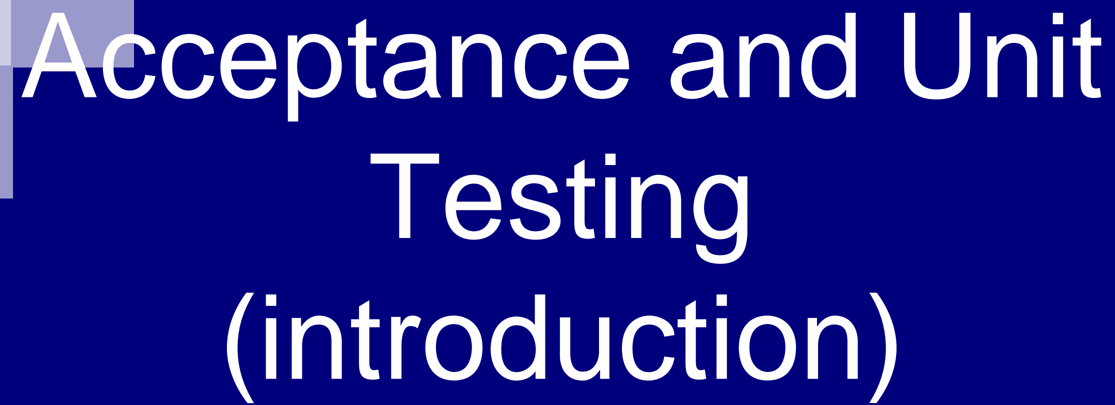
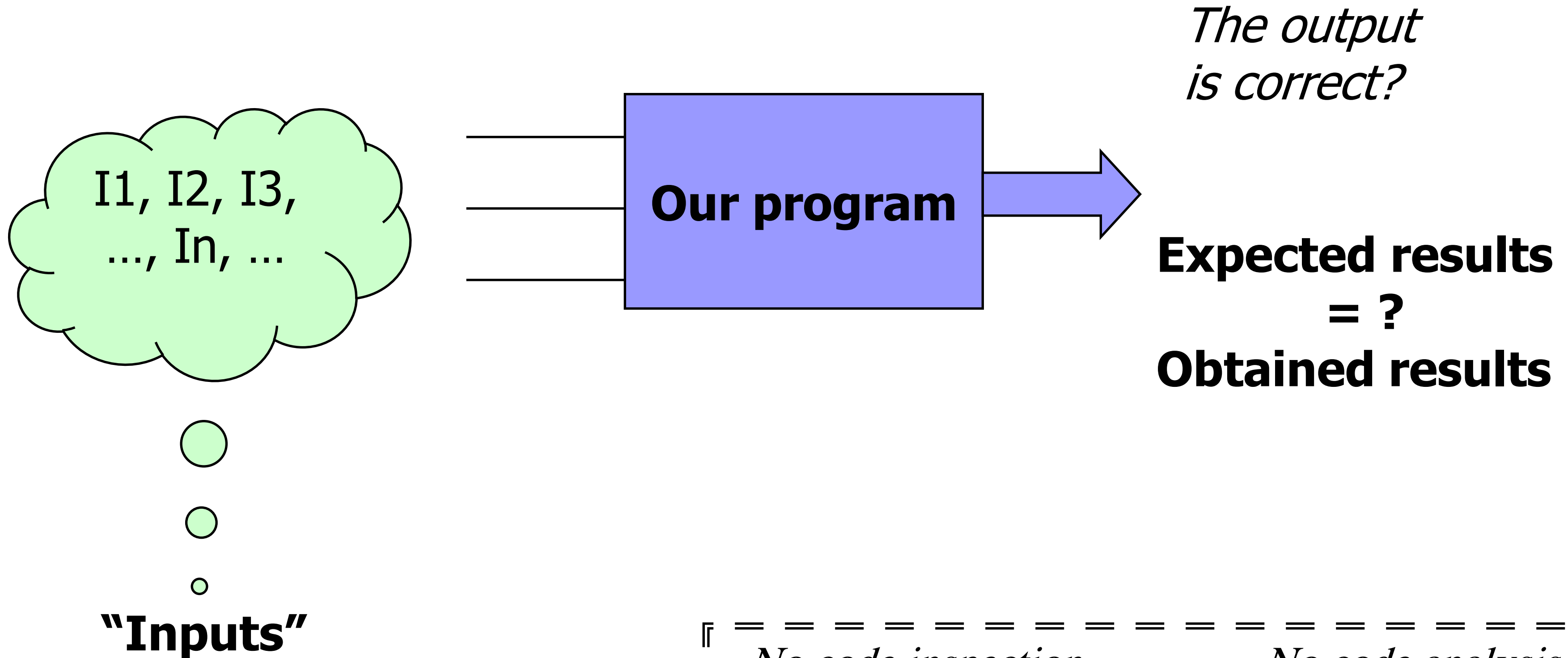
Acceptance and Unit Testing (introduction)



Testing

- One of the practical methods commonly used **to detect the presence of errors (*failures*)** in a computer program is to test it for a set of inputs.



Testing: four main questions

- **At which level conducting the testing?**
  - Unit
  - Integration
  - System
- **How to choose inputs?**
  - using the specifications/use cases/requirements
  - using the code
- **How to identify the expected output?**
  - Test oracles
- **How good test cases are?**
  - When we can stop the testing activity

Test phases

- **Acceptance Testing** – this checks if the overall system is functioning as required.
- **Unit testing** – this is basically testing of a single function, procedure, class.
- **Integration testing** – this checks that units tested in isolation work properly when put togheter.
- **System testing** – here the emphasis is to ensure that the whole system can cope with **real data**, monitor **system performance**, test the system's error handling and recovery routines.
- **Regression Testing** – this checks that the system preserves its functionality after maintenance and/or evolution tasks.

Testing tools

Abbot/JFCUnit/Marathon…

FIT/Fitnesse (*High level*)

GUI

Perfomance and Load Testing JMeter/JUnitPerf

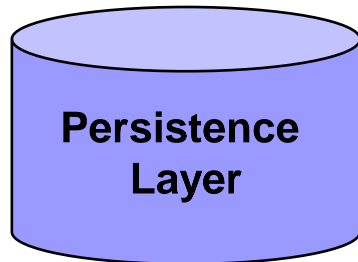
Cactus

Business Logic

HttpUnit/Canoo/Selenium

Junit (*Low level*)

Web UI



Junit/SQLUnit/XMLUnit

Unit Testing

- **Unit Tests** are tests written by the developers to test functionality as they write it.
- Each **unit test** typically tests only a single class, or a small cluster of classes.
- ***Unit tests*** *are typically written using a unit testing framework, such as JUnit (automatic unit tests)*.
- Target errors not found by Unit testing:

  - Requirements are mis-interpreted by developer.

  - Modules don’t integrate with each other

Unit testing: a white-box approach

Testing based on the coverage of the executed **program (source) code**.

Different *coverage criteria*:

- statement coverage
- path coverage
- condition coverage
- definition-use coverage
- …..

It is often the case that it is not possible to cover all code. For instance:

- for the presence of dead code (not executable code)
- for the presence of not feasible path in the CFG
- etc.

project

Acceptance Testing

- **Acceptance Tests** are specified by the customer and analyst to test that the overall system is functioning as required (*Do developers build the right system?*).
- **Acceptance tests** typically test the entire system, or some large chunk of it.
- When all the **acceptance tests** pass for a given user story (or use case, or textual requirement), that story is considered complete.
- At the very least, an **acceptance test** could consist of a script of user interface actions and expected results that a human can run.
- *Ideally **acceptance tests** should be automated, either using the unit testing framework (Junit), or a separate acceptance testing framework (Fitnesse).*

Acceptance Testing

- Used to judge if the product is acceptable to the customer
- Coarse grained tests of business operations
- Scenario/Story-based (contain expectations)
- Simple:
  - Happy paths (confirmatory)
  - Sad paths
  - Alternative paths (deviance)

Acceptance testing: a black-box approach

1. describe the system using a Use-Cases Diagram

   * a use-case of that diagram represents a functionality implemented by the system

2. detail each use-case with a textual description of, e.g., its pre-post conditions and flow of events

   * events are related to: (i) the interactions between system and user; and (ii) the expected actions of the system

   * a flow of events is composed of basic and alternate flows

3. define all instances of each use-case (scenarios) executing the system for realizing the functionality
4. define, at least, one test case for each scenario
5. (opt) define additional test cases to test the interaction between use-cases.

project

How to select input values? (1)

Different approaches can be used:

- **Random values**:
  - for each input parameter we randomly select the values
- **Tester Experience**:
  - for each input we use our experience to select relevant values to test
- **Domain knowledge**:
  - we use requirements information or domain knowledge information to identify relevant values for inputs

project

How to select input values? (2)

Different approaches can be used:

- **Equivalence classes**:
  - we subdivide the input domain into a small number of sub-domains
  - the equivalence classes are created assuming that the SUT exhibits the same behavior on all elements
  - few values for each classes can be used for our testing

- **Boundary values**:
  - is a test selection technique that targets faults in applications at the “boundaries” of equivalence classes
  - experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes

project

How to select input values? (3)

- **Combinatorial testing**:
  - test all possible combination of the inputs is often impossible

    e.g., method(a:int,b:int,c:int) .. how many combinations?

    with 10 values per input: $10^3 = 1000$

    with 100 values per input: $100^3 = 1000000$

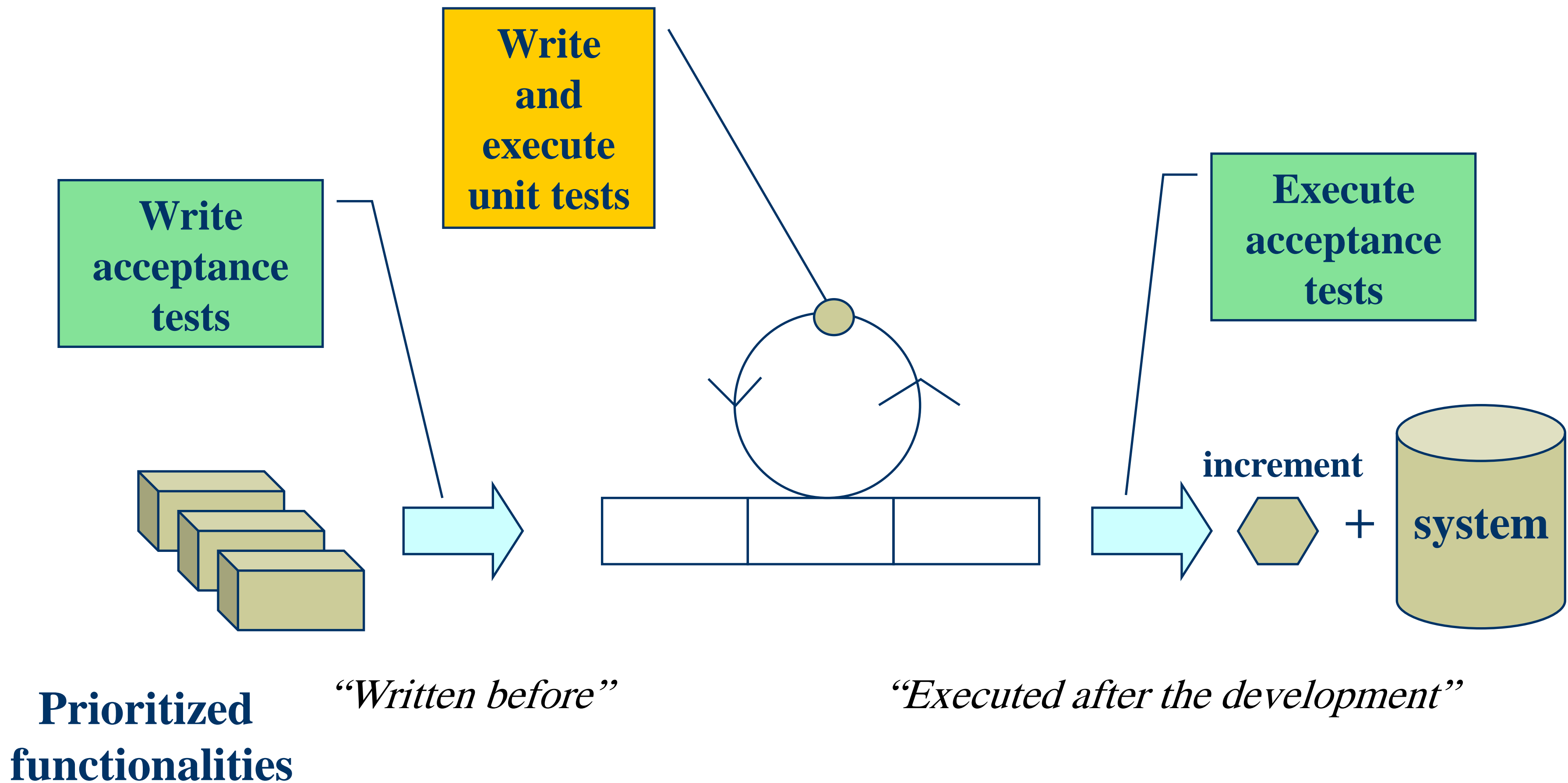
  - selection of relevant combinations is important
    - *Pairwise testing (aka 2-way)*: cover all combinations for each pair of inputs

      <a,b> <a,c> <b,c> = $10^2 + 10^2 + 10^2 = 300$

      don't care about the value of the third input

project

Iterative Software development



Acceptance vs Unit Testing

In theory:

Acceptance Tests	Unit Tests
Written by Customer and Analyst.	Written by developers.
Written using an acceptance testing framework (also unit testing framework).	Written using a unit testing framework.
(extreme programming) When acceptance tests pass, stop coding. The job is done.	(extreme programming) When unit tests pass, write another test that fails.
The motivation of acceptance testing is demonstrating working functionalities.	The motivation of unit testing is finding faults.
Used to verify that the implementation is complete and correct. Used for Integration, System, and regression testing. Used to indicate the progress in the development phase. (Usually as %). Used as a contract. Used for documentation (high level)	Used to find faults in individual modules or units (individual programs, functions, procedures, web pages, menus, classes, …) of source code. Used for documentation (low level)
Written before the development and executed after.	Written and executed during the development.
Starting point: User stories, User needs, Use Cases, Textual Requirements, …	Starting point: new capability (to add a new module/function or class/method).

Acceptance vs Unit Testing

In practice: The difference is not so clear-cut.

- We can often use the same tools for either or both kinds of tests.

Traditional Approaches for acceptance testing

- **Manual Acceptance testing.** User exercises the system manually using his creativity.
- **Acceptance testing with "GUI Test Drivers"** (at the GUI level). These tools help the developer do functional/acceptance testing through a user interface such as a native GUI or web interface. "**Capture and Replay" Tools capture events** (e.g. mouse, keyboard) in modifiable script.

Disadvantages: expensive, error prone, not repeatable, …

Disavantages: Tests are brittle, i.e., have to be re-captured if the GUI changes.

"Avoid acceptance testing only in final stage: Too late to find bugs"

Table-based Approach for acceptance testing

- Starting from a user story (or use case or textual requirement), the customer enters in a table (spreadsheet application, html, Word, …) the expectations of the program’s behavior.
- At this point tables can be used as oracle. The customer can **manually** insert inputs in the System and compare outputs with expected results.

Microsoft Excel - Rating.xls

File Edit View Insert Format Tools Data Window Help

D6 fx 20

	B	C	D	E	F	G
4	Team Name	Played	Won	Drawn	Lost	Rating
5	Arsenal	38	31	2	5	83
6	Aston Villa	38	20	2	16	54
7	Chelsea	38	35	1	2	93

Fit hide

inputs

output

Pro: help to clarify requirements, used in System testing, …
Cons: expensive, error prone, …

Table-based test cases can help in clarifying requirements

- It is estimated that 85% of the defects in developed software originate in the requirements (communication between customer and analyst, communication between analyst and developer).
- There are several "sins" to avoid when specifying requirements:
  - noise
  - silence
  - ambiguity
  - over-specification
  - wishful thinking,
- … => ambiguous, inconsistent, unusable requirements.

"order-processing system for a brewery"

- *if a retail store buys 50 cases of a seasonal brew, no discount is applied; but if the 50 cases are not seasonal a 12% discount is applied. If a store buys 100 cases of a seasonal brew, a discount is applied, but it's only 5%. A 100-case order of a non-seasonal drink is discounted at 17%. There are similar rules for buying in quantities of 200.*

DiscountStructure.xls - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

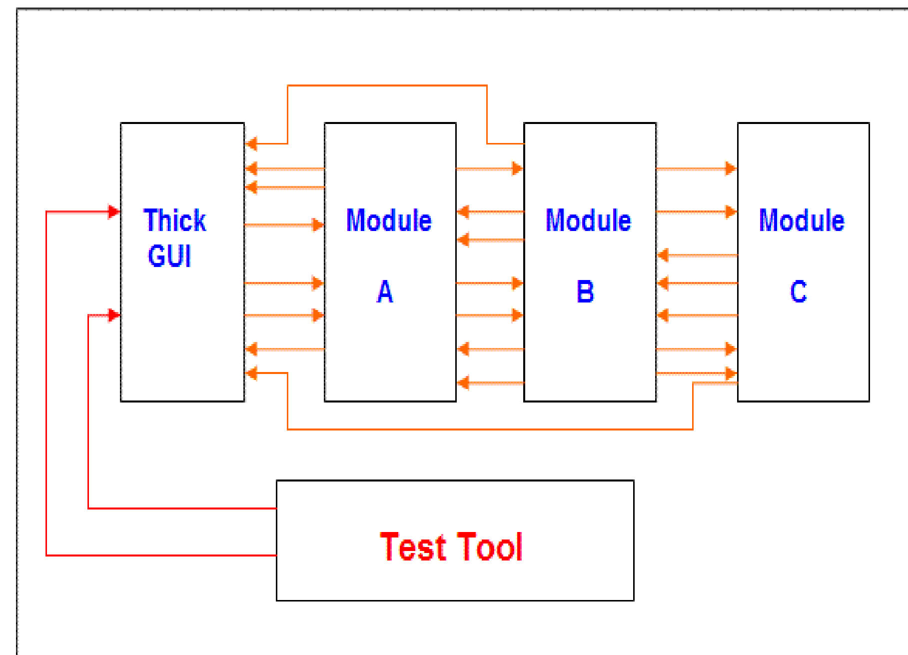
org.acme.store.discount.DiscountStructureFIT				
list price per case	*number of cases*	*is seasonal*	*discount price()*	*discount amount()*
10	10	true	100	0
10	10	false	95	5
10	50	true	500	0
10	50	false	440	60
10	100	true	950	50
10	100	false	830	170
10	200	true	1800	200
10	200	false	1600	400

Done

Badly designed systems makes testing difficult

- We have a thick GUI that has program logic. The interfaces between the modules are not clearly defined.
- Testing of specific functions (Unit Testing) cannot be isolated.
- Testing has to be done through the GUI => Fit/Fitnesse is not sufficient.
- Testing is difficult.

"Badly designed system"

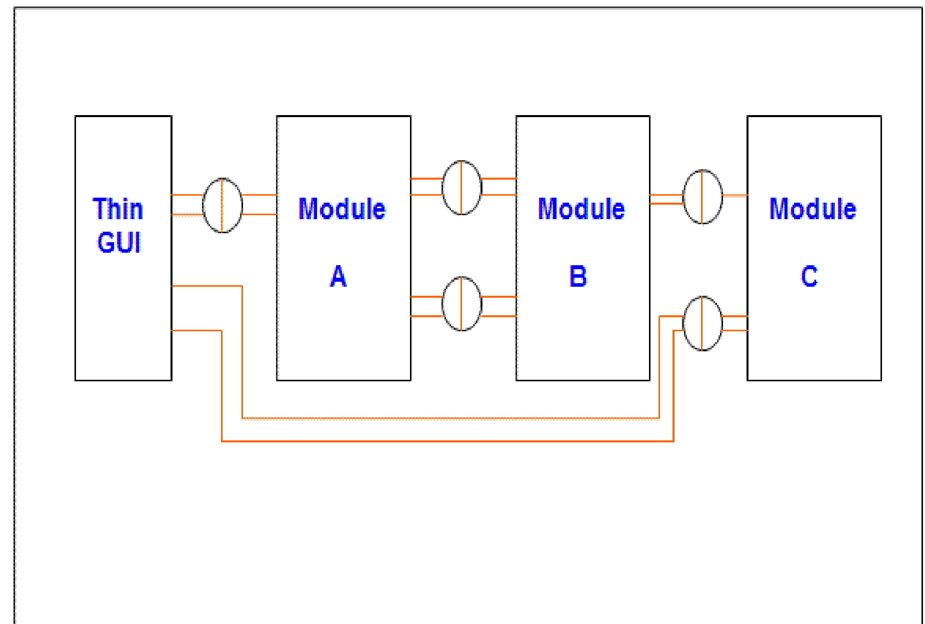


GUI Test Drivers

Well architected applications makes testing simple

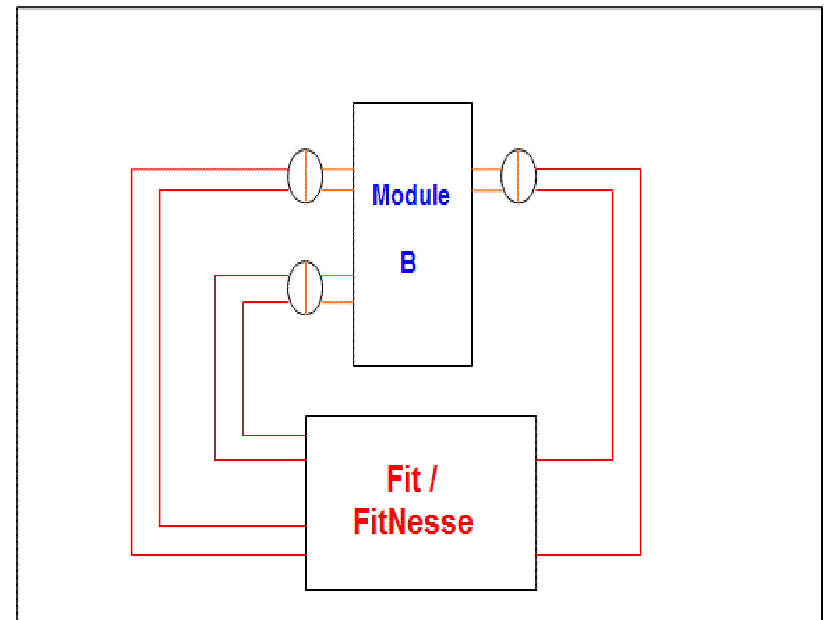
- The GUI does not contain any program logic other than dealing with presentation.
- The interfaces between the modules are well defined.
- This give us testing advantages. Unit and System acceptance testing are simpler.

"Well architected application"



Well architected applications makes testing simple: Testing a Module

- When an application has modules with well defined interfaces, each module can be tested independently from the other modules.
- Using this type of environment the developer can test the module to make sure everything is working before trying to integrate it with other modules.
- This system does not require Fit/ FitNesse. You could use any automated test harness that works for your application (i.e., Junit).



Test Tool = Fit/Fitnesse or Junit

Conclusions

- Badly designed systems makes testing difficult. Unit testing is complex and all end-to-end tests are through the GUI.
- Well architected applications simplify testing. Unit testing is simple and end-to-end tests are through interfaces of modules.
- The motivation of Acceptance testing is demonstrating working functionalities.
- The motivation of Junit is finding faults.
- Manual acceptance testing is expensive, error prone and not repeatable.
- Table-based test cases help to clarify “textual requirements”.
- Table-based test cases can be “requirements verifiable and executable”.
- Table-based test cases can be useful for Managers, Customers, Analysts and Developers.

Additional references

-Jim Heumann. Generating Test Cases From Use Cases. Online IBM journal. 2001 http://ww.ibm.com/developerworks/rational/library/content/RationalEdge/jun01/GeneratingTestCasesFromUseCasesJune01.pdf

- Peter Zielczynski. Traceability from Use Cases to Test Cases. online IBM journal 2006 http://www.ibm.com/developerworks/rational/library/04/r-3217/

- R.C.Martin and G.Melnik. Tests and Requirements, Requirements and Tests: A Möbius Strip. IEEE Software 2008. http://www.gmelnik.com/papers/IEEE_Software_Moebius_GMelnik_RMartin.pdf

-J. Aarniala, University of Helsinki. Acceptance Testing, Helsinki, October 30, 2006. www.cs.helsinki.fi/u/jaarnial/jaarnial-testing.pdf