Course: Software Engineering
Multimedia University Cyberjaya,

ANSWERS TO ASSIGNMENT NO. 7

1. **User Interface (or UI)** is one of those jargon terms that you hear from computer salespeople and other techno-geeks, but that you may have never heard defined. It's not a hard concept to understand, though. It's simply the parts of a computer and its software that you (the computer user) see, hear, touch, or talk to. It is the set of all the things that allow you and your computer to communicate with each other. The “user interface” influences the user’s perception of the software. The Good Book in Chapter 12 focuses specifically on User Interface Design.

(a) IBM claimed to have pioneered Object Oriented User Interface (OOUI) architecture and design (Reading-55.pdf). List down 10 of the UI design principles advocated by IBM.

**ANSWER**

The answer to this question is straightforward. The principles in UI design by IBM are as follows:

1. **Simplicity:** Don’t compromise usability for function
2. **Support:** Place the user in control and provide proactive assistance
3. **Familiarity:** Build on users’ prior knowledge
4. **Obviousness:** Make objects and their controls visible and intuitive
5. **Encouragement:** Make actions predictable and reversible
6. **Satisfaction:** Create a feeling of progress and achievement
7. **Availability:** Make all objects available at all times
8. **Safety:** Keep the user out of trouble
9. **Versatility:** Support alternate interaction techniques
10. **Personalization:** Allow users to customize

(b) When applied to computer software, User Interface Design (UID) is also known as Human-Computer Interaction or HCI (Reading-52.pdf). Other than software what other products apply User Interface Design? Please list down 5 examples of specific interfaces for these products. (Hint: Our 2 nostrils (the 2 holes in our nose) face downwards and not upwards)

**ANSWER**

Quote from the Good Book: “While people often think of User Interface Design (UID) in terms of computers, it also refers to many products where the user interacts with controls or displays. Military aircraft, vehicles, airports, audio equipment, and computer peripherals, are a few products that extensively apply User Interface Design.”

As long as there is “a required human interaction with something”, then there is User Interface Design. Some examples of UID for non software products are:
1. Car steering wheel, gear stick, driver’s seat, accelerator, clutch and brake pedals – the design and locations of those things we comfortably experience in driving a car today. They are properly designed.

2. “Steering wheel” – different for commercial jet aircraft compared to military jet fighters, and different for helicopters (joystick). Ha.. ha.. ha.. Then they have buttons for guns, missiles, bombs, etc. And they must be designed appropriately.

3. Bicycle – How do humans interact with a bicycle? You need a handle bar, brake control, a seat and a set of foot pedals, i.e. all the things on the bicycle that are in contact with humans, and they must be designed properly. For example, the seat must be sized suitably and different for an adult compared to a small kid, otherwise we get “PITA” ha ha ha “pain in the ass”.

4. Home Appliances – Your TV, stereo set, computer notebook, water heater, air conditioners, microwave oven, refrigerator etc all require some kind of human interface to them, either by manual control (touching and doing something) or remote control (still touching and doing something). The interface must be designed properly.

5. Human Body – We constantly interact with our environment, like earth, land, water and air. As examples: The Big Guy (God) created our eyes for seeing (lightning), hands for doing (whatever things) and touching (sensually), our ears for hearing (thunder) and our nose for smelling (flowers, food) and many more. The Big Guy designed our two nostrils to be two open holes, face downwards and not upwards because we walk vertically and our nostrils, if designed facing upwards will become water catchments when it rains. Ha.. ha.. aha.. So the Big Guy really has UID principles.

(c) List down 5 items on “shameful” UID practices for software (Reading-58.pdf) and another 5 items on “good” UID practices for software (Reading-59.pdf).

ANSWER

EXAMPLES OF 5 SHAMEFUL UID PRACTICES

1. Selecting the wrong control for a task or changing the way controls operate can often result in an inefficient and frustrating application.

2. Improper design of the visual elements in an application can often result in applications that are difficult to read and difficult to use.

3. The terminology used in many applications often leads the user to feel that the interface has been written in a foreign language.

4. Nobody likes a stupid computer. However, many applications interrupt the user to ask stupid questions, provide meaningless information, or require the user to make what should be an obvious selection.

5. The improper use color in an application can seriously impede the usability of the application.
EXAMPLES OF 5 GOOD UID PRACTICES

1. On your interface screen, ask only the required inputs. Non
   mandatory inputs must be properly labeled.
2. For input screens, provide a list box for specific options and arrange
   the most common expected answers from the top downwards.
3. If you have to display results or output various information, you should
   separate the displays into separate tabs and organize the information
   such that each tab screen displays only information related to the tab
   text label. (Do not mix unrelated information under the same tab
   screen).
4. When you need to put pictures or diagrams on your screen, the sizes
   and layout of your pictures against your descriptive text must be
   proportional to their importance or relative value of information to be
   conveyed (emphasis).
5. When you create graphical screen Menu Items, please follow standard
   designs and common expectations. People are familiar with the ‘Exit’
   command in the leftmost menu item (File) and the bottommost of the
   (File select list). Do not put it somewhere else, people will be very
   annoyed.

REFERENCES:
http://developer.gnome.org/projects/gup/hig/2.0/
http://homepage.mac.com/bradster/iarchitect/shame.htm
http://homepage.mac.com/bradster/iarchitect/fame.htm

2. The Good Book in Chapter 14 states: "At first glance, it would seem that very
   thorough white-box testing would lead to 100 percent correct programs".
   (a) Why is 100 percent white-box testing impractical?

ANSWER

This subject is addressed in the Good Book on page 423, Section 14.2.

In software testing, if you have a module for example and you tested every possible
path in the module, you call it "white-box" testing. It means for white-box testing,
you test from the inside (‘white-box’ or everything inside is known to you), so it is
achievable to test all possible paths.

If you test the ‘documented’ or ‘stated’ functionality of the module or the complete
system, but without testing every possible path of the module or of the complete
system, then the testing is called “black-box” testing. It means for black-box
testing, you test the interface (documented functionality) from the outside (‘black-
box’ or inside is unknown to you), whether you are testing just the module or the full
complete system.

Simply said, the correctness of "internal software program logic" is executed using
"white-box" test case design techniques. Whereas, the correctness of "software built
against user requirements” is executed using “black-box” test case design techniques.

Therefore, a 100 percent white-box testing is impractical when you do exhaustive testing for large software systems because it will be too time-consuming to execute as you have to go through all types of inputs, all ranges of inputs, most importantly all possible logical program paths covering all the software components and their message and data passing. It may be possible for simple and small software systems but it can be impractical for very large systems. Normally, people will perform the test by selecting a limited number of important logical paths.

(b) What is meant by static testing tools?

**ANSWER**

This subject is addressed in the Good Book on page 441, Software Tools under Test Case Design.

Static testing tools are software tools for testing programs that do not interact with the executing program. The case is opposite for dynamic testing tools which conduct its tests on software that is executing (running).

The static testing tools interact with the source codes and the documented user requirements. The 3 types for static testing tools:

- **In the code-based testing tool,** the tools read the source codes (static) as input and perform a number of analyses then generate a number of test cases. Using these test cases, you can manually run the tests.

- **In the specialized-language testing tool,** (like ATLAS – An Automated Software Testing System) someone must first convert the program logic into test diagrams (constrained directed graph software model etc, e.g. please see Reading-64 & Reading-65) understandable by the testing tool, then the testing tool generate test cases that can be run manually or automatically.

- **In the requirements-based testing tool,** someone must also convert or translate the documented user requirements into specific test cases understandable by the software tool, execute the test manually or automatically using the tool.

(c) What is meant by dynamic testing tools?

**ANSWER**

Dynamic testing tools interact with an executing program by checking its path coverage, testing assertions about the value of specific variables, and otherwise providing, recording, measuring or “instrumenting” the execution flow of the program.

- One example for a dynamic testing tool is a “Web Stress Test” that I have personally conducted on a running web server using a tool that simulates
concurrent access to the web server for a specific page or software functionality. By varying the number of simultaneous access, the software records the response times (including averages etc) against the number of accesses. It also does other dynamic tests. Please see Reading-66.

(d) Are there software tools (CASE) that automatically generate test cases? Name a few of them.

ANSWER

In Reading-67 on this course website we provided the latest but not necessarily comprehensive report (year 2005 in the Academic Open Internet Journal) on A Survey on Automatic Test Case Generation. The paper presents a survey on automatic test case generation techniques that are found in the current literature.

Most of the tools involve test cases generated from UML diagrams, now you realize why UML diagrams are important, because if you use them correctly, you can then easily and automatically generate test cases from them.

One example (Please see Reading-68 ) Jtest is an automated unit testing and static analysis tool that includes and extends JUnit to provide automatic test case generation and automatic checking of over 300 Java coding standards.

(c) In the Open Source Community, there is something on testing called JUnit test cases and JUnit test suites. In your own words, explain what this so called JUnit is all about.

(Hint: http://www.clarkware.com/articles/JUnitPrimer.html )

ANSWER

If you see the answer to question (d) above in the last paragraph we said that there is a direct relationship between Jtest and Junit namely: "Jtest is an automated unit testing .... and includes and extends JUnit ....". It cannot be clearer than that.

Please see Reading-69 on JUnit testing. JUnit is free.

In computer programming, a unit test is a method of testing the correctness of a particular module of source code. The idea is to write test cases for every non-trivial function or method in the module so that each test case is separate from the others if possible.

By saying it simply, JUnit is an open source testing framework for Java. It provides a very simple way to express the way you intend your code to work. By expressing your intentions in code, you can use the JUnit test runners to verify that your code behaves according to your intentions.

So about JUnit specifically:

- JUnit is a unit test framework for the Java programming language.
- JUnit was created by Kent Beck along with Erich Gamma.
- Since then, JUnit has served as an inspiration and a role model for a variety of other unit test frameworks for other languages.
Experience gained with JUnit has been important in the development of Test-driven development, and as a result, some knowledge of JUnit is often presumed in discussions of Test-driven development.

Eclipse is one of the IDE tools we have used in this software engineering class. In Reading-70 on this course website, you will see “Using JUnit Eclipse IDE” - an article that is going to introduce you to JUnit, a tool for project testing and debugging. After introducing the theory of test-driven development, we'll move on to a step-by-step explanation of how you can create your JUnit tests with the help of the popular Eclipse IDE. We'll show how something as simple as a Hello World program can be exposed to a JUnit test. Are you now excited about using it? Ha.. ha.. ha.. WRY

3. The Good Book (Chapter 14) also states that: "Black-box (behavioral) testing is not an alternative to white-box techniques”.

Explain the differences between black-box testing and white-box testing.

**ANSWER**

The simple explanations about white-box testing and black-box testing and their differences have been provided in the first 2 paragraphs in ANSWER 2(a) above.

The statement "Black-box (behavioral) testing is not an alternative to white-box techniques” means that you have to execute both types of testing. It is not an alternative simply means that it is not a choice.

If you do not do black-box testing then you are not checking your software against the software functional or behavioral requirements. If you do not do white-box testing then you are not checking your software codes for logical program errors.

You may however not do a 100% full white-box testing, but you must do a 100% functional testing (black-box) otherwise you client will not pay you as you did not fully meet your client’s user requirements as you are supposed to. Normally, people will go through a long checklist for functional testing during the final commissioning and full acceptance testing, normally done by many people. If you still fail some functional tests (“kena hutang”) your client may accept the system you deliver as partial fulfillment only. You can get paid but not in full.


(a) What does the term “Verification” mean to you? Answer this question according to the above IEEE standard and compare with the answer in the Good Book (Chapter 13).

(b) What does the term “Validation” mean to you? Similarly, answer this question according to the above IEEE standard and compare with the answer in the Good Book (Chapter 13).
ANSWER

In the Good Book Chapter 13 (page 388),
- Verification: Are we building the product right?
- Validation: Are we building the right product?

According to the IEEE Standard 1012-2004 (page 19 of 120) under DEFINITIONS:

3.1.35 validation:

(A) The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
(B) The process of providing evidence that the software and its associated products satisfy system requirements allocated to software at the end of each life cycle activity, solve the right problem (e.g., correctly model physical laws, implement business rules, use the proper system assumptions), and satisfy intended use and user needs.

3.1.36 verification:

(A) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
(B) The process of providing objective evidence that the software and its associated products conform to requirements (e.g., for correctness, completeness, consistency, accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance); satisfy standards, practices, and conventions during life cycle processes; and successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities (e.g., building the software correctly).

(c) Describe the 4 levels of software integrity scheme as stipulated in the said IEEE Standard.

ANSWER

On page 21 of 120 in the IEEE Standard 1012-2004, we find:

As an example, this standard uses the following four-level software integrity scheme. This example scheme is based upon the concepts of consequences and mitigation potential.

**Integrity Level 4**
Software element must execute correctly or grave consequences (loss of life, loss of system, economic or social loss) will occur. No mitigation is possible.

**Integrity Level 3**
Software element must execute correctly or the intended use (mission) of the system/software will not be realized, causing serious consequences (permanent injury, major system degradation, economic or social impact). Partial to complete mitigation is possible.

**Integrity Level 2**
Software element must execute correctly or an intended function will not be realized, causing minor consequences. Complete mitigation possible.
**Integrity Level 1**
Software element must execute correctly or intended function will not be realized, causing negligible consequences. Mitigation not required.

(d) In the said IEEE Standard in Annex G, we have an item with the subtitle "Algorithm analysis." And the words “verify” and “validate” have been used on the same thing. Explain the difference. (Hint: Please see Quote 1 and Quote 2 below.)

*Quote 1 - Verify the correct implementation of algorithms, equations, mathematical formulations, or expressions.*

*Quote 2 - Validate the algorithms, equations, mathematical formulations, or expressions with respect to the system and software requirements.***

**ANSWER**

Verify the correct implementation means making sure it is the intrinsically (absolutely) the correct thing to do. Build the product right absolutely.

Validate with respect to the system and software requirements, which is not necessarily the intrinsically correct thing. Just meet the requirements- just build the right product as the requirements specify.

Another view:

**Validation (for software):** The process of evaluating software to ensure compliance with specified requirements. (ISO 9000-3: 1991, 3.7) We build our software meeting the specified requirements. This answers the question “Have we built the right product?” as YES.

**Verification (for software):** The process of evaluating the products of a given phase to ensure correctness and consistency with respect to the products and standards provided as input to that phase. (ISO 9000-3: 1991, 3.6) This answers the question "Have we built the product right?"

Simply stated, for a given set of inputs to the software, upon testing we should get the expected set of correct outputs. This type of test verifies that the software has performed correctly. So **verification** answers the question “Have we built the product right?” as YES.