(1)

Test-driven development (TDD) :

a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. TDD often requires longer development time, but can ensure higher quality (lower defect rate).

Pros:

- TDD can lead to more modularized, flexible, and extensible code.
- Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program.[15] By focusing on the test cases first, one must imagine how the functionality is used by clients (in the first case, the test cases). So, the programmer is concerned with the interface before the implementation.
- Programmer need not (or less) to launch debug environment later in the process.
- Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. As a result, the automated tests resulting from TDD tend to be very thorough: they detect any unexpected changes in the code's behaviour. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Cons:

- Unit tests created in a test-driven development environment are typically created by the developer who is writing the code being tested. Therefore, the tests may share blind spots with the code.
- Writing and maintaining an excessive number of tests costs lots of time.
- TDD doesn't work sufficiently well when the suceess oor failure of the system needs full functionality tests(e.g. User interface, programs works with large database ....etc).

Waterfall model:

a sequential design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of conception, initiation, analysis, design, construction, testing,production/implementation and maintenance.

Pros:

- The waterfall model provides a structured approach. easily understandable and explainable phases and thus is easy to understand; it also provides easily identifiable milestones in the development process.
- Time spent early in the software production cycle can reduce costs at later stages. For example, a problem found in the early stages (such as requirements specification) is cheaper to fix than the same bug found later on in the process (by a factor of 50 to 200).

Cons:

- Once a stage(phase) ends, we can’t go back. So it is not so flexible in many situation, and no feedback between phases.
- Requires clear requirement and careful design, and customers can’t be involved in later phases.
- Only applicable in systems that don’t need lots of change during development.
- No working software is produced until late during the life cycle.

Case analysis:

CASE A:

waterfall model; Because the team member have high domain knowledge and program manager also has high expertise. Also, the project is comparatively small. So it is not that hard for the team to build clear requirement and applicable design, and they can use waterfall model to build understandable and explainable phases and make the development process clear and sequential. It can get rid of those tedious test-first process and save much time;

CASE B:

TDD: Because the team member is not that experienced compared to project A. So the project might be subject to change in different aspect. Thus, it’s not that applicable to use waterfall model. And the language used is more modularized(c++ & c#) so it’s easier to do unit testing. The block coverage is also high, indicating that the effect of test is good. Defects/KLOC using TDD compared to not using TDD is also very small (1/4.2), and the time increase of TDD is not that much(15%), all these gives the project B to be a good case to apply TDD.

(2)

TDD 在大多數時候都是一個好的開發方法，可以達到很好的模組化，也不會產生太多多餘的程式碼，debug 也方便得多。但是 TDD 會花費大量的時間，如果開發的時間並不充裕，那 TDD 可能就不是一個好點子。

而且，TDD 因為都是在寫 CODE 前就先針對單一功能寫測試，所以如果該系統有些功能要求整體性的測試才能判斷該功能的成效時就會遇到麻煩(例:需要使用者互動的功能、需要大量資料庫的功能、需要特定的網路組態的功能)。

如果你的 Project 需要很謹慎完善和不易變更的 Design，則 TDD 因為其性質的關係常常是由測試來驅動整個開發流程，有時會不那麼遵循原先的 Design 來進行，則在這種 project 中也可能造成問題。

另外，TDD 假設絕大部分的 CODE 都是由開發團隊從頭開始寫的，如果你的 Project 需要使用大量 Legacy code 的話，則可能會讓 TDD 的流程比較沒辦法那麼順利地執行。

最後，TDD 開發方法其實也是需要開發人員的訓練的，如果開發團隊的人沒有受過專業的 TDD 訓練的話，直接使用 TDD 來開發可能會造成許多難以預期的麻煩。

(3)

(a)

我們想要開發一個簡易的網路拍賣系統，
使用者可以註冊為會員，而會員分為買家與賣家(兩身分可以同時具備)
會員登入後可以購物、賣東西
訪客可以純粹瀏覽
每個人都可將商品放入購物車 但登入才能付費購買
訪客可以選擇註冊帳號

買家可以知道賣家出貨沒
賣家可以登入並在網站上陳列商品
可以呈現商品的屬性(類別、價格、顏色、尺寸、圖片)
登入可以知道商品販賣情況
賣家可以告知買家購買商品的出貨狀況

(b)

依據上述的功能，要實作的部分大致包括以下：
- 登入功能
- 註冊功能
- 選擇身分介面
- 瀏覽商品的介面
- 購物車系統
- 付款功能
- 上架功能
- 出貨狀況查詢
- 庫存查詢

(c)

整個開發流程，我們打算使用 Test Driven Development 來開發。
我們的需求基本上就如(b)所描述，而在進入實作階段前我們還必須建立清楚的 Architectural Design，了解每個 class 之間的 interface 以及之間的關係，方便每個工程師在實作時參考。
而因為使用 TDD 開發，我們每次在實作任何功能前都會先建立測試檔，實作完通過測試之後才把它整合進 Project 中。負責各個部份的 Programmer 在實作前都必須自己完成對應的測試檔才可以開始實作。

先建立選擇身分的測試檔: 若是買家，跳至商品頁面，若為賣家，跳至登入頁面，也可以選擇註冊帳號。
討論與畫出頁面雛型確認後開始實作
完成後建立測試案例驗收

先建立註冊頁面測試檔：輸入基本資料欄位，並建立檔案儲存會員資料。
討論與畫出頁面雛型確認後開始實作
完成後建立測試案例驗收

先建立瀏覽商品頁面測試檔：分類呈現賣家所上架的物品，可以點選加入購物車，可以進入購物車的選項。
討論與畫出頁面雛型確認後開始實作
完成後建立測試案例驗收

先建立購物車頁面測試檔：將買家所選商品做成清單，算金額，有結帳選項，若選擇結帳則跳往登入頁面。
討論與畫出頁面雛型確認後開始實作
完成後建立測試案例驗收

先建立登入頁面測試檔：給買家或賣家輸入帳號密碼，若是買家登入失敗則瀏覽商品頁面，登入成功就進入結帳區。若是賣家，登入失敗就重新登入，登入成功就進入賣家頁面。
討論與畫出頁面雛型確認後開始實作
完成後建立測試案例驗收

先建立結帳頁面測試檔：選擇取貨、繳費方式並結帳，可以選擇取消付款，若選擇取消則跳回瀏覽頁面，若選擇結帳則跳出購買明細。
討論與畫出頁面雛型確認後開始實作
完成後建立測試案例驗收

先建立購買明細測試檔：列出購買商品細節、取貨地點、繳費方式。
討論與畫出頁面雛型確認後開始實作
完成後建立測試案例驗收

先建立賣家頁面測試檔：可以選擇上架功能，並跳至上架頁面。另外會顯示所有商品狀況(庫存數量或已售出)，若已售出則可以選擇出貨狀況、並可以聯絡買家提醒取貨。
討論與畫出頁面雛型確認後開始實作
完成後建立測試案例驗收

先建立上架頁面測試檔：可以讓賣家在每個欄位填寫商品細節(類別、尺寸、顏色、價錢、物品名稱、備註)，並上傳商品圖片。
討論與畫出頁面雛型確認後開始實作
完成後建立測試案例驗收

　　最後，建立一個整合測試的測試檔，此整合測試檔必須測試到以上的所有功能，完整的模擬一個使用者使用此網站時的狀況。(Acceptance testcase)

用此整合測試整合驗收整體成果。

通過整合驗收測試之後我們就可以請 test team 來做額外的黑箱測試，如果通過測試則可以上架網站，並持續提供維護服務。