CSCI 3100 Software Engineering

Assignment 1

Jonathan Lam Pui Hong (1155093597)

Question 1

Quality	Internal	External	Product	Process
Correctness	√		√	
Reliability		√	√	
Robustness	√	√	√	
Performance	√	√	√	√
User Friendliness		√	√	
Verifiability	√	√	√	
Maintainability	√		√	√
Repairability	√		√	√
Evolvability	√		√	√
Reusability	√			√
Portability	√		√	√
Understandability	√	√	√	√
Interoperability	√		√	√
Productivity	√			√
Timeliness	√			√
Visibility	√	√	√	√

Correctness: The measure of a program's or software's correctness is (more) of software engineers' concern as they are responsible for its implementation and testing – hence, internal. The product is compared with the specification – hence, product.

Reliability: The measure of a software reliability is of users' concern as they are the one who uses the software/product and expect it to operate as specified – hence, external and product.

Robustness: Developers are those who provides the test cases and the specification to which the product tests against. The failure of an adequate test set will burden the developers, as they are responsible for maintenance, but also the users, as they are using the product and expects it to behave or function reasonably.

Performance: Developers are expected to enhance the performance of a software in all of the phases by utilizing the computer resources – hence, internal and process. However, whether the finishing products (or the ones along the way) are usable are evaluated exactly by its performance, such as run time or space allotments – hence, external and product.

User friendliness: A good user interface provides an easy-to-use front for users, i.e. the users and the product are communicating well with each other – hence, external and product.

Verifiability: A product's properties are verified – hence, product. It is usually an internal quality that developers have to keep in mind in the phases to ensure analysis methods are provided. As mentioned in lecture, sometimes it is an external quality such as when dealing with security of applications.

Maintainability, repairability, and evolvability: The product release is subject to modification and correction. It is of developers concern to maintain the usability of the product and carry out such changes, specifically on correcting existing product (repairability) and enhance its performance (evolvability). This would also require an anticipation of change during the designing phase to allow such modification – hence, also process.

Reusability and Portability: While reusability and portability are closely related to evolvability, its concern is less of one specific product but the possibility for more, mostly similar, products. It requires a careful planning for easy modification and transplantation to other program – hence, process. Portability is a special case of reusability that concern the use of a software across multiple platforms or environments – hence, product.

Understandability: The program has to be understandable for developers for continuous maintenance and improvements over the phases – hence, internal and process, but also for users for grasping what the software is doing – hence, external and product.

Interoperability: It is of developers' concern of whether the product is interoperable with other software. It is both a process quality since it has to be carefully planned and executed in the phases to ensure proper systems, interfaces, and environments are used, and a product quality since it is the product which would be interoperate with other systems.

Productivity and timeliness: Developers come together to manage the software development process to ensure its efficiency and performance, which will in turn determine the timeliness of software – hence, internal and process.

Visibility: Documentation itself is a product to both the developers and users in order to evaluate the action taken throughout the phases. This also requires careful planning and execution in the phases to make sure everything is documented – hence, process.

Question 2

a. True. Maintainability has three major aspects: corrective, adaptive, and perfective maintenance. Repairability mostly concerns the first corrective aspect while evolvability concerns the last two aspects. If a software has high repairability and evolvability, maintenance would be easier; meanwhile, a high maintenance *per se* entails easiness of repairment and evolvement.

b. False. A system may be correct but not necessarily robust. It relies mostly on the specification. While the system may be correct (compared to the software's specification), it may not be able to handle unexpected situation outside of the specification.

c. True. Good productivity means resources are distributed evenly among developers, which in turn means efficient implementation of the software, allowing the software to be published on time per the process design.

d. False. If a program is robust, it is more adaptive to error and unspecified environment, hence more reliable for continue usage of the software. The reverse, however, may not be true as a program's reliability, i.e. it operates as expected as in specification, does not concern of field outside of specification – the place where robustness mostly evaluation.

e. True. Performance, especially on space and time requirements, directly affects whether a program is usable. For example, a program with poor running time, say, takes a minute to compute a 30-second limit decision, may cause itself unusable.

f. True. For a program to be understandable, especially to developers, it would allow further modification and evolvements to the program for interoperation in other systems. When the software is also understandable to the users, they are more invited to interoperate the systems. For example, a music app allows sharing to another social media app. It is in the system, but only when users understand how to do that, it becomes interoperable.

g. True. To determine a software's correctness, it would require something to validate or verify the program – such something or such process is the software's verifiability, allowing the software's properties, including correctness, to be verified easily through various means.

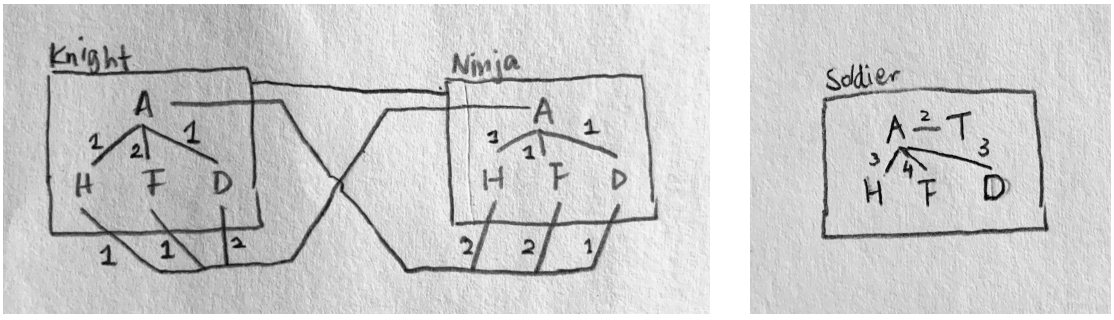
h. True. Even if the software has poor understandability, the documentations of every process ensure there are referrable documents to clarify such confusing bits originated in the software or program.

i. False. They differ in the target concerned. Usability is concerning of the specific software, whether it is usable or not, or whether the software is optimal for using; Reusability is concerning of multiple (possible) software, whether one software can be redesigned and transplanted to create more software. Reusability concerns more of things like functions in the program or phases of the software model, while usability concerns the product as a whole. A software can be unusable but parts of it may be used – i.e. low usability but high reusability.

j. True. Portability, the measure of a software's ability to run in different environment, is a case of reusability, the measure of a software's ability to branch out to different products. In a stricter sense, it is a new product already when a software branches out from its intended environment to another environment.

Question 3

a. Mary's design is better because it has higher cohesion and lower coupling.



Left: Tony's code, 4 + 3 = 7 cohesion, 4 + 5 = 9 coupling.
Right: Mary's code, 12 cohesion, 0 coupling.

b. If one does not inherit *soldiers* from a common parent class, any special interaction between classes (modules) has to be separately coded in different classes (modules). In addition, more modules combinations (new *void* method) are required in every class, such as *Knight* attack *Dragon*, *Ninja* attack *Dragon*, and vice versa. This will cause a larger workload and make the code harder to maintain were there any error arise, as error are spread out in different instances.

Anticipation of change, generality, and abstraction. Inheritance generalizes various instances , in this case *Ninja*, *Knight*, and *Dragon*, into one class, *soldier* with similar attributes and methods. It abstracts the details, which are not as important as the class they are in, and amplified the pattern they exhibit, which in this case is the *HP, force, defense,* and ability to *attack*. Inheritance, and specifically its generality nature, allows changes and updates to be applied easily, which in this case is the implementation of new soldier *Dragon*.

Maintainability, evolvability, reusability, and understandability. As mentioned, the code with inheritance allows new features (instances) under the same class to be easily implemented and added. This contributes greatly to the maintainability and evolvability of the software, where there is less work in configuring every instance and making sure everything agrees with one another were there not inheritance. It is reusable in a sense that it is kind of object oriented and any new features, thanks to the software maintainability and evolvability, would be counted as new project/software. The code is also more understandable as it generalizes instances into classes, showing a pattern instead of special interactions. In the case of *Warcraft, soldiers*' attack has an attack-counterattack pattern, rather than duplicating codes in different attack methods. (For example, the first two lines below can be read as the third line instead.)

```
this->HP -= 0.5 *    ninja.force - this->defense
this->HP -= 0.5 *   knight.force - this->defense

this->HP -= 0.5 * opponent.force - this->defense
```

Question 4

a.

Class	Variables		Function

`Soldier`	`int type`
	`int side`
	`int HP`
		`int force`
		`int defense`
		`Castle location`
			`attack(Soldier soldier)`
			`move(Castle castle)`

`Castle`	`int headquarter` `int soul`	`int timestamp` `Soldier occupant`	`produce_soldier()`

`Soldier::move(Castle castle)` checks the next `castle`'s `occupant`, if:

- null: move into the castle* i.e. update `location` to next `castle`
- same side: stay in current `location`
- opposite side: calls `attack(castle.occupant)`
    - if wins: move into the castle* i.e. update `location` to next `castle`
    - if loses: remove itself from `location` (`location=null`)

* Any successful move, `location` is updated, and a check is triggered to see if the new `location` is a `headquarter` and is occupied by an ally; if so, game ends; if not, any `soul` in the new `location` will be added to corresponding side's headquarter `Castle` (by back-tracking until castle is a headquarter.)

`Timestamp` will be incremented with time. Upon reaching to a certain threshold *t*, 1) if the `castle` is a `headquarter`, it will `produce_soldier` randomly according to its `soul` and, if success, automatically call `Soldier->move`; 2) if it is occupied, it will trigger the `occupant`'s `move` event.

b. eXtreme programming.

The foremost feature is the flexibility it provides. While traditional models like the Waterfall model provide a clear and straightforward development, in cases such as this where new soldiers or mechanism is needed while maintaining a working edition of software, eXtreme programming allows rapidly changing development stages where each release is independently complete and "builds on" the previous releases.

The second feature is the user involvement. Unlike traditional models where everything is well planned out by the developers before the next stage is processed, eXtreme programming relies heavily on users input on each release. This is particularly important in games like this where experience is mostly subjective to the large crowd the software reaches. Users involvement means more new mechanism and ideas could be implemented into the software and allow rapid change of plans – which eXtreme programming, being an agile development model, excels.