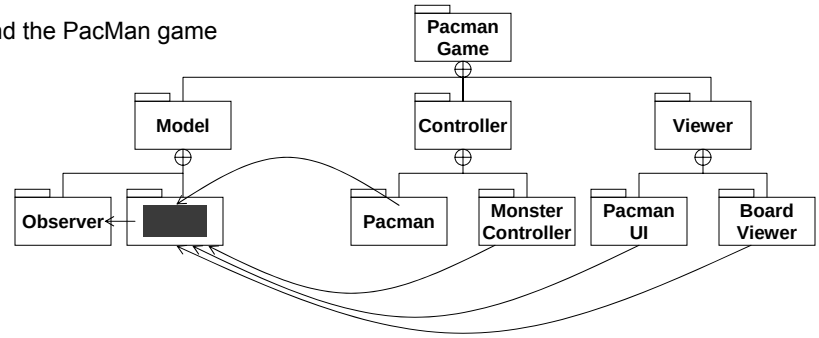
System Decomposition

Lets de-compose the PacMan game

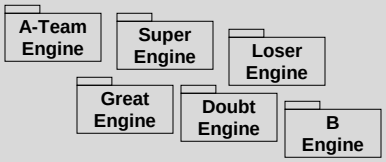


Component Reuse

Lets amend the PacMan game

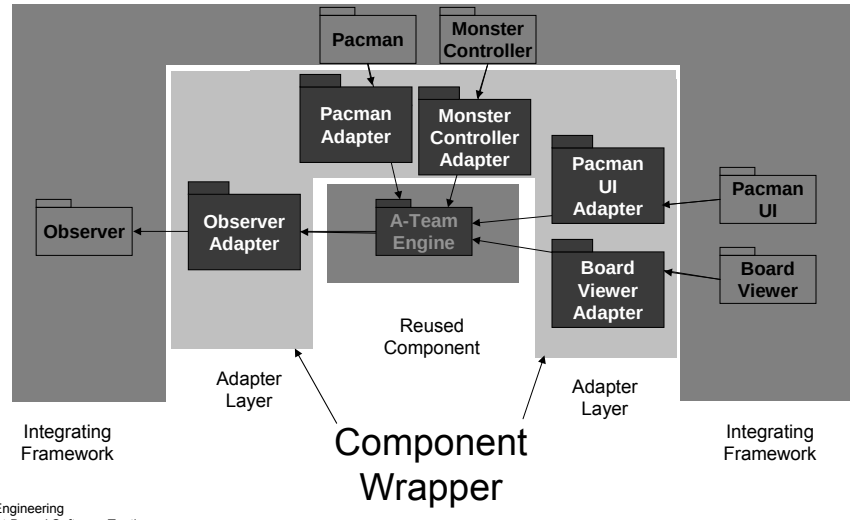


Engine Component Market Place



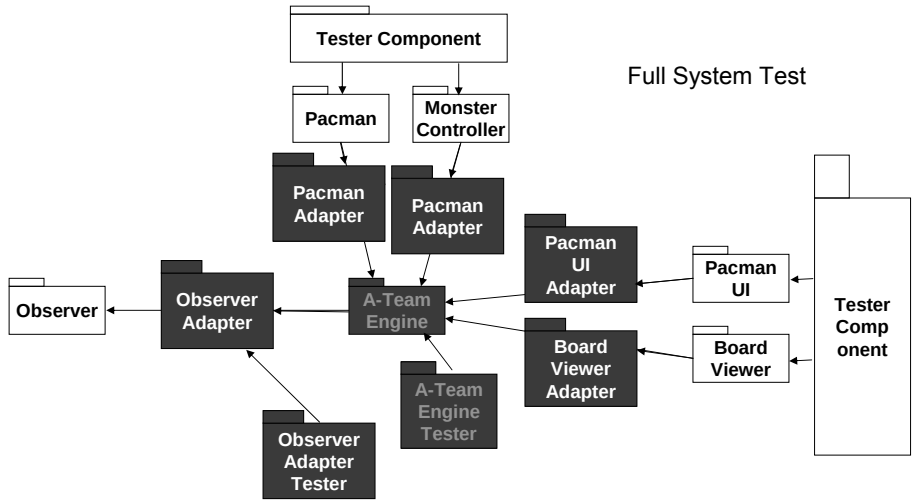
Component Integration

Lets integrate the new A-Team Engine Component



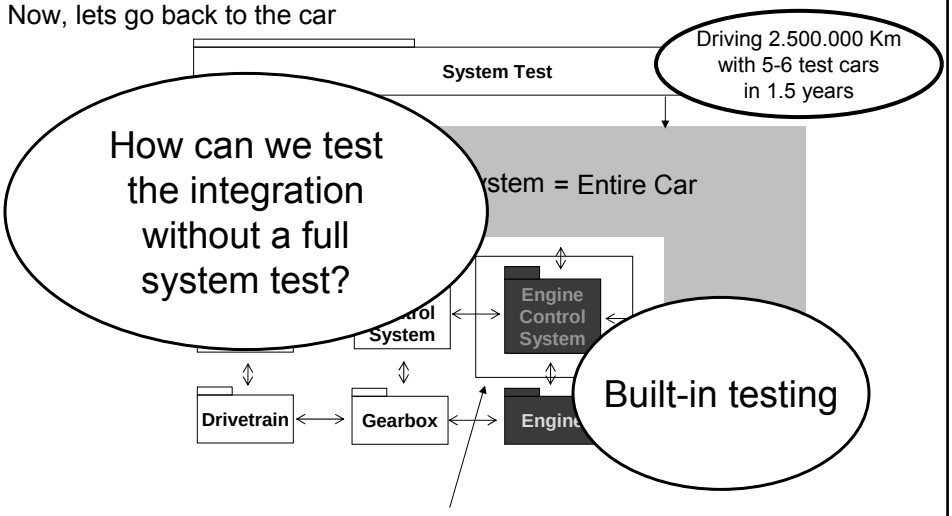
Integration Test (Small)

Now lets validate the integration



Integration Test (Large)

Now, lets go back to the car



Where should we apply tests ideally?

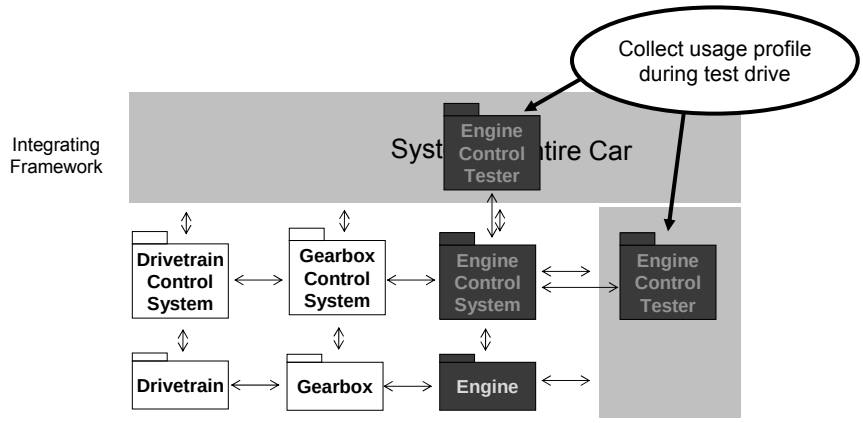
Built-in Testing

What do you think is built-in testing?

- Everything built into a system that facilitates testing
- Built-in testability code
  - Code that increases observability
  - Testing interfaces
  - Assertions
- Built-in testing code
  - Test cases, test suites, test components

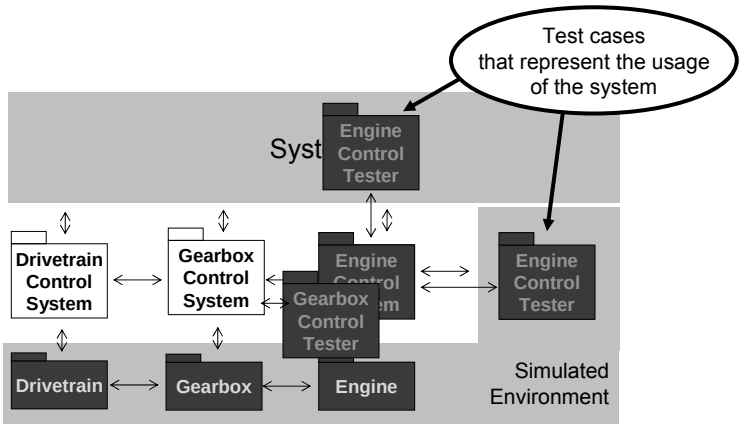
Built-in Testing Code

(1) Collect test cases according to the usage profile



Built-in Testing Code

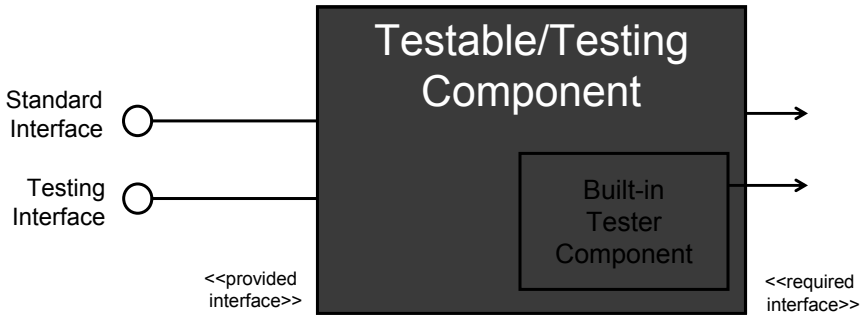
(2) perform the tests of the context in situ with the new component



Built-In Testability Code

How do we make a reused component more testable?

– Testing interface



Built-in Testability Code

What services should a testing interface provide?

Code that increases observability

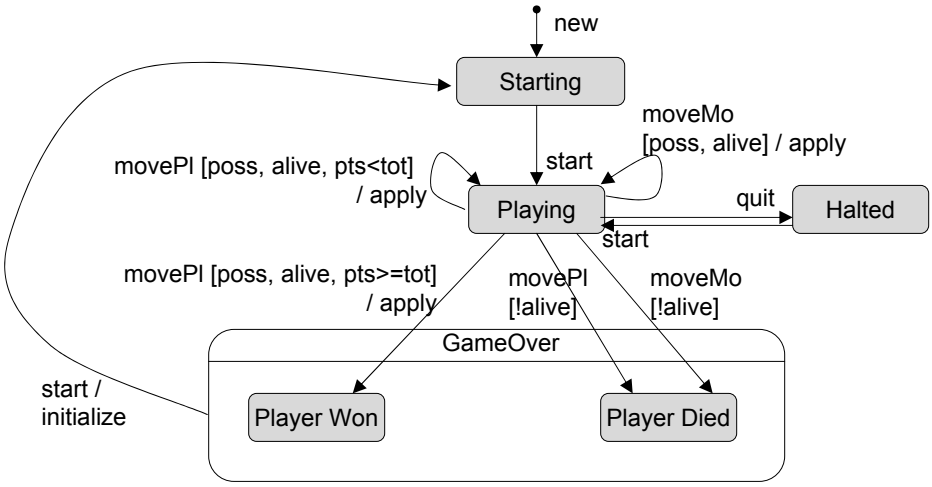
- Assertions
- Externally visible state information

Let's go back to the packman game

Testable PacMan.Engine
- Observer TheViewer; - Game TheGame; - …
<<standard interface>> + … <<testing interface>> + inStartingState ( ) : boolean; + inPlayingState ( ) : boolean; + inWonState ( ) : boolean; + inDiedState ( ) : boolean; + inHaltedState ( ) : boolean; + inGameOverState ( ) : boolean;

Built-in Testability Code

PacMan Engine State Model



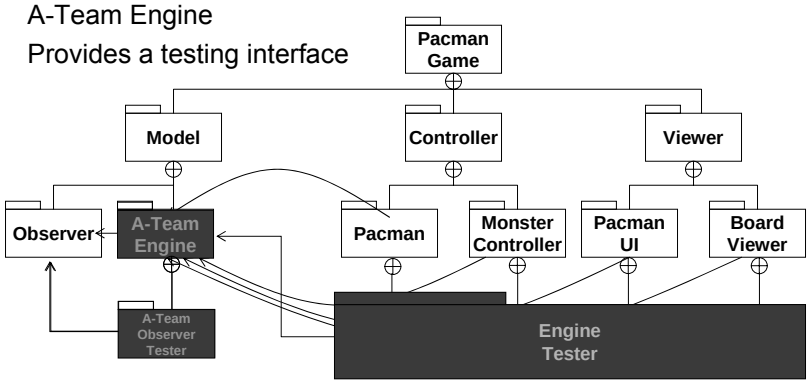
Exercise: State-Transition Testing

Test #	Check Initial State	Precondition	Transition	Postcondition	Check Final State
1	--	--	new	--	inStartingState
2	inStartingState	--	start	--	inPlayingState
3	inPlayingState	[poss, alive, pts<tot]	movePl	apply	inPlayingState
4	inPlayingState	[poss, alive]	moveMo	apply	inPlayingState
5	inPlayingState	--	quit	--	inHaltedState
6	inHaltedState	--	start	--	inPlayingState
7	inPlayingState	[!alive]	moveMo	--	inGameOverState & inPlayerDiedSta
8	inPlayingState	[!alive]	movePl	--	inGameOverState & inPlayerDiedSta
9	inPlayingState	[poss, alive, pts>=tot]	movePl	apply	inPlayerWonState
10	inGameOverS	--	start	initialize	inStartingState

Built-In Test Overview

- Built-in test components in integration testing

A-Team Engine
Provides a testing interface



Engine tester
Comprises state transition tests

Built-In Test Overview

- Every component owns one tester component for checking each of its peers
- Every component provides a testing interface
- The tests represent the component's expectations towards its peers
- Whenever a new system is assembled, the components are integrated (adapted) and their in-built tests are executed

Overall Conclusions

- A component should be tested according to the requirements of the integrating context
- A component integration should always be tested as if its parts have never been tested before
- The usage profile determines what should be tested
- A full system test is often not feasible, nor is it desirable

Summary

- You can now
  - Explain how testing component-based systems is different from testing systems built from scratch
  - Explain what a component wrapper is
  - Explain what a testing interface does
  - Demonstrate how testing should be carried out in a component integration step

Let's do a minitest …

Further Reading

- Ariane 5 Failure Resources, http://www.niwotridge.com/Resources/DomainLinks/Ariane5Failure.htm
- C. Atkinson, et al. Component-Based Product Line Engineering with UML. Addison-Wesley, 2002.
- G. Booch; Software Components with Ada; Benjamin-Cummings, 1987.
- Large-Scale, Component-Based Development; Prentice Hall, 2000.
- G.T. Heineman, W.T. Councill (Eds); Component-Based Software Engineering; Addison-Wesley, 2001.
- C. Szyperski; Component Software – Beyond Object-Oriented Programming; Addison-Wesley, 2nd Edition, 2002.