Softwaretechnik / Software-Engineering

Lecture 15: Architecture and Design Patterns

2015-07-04

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

- Introduction and Vocabulary
- Principles of Design
  - (i) modularity
  - (ii) separation of concerns
  - (iii) information hiding and data encapsulation
  - (iv) abstract data types, object orientation
- Software Modelling
  - (i) views and viewpoints, the 4+1 view
  - (ii) model-driven/-based software engineering
  - (iii) Unified Modelling Language (UML)
  - (iv) modelling structure
    - a) (simplified) class diagrams
    - b) (simplified) object diagrams
    - c) (simplified) object constraint logic (OCL)
  - (v) modelling behaviour
    - a) communicating finite automata
    - b) Uppaal query language
    - c) implementing CFA
    - d) an outlook on UML State Machines
- Design Patterns
- Testing: Introduction
Content (Part I)

- Architecture Patterns
  - Layered Architectures,
  - Pipe-Filter,
  - Model-View-Controller.

- Design Patterns
  - Strategy,
  - Observer, State, Mediator,
  - Singleton, Memento,
  - Inversion of control.

- Libraries and Frameworks

- Quality Criteria on Architectures
  - Development Approaches,
  - Software Entropy.

Architecture Patterns
Introduction

- Over decades of software engineering, many clever, proved and tested designs of solutions for particular problems emerged.

- **Question**: can we generalise, document and re-use these designs?

- **Goals**:
  - “don’t re-invent the wheel”.
  - benefit from “clever”, from “proven and tested”, and from “solution”.

**architectural pattern** – An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. (Buschmann et al. 1996)

Introduction Cont’d

- **Using** an architectural pattern
  - implies certain characteristics or properties of the software (construction, extendibility, communication, dependencies, etc.).
  - determines structures on a high level of the architecture, thus is typically a central and fundamental design decision.

- The information that (where, how, …) a well-known architecture / design pattern is used in a given software can
  - make comprehension and maintenance significantly easier,
  - avoid errors.
Example: Layered Architectures

(Züllighoven, 2005):

A layer whose components only interact with components of their direct neighbour layers is called protocol-based layer. A protocol-based layer hides all layers beneath it and defines a protocol which is (only) used by the layers directly above.

Example: The ISO/OSI reference model.

![Diagram of ISO/OSI reference model](image-url)
Example: Layered Architectures Cont’d

- **Object-oriented layer**: interacts with layers directly (and possibly further) above and below.
- **Rules**: the components of a layer may use
  - only components of the protocol-based layer directly beneath, or
  - all components of layers further beneath.

---

**Diagram:**

- GNOME etc. Applications
- GTK+
- Pango, GDK, ATK, GIO
- Cairo, GLib
**Example: Three-Tier Architecture**

- **presentation layer** (or **tier**):
  user interface; presents information obtained from the logic layer to the user, controls interaction with the user, i.e. requests actions at the logic layer according to user inputs.

- **logic layer**:
  core system functionality; layer is designed without information about the presentation layer, may only read/write data according to data layer interface.

- **data layer**:
  persistent data storage; hides information about how data is organised, read, and written, offers particular chunks of information in a form useful for the logic layer.

- **Examples**: Web-shop, business software (enterprise resource planning), etc.

**Layered Architectures: Discussion**

- **Advantages**:
  - **protocol-based**: only neighbouring layers are coupled, i.e. components of these layers interact,
  - coupling is low, data usually encapsulated,
  - changes have local effect (only neighbouring layers affected),
  - **protocol-based**: distributed implementation often easy.

- **Disadvantages**:
  - performance (as usual) – nowadays often not a problem.
Example: Pipe-Filter

Example: Compiler

Example: UNIX Pipes

```bash
ls -l | grep Sarch.tex | awk '{ print $5 }'
```

- Disadvantages:
  - if the filters use a common data exchange format, all filters may need changes
    if the format is changed, or need to employ (costly) conversions.
  - filters do not use global data, in particular not to handle error conditions.
Model-View-Controller

Example: Model-View-Controller
Example: Model-View-Controller

- **Advantages:**
  - one model can serve multiple view/controller pairs;
  - view/controller pairs can be added and removed at runtime;
  - model visualisation always up-to-date in all views;
  - distributed implementation (more or less) easily.

- **Disadvantages:**
  - if the view needs a lot of data, updating the view can be inefficient.

**Design Patterns**
Design Patterns

- In a sense the same as architectural patterns, but on a lower scale.
- Often traced back to (Alexander et al., 1977; Alexander, 1979).

Design patterns … are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. (Gamma et al., 1995)

Example: Pattern Usage and Documentation

Pattern usage in JHotDraw framework ([JHotDraw, 2007]) (Diagram: (Ludewig and Lichter, 2013))
### Example: Strategy

<table>
<thead>
<tr>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
</tr>
<tr>
<td>The only difference between similar classes is that they solve the same problem by different algorithms.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Have one class <code>StrategyContext</code> with all common operations.</td>
</tr>
<tr>
<td>• Another class <code>Strategy</code> provides signatures for all operations to be implemented differently.</td>
</tr>
<tr>
<td>• From <code>Strategy</code>, derive one sub-class <code>ConcreteStrategy</code> for each implementation alternative.</td>
</tr>
<tr>
<td>• <code>StrategyContext</code> uses concrete <code>Strategy</code>-objects to execute the different implementations via delegation.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image-url" alt="Diagram" /></td>
</tr>
</tbody>
</table>

### Example: Pattern Usage and Documentation

<table>
<thead>
<tr>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
</tr>
<tr>
<td>The only difference between similar classes is that they solve the same problem by different algorithms.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image-url" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image-url" alt="Diagram" /></td>
</tr>
</tbody>
</table>
**Example: Pattern Usage and Documentation**

**Observer**

<table>
<thead>
<tr>
<th>Problem</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple objects need to adjust their state if one particular other object is changed.</td>
<td>All GUI object displaying a file system need to change if files are added or removed.</td>
</tr>
</tbody>
</table>

**State**

<table>
<thead>
<tr>
<th>Problem</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>The behaviour of an object depends on its (internal) state.</td>
<td>The effect of pressing the room ventilation button depends (among others?) on whether the ventilation is on or off.</td>
</tr>
</tbody>
</table>
**Example: Pattern Usage and Documentation**

Pattern usage in JHotDraw framework ([JHotDraw, 2007](#)) ([Diagram: [Ludewig and Lichter, 2013](#)])

### Mediator

<table>
<thead>
<tr>
<th>Problem</th>
<th>Objects interacting in a complex way should only be loosely coupled and be easily exchangeable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>Appearance and state of different means of interaction (menus, buttons, input fields) in a graphical user interface (GUI) should be consistent in each interaction state.</td>
</tr>
</tbody>
</table>

### Other Patterns: Singleton and Memento

#### Singleton

<table>
<thead>
<tr>
<th>Problem</th>
<th>Of one class, exactly one instance should exist in the system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>Print spooler.</td>
</tr>
</tbody>
</table>

#### Memento

<table>
<thead>
<tr>
<th>Problem</th>
<th>The state of an object needs to be archived in a way that allows to re-construct this state without violating the principle of data encapsulation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>Undo mechanism.</td>
</tr>
</tbody>
</table>
**Meta Design Pattern: Inversion of Control**

“don’t call us, we’ll call you”

- **User interfaces**, for example:
  - define `button_callback();`
  - register method with UI-framework (→ later),
  - whenever button is pressed (handled by UI-framework),  
    `button_callback()` is called and does its magic.

- Also found in **MVC** and **observer** patterns:
  model notifies view, subject notifies observer.

vs.

- **Classical** (small) embedded controller software:
  - while (true) {
    // read inputs
    // compute updates
    // write outputs
  }

**Design Patterns: Discussion**

“The development of design patterns is considered to be one of the most important innovations of software engineering in recent years.”

*(Ludewig and Lichter, 2013)*

- **Advantages**:
  - *(Re-)use the experience of others and employ well-proven solutions.*
  - Can improve on **quality criteria** like changeability or re-use.
  - Provide a **vocabulary** for the design process,  
    thus facilitates documentation of architectures and discussions about architecture.
  - Can be combined in a flexible way,  
    one class in a particular architecture can correspond to roles of multiple patterns.
  - Helps teaching software design.

- **Disadvantages**:
  - **Using a pattern is not a value as such**.
    Having too much global data cannot be justified by “but it’s the pattern Singleton”.
  - Again: reading is easy, writing need not be.
    Here: Understanding abstract descriptions of design patterns or their use in existing software may be easy – using design patterns appropriately in new designs requires *(surprise, surprise)* experience.
Libraries and Frameworks

- **(Class) Library:** a collection of operations or classes offering generally usable functionality in a re-usable way.
  
  **Examples:**
  - libc — standard C library (is in particular abstraction layer for operating system functions),
  - libz — compress data.
  - libxml — read (and validate) XML file, provide DOM tree.

- **Framework:** class hierarchies which determine a generic solution for similar problems in a particular context.
  
  **Example:** Android Application Framework
Libraries and Frameworks

- **(Class) Library**: a collection of operations or classes offering generally usable functionality in a re-usable way.
  
  **Examples:**
  - libc — standard C library (is in particular abstraction layer for operating system functions),
  - GMP — GNU multi-precision library, cf. Lecture 6,
  - libz — compress data,
  - libxml — read (and validate) XML file, provide DOM tree.

- **Framework**: class hierarchies which determine a generic solution for similar problems in a particular context.
  
  **Example**: Android Application Framework

- The difference lies in **flow-of-control**: library modules are called from user code, frameworks call user code.

- **Product line**: parameterised design/code
  ("all turn indicators are equal, turn indicators in premium cars are more equal").

- For some application domains, there are **reference architectures** (games, compilers).

---

**Quality Criteria on Architectures**
Quality Criteria on Architectures

- **testability**
  - architecture design should keep testing (or formal verification) in mind (**buzzword** “design for verification”),
  - high locality of design units may make testing significantly easier (module testing),
  - particular testing interfaces may improve testability (e.g., allow injection of user input not only via GUI; or provide particular log output for tests).

- **changeability, maintainability**
  - most systems that are used need to be changed or maintained, in particular when requirements change,
  - **risk assessment**: parts of the system with high probability for changes should be designed such that changes are possible with acceptable effort (abstract, modularise, encapsulate).

- **portability**
  - **porting**: adaptation to different platform (OS, hardware, infrastructure),
  - systems with a long lifetime may need to be adapted to different platforms over time, infrastructure like databases may change (→ introduce abstraction layer).

- **Note**:
  - a good design (model) is first of all supposed to support the solution,
  - it need not be a good domain model.

Development Approaches

- **top-down** risk: needed functionality hard to realise on target platform.
- **bottom-up** risk: lower-level units do not “fit together”.
- **inside-out** risk: user interface needed by customer hard to realise with existing system,
- **outside-in** risk: elegant system design not reflected nicely in (already fixed) UI.
Software Entropy

- **Lehman’s Laws of Software Evolution** ([Lehman and Belady, 1985]):
  1. A program that is used will be modified.
  2. When a program is modified, its complexity will increase, provided that one does not actively work against this.

- ([Jacobson et al., 1992]): **Software entropy** $E$ (measure of disorder), claim:
  \[ \Delta E \sim E \]

  - “when designing a system with the intention of it being maintainable, we try to give it the lowest software entropy possible from the beginning.”
  - Work against disorder: **re-factoring**
    - (re-assign data and operations to modules, introduce new layers generalising old and new solutions, (automatically) check that intended interfaces are not bypassed, etc.)

  - **Proposal** ([Jacobson et al., 1992]):
    1. use “probability for change” as guideline in (architecture) design.
    2. i.e. base design on a thorough analysis of problem and solution domain.

<table>
<thead>
<tr>
<th>Item</th>
<th>Probability for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object from application [domain]</td>
<td>Low</td>
</tr>
<tr>
<td>Long-lived information structures</td>
<td>Low</td>
</tr>
<tr>
<td>Passive objects attribute</td>
<td>Medium</td>
</tr>
<tr>
<td>Sequences of behaviour</td>
<td>Medium</td>
</tr>
<tr>
<td>Interface with outside world</td>
<td>High</td>
</tr>
<tr>
<td>Functionality</td>
<td>High</td>
</tr>
</tbody>
</table>

Tell Them What You’ve Told Them...

- **Architecture & Design Patterns**
  - allow re-use of practice-proven designs.
  - promise easier comprehension and maintenance.

- **Notable Architecture Patterns**
  - Layered Architecture.
  - Pipe-Filter.
  - Model-View-Controller.

- **Design Patterns**: read ([Gamma et al., 1995])

- Rule-of-thumb:
  - library modules are called from user-code,
  - framework modules call user-code.

- Mind Lehman’s Laws and software entropy.
Code Quality Assurance

Content (Part II)

- Introduction
  - quotes on testing,
  - systematic testing vs. ‘rumprobieren’
- Test Case
  - definition,
  - execution,
  - positive and negative.
- The Specification of a Software
- Test Suite
- More Vocabulary
**Testing: Introduction**

*Quotes On Testing*

“Testing is the execution of a program with the goal to **discover** errors.”

(***G. J. Myers, 1979***)

“Testing is the demonstration of a program or system with the goal to show that it **does** what it is supposed to **do**.”

(***W. Hetzel, 1984***)

“Software testing can be used to show the presence of bugs, but never to show their absence!”

(***E. W. Dijkstra, 1970***)

**Rule-of-thumb:** (fairly systematic) tests discover half of all errors.

(***Ludewig and Lichter, 2013***)
Tests vs. Systematic Tests

**Test** – (one or multiple) execution(s) of a program on a computer with the goal to find errors. ([Ludewig and Lichter, 2013])

**Systematic Test** – a test such that
- (environment) conditions are defined or precisely documented,
- inputs have been chosen systematically,
- results are documented and assessed according to criteria that have been fixed before. ([Ludewig and Lichter, 2013])

(Our) Synonyms: Experiment, ‘Rumprobieren’.

Not (even) a test (in the sense of this weak definition):
- any inspection of the program,
- demo of the program,
- analysis by software-tools for, e.g., values of metrics,
- investigation of the program with a debugger.

In the following: test means systematic test; if not systematic, call it experiment.

More Formally: Test Case

Definition. A test case $T$ is a pair $(In, Soll)$ consisting of
- a description $In$ of sets of finite input sequences,
- a description $Soll$ of expected outcomes,
and an interpretation $\llbracket \cdot \rrbracket$ of these descriptions.

Plus, strictly speaking, for each pair a description $Env$ of (environmental) conditions, i.e., any aspects which could have an effect on the outcome of the test such as:
- Which program (version) is tested? Built with which compiler, linker, etc.?
- Test host (OS, architecture, memory size, connected devices (configuration?), etc.)?
- Which other software (in which version, configuration) is involved?
- Who is supposed to test when? etc. etc.

→ test-cases should be (as) reproducible and objective (as possible).

Note: inputs can be
- input data, possibly with timing constraints,
- other interaction, e.g., from network,
- initial memory content,
- etc.
Full reproducibility is hardly possible in practice – obviously (err, why…?).

Steps towards reproducibility and objectivity:

- have a fixed build environment,
- use a fixed test host which does not do any other jobs,
- execute test cases automatically (test scripts).

Executing Test Cases: Preliminaries

Recall:

**Definition.** Software is a finite description $S$ of a (possibly infinite) set $[S]$ of (finite or infinite) computation paths of the form $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$ where

- $\sigma_i \in \Sigma, i \in \mathbb{N}_0$, is called state (or configuration), and
- $\alpha_i \in A, i \in \mathbb{N}_0$, is called action (or event).

The (possibly partial) function $[\cdot ] : S \mapsto [S]$ is called interpretation of $S$.

From now on, we assume that states consist of an input and an output/internal part, i.e., there are $\Sigma_{in}$ and $\Sigma_{out}$ such that

$$\Sigma = \Sigma_{in} \times \Sigma_{out}.$$  

Computation paths are then of the form

$$\pi = \left( \begin{array}{c} \sigma_0^I \\ \sigma_0^O \end{array} \right) \xrightarrow{\alpha_1} \left( \begin{array}{c} \sigma_1^I \\ \sigma_1^O \end{array} \right) \xrightarrow{\alpha_2} \cdots$$

$$\in \mathcal{X} = \Sigma_{in} \times \Sigma_{out}.$$
Executing Test Cases

- A computation path
  \[ \pi = \left( \begin{array}{c} \sigma_0^i \\ \sigma_o^i \end{array} \right) \xrightarrow{\alpha_1} \left( \begin{array}{c} \sigma_1^i \\ \sigma_o^i \end{array} \right) \xrightarrow{\alpha_2} \cdots \]
  from \[\llbracket S \rrbracket\] is called execution of test case \((In, Soll)\)
  if and only if there is \(n \in \mathbb{N}_0\) such that \(\sigma_0^i, \sigma_1^i, \ldots, \sigma_n^i \in \llbracket In \rrbracket\).

- \(\pi\) is called successful (or positive) if it discovered an error, i.e., if \(\pi \notin \llbracket Soll \rrbracket\).
  (Alternative: test item \(S\) failed to pass test; confusing: "test failed")

- \(\pi\) is called unsuccessful (or negative) if it did not discover an error, i.e., if \(\pi \in \llbracket Soll \rrbracket\).
  (Alternative: test item \(S\) passed test; okay: "test passed")

- Note: if input sequence not adhered to, or power outage, etc., \(\pi\) is not (even) a test execution.

Test Case Example

- Software \(S\) is the Java program:
  ```java
  public int successor( int x ) { x = x + 1; return x; }
  ```

- Assume that \(\llbracket S \rrbracket\) just considers call and return, i.e. computation paths are of the form
  \[ \left( \begin{array}{c} \sigma_i^0 \\ \sigma_o^0 \end{array} \right) \xrightarrow{T} \left( \begin{array}{c} \sigma_i^1 \\ \sigma_o^1 \end{array} \right) \]
  \(\sigma_i^0(x)\) is the input value for \(x\) and \(\sigma_o^1(ret)\) is the return value.

- Example test case: \((In, Soll) = (27, 28)\) denoting
  \[ [27] := \{ \sigma_i^0(x) = 27 \} \quad [28] := \left\{ \left( \begin{array}{c} \sigma_i^0 \\ \sigma_o^0 \end{array} \right) \xrightarrow{T} \left( \begin{array}{c} \sigma_i^1 \\ \sigma_o^1 \end{array} \right) \mid \sigma_o^1(ret) = 28 \right\}. \]

- Then
  \[ \pi = \left( \begin{array}{c} x = 27 \\ ret = 0 \end{array} \right) \xrightarrow{T} \left( \begin{array}{c} x = 28 \\ ret = 28 \end{array} \right) \]
  is an execution of \((In, Soll)\).

- Is \(\pi\) successful or unsuccessful?
The Specification of a Software

- Same software $S$:
  
  ```java
  public int successor(int x) { x = x + 1; return x; }
  ```

- **Assume** 16-bit int, i.e. value of $x$ is in $[-2^{15}, 2^{15} - 1] = [-32768, 32767]$.

- Test case $(\text{In}, \text{Soll}) = (32767, 32768)$.

- What will $S$ compute?

  $$\pi = \left( x = 32767 \begin{array}{c} \text{ret} = 0 \\ \end{array} \right) \xrightarrow{\tau} \left( x = \begin{array}{c} -32768 \\ \text{ret} = -32768 \\ \end{array} \right)$$

- Is $\pi$ **successful** or **unsuccessful**?

- Well, we operated $S$ **outside its specification**:

  ```java
  successor(int x);
  ```

  - **pre-condition**: $x < 32767$
  - **post-condition**: $\text{ret} = \text{old}(x) + 1$

  If an input does not satisfy the **pre-condition**, $S$ may do "whatever it wants". Its behaviour is **not specified** in that case (aka. chaos).

- Test cases are usually supposed to test that the software satisfies its specification.

By The Way . . .

- **High quality software** should be aware of its specification.

- **successor()** should check its inputs and "**complain**" if operated outside of specification, e.g.
  - throw an exception,
  - abort program execution,
  - (at least) print an error message,
  - etc.

- **Not**: "garbage in, garbage out"
**Wait, Why a Set of Inputs...?**

**Definition.** A test case $T$ is a pair $(In, Soll)$ consisting of

- a description $In$ of sets of finite input sequences,
- a description $Soll$ of expected outcomes,
and an interpretation $\llbracket \cdot \rrbracket$ of these descriptions.

- Sometimes, a test case provides a degree of freedom or choices to the person who conducts the tests.
- For example, for the vending machine

$$In = C50, \text{WATER}$$

could specify

“`At some time` after switching on the vending machine, insert a 50 cent coin, and `some time` later request water.”

without fixing these times, thus there are many valid input sequences.

---

**Test Suite**

- A test suite is a set of test cases.
- An execution of a test suite is a set of computation paths, such that there is at least one execution for each test case.
- An execution of a test suite is called positive if and only if at least one test case execution is positive.
  Otherwise, it is called negative.
Specific Testing Notions

- How are the test cases chosen?
  - Considering only the specification (black-box or function test).
  - Considering the structure of the test item (glass-box or structure test).

- How much effort is put into testing?
  - execution trial – does the program run at all?
  - throw-away-test – invent input and judge output on-the-fly (→ “rumprobieren”),
  - systematic test – somebody (not author!) derives test cases, defines input/soll, documents test execution.

In the long run, systematic tests are more economic.

- Complexity of the test item:
  - unit test – a single program unit is tested (function, sub-routine, method, class, etc.)
  - module test – a component is tested.
  - integration test – the interplay between components is tested.
  - system test – tests a whole system.
Specific Testing Notions Cont’d

- Which **property** is tested?
  - **function test** – functionality as specified by the requirements documents.
  - **installation test** – is it possible to **install** the software with the provided documentation and tools?
  - **recommissioning test** – is it possible to **bring the system back to operation** after operation was stopped?
  - **availability test** – does the system run for the required amount of time without issues.
  - **load and stress test** – does the system behave as required under **high or highest load**? … under overload?
    - “Hey, let’s try how many game objects can be handled!” – that’s an experiment, not a test.
  - **regression test** – does the new version of the software **behave like the old one** on inputs where no behaviour change is expected?
  - **resource tests** – **response time**, minimal **hardware (software) requirements**, etc.

Specific Testing Notions Cont’d

- Which roles are **involved** in testing?
  - **inhouse test** – only developers (meaning: quality assurance roles).
  - **alpha and beta test** – selected (potential) customers.
  - **acceptance test** – the customer tests whether the system (or parts of it, at milestones) test whether the system is acceptable.
A First Rule-of-Thumb

• How to choose test cases?

• “Everything, which is required, must be examined/checked. Otherwise it is uncertain whether the requirements have been understood and realised.”

  (Ludewig and Lichter, 2013)

• In other words:
  Not having at least one (systematic) test case for each (required) feature is (grossly?) negligent (Dt.: (grob?) fahrlässig).

• In even other words:
  Without at least one test case for each feature, we can hardly speak of software engineering.

Tell Them What You’ve Told Them...

• Testing is about
  • finding errors, or
  • demonstrating scenarios.

• A test case consists of
  • input sequences and
  • expected outcome(s).

• A test case execution is
  • positive if an error is found,
  • negative if no error is found.

• A test suite is a set of test cases.

• Distinguish (among others),
  • glass-box test: structure (or source code) of test item available,
  • black-box test: structure not available.
References