# 笑談軟體工程
# 敏捷開發法的逆襲

Teddy Chen

Ver. 0.1.5

這份文件是《笑談軟體工程：敏捷開發法的逆襲》的原稿。本書由悅知出版社於2012年出版，書中內容大多來自作者Teddy的部落格搞笑談軟工，經過Teddy改寫與出版社編輯後出書。

由於本書已經絕版，因此Teddy將書本原稿公開，提供給鄉民參考。

Teddy Chen

泰迪軟體

2020/04/20

獻給所有圍觀鄉民

這是一本因你們而誕生的小書

# 序言

12/09 14:00-15:52

Teddy 的「第一次」在 1989 年的某個秋天的夜晚獻給了 Pascal（有掛「渦輪」的那一種），如果沒記錯的話那是五專一年級的時候程式設計課程老師所出的作業，題目很簡單，由 1 加到 10，主要是要讓學生練習迴圈的用法。剛拿到作業的時候 Teddy 對於如何開始動手寫程式完全沒有任何頭緒，但是在學期結束之後 Teddy 卻喜歡上了 coding，一種非常適合宅男（當年還沒有這個名詞）的手指與手腕運動。沒想到這個「運動」最後成為 Teddy 賴以維生的技能。從考古的角度來看，這門課應該算是這本書最原始的種子吧。

之後的 22 年間，Teddy 陸陸續續寫了不少的程式也打了不少嘴砲。算起來應該參與過數十個大大小小的專案，有民間企業的專案、政府標案、產品開發、學術類的軟體研究計劃案。在參與這些計畫的過程中，有一個問題始終困擾著 Teddy，那就是：

**在台灣，要如何開發軟體才能夠讓公司賺到錢而員工的日子又可以過得爽？**

這個問題相信鄉民們都看得懂，但是為什麼要限定「在台灣」？答案很簡單，因為 Teddy 只在台灣開發過軟體…Orz。不過這不是重點，重點是，相信鄉民們都知道台灣的硬體代工產業非常

發達，就好像一個超級黑洞一樣把資金，人才與政府的關注力全吸引過去。雖然長久以來三不五時就會有不知道從那裡冒出來的人會突然大喊一聲 ~~旺旺~~「軟體很重要」這種試圖 ~~混淆視聽~~ 振奮人心的話，但 Teddy 相信台灣絕大多數的軟體工程師心中想說的卻是另外一個字…至於是那個字，限於尺度的關係 Teddy 就不打出來了。

在全球化的時代，為什麼要限定台灣？不是有人說「只要在台灣敢開車上路，到全世界各地開車都沒有問題」。同理可證：「只要在台灣軟體搞得起來，到全世界開發軟體都沒有問題」。是不是這個意思？就是這個意思。再不然 Teddy 只能搬出「愛台灣」這個理由了。

關於這個問題長久以來 Teddy 都沒找到合適的答案，就在 Teddy 快要放棄希望的時候，在 2008 年初 Teddy 接觸到一帖新的「藥方」，重新燃起了 Teddy 的求生意志。稟持著「吃好到相報」的精神，Teddy 自 2009 年 7 月之後陸續將這些藥方的重點分享於「搞笑談軟工（http://teddy-chen-tw.blogspot.com）」部落格中。本書內容為集結該部落格中與軟體開發相關的文章而成，方便有需要的鄉民們可直接購買回家自行服用。

書中所記載之藥方均經過 Teddy 依照台灣人的體質加以調整，服用後若出現胃痛、高血壓、呼吸困難、頭昏眼花、皮膚紅腫與想要開車飆到彰化等症狀，**此為正常現象，請安心持續服用**。

# 目 錄

# 致謝

12/22 12:56-14:00

首先感謝「搞笑談軟工」部落格的粉絲，雖然你們甚少留言，但看到以極緩慢速度增加的粉絲人數，卻是讓 Teddy 持續寫作的重要動力。

要特別感謝鄉民甲，在某次演講中該位鄉民問到：「Teddy 你有打算出書嗎？」當場 Teddy 當然是斷然的說 No（除非鄉民甲你要先認購一千本）。但可能是這位鄉民的「~~怨念~~ 念力太強」，讓原本忙到沒時間考慮這個問題的 Teddy，突然賺到幾個月的「無薪假」，也終於有時間了結多年前 Teddy 想成為「作家」的願望，順便把欠下的一些文字債還一還。

感謝「貴公司」讓 Teddy 有機會可以嘗試 Scrum 與許多敏捷實務作法。老夥伴 Kay、Albert、Brian、Jess、Tony、Lion、Swind，默默承受一切的 Eric，飯友，其他不方便透漏個資的秘密證人們，以及奴才們，沒有你們 Teddy 無法學會這麼多寶貴的經驗。

沒有兩位指導教授在知識上、精神上與物質上對於 Teddy 的協助，也不可能有本書的出現。還有實驗室的學弟妹們，你們天

真無邪的想法與作法，三不五時也成為 Teddy 寫作的靈感來源（實驗室聚餐別忘了找學長回去喔）。

最後感謝 Teddy 的母親，雖然她跟這本書沒有任何直接的關係但是沒有她也就沒有本書的作者，小弟在下我。

# 第一部 現在

# 1 好深的怨念

12/09 21:20-21:55

根據 Teddy 的研究，會翻閱此書甚至願意從自己淺淺的口袋掏錢出來忍痛買一本回家的鄉民們，對於軟體開發這檔事，心中都充滿了各式各樣的「怨念」。以下列舉常見者：

- 客戶把需求說的不清不楚，分析師順勢把客戶的需求寫的不清不楚。
- 客戶的需求一變再變，程式只好一改再改。
- 估計專案開發時程的方法有兩種，一種是筊杯，另一種是用喊價的。
- 沒有不延遲的專案，所以只好（樓下繼續…）。
- 工作超時（22K 就有一批新鮮的肝可用喔）。
- 原來 PM 的全名不是 Product Manager（產品經理），也不是 Project Manager（專案經理），而是 PostMan（郵差）。因為 PM 只會反射性的把客戶的信件轉給工程師，然後再把工程師的回答轉給客戶。
- 人家的 PM（產品經理）是真正的 PM，我們的 PM 是人家丟掉的我們把他撿起來。歐一 A、歐一 A（這句要用唱的）。
- 說好的測試工程師呢？（團隊人手永遠不足）。
- 我不會寫單元測試耶…請影中，請傻笑….（團隊技術能力不

夠或成員是不願意配合）。

- 軟體裡面的「小強（bugs）」已經氾濫到怎麼打也打不完的狀況。
- 團隊成員**分工但不合作**。
- 業務曰：這個案子**量很大**喔，只要改一點點地方就可以賣了。
- 老闆曰：案子先搶下來再說。
- 工程師曰：等一下，不用說了，這齣戲從頭到尾都沒有 ~~攻城屍~~ 工程師說話的份。

這個列表相信繼續寫個兩大頁絕對沒問題。**隨著怨念越來越強，鄉民們的軟體開發能力變得越來越弱。**有看過卡通「花田少年史」的鄉民們應該都知道，人往生的時候若是有未完成的願望，或是「怨念太深」，都是無法成佛的。所以台灣的軟體界很需要「靈異一路」這樣的人才來消除軟體從業人員的怨念。

請大聲跟著 Teddy 說出：去去怨念走。

***

友藏內心獨白：不用找什麼靈異一路，找 Teddy 就不就行了。

# 2 老闆，軟體不是這樣開發滴

12/10 10:18-11:08

12/09 22:10-22:21

原文發表於 http://teddy-chen-tw.blogspot.com/2010/06/blog-post_12.html.

N 年前台北捷運藍線 (板南線) 通車之後，有一天 Teddy 和第一份工作的老闆朱先生一起搭捷運（忘了去做什麼好事）。上車之後我們聊到捷運的方便性：

Teddy ： 捷運通車之後節省了很多通勤時間。

朱先生： 我認為節省時間還不是最重要的，因為捷運使得通勤時間變的**可預測**，會因此改變人們的行為模式。

PS：當時捷運木柵線還沒有遇到強烈月光而變身為「詐胡線」，否則朱先生就不會這麼說了...XD。

重點來了，速度快當然很重要，但是捷運的**可預測性**卻是改變人們生活習慣的主要因素。Teddy 第一份工作的公司剛好在台北捷運忠孝敦化站附近，沒有捷運之前，Teddy 搭公車至少要 40~50 分鐘才能到公司（還不包括等公車時間）。有時候趕時間搭計程車也不一定比較快。因為雖然平常覺的台北的計程車多的跟螞蟻一樣，但是上班時間經常等了很久也攔不到一輛空的計程車…Orz。所以為了不想上班或是約會遲到，就必須要提早出門以**容忍**這些**不可預測性**。有了

捷運之後，從甲地到乙地（假設都在捷運沿線）的通勤時間就變得比較可預測，因此無論是人們買房子，找工作，開店，約會見面，就經常會挑選捷運沿線。這就是所謂的「改變人們生活習慣」。

可預測性相當大的程度是立基於**可靠性**（**reliability**）之上。以路上交通而言高鐵速度夠快了吧，時速 300 公里，台北到高雄 90 分鐘，~~好貴~~ 好方便啊。如果今天有人發明時速 9000 公里的超級高鐵，從台北到高雄縮短到只要 3 分鐘 ，但是有 0.01% 的出事率（就是平均搭一萬次會有一次翻車的機率），鄉民們除非是想要賺保險費，不然應該沒人敢搭。

所以有一陣子「詐胡線」被罵到臭頭也是相同的原因。「詐胡線」通車之後的確變得很方便，但是它的**不可靠性**卻總是讓乘客們「心裡毛毛的」。搭完「詐胡線」沒出事內心的感受有點像是**不小心中了 200 塊發票一樣湧起一絲小小的安慰**。大眾運輸能蓋成這樣子，也算是「台北奇蹟」了。

***

但是，做軟體就不一樣喔，反正一般的軟體出包又不會「死人」，因此這種「先研究不傷身體，再講求藥效」的理論就不適用。所以，做軟體有自己一套的公式：

**（交貨速度 = 收錢速度）>>>>> 軟體品質**[1]

所以，你的老闆便可以理歪氣壯的大聲說出以下名言：

- 做專案的不需要自動化測試，做產品才要。
- 解 600 多個 bugs 要靠找「domain know-how」很強的人來幫忙。（Teddy 內心獨白：老闆，啊你嘛幫幫忙）
- 沒有測試人員沒關係，工程師自己測這樣就很好了。
- 這個軟體可以準備上市了，先找幾個客戶試用。（小工程師內心獨白：報告老闆，軟體連內部測試都還沒開始耶）
- 軟體現在不能上市？！我已經等了 N 年了，我不想再等另一個 N 年。
- 我最多給你三天，在三天之內把這個 bug/功能給我改/做好。

台灣的硬體代工產業實在是太強了，強到把所有的養分都吸光以至於軟體變得很弱。**雖然口頭上喊著軟體很重要，但這些大老闆與高階主管們絕大多數還是以代工硬體的思維來看軟體開發**。

老闆：我隨便派一個工程師 3～4 個月就可以「獨立」設計一塊電路板，你們六~七個人做個「小」軟體搞了兩年還做不出來。

[1] (交貨速度 等於 收錢速度) 大大大大大於 軟體品質

大老闆與好不容易熬出頭的各級主管們，Teddy 知道你們每天都有開不完的會，加不完的班，都很忙，忙到沒時間去稍微了解軟體要怎麼開發。但請用你們聰明的腦袋稍微回想一下，設計一塊電路版的背後，有多少的協力廠商已經提供好解決方案了，並且這些廠商還會主動地跟公司推銷這些既有的解決方案。硬體分工很細，遇到問題，背後各有不同的協力廠商會出面協助解決。有很多在硬體公司開發軟體的工程師，名義上號稱是設計與開發軟體，但實際上可能是只有負責把公司從其他廠商（通常是國外廠商）買來的軟體或是韌體修修補補，調整成自己公司所需要的，比較少真正有自己的軟體產品。就算是有心想要做自己的軟體產品，老闆卻直覺的認為軟體開發沒什麼學問，什麼都自己來就可以了。

老闆：什麼，要花錢買軟體元件（software components）。15 萬，這麼貴，自己寫比較省。要導入 Scrum，10 萬，這麼貴，自己試就好了 。要導入自動化測試，8 萬，這麼貴，人工測一測就好了。要.....「賣夠共阿啦」，要什麼一律免談。

Teddy 的第一份工作曾經開發過「連鎖洗衣店門市進銷存系統」。大家一聽到「洗衣店」一定覺的這是一個很 LO 的產業，除了當兵的時候集體送洗過衣服的超噁爛經驗以外，Teddy 從來沒有過到乾洗店送洗衣服的經驗。當時 Teddy 的直覺反應也是：「洗衣店，欸，沒什麼了不起，沒什麼了不起」。

等到 Teddy 接觸到洗衣店老闆（是一位女士）之後，嚇了一大跳，人家可是「哈佛 MBA」。在訪談需求的過程中，更是發現老闆的經營策略與台灣傳統的洗衣店大大不同，不論從「門市人員聘用」,「教育訓練」，到「定價策略」，背後都有一套科學的作法，讓 Teddy 大開眼界（真的是人外有人，天外有天。人家洗衣店老闆很低調也很客氣，並沒有一天到晚把自己是哈佛 MBA 放在嘴上）。

人客啊，人家經營洗衣店這種「傳統產業」都這麼用心了，開發軟體難道不應該也「科學化」一點？清朝末年有一段時期，中國人覺的火車會破壞風水，極力反對蓋鐵路，現在中國土地上高鐵滿地跑（雖然最近出包的事件也不少）。老闆啊，軟體工程不是洪水猛獸，也不需要聽到軟體工程這幾個字就好聽到「髒話」一樣的反應。經營管理的書老闆們看的夠多了，抽空看看軟體開發的「閒書」吧（你現在看得這一本就是）。

***

友藏內心獨白：等一下又要去天瓏書局逛逛。

# 3 600 百多個 bugs 要怎麼修？

5/31 22:36~23:56

12/10 11:09-11:41

原文發表於 http://teddy-chen-tw.blogspot.com/2010/05/600-bugs.html.

某天晚上 Teddy 接到老朋友 X 小姐打來的電話，問了幾個她在新公司所遇到的問題。X 小姐是位非常稱職的 PM（專案經理），最近到了國內獲得 CMMI [1]認證的某大**資服業**（資訊服務業，就是到處去~~亂搶~~ 接案子的公司啦）的某部門負責維護專案。她們部門負責維護某軟體（在此稱之為 F 軟體）。X 小姐的公司另外有一個產品開發團隊負責開發 F 軟體，完成之後就用這個 F 軟體去搶專案，搶到之後由於每個專案都需要做**客製化**，因此每接一個專案就產生一份 **F'軟體**（F 軟體的客製化版本，主要功能和原本的 F 軟體一樣但是會因應客戶的特別需求在一些小地方加以調整或是為該客戶增加某些專屬的功能）。

專案結束之後，為這個專案所客製化修改的功能並不會回饋到原本的 F 軟體中（錢都 ~~騙~~ 到手了，誰還管那麼多）。X 小姐所在的維護部門，養了幾個程式設計師（programmers），每一個人負責十來個客製化專案的**維護工作**（就是修 bugs 啦）。但是這些負責維護的程式設計師其實並不全然了解整個系統的架構與運作原理。F 1.0 版是在

十年前用微軟 ASP 技術（Active Server Pages，一種用來開發網頁的技術）所開發出來的，目前有幾十個客戶，**也就是說有幾十個不同的 F 1.0 版本**。目前產品開發團隊即將完成以微軟新的.NET 技術所開發的 F 2.0 版。

十年前開發的 F 1.0 版與其衍生的幾十個版本以及新的 F 2.0 版都沒有任何的**自動化測試案例**。目前以 F 1.0 版為基礎的客戶所回報的 bugs 一共有 **600 多個尚未解決**。客戶對於 bugs 修復的進度緩慢感到非常不滿意，甚至直接打電話跟公司高層抱怨。

公司高層： X 小姐，妳評估一下需要多少的人，在多少的時間內，可以把全部的 bugs 解決？

X 小姐： ???（我又不是算命的...這句是 Teddy 幫她回答的）

***

X 小姐在電話中問 Teddy 幾個問題：

- 這 600 多個 bugs 要怎麼解決？（Teddy 內心獨白：我也不是算命的啊）
- F 1.0 版所衍生的幾十個專案，日後要如何維護？
- F 2.0 版即將完成，如果依照公司之前的模式，日後一定又會產生 N 個 F 2.0 版的衍生物，又會遇到維護的問題。怎麼辦？

人客啊，你說這 600 多個 bugs 要怎麼解？Teddy 怎麼可能會知道，只能依照軟體工程的常理來「隔空抓藥」：

1 先把這 600 多個 bugs 分類，看看能否找出 bugs 的 patterns（簡單說就是先將眾多的 bugs 分類）。例如，memory leak bugs、database transaction control bugs、UI validation bugs、business logic bugs、concurrent control bugs、exception handling 不良所產生的 bugs 等等。

2 從這幾十個 F 1.0 的專案中找出功能最多、目前 bugs 最少、或是客戶最急的專案作為基礎，然後實施 code review。對照步驟 1 所整理出的 bugs 分類，看看能否從 code review 的過程中找出一些發生錯誤的蛛絲馬跡。

3 補寫 test cases。

3.1 針對每一個準備修復的 bug，寫出一個自動化測試案例來重現這個 bug。

3.2 或是對於不了解的功能，以寫 test cases 的方式來驗證與理解這些功能。

3.3 針對準備 refactoring 的功能寫自動化測試。由於古早以 ASP 技術所開發的系統程式碼和 UI（user interfaces，使用者介面）大概都是混在一起，要寫自動化單元測試可能不是那麼容易。所以要依據程式結

構實際狀況來研究一下這種自動化測試要怎麼寫。

4 導入持續整合。

5 如果一切順利，則看看有沒有可能用同樣的方法來修正其他專案的 bugs。

X 小姐： 可是要求程式設計師寫自動化測試不太可能耶。其他專案的 PM（專案經理）會認為**寫程式的時間都沒有了哪有時間寫測試**。程式設計師也會有類似的想法。

Teddy 當下心裡在想：

- 這家公司不是通過 CMMI 認證嗎？！（說好的品質提昇呢？丟筆…）
- 自動化單元測試不會寫，失敗。不願意寫，失敗中的失敗。
- 一個獲得 CMMI 認證的公司，要他們寫點自動化測試居然是那麼的困難，可見 CMMI 真的是很博大精深。~~食神~~ CMMI，我真是猜不透你啊。
- 現在很多大學資工系的學生修程式設計的課都要寫自動化測試了，號稱專業的資服業者居然沒寫自動化測試案例。
- 台灣的軟體開發團隊對於軟體工程的知識與實踐上的「貧富差距」真的太大了啦（XXXX 下台，政黨輪替啦…XD）。
- 地鼠 (bugs) 是打不完滴，只有把地鼠給徹底剷除，剷完再除，牠們才不會一直不停地冒出來。

- 也許應該考慮採用 Scrum 或是 Kanban 來改善軟體開發與維護流程。
- 把公司的電話線和網路線剪掉，手機關機，這樣就不會接到客戶的抱怨。
- 這個故事再次驗證了 Dave Thomas 所說的 **All programming is maintenance programming** [2]這句話。

Teddy 和 X 小姐通了一個半小時的電話，講了一堆有的沒的，最後還是請她回去請示一下她的老闆，確認一下她老闆是屬於「袁世凱」還是「孫中山」。如果是前者，那就千萬不要想太多，做一天算一天，有空就更新一下 104 上面的履歷表，隨時準備開啟。如果是後者，則可以考慮找人幫忙導入 Scrum [3]或是一些軟體開發的實務作法。

***

友藏內心獨白：顧問公司到底都教了人家什麼東西呢？

***

## 備註

[1]. CMMI，Capability Maturity Model Integration（能力成熟度整合模型)是由美國卡內基美隆大學軟體工程學院（SEI）所制定一

套用以評估與改善軟體開發流程之特定能力成熟度的方法。以白話文解釋，CMMI 將受評鑑單位（可以是一個部門或是整個公司）的「軟體開發流程成熟度」分成五級，級數越高表示「越厲害」。要通過不同等級的認證，各有不同的要求（和玩 game 差不多，關卡越高魔王越難打）。那麼一個公司怎麼知道自己是那一級的玩家呢？很簡單，需要付一筆錢找 SEI 認證過的主任評鑑員（Lead Appraiser）來幫公司評鑑。基本上評鑑就是一種考試，有考試就一定有補習班。這幾年因為政府編列預算補助企業取得 CMMI 認證，所以也有許多 CMMI 補習班（輔導顧問公司）應運而生。講到考試咱們台灣人說第二地球上應該沒騎他人敢說第一，所以這幾年獲得 CMMI 認證的廠商很多，在政府推廣下成效卓著。

[2]. http://www.artima.com/intv/dry.html.

[3]. Scrum 是一種這幾年很流行的敏捷方法，其內容可參考第二部 Scrum。

# 4 這不是髒話

6/5 23:45~ 6/5 00:28

12/19 10:55-10:59

原文發表於 http://teddy-chen-tw.blogspot.com/2010/06/blog-post.html.

有點晚了，本來不該寫這一篇，以免睡不著，先講短一點的版本好了。第 23 頁「600 百多個 bugs 要怎麼修？」的主角 X 小姐昨天在 MSN 中告訴 Teddy：

[13:14:41] 我今天才一開口說要從軟體工程著手，應該要寫 testing code 就被打回票應說

[13:14:50] 客製化和產品是不同的

[13:15:04] 客製化無法從軟體工程做起，應該是要怎樣提升程式品質

[13:15:05] @@

[13:15:14] 所以我就沒繼續說了

[13:15:46] 他們覺得我過去做產品與作客製化專案是不同的

[13:16:05] 產品才需要不斷的作 automatic test 確保品質

[13:16:14] 無言

Teddy 看了差點沒被氣到吐血（還是應該笑到肚子痛？），真是…。

奇怪，**有些人只要一聽到「軟體工程」這幾個字，就好像聽到「三字經」（髒話）一樣**，立刻從椅子上跳起來，好像想找人打架似的。

「客製化無法從軟體工程做起，應該是要怎樣提升程式品質。他們覺得我過去做產品與作客製化專案是不同的，產品才需要不斷的作自動化單元測試確保品質」。

這幾句話比較像是火星人講的話吧，這是什麼邏輯，身為地球人的 Teddy 真的無法理解（另一種可能是 Teddy 才是火星人）。軟體工程不就是要「改善品質」、「提昇效率」嗎？做軟體的，不管是「產品」還是「專案」，最後給客戶的還不都是「軟體」。做產品的軟體才需要自動化測試，做專案的不用？這.....？？？？**所以，這是說專案的客戶拿到爛軟體算他「衰小...朋友」（活該倒楣）**，誰叫這是一個「專案」，不是「產品」。

X 小姐的公司還通過 CMMI Level N 認證勒，哇哩勒。靠...左邊走比較安全。Teddy 猜測該公司通過 CMMI 認證的應該是其他部門而不是 X 小姐所屬的部門，不過這不是重點。重點是，一個通過 CMMI Level N 的公司，居然有高層主管認為「客製化無法從軟體工程做起，應該是要怎樣提升程式品質」（現在是在搞「毅中各表」嗎？前後兩句加起來 Teddy 真的看不懂，中文程度太差）。光看第一句就好：「客

製化無法從軟體工程做起」，那敢問要從何做起？**難不成從「偷、搶、拐、騙 + 綁標 + 送水果禮盒 + 金錢豹」做起？**

基於道義更怕被告所以無法說出該公司名稱，不過這家公司 Teddy 以前也有接觸，從 10 幾年前作軟體的品質就有待加強，沒想到過了那麼久一直沒顯著的長進。不過，X 小姐說：「人家公司都有賺錢啊」。之前 Teddy 公司第二任總經理曾經告訴 Teddy 一句話：**「公司不賺錢是一種罪惡」**。的確，Teddy 完全認同（薪水發不出來的時候真的很痛苦）。不過，現在時代進步，應該要再加上，**「公司賺錢也要賺的讓人尊敬」**這個條件才行。有公司靠污染環境賺錢、靠壓榨勞工賺錢、靠炒地皮賺錢、靠官商勾結賺錢、靠違法亂紀賺錢、靠逃漏稅賺錢、靠發國難財賺錢、靠跳樓.....大拍賣賺錢。還有 X 小姐的公司，難道要靠客戶不滿意賺錢？XX 公司，真有你的。

***

友藏內心獨白：XX 公司會變成新人訓練中心也不是沒道理的啊。

# 5 改行寫網路小說算了(1)

Feb. 13 21:39~22:59

12/19 13:50-13:58

原文發表於 http://teddy-chen-tw.blogspot.com/2011/02/blog-post.html.

有一家歐商公司在製造「~~貴夫人~~ 貴公主自動泡咖啡機」，就是那種直接把咖啡豆放到機器裡面，加水之後按一個按鈕就可以煮出一杯好喝的咖啡。該機器在歐美十分暢銷，這家公司想把機器賣到亞洲以拓展市場。有一天業務找到一個台灣的代理商，代理商告訴他：現在在台灣喝咖啡太方便了，而且同類型的泡咖啡機市面上也很多，你這台機器可能會不好賣。 但是，現代人都很重視養生，如果你們公司可以讓這台機器也具備「自動煮豆漿」這個功能，那就可以同時滿足喝咖啡與喝豆漿的客戶。如果能做的這樣，那我就可以先下一萬台的訂單。

業務想，這可是筆大訂單，反正都是「豆子」，只要把「咖啡豆」換成「黃豆」，再「稍微調整一下線路」，那我們的產品「應該」可以同時具備這兩種功能。於是業務發了封 email 給開發人員：

親愛的 ~~攻城屍~~ 開發人員：

有一個大客戶詢問我們的「貴公主自動泡咖啡機」是否支援「煮豆漿」的功能，請幫忙確認是否將咖啡豆換成黃豆之後便可煮出豆漿。如果不行，請評估需要多久的時間可以完成此功能。這是一筆大訂單，客戶急著要，請儘速回覆。

謝謝。

業務 和聲 敬上

好死不死你正好是這個「親愛的開發人員」，收到這樣的 email 要怎麼回？以下是一個失敗的範本：

親愛的業務大大：

開發人員目前都沒空，無法幫你測試我們的「貴公主自動泡咖啡機」是否可以用來「煮豆漿」。麻煩請找測試部門的人幫你做這個測試。

開發人員 白木林 敬上

業務看到回信之後很不爽，怎麼可以把即將到手的大單子往外推（迷之音：這個量很大喔）。於是業務就寫了一封信給老闆：

皇上吉祥：

我不明白「白木林」為什麼要拒絕我提出的要求，公司的同事不是應該要互相幫忙嗎，怎麼可以回答「開發人員目前都沒空，無法幫你測試」這種話呢？眼看有一筆大生意即將談成，「白木林」的作法卻是把生意往外推。我們公司到底還有沒有想要推廣「貴公主自動泡咖啡機」到亞洲？請給予建議。

謝謝。

業務 和聲 敬上

老闆看了信之後就寫了封信給行銷主管（其實只有一句話）：

請儘速評估煮豆漿這個功能的市場需求。

以下是行銷人員的回信：

啟奏聖上：

「煮豆漿」這個功能在遠東地區，尤其是中國、台灣、日本、與韓國都有極大的市場，此項產品開發成功之後將大大增加公司的營業額。我會立即協調開發人員，將這個功能加入的我們的產品之中。

吾皇萬歲，萬歲，萬萬歲。

~~東廠廠公~~ 行銷主管 PLoP 跪拜

接下來的劇情請鄉民們自己發揮....

***

友藏內心獨白：皇上聖明，如果可以再加上「泡茶」這個功能豈不是更好？

# 6 改行寫網路小說算了(2)

March 15 22:31~23:15

12/19 13:59-14:03

原文發表於 http://teddy-chen-tw.blogspot.com/2011/03/2.html.

## 故事一

在某次產品**即將發表前**的會議當中：

聖上：咱們的**「雲端殺豬系統」**客戶試用後反應如何啊？

產品經理：啟奏萬歲，有台灣的客戶反應，既然能夠在「雲端殺豬」，何不利用已經殺好的豬，順便加上**「雲端燉東坡肉」**與**「雲端煮滷肉飯」**功能，這樣一定大賣的啦。

行銷經理：皇上聖明，根據微臣的調查，「東坡肉」與「滷肉飯」不只在台灣，在中國更是有很大的市場，的確是不可忽視的兩項功能。

聖上：看起來這兩項功能真的是不可少喔....~~工部侍郎~~ 九品工程師，這兩個功能要多久才能加到「雲端殺豬系統」中？

九品工程師：如果「東坡肉」要燉的嫩，「滷肉飯」煮的香，依奴才看要最少也要六個月的時間。

聖上：大膽，六個月後那些番邦們早就 ~~打過來了~~ 推出類似產品了...朕限你二個月之內完成，否則依軍法處置。

九品工程師：微臣領旨，吾皇萬歲，萬歲，萬萬歲。

聖上：退朝。

眾臣：吾皇萬歲，萬歲，萬萬歲歲歲...

***

九品工程師：喂，「醫靈肆人力銀行」嗎，請幫我找兩個會煮「東坡肉」與「滷肉飯」的廚師，這是急件，請在一周內找到合適的人才。

醫靈肆：請問該工作的職稱是？

九品工程師：**資深雲端廚師**。

***

## 故事二

在某次產品**即將發表前**的會議當中：

聖上：咱們的**「超級柴油汽車」**客戶試用後反應如何啊？

產品經理：啟奏萬歲，咱們的「超級柴油汽車」扭力大，爬坡強，客戶反應非常好。但是現在因為「節能減碳」的意識高漲，有很多客戶反應，如果我們的「超級柴油汽車」可以具備**「純電力發動」**的功能的話，這樣一定 ~~打騙~~ 打遍天下無敵手的啦。

行銷經理：皇上聖明，根據微臣的調查，**「純電動車」**已經是世界趨勢，在世界各國都有很大的市場，的確是不可忽視的一項功能。

聖上：看起來這一項功能真的是不可少喔....九品工程師，把咱們的**「超級柴油汽車加上純電力發動」**這個功能要多久才時間啊？

九品工程師：啟奏聖上，此事微臣在七日之內便可辦成。

聖上：啊，七天？！古有曹植七步成詩，今有卿家七日成車。好，若愛卿能如期完成此任務朕便升你為工部尚書。

九品工程師：微臣領旨，吾皇萬歲，萬歲，萬萬歲。

聖上：退朝。

眾臣：吾皇萬歲，萬歲，萬萬歲歲歲...

***

九品工程師：喂，「瘋甜汽車」嗎？我要買一台 XXX ...有現貨嗎？

瘋甜汽車：很抱歉，因為日本工廠遭遇地震與海嘯，暫時停工，所以沒有現貨。最快一批貨至少也要三個月以後才會到港。

九品工程師：頑皮，頑皮，推一推....XD。

九品工程師：喂，「醫靈肆」人力銀行嗎，我要應徵 YY 公司的**「資深雲端廚師」**工作。.

***

友藏內心獨白：請參考第 18 頁那一篇。

# 7 改行寫網路小說算了(3)

May 16 20:16~20:56

12/19 14:04-14:08

原文發表於 http://teddy-chen-tw.blogspot.com/2011/05/3.html.

話說「卡歪美食公司」準備推出了一款新的巧克力蛋糕，取名為「甜可麗」。在某次公司會議中...

老闆：這個「甜可麗」是公司近年來最重要的產品，為了確保客戶會喜歡，在推出之前要先找幾個客戶來 ~~試用~~ 試吃一下。

行銷主管：皇上聖明，奴才立刻去辦。

***

兩個月後，行銷主管寫了封 email：

各位同仁,

以下是客戶們試吃「甜可麗」之後的反應：

- 好想來杯咖啡。

- 好想吃鹹的。
- 吃完蛋糕之後應該要刷牙。
- 我想你們的「甜可麗」不適合我們，因為「甜可麗」的原料沒有採用台灣本土生產的小麥。

吃「甜可麗」的時候很容易掉屑屑，不小心會弄髒衣服。

***

兩個月零一天後...

老闆：上次開會讓你們找客戶試吃「甜可麗」，結果怎麼樣啊？

行銷主管：啟奏聖上，經過我們**大規模**的調查後發現：

- 吃完「甜可麗」之後有 49.999% 的人會想要來杯咖啡，所以「甜可麗」要 ~~增加咖啡的功~~能.....嗯嗯...應該跟咖啡一起「搭售」，可以提高營業額。
- 有 68.888% 的人在吃完「甜可麗」之後會想吃鹹的，所以如果我們能夠增加產品內容，順便賣茶葉蛋的話，應該會不錯...**這個量很大喔**。
- 26.66% 的人吃完「甜可麗」之後會立刻刷牙，奴才有管道可以批到一批便宜的牙膏，牙刷來賣。或是公司可以在現場

提供「牙刷出租服務」，讓顧客吃完之後可以立即享受五星級的刷牙服務。好吃又健康，讚的啦！

- 有 ~~0.000001%~~ 顧客建議我們採用台灣本土生產的小麥來製作「甜可麗」，如可方可證明我們是一家「愛台灣」的本土企業。因此為了提昇「甜可麗」的銷售量與公司的企業形象，我們應該成立「卡歪精緻農業公司」，立即著手將台灣休耕的農地拿來種植小麥。
- 最後一點，大家都知道吃蛋糕的時候一不小心很容易弄髒衣服，為了提供客戶一個 total solution，我們可以和乾洗業者合作，凡是因為吃「甜可麗」弄髒衣服而需要送洗者，都可以免費獲得一張打五折的乾洗折價券。

老闆：嗯，聽起來很不錯。愛卿所奏照准。這件事行銷主管辦得不錯，打賞黃馬褂一件，另外加封太子太保~~開~~銜，上書房行走。

行銷主管：謝主隆恩。

老闆：退朝。

***

九品芝麻官：調查了老半天，到底「甜可麗」好不好吃還是沒人知道？

零壹鄉民：跟牠認真，你就輸了。

***

友藏內心獨白：國王的新衣還真的有「生物」可以看的見耶，佩服，佩服。

# 第二部 Scrum

# 8 Scrum 是一種制度

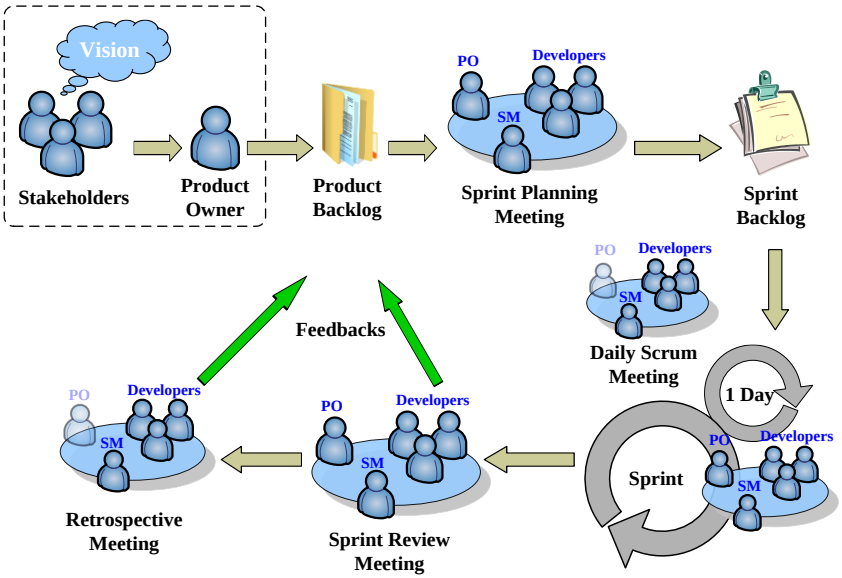
01/17 21:34~23:40

12/18 22:24-22:31

原文發表於 http://teddy-chen-tw.blogspot.com/2010/01/scrum.html.

依稀記得大概在兩年半前看到 Scrum 這個字眼，當時直覺的反應是：「又來了一個 agile method」。市面上 agile methods 這麼多，要怎麼學？... 阿就... ~~亂學~~ 挑選第一品牌 XP 就是了。抱著這樣的精神，Teddy 也就沒特別花時間去留意這個連英文單字都不知道是什麼意思的 Scrum。

幾個月後，也忘了什麼原因，實驗室的學弟被指派報告 Scrum。一如往常，聽完之後獲得**槓龜貼紙**一 枚。當時只記得聽到一些奇怪的名詞，什麼「Sprint、Backlog、Daily Scrum...」為什麼不叫做 iteration、requirement list、stand-up meeting 就好，還要自創武功名稱。這些老外是怕咱們英文單字不夠背嗎？當時 Teddy 最沒搞懂的，就是下面這張圖中央的這兩個圈圈。慧根不夠，一個 30 天的大圈圈，背著一個 24 小時的小圈圈，光看圖很難了解它的明白(軟體界的老背少？)。




後來，實驗室學弟 ~~被迫~~ 自願在某個專案上採用 Scrum，結果又累積了另一張槓龜貼紙。當時 Teddy 正忙著撰寫博速論文，也沒心思幫忙。不過主要的原因還是 Teddy 自己根本不懂 Scrum，插不上手。

無奈老天爺還是不放過 Teddy，就在 Scrum 連兩槓之後沒過多久，無意間看到了 *Scrum and XP from the Trenches: How we do Scrum* [1]這本免費的電子書。這本書只有 120 幾頁，用平鋪直敘的方式搭配大量具體的範例來分享作者自己實施 Scrum 的經驗。這本書 Teddy 反覆看了好幾次，再搭配 *Agile Software Development with Scrum* 這本書以及之前看過的一堆有的沒的敏捷方法書籍與論文，採用雞尾酒療

法的精神卯起來配一大口白開水一次服用，藥效很強。此時忽然有種被黃袍加身的錯覺，也想找隻白老鼠來試一下（PS：~~哥哥~~ 叔叔有練過，小朋友不要學喔）。

後來果真找到一隻小白老鼠，到現在一轉眼過了一年半。值得安慰的是，當年這隻白老鼠到現在還健康的活著 （至少 Teddy 認為牠還滿健康的啦）。這一年半的學習歷程，Teddy 有一點點感想整理如下：

**第一次接觸：**經驗不是很好，因為槓龜兩次。Teddy 覺的這不過是另一個 agile method 而已。

**第一次實施：**Scrum 是一個很簡單的框架（framework），規範了**角色** (Product Owner、Scrum Master、Team)，**活動** (Sprint Planning Meeting、Daily Scrum、Sprint、Sprint Review Meeting、Sprint Retrospective Meeting、Product Backlog Refinement Meeting)，**產出物** (Product Backlog、Sprint Backlog、Task Board、Burndown Chart、Potential Shippable Software)。但是 Scrum 並沒有特別規範 practices。從這個角度來看，要了解 Scrum 其實很簡單，但是要實際運用 Scrum 來開發軟體，還是需要紮實的軟體工程基礎（簡單講就是 agile practices, 像是 continuous integration、unit testing、refactoring、test-driven development、pair-programming、review、design patterns... 這一些都要真的懂而且要採用）。

**實施幾個月之後：**Scrum 很累... 要讓這些 ~~刁民~~ 團隊成員乖乖的按照書上說的方法來進行 Scrum 還真不容易。

**上完 Certified ScrumMaster 課程：**Scrum 框架本身元素很少，所以很簡單。但是 Scrum 的精神卻不容易達成。例如，Scrum 強調「持續改善」，這一點就很難。還有 Scrum 要塑造 self-managed team（講成白話文就是自我管理的團隊，就是一個老闆都不用管，就會把事情做的又快又好的團隊）也是很難。

**看了 Succeeding with Agile：**Software Development Using Scrum 這本書：這本書第五頁提到 Scrum 很難的六個原因，包含 (1) Successful change is not entirely top-down or bottom-up; (2) The end state is unpredictable; (3) Scrum is pervasive; (4) Scrum is dramatically different; (5) Change is coming more quickly than over before; (6) Best practices are dangerous。內容是什麼意思有興趣的鄉民們就去賣本書來看，Teddy 想講的是第三點「Scrum is pervasive」，也就是說，公司要導入 Scrum，不是只有某一個團隊，或是整個開發部門的事情而已，而是整個公司都要「換腦袋」。業務部門要知道在 Scrum 框架下如何跟客戶溝通，人力資源部門要改變傳統以個人來評量績效的方法（Scrum 講的都是團隊... 有點社會主義的感覺）。QA 部門不能只從規格來驗收軟體，還要從客戶是否真正需要來考慮。結論就是，這

根本是 **革命** 嘛。

各位 agile 先聖先賢們，咱們只是想好好開發個軟體，有那麼嚴重嘛！

總之，Teddy 現在的感覺是，Scrum 其實是一種「制度」，在這個制度底下，要做的事情其實很多。好比民主制度，主流價值觀都認為民主制度很好，但是真正實踐起來卻很難，而且很花時間。以咱們 台灣為例，民主國家形式上要有的元素都有了，但是實質面卻還是有很多進步的空間。買票、賄選、立法院打架、民粹、立法效率差、抹黑...等等一狗票的問題都還存在，但是不繼續走下去好像也不行（不然怎麼有這麼精彩的「毅中各表」世紀大辯論可看…XD）。實施 Scrum 的過程，有點像是從專制或是帝制換到民主共和制度一樣，以程式語言的術語來講，這就是一種「典範轉移」（paradigm shift）。會寫程式的鄉民們回想一下，從程序導向的思考模式要換到物件導向的思考模式花了你多少時間？所以實施 Scrum 要有長期抗戰的準備。

但是，先決條件是，你，你的團隊，還有最重要的是，你老闆要相信民主制度比較好。如果你的老闆是**袁世凱**，那實施 Scrum 是「祝恐怖（很恐怖）」滴。此時還是大喊一聲「皇上聖明」比較實際一點。

***

友藏內心獨白：誰說民國沒有皇帝？

***

## 備註

[1] Henrik Kniberg 所撰寫的 *Scrum and XP from the Trenches: How we do Scrum* 本書可從網路上下載免費的 pdf 格式電子書，也有販賣紙本書籍。

# 9 就是這個光：Scrum + Lean + XP

03/09 22:18~ 03/10 00:10

12/18 22:08-22:16

原文發表於 http://teddy-chen-tw.blogspot.com/2010/03/blog-post.html.

Teddy 凡致力軟體開發十餘年來，無不在追求一個「正確答案」：軟體到底要怎麼開發才會順？用 RUP (Rational Unified Process) 太沈重，用 XP (eXtreme Programming)又太自由。話說 10 幾年前 RUP、UML 正當流行的時候，當時 Teddy 有一個 8 人/年（8 個人做了一年）的專案，就是試著採用 RUP 的方式來進行。Teddy 和另外一位同事每天熬夜寫出十分詳細的分析設計文件，隔天立刻拿給程式設計師實做。由於設計文件寫得太詳細，所以實做上沒有遇到什麼大問題。大部分的工作只是寫寫 SQL，設計精美的 UI，以及實做控制流程。

這一年內公司停掉接專案的機會，全心投入所有人力開發這個系統。完成後產品還沒大賣，但資金卻已經花得差不多了，因此要找人增資。Teddy 跟著老闆東找西找，後來找到某軟體公司願意投資我們一筆資金。事後 Teddy 問這家公司負責評估的人，為什麼要投資我們。他說：**「我沒有看過台灣的軟體公司把開發文件寫得這麼詳細的」**。（迷之音：開發什麼軟體，早知道寫文件可以騙錢那卯起來寫文件不就得了！）

這...原來不是對方認為我們軟體做的好，而是覺的我們開發文件寫得好，所以軟體功力應該不錯，值得投資。當時 Teddy 有一句話一直不敢說出口：**「其實我們的文件已經和程式碼不同步了！」**軟體做好後，陸陸續續依據客戶要求還是有一些改變。而之後的功能修改根本不可能還有那美國時間回頭去更新文件。

不知道鄉民們是否和 Teddy 一樣有著相同的疑問：大部分 OOAD（Object-Oriented Analysis and Design，物件導向分析與設計）的書不都是教我們要寫很多不同的文件，而且越詳細越好。此外，還要保持文件與程式碼的一致性。實務上，台灣軟體公司大多是中小企業，客戶願意付出的費用通 常不足以支付「完美開發團隊」的費用。所以，怎麼辦？

這個問題當年困惑了 Teddy 許久。後來，XP 變得很熱門，也引起了 Teddy 的注意。在一次偶然的機會中 Teddy 讀了 Jack W. Reeves 所寫的 ***What is Software Design?*** [1] 這篇文章，當場茅塞頓開。簡單的說，傳統上軟體設計（廣義的說所有的工程設計）產出物就是文件 (現在的一般作法就是寫 UML 文件)，而「寫程式」這件事算是低層次的「施工」，或是「生產」。基於此種想法，自然會要求軟體開發者要把「設計文件寫得很清楚」，這樣程式設計師就可以**「按圖施工，保證成功」**。更進一步，程式設計師變成生產線的作業員，景氣好的

時候可以多找一些人來加速生產，景氣不好就放無薪假。找程式設計師也變得很簡單，只要高中程度稍加訓練即可。更極端一點，分析文件寫好直接就可以產生程式碼，連程式設計師都不用了。傑克，真是太神奇了。

而 Reeves 在該文章說明，**code (程式碼) 本身才是「軟體設計」的產出物**， 而不是文件。至於 compile、link 這些活動才是軟體生產活動，成本趨近於零 （IDE 上面按個按鈕就 OK!）。**從這個角度來看，coding 或是 programming 便成為一種「設計活動」，而非「生產活動」。**既然 code 是設計文件，而 coding 是設計活動，那麼傳統的文件重要性自然就大大降低了，而應該把焦點放在「如何寫好程式（等於如何做好設計）」上面。從 XP 的眾多活動中可以看出來這種 **code-centric design (以程式碼為中心的設計)** 的味道，每個 programmers 其實都是設計師。

剛接觸 XP 時，Teddy 簡直把 Kent Beck 當神一樣來崇拜，終於看到一種比較像是給人使用的軟體開發方法了。但是，實務上，XP 的採行還是遇到很多阻礙。一方面有些人覺的 XP 的主張太過極端（軟體領域的邪教？！），有些人則是誤解了 XP 的精神，加上大家對於改變總是怕怕的，因此在台灣實際實施的團隊應該不多。舉一個 XP 所提倡的活動 pair programming 來講，**老闆一聽到兩個人一起寫程式，馬上想到成本增加一倍，不被罵死才怪。**就算你跟老闆解釋：「這樣

可以隨時做 code review，提高品質」，老闆可能會回你一句：「code review 是蝦米碗糕？」。另外，程式設計師也不見得願意和別人一起寫程式，因為這樣會剝奪他 MSN 與玩 FB 的時間....%!@#!~@

一方面，Teddy 覺的 XP 的很多實務作法（practices）很棒，例如 testing、continuous integration、 pair programming、refactoring 等。另一方面，可能是 Teddy 沒把 XP 全部看懂，總覺的 XP 在 planning 這方面似乎稍微抽象了一點，真的要來規劃一整個專案時，還真不知道要如何著手。

一直到兩年前接觸到 Scrum，似乎可以用來填滿原本 XP 在 Teddy 心中懸缺那一塊空間。嗯，Scrum 所規範的 Roles (Product Owner、Scrum Master、Team)，Activities (sprint planning、daily scrum、sprint、sprint review、retrospective)，Artifacts (stories、product backlog、sprint backlog、tasks、burndown chart)，加上 XP practices，真是太速配了。難道這組搭配就是 Teddy 多年來尋尋覓覓的「正確答案」嗎？

很接近了，但是還差那麼一丁點。在某些情況下，Scrum 並不太合適。例如，假設你工作的團隊是要維護某個產品，每天可能都會有客戶回報關於這個產品的 bugs，而你必須要儘快處理。在這種情況下，sprint 長度就算是訂成一週，都嫌太長。如果訂成一天，又可能連一個 story 都無法做完。前幾天看了 Lean 的書籍，裡面提到了 Kanban (看

板) ，似乎可以填補這個空缺。就是這個光，Teddy 現在覺的這一套組合產品 Scrum + Lean + XP (ㄟ，其實都是敏捷方法) 滿有潛力成為 Teddy 心目中的「正確答案」。不過目前還沒有機會去嘗試 Kanban，所以暫時沒有任何實際的經驗可以報告。

***

友藏內心獨白：別再找了，根本沒有什麼「正確答案」，只有「適合或不適合的答案」。

## 備註

[1] J. W. Reeve, *What is Software Design?* http://www.developerdotstar.com/mag/articles/reeves_design.html.

# 10 導入 Scrum？謝謝再聯絡。

05/29 22:51~05/30 00:22

12/19 09:49-10:01

原文發表於 http://teddy-chen-tw.blogspot.com/2010/05/scrum.html.

在台灣很多軟體開發團隊還是採用很傳統的方法來開發軟體（路人甲：Teddy 你不愛台灣喔，一直唱衰台灣），PM（專案經理）不斷逼迫程式設計師要趕上進度，程式設計師不斷加班試圖趕上不可能達成的進度表。為了「回報」PM 的壓迫，程式設計師很大方的送給 PM 為數可觀的 bugs。這種合作共生的模式，讓大家看起來都有事可作，有班可加，工作「看起來」很辛苦卻又不用花什麼腦筋。搞不好年終時老闆大發慈悲，多賞幾個月的年 終或是多發幾張股票，以慰勞員工這一年來的辛勞。

如果軟體開發團隊導入 Scrum，則 PM 的角色消失了，取而代之的只剩下 Product Owner、Scrum Master、以及強調**「自我管理」**的 Developers。每天工作 8 小時，準時下班。整個 Scrum Team 一團和諧，大家互相幫忙，共同解決問題。每個 sprint 結束，便可立即看到可以 demo 的成果。Product Owner（顧客的代理人）在每個 sprint 開始可以依序對顧客價值的高低來決定實做功能（stories）的先後順序，或是修改需求的內容。團隊的開發流程有 Scrum Master 關照（團

隊是否有依循 Scrum 精神來開發軟體），並且將持續改善的精神落實到每個團隊成員身上。慢慢地，bugs 變少了，對於軟體開發進度的「猜測」越來越準，開發的功能和顧客的落差越來越小，每個 sprint 結束時都有一版可以交付給顧客使用的軟體，開發人員的流動率極低，每個開發人員也都有能力可以修改整個系統。

這不就是軟體開發團隊的「大同世界」嗎？

這麼棒的方法，人客啊，你要不要用？

***

鄉民甲：不要。

Teddy：Why?

鄉民甲：原因如下...

- 沒有 PM？那現在的 PM 要吃什麼？ 台灣的失業率已經夠高了，為了不讓吳院長下台，PM 第一個跳出來誓死反對。
- 整個軟體開發流程變得這麼透明，哪來的混水摸魚空間。Teddy 你沒聽過「水清則無魚」嗎，Scrum 違反 ~~人權~~ 人性啊。

- 我們的 developers 跟牛一樣，都很被動的啦，一定要一個口令一個動作，不可能「自我管理」。要不然到最後一定會亂成一團，變成沒人管理。
- 每個 sprint 都可以修改需求？！那不是被客戶搞死。
- 每天準時下班，這不是討打嗎。老闆會認為我們過的太爽了，不夠努力，操的不夠。不然就是進度訂的太鬆，原本三個月要完成現在改成一個月就要生出來，看你加不加班。
- 聽說 agile methods 都沒在計畫，不寫文件也不畫甘特圖，這樣老闆和客戶都不會同意的啦。
- 現在的模式 run 的好好地（操的是員工又不是我），幹麼改用這個連聽都沒聽過的 Scrum。萬一失敗不是害我被老闆罵，還是少做少做，不做不錯比較保險。
- 什麼，導入 Scrum 還要收費？太貴了啦，我們自己慢慢 try 就好了（自己隨便 try 了一個月後失敗，因此認為 Scrum 不適合公司文化）。

Teddy：**總而言之，你就是對我的 ~~撒尿蝦牛丸~~ Scrum 沒有信心？**

鄉民甲：謝謝再聯絡。

***

涉世未深的 Teddy 曾經認為軟體從業人員總是希望能用最好（至少也要是比較好）的方法來開發軟體。錯，Teddy 又再次大錯特錯。**如果大家的開發模式都一樣爛，那這種模式就不叫做爛了，而是叫做「主流」**。導入一種和現行作法差異很大的軟體開發新「思維」（流程），是需要很大的**「勇氣」**的。嗯，越來越能夠體會為什麼 Kent Beck 要把「courage」列為 XP 四個「價值」之一了。用的人和推導的人都需要有打死不退的精神才有可能成功。

***

李大人：「我可能不會導入 Srcum」。

程幼靖：「謝謝，感激不盡」。

***

友藏內心獨白：現在是在演那齣？

# 11 都市游擊隊

March 28 20:06~21:13

12/15 10:24-10:31

原文發表於 http://teddy-chen-tw.blogspot.com/2011/03/blog-post_28.html.

看到標題不要以為這一篇是要談 online game 或是恐怖份子，今天要談的還是 Scrum。

不知道鄉民們有沒有這種經驗，學習了 Scrum，任何軟體工程的方法或是所謂的 practices，想要在自己的團隊中大展長才，好好地應用一番，但是卻不時遇到亂流，地震，甚至是海嘯：

- 在硬體公司中軟體開發不受重視。
- 沒有資源，要什麼沒什麼。要 tester...自己測；要 echnical writer...自己寫；要測試...沒設備...
- 人力都已經極端不足了，還被抽調人手去支援其他專案。
- 計畫比不上老闆的一句話... 時程和需求一直變更。
- 隔 team 如隔山，而且還是喜馬拉雅山。所有工作只要牽扯到其他 team（同部門或不同部門）就變得窒礙難行。
- 三姑六婆與鄉民們總是在該講話的時候不講話，不該講話的時候亂放炮。
- 有些鄉民一時興起想要教你如何「用嘴巴開發軟體」。

- 另外有些鄉民想要「靠不停的轉寄 email」來了解客戶的需求。
- 還有些大爺們希望你帶領著「九條好漢」在一年內完成反攻大陸的偉業。

遇到這種情況，請問單兵該如何處置？

請鄰兵以火力掩護我....

很抱歉，鄰兵不是老早就已經「投敵」，就是正在以「省電模式」運行中，哪來的火力支持你。

Scrum 告訴我們，要改善團隊，改善產品，改善公司...拜託，人家都可以用飛彈去攻擊示威的民眾了，還是「趴著，趴著，卡賣中槍」。

***

無論你是 Scrum Master、Product Owner 或是 Developer，是不是也曾幾何時感到空虛，感到寂寞，感到冷，~~想找一個不是那麼臭的男人~~ 想找一家不是那麼臭的公司來混一混....

請看「建築家 安藤忠雄」這本書，學學他的「都市游擊隊」哲學。

「住吉長屋」是安藤忠雄的出道作品，該建築的特色是：

- 房屋四面都被牆包圍，沒有對外的窗戶。除了入口以外，沒有別的開口。
- 整個房屋都是清水混凝土的表面。
- 把已經很狹小的長方形建地再切成三等分，中間當作露天中庭。

有興趣的鄉民們 google 一下「住吉長屋」就可以找到一些照片。當 Teddy 讀到這邊的時候，也覺得很奇怪，哪有人蓋房子不開窗戶的，這是哪根經不對了才設計出這種封閉的建築？！

簡而言之，安藤忠雄認為（以下是 Teddy 的解讀）：

- 大環境不好（窗戶打開不是看到隔壁正在晒太陽的內衣褲，就是面對別人的抽油煙機或是冷氣機）。
- 在不好的大環境中，如何創造自己的小宇宙？
- 結論：
    - 把自己封閉起來，不要看到外面的醜陋（房屋四面都被牆包圍，沒有對外的窗戶）。
    - 將外部空間包覆於住家之中（把屋子切成三等份，中間當作露天中庭）。

安藤忠雄沒有能力（至少在當年）去改變整個大環境，只好以「都市游擊隊」的形式，將自己隔離於世，蓋起自我封閉的碉堡。

***

Scrum 說，要管理階層的支持。問題是，要是管理階層沒有意願支持怎麼辦？

Scrum 說，要有 Product Owner 代表客戶與團隊溝通。問題是，要是 Product Owner 是個馬屁精，只會轉寄 email 與壓榨 Developers 該怎麼辦？

Scrum 說，Developers 要「自我管理」。問題是，Developers 都還是「化外之民」該如何自我管理？

你，不幸剛好是這個團隊的 Scrum Master，在心灰意冷之際，不小心看到「建築家 安藤忠雄」這本書，豁然開朗。既然整個城市不可能打掉重練，那就只能先蓋個「都市游擊隊住宅」。換句古人的話，先「修身」→「修 team」→「修 Product」→「修公司」。無論環境再惡劣，第一關「修身」還是可做的到的。沒事多看「搞笑談軟

工」就算是一種修身了。感恩啊...

***

友藏內心獨白：本篇有點玄。

# 12 如何估算 story point？

5/03 22:29 5/04 00:30

12/12 17:22-17:51

原文發表於 http://teddy-chen-tw.blogspot.com/2010/05/story-point.html.

前幾天實驗室學弟問了 Teddy 一個在 Scrum 中如何估算 story point 的問題，在此 Teddy 談一下自己曾經試過的兩種估算方法。首先假設 Scrum 團隊有四個開發人員，每一個 sprint 長度為兩週 (10 個工作天)。

**方法一：首先將 1 個 story point 定義為「一個理想的工作天 (八小時)」。接著在估算 Story 與 Task 的時候採用相同的單位 (也就是都以 story point 來估算)，並使用 focus factor 來調整團隊實際可貢獻於專案的時間。**

這個方法是 Teddy 從 Scrum and XP from the Trenches 這本書學來的，在 sprint planning 開始的時後，要知道這個 sprint 可以挑多少 story 出來做，首先就要估算這個團隊可用的 story point。估算方法如下：

**開發人數 x 可以獲得的工作天數 x focus factor**

在一個四個開發人員，sprint 長度為兩週的專案裡 (沒有人請假，如

果有人請假要扣掉請假的天數)，可以得到

4 x 10 = 40, 也就是說理想上這個團隊有 40 個 story points 可以使用。也就是說，理論上這個團隊可以完成 40 個 story points 的需求。

但是大家都知道，不可能一天 8 個小時都在從事開發工作，有時候要開會、回 e-mail、MSN 打屁一下、泡杯咖啡、上廁所、吃下午茶等等。有時候要回答客戶問題，或是應付老闆三不五時丟出來奇怪的要求。也可能公司環境太吵或是在大熱天冷氣突然壞 掉，害你無法專心寫程式。有太多因素造成實際上不可能達到 40 story points 這個「理想值」。所以，就必須要把這個理想值打個折，至於要打幾折的這個數字，就叫做「focus factor」。focus factor = 1 就表示理想值 (買東西不打折)，facus factor = 0 就表示沒有任何生產力 (買東西不用錢啦)。當然 1 和 0 應該都是不合理的數值，至於這個值應該要給多少，每個團隊不一定，總之概念上就是 focus factor 越高表示團隊專注力越高，也就是大部分的時間都可以用來從事 sprint planning 所規劃的開發活動。對於一個剛開始採用 Scrum 的團隊，可以選用70%左右的值。

所以，我們得到了 40 * 0.7 = 28 (story points)

接下來假設 Product Owner 挑選了 A、B、C、D、E 這五個 stories，

團隊成員估算出這五個 stories 的 story point 分別為：

| **Story** | **Story Points** | **Story Points 累記** |
|---|---:|---:|
| A | 8 | 8 |
| B | 5 | 13 |
| C | 13 | 26 |
| D | 3 | 29 |
| E | 5 | 34 |

由於團隊預計只有 28 story points 可以使用，如果要完成 A-D，則需要 29 story points，因此如果保守一點那麼這個 sprint 就只會挑出 A-C (26 story points)這三個 stories 來施工。當然此時 Scrum Master 也可以詢問團隊成員，是否有信心可以完成 A-D，如果大家都覺的 OK，那麼也可以挑選 A-D（如果 sprint 結束前來不及完成所有的 stories 那麼可以選擇把沒完成的拿掉）。

假設我們保守一點選擇了 A-C 這三個 stories，接下來就是將 story 細分，列出完成每一個 story 所需的 tasks (task 才是真正的施工內容)。典型的 task 有：寫 acceptance test cases、設計物件、設計介面、設計資料庫、coding、unit testing、寫文件等等。基本上估算 task 所使用的單位應該是「小時」。但是由於 story point 是沒有單位的 (當然你可以用 1 story point = 8 hours 這種「匯率」來換算)，如果估算 story

用 story point，而估算 task 用 hour，那麼團隊成員會不自主的去換算，造成估算困擾，因此有一種簡單的方法就是 task 也用 story point 估算。

舉個例子，假設 Story A 是 8 story points，經過 task breakdown（工作細分）之後團隊列出了以下完成 Story A 所需的幾個 tasks：

- Write acceptance tests: 8 hours (1 story point)
- Design UI: 8 hours (1 story point)
- Write UI code: 13 hours (1.625 story point)
- Write DAO code: 13 hours (1.625 story point)
- Write DAO unit tests: 8 hours (1 story point)
- Test: 5 hours (0.625 story point)

Task 所需時間加起來 55 hours = 6.875 story points。這時候你會想，ㄟ，剛剛估算 Story A 是 8 story points，但是實際列出完成這個 story 所需的 tasks 只需要 6.875 story points，那麼是要把 Story A 改成 6.875 story points，還是增加 task 所需的時間，想辦法把 task 的總時間也變成 8 story points？不知道鄉民們在剛開始嘗試 Scrum 時會不會有類似的困擾？Teddy 剛開始採用 Scrum 時是有這樣的困擾，也嘗試過三種修正方案：

1. 不理他，還是用原本估算的 Story 所得到的 story point 來做為這個 sprint 可以完成多少 stories 的依據。

2. 不理他，但是改用所有 task 加總的時間來作為這個 sprint 可以完成多少 stories 的依據。
3. 改用 button-up 的方式來估算一個 Story 的 story point。也就是說，先不估算 A、B、C、D、E 的 story point，而是直接做 task break down，然後把每一個 story 的 task 的加總作為該 story 的 story point (有點繞口)。

先講修正方案三，這種方法開發人員很容易了解，但是其實是不太符合 Scrum 精神。因為原本估算 story point 的目的是用來評估不同 story 之間的「**相對大小**」，並不是要得到一個精確的數值說 Story A 是 6.25 story points，Story B 是 5.125 story points ...。**先估算 task 然後再把 task 加總的時數換成該 Story 的 story points 看起來很合理也比較「精確」，但是卻完全失去了原本估算「Story 相對大小」的精神，而且還會引伸出「當實際施工時，發現原本估算的 task 時間不準，那麼是否要重新調整 Story 的 story point 這類的問題」。**

後來 Teddy 上過 Certified ScrumMaster 課程之後，慢慢體會修正方案二應該是比較好的方法，這也衍生出 Teddy 在這邊要介紹的第二種方法：

**方法二：估算 Story 採用 story point，估算 Task 採用小時。Story point 本身沒有單位，每一個 Story 的 story point 是採用「相對大小」來估算的，不用拘泥於 1 story point = 8 hours 這樣的單位換算。至於**

**每個 sprint 可以完成多少 stories，則是以 task 加總的時數來計算。**

在 sprint planning meeting 開始的時後，要先計算團隊在這個 sprint 可用的「工作時數」，計算方法如下：

**開發人數 x 可以獲得的工作天數 x 每日工作小時**

在一個四個開發人員，sprint 長度為兩週的專案裡 (沒有人請假，如果有人請假要扣掉請假的天數)，可以得到

4 人 x 10 天 x 一天工作時數 (5 小時) = 200 小時

這邊雖然省略了 focus factor，直接填入團隊（平均）每天可實際投入開發的時數，當然這個時數也是打過折的，所以效果和 focus factor 是類似。**Teddy 曾經在某本書中 (忘了哪一本了) 看到一個員工平均一天可以有 5 個小時不受干擾的工作時間就已經很不錯了，因此後來 Teddy 都直接用 5 個小時來估算團隊實際可以投入的時間。**

得到了預計可以投入的工作時數之後，其他的步驟就和 Teddy 在方法一中所說得很類似了，只是這個 sprint 可以完成多少 stories 是在估算 task 階段才決定的。以剛剛的例子：

| Story | Story Points | Story Points 累記 |
|---|---|---|
| A | 8 | 8 |
| B | 5 | 13 |
| C | 13 | 26 |
| D | 3 | 29 |
| E | 5 | 34 |

當估算完 Story 的 story points 時，這時候還不確定可以完成多少個 stories。假設經過 task breakdown，得到完成這些 stories 所需的時間為：

| Story | Task 小時 | Task 小時累記 |
|---|---|---|
| A | 55 | 55 |
| B | 16 | 71 |
| C | 48 | 119 |
| D | 22 | 141 |
| E | 25 | 166 |

結果發現雖然 A-E 的 story points 為 34，但是實際施工預估時間卻只要 166 小時，所以在這個 sprint 除了可以完成 A-E 之外，還可以再選新的 story 進來。

鄉民甲：為什麼 Story A (8 story points) 需要 55 小時，而 Story C (13 story points) 明明比 Story A 要大，卻只要 48 小時？是不是代表 Story C 的 story points 要重估？

Teddy 回答：**不需要**。再強調一次，Story 的是估算彼此之間的「相對大小」，例如，你要做使用者新增，查詢，刪除，修改這四個 stories。團隊成員認為這四個 stories「大小」是差不多的，因此都估 5 story points。假設你在 Sprint 10 先完成了「新增」這個 story，由於這是這系列的第一個 story，你需要設計資料庫，所以你花了 35 小時，但是後面三個 stories 並不需要設計資料庫，你可能分別花了 28, 25, 31 小時。實際施工所花的時間會受到很多因素影響，例如認領工作的人對於該工作的熟悉度，是否在該工作中使用到新的技術，不確定性的高低，發生難解的 bugs，生病或是和男女朋友吵架導致精神不佳等等。但是這並不代表「Story 的大小改變了」，所以不需要因為 task 預估或是實際花費的時間來修改 story points，除非當初估算 story 的相對大小估錯了。

再舉個日常生活的例子。老師叫學生去操場拔草，小胖身強體壯，「預估」只要 1 小時就可以拔完。如果是換成瘦小體弱的小英，則「預估」需要 3 小時。**同樣一個 Story (操場拔草)，小胖 1 個小時可以做完，小英需要 3 小時，我們需要重估這個 Story 的「大小」嗎？當然不需要，因為操場裡面的草，不會因為要去拔它的人不同而有**

**任何改變。**所以，只要在估算所有 story 時採用「相對大小」的方式來預估，那麼就不需要因為「預估」所需完成該 story 的 task 時間或是「實際完成」的 task 來重新調整 story point 了。長此以往，團隊還是會得到一個穩定的 velocity (每一個 sprint 實際完成的 story points)。

***

友藏內心獨白：習慣之後就變得很簡單了。

# 13 Story point 為何沒有單位：相對論篇

07/28 22:20~23:30

12/18 21:33-21:35

原文發表於 http://teddy-chen-tw.blogspot.com/2010/07/story-point.html.

咳，咳，Teddy 不是物理學家，對於愛因斯坦的「相對論」，當然是完全不了解。相信懂相對論的鄉民們應該不會淪落到看這 ~~搞屁~~ 搞笑談軟工部落格。這幾天 Teddy 剛好看到兩篇滿有趣的文章，多多少少都和相對論有點關係，剛好可以用另一種角度來解釋 Scrum 裡面的 story point，以下請看。

## 故事一

請翻開「徐詩偉」所著「管得少，是我故意的」這本書第 184 頁：

近代有一位著名的天文學家 Fred Hoyel 認為，若從愛因斯坦相對論的這個理論出發，**觀測者**與**觀測標的**之間的關係，**都是相對的**。**而人類所定義的真理，也都是基於所觀測的角度而來的**。若一個人從地球上觀測太陽，太陽的確是繞著地球轉；但是從太陽系的角度來觀測，地球卻是繞著太陽周轉。從這個角度而言，教廷數世紀來所

主張的地球中心論，其實和伽利略說的太陽中心論，都是正確的，也都是錯誤的。

從以上這個故事先得到第一個小觀念：觀測是一種相對的概念，觀測的結果取決於「觀測者」與「被觀測者」之間的相對關係。

## 故事二

請翻開本期（2010 年 7 月號)「科學人」雜誌中文版，第 39 頁「時間只是幻覺嗎？」:

「卡」(還沒開講就被喊卡)。ㄟ，這個故事無法表達，Teddy 讀完之後雖然大部分都不懂，但是卻覺得很有趣。Teddy 敘述一下第 43 頁裡面的一張圖所表達的內容。

時間是描述物體運動或改變的快慢的一種方式，例如光波的速度，心臟的跳動，或是行星自轉的頻率...

- 光速：每秒 30 萬公里。
- 心跳：每分種 75 下。
- 地球：每天自轉一周。

我們也可以去掉上面這三個事件的時間單位：

- 地球每自轉一周，心跳 10 萬 8000 下。
- 每心跳一下，光走 24 萬公里。

因此，有些物理學家認為，**時間只是一種通用的貨幣，有了它，描述這個世界會較容易**，但其本身無法獨立存在。**以時間來測量過程就像以金錢而不是以物易物來買東西。**

從第二個故事得到另一個小觀念：藉由描述事件「彼此之間的關係」，我們可以不使用時間來描述事件。

***

通常講到這邊又該輪到路人甲上場了。

路人甲：這和 story point 有何關係？

Teddy 在第 65 頁「如何估算 story point？」中提到：

估算 story point 的目的是用來評估不同 story 之間的「相對大小」，並不是要得到一個精確的數值說 Story A 是 6.25 story points，Story B 是 5.125 story points ...。

舉例說明，老師叫學生去操場拔草，小胖身強體壯，「預估」只要 1

小時就可以拔完。如果是換成瘦小體弱的小英，則「預估」需要 3 小時。同樣一個 Story（操場拔草），小胖 1 個小時可以做完，小英需要 3 小時，我們需要重估這個 Story 的「大小」嗎？當然不需要，因為操場裡面的草，不會因為要去拔它的人不同而有任何改變。所以，只要在估算所有 story 時採用「相對大小」的方式來預估，那麼就不需要因為「預估」所需完成該 story 的 task 時間或是「實際完成」的 task 來重新調整 story point 了。

***

回想一下 Teddy 版的「亂談相對論」所開示的兩個重點：

- 觀察的結果，取決於觀察者與被觀察者之間的相對關係。
- 只要描述事件的相對關係，就可以不使用時間這個「貨幣」來描述事件。

Story point 的精神也是一樣，既然「一個 story」要做多久（**這是時間單位喔，例如三天或五天**）取決於「誰來做這個 story」，所以直接用「時間單位」來估算 story 是沒有意義也無法估算準確的。所以，套用第二個重點，只要描述「不同 story 之間的相對關係（相對大小），就可以不使用時間單位來描述事件」。

當然啊，即使是「估算相對大小」都不是一件容易的事（這不是件

容易的事，估不出來開會到死。讓它淡淡的來，讓它好好的去。到如今年復一年，我不能停止 delay，delay 一年，又過一年。但願那客戶瞎眼，爽快地趕快付錢，掐死你的 PM...XD），所以才會有人想出用「poker game」的方式集眾人的力量來 ~~殺價~~ 估算，以免不小心買貴了...

結論：經過 Teddy 的考證，證明這些搞 agile 的人物理都學的不錯喔。

***

友藏內心獨白：這種聯想力，就跟看香爐中的香灰去猜大家樂明牌有何不同？

# 14 end-to-end stories：切蛋糕篇

01/20 22:56~23:45

12/18 22:33-22:36

原文發表於 http://teddy-chen-tw.blogspot.com/2010/01/end-to-end-stories.html.

蛋糕這種食物，在物質條件如此富裕的寶島台灣，相信沒吃過的人應該很少。就算沒吃過，也應該看過，如果沒看過，請到 8X 度 C 的櫥窗看一下。

假設鄉民們買了一個生日蛋糕，除非你是買來砸人的，否則吃之前應該都要先切一下蛋糕。蛋糕有兩種切法，水平切法與垂直切法。一般正常人類，採用的是垂直切 法，所以切出來的每塊蛋糕，包含了整個蛋糕的每一層，一口咬下去，可以吃到最上層的鮮奶油，第二層的香草或巧克力夾層，第三層的布丁，第四層的水果... 依此類推，真是太好吃了。

至於打從火星來的鄉民們，則是採用水平切法，切出來每塊蛋糕，就只包含某一層的餡料。一口咬下去，噁...吃到滿口的鮮奶油，或是布丁，水果...。這算是在吃蛋糕嗎，還是在吃 N 種不同的食物？

***

軟體開發活動中的需求撰寫，無論是寫成 use cases、scenarios、或是 stories，認真一點的鄉民們，應該還記得教科書上一定會提到，要寫成 **end-to-end** [use cases、scenarios、stories]。想當年 Teddy 在當 OOAD 助教的時候，也不時提醒學生們：「你這個 use case 不是 end-to-end」不過說真的，在當下聽進去的人可能不多。

寫出非 end-to-end [use cases、scenarios、stories]，就好像切蛋糕沒切好，吃的人（你的客戶）可能吃到整片的鮮奶油。你告訴他，這就是蛋糕...不可分割的一部分，先把這個啃掉，下次給你 蛋糕的另外一層....N 個月後你就可以獲得一塊完整的蛋糕了。

所以，為了讓客戶在你切下每塊蛋糕的同時，就可以立即享用，你應該把一大塊蛋糕（某個需求），垂直切成很多塊的小蛋糕（很多較小的 stories）。每切完一塊蛋糕（每做完一個 story），客戶都可以吃到一整塊的蛋糕（得到一個對客位而言有價值的功能）。吃到一整塊蛋糕，即使很小一片，還是有算吃到蛋糕。吃到一整塊鮮奶油，即 使很大一片，還是很噁...不算吃到蛋糕。

如果以後還有人問你：什麼是 end-to-end [use cases、scenarios、stories]，如果是好朋友，就買塊蛋糕請他吃，如果是路人甲，就請他自己去看一下 8X 度 C 的櫥窗。

***

友藏內心獨白：肚子餓了。

# 15 功課寫完沒: The definition of done

12/19 08:56-08:59

原文發表於 http://teddy-chen-tw.blogspot.com/2009/10/blog-post_25.html.

根據事後諸葛亮表示，小朋友最怕父母問的一個問題，就是「**功課寫完沒?**」短短五個字，對幼小的心靈而言，卻具有如同原子彈般的殺傷力。因為這個問題的答案，決定了接下來的時間，小朋友們能不能：

出去玩

看海綿寶寶

Play Game

吃點心

看漫畫

眼看人生中這麼多美好的事情，就要被一個簡短的是非題給毀了，幼小的心靈學會了什麼叫做善意的謊言：「寫….完…了…啊（答的很心虛，事實是，完全沒動，或是亂寫一通）」

當父母的自然也不是省油的燈：「寫完了啊，拿出來我看一下」，當場就被抓包了。

***

隨著時間飛逝，小朋友長大了，從此過著幸福快樂的日子.... 錯！這個「功課作完沒?」的問題變成：

程式寫完了嗎、bugs 解了嗎、軟體可以上市了嗎

為了不傷老闆的心，你大聲說出：寫完了、解好了、隨時可以上市

此時的你，心中浮現小小的吶喊：

寫完了，還沒測試
這個 bug 解好了，但是以另外一個 bug 的身分繼續存在著
隨時可以上市，附贈 bugs 吃到飽以及口齒不清的說明文件

「你等會」曾說過：「阿共再大，也沒有我老爸大」，這句話一點也沒錯。爸爸媽媽可以抓出你功課沒寫完，而阿共卻看不出你案子沒做完。隨著年紀漸長，功課有沒有寫完這個問題越來越難回答，因為常常連做功課的人也不知道，功課的範圍有多少，何時才寫的完。

**定義功課寫完沒 （The definition of “done”, DoD）**

在 Scrum 裡面，Scrum 團隊如何知道需求是否已經做完，可以被 Product Owner 驗收？所以 Scrum 提到團隊需要有一個對於「完成的定義」。這個定義可能包含：

程式可以動、通過單元測試、通過其他各種測試（integration, function, system, security, performance, availability, robustness…）、重整程式（refactored）、更新重要設計文件、完成使用手冊、可以被安裝等等。

唯有事先定義好 done，才能知道每一個 sprints 真正完成了多少功能，而不是灌水的數據，也才有可能在每個 sprint 結束時，產生一份「可以交給老師的作業」（potentially shippable products）。

***

友藏內心獨白：DoD 也可以隨著時間而修正。

# 16 我不能採用 Scrum，因為我家人不同意

March 14 20:40~22:17

12/15 11:17-11:25

原文發表於 http://teddy-chen-tw.blogspot.com/2011/03/run-scrum.html.

在完美的世界裡，Scrum 是這樣玩滴：

- **Product Owner** 對於專案所要處理的 **problem domain** 有多年的經驗，經常與客戶互動並且熟悉競爭對手的產品。Product Owner 腦袋清楚，說話條理分明，懂得事情的輕重緩急，文筆流暢，寫的「一手好 stories」。Product Owner **很有 guts**，對外承擔專案成敗的責任，絕對不會把失敗歸咎於開發成員。
- **Scrum Master** 熟悉敏捷方法與各種敏捷實務作法的精神，個性樂觀，有很強大的**「正面力量」**，就算是遭遇到挫折也能夠從失敗中學習並鼓勵隊友。Scrum Master 是一隻很稱職的「看門狗」與「牧羊犬」，保護團隊成員不受不相關的人、事、物打擾，而且當團隊成員偏移 Scrum 精神（或 agile 精神）時可以適時地引導他們回到正軌。Scrum Master 知道如何幫助團隊逐步與持續地改善軟體開發流程。

- **Developers** 有著追求卓越的企圖心、有勇氣、誠實、積極主動、樂於互相協助幫忙。Developers 相信管理的最高境界就是**「自我管理」**，並深信採用敏捷方法來開發軟體將有助於達到自我管理的境界。

以上敘述，了解 Scrum 的鄉民們應該都已經知道了，也希望有一天能夠成為這「理想團隊」的一員。很可惜現實世界是很殘酷的，就好像雖然棒球選手人人都想進入大聯盟「洋基隊」嘗一 嘗一秒鐘幾十萬上下的感覺，但是可能偏偏只有「國民隊」對你有興趣，更慘的可能只能到小聯盟或是留在「簽賭聯盟」...XD。

***

**總之，人生最慘的事之一，莫過於你認識了一位美如天仙或是俊如潘安的女/男朋友，但卻因為「家人反對」或「社會觀感」，最後不能長相廝守（過兒…姑姑…）**。Scrum 也一樣，鄉民們可能不小心知道了 Scrum 這個「東東」，覺的用 Scrum 來開發軟體真是太讚了啦，但是卻因為「父母或是家人反對」（大，小老闆反對或是與不符合公司文化），而導致無緣繼續交往，殘念。

有哪麼嚴重嗎...你說....

當然有，尤其是如果鄉民們是那種**「在硬體公司寫應用程式的人」**，

那就更慘，因為在台灣絕大多數的硬體公司，不管老闆公開說軟體如何重要又如何重要，開發軟體的人永遠都只是配角中的配角。想偷偷娶個「門不當，戶不對」的太太（就是 Scrum 啦），門都沒有。

舉幾個最簡單的例子：

- 老闆說：蝦米是「死光」？
- 某大頭說：我不管你們採用什麼 engineering practices，公司有公司的「~~馬屁文化~~ 制度」要遵守，我建議你（Scrum Master）先搞懂公司的「制度」。
- 這個專案是公司很創新的一個計畫，要做「雲端殺豬系統」，目前世界上沒有人開發過，所以找不到有經驗的 Product Owner。
- 既然沒有合適的 Product Owner，不然你就來當這個 Product Owner 好了。
- Scrum Master 和老闆吵架當場離職走人，或是
- Scrum Master 從看門狗「進化」成哈巴狗。
- Scrum Master 說：什麼是 agile?
- Scrum Master 說：我叫你先做這個 task 你還不聽！
- Product Owner 說：我不管，這些功能三個月全部給我做完。
- Product Owner 說：客戶試用過我們的「雲端殺豬系統」（其實連 login 都沒有），他們建議要加上「雲端燉東坡肉功能」才要考慮購買。

- （Sprint 進行到一半）Product Owner 說：A 客戶急著「雲端煮滷肉飯功能」，只要做出來就馬上下訂單買一萬個授權，趕快在三天內生出來給他。
- Developer 說：我昨天在寫程式，今天準備繼續寫程式，沒有遇到問題。
- Developer 說：你（Scrum Master）又沒有告訴我這個 task 要先做...。
- Developer 說：你（Product Owner）需求又沒寫清楚...。
- Developer 說：為什麼要叫我寫 unit test？
- 要花多少時間才可以把員工從 cubicle（辦公室隔間）中拉出來改成「排排坐」？
- 公司可以接受 Scrum 團隊成員沒有個別考績這件事嗎？
- XX 主管：人家都用 14 吋的 notebook 寫程式，你寫什麼「大」程式要用到 24 吋的螢幕？
- XX 主管：別的工程師都買一台兩萬八的筆電，你們為什麼要用到四萬的？不准！
- XX 主管：用無線鍵盤，滑鼠寫程式會比較快嗎？
- XX 主管：你寫什麼程式要用到 8G 記憶體？
- 硬體部門都有沒專屬會議室了，不可能給軟體部門專用的會議室。

***

前幾天日本發生了芮氏地震規模 9.0 的超級大地震並且引發大海嘯與核電廠危機，很多人都在談日本人對於災害應變的規劃做的真是好，要是這麼大的地震發生在台灣，老早就.....（雖然事後發展看起來日本的應變沒有想像中的那麼好）

是「方法的問題嗎」？是日本人不肯把災害應變規劃跟我們分享嗎？還是我們**「不肯被分享」**？光有「好對象（好方法）」，「父母觀念不改」、「社會文化不進步」，梁山泊與祝英台、羅密歐與茱麗葉、小龍女和楊過的故事還是會繼續發生滴。

***

友藏內心獨白：「危大路屁」對白-- Is it good to drink...麥共家綴（別說這麼多），喝看看就知道...

# 17 0 與 1 的距離

Oct. 04 20:17~23:42

12/19 11:36-11:39

原文發表於 http://teddy-chen-tw.blogspot.com/2010/10/0-1.html.

幾年前 XP (eXtreme Programming) 正在流行的時候，其中一個經常被問到的問題：導入 XP 是不是一定要全盤採用 XP 的 12 的實務作法（the planning game、small releases、metaphor、simple design、testing、refactoring、pair programming、collective ownership、continuous integration、40-hour week、on-site customer、coding standards）？

在當年，軟體開發團隊掛上 XP 可是很酷的一件事，所以，有些團隊可能不了解 XP 到底是什麼，只是隨便寫寫 test cases 或是做做 refactoring 就宣稱「我們團隊採用 XP」，感覺走在時代尖端。為了刷掉這些名不符實的團隊，XP 的虔誠信徒會告訴你：「要買就要買整套的，缺一不可」（Teddy 內心獨白：這還真不容易，感覺有點像是在收集小丸子文具組...救人啊，怎麼 Teddy 拿到的都是「野口」！）。但是，務實派的人會說：「只拿對你有幫助的就可以了（就算是拿到「野口」也 OK）」。此時 XP 的虔誠信徒會說：「~~這不是肯德基~~ 如果不全盤採用，就不要對別人說你是 ~~華山派~~ XP 的門下弟子」。

這種「派系之爭」的歷史一再重演，這幾年 Scrum 竄起，很多人又想跟 Scrum 沾上邊。因此，Scrum 大師們怕大家沒事往自己臉上貼金，或是利用這個噱頭到處騙錢，信口開河說自己或是團隊擁有 Scrum 的經驗，因此也要時時提醒，哪些情況是真正符合 Scrum 的精神，哪些是濫竽充數。

Teddy 曾經因為這種「血統之爭議(正港的 Scrum 或是冒牌的 Scrum)」困擾了一陣子，為什麼呢？因為咱們寶島台灣，要找到一家公司可以完完全全、徹徹底底的實踐「理想中的 Scrum」相信是不太容易滴（也許用一雙手可以數的完），那麼其他廣大開發軟體的人，難道就因為「無法達到理想的條件」不可以親近 Scrum 嗎？

舉個例子，一個 5~6 人的小公司，準備開發 iPhone 軟體，他們可以找人扮演 Scrum Master，開發團隊成員也沒問題，但是沒有「專門的 Product Owner」。怎麼辦？最後決定由最有經驗的人（很可能是扮演 Scrum Master 的那個人）同時扮演 Product Owner。完了，Scrum 的書上說 Scrum Master 和 Product Owner「絕對」不可以是同一個人啊？那怎麼辦？

再舉個例子，在 Scrum 中「績效考核」是以看「整個團隊」的績效，而不是看個人。但是，如果公司打考績的時候，你跑去跟老闆說：「因為我們採用 Scrum，所以每個團隊成員的考績都一樣」，Teddy 相信

首先陣亡的人應該是你。

再來，Scrum 的目的希望能營造出一個「自我管理的團隊」... 理想很好，但是這些「刁民（開發人員）」跟牛一樣，進一步退兩步，要維持好不容易已經改善的一點點現況都很難了，別談什麼自我管理。

最後，雖然 Scrum 沒有硬性規定「不可以加班」，不過敏捷方法的精神應該都還是以「不加班為原則（一週工作四十小時）」。光是這一點全台灣可能 99.999% 以上的「高科技」公司就不符合了，那不是沒搞頭了。

***

實際上，日子還是要過下去。不完美的環境，並不能阻止我們追求更好的工作方法與生活品質。舉個例 子，Teddy 住在艋舺一棟快四十年的四層樓老公寓。Teddy 到國外旅行時，看到法國、瑞士、日本、美國的超優生活環境，回國之後難道要把自己家裡「炸掉」重蓋（不可能，因為口袋不夠深，而且不合法…Orz）或是繼續「苟且偷生」不做任何改變？在這兩個極端的選擇當中，還是有其他的選項。

- 老屋拉皮。
- 重新油漆。
- 更新照明。

- 種花。
- 掛畫。
- 貼明星海報 ... XD。

Teddy 在第 493 頁介紹的「The Quality Without A Name (QWAN)」這個概念，對導入敏捷方法也好或是實施 CMMI 也罷，要追求的是軟體開發的 QWAN。認清自己的現況，了解軟體開發的 QWAN（軟體開發的本質），兩者相減就是自己可以努力的空間。

**改善的過程並不是 0 或 1 的問題（要嘛什麼都不做，要做就一步到位），而是 0 慢慢地、慢慢地往 1 靠攏的過程**。0 與 1 之間，還是有很廣大發揮的空間。

***

友藏內心獨白：那一天才能夠爬到 1 的那一端？

# 18 Scrum 之逆練九陰真經

March 21 22:42~ March 22 00:21

12/15 10:35-10:51

原文發表於 http://teddy-chen-tw.blogspot.com/2011/03/scrum.html.

話說 Teddy 兩個多禮拜前因為多嘴在 Facebook 上和某人多聊了兩句，最後居然糊里糊塗的答應要去某個活動分享 Scrum 導入經驗... 今天沒什麼料，就來談一下 Teddy 準備分享的某個主題 –「逆練九陰真經：不完美的 Scrum」。

鄉民們應該都看過「射鵰英雄傳」，話說郭靖與洪七公被 ~~吸毒~~ 西毒歐陽峰困在一艘船上，歐陽峰逼郭靖把九陰真經默寫出來，郭靖當然是不肯，但礙於形勢所逼，最後洪七公想到一個辦法，要郭靖寫一本「九陰假經」給歐陽峰。由於世界上只有郭靖懂九陰真經，所以他把經書裡面的招式亂寫，歐陽峰也看不出來。

最後歐陽峰練了郭靖所寫的「九陰假經」，雖然是「九陰假經」，但是歐陽峰的武功還是大有進步，只不過代價是走火入魔，整個人變得神智不清。

以上所說和 Scrum 有何關係？鄉民們如果有上過 Certified

ScrumMaster 課程，或是有看過 Scrum 相關書籍，應該會注意到 Scrum 有提到「**Product Owner 和 Scrum Master 絕對不可以是同一人**」這一點規範。其他 Scrum 團隊不應有的現象還包含「某個團隊宣稱採用 Scrum 但實際上在 daily scrum 時還是每一個團隊成員都向 Scrum Master 報告」或是「採行 Scrum 最好要有高層支持」等等。有些人甚至會很嚴格的說，如果沒有遵循 Scrum 的精神或規範，那你不可以說你的團隊採用 Scrum，只能說是「Scrum minus」或是「死光」。

要求 Scrum 團隊要嚴格遵循 ~~古法釀製~~ Scrum 或是敏捷精神的原意應該是怕很多人「掛羊頭賣狗肉」，好比某些國家，明啊明是獨裁統治，卻偏偏要說自己是民主國家，這樣可不行。但是，如果標準放得太高，一定要做到美國或是西歐那種程度才算是民主國家，那很多國家就只能算是「民主 minus 國家」。

Teddy 相信在台灣很多聽過 Scrum 的鄉民們可能都曾經有一股想要在自己的專案中採用 Scrum 的衝動，但是，一開始可能先卡在 Scrum 框架所要求的「基本條件」上面。試想一下你在一家硬體公司開發軟體（寫驅動程式或是搭配硬體的應用軟體），公司的主管幾乎都是硬體出身的，不懂也不想懂軟體開發。有一天你這個小蘿蔔頭興沖沖的跑去跟老闆講「我們來 rum Scrum 吧」！下場會是（請選擇...）：

1. **巴林與利比亞**：二話不說，直接派兵亂槍打死。

2. **中國**：連想的機會都沒有...往好的方面想，至少保住一條小命...Orz。
3. **埃及**：抗議→亂槍→沒死算你命大→改變→原來是假的→繼續抗議→亂槍→沒死算你命大→…（說好的迴圈終止條件呢？）。
4. **突尼西亞**：抗議→抗議→改變。
5. **Teddy 共和國**：不用等鄉民開口，已經採用 Scrum。

Teddy 大膽猜測一下，大概前三者的比例比較會高一點。但是，鄉民們難道因此就要放棄 ~~民主制度~~ Scrum 嗎？ㄟ... **要看你有多不怕死**（上有高堂，下有妻小，步入中年口袋又不深的鄉民們請先閃一邊）。總之，明的不行，咱們可以來暗的。民國成立之後，不是有所謂的**「軍政、訓政、憲政」**分三個時期來實施民主制度嗎？鄉民們比歐陽峰還要有優勢，因為歐陽峰不知道什麼是「九陰真經」，只能傻傻的「逆練九陰真經」。鄉民們則是在「知道什麼是九陰真經（了解 Scrum 與敏捷精神）」的情況下，暫時逼不得已「逆練九陰真經」，所以走火入魔的風險稍微小了那麼一咪咪（路人甲：最後還不是走火入魔，只是時間早晚的問題...）。

***

舉個例子，Product Owner 是 Scrum 裡面相當重要的一個角色，負責

定義需求（撰寫 stories），對外負責整個產品的成敗。但是，**如果一個專案沒有 Product Owner 怎麼辦？**

路人甲：怎麼可能沒有？

ㄟ，沒有去找一個不就得了（老梗內心獨白：身份證掉了怎麼辦？撿起來不就好了）。應該是說，沒有「專任的 Product Owner」或是說「沒有對於 problem domain 很有經驗的 Product Owner」。講這樣鄉民們應該就懂了。很多軟體開發，都是「老闆有個念頭」→「員工做到 ~~禿頭~~ 白頭」。也就是說，很多所謂「新產品開發」不見得公司立刻都可以找到有經驗的 Product Owner 來帶領。那怎辦辦，案子還是要做啊，薪水還是要照領啊，總不能跟老闆說**「找不到 Product Owner 請換題目」**，或是**「等你找到 Product Owner 我再來開工」**。如果真的這樣作，那麼找到 Product Owner 之前的這一段空窗期總不能奉旨放無薪假吧。

那怎麼辦？說實話，很難辦，最後只能使出以下幾個爛招：

- **別人的需求，就是最好的需求**：一字曰之**抄**。
- **在爛蘋果裡面挑一個比較不爛的**：找一個團隊中最有概念的人來當 Product Owner，加上樓下這個方法…

- **三個臭皮匠**：專案一開始的時候把大家找來一起研究要「抄」的對象並研究從何抄起。

這樣能做出好產品嗎？Teddy 不保證，但是至少能讓你暫時繼續有薪水可領...XD（好死不如賴活著）。

***

總之，逆練九陰真經是很危險滴，非不得已不要輕易嘗試。君不見，很多獨裁者年輕的時候也是改革派，也做了很多對國家有益的事，只不過掌握權力久了之後不免就腐敗了（走火入魔）。記得，暫時「逆練九陰真經」或可增加功力，等情況好轉就要想辦法改邪歸正，以免長久下去有傷身體。

***

友藏內心獨白：這算是哪門子的 Scrum...

# 19 同誰，九陰真經不是這樣子練滴

March 23 20:44~22:11

12/15 11:43-11:51

原文發表於 http://teddy-chen-tw.blogspot.com/2011/03/blog-post_23.html.

上一篇寫了「Scrum 之逆練九陰真經」，希望鄉民們不要以為 Teddy 鼓勵大家「假 Scrum 之名，行亂搞之實」。上一篇的重點是，就算你是「逆練九陰真經」，好歹練功的內容還是和「九陰真經」相關連，總不能說當年郭靖給歐陽峰一本「瑜伽大全」或 是「第一次學有氧舞蹈就上手」然後騙他說這是「九陰真經」，這樣就「騙太大了」。

幾個禮拜前在某個 sprint demo meeting 中，某人用很認真的表情說了一句對 Scrum「大逆不道」的話，令 Teddy 印象深刻：

因為這個 story 太大了，在這個 sprint 中做不完。接下來我打算不要開始下一個新的 sprint，等我繼續把這個 story 做完再說。

這好比你去參加超市所舉辦的 「一分鐘大搬家」活動，在限時一分鐘內隨你搬任何超市內的東西。就在時間截止的時候，你跳出來說「等一下.... 我還沒搬完，等我搬到爽之後你們才可以換下一組人馬」。

***

Teddy 還曾經看過一個類似這樣的 story：

身為程式設計師，我可以設計一個具有擴充性的軟體架構。

不要笑（牙齒白啊），地球上就是有這種 story，說不定鄉民們不經意也會寫出類似的 story。這種 story 要怎麼施工，要如何 demo？Teddy 也可以舉一反三寫出類似的 stories：

- 身為食神，我可以做出宇宙無敵好吃的飯菜。
- 身為歌神，我可以舉辦場場爆滿的演唱會。
- 身為唬神，我可以寫出超級賣的企劃案。
- ~~身為員工，我可以月入數十萬~~。

反正 Scrum 說要把需求寫成 story 啊，好啊，你要 story，我就給你 story，誰怕誰啊，反正吹牛又不用繳稅！於是產生了上述的 stories....Scrum 的**「形式」**是有了，但是卻沒有抓到重點。這樣的練功方法，不要說「逆練」，就算是學小龍女躺在「寒冰床」上練，甚至是跑到「火星去練」，練的再久都沒用。

***

請問哪個程式設計師不想設計出「具有擴充性的軟體架構」，那個廚師不想做出「宇宙無敵好吃的飯菜」，那個歌手不想「舉辦場場爆滿的演唱會」，那個企劃人員不想「寫出超級賣的企劃案」？問題是，把「~~幻想~~ 願望」以 story 的形式寫出來不表示這個願望就可以實現（就算你把 product backlog 當成許願池，要許願前也要先投錢啊）。

那麼「具有擴充性的軟體架構」要怎麼達成？很簡單，利用「完成若干個功能性的 story 來達成」。這樣講沒人聽得懂，舉例說明，假設你要開發一個「具有擴充性的會計系統」，stories 可以這樣寫：

- 身為使用者，我可以安裝新的會計模組。這樣寫出來的 story 太大了，無法施工，繼續細分：
    - 身為使用者，我可以安裝薪資模組
    - 身為使用者，我可以安裝進貨模組
    - 身為使用者，我可以安裝 xx 模組

上面這幾個 stories 完成後，系統就具備了「功能模組擴充性」。接下來繼續寫 stories：

- 身為使用者，我可以設定薪資規則。一樣可以繼續細分：
    - 身為使用者，我可以計算全職人員的薪資
        - 身為使用者，我可以計算全職人員國內出差費

        - 身為使用者，我可以計算全職人員國外出差費
        - 身為使用者，我可以計算全職人員的加班出差費
        - .....
    - 身為使用者，我可以計算兼差人員的薪資
    - 身為使用者，我可以計算派遣人員的薪資

以此類推，可以一直寫下去。「具有擴充性的軟體架構」是一個很抽象的非功能需求（non-functional requirements or quality attributes），要達到此需求，首先先定義「什麼東西需要被擴充」。**藉由將「需要被擴充的功能寫成 stories」並「逐一完成這些 stories」，一個具有擴充性的軟體架構就完成了。**這些 stories 給「具有擴充性的軟體架構」規範了一個 context，在此 context 底下去實現此軟體架構才有意義。有點類似 UP（Unified Process）談的 use case driven 的概念，只是在 Scrum 中改成 story driven。

至於第一個問題，一個 story 如果太大在 sprint 快結尾時才發現做不完該怎麼辦？幾個比較可行的方法包含：

- 將這個 story 移到下一個 sprint 繼續做（在目前的 sprint 中，這個 story 就不算完成，也不用 demo）。
- 如果這個 story 就只有這個 story 那怎麼辦？

    - 承認這個 sprint 失敗，並檢討原因，是因為 sprint planning meeting 將 story 估的太大，還是 sprint 進行中發生什麼意外（bugs 太多，員工被抓去開會，支援其他案子等等）...
    - 如果這個 story 可以被細分，那麼看看這個 sprint 完成的內容可否完整的自成一個 story，如果可以那麼沒做完的另外寫一個 story 到下一個 sprint 繼續。
- 就算是這個 sprint 有很多 stories，而你認為這個未完成的 story 可以被切割，你還是可以 demo 已經完成的內容，並且將沒做完的需求另外寫一個 story 到下一個 sprint 繼續。

理想上 story 沒做完就是沒做完，應該移到下一個 sprint 繼續做。但是有時候這個 story 已經完成的部份的確是可以被單獨 demo 的，那麼倒不一定要強制整個 story 移到下個 sprint。例如，某個 story 原本要同時支援 Windows 與 Linux 平台，但是 sprint 快結束時 Linux 平台的支援還有一點問題。你可以選擇把整個 story 都不要 demo，也可以選擇把這個 story 切割成(1) for Windows 平台，(2) for Linux 平台，這個 sprint 就可以先 demo 已經完成的 Windows 平台功能（請不要說為什麼一開始的時候不直接把這個 story 拆成兩個...千金難買早知道啊...）。

***

採用 Scrum 遇到「問題」的時後，不是說「老子（老娘）想怎麼樣，就怎麼樣」，Teddy 建議可以視情況所需偷偷地「逆練九陰真經」，但是不是鼓勵鄉民們「亂練九陰真經」，到時候練壞身體不要怪 Teddy... 只好善用你的健保卡...XD

***

友藏內心獨白：為什麼開會的時候常常想學電視上表演那種「從椅子上跌下來」的橋段。

# 20 放下心中舉起的中指

04/14 22:48~ 04/15 00:20

12/19 09:38-09:46

原文發表於 http://teddy-chen-tw.blogspot.com/2010/04/blog-post.html.

對於 PM 或是 project leader 而言，讓人最沮喪的事情，莫過於團隊成員告訴你程式都寫好了，但是你稍微使用一下卻發有問題。遇到這樣的情況，典型的心情的轉折為：「沮喪」→「生氣」→「想罵三字經」。

無論最後那句 XXX 有沒有說出口，此時的心情是十分低落的。心中不自覺的響起許多怪東怪西的聲音：

- 這些該死的刁民 (developers)，程式寫好都不測試的啊！
- 這麼簡單的問題也要我來告訴你 (developers) ？你不是「應該」要自己想到的嗎？
- 真是太欠罵了，這個問題都發生過幾次了...
- 這就是我帶領的團隊嗎？是不是自己領導的太失敗了...

***

最近 Teddy 也遇到上述類似的情況。Teddy 目前的團隊採用 Scrum 方

法，但是團隊成員人數不多，並沒有專門的人來負責驗收測試。為了確保每個 sprint 結束時所完成的系統品質不至於太差，因此我們在每個 story 裡面都加了一個 testing task（這個 testing task 是做 functional testing，至於 unit testing 已經包含在 programmers 開發功能之中），以確保 story (需求) 從「未完成」變成「完成」之前，都被團隊成員測試過。通常我們會讓非開發該 story 的人來領取這個 testing task，以避免測試自己所開發的功能（因為「自己寫，自己測」通常更找不到問題）。最近 Teddy 比較少參與程式撰寫的工作，轉而認領比較多的功能測試工作。因此，經常會發現，~~許多~~ 有一些宣稱已經做完的 tasks，其實還是存在著 bugs。看到這邊，也許鄉民們或說：「程式有 bugs 是很正常的啊」。的確，以前 Teddy 也是這樣想，因此不知不覺安慰自己：「我們發生的 bugs 其實也不算很多或是很嚴重」。最近看了 Lean 與 Toyota Production System (TPS) 的書之後，提高了 Teddy 對於「品質」以及「減少浪費」的意識 （測試找出 bugs，修改 bugs，再測，這些都是額外的浪費），因此越發覺的應該要重視這個問題。

Teddy 遇到的 bugs 可以簡單的分成兩大類：

- **情有可原的 bugs**：雖然程式設計師有撰寫單元測試程式，但是 client 卻用某種「預料之外」的方式來呼叫自己的程式。這種情況勉強算是「情有可原」。
- **不可原諒的 bugs**：此類 bugs 又可細分為兩種

- **偷懶**：有時候程式設計師偷懶，只寫最簡單的單元測試。在正常的操作之下，只要步驟或是輸入資料稍微複雜一點點，就會出錯。
- **自作主張**：程式設計師在寫程式的時候，對需求不是很清楚（尤其是使用者介面的設計與操作方法），但又不肯問就坐在他旁邊的 product owner，而依照自己的想法把功能做完（通常都是過度簡化）。雖然完成的功能本身可能沒什麼錯誤，但是這樣的功能並不完全符合 product owner 所需要的需求。

當遇到「不可原諒的 bugs」時，真的很想罵 XXX。但是，生氣，除了讓 Teddy 的胃潰瘍更加嚴重之外，並不能解決問題（醫生曰：Teddy 你的胃潰瘍是因為壓力而造成的）。此外，因為 Teddy 自己也寫不出零錯誤的程式，因此也沒立場要求其他人不准犯錯。Teddy 選擇在 retrospective meeting 時把個問題提出來討論。根據討論結果，關於「不可原諒的 bugs」發生的原因為：

- 因為有一些和資料庫有關的程式，在測試的時候需要在資料庫中設定測試資料。而準備這些資料和執行這些測試程式都很花時間。因此，有時候就偷懶沒測。（PS：Teddy 也知道這個理由有點牽強，不過，總是反映出和資料庫有關的測試案例不容易撰寫這個問題）

- 關於自作主張（通常發生在使用者介面）的問題，程式設計師經常是用最容易施工的方式來設計，沒有特別思考到是否容易使用或是一致性的問題，也沒有將設計好的雛型先請 product owner 看一下。

針對這兩個原因，團隊討論出改善方案。

1. 在下個 sprint 規劃一個 technical story，設計一組產生資料庫測試資料的公用程式，以簡化測試案例的撰寫。
2. 著手整理 UI checklist 與 UI Patterns，降低使用者介面不一致與不方便使用的問題。
3. 對於全新開發的功能，一律增加一個撰寫 acceptance test cases 的 task，以減少由於對於需求細節不清楚造成自作主張的問題。
4. 對於所有發現的 bug，一律要先撰寫一個自動化的 unit test 來重新產生該 bug。

當然還可以列出更多的作法，但是先從這四點做起，觀察一陣子之後再依據實際實施與改善情況來決定後續的改善方法。

***

**QA 時間**

鄉民甲：UI checklist 與 UI Patterns 不是早就應該整理了，為什麼要等到現在才整理？

Teddy：由於在開發之初，我們並不確定這個系統會長成什麼樣子，因此我們並沒有為了 UI 做很多 up-front design。隨著系統功能越來越多，product owner 在操作過系統之後才慢慢有一些具體的建議。因此，依照 agile 精神此時整理 UI checklist & patterns 應該算是合理的作法。

鄉民乙：「對於所有發現的 bug，一律撰寫一個自動化的 unit test 來重新產生該 bug」這個作法不是 agile methods 的基本要求嗎？為什麼現在才實施？

Teddy：之前就已經宣導過這個觀念，但是並沒有強制要求。此次是團隊成員都同意要如此做（我們暫時先排除與 UI 相關的自動化測試），而且當有人回報錯誤時，Scrum Master 也會特別提醒要寫先寫一個自動化測試來暴露該 bug。

***

友藏內心獨白：放下中指，立地改善。

# 21 Redundancy

March 16 22:42~23:45

12/15 11:29-11:39

原文發表於 http://teddy-chen-tw.blogspot.com/2011/03/redundancy.html.

這次日本福島第一核電廠發生爆炸與輻射線外洩事件，引起了全世界的關切。台電急忙出來說明，台灣的核電廠比福島第一核電廠多一部緊急柴油發電機（每部核能機組配備兩台柴油發電機，外加一部備用機，一共五台，10 秒內可供電），2 部氣渦輪發電機（10 分鐘可供電）。此外，核一廠還有 10 萬噸，核二廠有 4 萬噸深水池，可經用來降溫或滅火。

**容錯**（fault tolerance）的一個最基本的方法就是 redundancy，A 計畫失敗換 B 計畫， B 計畫失敗換 C 計畫，C 計畫再失敗就...雙腳開開，準備投胎...。總之，設計一個容錯系統必須依據人們對於**「錯誤發生的機率，嚴重性，與可承受性」**的不同，來決定要準備多少套「備案」。

講到這邊鄉民們會想，噯呀，當初福島第一核電廠如果設計成「可以防止 10 級地震」那就 OK 了啊，因為地球上還沒有紀錄過超過 10 級的地震...或是準備個 10 台緊急柴油發電機，外加一個翡翠水庫大小的深水池，以及「原子能研究所」所使用的「防護罩」，就不會有今天的問題發生了。姑且不論這樣的「設計」是否有可能被實做出來。就算可以，蓋一座這樣的核電廠可能需要 100 兆新台幣...你出錢嗎？

***

鏡頭轉到軟體開發，從 software process 的角度來看，一個軟體專案會「失敗」有很多原因，好的 process 會想辦法從**多種角度**來避免這些錯誤發生。例如，defects （或是 bugs）對於軟體開發而言是最直接可被觀察到的錯誤，一個 defects 太多的專案要成功也很難，光是接客戶抱怨的電話就什麼事都不用做了。敏捷方法，像是 XP 就採取多重手段來避免 defects 的發生(*Extreme Programming Explained*, 2nd, p. 31），例如：

- **Pair programming**：簡單的數學：兩個腦袋 + 四隻眼睛 > 一個腦袋 + 兩隻眼睛。採用 pair programming 通常可以的到較佳的設計與較低的錯誤率。
- **Continuous integration**：衛生署提醒您：「定期做健康檢查，早期發現，早期治療」。Kent Beck 提醒您：「導入持續整合，早期發現整合錯誤，早期修復」。
- **Sitting together**：開發人員都坐在一起（在同一個房間），萬一有一個 defect 你沒看到，可能會「不小心」被你的同事找出來。
- **Real customer involvement**：開發軟體最可悲的一件事情，莫過於軟體做好了，設計的很漂亮，品質也很好，但是卻不是客戶要的。把錯誤的需求做的很漂亮，還是錯誤的需求，白搭。所以讓 real customer 參與軟體的開發，或是至少和開發團隊有很密切的互動，將可大幅減少這個問題。
- **Daily deployment**：平常家裡亂的跟豬窩一樣，突然明天有客人要來家裡，今天晚上才熬夜打掃也來不及。平常都把家裡整理得很乾淨，便可隨時都歡迎朋友到訪。軟體也是一樣，如果是在準備 release 之前才開始考慮「軟體佈署」的問題，那麼很多 defects 此時才會出現而且時間很緊迫可能會來不及在上市之前修復完成。如果可以**「每天都讓開發中的軟體保持可佈署的狀態」**，那麼就可以將軟體的品質保持在一定的水準。

有些軟體開發人員終其一生在尋找「銀子彈」，希望能有某種單一方法能夠將 defects **一槍斃命**，很可惜這種特效藥還沒被發明。Kent Beck 說：

*You can't solve the defect problem with a single practice. It is too complex, with too many facets, and it will never be solved completely. What you hope to achieve is few enough defects to maintain trust both within the team and with the customer.*

***

友藏內心獨白：「小三」算不算是一種 redundancy ?

# 22 Shared Code：讓我們變成博格人吧

03/10 21:51~23:48

12/18 22:02-20:07

原文發表於 http://teddy-chen-tw.blogspot.com/2010/03/shared-code.html.

在「星艦迷航記」系列的影集裡面可能有幾千種不同的種族， 其中最厲害的應該就屬「博格人」。博格人可以直接同化其他種族，把他們的知識直接吸收過來，形成一個集合體（宇宙中數十億的博格人都擁有一個共同的集合意 識）。由於博格人擁有集體意識因此能夠有效地協調分派工作，而博格人做事情在宇宙間也以「高品質，高效率」著稱。

XP 裡面有一個實務作法叫做 shared code (原本叫做 collective ownership)，大意就是說**專案的程式碼屬於所有開發人員所共有**。 無論程式原始的作者是誰，只要大家看到程式中有問題或是設計不良的地方，都可以動手修改。數年前 Teddy 看到 collective ownership 這個作法，直覺就想到難道這是在模仿博格人的**「集體意識」**嗎？在軟體開發中，collective ownership 的概念讓 source code 成為所有開發人員的「集體意識」，這個「集體意識」（source code）由所有的開發人員所貢獻，維護，成長。這種作法至少有以下三點好處：

1. **高品質的 source code**：由於 source code 是大家共同擁有的，因此有很多雙眼睛盯著它。當開發人員把自己寫好的 code 貢獻進去這個「集體意識」的時候，如果程式寫得太爛，也會怕別人看到會偷笑。 所以，潛意識中不知不覺就會想把程式設計的好一點，code 寫得乾淨一點。這就像就算你家裏亂的跟豬舍一樣，當朋友到你家作客的時候，總是要稍微打掃一下，留點名聲給別人探聽的道理是一樣的。其次，因為任何人有權 只要看到「集體意識」有不良成份，就可以直接改善，因此長久下來品質自然會變好。最後，當你發現原本你所寫的程式被別人改過之後，你也學到了如何把程式設 計的更好的技巧。當然也有可能是被你的同事越改越爛，此時就換你機會教育對方了。總之抱持著「互相漏氣求進步」的精神就對了。
2. **提高工作分派的彈性**：幾乎所有的軟體開發方法，都會告訴我們要挑選最優先的需求（通常是對客戶價值最高的需求）來開發。理論上是如此，但實際上卻 很難。為什麼？假設現在客戶覺 UI 很難用，要求做出容易操作的系統。此時可能絕大部分的工作都和 UI 的開發有關。如果 source code 是大家的，通常表示每個人或多或少都有能力處理不同模組的程式。所以長期下來，因為開發人員專業分工不同而導致派工困難的問題就會比較少一點 （但也許不可能全部消失）。另外，團隊也比較不會因為某人請假，而導致功能無法開發的

問題。放假的人也可以放心去休假，不用怕被電話騷擾。萬一團隊中有人 離職，對於專案的影響也會比較小。

3. **提昇開發人員的技能**：由於 source code 是大家共有的，因此開發人員有機會可以輪流開發不同性質的程式，長期下來可以提高自己的專業技能。這樣可以避免開發人員長久下來都做同樣的事情，造成工作疲乏的現象。

看到這邊，相信很多鄉民必然抱持不贊同的態度。因為大家都習慣把程式當成自己生命的某種延伸，寫程式也都會有特定的習慣，如果把對於程式的界線 或是控制權開放出去，這樣到時候被亂改一通，倒楣的還不是自己？對老闆而言，shared code 更是「違反祖制，大逆不道」的行為（來人啊，拖出去斬了）。程式碼沒有 owner，到時候出系統問題（而且一定會出問題）老闆要找哪一個倒楣鬼負責？

在傳統的心態下，同一個團隊的人，每個人負責若干的模組，界線劃分的非常清楚。好處是，由於你一直在做類似的工作，你這一項技能變得越來越強，因此看起來 你的工作效率也越來越高。另外，乍看之下責任很清楚，哪個模組有問題就找那個負責人。這樣做的缺點是，每個人矇著頭各做各的，程式的品質好壞通常沒有人關 心（除非團隊有持續做 code review，不過應該很少）。當你遇到問題的時候，由於你的同事正好也在忙著他自己所負責的模組，所以通常沒有時間也沒有很大的熱誠來幫助你，因此你需要花費更久的時

間來自行摸索。另外，開發人員萬一離職對於整個專案的影響就比較大。由於每個人的地盤劃分得很清楚，要是你不小心踩到別人的地盤，例如問了一句：「這個功 能怎麼會需要做那麼久」，那麼你很可能被反嗆一句：「你那麼厲害不然你來做好了」。最後，大家各自負責的結果，可能導致「整合」的問題沒有人管。每一個人 負責的模組都宣稱沒有問題，但是整合起來就有問題。那誰要管？

當然，要實施 shared code 也是有它的困難。大家都聽過「三個和尚沒水喝」，如何確保這份「集體意識」不會被越改越亂而又沒有人要負責？Teddy 目前的經驗如下：

1. 剛開始的時候，還是採用傳統的方法，每個人依據專長或興趣分配到若干個專案。
2. 要盡早導入持續整合，確定不同的模組可以整合在一起。
3. 要求開發人員一定要寫單元測試。
4. 等系統雛型大致穩定之後，鼓勵大家 pair programming。在這個過程中，由不同模組的創始人帶另一個開發人員，讓他可以慢慢具備接手的能力。平均起來這個過程最短通常也需要 1~2 個月。
5. 持續下去，至少達到每一個模組至少都有兩個人非常熟悉為止。

目前 Teddy 還沒有達到團隊全部成員都完全了解整個 codebase，但是慢慢地朝向這個目標邁進。這是一條有點艱辛，緩慢，且遙遠的路程，不過報酬卻是十分豐厚。這麼說好了，就好比要去「西天取經」，亦 或是「魔戒遠征隊」，出發之前大家都知道旅途十分凶險，達到目的地之後卻可以拯救世人啦。

***

友藏內心獨白：當博格人又不好，博格人當中只有 Voyager 的「7 of 9」比較討喜，而且還是在變回人類之後。

# 23 口袋不夠深

07/14 23:15~23:50

12/18 21:41-21:46

原文發表於 http://teddy-chen-tw.blogspot.com/2010/07/blog-post_14.html.

導入 Scrum（或是任何新的制度）都是很辛苦的事，特別是在缺乏「層鋒人士」力挺的情況下做起來更是「看」聲連連。幾個月前 Teddy 在讀某本 Scrum 書籍的時候，看到一句話**「A dead Scrum Master is a useless Scrum Master」**，大意是說該書作者之前因故與他所負責導入 Scrum 公司的高階主管意見不符，最後作者負氣不幹了。事後回想起來作者覺的這樣的舉動並不是很好，因此寫下了「A dead Scrum Master is a useless Scrum Master」的感想。鄉民們，牧羊犬（Scrum Master）自己都先陣亡了，那留下來的羊群還不是任人宰割嗎？所以作者鼓勵扮演 Scrum Master 角色的人要忍辱負重，不要輕言放棄。

這說來容易，但是做來真的很難。世界上同時具備**「高階主管」**與**「豬頭」**這兩種身份的人何其多啊，可是能夠「臥薪嘗膽」的 Scrum Master 實在不容易尋覓。許多有本事的人，都是很有個性滴。

***

施文彬在 1998 年 11 月的時候出了一張台語專輯叫做「按怎死都不知」，裡面有一首國語歌叫做「薪水寒」，歌詞片段如下：

作詞：武雄　編曲：江建民

風蕭蕭兮易水寒啊咧寒　**壯士一樣要上班啊咧班**
世上沒有鐵飯碗　人在江湖自然要闖一闖

***

鄉民甲：上面這兩段好像有點接不太起來？

Teddy 內心獨白：Scrum Master 們，**壯士一樣要上班**，退休金還沒存夠之前繼續熬下去吧。

***

友藏內心獨白：A dead Scrum Master is a poor Scrum Master。

# 24 我鬧，故我在

07/02 20:41~21:35

12/18 21:48-21:50

原文發表於 http://teddy-chen-tw.blogspot.com/2010/07/blog-post.html.

這幾天沒寫部落格，一方面是 Teddy 的整隻右手疼痛的老毛病又犯了（電腦一族的宿命），另一方面是白天忙著當瘋狗咬「狼」，回家後心力交瘁，實在沒力搞笑。之前看了某本書（還是某篇文章）提到，**Scrum Master 就像「牧羊犬」一樣，要保衛「羊兒」（developers）不被大野狼叼走**。Teddy 牢記於心，剛開始擔任 Scrum Master 沒幾個月，有一回 Teddy 發現大野狼蹤跡，於是費盡「喝珍珠奶茶」的力氣卯起來吠了一聲。沒想到羊沒有保護到，反倒是因為叫的太大聲而犯了**「驚駕罪」**。這在「大清」可是「斬立決」的死罪，還好現在已經是「民國」了，Teddy 命大逃過一劫，只是不小心被**「曬黑」**而已。

沒想到「草原」上居然這麼不安全，隨時都有大野狼「肖想」這些肥羊。有些大野狼嘗試一兩次失敗也就暫時作罷，最可惡的是哪種「閒閒沒事幹」，三不五時就跑來鬧一下的大野狼，這種大野狼簡稱 ~~「野狼 125」~~「野狼 X」。以下為「野狼 X」的行為特徵，請鄉民們密切注意，小心提防：

- 屬於草原上的資深動物，卡位卡的不錯。
- 自己的「本業」閒閒沒事幹，到處串門子或假好心幫助其他部落的狼，以累積自己的「狼脈」。
- 由於平常拍「狼屁」得宜，在「遠方大大野狼首領」的心目中，「野狼 X」絕對是草原上最優秀的一匹小狼~~犬~~。
- 在 email 中力求表現，把小羊兒的功勞全部搶走。
- 不定時騷擾羊群，以彰顯自己的存在（**我鬧，故我在**）。
- 配備 12 英吋的加農嘴砲。
- 立志於升官發財，只要自己能拿到一塊錢的好處，就算是把整個草原搞爛害羊兒沒草吃也在所不惜。

Scrum Master 遇到「野狼 X」也只能自認「衰小...朋友」（倒楣）。對於從小聽到大的「內除軍閥，外抗強鄰」這兩句歌詞，Teddy 終於有了深切的體認。

結論：反正 Teddy 已經曬的夠黑了... ~~宋世傑，哈， 爛命一條，早就料到。~~

***

友藏內心獨白：內除軍閥，外抗強鄰，為 Scrum 而反狼，圖羊群之復興。

# 25 Teddy 的 Pair Programming 之旅

01:00~02:10

12/18 22:45-

原文發表於

http://teddy-chen-tw.blogspot.com/2009/11/teddy-pair-programming.html.

簡單的說，pair programming 就是兩個人共用一部電腦，坐在一起開發軟體，一個動手，另一個~~開著~~ 看著，並不時提供意見 (如果沒在啃雞腿、吃泡麵、看連續劇、或睡著的話)。也許叫做 pair developing 會比較貼切一點。問題是 Kent Beck 大哥用了 pair programming 這個名稱，晚輩們不敢造次，繼續沿用。

此時 Teddy 又聽到鄉民們的怒吼：「什麼，人都不夠用了還兩個人一起寫，這麼爽。兩個人分開寫不是比較快」

的確，連小學生都會的算數：一個人一小時寫 20 行程式，請問兩個人分開同時寫，一小時可以寫幾行?

答案是：「40 行」

那麼，如果兩個人一起寫，一小時可以寫幾行?

答案是：「20 行」

所以，你的老闆一定反對 pair programming，因為生產力變成一半。

***

鄉民們有看過「實習醫生」這部影集嗎？片中的實習醫生或是住院醫生，想盡各種辦法，都要陪著「主治醫生」一起進開刀房動手術。手術室中除了若干位醫生，還有護士、麻醉師、操作醫療儀器的技術員等。耶，那開醫院的人一定都是笨蛋，因為這樣生產力變成 1/N。

為什麼動手術需要這麼多人？**因為「品質(人命)」比「速度 (生產力)」來的重要。**

此外，很多位醫生一起動刀還肩負著 ~~相互監視~~「經驗傳承 (教學)」的目的。在動手術的過程中，資深醫生會和實習醫生對話，測試他們對病況了解的程度與應對方針是否洽當。時機成熟也會讓實習醫生們動動刀 (可憐的病人...)。

當遇到很棘手的案例時 (例如，切除不良腫瘤)，有時需要動用到多位資深的醫生一起動刀。所以不一定都是資深醫生搭配實習或住院醫生。

***

如果我們看重的是「程式的品質」而不僅是「寫程式的速度」，那麼 pair programming 就很有意義了。因為 pair programming 同時具備 coding、review、testing、refactoring、learning 等目的，是一種殺傷力很強的武器。但大家也都知道實施 pair programming 不容易。為什麼？扣除大家熟知的環境因素 (~~沒有開刀房~~ 沒有大桌子、大螢幕)、配對因素 (熟手配熟手、熟手配生手、生手配生手、男生配男生、男生配女生、女生配女生、團團配圓圓、地球人配火星人...)、不信任 (覺得生產力剩下一半、浪費時間、無聊...) 以外，在 Certified ScrumMaster 課程講義中有提到一點：

**因為 pair programming 打破人的界線 (boundary)**

Teddy 很同意，咱們台灣人連「藍綠蜘蛛網」都打不破了，更別說要打破人的界線。

***

Teddy 提一下自己擔任 Scrum Master，在專案中鼓吹 pair programming 的經驗：

- phase 1，政令宣導：鼓勵團隊有機會的話儘量採用 pair programming。結果是，沒人理 Teddy。
- phase 2，自己下海：找一、二個團隊成員來和 Teddy pair。成效還不錯，不過自己累的半死。當 Teddy 停止抓人來配對之後，也沒有人繼續嘗試。
- phase 3，強迫中獎 I：告訴團隊這個 sprint 所有開發工作都要以 pair programming 的方式進行。結果，成效很好但只維持兩個 sprints。畢竟強制的方式無法持久。
- phase 4，無為而治：。從此不再特別強調，由團隊自己決定。經 Teddy 非正式估算大概只有 10~20%左右的開發工作會採用 pair programming。

說真的，到了這個階段 Teddy 已經沒輒也沒力了。Teddy 自己非常相信 pair programming 的「藥效」，但良藥苦口，團隊不肯安心按時服用，Teddy 也不能強迫 (難道真的是叔叔有練過，小朋友不肯學嗎？)。雖然在 retrospective meeting 經常會有人提出需要增加 pair programming 的時間，但神奇的是，等到真正領取工作的時候，又是各做各的比較多（也許 retrospective meeting 缺少 action plan?）。

- phase 5，強迫中獎 II：以往在估算 tasks 的時候，團隊只估算一個人做這件 task 所需的時間。也許是這個因素，導致於一個 task 通常只由一個人完成。從上週開始，團隊在 sprint planning meeting 時，就大致決定哪些 tasks (約略 80%) 需要採用 pair programming ，將其所需時間乘 2，並且在 tasks 上面寫一個 P 字（不是維大力 P），提醒團隊這是一個經過大家同意，需要 pair programming 的工作 (而且已經分配兩倍的時間)。由於才實施一週，因此長期成效尚難斷定。不過目前感覺起來這是 Teddy 試過的幾種方法中比較好的。當然，決定採用這種方法之前，Teddy 還是循循善誘一番，且經過團隊同意才實施。

***

友藏內心獨白：什麼是「藍綠蜘蛛網」？

# 26 Retrospective Meeting=許願池

11/25 22:24~23:55

12/18 22:38-22:44

原文發表於

http://teddy-chen-tw.blogspot.com/2009/11/retrospective-meeting.html.

許願池，常見於各大廟宇以及名勝古蹟之中。不管願望能否實現，丟個銅板，就可以擁有「有夢最美，希望相隨」的片刻。對於出門在外手頭不方便的鄉民而言，順手撈幾個銅板就可以換一個國民便當或是買一張回家的車票，省去攔路要錢的尷尬。Producers 和 Consumers 各取所需，王不見王。從軟體架構的角度來看，此種 decoupling 的特色，相當具有擴充性。

許願池的種種優點，做軟體的朋友們當然也發現了，就在 Scrum 中，每個 sprint（iteration，一個固定長度的開發週期，通常介於 2-4 週之間）結束時都要進行的活動－retrospective meeting－就是開發團隊的許願池啦。

Retrospective meeting 照字面解釋，就是「回顧會議」，這是 Scrum 設計用來改善開發流程的一道關卡，每一個 sprint 結束之前都要過這一關。在這個會議當中，開發團隊討論在這個 sprint 當中，哪些事情作的很好，要繼續維持。哪些作的不好，應該改善，並挑出幾個比

較重要的改善項目列出改善的行動方針。以下是幾個例子：

好的部分：

- 導入 Scrum
- 有一個公開的 product backlog
- 有訂定產品的 release plan
- 有持續整合的環境
- 有自動化功能測試案例與測試環境

有待改善：

- 電腦效能不佳 (都是 Vista 的陰謀啦)
- 沒有寫 unit test
- 沒有 pair programming
- Tasks 完成條件定義不清
- ezScrum 太慢 (這是真的有人提出...嗚嗚...)

改善行動方針：

- 長期目標：增進工作環境效率
    - Action Plan: 增加記憶體到 4G
    - Action Plan: 安裝 Vista SP2 (或是「窮得只能超頻」?)
- 長期目標: 所有 methods 至少都有三個對應的單元測試案例

    - Action Plan: 安排 1 個小時的 workshop 教導 JUnit 使用方法
    - Action Plan: 下個 sprint 每個人至少寫 3 個 test cases
- 長期目標：50% 的開發工作都以 pair programming 方式進行
    - Action Plan: 因為目前工作環境並不適合 pair programming，因此先改善工作環境。首先向公司申請 22" 螢幕、無線鍵盤、滑鼠。

由於開發流程改善是一條很漫長的路，有多長，**這麼長**... 長到你看不到底。所以，很多改善項目需要長時間的關注。例如，光是做測試這件事，就足夠玩個一年半載以上，才能小有成果。所以，為每一個改善項目訂定長期 標，然後列出本次的 action plan (行動計畫，就是下個 sprint 準備採取的改善措施)。這樣，週而復始，無限迴圈卯起來改善流程，便可達到：「人人有事做，月月有錢領」的最高境界 (一直到 stack overflow 為止)。

***

眼尖的鄉民看到這裡應該會問：這和許願池有什麼關係。是滴，以往沒有 retrospective meeting，開發團隊可能老早就發現很多問題，但是苦無 call in 管道可以暢所欲言，以至於長期悶在心中，很容易得內傷，影響工作效率。久而久之甚至導致團隊成員移民到其他公司。

有了 retrospective meeting 之後，給了各位頭家一個「嗆聲」的機會和管道。就像許願池一樣，錢丟下去，願望會不會實現是另一回事，至少聽到噗通一聲當場爽一下也好，可以減少得內傷的 機率。

Teddy 在 retrospective meeting 中看過一些比較有趣的願望包含：

- 冷氣太冷，快得 ~~H1N1~~ 感冒，影響工作效率。
- 要去參觀資訊展，~~看 show girls~~ 了解一下競爭對手的產品。
- 政躬康泰 ，皇上吉祥
- 股市上萬點
- 世界和平
- !#%!@#~

總之，每個 sprint 花一、兩個小時來了解民間疾苦、收攏人心、開開競選芭樂票、罵罵老佛爺，比起傳統無聊到爆、火氣很大、忙著找代罪羔羊的大鍋炒會議，這種 C/P 值超高的活動，還不趕快去弄一個來玩。

***

友藏內心獨白：許願前記得先丟枚銅板。

# 27 Certified Scrum Master, Day 1

12/19 09:01-09:06

原文發表於

http://teddy-chen-tw.blogspot.com/2009/10/certified-scrum-master-day-1.html.

Certified ScrumMaster 課程終於巡迴到台灣，今明兩天在台北神旺飯店三樓盛大舉辦。上課時間早上 9:00 到下午 6:00，講師為 Bas Vodde（荷蘭人，現居住在新加坡，好像有六年以上採用 Scrum 的經驗…不確定有沒有聽錯…） 與 Clinton Keith（老美，參與過許多戰鬥機與遊戲專案），都是經驗十分豐富的專家。上課的學員，包含 Teddy 共有 14 人，其中有一位是 Bas 公司的員工，來自香港（長的很像 Teddy 以前的同事，相似度 85%）。所以正港的臺灣人只有 13 位，其中某遊戲公司與某 ~~病毒~~ 製藥公司各派出四名員工參加（有富爸爸的感覺真好，一套四萬的鯊魚衣，喜歡嗎…爸爸買給你…YES）。另外有兩位曾經與 Teddy 一起上過 CMMI 課程的前輩，還有一位來自三個 I，加上 Teddy，耶，還有一~~隻~~位漏網之魚的背景沒有調查清楚。

一疊厚厚的四萬塊新台幣要拿去換一張薄的幾乎讓人忘了它的存在的證書，對 Teddy 而言 C/P 值不夠高，原本是沒有打算報名參加的。後來在某位善心人士熱心贊助之下，Teddy 欣然接受，就報名了。先講結論，第一天上課的經驗是：「**這兩位遠來的和尚還挺會念經的**」，

Bas，很幽默的，而且講英文的時 候特意放慢速度，咬字非常清楚，大家都聽的懂。

為了把這筆善款的價值提到最高，Teddy 事先把之前看過的 Scrum 書籍拿來出來複習一遍，抱持著自助火鍋「吃到飽」的精神，先寫下 10 個問題準備發問。上午的課程苦無發問機會，眼看就要吃午飯，可能會各作鳥獸散。不過不知是幸運還是不幸，午餐居然是 16 個人在一個包廂內圍著一個超大圓桌一起吃 飯，Teddy 當場傻眼…好像幫派或是政客在「喬」事情的場景。好死不死，Teddy 左手邊沒人坐，晚到的 Bas 沒得選只能坐在 Teddy 旁邊。利用上菜 空檔與吃飯時間，Teddy 用「菜英文」向 Bas 請教了幾個問題。聊了幾句之後，服務生上了第一道菜，Bas 忽然用「流利」的中文問服務生這道菜是什麼。 哇哩勒，難道阿斗仔也學會咱們台灣人「莊孝唯」這一套…一問之下，原來 Bas 的老婆是北京人（不是住在周口店的那種），他在家裡都跟他老婆講中文。

關於 Bas 中文一級棒這件事，還不是今天最令 Teddy 感到意外的。讓 Teddy 最感意外的事，居然是 Bas 說 Scrum 裡面不使用 focus factor。這個 Teddy 在一年半前從 *Scrum and XP from the Trenches* 這本書學來的方法，居然被 Bas 說成：「Scrum 沒有 focus factor 這種東東」。不過 Bas 補充說明，如果 Teddy 服用 focus factor 之後「自我感覺良好」，表示此藥物沒有副作用，可以放心繼續服用，不必因噎廢

食。各位看官，如果是你，你還啃的下去嗎？

最後，Teddy 拿出 Ken Schwaber（身分等同 XP 的 Ken Beck）1995 年在 OOPSLA 研討會所發表的論文「*Scrum Development Process*」（請參考下圖，節錄自該論文）來請問 Bas，「為什麼這篇論文裡面提到的 Scrum，還包含了 Pregame（planning & system architecture/high level design）與 postgame（closure）兩個 phases?」Bas 輕描淡寫的說：「這篇論文太舊了，現在 Ken 已經不談這兩個 phases 了」。真帥，一槍斃命。

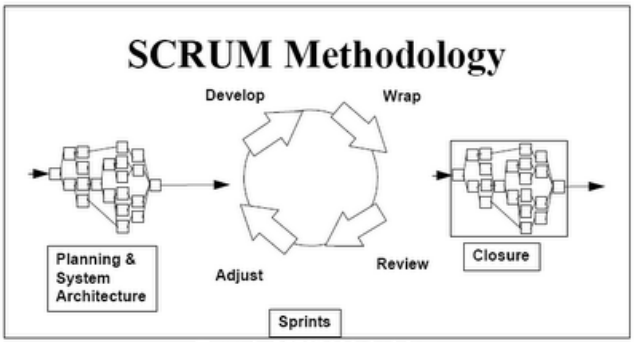


Figure 6 : SCRUM Methodology

眼尖的鄉民們可能注意到了，上圖中 Ken 稱呼 Scrum 為「Methodology」，現在似乎比較常用 Scrum Framework。Teddy 一向對於「大家來找碴」這類活動不甚擅長，沒有留意到 Ken 好心留下的線索。**再提醒一次，「Planning & System Architecture」和「Closure」已經過了保存期限，擅自服用者請自負風險。**

說真的，第一次看到這張圖，覺得好眼熟…怎麼有點 RUP 的感覺：

Planning & System Architecture = (Inception + Elaboration * 0.5)
Sprints = (Elaboration * 0.5) + Construction
Closure = Transition

這個問題 Teddy 一直苦思不解，雖然也曾經懷疑這篇 paper 的時效性，但畢竟只是猜測。在 Bas 解惑之後，Teddy 又想到古人的名言：「盡信 papers 不如無 papers」。YES，老祖宗的智慧真的不可忽視啊。

今天的課程如果真要說有什麼美中不足之處，除了某位疑似感冒的學員沒有戴口罩之外，就是「吃的東西太少了啦」。這些搞敏捷方法的人，不都是最會吃吃喝喝的嗎？怎麼現場只有咖啡、茶，以及三樣小點心（餅乾和超小蛋糕之類的），好像稍微少了一點。如果有，ㄟ，燒賣之類 的熱食，就「萬德佛」了。畢竟，古人有云：「吃完甜的，都嘛會想要來點鹹的，此乃人之常情是也」。

***

友藏內心獨白：「過期的 ~~paper~~ 東西，請自動下架」

# 28 Certified Scrum Master, Day 2

12/19 09:07-09:09

原文發表於

http://teddy-chen-tw.blogspot.com/2009/10/certified-scrum-master-day-2.html.

話說昨天的課程，Teddy 可是拼著老命去參加的，因為就在禮拜五的清晨，Teddy 突然牙痛，一整夜幾乎沒睡。還好課程內容有料，在服用兩杯咖啡之後， 成功的阻擋 Teddy 與周公的 meeting。昨天下課回到家之後，立刻直衝牙醫診所，還好平常幫 Teddy 看牙的醫生當晚有看診。醫生檢查了老半天，也 不能確定是哪一顆牙齒有問題，醫生告訴 Teddy：「我們有兩個嫌疑犯，牙齒 X 和牙齒 Y，我認為牙齒 X 的嫌疑比較大，這次先把牙齒 X 的神經給抽掉，如果回 家之後過幾天還會痛，我們再來處裡牙齒 Y。」我ㄌㄟ…靠過來一點…Teddy 口中的居民已經沒剩下幾個了，拜託醫生你要猜的準一點啊。

離開診所。從此之後，Teddy 又「少了一根經」。

以上關於 Teddy 個人健康情況的報告，請直接跳過…（鄉民甲：切，我都看完了你才叫我不要看）

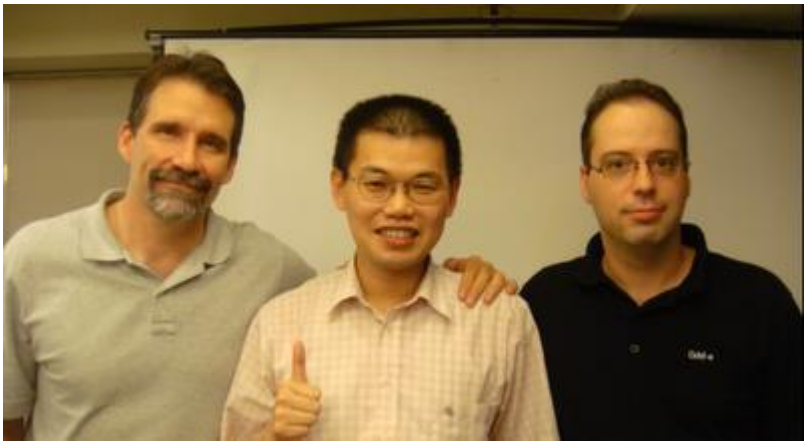
言歸正傳，第二天的課，講了幾個 Teddy 自以為早已知道其實不然

的東西：

- Product backlog
- Product backlog refinement (這個 Teddy 沒有印象在 Scrum 的書裡面有看到過)
- Sprint backlog
- Daily scrum
- Sprint retrospective
- Working agreements (這個好像也是第一次聽到)

另外，Teddy 以前看到 scaling scrum 相關的議題都直接跳過，Why… 如果你的 team 只有 ~~6, 5,~~ 4 個人，有需要擔心這個問題嗎？依據 agile methods 的精神… deciding as late as possible, 下一頁。好里佳在，今天的課程也有涵蓋這個範圍，因為善心人士可能會問 Teddy 這個問題。

課程結束之後，Teddy 拿了一本書給 Bas 簽名，並且跟兩位講師合照一張相片。



Scaling Lean & Agile Development

Good luck Promoting Scrum in Taiwan. Its needed.

Ban Vodde

夜深了，本集播放到此結束。以後有機會再整理一下這兩天上課的心得。

***

友藏內心獨白：說好的 Certified ScrumMaster 證書呢?

# 29 我變成有牌的 ScrumMaster 了

00:55~01:20

12/19 09:11-09:14

原文發表於 http://teddy-chen-tw.blogspot.com/2009/10/scrummaster.html.

話說上禮拜四、五上完了 Certified ScrumMaster 課程之後，我才想到，阿...怎麼沒有發證書呢? 這可是關係到四萬塊大洋...後來才知道，上完課程之後，會得到一個 http://www.scrumalliance.org/ 網站的帳號，登入之後填一些基本資料，然後便成「Certified ScrumMaster 候選人」(這是什麼東東?)。之後還要參加一個線上測驗 (今年 10 月 1 日才有的規定，之前好像不用)，要在一個小時內回答 61 題，通過之後才會成為 Certified ScrumMaster。


證書長這樣子

這個認證只有兩年期限 (這兩年的會員費用已經包含在上課費用裡面)，兩年過後要「續約」繳交會員費，認證才會繼續有效。~~不愧是老外，真會賺。~~

***

友藏內心獨白：證書請自行彩色輸出並拿去裝裱，以流傳後世。

# 30 傳福音

Oct. 01 19:12~20:04

12/19 11:23-11:33

原文發表於 http://teddy-chen-tw.blogspot.com/2010/10/blog-post.html.

今天搭 307 公車的時候，坐在 Teddy 身旁的陌生人 A 君，在公車快到了華中橋時突然開口。

A 君：請問這公車有到**車站**嗎？

Teddy：有。

在反射性的回答「有」之後，Teddy 心想由於 307 公車有經過「台北車站」和「板橋車站」，因此好心的 Teddy 就問 A 君。

Teddy：你是要到「台北車站」嗎？
A 君：對啊，到台北車站轉搭捷運到龍山寺。

身為「艋舺」在地人的 Teddy，忍不住告訴 A 君。

Teddy： 你要到龍山寺，在西門町下車搭捷運比較快，不用到台北車站。

A 君： 在西門町下車喔，那要搭那一線捷運？

Teddy： 板南線。

A 君： 到站的時候你可以告訴我嗎？

Teddy： Teddy：好啊，我也要去西門站。

A 君： 你該不會也要去龍山寺？

Teddy： 沒那麼巧啦。

過度熱心的 Teddy 不小心開啟了 A 君的話匣子。

A 君： 你是學生，還在唸書吧？

Teddy： 沒有啦，早就畢業了。

A 君： 結婚了嗎？

Teddy： 沒有。

A 君： 你有信仰嗎？

Teddy： 沒有。

A 君： 我是信基督教的，有人跟你傳過福音嗎？

Teddy： 沒有（此時心中出現三條線）。

A 君： 沒有？

Teddy： 應該說，我沒有興趣。

A 君： 我告訴你，耶穌（還是上帝，Teddy 有點忘了）就像空氣一樣，到處存在，隨時都可以取用，幫助你。

Teddy： 點頭傻笑...

A 君： 我以前曾經在大陸工作，遇到搶匪拿刀，拿槍搶劫，差點沒命。後來是靠禱告，把自己交給上帝，才渡過難關。

Teddy： 繼續點頭傻笑...

之後 A 君又繼續他的傳福音工作，Teddy 繼續點頭傻笑（因為 Teddy 坐在公車最後一排靠窗的位置，想跑也跑不掉啊！）。

轉眼間公車已經到了「西藏路口」，前面停了一輛 62 路公車，此時 Teddy 靈機一動。

Teddy： 如果你是要去「龍山寺」，可以在下一站「萬大路口」下車，轉搭前面的 62 路公車，這樣比較快。或是可以走路，大概 800 公尺左右。

A 君： 喔，那我走一下路好了，順便看看路上有沒有賣水果的。

Teddy： （內心獨白） YES。

***

Teddy 講上面這個故事不是要取笑 A 君，事實上 Teddy 還滿佩服他的。年輕的時候，Teddy 對於這種「侵略式」或是說「主動式」傳福音的人的確有點反感，直到有一次 Teddy 在跟指導教授聊天時，指導教授說：**「其實要去推廣 Scrum (或是其他軟工實務作法)，就好**

**像是傳教一樣，你自己認為有好東西要告訴別人，希望別人也能受益，但是想想看如果你在路上遠遠看到摩門教徒，你的第一個反應一定是趕快繞路逃開，以免他們來煩你」。**

啊，說得真是太對了。其實人都是活在習慣中，對於未知與陌生的環境都多多少少都會有點「不安」或是「排斥」感，要打破這種「慣性定律」是很難的。A 君信基督教，他相信這是很好的，所以想把「福音」傳給 Teddy，而 Teddy 的內心卻是抱持著「我不需要，不要煩我」的心態。相同的，Teddy 相信 Scrum 與軟工，也想把這種「福音」傳給身邊的人，但卻也是困難重重。

***

洪蘭女士在她的書中說過，當年她要去美國留學時，她爸爸告訴她：「妳到國外，對當地人而言就是外國人，被人家歧視，受委屈是理所當然的，不要怨天尤人，要靠自己努力證明給別人看」（大意大概是這樣）。改變現況，從來都不是一件容易的事，別人不接受，抗拒，搞破壞，都是正常的。**所以在公司內要嘗試導入 Scrum （或是任何新制度）的人，最好要有「傳教士」的精神，傳福音沒什麼好氣餒的，永遠保持正面的力量。**

西門站，陽光普照，嘴角一抹微笑。

***

友藏內心獨白：不要變成「電視名嘴」，光說不練。

# 第三部 Lean

# 31 軟體庫存

4/21 23:07~ 4/22 00:33

12/18 20:36-20:40

原文發表於 http://teddy-chen-tw.blogspot.com/2010/04/blog-post_21.html.

2009 年下半年金融海嘯發生的時候，國內某家以產品品質「堅若磐石」為賣點的電腦公司發生了創業以來第一次的虧損，據報導主要的原因在於該公司對於景氣過於樂觀，製造了一大堆的庫存。沒想到金融海嘯席捲全球，這些庫存賣不出去，造成不少庫存跌價損失。

公司的（成品或半成品）庫存水位很高，其實是一種警訊，它會掩蓋許多流程上的問題。例如，公司的行銷或是業務部門沒有做好市調或是市場研究，因此生產了一 堆沒人要買的產品堆在倉庫。而公司為了打消庫存，可能會想一些「手段」想辦法把客戶不需要的產品「塞」給他們。對公司與客戶來講，都是一種浪費。因此「**精實生產（Lean Production）**」的一項重要改善活動就是要降低庫存，期望達到「訂購生產」（客戶下了訂單才開始生產，但是要想辦法在很短的時間交貨給客戶）的目 標。

***

那麼，軟體呢？如果軟體庫存太多會怎樣？

鄉民甲：騙肖ㄟ，軟體看不到也摸不著，那裡會有庫存的問題？

以下 Teddy 列幾項軟體庫存的例子給鄉民們參考，參考：

- **部份完工的功能（partially done work）**：這種類型的庫存量在開發中的軟體系統中佔了非常大的量，例 如，沒有測試的程式碼，沒有整合的模組，沒有重整過的程式，沒有說明文件的功能，無法安裝的系統，沒有解決的 bugs。這一類的軟體只能算是「半成品庫存」，看起來好像系統開發的進度很不錯，但是實際上沒有一項功能可以真正被使用。
- **額外功能（extra features）**：這種「沒有列在需求中的額外功能」通常是開發人員「預留伏筆，以求自保」的招數。「既然都做到這裡了，以後 users 應該也會需要這個和那個，所以就順便一起做一做好了」。這種「要五毛給一塊」的情節也不算少見。
- **過度設計（over design）**：做軟體的人都知道 users 和需求都是善變的，因此傳統的軟體工程教育我們要「為未來做好打算」（搞得這些做軟體的人好像都要變成算命師）。因此，很多人習慣在軟體開發之初便要設計 一個可以「應付未來」的架構。所以，這邊增加一個 XML 設定，那邊套用一個 pattern，上面加個 MVC，底下補個 OR-Mapping，中間再應用 SOA + Cloud + Plug-ins + 龜派氣功 + 無敵風火輪 + 黯

然消魂掌 + 芋頭牛奶 ...（有怪獸，有怪獸）。最後開發出一套不到 1000 行的留言板系統，寫程式花了三天，設計架構可能花了三個月。這些過度設計所造成的庫存當然也沒推銷出去，最後放到保存期限過期。

鄉民們如果仔細端詳一下，許多敏捷方法的精神與作法都與**「消除軟體庫存」**有關，而改善軟體流程的手段，也可以從「如何消除軟體庫存」來思考。例如，如果鄉民們採用 Scrum，發現經常性的在每個 sprint 中都有幾個 stories 是在解決之前 sprint 發生的 bugs，這就有可能是軟體開發團隊在之前的 sprint 產生太多「半成品庫存」的現象。解決方法可能是要研究產生這些 bugs 的根本原因（root causes），針對這些原因提出改善方案。或是檢視每一個 story 是否有清楚定義出「DONE」的條件，這些條件是否需要修正？開發人員是否清楚這些條件且確實遵循？

軟體庫存也會隱藏流程上的問題，例如，因為流程規劃不當或是工作分派不均，造成團隊中有些開發人員已經閒閒沒事做了。這些「櫻櫻美代子」的開發人員只好沒事找事做（記得侏儸紀公園的至理名言：生命會自己找到出路），一不小心就製造了許多軟體庫存。乍看之下這些開發人員的確有很努力的在工作啊，搞不好還加班到 11、12 點。但是如果這些產出只是增加不必要的軟體庫存，那麼很

明顯地是開發流程出了問題，需要改善。**Teddy 再強調一次，庫存會隱藏流程上的問題，需要密切控管。**

***

庫存可能會漲價，也可能會跌價。對於軟體庫存而言，跌價的機會似乎比較高一點。此外，製造出這些軟體庫存也是要花成本的，這些成本都算是一種時間與金錢的浪費。鄉民們，抽空盤點一下你的系統，想辦法消除不必要的庫存。

***

友藏內心獨白：做軟體的人可真是會 ~~抄襲~~ 學習，建築的也抄，生產管理的也抄，算是現代版的吸星大法。

# 32 消除浪費 (1)：Partially Done Work

Oct. 05 22:21~23:28

12/18 20:41-20:45

原文發表於

http://teddy-chen-tw.blogspot.com/2010/10/1partially-done-wrok.html.

在「軟體庫存」這一篇，Teddy 提到可以藉由降低軟體庫存來改善軟體開發流程。這裡面的想法其實是來自於 *Implementing Lean Software Development: From Concept to Cash* 這本書。該書將「Toyota Production System (TPS)」所提到藉由減少七種不必要的浪費來提昇產品品質的精神，套用到軟體開發中，用以降低開發成本並且改善軟體品質。

這七種浪費分別是 (括號為 TPS 中所採用的名稱)，分七次逐一說明：

1. Partially Done Work (In-Process Inventory)
2. Extra Features (Over-Production)
3. Relearning (Extra Processing)
4. Handoffs (Transportation)

5. Task Switching (Motion)
6. Delays (Waiting)
7. Defects (Defects)

## Partially Done Work

東西沒做完，這些半成品無法出貨給客戶，只能放在家裡，時間一久可能會有發霉的疑慮。屬於這類的例子有「尚未被實做的需求文件，尚未 check-in 的程式碼，尚未被測試的程式碼，尚未被佈署的程式碼」。Partially Done Work 是一件很可怕的東西，首先，它會讓開發團隊有一種「進度正常甚至超前」的錯覺。

程式設計師們：　我們功能都寫好了啊（其實還沒完整測試，所以只能算「半成品」）。

專案經理：　好開心，好開心，每個人的工作都做好了。

(經過 N 天之後，專案截止日期快到了。)

專案經理：　騙肖ㄟ，系統連安裝都有問題，是誰說做好的？（Teddy 內心獨白：胃潰瘍又發作了。）

***

其次，這種半成品數量太多，時間一久，開發團隊都忘了系統中哪些東西是可以用的，哪些是半成品。到時候整個系統要整合起來，又是一大問題。再者，東西放太久沒用是會「壞掉」的，沒錯，軟體或文件也會臭酸（路人甲：那放在冰箱不就好了...）。

例如，以傳統的軟體開發方式，撰寫程式之前，一定要先做「需求分析」，然後產出一本厚厚的「需求分析書」（路人甲：人家的需求分析書怎麼只有薄薄的幾張 A4 紙，而且還是用雙空白行排版過的...XD）。假設這本分析書是用 use cases 格式撰寫，裡面包含 100 個 use cases，此時這本分析書就是「尚未被實做的需求文件」，屬於 Partially Done Work 的一種。假設開發團隊每一個 iteration (兩週) 可以實做完成 3 個 use cases ，經過半年後也才完成 36 個 use cases。剩下的 64 個 use cases，經過這半年後，還有多少是屬於「值得信賴」的 use cases 呢（不需要修改依然有效）？

開發過軟體的人應該都知道，「需求一直在變」是軟體專案唯一不變的真理，所以，從這個角度來看，「太早把需求寫完」並不是一件好事，因為太早寫完之後如果沒有足夠的資源能在短時間內實做完成，那麼這些沒實做完的需求，放久之後是會「壞掉」滴（feedback loop 太長，時間拖太久）。

東西放著讓它壞掉，你說這是不是一種浪費？

所以，在 Scrum 中，希望開發團隊要為每一項 task 與每一個 story 定義完成條件 (the definition of done)，以避免這種 Partially Done Work 的現象，這就是一種消除浪費的手段。

***

友藏內心獨白：有沒有數位冰箱可以存放尚未實做完成的文件和程式碼？

# 33 消除浪費 (2)：Extra Features

Oct. 06 22:27~23:46

12/18 20:49-20:54

原文發表於 http://teddy-chen-tw.blogspot.com/2010/10/2extra-features.html.

**Extra Features （多餘的功能）**是軟體開發七種浪費行為中，最嚴重的一種浪費。為什麼？道理很簡單，如果某項功能連寫都不需要寫，那就不會因為要開發該功能而產生「Partially Done Work、Relearning、Handoffs、Task Switching、Delays、Defects」這些浪費的行為了。與其做出客戶暫時不需要的功能，而日後需要「時時勤拂拭，勿使惹塵埃」(開發與維護這些功能使其正常工作)，還不如採取「本來無一物，何處惹塵埃」的策略，不是省事多了。

路人甲： 那乾脆什麼都不做，不是最好。

Teddy： 我也想什麼都不做啊，可是到了這種極端，你的銀行帳戶也會跟著「本來無一物，何處惹塵埃」。畢竟鄉民們大概都還是屬於俗世中人，尚未修煉到不食人間煙火的境界，所以「該做的，還是要做」。至於哪些是該做的，哪些是不該做的，就要靠各位的「智慧」去判斷了..XD。

***

舉個例子，Teddy 最近看到 Kay 買了 iPhone 4，覺的好心動，不由自主的便稍微研究了幾款智慧型手機，包括 HTC Desire、HTC Desire HD、HTC Desire、Nokia N8...，每一款都好想買。於是，Teddy 心中的小惡魔就不斷地找理由說服 Teddy，「智慧型手機好棒，可以為你的生活帶來極大的便利，趕快買一隻，趕快買一隻」。

在激情過後，Teddy 發現不管是那一款手機，目前暫時都不需要，先用 Kay 的 iPhone 4 就好了。Teddy 想起「住左邊，住右邊」這齣連續劇，裡面有一句對白，「公家有，用公家，公家沒有用朋友」，這樣最省的啦。如果 Teddy 當時衝動之下跑去買了不需要的手機（Extra Features），那就是大大的浪費了（Teddy 內心獨白：如果口袋夠深的話，好想給它浪費一下...XD）。這種情況在逛大買場的時候也常常發生，就算是事先寫好了購物清單，一不小心還是買了不必要的東西回家。

***

各位鄉民以前求學的時候，或多或少都有受到傳統軟體開發方法的洗腦，認為軟體專案事前的（需求）分析做的越多越好，系統要很有彈性，以便應付日後未知的新增需求（請參考「你的軟體架構有多軟」），搞得這些做軟體的人好像「算命師」一樣，要能預測未來。

這種**「過於彈性的軟體架構」**或是程式設計師基於**「追求使用最新技術的爽度所幻想出來的需求」**，就好像你的下一隻智慧型手機，下一雙高跟鞋，下一件衣服，下一台 3D 電視，~~下一個女朋友~~，下一個 XXX 一樣，過早投資，通常都是一種浪費。

問題來了，古人有云，要「未雨綢繆」，難道事先準備有錯嗎？不是都說「有備無患」，怎麼現在「綢繆」和「有備」都變成一種浪費啦，那是不是說每天只要躺著在家裡睡覺就可以當選啦。~~食神~~ 軟工，我真是猜不透你啊！

長話短說，把握幾個敏捷精神：

- 最高境界：本來無一物，何處惹塵埃。不需要的功能，就不要做。
- 撐到最後一刻逼不得已時再做決定。問氣象局一個禮拜後的天氣，準確度可能只有 50%。若是問「今天」的天氣，準確度可能有 90%。如果問「昨天」的天氣，準確度就有 100% 啦。所以，愈晚做決定，通常比較不會出錯。這就是為什麼人人都可以當「事後諸葛亮」的原因。
- 採用演進式設計。
- 創造支援「逐步成長」的開發流程，開發環境，與軟體架構。

- 對於已經開發完成的功能，要「時時勤拂拭，勿使惹塵埃」。這跟養小孩一樣，要嘛就不生，既然生了，不管長相如何，身體是否健康，都要好好照顧。

講完收功。

***

友藏內心獨白：以上所言，不適用於「老子就是錢多，就是要促進消費，你咬我啊」這種類型的人。

# 34 消除浪費 (3)：Relearning

Oct. 09 22:20~23:41

12/18 20:57-21:04

原文發表於 http://teddy-chen-tw.blogspot.com/2010/10/3relearning.html.

今日談談「~~七武器~~ 七浪費」之三：**Relearning**，也可以稱為「rework」。如果一次就能做好，就不需要「rework」，所以 「第二次（或以上）」的 work 當然就是一種浪費。但是為什麼不直接說 rework，而要咬文嚼字的說 relearning？

由於軟體開發事實上就是一種**「學習新知的過程」**，因此減少重新學習已知的知識，就可以減少浪費，所以特別用 relearning 來強調「軟體開發實為學習新識」的特質。例如，團隊中有老鳥跟菜鳥，菜鳥被分配到開發「用 Java NIO 實做非同步檔案讀寫」的某項功能。老鳥對於此功能的技術十分熟練，但是被分配到該工作的卻是菜鳥。於是：

- 菜鳥先花了一整天搞懂什麼是 Java NIO。
- 菜鳥再花一天實做非同步檔案讀寫與撰寫單元測試。
- Tester 花了兩小時測出菜鳥所寫的程式有 5 個 bugs。
- 菜鳥解了其中的 3 個 bugs，另外 2 個看不出來問題再哪裡。

- 菜鳥請老鳥幫忙做 code review，老鳥只花了 10 分鐘就找出 8 個 bugs (原本尚未解的 2 個 bugs 外加 6 個 tester 沒測出來的 bugs)。
- 菜鳥將這 8 個 bugs 解完。
- 經過這一番過程，菜鳥獲得 10 點經驗值，菜鳥一級升級為菜鳥二級。

以上整個過程花了 6 個工作天，如果可以縮短學習的時間，便可減少時程的浪費。

***

大量的 relearning 對於軟體開發是一件「很傷」的事情，但是卻很難完全避免 relearning。團隊中幾乎不可能每一個人的知識與經驗都相同（如果能夠找「博格人」來開發軟體就太好了...什麼，不知道博格人是什麼？來人啊，拖出去...看...*Star Trek*），因為「派工（分工）」的關係，又不可能把每件工作都交給最懂該工作的人來實做，因此便會造成 relearning 的現象。

另外，人是很「健忘的」，以前做過的事，曾經發生過的問題，日子一久，經常會忘記，因此相同的問題可能一再的發生，這也是 relearning（此時如果能找到「曾子」來開發軟體就好了，因為論語有紀載：「曾子不貳過」，這種人才找來寫程式最好）。

***

光知道 relearning 是一種浪費，那麼要如何減少 relearning 造成的浪費，這就傷腦筋了。請鄉民們想一下，有沒有哪一個 agile practices 是可以減少 relearning？知道答案的人，請將答案寫在明信片上，來信請寄「台北郵政 5408 號信箱」，前五名答對者本節目將提供神秘小禮物... XD

其實很多 agile practices 都和減少 relearning 有關，例如：

- 自動化測試：將程式正常行為的「知識」紀錄在自動化測試程式中，當自己或別人不小心產生 bugs 時（破壞程式正常行為），能夠透過紀錄下來的知識立即發現問題（就是這些自動化測試案例），如此便可減少 relearning 所花的時間（可參考第 471 頁「需求分析書中最重要的資訊是什麼？」這篇）。
- 採用 Design by Contract [1] 或是 assertion。理由類似自動化測試，這些技術在 runtime 都可自動驗證程式的行為是否符合原本的預期。
- Pair programming：師徒制是傳遞知識最好的方法之一，雖然過程辛苦了點。

族繁不及備載，請各自發揮。

***

友藏內心獨白：程式設計師徵人條件：具有博格人和曾子的性格。

## 備註

[1]. B. Meyer, Object-Oriented Software Construction, 2nd, Prentice Hall, 2000.

# 35 消除浪費 (4)：Handoffs

Oct. 10 22:39~23:40

12/18 21:09-21:14

原文發表於 http://teddy-chen-tw.blogspot.com/2010/10/4handoffs.html.

「七浪費」之四：**Handoffs**，英翻中的解釋是「交接」。為什麼交接是一種浪費？舉個例子，有點年紀的鄉民們應該都知道一種叫做「喝水傳話」的遊戲，玩遊戲的人排成一排，主持人先把謎底（通常是一句話）告訴第一人，然後這個人嘴裡要含著一大口水，頭朝上把剛剛主持人告訴他的那句話傳給第二個人，依此類推一直傳下去。通常最後一個人所聽到的內容和原本那句話早已差了十萬八千里。

一件工作轉過一手之後，所剩下來的知識若能有原本的 50%就算是很不錯了，所以，如果這件工作轉了 5 手，那麼就只剩下 3%的知識：

第一手 50% →第二手 25%→第三手 12%→第四手 6%→第五手 3%

知道的東西越來越少，就需要花更多的時間才能把事情做好，所以交接是一種浪費的行為。

***

「交接」，從另外一個角度來看，又可以稱為：把「屎虧（不想去碰的工作）」丟給別人去做。相信大部分的鄉民都有把工作交接的別人，或是從別人那裡獲得交接工作的經驗，通常這種經驗都很差。

交接者：　我已經使盡渾身解數把知道的告訴你了，沒學到算你笨。

被交接者：　東西這麼多，交接時間那麼短，你又講的這麼籠統，我怎麼可能學得會。一定是存心想要留一手不告訴我。（Teddy 補充：套句學弟常常說的話，「學長又沒交接給我！」）

一件工作丟來丟去，經驗傳承下去的比例也就越來越少，所以接到「屎虧」的人就要花更多時間去完成這件工作，這種經驗流失與需要耗費額外時間（包含把工作交接給別人所花的時間）才能完成工作當然也是一種浪費。

***

那麼，要如何減少交接的浪費呢？書上提了幾個方法：

- 降低交接的次數（Teddy 內心獨白：這不是廢話嗎...XD）。

- 組織 **design-build** 團隊（就是 Scrum 所說的 **cross-functional teams**）。關於這一點 Christopher Alexander 早就提出類似的建議，請參考 Teddy 另一篇「Architect-Builder」。
- 使用「寬頻」溝通方法。盡可能用面對面溝通（face-to-face）取代文件式溝通。（Teddy 內心獨白：Apple 聽到您的心聲，所以開發了 FaceTime。眾鄉民們還不趕快乖乖掏錢去買一隻 iPhone 4。）
- 儘早且頻繁地釋出部份（先期）作品以便於獲得回饋。這句話 Teddy 要解釋一下，原文比較長：「*Release partial or preliminary work for consideration and feedback--as soon as possible and as often as practical.*」假設團隊中有 programmers 和 testers，如果 programmers「自認為」程式已經做完了，所以把程式交給 testers 去測試（這就是一種「交接」）。但是如果在開發的過程中，programmers 從來都沒有跟 testers 討論過，那麼經過交接之後，testers 很可能不知道要如何測試，或是說無法測出真正的問題，甚至會測錯（都算浪費）。如果 programmers 在開發的過程便可以和 testers 討論，那麼便可以減少這樣的問題。

最後補充一點， Teddy 認為用來減少 relearning 的 agile practices，像是 pair programming 或 collective ownership 也都可以減少 handoffs 所造成的浪費。

***

友藏內心獨白：國慶日還沒休息，夠意思了吧。

# 36 消除浪費 (5)：Task Switching

Oct. 19 21:54~22:42

12/18 21:20-21:23

原文發表於 http://teddy-chen-tw.blogspot.com/2010/10/5task-switching.html.

「七浪費」之五：**Task Switching** (工作切換)。等一下， 為什麼工作切換是一種浪費？現在這個高科技時代，不管是人還是機器，不是都強調要具備「多工 (multitasking)」的能力嗎？所以：

老闆： 每個工程師手邊同時至少要有三個專案，一個快結案，一個進行中，一個剛開始。這樣才可以把工程師超到過勞死...的邊緣。

消費者： 都什麼時代了，iOS 4.x 之前的版本居然膽敢不支援多工。難道我就不能邊玩 game 邊寫 mail 嗎... XD。

學生：我最強，我可以邊上課，邊睡覺，邊聊天，邊玩手機，再加上邊吃泡麵邊啃雞腿。

學過作業系統的人都知道，工作切換便會形成 **context switch**，只要有 context switch 就形成浪費。對電腦而言，context switch 可以是很 routine 的工作，對人腦就不同了 。如果手邊的工作類型是屬於「不

需要大腦也可以搞定的工作」，例如挑大便或是拔草，那麼工作切換所造成的浪費就比較少。但是，對軟體開發而言，大部分的工作卻都是需要「絞盡腦汁」的工作（路人甲：真的嗎？！）。如果時常切換不同的工作，有可能最後花在 context switch 時間比真正去做事的時間還來的多。引用一段書中的話：

*When knowledge workers have three or four tasks to do, they will often spend more time resetting their minds as they switch to each new task than they spend actually working on it.*

問題來了，實務上，你的老闆絕對沒那麼好心，讓你只做一件工作。所以，工作切換畢竟是難免的。書中舉一個例子說明如果減少工作切換的浪費。假設你的軟體已經釋出，因此你同時必須要面對「維護舊軟體」與「開發新軟體」這兩件工作。以下是書中提到減少工作切換浪費的方法：

- 讓兩個人（或兩個小組）每個月（或每個 iteration）輪流負責開發與維護的工作。
- 每天早上花兩個小時讓整個團隊處理維護的工作，其餘時間專注於新功能開發。
- 仿效「急診室」的作法，先將每一個客戶所提出的維護需求加以分類，只立即處理「緊急」的維護需求（理論上只有很少量的維護需求是屬於這一類的），然後每一週或兩週在看

看有哪些維護需求是真正需要處理的（有時候客戶的維護需求放久了會自動消失...^o^）。

- 把所有不同客戶所使用的軟體版本都放在單一的 code base，然後每週或每一個 iteration（定期）釋出一個更新版本。

***

結論：在實際的生活中，task switching 是難免的。但是請牢記以下兩點：

- 不是塞越多的工作給員工就可以得到越多的產出，有時候塞了太多的工作，員工只是在不同的工作間「空轉」而已，最後一事無成。
- 就算是需要 task switching，也有比較好的工作切換模式。例如，不要每隔 5 分鐘檢查一次 mail，那只會害你分心。最好是採用**「定時定量」**的方法，例如每天早上一到公司的時間，下午 1:30 以及下班前。

***

友藏內心獨白：為什麼總是在忙碌的時候，不斷收到 MSN/Skype 的訊息？

# 37 消除浪費 (6)：Delays

Oct. 21 22:09~23:22

12/18 21:25-21:30

原文發表於 http://teddy-chen-tw.blogspot.com/2010/10/6delays.html.

七浪費之六：**Delays**，Teddy 覺的用原本 TPS 的用語 **Waiting** 好像比較容易了解。軟體開發的 delays 情況有：

- **等待某人有空**：例如，等待 DBA (資料庫管理員) 幫忙設定資料庫；等待 technical writer 幫忙寫使用手冊；等待「隊友」幫忙一起解 bugs；等待公司招募到新人。
- **等待資源**：等待測試設備；等待購買的軟硬體設備到貨；等待從 Amazon 買的書寄到台灣。
- **等待某項知識**：例如，等待 Product Owner 明確的告訴開發者所有的需求細節；等待 team leader 告訴你 bugs 要如何解，程式要如何寫；等待新人搞清楚專案狀況。
- **等待核准**：例如，等待老闆核准專案進行；等待法務審核合約；等待客戶畫押需求；等待批准需求變更。
- **等待功能或產品完成**：例如，等待系統分析師生出完整的軟體架構；等待系統設計師畫出全部的 UML 設計圖；等待程式全部完成以便測試；等待程式通過測試。

根據書上的說法，開發人員**每 15 分鐘就會做出一個重要的決定**（一秒鐘幾十萬上下？！），如果開發人員對於他正在做的工作有足夠的知識（例如，知道客戶到底要他做什麼），或是當他有問題時馬上就可以獲得解答，那麼開發人 員就可以快速做出（較高品質的）決定。如果開發人員缺少足夠的知識或是沒有人可以回答他的問題，那麼聰明的人類只會有三種反應：

- **停下手邊的工作，試圖去找出答案**：原本的工作被中斷，如果找答案的過程很冗長，造成 task switching 的浪費。
- **停下手邊的工作，找其他事情來做**：柿子挑軟的吃，原本的工作可能變成 partially done work。最慘的就是不斷地找新的事情，沒有一件事完成，一直到 stack overflow 為止。
- **猜測可能的作法，硬著頭皮繼續做下去**：蠻幹，最後的成果可能不是客戶所需要的，需要 rework 或 relearning。這個現象常常在很多「有理想，有抱負外加有創意」的開發人員身上發現。

如果找答案的代價很高或是程序很繁雜（例如要聯絡客戶，協調其他部門的人），那麼屬於「宅男」，「宅女」類型的開發人員因為害羞與偷懶等原因， 通常會選擇後面兩項作法。在情況允許下，如果瞎耗一段時間不做事又不會被發現，開發人員也可能會執行一個

「空迴圈」不斷地等待下去，期待答案自己出現。這 些行為都是一種浪費。

***

消除 delays 的方法很多，書上提到一個最簡單的做法：

*Complete, collocated teams and short iterations with regular feedback can dramatically decrease delays while increasing the quality of decisions.*

Teddy 幫鄉民們把上面這句翻成白話文，就是「採用敏捷方法就對了啦」。

PS：Teddy 氣象小叮嚀：鄉民們，不必再等待不可能的「颱風假」，除非你的公司在宜蘭縣。

***

友藏內心獨白：等待「發薪水」和「下班」應該不算浪費吧...XD。

# 38 停掉生產線

3/26 21:39~23:05

12/18 21:56-

原文發表於 http://teddy-chen-tw.blogspot.com/2010/03/blog-post_26.html.

「品質是內建，不是外加的」這句話應該算是常識了，現在大部分的人讀起來已經覺的沒什麼特別。但是在 19 年前，當 Teddy 還是個青澀少年時，在「由 C 到 C++：物件導向革命」這本書中第一次看到這個觀念，卻是覺得很新奇：

哇，原來產品的品質在做好之後便已經決定了，更多的測試並不會增加產品的品質，所以要提昇產品的品質便需要提昇產品製作流程與流程中的每個活動的品質。

聽起來很有道理，但是，哪有可能！程式寫好不交給別人測試，能有這樣的勇氣直接拿給客戶用嗎？「暈倒死」的「藍色死亡畫面」看多了，電腦上面 reset 按鈕也按到手軟，再聽到「如何寫出零錯誤的程式」這類的話，心裡都會偷笑。久而久之，軟體有 bugs 算是正常，用到沒有 bugs 的軟體才是要懷疑是不是看到鬼。長此以往，開發人員對於 bugs 見怪不怪，因此對於如何提高軟體品質這檔子事的

警覺性，也越來越低了。在業界中屢見不鮮的例子：

PM： 這個功能做好了嗎？

Programmer： 做好了，沒有問題。

PM： 我剛剛隨便測了一下就發現很多 bugs 耶。

Programmer： 你咬我啊...

夜深人靜時，相信開發人員的內心偶而也浮出小小的吶喊「我到底是在開發功能還是在開發 bugs？」。

軟體工程的主要目的之一，就是要探討如何做出高品質的軟體。達到這個目的有很多不同的手段，Teddy 在念研究所時，曾經修過 PSP (Personal Software Process) 這門課，PSP 希望開發人員可以藉由多種不同層次的 review（例如 design 和 code）、紀錄一狗票的資料 (以分鐘為單位的 time log, 新增幾行程式碼, 修改幾行程式碼, 開發過程中所有發生的 bugs...) 、以及定期提出改善方案等手段來提昇品質。立意良好，但光是聽起來就覺的有點「違反人性」，窒礙難行。光是修課過程中寫 10 支小程式就快被搞死了，玩玩 Sony PSP (PlayStation Portable) 還差不多，怎麼可能用 PSP 來開發軟體...。

由於程式是人寫出來的 (程式產生器也是人寫的)，所以各種軟體品質改善方法或是開發流程，無不強調如何減少人為錯誤。因為很多

programmers (至少在台灣是這樣吧) 已經養成「差不多先生」的心態，因此 Teddy 認為無論採用何種方法，首要之務便是以**「符合人性的方式」**提昇 programmers 的 ~~羞恥心~~ 警覺心，讓他們慢慢覺沒有 bugs（或是很低的 bug rate）是常態，而非不可能。要怎麼做？

Teddy 最近在讀 Lean 和 Toyota Production System (TPS) 的書，其中有一個作法叫做 **Stop the Line--停掉生產線**，Teddy 覺的滿有趣的，也很想來試試。Stop the Line 大意就是說：**如果一發現不良品，就馬上停止整條生產線，一直到找到問題的根源 (root cause) 為止**。乍看之下覺的日本人怎麼這麼笨，整條生產線停工，那成本有多高？但是仔細一想，如果不徹底找出產生不良品的原因，這些生產出來的不良品都是浪費。更糟的是，如果不良品流入市面，商譽的損失與後續維護成本更是高的嚇人。

Teddy 好想在公司裡面裝一個類似救護車或是消防車上面的「警示燈」，當一發現 bug 的時候就按下這個燈，啟動之後還會發出「偶一、偶一」的聲音。此時大家都放下手邊的工作，一起來徹底找出造成 bug 的根本原因，並討論如何避免下次再發生同樣的 bug。這樣做的好處至少有以下兩點：

- 提高警覺：由於一發現 bug 需要暫停所有人的工作，所以大家都不會希望這個 bug 是自己造成的。因此在施工的時候，便會想辦法提昇軟體的品質。例如，主動找人幫你做 code

review、執行 pair programming、多做 unit testing、實施 test-driven development（TDD）等等。甚至如果有人覺的 PSP 有幫助，也可以採用 PSP。總之，改善方法不限，由團隊自行決定。

- 提高共識：所有人一起找出原因並且討論改善方案，因此對於如何發生錯誤以及如何避免再次發生都會有比較高的共識。

可惜，Teddy 的團隊並沒有專屬的辦公室，工作環境中還有其他團隊成員。如果真的裝了一個警示燈，以目前找到 bugs 的頻率，可能會把其他人吵死吧。

***

友藏內心獨白：或許可以考慮每個人發一個像是在「伯朗咖啡」點餐時所拿到的「振動器」來取代警示燈。

# 第四部 加班

# 39 加班，加班，我愛你

06/16 22:09~23:25

12/12 16:19-16:32

原文發表於 http://teddy-chen-tw.blogspot.com/2010/06/blog-post_16.html.

為了讓非軟體產業的鄉民們對於台灣軟體業有所了解，請容許 Teddy 簡述一下許多軟體專案進行的方式。

**專案成員**：

- 夢想與天一樣高或瘋狂想賺錢的老闆，或兩者皆是。
- 標準配備為十二英吋口徑加農嘴砲的業務（永遠搞不懂，也不需要懂他在賣什麼東西）。
- 只會畫甘特圖與 WBS 的專案經理（可能擁有 PMP 證照）。
- 該死的工程師（實際上真正做事的人）。

**專案進行方式**：

- 老闆或是業務靠著「冥想」、「政商關係」、「人脈」、以及「眼睛被蛤蠣蓋住的客戶」加持之下，接到專案。至於專案內容是什麼，拜託，這一點都不重要。

- 在老闆一聲令下，決定了專案的時程。訂定時程其實很簡單，依據 ~~專案的大小~~ 收錢的多寡，只要遵循 一、三、六、九、十二 這個原則就 OK 了。在這幾個數字中隨便挑幾個用來，一、三、六可能是最常使用的，不管專案要做什麼，反正老闆都認為三個月就應該可以完成。
- 專案經理依照聖上（老闆）的聖旨，以緊迫釘人的方式，督促工程師趕上進度。
- 工程師在還沒搞清楚要做什麼之前，三個月的期限已經到了。

看到這邊鄉民們可能會以一個疑問：「啊，三個月到了案子沒做完，那誰負責」。

老闆： 專案經理，案子進行的如何？

專案經理： 進行的很順利，工程師每天都加班到很晚，功能都做的差不多了，

專案經理： 只剩下測試還有「幾個」小 bugs 要修。

老闆： 測試就不用了，讓客戶自己測就好了。這樣還需要多久？

專案經理： 大概.... 三個月吧！

老闆： 老闆：一個月之內做完。

***

鄉民乙： 那專案到底多久做完？

Teddy： 這不是重點，結案才是重點。

鄉民乙： 鄉民乙：沒做完也可以結案？

Teddy： 當然。

鄉民乙： 那要如何結案？

Teddy： 這是秘密，講出來會動搖國本。

鄉民乙： （消音）。

***

Teddy 今天想提的，其實是「加班」這件事。每個工程師都知道一件事，就是**專案的「時程 (schedule)」就像 X 鐵的火車時刻表一樣，僅供參考之用**。有的專案更誇張，居然膽敢訂出「反攻大陸」時程表。

老闆：給你一個班的兵力，兩年內完成反攻大陸的偉業。

老闆要做夢鄉民們也犯不著吵醒他，但是又要保護自己在夢醒時分不會遭遇不測。因此，雙方發展出一套遊戲規則：「加班」。

老闆內心獨白：我的員工好認真，每天加班到 12 點，假日也來上班，

統一大業指日可待。

員工內心獨白：我每天都加班，假日也來加班，東西做不出來也不能怪我。啊不然你是要怎樣。

**有了「加班」這個方便法門，老闆不用去花腦經去提昇管理績效，員工也不用花腦經去改善做事方法。反正，「這是主流」啊，順著潮流走就對了。**（PS：花腦筋是很違反人性的一件事，這也是為什麼周星星的電影那麼多人看，還看了不只一遍。）

***

Agile methods 提倡「每週工作 40 小時」，就像 TPS (Toyota Production System) 提倡「零庫存」是一樣的，目的就是要**把問題暴露出來**。每天加班事情還是做不完，大家已經習以為常，不用花腦經去想，多方便。**那，如果不加班要把事情做完呢，這就難了**：

- 我們專案時程根本亂訂，業務搶到一堆案子，做一個賠兩個。
- 我們的 bugs 太多了，所以事情做不完。要準時下班，就要想辦法減少 bugs。
- 程式亂寫，後續根本無法維護，也找不到願意維護的人。
- 工程師看不到未來，無法成長，人員流動太快。
- 沒有自動化測試，改一行程式錯 10 個地方也沒人發現。

上面這個列表要再加上個 100 條也是輕而易舉的事，這些問題平常都被掃到地毯下，沒人願意去管。**反正有「加班」這個萬用武器，何必花時間與力氣去做些吃力不討好的事。**

要拯救一個快散掉的軟體開發團隊，首先想想，要如何可以讓團隊成員不再加班也可以達成進度。不管這個團隊有多少問題，這都是一個不錯的出發點。

***

友藏內心獨白：Teddy 年輕的時候也是幾乎天天加班到很晚，案子還不也是做的一團亂。

# 40 非加班不能搞定之台灣經濟奇蹟 幕後無名英雄

02/11 00:30~01:54

12/16 10:08-10:17

原文發表於 http://teddy-chen-tw.blogspot.com/2010/02/blog-post.html.

今天聯絡了一位以前公司的同事，告知他 3/11 日有一個 Scrum 講座，看看他有沒有興趣去聽。這位前同事目前從事嵌入式系統領域的工作，而他的碩士論文又是研究 XP，所以 Teddy 認為他應該會有興趣。

不料，前同事的第一個反應卻是：「這個活動在禮拜四下午喔...如果是晚上或是假日，我還可以喬一下時間...」奇怪了，Teddy 心裡想，這種活動辦在上班時間不是很正常嗎，晚上和假日誰還要來聽啊？

前同事目前工作專案團隊包含他老闆一共有六個人，平常他都忙到 11~12 點左右才下班。「這在業界很正常啊」... 前同事說....聽到這裡，Teddy 也了解前同事的難處。工作這麼忙，怎麼可能利用上班時間聽這個和工作看起來沒什麼關係的 Scrum。對一般公司而言，除非是 Linux kernel 或是 Intel 新 CPU, 晶片組介紹，公司才可能放員工在上班時間出來聽課吧。想在上班時間外出聽軟體工程相關的課程...想太多...不予通過...

不過，Teddy 還是很厚臉皮的想把他騙出來。

Teddy： 你可以把資料寄給你老闆，找他一起來。

前同事： 我老闆更忙，每天都在開會。

Teddy： !@#%~@@zzzZ

這個對話最後以交換 Facebook 帳號作為結束。

***

以下是 Teddy 聽來據說是真人真事的故事。某個「X 碩」的員工，每天下班都搭最後一班的捷運。有一天下班後他在台北車站換車，在月台上遇到一位不認識的老先生，這位老先 生突然對他說：「年輕人，你看起來氣色很不好，要多注意身體」。連不認識的陌生人的看得出來「X 碩」是很操滴。

等一下，看到這邊「X 碩」的員工可能會不服氣的說：「我們都很少加班啊...因為過了 12 點才算加班...」所以，不算操，OK 的啦。

***

相信各位鄉民們或多或少都有親身經歷或是聽過這些「台灣經濟奇蹟幕後無名英 雄」的加班史。想當年 Teddy 還是一尾活龍的時候，也是經常過了 12 點才下班，有時候還直接睡在公司，晚上還煮稀飯當宵夜和同事一起分享。現在的 Teddy，以每天 6:30 下班為努力目標，已經變成人人喊打的「台灣經濟奇蹟幕後的無名米蟲」。的確，如果從待在公司的時間來看，9:00 AM ~ 6:30 PM，扣掉中午吃飯的那一個小時，Teddy 每天「只」上班 8.5 小時，遠遠低於台灣人的平均工時。這如果是在學校修課，Teddy 已經被死當了。

這可能是 Teddy 身體太虛了，如果某天上班剛好一整天都在 pair programming，做了大概五個小時就快要「縮缸」了，如果 pair

programming 超過六個小時，回家就可能要貼「撒隆巴斯」了...因為手酸背痛 。

根據一些來源不可考的馬路消息，平均一天能有五個小時有效率，專心的工作時間就已經很了不起了。

**鄉民甲：亂講，那我每天在公司待那麼久是在做假的喔...**

關於工時和工作成果的討論，長久以來已經有太多文獻可以參考，在這邊 Teddy 就不再說明（偷懶一下）。不過鄉民們如果剛好是程式設計師的話，可以做一個實驗，如果我們狹義的單獨計算每天「專心寫程式」所花費的時間，平均下來這個時間應該也是五～六小時左右吧。等一下，不要把「無意識 coding、無意識 debugging、開批鬥大會、MSN 打屁、上 Facebook、收發 email、看新聞、看 mobile01、線上購物、搞網拍、裝潢房子、看小說、玩 game、打小報告、研究旅遊行程、在走廊上聊八卦、泡咖啡、喝下午茶、上廁所、看股票、使用網路 ATM 或網路銀行、填假單、申請費用、發呆假裝思考與休息眼睛」的時間也算進去喔。

除了加班以外，是不是有可能藉由改善做事情的效率或是減少做錯事情的機率，來提昇競爭力呢？Teddy 覺的 Scrum 是一個不錯的入門方法，門檻低，很容易上手，可以分階段逐步改善軟體開發團隊的

做事方法並且提高生產力。有空可以來聽聽或是自己找資料看。

***

友藏內心獨白：現場有沒有請 show girls?

# 41 過勞死之軟工無用論

Sept. 28 22:20~Sept. 29 00:08

12/16 10:18-10:30

原文發表於 http://teddy-chen-tw.blogspot.com/2010/09/blog-post_28.html.

這幾天有一則新聞，某 29 歲任職於「南邊來了個 亞 啞巴」之高科技公司員工「疑似」加班過度而過勞死，公司的反應卻是極力想撇清關係。據水果日報報導：「南邊來了個啞巴科技副總說，**公司對較有能力的工程師及主管採責任制，上班時間由員工自己判定，也積極宣導員工盡量休假、上下班時間正常。** 」

這種說法是不是說：

- 「公司對**較有能力**的工程師及主管採責任制」，這麼說起來責任制是一種光榮耶。不用懷疑，當年日本**「神風特攻隊」**一定也是採取「責任制」。
- 不屬於責任制的員工們，立正站好，好好反省一下，這表示你們「比較沒有能力」。不過別難過，老子云：柔弱生之徒。恭禧各位沒能力的非責任制員工得以僥倖存活下來（Teddy 內心獨白：學生算不算責任制？）。

- 「上班時間由員工自己判定」… 現在是怎樣，「毅中各表」嗎？

Teddy 內心有種感覺，台灣公司老闆普遍存著「加班就是好員工，不加班就是 ~~不孝~~ 不認真」的心態。反正：「別人的孩子死不了」，離職再找人就好。反正老子有錢，叫你作你就做，做不完，反正「上班時間由員工自己判定」，你就好好地「自己判定」一下。

***

Teddy 從事軟體業（ㄟ... 在硬體公司寫程式算軟體業嗎？），常常聽到其他人說：「啊，軟體工程在業界行不通啦」，「兩天的訓練課程要一萬塊，這麼貴（Teddy 內心獨白：那請問多少錢您老大才覺的合理？）」。**這種打從心底認為「開發軟體沒什麼學問」的心態，當然會做出「沒什麼學問的軟體」出來**。偏偏這些老闆們又認為這些開發軟體的員工屬於「有能力工程師」（這不是自相矛盾嗎？沒學問的事情應該不需要有能力的人來處理吧。），反正只要員工們好好的「自己判定一下上班時間」，任何問題都可以搞定，Yes，you can。

軟體工程到底有沒有用，Teddy 舉的例子。軟體工程裡面有一種軟體設計的方法，稱作 Design By Contract（DBC），DBC 的原則十分簡單，但威力卻很強大。基本上寫程式就是「我 call 你的 code (API or

method)，你 call 我的 code」。在這種 「彼此互相 call 來 call 去」，的互動當中，程式可以區分為兩種不同的角色：「caller」和「callee」。

- Caller：呼叫別人以獲得服務的人（有點繞口），在 DBC 中稱為 client。例如，我去銀行請行員幫我開戶，我就是 client。
- Callee：被別人呼叫並提供服務的人，在 DBC 中稱為 supplier。例如，我去銀行請行員幫我開戶，行員就是 supplier。

有了這個觀念之後，接下來的規則就很簡單了。每個程式（姑且就先想成一個 method 好了）都應該有它自己的 precondition 與 postcondition。

- Precondition：想要執行這個 method，那麼這個 method 的 precondition 必須要成立才可以。 這就好比日劇「大和拜金女」裡面松島菜菜子對於擇偶條件一定要是「好野人」一樣，這就是「要跟大和拜金女交往的 precondition」。
- Postcondition：執行這個 method 之後，該 method 保證一定會成立的條件。例如「大和拜金女」為了吸引「好野人」跟她交往，可能訂出「聯誼一次可獲得香吻一枚」這種 postcondition。

鄉民甲：這和程式設計有何關係？

問的好，今天先談一下 precondition 的好處。 在開發軟體的時候，其實 programmers 經常做了很多「大膽的假設」與「小心的檢查」：

- 這個傳進來的參數會不會是 null 呢...ㄟ，不知道，那就 if (str != null) {do something}。
- 讀取一個由其他程式所產生的檔案，萬一檔案格式不對怎麼辦？

很多人直覺的反應就是**「既然不知道別人傳進來的資料是否正確，那就自己再多檢查一遍啊，反正多檢查一次也不會怎樣」**， 這種作法就叫做「defensive programming」，看起來很不錯啊，但這卻很可能是一種造成加班的原因。為什麼？想像一下，如果你是銀行行員，有人拿「黃金」來「存錢」，你需要 先幫顧客把黃金換算成現金，然後再把這些現金存到顧客的戶頭嗎？。大部分的銀行應該沒有提供這麼感恩的服務吧，所以，關於存款這個服務，應該會有類似的 preconditions：

- 必須是（新台幣）現金（其他貨幣或是有價物品一概不收）
- 必須是真鈔（否則報警）
- 必須是現行流通的版本（拿舊版的新台幣就可以不用處理）
- 貨幣本身必須完整可辨識（被火燒過或是被蟲蛀的紙鈔請先找調查局鑑定）

有了這些 preconditions 之後，這個 supplier 的實做 (行員的服務內容) 就變得很明確，講成白話文就是，如果需要寫一隻支援行員的程式，那麼這隻程式的「輸入」就很明確，也就不需在程式裡面做一些有的沒的檢查，需要寫的 code 自然也變少了 (大體上是這個味道，想了解細節還是要看一下 *Object-Oriented Software Construction* 這本書)。

***

扯了這麼一大段，回到主題。記得 Teddy 曾經看過某本書，書中提到有效率跟沒效率的 programmers 其生產力好壞可以差到 10 倍。如果企業文化就是「開發軟體沒什麼學問」、「軟工無用」、「加班萬歲」、在這種環境的 programmers 那可能去管什麼「DBC、Exception Handling、Refactoring、Agile、SCRUM、Design Pattern、Unit Testing...」(就算 Teddy 雞婆想要去教免錢可能還被對方嫌沒空)。反正，這一堆「沒什麼學問的東西」老闆，主管也不懂啊，只有**「程式能動才是王道」**，其他的任何事都不要來煩我，因為我是「較有能力」的工程師，我只需要「上班時間自己判定」這個萬能的武器就夠了。

感覺好像現代版的「神風特攻隊」。

***

友藏內心獨白：如果被公司宣判屬於「較無能能力的員工」是不是就不用加班？！

# 42 我可能不會 18:30 下班

December 16 01:50~02:57

12/16 10:31-10:34

原文發表於 http://teddy-chen-tw.blogspot.com/2011/12/1830.html.

23:40 就寢但是翻來覆去都睡不著，想說起來看個電視培養睡意，看了 30 分鐘還是睡不著，那乾脆寫篇部落格文章好了。這個禮拜行程滿檔，週一至週二連續兩天去榮總看醫生，週三至週四下午去面試，週五（幾個小時之後 Orz...）又要回榮總看照胃鏡的報告，其他空檔的時間 Teddy 幾乎都在寫書。說「寫書」是比較好聽一點的講法啦，Teddy 其實只是把部落格的文章挑一些出來整理與分類一下，之後再花點時間排版與修飾一下，就這樣而已，也不是什麼真的從無到有生一本新的出來（將軟體的 reuse 精神發揮到極致也是一種力行環保）。

進度報告一下，現在已經整理了大概 100 頁左右，應該不會被鄉民們火力攻擊了（你，還看別人，就是在說你，槍放下來先...XD）。

***

最近民視剛播完一部很熱門的偶像劇叫做「我可能不會愛你」，Teddy

借用這個劇名的「梗」，這禮拜的兩次面試 Teddy 很 ~~白目~~ 坦白的告訴對方說「**我可能不會 18:30 下班**」。原本以為會遭受到猛烈的砲火攻擊，沒想到對方居然異常體貼的表示「理解」。奇怪，這怎麼和 Teddy 從網路上看到的印象完全不一樣啊，難道是 Teddy lag 了嗎，還是 Teddy 在做夢（誰來賞 Teddy 一巴掌..XD），亦或是對方採取**「欺敵政策」**？總之這兩次面試讓 Teddy 學到不少東西。

這幾天 Teddy 在網路上 search 了一下，發現很多台灣的公司都要求員工要加班，而且奉「責任制」之名可以名正言順的不必給加班費，有些公司甚至要求新人 22:00 之後才可下班。但是，員工也不是笨蛋，因為大家都看過「侏儸紀公園」，都知道「生命會找到出路」。此外，員工的中文程度也很好，都知道「上有政策，下有對策」這句話。所以呢，好啊，你要我晚走，OK 啊，我就白天的時候打混摸魚，晚上吃完飯之後混到 8 點多再回來，東摸摸西摸摸 10 點左右再發幾封 e-mail，搞定...最好回家之後晚上 2-3 點起床尿尿再順便發幾封 e-mail（Teddy 內心獨白：幹麼那麼累，寫個批次程式自動發 e-mail 不就好了）。哇哩勒**靠**...近一點看，**工時這麼長**，好認真的員工喔，好心疼，好心疼，來，阿姑親一個...XD。

請問現在是比賽「挑大糞」嗎？工時長看誰挑的多？拜託眼睛上面的「蛤蠣肉」拿下來先好嗎...員工有沒有實質的產出不去管他，光去看工時有什麼用啊。

公司要養人才，還是養豬？要養豬的話請要求員工 24 小時都待在公司以方便定時灌食，這樣才養得肥。好的人才 8 個小時做完奴才 1 個月的工作量，你老大還在嫌人家「我可能不會 18:30 下班」。那...套句 mobile01 上面鄉民常講的一句話...幫不了你。

***

老師在講你到底有沒有在聽啊，沒有啊（~~丟手榴彈~~ 丟筆），「加班，加班，我愛你」，「非加班不能搞定之台灣經濟奇蹟幕後無名英雄」，「過勞死之軟工無用論」，這幾篇回去大聲朗誦 10 遍先。

***

友藏內心獨白：還好長達一個小時的筆試最後沒有出現（擦汗...）。

PS：櫃台和 HR 怎麼都那麼正啊，有篩選過喔...（迷之音：都是電腦挑的啦...）。啊，歹勢，沒圖沒真相...偷拍器材忘記帶出門...XD。

# 第五部 洗腦

# 43 學習犯錯

September 04 21:36~23:08

12/10 20:30-20:52

原文發表於 http://teddy-chen-tw.blogspot.com/2011/09/blog-post.html.

在古早，古早以前，許多人的求學過程中，都經歷過那段「少一分打一下」的不愉快時光。拜託，考 90 分已經很了不起了，為什麼還要被打 10 下？現在回想起來，覺的實在是很可笑。這種「追求 100 分的心態」，某種程度也迫使學生不得不「~~造假~~ 微調數據」。記得當年 Teddy 在念五專的時候，需要上化學實驗課以及電子實習課。這兩門課，都需要在實驗室做實驗。其實這些實驗也沒什麼了不起的，就是把書本上教過的一些化學現象或是電子特性，到實驗親自動手確認一次。Teddy 依稀記得當時在做實驗的時候，常常會遇到量測出來的數據和課本上面的理論值有一段頗大的差距。「**照道理講**」，這時候應該是要好好的檢討一下自己做實驗的步驟或是測量方法有沒有錯誤，找出問題的原因，但是 Teddy 當時並沒有認清到這些實驗的重要性，心裡只想著「~~準時下班~~ .... 準時下課」，所以 Teddy 就犯了「全天下學生都會犯的錯」... 把實驗數據改成「長得比較接近理論值的樣子」... 順利過關。

其實「~~竄改~~...嗯嗯...調整實驗數據」這件事，也不能全怪 Teddy...想一

想，一班有 50 個學生，只有一個助教在帶實習課。做實驗遇到問題的也不只 Teddy 一個人，要是每個人真的都「按照道理」做事，非得把實驗結果失敗的原因給找出來為止，那麼短短三個小時的實驗課絕對是不夠用的。學生們為了「大局著想」，讓助教可以準時下課，實驗室可以準時關門，更重要的一點，讓自己可以準時畢業，只好「想辦法」把實驗給做出來。

想在回想起來，為什麼上這些實習課時，助教們總是期望學生「**第一次做實驗就上手**」？如果學生很認真的做實驗，但結果失敗，這次實驗拿到 0 分，這樣有誰願意承認自己實驗失敗？反正「**正確答案在書上都有啊**」，把書上的數據拿來「調整一下」就 OK 了啊，反正助教想看的也只是這些正確答案。

***

在某一個 sprint planning meeting 中，有一個 story 長成這樣：

*As a user, I can modify common attributes of multiple objects.*

使用者同時間可以在畫面上選多個「物件」，然後可以一次修改這些物件的「共同屬性」，使用者按下確定之後系統會更新這些物件在資料庫中的值。在討論這個 story 的時候，有人問到：「**要不要將多個**

**物件的修改放到一個 transaction 當中**」？一開始有人贊成，有人反對，經過一番討論之後，最後大家同意「要用一個 transaction 將多個修改動作包起來」，原因是因為，如果沒有把多個修改動作都放在一個 transaction 當中，那麼假設使用者選了 10 個物件，修改之後 6 個成功，4 個失敗，可是使用者的原意就是要修改這 10 個物件啊，所以最後使用者還要自行去把那 4 個修改失敗的物件再挑出來修改一次（等於手動 retry），這樣感覺不太方便，所以最後大家同意用一個 transaction 把多個修改動作包起來。

這個 story 做完之後功能正常，一直到這個 sprint。幾天前 Teddy 閒閒沒事做了一個測試，在系統中建了 1000 個物件，然後一次修改這 1000 個物件。由於這 1000 個物件的修改被包成一個很大的 transaction，隨著這個功能的執行，資料庫的幾個 tables 都被 lock 住了（每一個修改動作都需要花一點時間，因此整著 transaction 花了很長的時間才完成）。由於這個系統還有其他幾十個 threads 在執行，也會去更新資料庫，因為需要更新的 tables 被 lock 住了，因此這些 threads 被暫停執行，最後看起來整個系統像是「當機」一樣。

此時 Teddy 才想起來，10 幾年前 Teddy 用 Java 開發第一個 multiple users 的進銷存系統時，也遇到過類似的問題，只是年代久遠，上個 sprint 在討論這個 story 的時候，完全沒有意會到把多個物件的修改放到一個 transaction 中會有這樣的問題。

要怪誰？怪當初建議這個 solution 的 developer？還是怪 Teddy 當時沒有想到這個問題？還是要怪測試這個 story 的人有沒好好測，沒把這個問題給找出來？

***

沒什麼好怪的，做軟體就是這樣，「千金難買早知道」。其實把「多個物件的修改放到一個 transaction 中」這件事情本身是沒有錯滴，問題在於在該系統中，每一個修改動作夾帶了另一個修改動作（有點玄...總之就是一個修改動作同時會改到兩個 tables）。這原本也不算是個問題，但是第二個修改動作會隨著「物件數量的增加而變得很慢」。當物件數目很少的時候，整個 transaction 在幾秒內就做完了。但是，當物件數目到達 1000 的時候，每修改一次需要約 25 秒，因此整個 transaction 花了 7 個多小時才完成，整個系統後端的 threads 也被「卡」了 7 個多小時。

昨天在讀 *Agile Testing* [1] 這本書的時候，看到 p.25 提到 Have Courage 這一段，其中提到：

*We need courage to let ourselves fail, knowing that at least we'll* ***fail fast*** *and* ***be able to learn from the failure****. After we've blown an iteration*

*because we didn't get a stable build, we'll start thinking of ways to ensure it doesn't happen again.*

***We need courage to allow others to make mistakes, because that's the only way to learn the lesson.***

傳統上，有些人做事會有「多做多錯，少做少錯，不做不錯」的心態。如果一個敏捷團隊沒有容許犯錯的空間，也沒有從錯中學習的機制，那麼這樣的敏捷團隊就很難稱得上是敏捷團隊。

做軟體是沒有「正確答案」可以抄的，所以「**答錯也要給分**」，因為你已經排除了一種不可能的作法。

最後以 Linda Rising 女士關於 Agile software development 的看法作為結尾（節錄自 Linda Rising 在 Agile 2011 研討會 Keynote speech 的投影片 [2]）：

- Fail early, fail often.
- Fail fast, learn constantly.
- Failure *IS* an option.
- Without failure how can learning happen?
- Perfect is a verb.

***

友藏內心獨白：做軟體可不能只是偷改測試案例，把錯的改成對的就沒事了...XD。

***

## 備註

[1]. L. Crispin and J. Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*, Addison-Wesley, 2009.

[2]. http://program2011.agilealliance.org/event/93ce733f9316d5122da7bb531763c704 , 可下載 pdf 檔案。

# 44 為什麼不問問題？

November 17 20:40~21:40

12/10 20:57-21:18

原文發表於.
http://teddy-chen-tw.blogspot.com/2011/11/blog-post_17.html.

2009 年 10 月下旬 Teddy 去參加了 Certified ScrumMaster 兩天要價四萬新台幣的訓練課程。這麼貴的課，上課之前 Teddy 就想著要如何利用短短地兩天「多撈一點」，於是 Teddy 列了一個問題清單，大概有 10 來個問題，準備利用上課的機會來釐清這些問題。

到了上課那天，Bas（講師之一）準備了一個小板子，要上課的學員們如果有問題的話可以寫在紙上貼到後面（好像是一個人可以問三個問題...有點忘了）。此時 Teddy 就把準備好的問題列表拿出來，同桌的其他學員了嚇了一跳，想說怎麼有人那麼認真....^O^。

不過這不是重點，重點是，後來吃中飯的時候 Teddy 剛好坐在 Bas 旁邊，所以利用吃飯機會使出 Teddy 苦學多年的「菜英文」把準備好的問題跟 Bas 請教了一遍，算是頗有收穫。

***

上禮拜六 Teddy 擔任北科大 ezScrum 團隊所舉辦的 Scrum 分享活動的其中一場的講師（這個句子有點長...XD），Teddy 雞婆請主辦單位在活動前幾天詢問參加者如果有問題可以先將問題寄給主辦單位，這樣 Teddy 就可以針對參加者的問題先準備一下。

**考考各位鄉民，報名的有 80 個人，請問主辦單位收到幾個問題？**

有沒有 40 個？沒有。有沒有 10 個？沒有。有沒有 4 個？沒有。

有沒有 **0** 個？有！！！

雖然當天還是有不少人問了好幾個問題，但是問題五花八門有些問題「來的太突然」，Teddy 當下也不一定回答的很好。

***

會問問題真的是一門學問，Teddy 以前也是很不會問問題。記得念專科的時候，Teddy 要幫社團去借電腦教室（借一整個學期）。這間電腦教室是某位助教負責管理的，社團的前任社長（大 Teddy 一屆的學長）跟這位助教關係很好，所以之前已經成功借了一年。Teddy 要去借場地之前，前任社長與 Teddy 的對話：

前任社長： 你要怎麼跟助教借電腦教室？

Teddy： 就問助教說：「**助教，這學期電腦教室可以繼續借我們使用嗎？一週兩個晚上**」。

前任社長： No, No, No。你要這樣說：「**助教，這學期那兩天晚上電腦教室可以借我們？**」。

Teddy： 那有什麼不一樣？

前任社長： 你的問法，是 Yes/No 的問句，助教腦中思考的回答是 Yes or No，萬一他說 No 這樣還要幺回來就很麻煩。我教你的問法，是直接問他「那兩天可以借我們」... 雖然有點強迫中獎的感覺，但是我們都這樣問了，對方就比較不好意思回答「不借」。

Teddy： ~~沒想到學長你「貌似忠良」，內心卻是如此奸詐狡猾...XD~~。嗯嗯...學長您真是太厲害了，又學了一招。

***

上禮拜六 Scrum 活動中有鄉民問 Teddy：「一個團隊同時負責 5 個案子要怎麼 run Scrum？」。結果 Teddy 反問他：「如果你是國民隊的老闆，你會讓王建民早上跟洋基比賽，下午跟紅襪比賽，晚上再對上大都會隊嗎？小王年薪這麼高，不是應該一個人扛很多案子，好好利用他？」。思考完 Teddy 問的這個問題，原本鄉民所問的問題答案應該就很清楚了。什麼，還是不知道...來人啊，鞭數十，驅之別院...XD。

***

**肯開口問問題就代表有在動腦筋（切記，開發軟體最難的一件事就是讓大家動腦）**，就算一開始問的都是一些很白目的問題，還是要勇敢的問出口，問到了，學到了，就是自己的。多問，多看，多學，就會進步。

***

友藏內心獨白：結尾好像太勵志了一點...XD。

# 45 風範

October 27 21:17~22:46

12/10 20:05-20:17

原文發表於 http://teddy-chen-tw.blogspot.com/2011/10/blog-post_27.html.

今天收到一封信，讓 Teddy 想起了以前唸書的一件事。有一天指導教授找 Teddy 談話，提到有一個廠商要找實驗室合作一個關於網路安全的案子，想找 Teddy 一起參與。Teddy 由於某種原因，不想參與這個計畫，於是 Teddy 當下就告訴指導教授說「我不想參與這個計畫」。

可能有鄉民會想，「**得罪了方丈，還想走**」。但是，Teddy 和指導教授認識了 10 幾年，非常了解指導授認的為人，也因此 Teddy 才敢坦白地直接拒絕。假設 Teddy 不想參與這個計畫，但是又不好意思拒絕（很多台灣學生可能會這樣），於是勉強參與了計畫案，但是又做的心不甘，情不願的。在這種情況下，自己學習的動機沒了，相信案子也不會做的太好。而且日後還可能心裡一直記恨，說當年被ㄠ被迫去做 xxx 案子。

在唸書的過程中還有發生許多次跟指導教授意見不同的爭論，指導教授都很包容 Teddy 有不同的意見。如果 Teddy 跟了一位只希望學

生唯命是從的指導教授，相信 Teddy 的書應該是念不下去了。

（Teddy 內心獨白：學長有練過，學弟們不要學...學長不是鼓勵你們「抗命」，還是要乖乖聽話喔...XD）

***

Teddy 的修養就遠遠比不上指導教授。以前在跟學弟討論研究內容的時候，由於 Teddy 年紀「稍長」，開發與設計軟體的經驗相對的比學弟們來的多一點，經常等不及聽完學弟們的想法就發表自己的意見，或是太直接的質疑學弟們的作法。就算是出於一片好意，想要節省學弟們嘗試錯誤的「時間」，但是某種程度也剝奪了他們嘗試錯誤的「機會」。

以前有人問過 Teddy，從工程師變成 team leader 最困難的是什麼？Teddy 不假思索的就回答：「放手」。工程師出身的 team leader，尤其是好的工程師，很多都會有「**不放心別人做出來的東西**」的通病，搞到最後自己變成團隊的瓶頸，什麼事都卡在自己身上，團隊也無法成長，變大。

關於放手這件事，Teddy 也還在學習當中。

N 年前 Teddy 還在做 e-learning 系統的時候，學會了一個 HR（人力資源）領域的詞叫做「**接班人計畫**」，大概的意思是說，每一位主管都應該要尋找或是培養自己的接班人，最高境界就是達到「這個位子沒有我也可以（因為平日就已經積極地幫公司培養了接班人）」。剛開始聽到的時候覺得很扯，哪有可能，哪一位主管不是想踩在別人的身上往上爬，一旦佔住位子更是不可能輕易離開，這種把自己幹掉的計畫也太天才了吧。

***

Teddy 的第一份工作在一家剛開始不到 10 人的小公司寫程式，剛進公司的前幾年公司所有的人都是沒有 title（職稱）的。老闆朱先生認為公司每個人都很重要，認真做事比 title 來的重要，不要被 title 限制住你所能夠做的事情。雖然 N 年後 Teddy 離開了這家公司，但是對於朱先生的處事態度還是十分佩服與懷念，從他的身上學習到很多作人，做事的氣度。

***

同學留言：你還是老樣子，不只髮型，個性也沒變 XD。

# 46 傻的願意相信

March 12 16:33~18:02

12/11 16:55-17:07

原文發表於 http://teddy-chen-tw.blogspot.com/2011/03/blog-post_12.html.

Teddy 當年還在唸書的時候當過兩年研究所 OOAD（物件導向分析語設計）課程的助教，其中一項最主要的工作就是 review 學生的作業。老師一共用過兩本教科書，分別是 Craig Larman 所寫得 *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*（好長的書名）以及 Michael Blaha 和 James Rumbaugh 所寫的 *Object-Oriented Modeling and Design with UML*。每年的作業都很相似，基本上讓學生自己選一個題目（自行決定 1 人 1 組或是 2-3 人 1 組），題目也可以自己決定（找不到題目的人可以跟老師索取參考題目），內容大小以能夠在一學期完成 2-4 個 use cases 為限。雖然這門課叫做 OOAD，可是每一位修課的學生不是只有寫寫 use cases，畫畫 UML diagrams 而已，而是包含 coding 和 testing，最後把系統做出來（套用「吃掉你自己的狗食」這個原理，自己的設計自己實做出來才會知道自己設計的有多爛...XD），所以修這門課還滿花時間的。

由於題目是學生自己挑的，因此助教在改作業時就有點辛苦了，因

為：

- 要先弄懂學生所挑題目的那個 domain。
- 要看學生寫的不清不楚的 use cases 然後還要告訴他們為什麼這樣寫不好。
- Conceptual model 和 design model 也要幫忙看。

作業改到後來感覺好像是助教在幫學生做題目...XD...基本上就只有 coding 和 testing 的細節比較沒關注到，其他關於需求分析與系統設計的內容 Teddy 在自己能力範圍內算是盡心盡力幫忙 review 了。有時候還會讓學生們覺的這個學長（就是 Teddy）太「雞蛋裡挑骨頭了吧」，設計本來不就是「這樣也可能，那樣也行」嗎，反正最後我程式寫得出來，軟體可以跑就好了，學長你幹麼管那麼多？

說實話當年 Teddy 內心覺的這些「刁民」真是不知好歹，能找到 Teddy 來幫你們 review 算是你們上輩子燒好香，居然還敢有「悖逆之心」，不好好給我回去重寫作業還在這裡做垂死的掙扎，希望 Teddy 被你們說服....（Teddy 內心獨白：等你們練成了絕地武士的催眠功夫再說吧）

其實基本上這門課的作業其實滿簡單的：

- 寫出 problem statement 並畫出 context diagram。
- 畫出 use case diagram 並寫出 use cases。
- 畫出系統架構圖。
- 畫出 Conceptual diagram showing concepts with association and attributes。
- 畫出 System sequence diagram 並寫出 contracts。
- 畫出 Design class diagram with associations。
- 說明一個設計中最重要的 class。
- 列出 test cases 並說明其中一個最重要的 test method。
- 程式畫面。

分四次把上面所列的這些東西在最後學期結束時都學會了，也就及格了。很多學生其實不太能夠適應，或者是說「不相信這一套方法可以奏效」，總是有些「自認聰明」的人，覺的自己對於開發軟體「很有經驗」，不必拘泥於書上所講得方法。但是也就是因為如此，這些人很有可能修完課之後，還是對於所謂「OOAD」方法不了解。

***

對於「OOAD 」方法不了解有什麼大不了的，還不是活得好好地。沒錯，說實話 Teddy 現在開發軟體也沒有按照書本中所描述的步驟一步一步慢慢來，但是由於 Teddy 曾經用力花時間去了解這套方法，

因此當遇到自己不熟悉的領域，無法靠「直覺」來設計系統的時候，這些在 OOAD 課程所學到的方法就變成一種「參考模式」，用來解決不熟悉問題的參考模式。

Teddy 的指導教授在課堂上常常告訴學生要「**傻的願意相信**」書上教的方法，自己要先「願意相信」這是可以奏效的方法，嘗試照著去做，而不要在學會之前就先急著去否定這些方法。聽起來好像很簡單，但是要咱們「聰明的台灣人」傻傻的一步一步按照規矩辦事，說真的還真是不太容易。

回到 Teddy 上一篇的主題 「Ten-Minute Build」，如果今天 Teddy 跑去跟別人講，你的專案要達到 Ten-Minute Build 的要求喔，Teddy 可能會被當作瘋子，因為根本沒人相信。對方可能會說，「**哎喲，這都是書本上的說法啦，你們不懂業界的實況，所以.....**」

不知道為什麼，Teddy 對於某些人所講的話就特別相信，Kent Beck 就是其中一位，因此當 Teddy 遇到「build 時間太長」這個問題時，就到書上看看 Kent Beck 怎麼說，嗯，他說「Ten-Minute Build」，Teddy 就先「姑且信之」，於是 Teddy 開始思考一連串所謂的「流程改善」以便達到這個目標。改善內容包括了：

- 更新 build server。

- 挑選現有可快速執行的 test cases。
- 讓 test cases 執行的更快。
- 想辦法找出此次異動的程式，並只執行受影響的 test cases。
- .....（其他）

總之這個「Ten-Minute Build」就是努力的目標就對了，如果繼續抱持著「不相信」的心態，那麼就只會一直維持現狀。基本上 Teddy 所知道的軟體工程知識應該不會比任何一個資工科系畢業的研究生要多太多，很多都是修完軟體工程這門課之後就有的常識，但是 Teddy「傻的願意相信」的決心可能比很多鄉民要來的強，也許就是這一點小小的信仰驅動著 Teddy 把「理論」慢慢變成「~~食物~~ 實務」。

***

鄉民們可能三不五時會聽到什麼 agile methods、SCRUM、continuous integration、pair programming、automatic acceptance testing、incremental design、shared code 等等一堆有的沒的名詞，如果真的把它們當成「名詞」，那就真的與你無緣。如果把這些當成「動詞」，著手去做，也許多多少少能「撈到一些好處」。

想當年 Teddy 聽到 GOMS（請參考「歪批 GOMS (1)」）這個作法，第一時間也是覺的「這個比扯鈴還要扯的方法怎麼可能用在真實世

界的軟體專案？」，但是一旦花點時間去了解之後，身邊又多了一把「小扁鑽」可以使用，偶爾拿出來應付一些特殊場合還是不錯用滴。

***

友藏內心獨白：這可能就是日本地震後隆起的柏油路居然比台灣的道路還要平坦的原因...XD...每次想到路平專案 Teddy 就想笑...苦笑...

# 47 造船的目的

2010 August 06 22:05~23:23

12/11 17:18-17:26

原文發表於 http://teddy-chen-tw.blogspot.com/2010/08/blog-post.html.

最近部落格績效不彰，產量銳減。主要原因是 Teddy 白天都在寫軟體使用手冊，寫了一天的文字，腦細胞早就虛脫了，手和肩膀也酸痛不已。回家除了努力保持呼吸與抽空看海綿寶寶之外，那還有精神搞笑。

前天看了洪蘭女士所寫的「理直氣平：勇於改變才會進步」這本書，書中收錄洪蘭女士發表在報章雜誌上的小短文，每一篇各自獨立，很符合敏捷精神。這本書講了一堆做人做事的道理，照常理推斷應該是很無聊的書，但是 Teddy 卻覺得十分有趣，也很有勵志效果。

Teddy 今天想談一下這本書第 151 頁的一篇文章：「**失敗比不曾試過好**」。 這篇文章的大意是說：

洪蘭的朋友的兒子，出社會工作還不到 2 年，就決定把工作辭掉自行創業。這位朋友很擔心，因此請洪蘭去找他兒子談一下。朋友的兒子說上班壓力很大，老闆喜怒無常，覺的自己連靈魂都買給老闆

了。洪蘭提醒他創業的風險，對方說：「沒有失，哪有得？人總是去闖一下，才不負少年頭」。

最後洪蘭回頭去勸她的朋友，讓他兒子自己作主並自己承擔後果。相信日後他兒子會告訴他「失敗的感覺還是比不曾試的感覺好，錦衣玉食無法彌補不能做自己的痛苦」。

接下這篇文章最後兩句話是 Teddy 最喜歡的：

**停留在港口的船是最安全的，但那不是造船的目的**

***

Teddy 是在睡覺前讀這本書的，看到上面這句話讓我當天失眠了。每個人從小到大，在父母的期許，社會的壓力，鄉民的七嘴八舌，以及各種莫名其妙的原因逼迫之下，不斷的競爭，到底是為了什麼？

在可以為社會貢獻一點力量之前，這一段「造船」的過程，充滿了考考考，補補補（考試、補習）。有多少人為了那一點點分數，和好朋友反目成仇。為了補習，浪費了生命中多少美好的事物。以前大部分的人可能認為大學畢業就算是完成第一階段的「造船運動」，現在則是「滿街是碩士」，也許不久之後連接電話的總機都有博士文

憑。這些都姑且不談，好不容易把船造好之後，這些船都在幹麼？是找一個好的港口，然後停下來混吃等死，還是航向大海去探索未知的世界。

N 年前台灣股市正熱的時候，很多優秀的人才到竹科當「科技新貴」，領了豐厚的股票，穿著無塵衣做著只需要高中或大學程度就可以勝任的事。 現在想想 Teddy 以及系上畢業的學弟們，學了許多軟體工程的技術，投入社會之後，真正在工作上施展的又有多少呢？這當然是一個很複雜的問題，每個公司，團隊都有一套自己做事的方法或是固有文化，無論是好是壞，靠一個人的力量去改變是極度困難的。剛畢業的新鮮人也許有極大的熱情，但是不斷碰壁之後，就算現在這個「港口」再爛，總還是比大海安全，所以自己這艘船也就停了下來。什麼軟體工程，卡早睡卡有眠。

（以上廢話一言以蔽之就是「向下沉淪」）

偶而看看這種「勵志小品」可以喚起心中那一絲絲快熄滅的向上提昇的力量。就算是船現在停在港口，也要勤作維修，這樣萬一真的要開出海，才不會還沒出港口就沉了。

***

路人甲：以上和軟體開發有何關係？

Teddy：請注意這本書的副標題：「**勇於改變才會進步**」。Kent Beck 告訴我們什麼？*Embrace Change*。Mary Kynn Manns 和 Linda Rising 告訴我們什麼？*Fearless Change*。 Scrum 告訴我們什麼？（請鄉民們自己找答案）。

***

友藏內心獨白：再引用書中的一句話：「其實大家都知道改革的關鍵在社會每個人的觀念，人的觀念改了，制度自然就改了。只是觀念是天下最難改得東西，它需要時間」。

# 48 發語詞，無義

2010 07/30 22:02~23:15

12/11 17:27-17:36

原文發表於 http://teddy-chen-tw.blogspot.com/2010/07/blog-post_30.html.

「追求卓越」這四個字，這幾年在台灣已經被搞爛了，好比當年的「反攻大陸」一樣，正式宣告由「動詞」變成「發語詞」(**國中課本有教：發語詞，無義**)。什麼「XX 卓越計畫」,「XX 卓越中心」，五年五百億，說穿了都只是「編列預算」與「消化預算」的藉口而已，基本上算是現代版的「奉旨行搶」，咱們這些領死薪水的上班族，辛辛苦苦所繳的「一咪咪」稅金，就這樣被這些計畫給消化掉走了。

嗯嗯，言歸正傳。敏捷方法精神當中，有一點就是要「追求技術卓越」，這一點看起來是那麼的合理 （有哪個軟體開發方法會說自己要追求擺爛的？），哪麼的自然，以至於讓人幾乎「忘了它的存在」（靠...邊站，又變成發語詞了）。

其實這一點真的是很重要，這也是所謂的「**工匠精神**」。Teddy 認為在台灣很多試圖採用敏捷方法但是卻槓龜的團隊，其中團隊成員缺少「追求技術卓越」的精神很有可能是一個主要的原因。很多公司的老闆或是主管，認為只要把人丟下去，以「**不斷漫罵與長時間加**

**班（棍子）**」搭配「**豐厚分紅（紅蘿蔔）**」，兩帖藥方同時服用，時間到了東西自然就會生出來。在這樣的公司文化中，想要導入 Scrum，ㄟ，套句聖人講的話：「卡早睡卡有眠」。

一個籃球隊平平是五個人，為什麼有的人只能在河濱公園打球，有的人卻可以在 NBA 打球。（路人甲：因為前者只想運動，而後者想賺錢） 想要導入 Scrum，一開始會遇到一些「框架上的障礙」，就是說需要花一點點時間讓團隊成員了解 Scrum 長得是圓的還是扁的，例如：

- Story 怎麼寫
- Story point 怎麼估
- 如何將一個 story 細分為 tasks
- 如何挑選 story
- Sprint 要定多長
- 如何進行 daily Scrum
- 如何進行 retrospective meeting
- 如何認領 task
- 如何定義 done
- 如何...

雖然要搞定這些事情一開始就夠你累的了，不過這都算小事，幾個月，最多一年好不好，應該都可以做的不錯。但是，如果團隊中有

那種「節能減碳」的成員（**為了節能減碳腦袋平常都不開機，就算是開機也都維持在省電模式，只維持最基本的生命跡象**），那就會遇到瓶頸。這就好比為什麼有人烈火掌可以練到第九重，而有的人卡在第三重就上不去了。

很久以前 Teddy 看過一本書，裡面提到「**在軟體開發中，最重要的一件事就是讓團隊成員動腦筋**」。當年 Teddy 其實不太了這一句話的意義，做軟體哪有不用動腦的？現在終於慢慢體會這件事情的重要性與困難度是很高滴。就好比上課的時候，有些人內體明明在教室裡面，但是靈魂卻早已蹺課，不知道跑哪裡去了。有的人看起來是有在寫程式，但寫出來的不知是功能還是 bugs。

結論：如果軟體開發團隊也能有類似美國職棒大聯盟的制度，可以交易球員，那該多好。把不適任的交換出去，或是不續約，或是調到小聯盟，然後從小聯盟調一些表現不錯的人上來，或是買進有潛力的球員。

***

友藏內心獨白：如果團隊成員意在參加，不在得獎，也許真的應該把 ~~牠~~ 他帶到河濱公園放生。

# 49 軟體是長出來的

06/23 21:40～23:25

12/12 09:44-09:51

原文發表於 http://teddy-chen-tw.blogspot.com/2010/06/blog-post_23.html.

Teddy 在「老闆，軟體不是這樣開發滴」提到台灣的硬體代工製造太強，所以很自然的就會以硬體代工的模式來看待軟體開發。關於軟體開發本質的爭論相信鄉民們早已經聽到耳朵都長（消音）了，不管軟體開發到底是屬於「工程」、「藝術」、「技藝」還是其他亂七八糟的大雜燴，今天 Teddy 要談的觀念是「軟體是長出來的，不是組裝起來的」。

這幾天 Teddy 在讀 Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum 這本書（PS：好長的書名啊，一口氣還念不完！）看到書中談論的一個觀點：

**Growing Software vs Building Software**

Building software 是大家普遍比較熟知的軟體開發模式，Teddy 把它稱之為「**組裝軟體**」。意思是說，軟體可以和硬體生產一樣，「規格

確定之後」尋找不同零件供應商，然後把這些零件全部兜起來組裝好，就變成產品了。這種製造模式，會造成目前常常在媒體上聽到的「iPad 概念股」或是「iPhone 概念股」。因為這些產品賣得好，所以台灣負責生產週邊零件的廠商也跟著「人家吃肉我喝湯」而賺了一筆。

代工代上癮的大、小老闆們，毛利 5 趴、3 趴賺到都快要趴到地板上了（有時候一不小心還會趴到地下室），聽說軟體的「毛利」很高喔，很自然的想到把硬體代工這一套擴展到軟體代工，或是藉由軟體提高硬體的附加價值。但是，軟體規格偏偏比孫悟空的七十二變還要多變，又不像硬體有那麼成熟的協力廠商供應鍊，真的傻傻地想要「組裝軟體」，就算是請了 1000 個人來組成「軟體生產線」，到時候「生產」出來的東西組的起來還真是有鬼。

其實軟體開發就像「種菜」,「種花」,「種水果」一樣，軟體需要靠開發人員每日細心呵護，才能慢慢「**成長茁壯（growing）**」。翻土、插秧、施肥、灌溉、修剪、除草、抓蟲、防風、防曬，該做的功課不能少，才能種出好作物出來。今年如果颱風特別多，就要小心風害；如果有強烈寒流，就要小心作物凍傷；太陽太大要幫水果穿衣服，有時候缺水還要休耕改種耐旱作物。軟體開發就跟種田是一樣滴，無法靠「組裝」而產生產品。

看看前輩們如何說：

- Andy Hunt 與 Dave Thomas 說：「Rather than construction, programming is more like gardening」;「All programming is maintenance programming」。
- Christopher Alexander 認為設計是一種 「differentiating process」而非「synthesis, a process of putting together things, a process of combination」。所以，建造好的建築物是要靠「participation」（客戶的參與）以及「piecemeal growth」（逐步成長）。
- Steve Freeman 與 Nat Pryce 寫了一本「Growing Object-Oriented Software, Guided by Tests」的書。注意，是 growing 不是 building 喔。
- Agile methods 採用 iterative and incremental development。
- Martin Fowler 說「Evolutionary Design 和 Refactoring」。

以上都是強調軟體跟人一樣，是從小 baby 長成大人，應該沒辦法「蓋」或「組裝」一個真人出來吧...機器人倒是有可能啦。

***

Teddy 扯了這麼一堆，鄉民們可能早就知道了這個概念了。很遺憾的，你的老闆極可能是屬於「~~侏儸紀~~ waterfall 時代」的靈長類，每

天還在想著如何靠組裝模式賺大錢。怎麼辦？Teddy 也沒辦法啊，不然幹麼沒事寫部落格來抒發情緒...XD。

***

友藏內心獨白：不寫部落格，就只能回去多喊幾聲「皇上聖明」...實在是喊不出口。

# 50 咸豐皇帝是怎麼死的？

06/20 20:59~22:20

12/12 15:32-15:44

原文發表於 http://teddy-chen-tw.blogspot.com/2010/06/blog-post_20.html，標題為「對症下藥」.

Teddy 曾經在第四台看過一部電影，劇情有一段提到清朝咸豐皇帝很喜歡「尋花問柳」，有一次咸豐皇帝生病，被太醫診斷出是「花柳病」。這個病照理講應該不算是什麼不治之症，但是皇太后為了怕咸豐皇帝得花柳病的事情洩漏出去有損「朝廷顏面」，因此吩咐太醫對外就說皇上得了「天花」，按照醫治天花的方法來醫治皇上。結果如何看官們可想而知，不久後咸豐皇帝就一命嗚呼去見老祖宗了。

**不願面對的真相**自古就有，這種故意將「**花柳、天花，傻傻分不清楚**」的事情可是多的很。萬一有人「白目」不小心戳破「國王新衣」的秘密，可是會死無葬身之地（本書作者就是一個活生生，血淋淋的例子...Orz）。識相一點還是乖乖的按照「天花」來醫治就好，反正死得是皇上，又不是太醫，更不是皇太后。不過，當太醫的也要小心，萬一皇太后耍賤招，太醫可是會因為醫治皇上不力而被「賜死」。

***

上述橋段在軟體開發裡面也是履見不鮮，軟體專案常見的病症與藥方有：

- 需求經常變動→請客戶畫押。
- 專案時間到了系統還沒完成→時程一再往後延直到功能做完或專案葛屁。
- 專案時間到了系統又還沒完成→直接丟給客戶先結案收錢再說。
- 時程延誤→開會。
- 時程嚴重延誤→開更多的會。
- 時程又延誤→加班。
- 時程又嚴重延誤→加更多的班。
- 程式品質不良→沒事，回家多喝開水就好了。
- 程式品質很不良→回家多喝開水外加蓋厚棉被睡一覺留點汗。
- 程式品質嚴重不良→我的眼睛，只看得到我想看的東西，哪裡有 bugs？
- 團隊士氣低落→訂定不合理的時程表。
- 團隊士氣嚴重低落→訂定超級機車的時程表。
- 專案一塌糊塗→報告老闆，一切都按造計畫中進行（我們原本就計畫要把案子搞的一團亂）。

當公司大到一定的程度，老闆通常沒有那個美國時間去關注太多的「細節」，所以這時候就有很大的「欺上瞞下」空間。清朝乾隆皇帝派兵去攻打緬甸， 明明是「大敗」，前線將軍卻可以回報「大捷」，而乾隆皇帝倒也樂得開心。很荒謬，Teddy 原本也這麼認為，但是現在越來越能體會箇中奧妙之處。

**會升官發財的人，心目中的第一優先，永遠是「把事情做對」，而非「做對事情」**，兩者有何差別？

- **把事情做對**：不管是偷、搶、矇、拐、騙，總之要做好可以討好「聖上」的事。打敗仗，要說成大捷；殺敵 100，要謊報殺敵 1 萬。龍心大悅之下，步步高升指日可待。
- **做對事情**：效法「劉羅鍋」每日上朝奏三本，針貶時政，順便煩死皇上。

天縱英明的聖上，會喜歡哪一種人？恐怕是天天喊著「皇上聖明」的前者吧。

軟體專案會發生問題，也許有很多因素並不是軟體本身「有病」所引起的問題。軟體的病，只要願意對症下藥，大體都還有的救，只是**醫藥費多寡**與**痊癒時間長短**的問題。就怕是硬把「花柳」當「天

花」，那就只能拖一天，算一天了。照 Teddy 看來，還不如早死早超生算了。

***

朋友臨別贈言：貴公司還是乖乖作它的硬體就好，學人家開發什麼軟體啊，七刀。

***

友藏內心獨白：誰說民國沒有太監？

# 51 精神不好的時候

4/27 21:07~22:04

12/13 17:06-17:13

原文發表於 http://teddy-chen-tw.blogspot.com/2010/04/blog-post_27.html.

這幾天 Teddy 感冒了，上禮拜五下午開始覺的精神不濟，但還不確定是得了感冒。由於上禮拜五剛好是 Teddy 的軟體開發團隊 sprint 結束的日子，下午 Teddy 利用 retrospective meeting 結束後的一點空閒時間整理放在 ezScrum 上面的 release plan 以及下一個 sprint 所需的 stories (備註：ezScrum 是一套由台北科技大學資訊工程系軟體系統實驗室所開發的開發原始碼 Scrum 輔助軟體)。Teddy 在一個恍神之下，居然把下個 sprint (sprint n) 的資料從電腦中殺掉了(原本是要將某一個 story 從下個 sprint 中移除)。**更慘的是，ezScrum 不能讓我重新建立這個編號 n 的 sprint，只能建立 sprint n+1，變成跳號啦。**由於 sprint 編號對於 Teddy 而言是有意義的，因此如果不能重建 sprint n 將會很困擾。好里加在，ezScrum 是 Teddy 的學弟開發的，趕快 call out 尋求幫助。

在學弟的幫助之下，經過約半小時之後，Teddy 終於重新建立了 sprint n。**可是沒想到更倒楣的事還在後頭，Teddy 在不知不覺中居然把這個 sprint 已經完成的三個 stories 給刪除了**（這 ezScrum 也有點問題，

既然這些 stories 的狀態都已經是 done 了，怎麼沒有防呆，還讓 Teddy 把他們刪除了！）。怎麼辦？再度 call out.又搞了快 20 分鐘，才把資料復原。

結果原本在 30 分鐘內就可以處理的完畢的事情，Teddy 那一天大概花了 1.5 個小時。

***

不曉得鄉民們有沒有那種熬夜寫程式，隔天睡醒之後發現昨天熬夜所寫的程式錯誤連篇，以至於要整個拔掉重寫的經驗？抑或是，花了好幾個小時解一個 bug 卻是怎麼都無法搞定，反而把程式**越改越亂、越改越亂、越改越亂**。最後終於受不了了，跑出去散個步，喝杯咖啡，大個便，洗個澡，睡個覺... 說也奇怪，靈感突然來了，重新回到座位上，問題迎刃而解。

這也是 Teddy 不贊成加班的另外一個原因。平心而論，一天只要能夠有 5-6 小時很認真寫程式的時間，就已經夠了不起，夠累人的了（難道是 Teddy 年紀大了，不耐操... XD）。加班寫程式，也不是說不會有生產力，短期來講會有不錯的效果，但是如果變成常態，人也麻痺了，專注力可能會變差，不但降低正常上班時間的工作效 率，甚

至於「長工時」所增加的生產力很有可能變成負的。因為很有可能做出更多的 bug 等著自己明天來 fix，導致於「明天的 bug，今天就給你傳便便」的現象。

結論就是，ㄏㄧ (hee)... 要休息。

***

友藏內心獨白：Teddy 沒有打 H1N1 疫苗啦....不要傳染感冒給他的啦...

# 52 剽竊

05/25 22:06~23:05

12/12 17:03-17:11

原文發表於 http://teddy-chen-tw.blogspot.com/2010/05/blog-post_25.html.

當年 Teddy 在念研究所的時候，修了一門「英文科技論文寫作」的課。老師在課堂中提到，**剽竊 (plagiarism)**這種行為對於學術論文寫作來說是一個大忌，千萬要避免，否則一旦被抓到將死無葬身之地。對於習慣「抄襲」、「盜版」、「copy」、「山寨」(ㄟ，那時候 還沒這個名詞）的研究生們，要發表學術論文千萬不可以秉持「天下文章一大抄」的態度來寫作。因此，老師特別幫我們定義了何謂「剽竊」:

**Writing facts, quotations, or opinions that you got from another source without citing them is plagiarism.**

接著，老師列出了四種寫作上的剽竊 (Teddy 內心獨白：抄襲就抄襲，居然還可以搞到分類，真是太神奇了。)

- **Language Plagiarism**：這類的抄襲最容易了解也很常見，就是把別人寫的內容原封不動直接拿來變成你的。
- **Structural Plagiarism**：這是指抄襲別人文章的結構。這一點 Teddy 當年其實沒有聽懂，不過自由發揮一下。例如，鄉民

們要寫一個 design pattern，而目前已經有很多種常見的 design pattern 寫作結構。如果鄉民們採用了某種結構，但是在文章中沒有提到這是你引用別人的結構，這樣也算是一種剽竊。

- **Idea Plagiarism**：這個很容易了解，抄襲別人的想法據為已有。例如，鄉民們從別人口中聽到一個不錯的研究方法，直接以自己的名義寫成文章發表，這樣也算剽竊。
- **Mosaic Plagiarism**：這種剽竊方法現在的學生常常使用，就是把別人的文章打上「馬賽克」然後就變成自己的文章。例如，老師要學生交報告，學生就把 google 到的內容 copy 下來，東改幾個字，西改幾個字，或是把句子前後交換等等，以為可以偷天換日，瞞天過海。很遺憾，這樣還是剽竊。

自從上了這門課之後，Teddy 就更加了解到寫作時要避免有心或無意的剽竊。

***

除了寫作上要避免剽竊，其實「為人處事」更應該避免剽竊。**「避免剽竊」就是當鄉民們在「合理使用」別人的「智慧財產」時，要把「功勞」(credit) 給原作者，不能讓別人以為這是你自己的貢獻。**試想，假設（公堂之上假設一下應該不犯法吧…XD）鄉民們有一位同事（姑且稱之為 X 先生），當他來跟你請教一些問題的時候，你很大

方的把你所知的都告訴他。之後，X 先生就把你告訴他的解決方案，跑去跟別人（這個「別人」，可能是你們的主管，其他部門的主管，或是任何的阿貓阿狗）講，並且完全沒有提到這是你告訴他的作法，讓別人認為：「哇，X 先生真行」；這就是一種剽竊。久而久之，當鄉民們發現 X 先生是這樣的人，也就不想與他分享知識。

鄉民甲：以上所說，和軟體開發有何關係？

Teddy：有滴，**因為軟體開發是一種「獲得知識的過程」。**如果你的團隊裡面，有這種喜歡剽竊的人，就會造成團隊成員不願意分享知識，因而影響軟體開發的速度或是品質。不知道有沒有人研究過這種人在軟體開發團隊中佔了多少比例？如果團隊中有這樣的人，過著防同事跟防賊一樣的生活，還真是辛苦啊。

PS：請勿對號入座。

***

友藏內心獨白：應該去買「小人玩偶」嗎？

# 53 你重視什麼？

05/15 22:21 ~23:50

12/12 17:14-17:19

原文發表於 http://teddy-chen-tw.blogspot.com/2010/05/blog-post_15.html.

今天上午是北科大資工系創系十週年聚會，下午指導教授邀請實驗室畢業的校友留下來聊聊天。在逐一自我介紹之後，各自帶開成小組討論隊形。這個故事便開始於指導教授，Teddy 與另一位學弟聊天的場景中。當話題正圍繞在 Scrum 的時候...

| | |
|---|---|
| 學弟： | Scrum 在我們公司可能不太適合。 |
| 指導教授： | (偶然往書櫃方向一看) 耶，實驗室怎麼會有一本 PMP 的書？ |
| 學弟： | 我們公司也有很多人去考 PMP。 |
| 指導教授： | PMP 的作法與 Scrum 完全不同... |
| 學弟： | 我們公司好像也沒重視 PMP。 |
| 學弟： | 公司去考 PMP 的人原本以為考上之後可以加薪，但是結什麼都沒有。 |
| Teddy ： | 哪你們公司重視什麼？ |
| 學弟： | ... (苦思五秒) ... (無言) |
| Teddy ： | 重視拍馬屁？！ |

***

Teddy 是一個有話直說，經常在有意無意中得罪人的人。更慘的是，Teddy 還樂此不疲... 這種天生反骨的個性，在古代應該是早就被「推出午門」不知道斬了幾次。Teddy 很幸運，第一份工作遇到的老闆（懷念朱先生...），以及念研究所時的指導教授們，都是很有肚量，也不是喜歡被吹捧的人，因此讓 Teddy 可以真誠的說出自己的想法，不必為了「揣摩上意」而講一些言不由衷的話。這一兩年 Teddy 才慢慢見識到什麼叫做「拍馬屁」與「揣摩上意」的最佳實務作法（這種事也有 best practices... 應該可以寫成 pattern languages...）。在小公司裡面，苦幹實幹的人比較有可能被老闆重視，但是隨著公司規模逐漸擴大，老闆無形之中覺的越來越覺的自己**天縱英明**，也聽不進去反對意見，此時要升官發財就需要向「和珅」學另一套本領。

耶，這怎麼跟古代的皇帝那麼像？！沒錯，就是這麼像。Teddy 之前在談 Scrum 的時候曾經說過：「**如果你的老闆是屬於袁世凱這一類型的，你跟他談民主只是找死**」。然而，一個組織如果只有一種聲音是很危險的，就好像生態系中只有少數物 種，是很容易因為環境改變而整體滅亡。

***

這幾天 Teddy 在讀「**你可以不一樣：嚴長壽和亞都的故事**」這本書，看了很感動。其中有一段提到：

有一天，嚴長壽無意中讀到一本跟佛教有關的書，看到「**借觀**」二字，他像忽然被打醒一般。一個人一輩子所能擁有有形的物

質、財富，其實都不過是「借」來看看而已...

很可惜 Teddy 天生笨手笨腳，體力又差，又只會開發軟體，沒有其他一技之長，也沒有老家在鄉下。否則有時還真的很想學習日本「自給自足過生活 (自給自足物語)」電視節目中的那些人，反璞歸真過過自給自足的生活。

Kay：Teddy 你應該會餓死...因為睡太晚農作物都沒有照顧，雞也沒餵...

***

友藏內心獨白：本集的重點到底是什麼？

# 54 The power of duplicate code

05/27 22:22～23:18

12/12 16:41-16:50

原文發表於

http://teddy-chen-tw.blogspot.com/2010/05/power-of-duplicate-code.html.

本篇的主題是 The power of duplicate code，翻成中文是**「重複程式碼的力量」**。奇怪，duplicate code 不是一種寫程式應該要避免的 bad smell 嗎，為什麼這種壞東西還會有「力量」？別忘了星際大戰中尤達大師也曾經說過要留意**「the power of dark side」**，這種暗黑勢力有時候比正義的力量來的還要強大（不然黑武士為什麼那麼強）。

那麼，duplicate code 到底有什麼力量？**Teddy 發現 duplicate code 最大的力量就是可以讓 programmers 看起來很忙且生產力很高。**多麼神奇的力量啊！

講一個 Teddy 從朋友那裡聽來的的故事，真的是聽來的喔，絕對不是發生在 Teddy 周遭的人身上。故事是這樣的，有一個小型的軟體開發團隊，在沒有採用 Scrum 之前，team members 都各自負責若干個 sub-projects，彼此也幾乎不會去看其他人所寫的 code。後來，這個團隊不小心無緣無故被強迫中獎採行了 Scrum。經過若干 sprints 之後，閒閒沒事的 Scrum Master 突然去 review 某一個 sub-project，

發現雖然是用 OO (物件導向) 語言來開發系統，但是裡面的 code 設計的很不 OO ，理解度大該只有 10%不到( 另一種說法是這個 Scrum Master 太遜了，看不懂別人寫什麼... )。但是由於當時這個 sub-project 的程式可以正常執行，Scrum Master 也就沒有立即建議要 refactor 該模組，反而是偷偷保佑永遠都不需要改到它。

但是，如果真的不用修改那麼這個故事也就講不下去了。某一天，該團隊突然發現這個 sub-project 在 multiple-threaded 的情況會出錯，而原本的負責人（在此稱之為 X 先生）在事蹟敗露之前早已想辦法「落跑」到其他專案中。為了解決此問題，Scrum Master 花了一點時間以 OO 的方式來重新設計該 sub-project，然後與其中一位有一點了解這個 sub-project 的 team member 一起採用 pair programming 的方式重新寫了這個 sub-project。完成之後，原本在 multiple-threaded 環境所遇到的 bug 也自然消失了。不過這不是重點，重點是 Scrum Master 很無聊的去算了一下該 sub-project「改造前」和「改造後」的 LOC (line of code)：

- 改造前：約 10000 行 。
- 改造後：約 2000 行（包含 unit tests）。

再仔細一看，原本的程式充滿了 duplicate code 以及很奇怪的參數傳遞方式。除了原作者以外，地球上可能找不到第二個人可以維護這個系統。改造後，由於採用良好的 OO 設計，不但程式行數大減，而

且所有的 team members 都有能力也實際參與了這個 sub-project 的開發。

***

在真實世界中，有多少人（多少主管或同事）會幫你做 code review？在台灣應該極少，因此誰說 duplicate code 是 bad smell，從 X 先生的觀點，duplicate code 可是 good smell 呢！要不是有 duplicate code，如何讓每天都有生產力呢。

老闆：我的員工每天都加班，工作好認真，好辛苦喔。

Teddy ：你確定...

也許大部分的老闆或主管都忙到只能對員工實施「black-box testing」，無法實際了解員工每日的工作。此時，duplicate code may be a good smell。如果鄉民們的公司採行的是 TPS（Toyota Production System），強調「現場管理」，那麼 duplicate code 肯定是 bad smell。但是，話說回來，在台灣有誰開發軟體那麼「精實」啊。

結論，duplicate code 因為文化因素因此在台灣可能有 90%以上的機率是屬於 good smell。

***

友藏內心獨白：故事真的是聽來的喔。

# 55 這不是整人遊戲之 time log 紀錄方式

6/14 22:35~22:57
6/14 23:07~6/15 00:20

12/19 10:07-10:19

原文發表於 http://teddy-chen-tw.blogspot.com/2010/06/time-log.html.

Teddy 在 2005 年春天修了一門 **PSP -- Personal Software Process** 的研究所課程（個人軟體程序，聽起來就是無聊到爆的一門課。如果是 Sony 的 PSP 那該多好），沒記錯的話 PSP 這玩意兒（請捲舌）是 Watts S. Humphery 博士所發明的，目的就是要利用「工程」的方法，來提昇軟體開發的品質。基本精神就是，**「任何東西只要無法測量，就無法改善」**。因此，為了要改善軟體開發的品質，出發點就要從「單兵作戰守則」開始，要求單兵（programmers）卯起來紀錄（測量）在開發過程中一些你從來也沒有想到需要去紀錄的資料，包含：

- Time log: 詳細紀錄不同開發階段，例如 Plan、High-Level Design、High-Level Design Review、Design、Design Review、Compile、Code、Test、Project Manage 各花費多少「分鐘」。

- Size: BASE PROGRAM LOC (基本的程式行數，就是說開發一個新的功能之前程式有幾行), LOC DELETED(此次開發刪除了幾行), LOC MODIFIED (此次開發修改了幾行)。
- Defect: 紀錄 defect 的 date (哪一天被發現的), number (第幾個 defect), type (哪一種類型的，邏輯錯誤，打字錯誤，資料結構錯誤等等), inject (在哪一個階段所產生的，例如計畫階段，設計階段，寫程式階段等等), remove (在哪一個階段被修正，例如 Design Review 階段或是測試階段), fix time (花了多少時間來修正)。

剛剛去「批衣服」現在繼續。

除了這些以外，還要寫一堆資料，一個學期搞下來，雖然只寫了 10 支（1A~10A） 總行數加起來可能不到 2000 行的小程式，整個人卻是累到走路必須要靠北邊走，有一種被掏空的感覺。今天要談的主題是 time log，因此看一下 Teddy 當初修課作業所紀錄的 time log 範例：

**Table C16 Time Recording Log**

Student 陳xx　　Date 6/18/2005
Instructor 劉xx　　Program # 10A

| Date | Start | Stop | Interruption Time | Delta Time | Phase | Comments |
|---|---|---|---|---|---|---|
| 6/18 | 15:09 | 15:24 | | 15 | Plan | |
| | 15:32 | 16:04 | | 32 | Plan | Cenceptal design & size estimated |
| | 16:20 | 16:33 | | 13 | Plan | Plan summary |
| | 16:34 | 16:58 | | 24 | Plan | Cyclic summary |
| | 17:00 | 17:13 | | 13 | Plan | Test case design |
| | 23:41 | 23:59 | | 18 | HLD | High-level design |
| 6/19 | 00:01 | 00:06 | | 5 | HLDR | High-level design review |
| | 00:12 | 00:43 | | 31 | Design | Cycle 1, 高斯消去法 |
| | 00:45 | 00:46 | | 1 | Design | Cycle 1, LinkedListTool |
| | 00:47 | 01:09 | | 22 | Design | Cycle 1, MultipleRegression |
| | 01:10 | 01:16 | | 6 | DR | Cycle 1 |
| | 01:17 | 01:29 | | 12 | Code | Cycle 1 |
| | 01:29 | 01:33 | | 4 | CR | Cycle 1 |
| | 01:33 | 01:34 | | 1 | Compile | Cycle 1, note: 請參考 PIP1 |
| | 01:40 | 02:09 | | 29 | Test | Cycle 1 |
| | 02:33 | 02:55 | | 22 | Design | Cycle 2, MRPredictionInterval |
| | 02:55 | 02:58 | | 3 | DR | Cycle 2 |
| | 02:58 | 03:14 | | 16 | Code | Cycle 2 |
| | 03:14 | 03:17 | | 3 | CR | Cycle 2 |
| | 03:17 | 03:18 | | 1 | Compile | Cycle 2, note: 請參考 PIP1 |
| | 03:19 | 03:41 | | 22 | Test | Cycle 2 |
| | 15:11 | 15:52 | | 41 | PM | |

修完課之後，只有一個感想：「**這是整人遊戲嗎？**」Teddy 這一輩子應該沒有這個榮幸可以去寫戰鬥機、核電廠、或是太空梭所使用的程式，真實世界中的普羅大眾誰跟你紀錄這些。鄉民們信不信把這一套拿給你們老闆瞧一瞧，如果沒有被痛貶一頓的話，請注意你一下老闆的精神狀態是否正常。

但是，Humphery 博士也不是省油的燈，PSP 或是進階版的 TSP (Team Software Process) 或是宇宙無敵世界無雙的 CMMI (Teddy 知道，CMMI 不是 process...不要太挑剔) 裡面提到的許多「觀念」（再次強調，觀念）出發點都十分正確，但是「落實」的方法卻「沒人性」。

經過這麼多年，Teddy 已經了解到，**反是「違反人性」的制度或是作法，最終大多以失敗收場（上有政策，下有對策。生命會自己找到出路滴）**。紀錄 Time log 的想法，就像是減肥的人每天紀錄吃掉多少「卡路里」，或是想存錢的人每天紀錄「花了多少錢」是一樣的。立意良好，做到的沒幾個。但是一旦能夠作到，的確有 其效果。因此，後來 Teddy 就自創（ㄟ，任何人都可以想到啦）一套簡化的紀錄 time log 格式，在唸書的這幾年當中，持續紀錄。請參考 Teddy 某幾天的 time log：

2006/11/21

1. 研究 Eclipse editor
   * 研究 syntax highlighting
   (09:30~12:10, 160 mins)
2. 研究 Eclipse editor
   * 研究 syntax highlighting
   (12:40~15:10, 150 mins)
3. MAUT meeting
   (15:10~15:40, 30 mins)
4. 回答 鄭老師 中央演講投影片 的一些問題
   (15:40~16:20, 40 mins)
5. MAUT meeting
   (16:20~16:40, 20 mins)

6. 研究 Eclipse editor

   * 研究 syntax highlighting (完成雛型)

   (16:40~18:00, 80 mins)

2006/11/22

1. 研究 Eclipse editor

   * 研究 outline view

   (10:30~13:20, 170 mins)

2. 研究 Eclipse editor

   * 研究 outline view (完成雛型)

   (13:50~17:50, 240 mins)

2006/11/23

1. 研究 Eclipse editor

   * 將 nodes 改成支援 visitor pattern (未完)

   (10:30~13:10, 160 mins)

2. 和老師去吃飯

   (13:30~16:40, 190 mins)

3. 研究 Eclipse editor

   * 將 nodes 改成支援 visitor pattern (大致 OK)

   (16:50~19:10, 140 mins)

就這樣，格式很簡單，基本上實際紀錄都最小是以 5 分鐘（為了偷懶，Teddy 實際上通常最小是 10 分鐘為一個單位）來紀錄，只紀錄「有意義」的事情。太瑣碎與沒有意義的時間片段就忽略，例如，尿尿花了三分鐘，泡咖啡花了七分鐘這種就可以不用紀錄。

用紀錄流水帳的方式來紀錄 time log 的好處是**「為了紀錄所額外花費的成本很低」**。記得物理領域有一個叫做「測不準原理」的嗎，為了測量，觀測者其實已經改變了被觀測對象的行為。所以，越是輕量級的紀錄方式，看起來好像「不太精確」，但是卻是實際可行又具備參考意義的方法。

***

看到這邊，還沒閃人的鄉民們，可能會以為只有 Teddy 吃飽沒事幹才寫 time log。錯，Teddy 所屬的「台北科技大學資工系軟體系統實驗室」的所有博、碩士生，都在寫 time log，而且連續實施這個作法至少應該超過 6 年以上。有什麼好處？主要的好處是協助時間的分配。例如，參考學長，學姊的 time log，可以知道修一門課，像是 OOP，平均需要花多少個小時。當有新生修課投入時間太少，老師馬上就可以看出來並提醒他要投入足夠的時間。如果投入時間太多也是問題，可能是其他課投入 時間太少，或是遇到問題無法解決。這麼多年實施下來，真的幫助了不少學生。

對於做研究也有幫助。例如，經過統計，出一篇好的 journal paper 從頭到尾大概需要 500 小時，寫一本碩士論文可能需要 200~250 小時。如果有研究生要求畢業，指導教授只要稍微看一下研究生投入的時間和產出物，就有一個比較客觀的基礎可以來決定能不能讓他畢業，研究生也不會不服氣覺得被教授幺。Teddy 也利用 time log 來預估寫一份計劃書需要多少時間（大概要 40 ~ 80 小時）。

最後要注意幾點：

1. 紀錄 time log 是「良心事業」，**灌水是沒有意義的**。
2. 所紀錄的時間是「實際工作時間」，例如 Teddy 紀錄寫一份計劃書需要 40 小時，這是「實實在在」的 40 的小時，而不是說一個工作天算 8 小時，做了五天因此得出 40 小時。有什麼差別？如果是後者，雖然做了五個工作天，可是實際每天有效工作實數可能只有三小時，其他時間都在打 B、聊天、噗浪、玩 FB。因此實際只有 3 * 5 = 15 小時。只要看了計劃書內容，40 小時和 15 小時所做出來的東西是差很多的，很容易露出馬腳。
3. 每個人的能力與經驗不同，在參考別人的 time log 時必須考慮進去。例如，假設 Teddy 寫一份計畫需要 40 小時，如果是找研二的學弟來寫出類似品質的計劃書，時間可能

至少需要乘上 2 或 3。（PS：學弟，學長有練過，不要傻傻的直接參考學長的數據）。

***

友藏內心獨白：劉老師，PSP 的訓練還是滿有用的，要繼續開課喔。

# 第六部 設計

# 56 Problem Domain vs. Solution Domain

4/29 22:50 4/30 00:10

12/13 16:51-17:03

原文發表於
http://teddy-chen-tw.blogspot.com/2010/04/problem-domain-vs-solution-domain.html.

實驗室有個提早報到的新進準碩士班學弟被指派研究 CruiseControl，從最基本的安裝，學習如何撰寫 ANT scripts，在建構活動中執行編譯與測試，最近進行到可以用 Cobertura 來跑 test coverage。某日，Teddy 在看過學弟的 demo 之後，問了他幾個問題：

1. Cobertura 所產生報表，裡面有 line coverage、branch coverage、與 average complexity。這三個數據各代表什麼意思？
2. 你覺的使用 CruiseControl (廣義的說，持續整合) 最困難的地方是什麼？
3. 你接下來打算做什麼？

學弟的回答：

1. 不清楚這些數據的意思(PS：根據 Teddy 觀察，學弟認為只要把 Cobertura 的報表掛到 CruiseControl 中就代表完成了該項工作)。
2. 在寫 Ant scripts 的時候，需要依據不同的工作設定不同的參數，很容易出錯。
3. 預計把 StatSVN 的報表掛到 CruiseControl 中。

有帶過研究生經驗的鄉民們可以發現，這位學弟的回答是很典型的學生思考模式：**只看到「Solution Domain」，而忽略了「Problem Domain」。**

***

Problem Domain 就是問題發生的地方，也就是軟體開發所謂的**需求**。而 Solution Domain 就是問題的解決方案，也就是軟體開發所謂的「設計」或是**實做**。舉凡像是演算法，design patterns、refactoring、architecture 等等，都算是 Solution Domain 的範圍。

**時時提醒自己哪些概念是屬於 Problem Domain 或是 Solution Domain 是一個很簡單，但卻很有用的思考與分析工具。**以「持續整合」這個領域為例子，如果眼中只有 Solution Domain，那麼這位學弟最後可能變成 ANT 大師，熟用數十種工具，但是卻不知道要如何

利用這些 Solution Domain 的工具來幫助真實世界（Problem Domain）的軟體開發團隊來落實持續整合。為什麼，因為持續整合的 Problem Domain 是「軟體開發」，如果對於持續整合可以解決哪些軟體開發活動中所遭遇的問題未加以分析清楚，則光是會使用工具，並無法解決問題。這就像有一陣子很多公司花大錢買了 Rational Rose 的工具，以為這樣就增進軟體開發的速度並且改善軟體品質是一樣的。

**光是知道問題，答案做不出來，也是白搭。答案寫出來了，但是答錯問題，更是白忙一場。**所以，對於軟體從業人員而言，要能夠具備分析 Problem Domain 和「生出」Solution Domain 都是同等重要的。以下列出幾個屬於持續整合 Problem Domain 的概念：

- 持續整合的主要目的，就是要找出「整合」的問題。哪麼，那些算是「整合」的問題？
- 針對不同性質的專案，持續整合的內含需包含哪些（例如，編譯，測試）？
- 當一個軟體專案大到一定程度，為了分工以及組織 (模組化與管理相依性) 的目的，因此專案通常會被切割成若干個小專案，而最終的產品將由這些小專案組合而成。
    - 專案相依性對於持續整合有何影響？
    - 公用（共用）元件對於持續整合有何影響？
    - 持續整合的產出物有哪些？
- 持續整合的執行速度使否會影響到軟體開發活動？

大致分析這些問題之後，當鄉民們需要評估 Solution 的時候，例如，選擇哪些持續整合系統，使用哪些工具，要如何應用，如何改變專案結構與開發流程來使得持續整合更順暢等等，才有個取捨的依據。

***

Teddy 不是一個聰明人，在處理事情的時候所使用的招式也都十分簡單。「區分 Problem Domain 與 Solution Domain」這一招 Teddy 已經用了好幾年了，非常用有。下次有人請你幫他 review 東西的時候，可以拿出來試看看，也許可以輕鬆幫對方找出一些盲點。

***

友藏內心獨白：感冒什麼時候才會好啊...

# 57 再論 Problem Domain vs. Solution Domain

6/6 21:08～22:50

12/19 11:01-11:08

原文發表於
http://teddy-chen-tw.blogspot.com/2010/06/problem-domain-vs-solution-domain.html.

前一篇「這不是髒話」還沒寫完，本集繼續。X 小姐後來又提到：

[13:24:49] 而且說那找一位好一點的分析師來有沒有幫助，還說沒幫助

[13:25:03] 因為不了解 Domain Know-how 太好的分析師來也沒用

[13:25:50] 他們說要找懂 Domain Know-how 深的系統分析師而不是只是很會系統分析的系統分析師

X 小姐轉述她主管的看法，這就軟體開發或是做專案的人很典型的一種心態，認為只要找到所謂「domain know-how」很強的人來分析需求，做出來的系統就沒問題。錯，錯，錯，連三錯。Teddy 的意思不是說 domain Know-how 不重要，相反地，domain know-how 真的

很重要，但是實做的能力也同樣重要。用講的太慢，乾脆用唱的...不對，是用看的。請看下面這張圖：

Problem Domain

| 對 | 錯 | |
|---|---|---|
| A 正確的系統 | C 把錯的事情作對 | 對 |
| B 把對的事情作錯 | D 把錯的事情作錯 | 錯 |

Solution Domain

Teddy 之前寫過一篇「Problem Domain vs. Solution Domain（第 255 頁）」，X 小姐主管遇到的問題便是沒有弄清楚這個概念。先看看 problem domain（就是需求），所謂「domain know-how」很強的人的就很懂問題領域，因此可以整理出很好的需求出來（寫出很符合各客戶所需的需求，甚至是客戶沒想到的都幫他考慮到了）。再看看 solution domain，以軟體開發而言，solution domain 就是軟體設計，實做，測試這些技術（姑且先把分析歸類為 problem domain 的活動）。

看看上圖的，一共可以分成四個象限：

- A：需求和實做都做對了，這也是我們開發軟體或廣義的說開發任何系統所要追求的。
- B：需求做對，但是實做一團糟。這比較接近 X 小姐主管所希望的「找到 domain know-how 很強的人來分析需求」，但是

卻不在乎如何改善 solution domain (如何改善軟體開發實做面的問題，例如自動化測試)。

- C：需求錯了，但是實做正確；結果還是白忙一場。人家要吃「素食」，結果你去買個「麥當勞」牛肉漢堡回來。
- D：需求錯，實做也錯。也許 X 小姐所負責的專案比較接近這個情況，所以才會（1）累積了 600 多的 bugs（實做錯）;（2）她的主管希望找到 domain know-how 很強的人來分析需求（需求錯）。

經過 Teddy 一番分析，鄉民們應該就很清楚了，這個 Problem Domain 和 Solution Domain 的觀念真的是簡單到不行，可惜有些大官卻不是很了，或是有意輕忽 solution domain 的重要性。**畢竟有許多雖然身處軟體業的主管們依舊把「開發軟體」簡化為只有「coding」，而他們又很瞧不起 coding**。說實話，就算是把開發軟體簡化為只有 coding，也不可輕忽 coding。畢竟，軟體最後還是一堆程式，要對程式碼心懷敬意啊。

***

寫到這邊 Teddy 再附贈一個故事，話說當年（又是 10 幾年前的陳年舊事）Teddy 還是青澀少年時，曾參與一個用 Java 開發的「Intranet 進銷存系統」。當時參與的三個 programmers 都是沒有進銷存 domain know-how 的人，於是公司第一任總經理就找了他的一個朋友，就是

所謂 domain know-how 很強的人來做分析。光是分析就搞了一年左右，寫了一本幾十頁的分析書之後就下落不明，剩下這三隻誤入叢林的小白兔，獨自與這份破綻百出的分析書以及無辜的客戶奮鬥。

經過 N 年之後，Teddy 已非當年的年幼無知的小白兔，也了解到這種「把文件丟過門」的方式是行不通的。很遺憾，講句不要臉的話，並不是所有人都和 Teddy 一樣有一直在看書與思考。很多公司的主管，還是抱持著 N 年前做案子的模式。**要「慈禧太后」相信「洋槍洋砲」比「義和團」還要厲害真的不是那麼容易的一件事**，就算是現在都快民國 100 年了還是一樣。

鄉民們想想「醫生」這個行業。醫生要看病，所以一定要懂 domain know-how，否則無法找出病因。接著，醫生還要提 solution，可能是做檢查，吃藥，開刀，或是其他治療方式。所以，醫生是 domain know-how 很強，而 solution 也很強的人（不過請注意，醫生的專業分工很細，每一科的領域很窄。但是軟體卻是包山包海，什麼領域都有。）。試想一下，如果醫生只有 domain know-how，他可能會告訴病人：「恭喜，您被診斷出得了胃潰瘍，至於如何治療不關我的事」。相反地，如果醫生只有 solution 很強（很會開刀好了），那麼醫生會告訴病人：「麻煩告訴我你想被切那一塊」，這像話嗎。

軟體開發，需求是一直改變的，就好像是病人的病情隨時在變化，

不可能看一次醫生（需求分析）吃固定的藥（solution）就 OK 這是傳統 waterfall 的作法）。在病人的病情隨時會變化的情況下，醫生與病人一定要隨時互動，才有可能掌握病情。想想 Scrum 的 sprint planning meeting、daily scrum、sprint、sprint review meeting、retrospective meeting，精神都是一樣滴。丟掉傳統那一套分析師（架構師）做完需求分析就沒事的思維吧。

結論，講了那麼多了，再不懂這個觀念 Teddy 也沒沒轍了。

***

友藏內心獨白：X 小姐的公司 F 1.0 版都做了 10 年了，domain know-how 還不強，真的是太超過了。

# 58 要抄就要抄最好的：架構師篇

4/22 21:17~20:30

12/13 17:15-17:26

原文發表於 http://teddy-chen-tw.blogspot.com/2010/04/blog-post_22.html.

史提芬周： 「只要有心，人人都可以成為食神」。

Teddy： 「只要有心，人人都可以成為 architect」(啊，沒有押韻...)。

在「食神」電影中，有人問「史提芬周」要如何才能做出好吃的菜。史提芬周回答：「一字謂之心」。在白日夢裡，有人問 Teddy 要如何才能成為 architect，Teddy 回答：**「一字謂之抄」**。沒錯，就是「抄」，英文叫做「copy」，內地人叫做「山寨」，老台灣人叫做「海盜」。俗話說：「**有樣不抄，對不起父母。抄光學光，為國爭光**」。不只要抄，而且還要挑最好的，最高檔次的來抄（看看人家三爽做的多成功...）。

***

對軟體開發人員來講，有朝一日能夠成為 architect 應該是職涯中相當重要的一個階段，通常代表自己歷經風霜，飽嘗客戶與老闆的荼毒，

算是見過大世面的高級技術打工仔。這樣的人才，相當於武俠小說中的「高手」。雖然在武俠小說中高手要練得絕世武功動不動都要耗費個幾十上百年的，但是也有所謂的「少俠」，十分好狗運，吃了什麼「千年靈芝」、「何首烏」、「大還丹」之類的補品，又不小心撿到「九陰真經」、「九陽神功」之類的的秘笈。其結果就好比娶了有錢人的獨生女，或是嫁給那一個富二代，少奮鬥了幾十年。

軟體界有那麼好康的事嗎？有滴！Teddy 是不知道有沒有什麼東西吃了可以讓程式設計的功力變得比較好一點，不過秘笈倒是有滴。只要將這本秘笈中的招式學個三到四成，就可以略有小成。要是學個七到八成的話，不得了，躺在家裡都不怕 ~~選不上~~ 系統做不出來。什麼書那麼強... 各位觀眾 (奏樂...)：

*Contributing to Eclipse: Principles, Patterns, and Plug-ins, by Erich Gamma and Kent Beck*

鄉民甲：Teddy 你又在話唬爛，整本書名沒有一個字在講 architecture，甚至連個 a 開頭的英文字都沒有。

沒錯，這是一本介紹 Eclipse 設計的書，雖然整本書名沒有提到 architecture 或是 architect，但是這本書的內容呼應了本篇的主旨：**「要抄就要抄最好的」**。Eclipse 的強大不需要 Teddy 多費唇舌說明了，

多麼棒的設計啊。有人寫書介紹這些架構的設計理念（先研究不傷身體，再講求藥效 !@#%$），只要花 USD 39.99 就可以帶回家，實在是太划算了啦。**不買遺憾終身，買了不看終身遺憾。**

究竟這本書內容在談些什麼呢？還是那句老話，自己看 (鄉民乙：挺不負責任的部落客...)。請原諒 Teddy，因為寫到這邊已經很累了 (今天找了 10 幾個 bugs...心中原本已經放下的某根指頭又蠢蠢欲動....) ，再加上秘笈內容博大精深，網路上高手又多，萬一 Teddy 一不小心講錯，豈不是自己討打。

***

友藏內心獨白：其實這一篇是硬擠出來的，湊湊數。

TO DO:
考慮補上書本內容簡要說明

# 59 你的軟體架構有多軟

04/17 22:49~04/18 00:42

12/19 09:19-09:24

原文發表於 http://teddy-chen-tw.blogspot.com/2010/04/blog-post_17.html.

「軟體」這個名詞有時候真是害苦了從事這個行業的人。「軟體」→「軟的物體」，顧名思義就是一種像是「黏土」一樣可以捏來捏去，沒有固定形體的東西。原本客戶一開始要的是小花貓，你作到一半他忽然要改成頭上有個「王」字的大老虎，你也要想辦法生給他。

客戶：這個很簡單的啦，不是改幾行程式就好了?

你： 走路請...靠... 左邊走...

根據 Teddy 的經驗，大部分的軟體系統其實都很「硬」，隨便改了一行程式都可能會額外「製造」出意想不到的 bugs。**雖然實務上軟體是很硬的，但是你的客戶或是老闆卻不斷的催眠你：「軟體是軟的，軟體是軟的」**。久而久之，涉世未深的 programmers 居然也開始萌生「軟體應該是軟的喔」這樣的想法。為了讓自己手邊硬到不行的軟體變軟，programmers 們便開始嚐試各種作法，其中一個常見的作法是：設計一個 ~~包山包海~~ 具有可擴充性的軟體架構。

鄉民們如果查一下軟體架構的內含，應該知道軟體架構主要是用來滿足 availability、modifiability、performance、security、testability、usability 以及其他族繁不及備載的 nonfunctional requirements (或稱為 quality attributes)。傳統上（應該說主流的想法，因為到現在大部分的人也都還這麼想）大家都認為要在系統開工之後才來考慮 nonfunctional requirements 通常會造成系統大改，成本太高。這有點像是，當你蓋好了「一品苑」之後才發現它長得太高了，可能會讓有心人士可以直接把總統官邸當成靶場，違反了「security」這個 nonfunctional requirements。房子都蓋好了不然是要怎樣？（套句遙遙的廣告台詞：這是你的 security，不是我的 security ...XD）不管是要總統搬家（誰想出這個乞丐趕廟公的餿主意？）或是停發使用執照，成本都很高。

所以，雖然 agile methods 告訴我們不要做 big up-front design，但是軟體架構身份這麼特殊，是否應該享有特權，等軟體架構設計好了再開工？

***

問題來了，怎麼才算是「軟體架構設計好了」？仔細想一想，這是一個「先有雞，還是先有蛋」的問題。如果鄉民們相信 agile methods 所倡導的 iterative and incremental development，那麼應該知道：

- 三心二意的客戶以及劇烈變化的市場隨時都可能改變需求。
- 客戶的需求通常在他看到或用到軟體之後會越來越明確。
- 因此，軟體專案不必也沒辦法等所有需求都確定才可以開始（因為需求永遠都在變啊，要等所有需求都確定才開始那不就等於永遠都不會開始）。

在需求會變動的情況下，怎麼設計一個軟體架構來滿足可能會變動的需求？另外，如果軟體架構可以或應該在軟體開發之前就完全決定，那是否表示需求就要固定？

以上兩句繞口令可直接忽略，重點是，如果依據傳統的作法，在 coding 之前就要把軟體架構設計好，會引誘開發團隊進入 big up-front design 的陷阱。更慘的是，就好像在 waterfall 流程中通常會要求客戶對需求畫押保證不再變動一樣，結果就是.... 不管你事前花了多少時間，設計好的需求與軟體架構，在 coding 之後幾乎不可能不改變（請不要拿美國 NASA 或是國防部的案例來嗆聲，台灣大部分的開發人員在有生之年應該都沒有這個榮幸做到這種案子）。

講了這麼多，那到底要怎麼做？首先就是心態要調整一下，要先相

信軟體架構是有可能採用「逐步成長」的模式來慢慢成型。這一點當然是「說得比做的容易」，因為**「選對軟體架構可以讓開發團隊上天堂，選錯軟體架構就只能住套房」**。看到其他人用好的軟體架構或是 framework 過的自由自在的日子，而自己卻在住在套房中被綁手綁腳的，這種感覺會讓開發團隊覺的自己很遜，最後可能變成「砍掉重練」（傳說中的設計魔人是否就是這樣誕生滴？）。

由於這個問題太難了，因此 Teddy 就不要不懂裝懂，留待各位鄉民們自行發揮。不過以下有一點資料可以參考一下。最近從 Teddy 在讀 *Implementing Lean Software Development: From Concept to Cash* 這本書，裡面提到：

*The objective of a good software architecture is to keep such irreversible decisions (nonfunctional requirements such as security, performance, extensibility, etc) to a minimum and provide a framework that supports iterative development.*

…

*It is time to abandon the myth that architecture is something that must be complete before any development takes place.*

要如何 “keep irreversible decisions to a minimum and provide a framework that supports iterative development” 說真的就要靠硬功夫了，以下是幾招常見的標準招式：

- Plug-in Architecture
- Layered Architecture
- Model-View-Controller
- Service-Oriented Architecture
- Component-Based Development

前面三招是目前 Teddy 的最愛...

***

友藏內心獨白：歡迎喜歡吃「軟飯」的人加入軟體開發行列，讓軟體變得更軟 !@#$!#%&

# 60 設計最難的部份是什麼？

December 20 22:28~23:20

12/22 07:34-07:

原文發表於 http://teddy-chen-tw.blogspot.com/2011/12/blog-post_20.html.

最近部落格都在講 Teddy 看病和找工作的事，今天主題回到軟體上面來。Teddy 去年九月寫過一篇「需求分析書中最重要的資訊是什麼？」不知道鄉民們還記得答案嗎？今天想談一個類似的問題「設計最難的部份是什麼」？

請鄉民們花一分鐘想一下這個問題，或是我們把原來的問題簡化一下，改成「**設計軟體最難的部份是什麼？**」 是如何寫 stories or use cases、如何套用 design pattern、如何設計 software architecture、如何用 UML 畫出漂漂亮亮嚇死人不償命的圖、如何 coding、如何 testing、還是如何喝酒...Orz...（問業務就知道最後一項技能對於結案的重要性）

一分鐘差不多到了，公佈答案，這個答案是從 *The Design of Design* 這本書第 22 頁抄過來的：

*The hardest part of design is deciding what to design.*

(設計最難的地方在於決定要設計什麼東東)

假設鄉民們先暫時同意作者 Brooks 老先生在書中的說法，那麼重點在後面（23 頁）的幾句話：

*Then, slowly, I came to realize that the most useful service I was performing for my client was helping him **decide** what he really wanted.*

這 一段話 Teddy 最近看了更有感覺，為什麼？因為最近 Teddy 在思考未來的職涯規劃，有人可能看上 Teddy 的 Scrum 與 agile practices 經驗，有人可能覺的 Teddy 的 software architecture 設計能力應該還不錯，也有人看上 Teddy 去上過 CMMI 課程而想找一個懂 CMMI 的 PM（此人一定沒看過搞笑談軟工...XD）。其實 Teddy 這兩個禮拜也一直在問自己同樣的問題：Teddy 能夠做什麼？

不過剛剛在擠料的時候，隨手翻到這本書，看到這段文字，嗯，覺的上面這段話的「abstraction（抽象化）」做的真好，一言以蔽之可以代表 Teddy 的心聲（至少 Teddy 內心希望是這樣啦）。

***

既然設計最難的地方在於決定要設計什麼東東，那是不是說「需求

定義」很重要？Yes，可以這麼說。那是不是說回歸到傳統軟體開發流程告訴我們的：

需求分析 → 架構分析 → 設計 → 寫程式 → 測試

這種流程，所以在專案開始實做之前要把所有的需求都找出來？

Brooks 老先生很好心的告訴鄉民們，No. 一樣還是第 23 頁：

*Not only is the design process iterative; the design-gold-setting process is itself iterative.*

*... knowing complete product requirements up front is a quite rare exception, not the norm.*

其實這本書看到這邊 Brooks 老先生就被 Teddy「看破手腳」了，這根本是另類的敏捷方法代言人啊。

***

對一個軟體專案來講，定義需求的確是不容易的一件事，尤其如果要處理的問題領域（problem domain）本身就非常複雜，那麼光是要

把「what to design」...的雛型...搞清楚，就夠花時間了。更難的還在後面，由於「design process」與「design-gold-setting process（搞清楚要設計什麼的流程）」都是 iterative（就是要跑個好幾次，一步一步的來，才能把設計，以及到底要設計什麼搞懂），所以軟體開發才會常常被弄的很亂。

上面這段有點玄，請多看兩次。

Teddy 再補充一下，因為需求隨著時間越來越清楚，換句話說需求隨著時間一直在變（由模糊變清楚，或是由錯誤變正確），而相對的設計本身因為需求變了所以也要變。所以，如果軟體沒有「設計好」（或是說專案沒有帶好），那麼在這種「**需求與設計的愛恨情仇交互作用之下**」，最終的產品就不容易成功。

***

結論就是，老王賣瓜一下：做軟體要找到像 Teddy 這種「品德兼修（需求與設計）」都熟的人來帶專案，要不成功也難啊。

***

友藏內心獨白：最近突然覺的書看太多好像沒什麼鳥用，英文學好比較重要...Orz。

# 61 Program to an interface

06/22 22:00～23:20

12/12 15:11-15:18

原文發表於

http://teddy-chen-tw.blogspot.com/2010/06/program-to-interface.html.

看過 Design Patterns 這本書的人，一定會記得第 18 頁的這一句至理名言：

**Program to an interface, not an implementation.**

讓我們看一下書中這一句的前後文：

*There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:*

1. *Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.*
2. *Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.*

*This so greatly reduces implementation dependencies between subsystems that it leads to the following principle of reusable object-oriented design:*

*Program to an interface, not an implementation.*

*Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class.*

先用白話文解釋一下這句話，假設鄉民們要幫公司在資訊展上面找 show girls 站台，物件導向技術學得好的人就知道要符合 **program to an interface** 的精神來開出條件給公關公司找人。

有酷似「豆花妹」甜美的笑容，媲美「遙遙」殺很大的身材，外加「林志玲」的娃娃音。

這樣子應該很容易可以找到一票符合條件的 show girls。反之，如果是 **program to an implementation**，就會變成：

我要「豆花妹」,「遙遙」, 外加「林志玲」。

都已經「**指名購買**」了，再怎麼找全台灣也就是這三個人。萬一當

天有人臨時生病，可是連個替代人選都沒有（所以好的設計要避免 implementation dependencies）。

因此，program to an interface 的好處就很明顯了：

- 不管黑貓，白貓，只要能抓老鼠的（符合 interface）就是好貓。也就是說，可選擇性變多了，不會被綁死在某一個人或是品牌上面。
- 有一天發現世界上有一種比黑貓和白貓更厲害的「熊貓」，只要符合介面直接換掉程式都不用改。

結論：**Thinking about what you can do rather than who you are**（忘了這句從那抄來了…Orz）.

***

友藏內心獨白：最難的就是定義 interfaces 啦。

# 62 Design Patterns 分成三大類

06/24 22:47~23:55

12/12 15:20-15:28

原文發表於 http://teddy-chen-tw.blogspot.com/2010/06/design-patterns.html.

今天 Teddy 有點累，談點簡單一點的主題：「Design Patterns 的分類」。

鄉民甲：我知道，就是 Creational、Structural、Behavioral 這三類的啦。

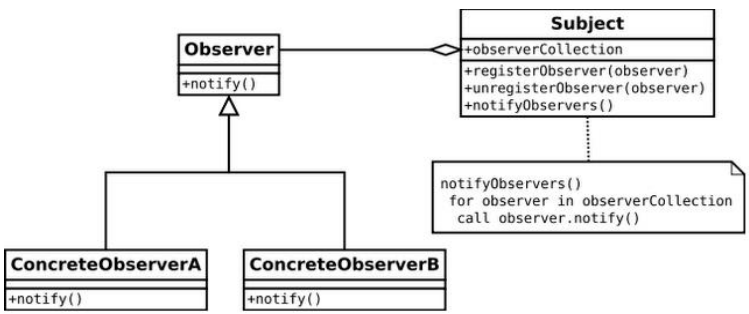
**全錯**。如果是這三類，鄉民自己看 GoF 的書就好了，用不著 Teddy 出馬。Teddy 要介紹的這三類，是出自於 Wolfgang Pree 所撰寫的 *Design Patterns for Object-Oriented Software Development* 這本書中第 98 頁（沒看過吧，嘿嘿嘿）。

- Patterns relying on **abstract coupling**
- Patterns based on **recursive structures**
- Other patterns

GoF 的 Design Patterns 這本書中大多數 patterns 都是屬於第一類的，例如 State、 Factory Method、Observer、Bridge、Builder、Command、Visitor、Interpreter、 Mediator、Adapter、Prototype、Proxy、Strategy。

手邊有書的人翻開看一下，或是看下圖，Observer pattern 中的 Subject 和 Observer 是有 **coupling**，也就是彼此相依。但是，軟體設計最基本的兩個法則其一就是要**降低 coupling**（另一個是要**提高 cohesion**），但是 coupling 又不可能降低到零，否則所有物件彼此都沒任何關係，玩不下去了。為了求得平衡點，這個關係就要想辦法搞成「有點黏，又不會太黏」，於是乎誕生了 **abstract coupling**。要 coupling，OK 啊，但是只能有「**抽象關係（精神外遇？）**」，不能有 ~~肉體耦合~~ **實做耦合（implementation coupling）**。

有準時收看 Teddy 部落格的鄉民們，有沒有發現，這就是 **program to an interface, not an implementation** 的例子啦。

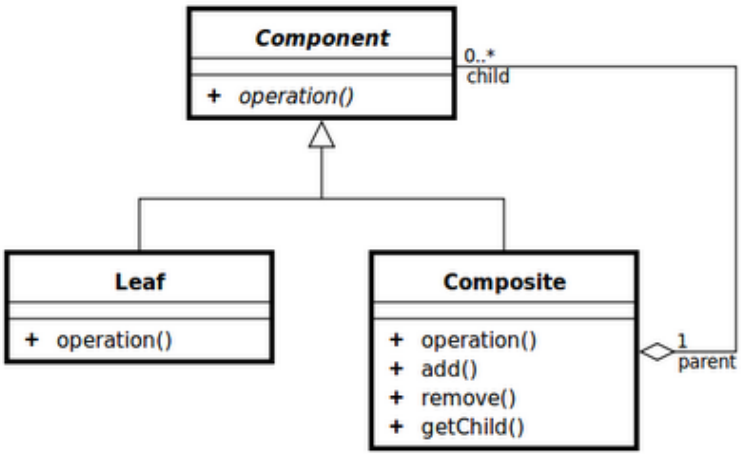



圖片來源：

http://upload.wikimedia.org/wikipedia/commons/8/8d/Observer.svg

第二類，recursive structures 有 Composite、Chain of Responsibility、Decorator。應該不用再多做解釋了，這一類的 patterns 就是有遞迴

結構。




圖片來源：

http://upload.wikimedia.org/wikipedia/commons/5/5a/Composite_UML_class_diagram_%28fixed%29.svg

第三類，其他。ㄟ，通常分類到最後分不下去了都會跑出這一類。屬於這一類的有 Abstract Factory、 Flyweight、Singleton、Template Method。

至於知道這些分類有什麼用？至少改天鄉民們想自創武功的時候，可以從 **abstract coupling** 和 **recursive structures** 作為出發點來思考，這總比 Creational、Structural、和 Behavioral 分類要具體一點吧。

***

友藏內心獨白：還有另外一個用途就是可以拿來寫部落格啊。又混過一篇...safe。

# 63 時間到

April 13 21:08~22:19

12/15 09:49-10:14

原文發表於 http://teddy-chen-tw.blogspot.com/2011/04/blog-post.html.

相信鄉民們一定有那種和別人約好要見面，可是卻苦等不到對方的經驗。如果是男女朋友約會，通常是男方比較早到，女方遲到個 30 分鐘算是正常，遲到 1-2 個小時也是剛剛好而已。在熱戀中的男方，心目中的 timeout 預設值應該要有「大於 N 小時」的覺悟，但是一旦~~騙到手~~ 結婚之後，timeout 時間可能會被調整成「小於 5 分鐘」，真是太現實了。

不管如何，具有 timeout 的觀念是很重要的，真實世界沒有什麼**「不見不散」**,**「等你一生一世」**的那種事（迷之音：李大仁，你在那裡…），像「王寶釧」這種把 timeout 設成「18 年」以上的人，已經算是瀕臨絕種的稀有寶物了。

***

Timeout 對於程式設計師而言也是很重要的觀念，之前 Teddy 在「還少一本書：Release It! Design and Deploy Production-Ready Software 」有稍微介紹一下書中 **Use Timeouts** 這個 stability pattern。今天 Teddy

解了一個 bug，剛好也是和 timeout 有關，順便再幫鄉民們複習一下。

這個 bug 和使用 Jav 建立 socket 有關，假設你在寫網路應用程式，有下列幾個步驟：

1. client 建立一個 socket 連到 server。
2. client 透過 socket 得到一個 output stream，利用這個 output stream 送出 request。
3. client 透過 socket 得到一個 input stream，利用這個 input stream 讀取 server 傳回來的 response。

以上為一個很典型的網路應用，由於 Teddy 有看過 Release It 這本書，知道開發網路應用程式要套用 Timeouts 這個 pattern。經過分析，上述第 3 個步驟可能會因為 server 很忙以至於過了很久都沒有把結果傳回來，導致 client「卡住」，很容易讓使用者以為程式當掉，所以這邊就要利用到 timeout 機制。還好 Java 都有幫鄉民們考慮到這個問題，Socket 物件有一個 setSoTimeout method ，看一下 JavaDoc 的說明：

**setSoTimeout**

public void **setSoTimeout**(int timeout)

throws SocketException

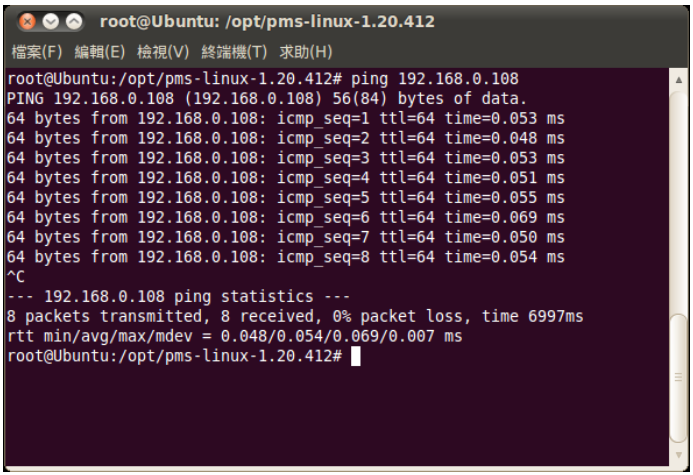
Enable/disable SO_TIMEOUT with the specified timeout, in

milliseconds. With this option set to a non-zero timeout, a read() call on the InputStream associated with this Socket will block for only this amount of time. If the timeout expires, a **java.net.SocketTimeoutException** is raised, though the Socket is still valid. The option **must** be enabled prior to entering the blocking operation to have effect. The timeout must be > 0. A timeout of zero is interpreted as an infinite timeout.

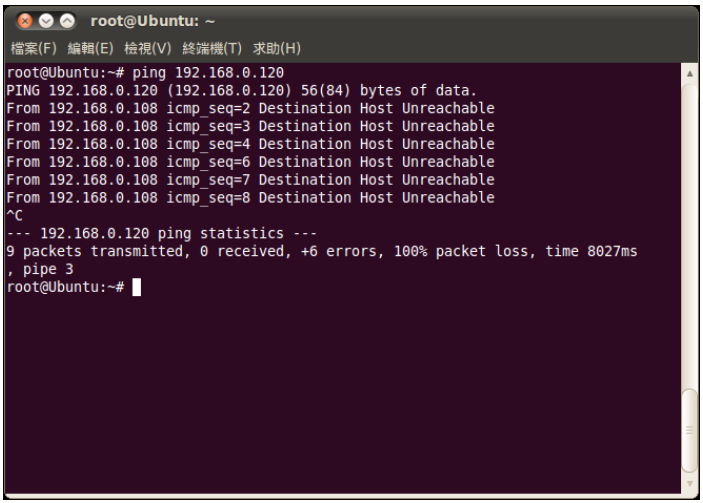
**Parameters:**

timeout - the specified timeout, in milliseconds.

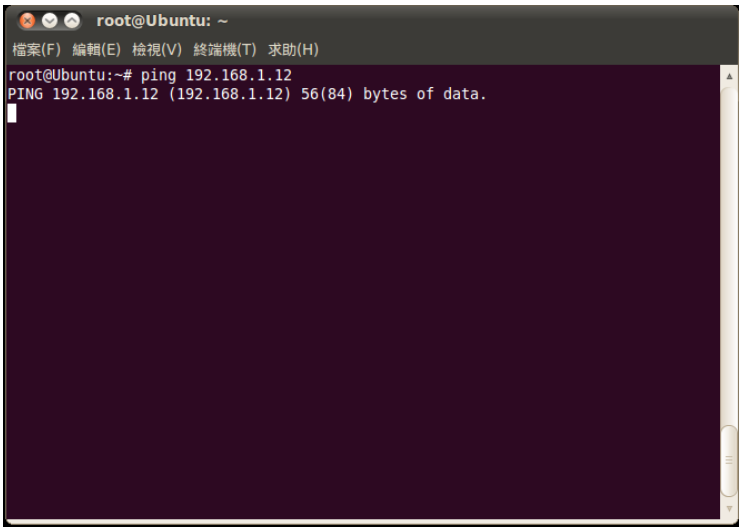
以上 JavaDoc 說明已經很清楚了，但是，只有 read 會卡住嗎？Teddy 今天遇到的問題就是卡在個步驟 1。Teddy 用一個簡單一點的例子來說明，假設你的程式 ping 某個 IP address，在正常的情況下，**很快就會有回應**，如下圖所示。




如果 ping 一個沒人使用的 IP address，應該會出現如下的結果，**也是很快就會有回應**。



但是，如果隨便 ping 一個 IP address，整個程式就會卡住，如下圖所示。




如果你的程式是呼叫 public Socket (InetAddress address, int port) 來建立 socket，那麼就可能在個步驟 1 卡住。看一下這個 constractor 的 JavaDoc：

public **Socket**(InetAddress address,
int port) throws IOException

Creates a stream socket and **connects** it to the specified port number at the specified IP address. If the application has specified a socket factory, that factory's createSocketImpl method is called to create the actual

socket implementation. Otherwise a "plain" socket is created.

使用這個 constractor 當 Socket 物件被建立的時候，同時也 connect 到遠端。仔細看一下 Socket 物件所提供的 9 個 constractor，沒有一個可以指定 timeout 的，所以程式要改成下面這樣。

1. 使用這個 Socket() 不帶參數的 constractor，根據 JavaDoc 的說明這個 constractor「Creates an **unconnected** socket」。
2. 使用 connect (SocketAddress endpoint, int timeout) 來建立連線，根據 JavaDoc 的說明「Connects this socket to the server **with a specified timeout value**. A timeout of zero is interpreted as an infinite timeout. The connection will then block until established or an error occurs. 」。

故事就這麼簡單，程式改完之後建立連線時就不會一直卡住了。由於 connect (SocketAddress endpoint, int timeout) 是在 JDK 1.4 版之後才出現的，所以鄉民們到網路上亂找範例的時候，可能找到類似呼叫 Socket(InetAddress address, int port) 這樣的範例（產生 Socket 物件並建立連線）就給它直接抄過來用，所以這類的程式碼（建立 connection 時沒有指定 timeout）其實還滿常見的。

***

如果講到這裡就結束那就遜掉了，還有一個重點是**這種 bug 其實不太好找**，為什 麼？因為同樣的 code 在 Windows 上面並不會有問題，而在 Linux 上面卻會。不要傻傻以為 Java 是跨平台，Java 底層很多程式還是呼叫 native code，網路程式就是一個例子，所以有一些行為還是跟作業系統相關的。總之，跟著 Teddy 大聲念三遍：

**寫網路應用程式的時候要記得 Use Timeouts。**

***

友藏內心獨白：如果要害一個人，就送他一張不限航點的飛機艙位升等券...這一句有緣人才看得懂...XD

# 第七部 HCI

# 64 窮人 HCI 設計入門

Feb. 23 21:01~22:35

12/22 08:38-08:55

原文發表於 http://teddy-chen-tw.blogspot.com/2011/02/hci.html.

如果鄉民們和 Teddy 一樣都是程式設計師的話，在開發軟體的時候一定會遇到一個問題：「**使用者介面要如何設計？**」平平都是做手機的，人家 Apple 的 iPhone 使用者介面就設計的那麼好，而我們的卻那麼...鳥...（這一句要用唱的：人家的介面是真正的介面，我們的介面是人家不要的我們把它撿起來...）。你可能會不服氣的說，人家 Apple 或其他國際級的大公司，都有專門研究「人機介面」或是「使用者介面」的人，設計的好是應該的。好吧，有鑑於台灣都是中小企業居多的這個特性，想要和這些財閥相比是很難滴，但是軟體還是要寫啊，就算不是要賣錢的，只是給公司內部使用的系統，軟體介面不好用也是會常常被使用者念個不停甚至被恥笑。

Teddy 相信這個問題一定困擾著不少的程式設計師，因為你可以把程式寫得很好，但是卻不一定知道要如何把「使用者介面設計的很好（容易）使用」。有些開發團隊雖然有所謂「設計介面」或是「做網頁的人」，這些人雖然可以把「畫面做出來」，但卻不一定具備設計容易使用介面的知識與能力。從軟體開發的角度來看，研究這個問

題的領域叫做「人機互動」(Human-Computer Interaction, HCI) 或是「人機介面」(Human-Computer Interface, HCI)。由於台灣人常常抱持著「~~先研究不傷身體，再講求藥效~~ **先讓軟體可以動起來，再研究好不好使用**」 的心態，因此在人力不足的情況下，軟體的介面設計的工作常常落到 developers 的身上。而 developers（不要看別人，就是你啦）在缺少對於 HCI 相關訓練的情況下，要設計出「有質感又超級貼心的介面」的機會大概比中統一發票特獎的機率還要低一點。

假設你是 team leader，由於上輩子燒好香，在你的團隊中擁有所謂「做網頁的人或是設計 UI 的人」，你是否常常會覺的這些人的「網頁（UI）設計的不夠好」但是卻因為自己缺少 HCI 的知識而經常有**「就算想開罵卻不知從何罵起」**的遺憾…Orz（友藏內心獨白：讓恁爸來罵一定可以罵到會「牽絲」)。來來來，有以上困擾的鄉民們，今天 Teddy 介紹這一帖藥，服用後只要「光呼吸就可以達到減肥的效果」，保證一週內最少可廋 5 公斤...XD。

藥方就是 Jenifer Tidwell 的...知道這個作者的人會以為 Teddy 要介紹 ***Designing Interfaces*** 這本書，答錯。這本書太厚了，看不下去（而且要錢)。這邊有免費的文章可看：

COMMON GROUND: A Pattern Language for Human-Computer

Interface Design

這篇文章顧名思義就是要介紹好幾個 HCI 的 patterns。可能是 Teddy 看 patterns 看習慣了，總覺的 Jenifer Tidwell 的這篇文章寫得比 Designing Interfaces 這本書要容易閱讀。Jenifer Tidwell 在文章中將她所提出來的 HCI patterns 分了好幾的類別，每一個 pattern 的內容就請鄉民們自己花時間去看（Teddy 可以幫鄉民們們省錢但是卻沒辦法幫鄉民們省時間啊），在這邊看一個 Teddy 覺的很有用的分類和其所包含的 patterns 與每一個 pattern 的 solution：

How does the content or available actions unfold before the user?

- **Navigable Spaces**：Create the illusion that the working surfaces are spaces, or places the user can "go" into and out of.
- **Overview Beside Detail**：Show the whole set of objects, or the whole undetailed data set, in one part of the display area, to act as an overview of the content.
- **Step-by-Step Instructions**: Walk the user through the task one step at a time, giving very clear instructions at each step.
- **Small Groups of Related Things**: Group the closely-related things together, nesting them in a hierarchy of groups if needed.
- **Series of Small Multiples** (unwritten): 歹勢，作者沒寫。
- **Hierarchical Set**: Show the data in a tree-like structure.

- **Tabular Set**: Show the data in a table structure.
- **Chart or Graph**: Show the data plotted against time or some other variable. Plot it together with other variables for further comparison
- **Optional Detail On Demand**: Up front, show the user that which is most important and most likely to get used. Details and further options which won't be needed most of the time -- say 20% or less of expected uses -- can be hidden in a separate space or working surface (another dialog, another piece of paper, behind a blank panel).
- **Disabled Irrelevant Things**: Disable the things which have become irrelevant.
- **Pointer Shows Affordance**: Change the affordance of the thing as the pointer moves over it.
- **Short Description**: Show a short (one sentence or shorter) description of a thing, in close spatial and/or temporal proximity to the thing itself.

以 **Step-by-Step Instructions** 為例，這個 pattern 所要解決的問題是：

How can the artifact unfold the possible actions to the user in a way that

does not overwhelm or confuse them, but instead guides them to a successful task completion?

也有人把這樣的 pattern 叫做 **Wizard**（精靈），講這個名字大家就瞭了。當各位在設計使用者介面，或是審閱別人設計的介面時，如果了解了這些常用的 HCI patterns，至少就有一種可以用來評估的標準。舉一個最簡單的例子，假設你的程式有一個 Print 的功能，可是使用者的電腦並沒有裝印表機。「照理講」一般正常的地球人在此情況下會把這個 Print 功能（例如畫面上的 Print button）給 disable。但是，恰巧你聘請的是遠從火星來的 programmer，沒有把 Print 功能給 disable。

路人甲： 有人這麼白目的嗎？

Teddy： 別懷疑...

相信鄉民們都是受過高等教育的地球人，當遇到這樣的火星人一定要用有教養的方式來教育他。

地球人： 你是「丁丁」啊... 說溜了嘴，重來一次愛心版本。來來來，靠...近一點，先來個愛的抱抱，再讓我教你，這邊應該要套用 **Disabled Irrelevant Things** 這個

pattern 喔。

火星人： 喔...嗯...那泥...原來還有這一招...偶了解啦...（偶是許 x 美，不是丁丁啦）。

地球人： 一定要幸福喔...XD。

以上...差不多就是這個意思了。

***

不知道鄉民們有沒有注意到，這篇文章的標題叫做 COMMON GROUND，看到這裡應該就了解了吧，如果不知道讓 Teddy 說給你聽。在還沒有學習這些 HCI patterns 之前，地球人和火星人在溝通使用者介面上遇到困難。但是，雙方都學習了 HCI patterns 之後，就有一個「**共通的基礎**」可以相互討論了。

看到這邊一定會有鄉民們想說：「我聽 Teddy 你在哪邊放風吹（風箏）」，實際上 UI 設計哪有那麼簡單。是沒那麼簡單，HCI 還包含了很多議題，例如 task analysis 和 usability 以及設計精美的小圖示。但是在練會絕世武功之前，好歹先買把「小扁鑽」隨身攜帶防身，不然是很難在「~~艋舺~~ 軟體界」立足滴。免錢的加減用一下。

***

友藏內心獨白：有便宜的牛內還不趕快搶。

# 65 歪批 GOMS

Feb. 24 22:25~23:46

12/22 08:56-09:11

原文發表於 http://teddy-chen-tw.blogspot.com/2011/02/goms-1.html.

昨天在「窮人 HCI 設計入門」提到了使用 patterns 來設計與分析使用者介面的方法，剛剛在「擠料」的時候，突然想到了好幾年前曾經看過某篇介紹用 **GOMS**（Goals, Operators, Methods, and Selection rules）來分析使用者介面設計好壞的論文。由於年代過於久遠那篇論文到底跑去哪裡 Teddy 一時也找不到，只好匆匆到 Wikipedia 上面惡補一下。

先說明一下這四個名詞：

- **Goals**: 使用者想要達成的目標，例如，一個文書處理器（word 或是小作家）的使用者，想要「列印文件」。「列印文件」這件事就是使用者的 goal。
- **Operators**: 使用者為了達成某項 goal 所需要執行的（單一最小） action （動作），以「列印文件」而言，使用者所需要的 operator 可能包含了:
    - move mouse
    - click mouse left button

- release mouse left button

- **Methods**: 為了達到 goal 所執行的一連串 operators 就稱為一個 method。一個系統同時間可能會提供好幾種不同的 methods 來達成同一個 goal。以列印文件為例，
  - Method 1: 用滑鼠透過 tool bar 來列印文件
    - 移動滑鼠到 tool bar 上面 Print 這個 icon 上
    - 按下滑鼠左鍵
    - 放開滑鼠左鍵
  - Method 2: 用滑鼠透過 menu 來列印文件
    - 移動滑鼠到 file menu
    - 按下滑鼠左鍵
    - 移動滑鼠到 print item
    - 按下滑鼠左鍵
    - 放開滑鼠左鍵
  - Method 3: 用鍵盤透過 menu 來列印文件
    - 按下 alt-f
    - 放開 alt-f
    - 按下 p
- **Selection rules**: 如果同時存在一種以上的 method 都可以達成相同的 goal，則我們透過 selection rule 來描述在何種情況下應該選用哪種 method 會比較合適。

由於 GOMS 所描述的最小單位是 operator（按下鍵盤上的一個 key，移動一次滑鼠都算一個 operator），因此我們只要很簡單的去「加總」某個 method 所包含的 operators 就可以比較出用該 method 來達成某個 goal 是否合適，或是還有沒有改善的空間。

***

再舉個例子，看一下 Teddy 家裡 Firefox 所顯示的「印列對話盒」：


從這個畫面可以看出來 Teddy 有三個印表機設定（實際上有兩台印

表機)，由於「印列對話盒」會記住 Teddy 的預設印表機，因此當 Teddy 要透過這個畫面列印文件的時候，只需要：

method 1：

- 移動滑鼠到「列印」按鈕上
- 按下滑鼠左鍵
- 放開滑鼠左鍵

或是 method 2：

- 按下 alt + p
- 放開 alt + p

但是，如果「印列對話盒」沒有記住 Teddy 的預設印表機，那結果就變成：

method 1：

- **移動滑鼠到預設印表機上**
- **按下滑鼠左鍵**
- **放開滑鼠左鍵**
- 移動滑鼠到「列印」按鈕上
- 按下滑鼠左鍵
- 放開滑鼠左鍵

硬是多了三個 operators

或是 method 2：

- **按下 tab 讓 focus 移到選擇印表機的這個 UI 元件**
- **放開 tab**
- **按下「方向鍵」(向上或向下） 移到所要選擇的印表機上**
- **放開「方向鍵」**
- 按下 alt + p
- 放開 alt + p

活生生又是多了四個 operators

***

路人甲：一定要玩到這麼「龜」嗎！連一次滑鼠移動或是按一個按鍵都要計較。

沒錯，就是要「斤斤計較」。通常開發人員可能只有在開發軟體的時候會使用自己所開發的軟體，在那之後就很少會用到。但是，你的客戶很可能是每天都要用好幾個小時以上。就算是使用者為了達到某個 goal（例如，新增一筆客戶資料）只增加三次不必要的 operators（例如，移動滑鼠、按下滑鼠右鍵、放開滑鼠右鍵），萬一這個使用

者一天要新增 100 筆客戶資料，那他就活該倒楣要多操作 3 * 100 = 300 operators。

這還只是為了完成一個 goal 而已，一整天下來如果要完成 N 個 goals，這個倒楣的使用者不腰酸背痛才有有鬼。長久下來，不到 35 歲就需要去看復健科門診了。

很多使用者介面，雖然好像只是「沒有把 cursor 移到預設的位置」，「沒有設定正確的 tab order」,「沒有 focus 在正確的 UI 元件上」這種小事，對於經常使用該軟體的使用者而言卻是大事。Teddy 還記得第一次跟 Kay 借用 iPhone 來玩，那種「增一分太肥，少一分太瘦（恰到好處）」的感覺，不禁發出「為什麼人家的使用者介面可以設計的那麼好用」的感嘆。相較起使用某些人設計的軟體，卻發出了「為什麼用完之後 Teddy 的手肘那麼的酸痛」的感嘆... XD。

***

程式寫得再美，使用者不會去幫你做 code review，看不到你的「內在美」，所以還是賣不出去。介面設計的好，使用者用起來很爽，就有賣點了。也許這樣講不盡公平，你會說介面設計的好但是玩兩 分鐘就當機了啊。但是，至少你的客戶願意花兩分鐘來玩。介面設計的爛，客戶可能只願意花「一眼」，連動手都省了。

***

友藏內心獨白：程式設計師除了寫程式以外還是要懂一些 HCI 設計方法，「小扁鑽」還是有需要滴，捅人自捅兩相宜。

# 66 HCI 分類開張: Designing for Error (1)

Feb. 26 22:24~23:36

12/22 09:15-09:32

原文發表於

http://teddy-chen-tw.blogspot.com/2011/02/hci-designing-for-error-1.html.

雖然不是 HCI 專家，但前兩篇（窮人 HCI 設計入門 ，歪批 GOMS (1) ）Teddy 剛好談到這個主題，剛剛在「擠料」的時候，就想那乾脆把 Teddy 所知有限的 HCI「小扁鑽」介紹給鄉民們好了，順便幫 Teddy 的部落格建一個 HCI 分類，把前兩篇文章從「軟工」移到「HCI」這個分類。

***

今天談談**「使用者犯錯」**這個問題（大哥內心獨白：我犯了全天下男人都會犯的錯誤）。Teddy 曾經在 Discovery 上看到一個節目，討論英國某航空公司發生空難的案例。內容大概是該航空公司的某架飛機駕駛艙前方（或側面，有點忘了）的玻璃在飛機飛行中突然整片飛出去，導致飛機在高空中失壓。事後調查結果發現，該飛機在前天晚上才剛剛執行過一次例行的保養。為了避免「金屬疲乏」，技師才全部更換過用來鎖住玻璃的螺絲。最後的調查結果發現，是這些

螺絲出了問題，因為它們的尺寸不合，小了一號（可能只差了 0.01 公分之類的）。

為甚麼螺絲會小一號？是買到「山寨版螺絲」嗎，還是用到「瑕疵品」，亦或是原廠裝貨的時候搞錯了尺寸？。都不是，原因是因為技師沒有依照「標準流程」拿取螺絲。標準流程規定要跟換螺絲時，要先去翻閱技術資料，確認飛機型號與其所使用螺絲的產品編號，然後在到領料處依據個產品編號來找出所要更換 的螺絲。

如果這件事情發生在寶島台灣，鄉民們認為應該如何「處置」這位技師？

- 人肉搜索，把他的祖宗八代的資料全部公佈在網路上。
- 偷懶不按標準流程做事，危害飛安，抓起來槍斃...XD。
- 找一群名嘴在政論節目中連續痛罵他三天，並開放「當事人澄清專線」讓他 call in 為自己辯解，之後繼續狠批他的辯解。

還好這個事件是發生在英國，這位技師安然保存住性命。言歸正傳，為什麼技師不依據「標準流程」辦事，經過後續的調查發現：

- 該技師已經有多次換過同型飛機的螺絲的經驗，因此他自認為不需要在逐一確認詳細的飛機型號與螺絲編號。

- 該技師平常的工作量相當大，而且當天已經**工作超時**（A 去踢嘻，看到沒，工作超時是很危險滴）。在更換那架飛機螺絲的時候（當天最後一項工作），已經是深夜了，他想儘快完成後回家。
- 因此，技師就把舊的螺絲拆下，拿到庫房去，用「肉眼」比對找出大小一樣的螺絲。沒想到這些新的螺絲肉眼看起來雖然和舊的一樣，但是實際上卻小了一點點。
- 原本這一點點的差異並不會造成問題，巧就巧在飛機駕駛艙前方鎖住玻璃的那塊金屬已經有一個肉眼看不到的細小裂縫，在高空因為壓力，溫度等問題，最後造成整片玻璃脫落。

最後，航空公司並沒有特別懲罰該名技師，反而是檢討公司原本維修飛機的流程，來避免問題再次發生。

***

路人甲：這和 ~~hT 嘻~~ HCI 有何關係？

想一下，身為開發人員，當有人向你回報說：「客戶操作我們的軟體，出現了 xxx 錯誤...」，你的典型反應可能是：**「客戶不會用啦，都沒在看使用手冊的」**，**「這個地方我上次就告訴他要注意了啊，怎麼又犯錯了」**。這些當然是很方便的回答，不用改任何軟體，反正就是客戶

白爛就對了，先叫他們立正站好聽訓然後回家深切反省並好好 ~~讀書~~ 讀手冊。也許有些情況下的確是如此，但如果總是以這樣的心態來思考客戶的問題那麼軟體就不可能進步（Teddy 內心獨白：使用手冊有寫這句話最近 Teddy 也常常掛在嘴邊...XD）。

軟體（或任何設計）應該要 Designing for Error，具體的作法在 ***The Psychology of Everyday Things***（p. 131）這本書有提到。

- Understand the causes of error and design to minimize those causes.
- Make it possible to reverse actions- to "undo" then - or make it harder to do what cannot be reversed.
- Make it easier to discover the errors that do occur, and make them easier to correct.
- Change the attitude toward errors. Think of an object's user as attempting to do a task, getting there by imperfect approximations. Don't think of the users as making errors; think of the actions as approximations of what is desired.

明天要早起，有人在催 Teddy 準備就寢了，就先寫到這裡，明日繼續。重點是，心態轉變，「問題（錯誤）」也可能變成改善的好朋友。

***

友藏內心獨白：不思考問題發生的原因並從根本加以改善，直接把犯錯的人弊了還是無法根絕問題。總不能把兩千三萬人都弊了吧。

# 67 Designing for Error (2)

Feb. 27 21:32~22:58

12/22 09:33-09:44

原文發表於

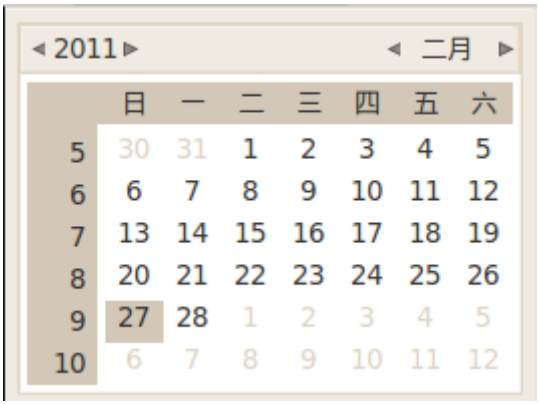
http://teddy-chen-tw.blogspot.com/2011/02/designing-for-error-2.html.

昨天談到 The Psychology of Everyday Things（p. 131）提到的四點 Designing for Error 作法：

- **Understand the causes of error and design to minimize those causes**.

了解錯誤發生的根本原因，在設計時盡可能避免這些錯誤發生。**換句話說，這是 error prevention**。一 個最簡單的例子，如下圖所示，用行事曆的 UI 讓使用者選擇一個日期，這樣可以避免使用者打出 2011/02/29 這種不存在的日期。這個方法方法聽起來好像很簡單，如果有點年紀的鄉民們回想一下，在 Web 才剛剛流行的那個年代，透過網頁輸入「日期」的資料，大多還是只提供一個 text box 讓使用者自行輸入類似「2011/02/29」的期，或是三 text box 讓使用者自行輸入年、月、日。這種作法持續了一段時間，後來才有用 Javascript 做的行事曆元件可用。所以，雖然「Understand the causes of error and design to minimize those causes」可以算是常識，但是在設計使用者介

面時常常因為種種原因很容易被忽略。

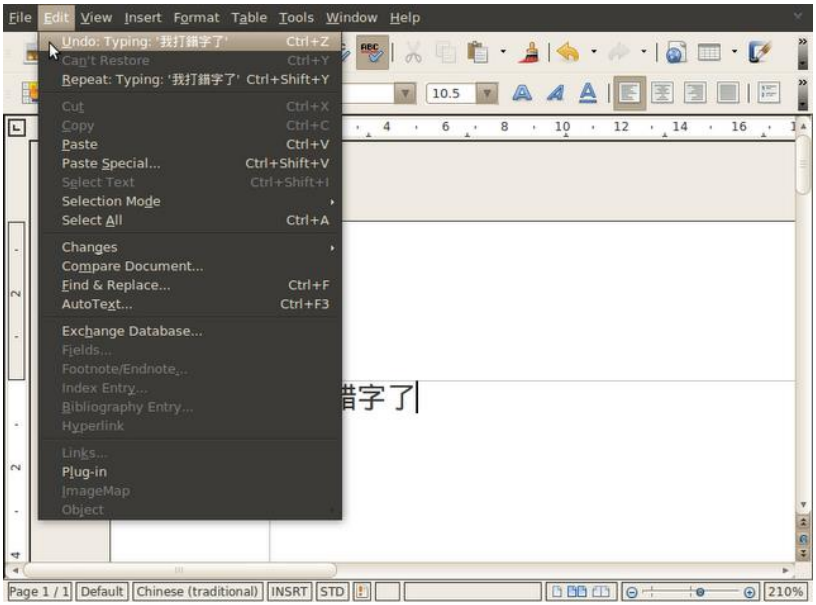



- **Make it possible to reverse actions- to "undo" then - or make it harder to do what cannot be reversed**.

既然犯錯是難免的，如果有「月光寶盒」或是「時光機」可以回到過去「修正」自己所犯的錯誤，修正之後錯誤就不存在啦。**換句話說，這是 error recovery。**最常見的例子就是文書編輯軟體都有提供的無限次數 undo 功能，因為有了這個功能，使用者可以「無罣礙的（放心的）」編輯文件，反正萬一改錯了，只要 undo 就可以回到上一個狀態了。**如果人生也能有 control+z 該多好**。

如果有些操作和人生一樣是無法 undo（復原）那該怎麼辦？例如，

鄉民們使用網路 ATM 轉帳，網路 ATM 軟體人總不可能提供 undo 功能，讓鄉民們們在「不小時把錢轉給詐騙集團時」可按下 undo 按鈕吧。雖然這個功能聽起來很不錯，不過轉帳是不可逆也不可以 undo 的，而且不小心轉錯會「很傷」，因此，當使用者執行這類的功能時，軟體都會「再三提醒」，這也是這個作法第二部份所說的 **make it harder to do what cannot be reversed**。




- **Make it easier to discover the errors that do occur, and make them easier to correct**.

如果錯誤無法從根本上（在設計的時候）避免，那麼至少要能夠想

方法讓「偵測錯誤」與「修復錯誤」變的簡單一點。舉個車子的例子，車子製造商無法 用設計的方法「避免」使用者「忘記加油 （對車子而言這沒油了是一種錯誤）」，所以，他們設計了「油量指示表」，而且主動在「油量只剩下 10% 的時候亮紅燈或是發出 BBB 警告聲」（discover the errors），如果車子有 GPS 此時能夠自動結合 GPS 告訴車主最近的加油站在哪裡那就更棒了（make them easier to correct）。

再舉個軟體的例子，假設鄉民們正使用 Eclipse 用 Java 在寫程式。大家都知道，**「寫程式是一件超級容易犯錯的工作」**，Eclipse 的設計者如何幫助使用者（就是你）來 discover the errors 而且 make them easier to correct？人家想出了 **quick fix** 這個超讚的功能。以下圖的程式範例為例，Eclipse 偵測到一個語法錯誤（main method 遇到一個 checked exception 但是沒有 catch 或是 declare 這個 checked exception），在錯誤發生的那一行的左邊顯示了一個小 X icon 稱之為 **marker**（以上是 easier to discover）。接下來是 easier to correct 的作法，當使用者按下這個小 X icon 之後，Eclipse 顯示建議修改方案（**resolution**），使用者點選合適的修改方案 Eclipse 便自動幫忙修復這個錯誤。

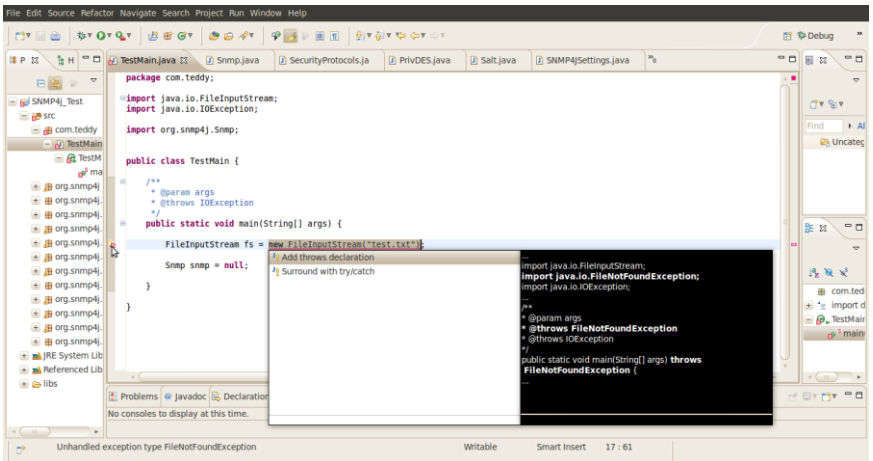
當年 Teddy 曾經研究過 Eclipse 的 quick fix 機制，大體上就是：

- 利用 visitor 去拜訪 Eclipse JDT（Java Development Tools）所建立的 AST (Abstract Syntax Tree)。Error detection 的邏輯就

是寫在 visitor 裡面，plug-in 的開發者可以自行擴充新的 visitor 以便加入自己的錯誤檢查邏輯。

- 當發現錯誤時，產生一個 ~~麵包屑~~ marker 用以標示錯誤發生的「位置」（程式行號）。
- 針對不同種類的 marker，plug-in 開發者提供自定的 resolution（解決方案）以協助使用者排除錯誤。由於 marker 是 plug-in 開發者依據「自訂檢查錯誤邏輯」所產生的一個「錯誤標記」，因此如果此錯誤「存在可能的解決方案」的話，plug-in 開發者自然可以事先提供 resolution 來協助使用者排除問題。當然，有時後這些 resolution 並不一定合用，需要使用者自己動手修改。

說真的，對於使用者而言，如果一個軟體系統有提供這種 quick fix 的貼心功能，真的是很方便。不過，要設計出這樣的功能是需要多耗費一點精神滴，對於各位加班加到暴肝的鄉民們而言，沒事應該不太想嘗試吧（Teddy 內心獨白：要向上提昇啊...）。




- **Change the attitude toward errors. Think of an object's user as attempting to do a task, getting there by imperfect approximations. Don't think of the users as making errors; think of the actions as approximations of what is desired**.

最後這一點用書上的一段話來解釋並做本篇結尾 (p. 131)：

*When someone makes an error, there usually is good reason for it. If it was a mistake, the information available was probably incomplete or misleading. The decision was probably sensible at the time. If it was a slip, it was probably due to poor design or distraction. Errors are usually understandable and logical, once you think through their causes.* ***Don't punish the person for making errors. Don't take offense. But most of***

***all, don't ignore it.*** *Try to design the system to allow for errors. Realize that normal behavior isn't always accurate. Design so that errors are easy to discover and corrections are possible.*

以上四點以及上面這一段文字念個幾十次對於 designing for error 就會慢慢有 fu 了。

***

友藏內心獨白：好久沒翻這本書了，都快忘了這本書寫得有多好。

# 68 Designing for Error (3)：knowledge in the world and in the head

March 01 21:16~22:36

12/22 09:48-09:55

原文發表於
http://teddy-chen-tw.blogspot.com/2011/03/designing-for-error-3knowledge-in-world.html.

今天談一下 The Psychology of Everyday Things （p. 140）對於 Designing for Error 的總結，一共有三個重點，分三次介紹。

第一個重點：

***Put the required knowledge in the world. Don't require all the knowledge to be in the head. Yet do allow for more efficient operation when the user has learned the operations, has gotten the knowledge in the head.***

相信鄉民們都有到餐館點餐的經驗，不管是中文或是英文的菜單，

有時候光從「菜名」還真不知道這道菜的「組成份子」是什麼東東。什麼「貓耳朵」,「螞蟻上 樹」,「青龍皮皮挫」,沒吃過的人還以為這菜裡面真的有「貓」「螞蟻」和「青龍」。但是，如果菜單上面有照片，再加上適當的文字說明那麼就比較容易了解。 所以這本書建議在設計使用者介面（任何物品）要 **put the required knowledge in the world**， 以上面這個例子，菜單的照片，文字是了解這些奇怪菜名的 required knowledge，因此要將這些資料放在菜單上面（the world, 你要操作的那件物品）。如果在待操作的物品上（菜單）缺少這些資訊，那麼除非客戶的腦袋中（the head）已經有了這些點菜的必要知識，否則點錯菜是必然的。

當然，對於經常上館子的老饕，不需要看菜單也能點菜。因此把每道菜的照片和說明放在菜單上並不會限制老饕點菜的速度 （**do allow for more efficient operation when the user has learned the operations**）。

在這邊穿插一個真人真事，話說某人到美國出差，放假的時候和同事出去玩。到了用餐時間，到某間餐廳點餐，由於大家都看不懂菜單，於是某人想到一個安全的點 菜方法，就是不同種類的餐點都各點一份。當他點好菜之後，服務生面有菜菜色的和他再次確認。結果，最後上了四道菜... 全部都是「湯」。因為某人點了四道套餐所附的湯。

要怪誰？依照 *The Psychology of Everyday Things* 的思考邏輯，當然不能苛責某人啊，要怪就怪餐廳的菜單設計的太爛了，不 user friendly 的啦。

***

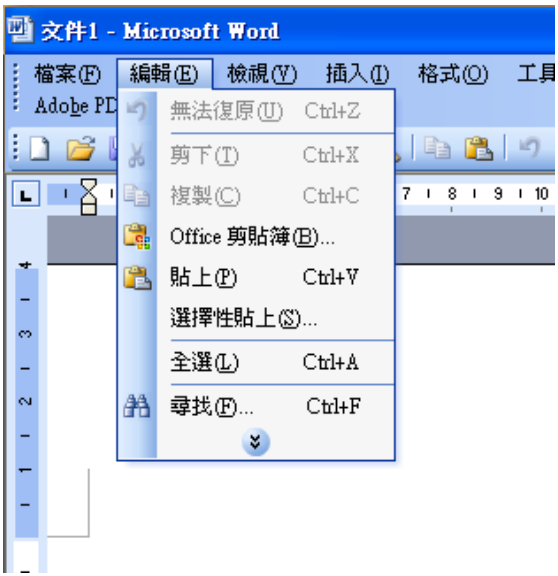
再舉一個軟體的例子，Microsoft Word 和 OpenOffice 都有一個「選擇性貼上」的功能，可以把，例如從網頁或是其他文章段落中所 copy 下來的文字以「未格式化文字（純文字）」的方式複製到 Word 或是 OpenOffice 中（這個功能 Teddy 經常使用）。為了做到「put the required knowledge in the world」，這些軟體允許使用者透過 Edit→Past Special...的方式來執行此功能。同時，為了「do allow for more efficient operation when the user has learned the operations」，OpenOffice 支援 Ctrl+Shift+V 這組熱鍵執行「選擇性貼上」的功能。很可惜的是，Teddy 在 Microsoft Word 中反而找不到這相對應的熱鍵，所以單就這個功能來看（先假設 Word 真的沒有），Word 並沒有做到「do allow for more efficient operation when the user has learned the operations」這一點。

Teddy 用「歪批 GOMS」所介紹的方法來分析一下在 Word 中要執行此功能所需要的步驟：

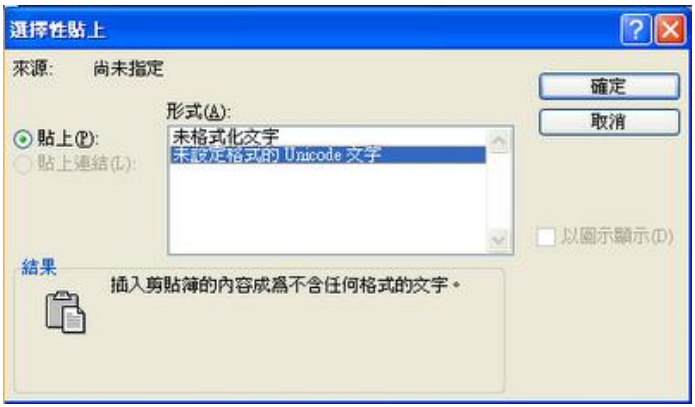
method 1：在 Word 中用滑鼠執行選擇性貼上

- 移動滑鼠到「編輯」menu 上
- 按下滑鼠左鍵
- 移動滑鼠到「選擇性貼上」這個 item
- 放開滑鼠左鍵 (此時 Word 彈出一個選擇性貼上對話盒)
- 移動滑鼠到選擇性貼上對話盒上的「未格式化文字」上
- 按下滑鼠左鍵
- 放開滑鼠左鍵
- 移動滑鼠到確定按鈕上
- 按下滑鼠左鍵
- 放開滑鼠左鍵

需要 10 個 operators




Word 主畫面



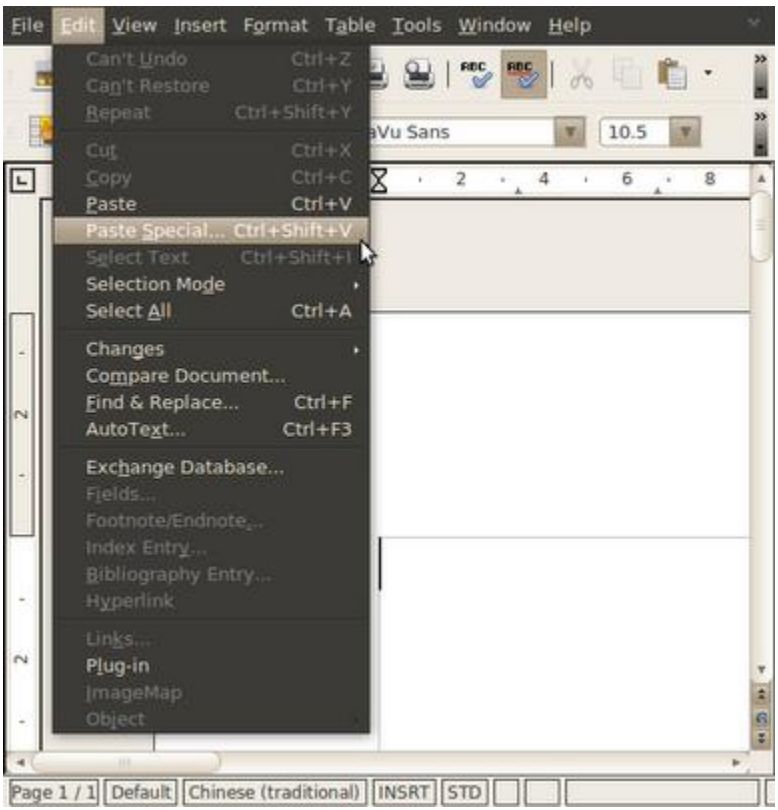

選擇性貼上對話盒（Word 畫面）

如果在 OpenOffice 上用熱鍵就簡單多了。

method 2：在 OpenOffice 中用熱鍵執行選擇性貼上

- 按下 Ctrl+Shift+V
- 放開 Ctrl+Shift+V

只需要 2 個 operators，就算是把 Ctrl, Shift, V 當成 3 個 operators，按下與放開這三個鍵也只需要 6 個 operators，比在 Word 中用滑鼠還少 4 個 operators。**當你在編寫文章而需要經常使用「選擇性貼上」這個功能的時候，你就會發現「do allow for more efficient operation when the user has learned the operations」這一點的重要性了。**




OpenOffice 主畫面

***

友藏內心獨白：這麼一來，是不是說如果介面設計的好，就算是「使用者的腦袋裝大...便當一個只要 50 塊...」也沒有關係嘍？！

# 69 Designing for Error (4)：constraints, forcing functions, and natural mappings

March 02 22:27~23:49

12/22 09:56-10:03

原文發表於

http://teddy-chen-tw.blogspot.com/2011/03/designing-for-error-4constraints.html.

The Psychology of Everyday Things（p. 140）對於 Designing for Error 的總結，一共有三個重點，今天談第二個重點：

*Use the power of nature and artificial **constraints**: physical, logical, semantic, and cultural. Use **forcing functions** and natural **mappings**.*

利用強加「限制（constraints）」的方式來避免使用者犯錯是一個常見的作法。什麼叫做強加限制？最簡單的一個例子，「時速限制」。因為怕開車開 太快容易發生車禍（等於產生 errors），因此政府對不同的路段設計不同的時速限制，這就是 constraints。Constraints 聽起來

雖然感覺是一個「負面」的字，因為它限制了使用者「任意使用」某物品的自由，但是往好的方面想，它也減少了使用者犯錯的可能。

路人甲：但是還是很多人超速啊！？

Teddy：自然有法律會去制裁他。

不知道鄉民們去 ATM 領錢的時後有沒有注意到，當選完要領的金額數目之後，ATM 會問你「是否要繼續交易」，如果選否，則 ATM 會把提款卡會先退出來。除非你把提款卡拿走，否則 ATM 是不會把錢吐出來的。**相信大部分的人不會忘記要拿錢** ~~只會忘記拿帳單~~，因此這種順序上的限制（先拿卡再拿錢）可以避免使用者忘記拿走提款卡的機率。想一想，如果反過來，先拿錢再拿卡，應該會有很多人一拿到錢太高興了就把提款卡給忘了。

***

看到這裡鄉民們應該會問，什麼是 forcing functions？依據書上的解釋（p. 132）：

*Forcing functions are a form of physical constraint: situations in which the actions are constrained so that failure at one stage prevents the next*

*step from happening.*

書上介紹有三種 forcing functions （p. 135）：

- **Interlock**：An interlock forces operations to take place in proper sequence。舉例說明之，如果使用者在微波爐啟動的時候不小心把微波爐的門打開，那麼微波外洩可是很危險滴，據江湖傳言輕則導致不孕（開爐自宮）或 是變成盲劍客，重則有生命危險。因此，微波爐就被設計成**「只要門一打開就停止運轉」**，或是說**「只有在門關起來的時候才可以運轉」**。回頭再讀一次 interlock 的定義：**一個 interlock 強制某些操作只能在特定的順序之下才可發生。** 「先關門-->後啟動」就是一種「特定順序」，如果違反此順序（限制）則該操作（啟動微波爐）就沒有作用。鄉民們使用微波爐的時候不知道有沒有想到 如果沒有這個 interlock 可是「祝恐怖」滴...搞不好微波爐就變成：「好微波爐！微波爐的奧妙之處，在於它可以藏在民居之中，隨手可得，還可以假裝做菜或是加熱食物來隱藏殺機， 就算被警察抓了也告不了你，真不愧為七大武器之首！」（Teddy 內心獨白：萬一停電不就沒搞頭了...XD）
- **Lockin**：A lockin keeps an operation active, preventing someone from prematurely stopping it。想像一下你正在使用 Microsoft Word，文件編輯到一半突然不小心「手ㄎㄟˊ」 去按到「關閉」功能。萬一 Word 真的傻傻的就直接離開，不帶走一片

雲彩，那麼你剛剛所打的那一堆資料就真的是「悄悄的來，又悄悄的走」。還好文書編輯軟體都針對「如果有尚未存檔的文件， 在結束程式之前必須要讓使用者再次確認」的機制，也就是 lockin 所要達到的 keeps an operation active, preventing someone from prematurely stopping it（讓 Word 繼續活著，除非使用者確定不儲存未存檔的文件）。

- **Lockout**：A lockout device is one that prevents someone from entering a place that is dangerous, or prevents an event from occurring. 鄉民們應該有看過類似「惡靈古堡」這類的電影，就是某個研究病毒的機構，有一天突然不小心病毒外洩，此時電腦自動「關門放狗」（放下安全門），不讓任何人 進入受到污染的區域（prevents someone from entering a place that is dangerous）。當然，萬一不幸你是那一個處在污染區域的人，那你就被「lockin」關禁閉直到變成 ~~薑絲~~ 僵屍才出得來...XD

***

最後談談 natural mappings，舉一個最簡單的例子，假設鄉民們家裡有五盞電燈，位置排列如下。

電燈 1　電燈 2　電燈 3　電燈 4　電燈 5

如果電燈開關排列的位置和電燈的位置一樣（如下所示），那麼開關電燈的時候就不容易搞錯。為什麼，因為控制端與被控端有著 **natural mapping**。

開關 1　開關 2　開關 3　開關 4　開關 5

但是，如果開關排列的位置長成這樣子：

開關 3　開關 5　開關 1　開關 2　開關 4

那麼每次開電燈都要搞得像是「樂透開獎一樣」，能猜中還真不容易。

***

友藏內心獨白：電影裡面還真的有用微波爐來殺人的情節，難道是破壞了 interlock？

# 70 Designing for Error (5)：execution and evaluation

March 03 21:39~10:52

12/22 10:05-10:12

原文發表於

http://teddy-chen-tw.blogspot.com/2011/03/designing-for-error-5execution-and.html.

終於到了 Designing for Error 這個系列最後一篇（希望是），不知道有沒有鄉民已經等在 ~~電視機~~ 電腦前面等著收看這一集的實況轉播，每天寫也是會累滴。

Teddy 記得在「美味關係」這部電影裡面，女主角之一的「茱莉鮑爾」立志用一年的時間，每天都要按照「掌握法國烹飪藝術」這本食譜上的菜單親自做菜，並把做菜的心得發表在自己的部落格上。等 Teddy 自己開始認真寫「搞笑談軟工」之後才發現，這怎麼可能，每天要「擠料」出來可是很傷腦細胞的一件事。就算是腦細胞撐的下去，每天打那麼多字，手腕，手肘和肩膀也會受不了（路人甲：Teddy 你好弱喔...）。

言歸正傳，今天要談的是 The Psychology of Everyday Things （p.

140）對於 Designing for Error 的總結，一共有三個重點，今天談第三個重點：

*Narrow the gulfs of **execution** and **evaluation**. Make things visible, both for execution and evaluation. **On the execution side, make the options readily available. On the evaluation side, make the results of each action apparent**. Make it possible to determine the system state readily, easily, and accurately, and in a form consistent with the person's goals, intentions, and expectations.*

這一點應該是很容易理解的（雖然並不見得容易做到），舉的例子先。假設你想要趕流行學人家「霸凌同 學 事」（PS：叔叔有練過，小朋友不可以學喔），於是有一天上班你帶了一根棍子到公司，遠遠看到被霸凌對象迎面而來，此時你高高舉起手上的棍子往對方頭上猛 K 下去。以上動作叫做 **execution**，你事先擬定了一個霸凌同事的「目標 （goal）」，為了達到此目標，你選擇了用「棍子猛 K 對方」的個作法（ method ），當看到對方之後，你採行一連串的行動（operators）以便達到此目標。啊，一不小心又 GOMS 上身了...

K 完一棒之後就沒事了嗎？當然不是，因為在採取行動（execution）之後你還必須要評估一下行動是否成功（ evaluation ）。按照常理推斷，對方被你 K 了一棒之後，應該要一邊用手摸著被 K 的地方，同

時發出一聲慘叫外加大聲重複背誦三字經。如果這些事件都發生了，你就可以確定此次的霸凌成功，否則就是失敗。

***

On the execution side, make the options readily available. On the evaluation side, make the results of each action apparent 是避免使用者操作錯誤相當有用的方法。翻成白話文就是，在設計介面時，對於使用者可以操作的功能有哪些選項要清楚的表達出來。例如，如果一個檯燈用的是 「省電燈泡」，只有「開」和「關」的功能（不能調亮度），那麼這個檯燈的開關就應該設計成只能「開」和「關」。

**採用 on/off 開關**

| execution side（開關） ： | evaluation side（電燈）: |
|---|---|
| on | 亮 |
| off | 暗 |

如果開關設計成可以調整亮度（鎢絲燈泡）的那種旋鈕開關，那麼就會很奇怪了。為什麼奇怪？因為：

**採用旋鈕開關開關**

| execution side（開關）： | evaluation side（電燈）: |
| --- | --- |
| 往右轉到最底 | 亮 |
| 往左轉到最底 | 暗 |
| 非上述兩個狀態 | 沒反應 |

就沒有達到 make the results of each action apparent 這一點要求，**也就是說使用者明啊明就有做了某些動作，但是卻沒有觀察到任何反應**。 以剛剛霸凌的例子來講，原本 K 了一棒就要快閃，受害者痛一下也就沒事了，但是萬一好死不死這個受害者硬撐不喊出聲音來，你可能以為沒 K 到或是 K 的太輕，於是卯起來連續 K 了 100 下，一直到受害者真的不可能再有任何反應為止....這是很恐怖滴...所以....伊...想哭不要硬撐。

***

Execution and evaluation 的例子很多，平常經常看到的 progress bar 就是讓使用者知道他剛剛執行的動作（execution）的狀態（evaluation）。或是當使用者按下「確定交易」的按鈕之後，系統回覆「交易成 功」也算。想一想，如果一個系統只有 execution（提供使用者操作否個功能的介面）而沒有 evaluation，這樣的系統是很難被正確使用的。

Teddy 最近就遇到一個慘痛的經驗，Teddy 公司和家裡的電腦都安裝 Ubuntu 作業系統，因為工作的需要經常會用到 Skype 來 ~~聊天~~ 談公事和傳輸檔案。還好 Skype 有提供 Ubuntu 版本，雖然有一些小問題，不過大致還堪用。

有用過 Skepe 的人都知道當你傳輸檔案給對方的時候，Skype 會跳出一個「檔案傳輸對話盒」（如下圖所示），讓你知道目前傳輸檔案的進度。不知道為什麼，在 Teddy 的電腦上有時候這個「檔案傳輸對話盒」就是不會出現。明明對方都已經接收到檔案了，Teddy 還以為檔案沒有傳出去（因為沒有得到任何 feedback 所以無法 evaluate 剛剛的檔案傳輸動作是否有正確執行）。有一次同事就問 Teddy，同一的檔案幹麼連續傳 10 幾次給他，此時 Teddy 才發現這個問題。

***

節目到這邊已經接近尾聲，鄉民們應該會發現其實這些 Designing for Error 的方法都很簡單，看起來也沒什麼學問。沒錯，這本書好就好在這裡。因為大部分都看的懂，所以看完之後有「自我感覺非常良好」。但是別忘了「知易行難」的道理，意思懂了不代表應用到實際設計上的時候都還記得這些道理。好里加在好心的 Teddy 把這些重點幫鄉民們記錄下來（鄉民甲：有重點嗎？！），上班上到腦袋空空的時候來逛一下，久而久之就不會忘記了。

***

友藏內心獨白：K 人的例子會不會太暴力一點...

# 71 HCI 之博士熱愛的算式

March 06 10:57~13:15

12/22 10:22-

原文發表於 http://teddy-chen-tw.blogspot.com/search/label/HCI.

幾個禮拜前在 Kay 的推薦之下 Teddy 讀了「博士熱愛的算式」這本書，內容敘述某位數學博士在 1975 年發生的車禍中傷了腦部，他保有車禍發生前的記憶，但在那之後他的記憶只剩下 80 分鐘，時間一到自動 reset。

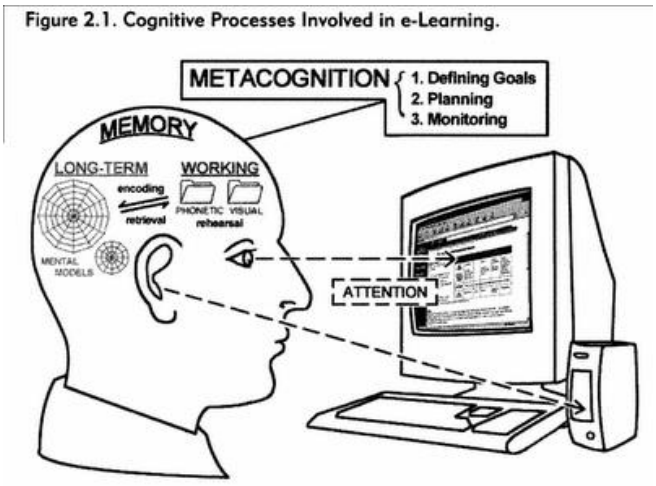
由於博士「生活無法自理」，~~所以不用入監服刑~~，發生車禍之後博士就由他的大嫂（博士的哥哥已經過世）照料。大嫂幫博士 請了一位管家照顧他，但是由於博士的記憶只剩下 80 分鐘，所以這位管家對博士而言每隔 80 分鐘都是一位「新人」。博士每天都穿著一件舊的西裝，他在西裝上身上夾滿了「小紙片」，用以彌補他的記憶只有 80 分鐘的缺陷。在眾多的小紙片中，其中有一張畫著管家的素描並註明她是管家這一點事實。

在電影中博士身上只是隨便敷衍的夾了幾張紙，關於這一點 Teddy 頗有意見，因為從小說的敘述中覺的西裝上應該是「夾滿紙片」才對。此外，電影中的博士看起來也太年輕了一點（小說中的博士應

該是有點駝背之類的)，而管家也長得太可愛了吧，感覺有點像女僕而不是管家...XD。

有人說這本書是「數學科普小說」(這是什麼東東？)，Teddy 數學極爛所以並不想要介紹這本書和數學有關的內容，因此關於本書的背景說明到此就夠了。 今天的重點還是要延續「Designing for Error (3)：knowledge in the world and in the head 」這個主題，談談 knowledge in the world and in the head 。

***




上面這張圖節錄自 *e-Learning and the Science of Instruction* 這本書第

35 頁，原本是在解釋地球人如何學習知識的心路歷程，在這邊借來說明人機互動的設計原則。概念其實很簡單：

- 人的記憶體區分為「long-term memory」和「working memory」。Long-term memory 類似電腦的硬碟（永久記憶體，關機之後不會消失）容量大但是存取速度慢，而 working memory 類似 RAM（揮發性記憶體，停電或關機之後資料會消失）容量小但是存取速度快。
- Working memory 的容量是有限制的（請不要說你家的電腦有 512 GB 的 RAM，就算是 512 GB RAM 和硬碟相比也還是「有限」）。一般常聽到的「7 +- 2 原則」（人的短期記憶可以記住 7 +- 2 個東西）就是指 working memory 的大小。
- 視覺和聽覺的資料經過眼睛和耳朵接收之後進入 working memory。
- Working memory 的資料經過處理之後儲存到 long-term memory ，變成人的 mental models 的一部分。

知道這幾點就差不多了，現在回頭看看一個 HCI 設計原則：**不要讓我花腦筋**。

假設使用者在執行「員工個人資料維護系統」，在該系統的畫面上有兩個按鈕，分別是「存檔」與「列印」這兩個功能。鄉民們是設計這個系統的 programmer，當初你為了「code reuse」，把所有執行成功的訊息寫成一個公用對話盒，所以當使用者執行「存檔」或是「列

印」功能的時候，畫面都會出現相同的「執行成功」這個訊息。
問題來了，畫面上有「存檔」與「列印」這兩個功能，但是不管執行那一個功能所顯示的訊息都一樣，都是「執行成功」，那萬一使用者**恍神**要存檔卻不小心按成列印，而程式卻又告訴使用者「執行成功」，那怎麼辦？

切，鄉民們會說，這有什麼好大驚小怪的，又不是飛航系統，按錯又不會死人。但是想一下：

- 使用者以為存檔完成但是卻是把資料印到「公用印表機」而不自知，所以使用者也不會去印表機前拿資料。如果這些員工個人資料有包含薪水或是退休金提撥（可以反推薪水）等資料被其他人拿走，那就不太好了。
- 使用者以為資料已經存檔完成，此時使用者電腦突然當機。下次使用者再進入「員工個人資料維護系統」之後，發現自己的資料怎麼還是舊的？會以為軟體功能有問題或是被駭客入侵而回報一個 bug，結果查了老半天才發現是自己的 ~~腦袋有問題~~ 操作疏失。

所以，如果有好好應用 put the required knowledge in the world 這個原則，那麼這個訊息就要改成「存檔成功」和「列印成功」。**不要把「請自己記住你剛剛執行了什麼功能，按了什麼按鈕」的責任丟到使用者的身上**（Don't require all the knowledge to be in the head.）。

因為 working memory 容量有限，所以如果沒有善用 put the required knowledge in the world 這個原則所設計出來的介面會增加使用者小腦袋的負擔。更慘的是，有些使用者介面操作的習慣和人們一般的認知不同或是相反，這種介面還會增加使用者的 **mental loading**，每次操作這種「與自己認知不同」的功能時都要「花腦筋」想一下。大部份的人應該都同意「花腦筋」是一件很累人的事情，所以看「夜市人生」與「海綿寶寶」的觀眾群才會遠大於看「Discovery」的人...XD。總之，需要讓使用者經常「花腦筋」的介面用久了會有種「身心具疲」的感覺，而且容易出錯。舉個 例子，在台灣開車駕駛座在左邊，對駕駛而言這已經是一種「習慣」，但是如果要右駕的國家例如日本或是英國開車，雖然同樣都是開車，所需要技能基本上是一樣 的，但是剛開始開車的時候通常會很緊張，時時提醒自己轉彎的時候方向不要搞錯（增加 mental loading），使得原本很輕鬆的開車之旅變成很「用力」的開車之旅。

再舉個例子，Teddy 平常工作的時候用到兩台電腦，透過 KVM 來共用一組鍵盤、螢幕、和滑鼠。這個 KVM 可以透過 Scroll Lock + Scroll Lock + [1, 2, 3, 4] 來切換到不同的電腦（這是一個 4 ports 的 KVM），但是很奇怪的一點，有時候（不是每一次）Teddy 按下 Alt + Tab 的時候，KVM 也會切換到不同的電腦。由於 Teddy 老早就習慣用 Alt + Tab 來切換不同的應用程式（Windows 和 Linux 上都是

用這組熱鍵），但是這個該死的 KVM 不知為什麼有時候會把 Alt + Tab 當成是要切換到不同的電腦的信號，Teddy 讀了手冊也看不到 KVM 有提到這一點，也不知道這是 KVM 的 bug 還是功能。總之每次當 Teddy 反射性的要按下 Alt + Tab 切換應用程式的時候，就被迫要「動一下腦筋」提醒自己改用其他方式切換應用程式。有時候忘了還是使用 Alt + Tab，就要再手動切換回剛剛正在操作的那台電腦，次數發生多了還真是挺不爽的。

***

回到博士熱愛的算式，出車禍之後博士已經失去增加 long-term memory 資料的能力，而且他的 working memory（short-term memory）每隔 80 分鐘就會消失（每隔 80 分鐘重新開機），怎麼辦？沒有辦法 put the required knowledge in the head 那只好 put the required knowledge in the world（寫在小紙條上），日子還是過的下去。

***

友藏內心獨白：要買到一台好用的 KVM 還真難，不是無線滑鼠不能用就是切換熱鍵有問題。

# 第八部 測試與整合

# 72 有 test cases 改遍天下，無 test cases 寸步難行

01/23 02:24~03:28

12/18 22:18-22:22

原文發表於

http://teddy-chen-tw.blogspot.com/2010/01/test-cases-test-cases.html.

如果要列舉 agile methods 對於軟體開發行為模式的影響，除了 iterative and incremental (IID)之外，最重要且影響最深遠的應該首推自動化測試了（感謝 JUnit）。幾個很重要的 agile practices，包含 refactoring 和 continuous integration，如果沒有自動化測試就「破功了」。

Teddy 本身對於自動化測試的進化路徑如下：

1. 沒寫過（不會寫也不願意寫）自動化測試。有沒有搞錯，程式都寫不完了，哪有時間寫測試。
2. 開始用 JUnit 寫單元測試。
3. 用 JUnit 寫很多單元測試。
4. 用 JUnit 寫一些整合測試。
5. 用 JUnit 寫一些功能測試。

6. 嘗試在 JUnit 中用 mock object framework。
7. 很少在 JUnit 中用 mock object framework。不是不好，而是不合 Teddy 的胃口。
8. 用 JUnit 寫很多功能測試。
9. 當發現 bug 時，想辦法用寫一個測試來還原這個 bug（test-driven debugging）。
10. 三不五時想到的時候，在開發新功能時，先寫 test code 再寫 production code (玩票性質的 test-driven development)。

最近 Teddy 與同事修改專案中某個模組，增加新的功能也同時重整 (refactor) 該模組許多類別的介面。還好該模組有相當多的測試案例，所以這麼大幅度的修改，「只」產生大概 10 個左右無法立即發現的 bugs。這些 bugs 都是現有測試案例沒有涵蓋到的（廢話...），所以當 Teddy 與同事一起 pair debugging 時，我們第一步就是寫一個測試案例來暴露出這個 bug。講白話文就是寫一個一定會失敗的測試案例，當這個 bug 修正之後，這個測試案例就會成功。

這次 refactoring 主要反應了某些軟體架構上的改變，算是所謂的 big refactoring。雖然前前後後一共花了三～四天才把全部的 bugs 改完（包含寫新的測試案例），整個過程還算順利。如果沒有這些測試案例，修改的過程應該早就失控了。這種「越補越大洞」的經驗，對於廣大的鄉民們而言應該 不陌生吧。

很多有心想要導入 agile methods 的人會說：「要 programmers 寫測試案例好難」，「不知道要如何開始作 test-driven development」。Teddy 的經驗是，雖然 test-driven development 是一種透過寫測試案例來達到設計程式的目的（就是 TDD 應該算 design 而不是 testing），但是除非 programmers 已經將寫測試案例當作開發系統不可分割的一部分，否則並不容易推行 TDD。如果 programmers 寫測試案例就跟「喝開水」一樣簡單與自然，那麼一不小心他們就會慢慢演化成 TDD 動物。

結論就是：不能沒有你，test cases。

***

友藏內心獨白：這麼晚了還不睡，寫什麼部落格，小心明天被罵！

# 73 忙到爆的五月天

May 12 22:02~23:00

12/14 10:29-10:56

原文發表於 http://teddy-chen-tw.blogspot.com/2011/05/blog-post.html.

快二十天沒寫部落格了，俗話說的好：計畫趕不上變化，這二十幾天也不知從哪裡冒出來那麼一堆多如牛毛的小事，搞到沒腦力寫部落格。剛剛 21:00 整點的時候原本要去植物園散步，但是突然下起大雨，有點時間那就上網隨便唬爛一下，用以感謝老天爺天降甘霖，順便灌溉一下快枯萎的「搞笑談軟工園地」，以免荒廢太久僅存的「一小撮」小粉絲都跑光光了（別轉台啊...跪求...）。

到底有什麼好忙的？

- 某人為了測試需要產生一筆 uncorrectable ECC error 的 log，此人居然在電腦開機的狀況下，直接把整條記憶體拔起來（這可不是蘿蔔啊...XD）。更糟的是某人在遙遠的他鄉，靠 e-mail「凸」了好久才搞清楚原來某人是這樣搞的。恕 Teddy 孤陋寡聞，**不知道記憶體可以「熱插拔」**？！好前衛的測試手法啊，佩服，佩服。

- 另一位遠在天邊的工程師，需要請他修改某個程式，同樣「凸」了好久... 還沒搞定。這位老兄的**「太極拳」**著實打的不錯，算是頂尖武林高手，Teddy 自嘆不如，甘拜下風…Orz。
- 還有很多喜歡「挑毛病」的朋友們，秉持著「看到黑影就開槍」的精神，寧可錯殺一百，不可錯放一個。怎麼說呢，這種感覺有點像是傳說中的警察杯杯為了業績而亂開罰單的那種行為，**所以說，QA 部門不可以用「回報 bugs 數目多寡來算業績啊」，這種「亂開罰單」的行為是很不可取滴。**
- 自認自己很行的另一位老兄，三不無時打電話來鬧一下，見面也鬧一下，連寫 e-mail 也要鬧。光是應付這「三鬧」就不知道浪費 Teddy 多少時間。
- 還有一件秘密行動，日後另行公佈。

***

這幾天有一則新聞，就是宏碁前執行長蘭奇接受專訪表示：**「要在台灣落實軟硬體整合，是不可能的；得走出台灣，無論是中國、印度，甚至美國或歐洲，任何能找到軟體資源、軟體專業知識的所在。」**

Teddy 是還滿認同的，「理論上」台灣應該是有能力可以開發出好軟體，但是「把軟體做好」只是整個「軟硬體整合」value chain 的一環而。電路設計的好、BIOS 寫得好、firmware、deriver 寫得好、AP 寫得好、測試做的好、文件寫得好、組裝組的好、供應商選的好、

品質控管的好、時程控管的好、開發成本合理、最後加上開發展品流程中的每一個關卡都「銜接的好」，這樣才能夠做好「軟硬體整合」，否則變成 Teddy 經常講的**分工...但不合作**，那就沒搞頭了，加班吧你。**「整合」這件事情是很難搞滴，要打通那麼多關卡，每一關都有那麼多大老爺，大小姐與皇親國戚，三朝元老要去擺平。你看我不順眼，我看你不爽，整什麼合，整死自己先。**

Teddy 有個朋友，平時看起來有點傻傻的，每天無所事事的樣子，到處打聽大小消息，私底下肩負著維持「兩岸（太平洋兩岸）」和平的神聖使命。有一天他老闆要他做某件事，這件事牽扯到其他部門的人，事情進行的很不順利，最後不了了之。有一次機會 Teddy 跟這位朋友一起吃飯的時候朋友談到這件事情， ~~他的狗嘴居然吐出子一根象牙~~ ...嗯嗯...他突然講了一句很有哲理的話，差點讓 Teddy 把手中的平板電腦摔到地上：

**有些事情不是能力的問題，而是整個流程都卡住了。**

Teddy 默默地在心中幫這位朋友按了十個讚。是啊，有些事不見得是個人能力的問題，就算是很有能力的人（例如 Teddy...XD），遇到眾多「太極拳高手」或是「乒」，就算沒被打死，也只能陪他們「慢慢練太極」啦。

來人啊，哪裡有衝鋒槍？

***

友藏內心獨白：朝「呆伯特」境界又邁進了一大步，應該高興嗎？

# 74 Ten-Minute Build

March 10 21:06~22:45

12/11 11:57-12:13

原文發表於 http://teddy-chen-tw.blogspot.com/2011/03/ten-minute-build.html.

和 Teddy 差不多年紀的鄉民們應該還記得當年 SOS（大、小 S）紅遍大江南北的「十分鐘的戀愛 」這首歌：

（下課）十分鐘的戀愛　雖然有一點短暫
你的笑填滿我 心中所有的遺憾
（下課）十分鐘的感情　我將全部屬於你
多希望能夠永遠不分離

可惜做軟體的人沒那麼好命，平平是 10 分鐘，人家可以拿來談戀愛，咱們只能拿來建構（build）軟體...XD

***

話說當年 Teddy 在讀 *Extreme Programming Explained: Embrace Change* 這本書的時後，看到第 49 頁寫著某一個 XP 的 primary practices：

**Ten-Minute Build**

*Automatically **build the whole system** and **run all of the tests in ten minutes**. A build that takes longer than ten minutes will be used much less often, missing the opportunity for feedback. A shorter build doesn't give you time to drink your coffee.*

當時 Teddy 看到這段話其實沒什麼特別的感覺。喔，10 分鐘將整個軟體建構完畢並跑完全部的測試...了解，收到....轉身後繼續過自己的日子....

好幾個月前聽指朋友談起某開發「~~倒航~~ 導航軟體」的公司，因為建構一次軟體要花很久（幾個小時吧）的時間，而且全公司就只有「一個人」知道要如何建構這個軟體，所以經常會發生程式設計師應觀眾要求「私底下」先把軟體拿給客戶使用（這個客戶是公司內部的員工，需要用這個軟體來建地圖景點資料）。但是曾經發生過員工已經用「私底下版本」的軟體花了幾百個「人/時」來建地圖資料，但是就在「那個人」心血來潮去建構一下軟體的時候，發現建構失敗，軟體需要修改，因此導致之前所建立的地圖料要全部重新輸入的情形。

由於這是發生在別人身上的慘劇，聽的人哈哈一笑，總覺的「哪有

那麼扯的事」也就沒放在心上，一直到最近 Teddy 經常需要建構某個軟體，才慢慢體會到 Ten-Minute Build 這件事情的重要。Teddy 所要建構這個軟體，全部重新建構一次需要 15 分鐘左右。你可能會想「15 分鐘和 10 分鐘差不多啊，已經很不錯了」。錯，因為這 15 分鐘只是去建構這個軟體並產生給 Windows 與 Linux 平台使用的安裝程式（installer），並沒有跑「test cases」。要是真的去跑 test cases（用 JUnit 寫的 unit tests），可能會超過 2 小時以上。

其實這裡面有很多可以「改善」的地方：

- **花錢**：沒錯，最快的方法就是花錢升級 build server。Teddy 目前使用的 build server 是快四年前的機器了，CPU 是 Pentium 4 2.8 GHz，DDR2 1 G RAM 跑某個版本的 Linux OS。今天把 1G RAM 換成 2G 速度馬上提昇..... 30 秒.....有點少...沒關係，還好公司還有一些「私藏軍火」，找到一台空機，準備了兩顆 Xeon 5500 系列的 CPU，申請四顆 2.0 T SATA 硬碟準備做 RAID 10，還有申請 6 條 DDR3 4G RAM...不過東西都還沒來....等裝好之後再看看速度提昇多少。
- **調整 Build Script**：把 build script 寫得聰明一點，例如如果某個 sub-project 以及它所相依的其他 sub-project 如果都沒有異動就不需要重新 build，或是只測試有異動的程式碼。
- **調整要執行的 test cases**：雖然大家都說「用 JUnit 寫自動化單

元測試」，但是嚴格講起來大部分的人所寫的「自動化單元測試」其實都不是「單元測試」，而是大小不一的「整合測試」。因為真正的單元測試要達到「test in isolation 」，這樣才不會因為「別人帶賽」而導致 test 失敗；而且這樣的單元測試也才能跑得快。所以，在每次建構的時候，可以先把那些「真正的單元測試」挑出來跑，看看能不能把整個時間控制在 10 分鐘。

- **電腦上晚班**：在人腦下班之後，設定時間自動將整個軟體全部建構一次並且將那些需要耗時執行的測試案例拿出來跑，這樣可以確定**至少每天都有完整 建構一次整個軟體且將全部的測試案例執行完畢**（勞基法沒有規定電腦的每月工作時數，所以就用力的操吧）。

再想想之前 Teddy 去上 Certified ScrumMaster 課程時授課講師 Bas 所說的話：

「很多專案都因為測試設備不足而導致開發速度無法提昇」，這是一個很簡單的道理，但是否每個老闆都有這樣的認知，願意投入資源在改善「建構」與「測試」的環境就很難講了。

***

友藏內心獨白：Teddy 沒有亂花錢，這是軟體流程改善的投資喔，趕

快審核通過採購申請單吧。

## 備註

1. 建構（build）
2. XP
3. primary practices
4. build server
5. test in isolation

## 參考資料

[1]

[2]

# 75 人客的要求：Ten-Minute Build 後續報導

March 29 20:33~21:18

12/11 16:25-16:30

原文發表於

http://teddy-chen-tw.blogspot.com/2011/03/ten-minute-build_29.html.

有鄉民留言說想知道「Ten-Minute Build」硬體升級之後的改善結果，今天來個極短篇說明一下後續狀況。

## 「美容前」的狀況

Teddy 原本使用的 build server 是快四年前的機器了，CPU 是 Pentium 4 2.8 GHz、DDR2 1 G RAM 跑 RHEL 5.x。每次建構軟體約需 15 分鐘，為了提昇建構速度，將 1G RAM 換成 2G，但是速度只提昇了 30 秒。

## 改善方案

公司什麼都沒有，就是電腦特多。尋找庫房內的「私藏軍火」，找到

一台 1U 的閒置 server，以下為主要配備：

- 主機：1U Server，免費。
- CPU：Intel Xeon 5580 3.20 GHz　* 2，人家送的，免費。
- RAM：DDR3 4G Reg. DIMM * 6 = 24 GB，約 9900 NTD。
- 硬碟：2TB 3.5 吋 SATAIII 硬碟 * 4 做 RAID 10， 約 20000 NTD。
- OS：CentOS 5.x，免費。

本次 ~~改造~~ 升級費用約 29900 NTD。

## 改善結果

- 當安裝 4 GB RAM 時，建構時間縮短成只需約 3 分鐘。
- 當記憶體加到 24 GB RAM 時，建構時間再減少 20 秒左右（3 分鐘- 20 秒等於 2 分 40 秒）。

## 結論

花不到三萬（不包含主機與 CPU 的費用）可以大幅提昇建構速度算是很划算的投資，可以減少很多開發等待的時間。對於家裡沒有閒置 Server 也沒有人送 CPU 的鄉民們，如果要幫 Builder Server 升級可

能就要準備個 15-20 萬左右。但是如果不要買到 Server 等級的電腦，也許 10 - 12 萬就可以搞定，再省一點 5 - 8 萬應該也 OK。

另外，Teddy 不確定使用雙 CPU 對於縮短建構時間是否有幫助，因為剛好有多的 CPU，所以就直接插上去。還有，因為建構需要使用到很多 I/O 處理（廢話，每個檔案都要編譯，還要產生 ja 檔還有安裝程式等等），**按常理推斷**如果改用 SSD 應該可以進一步加速建構時間。雖然後來採購 SATAIII 的硬碟，但是主機板只支援 SATA II，所以實際上硬碟還是只有 SATA II 3Gb 的速度。

最後強調一點，以上時間都不包含測試，所以如果以 Ten-Minute Build 為目標，趕善之後還有 7 分鐘左右可以拿來跑單元測試。報告完畢。

***

友藏內心獨白：好想知道如果改用 SSD 會加快多少....

# 76 Ten-Minute Build 後續報導 (2)

September 06 21:28~22:58

12/11 16:05-16:22

原文發表於

http://teddy-chen-tw.blogspot.com/2011/09/ten-minute-build-2.html.

自從六個月前寫了「Ten-Minute Build 」與「人客的要求：Ten-Minute Build 後續報導」之後，Teddy 可是沒有閒著，時時都在思考如何改善 CI (Continuous Integration）的效率。

將原本老舊的 Pentium 4 2.8 GHz，DDR2 1 G RAM 的 server 換成另一台有兩顆 Xeon 5580 3.2 GHz CPUs，四顆 2.0 T SATA 硬碟做 RAID 10，以及 6 條 DDR3 4G RAM（共 24 G）的新 server 之後，產生 installer 的時間由 15 分鐘縮短到 3 分鐘。後來也曾經嘗試將 CI 系統的 workspace 放到 RAM disk 裡面，不過測試結果整個建構時間並沒有顯著的縮短，因此關於硬體方面的改善暫時告一段落，接下來該是整頓一下 CI 系統：老舊的 CruiseControl。

不過，CruiseControl 介面雖然有點陽春，但也算是頗「盡忠職守」，俗話說得好「東西沒壞就不要修」，一想到要是那一天真的把 CruiseControl 換掉而出了什麼狀況，到時候可是問題很大。所以「換

掉 CruiseControl」這件事情想了好久卻一直都沒有付諸行動。

前一陣子團隊中來了一位新人，剛好對 CI 系統很熟，於是 Teddy 就放心的連續在幾個 sprints 中安排一些 tasks 作為換掉 CruiseControl 的準備。現在，新的 Jenkins 系統已經上線兩個 sprints 了，團隊成員都非常的滿意（除了一個不知名的 bug 還有待解決以外）。

***

先說說原本使用 CruiseControl 遇到的兩個問題，這兩個問題也是 Teddy 想要換掉 CruiseControl 的主要原因：

- 我們的軟體由很多個 projects 所組成，平常 CI 系統產生一個新的 installer 的時候，並不會從頭去 compile 所有的 projects，而是將個別 projects 已經 compile 並包裝好的 jar 檔以及一些有的沒的設定檔給它再次打包。這種作法的好處是可以快速的產生 installer；由於團隊經常（每天 N 次）會使用 installer 來做 end-to-end functional tests 與 usability tests，因此快速產生 installer 是很重要的。但是，可能是有些專案的 build scripts 沒寫好或是 project dependency 沒有處理好，或是其他神秘不可知的原因，有時候用這種方式包出來的 installer 會包到舊的 jar 檔。當這個問題發生的時候，就很討厭，需要手動去 build 某些專案，等新

的 jar 檔產生之後，再去 build 一次 installer。

- CruiseControl 被設定為採用 pooling 的方式每隔一段時間去 SVN 上面看看專案的內容有沒有改變，如果有就 build 該專案。Pooling 的作法雖然簡單，但是「時效性」很差。由於 projects 之間有 dependency 的關係，Teddy 經常懷疑採用 pooling 的方式就是導致第一個問題（有時候會包到舊的 jar 檔）的原因之一。

嚴格講起來，這兩個問題其實不算是 CruiseControl 的問題，真的花時間下去，還是可以解決。但是，後來新進同事介紹了 Jenkins 之後，團隊認為用 Jenkins 來解決這兩個問題變得比較簡單，而且加上 Jenkins 有許多方便好用的功能，因此毅然決然的展開換掉 CruiseControl 的活動。

***

整個轉換的過程大致如下：

- 研究與整理每一個 project 的 ant build script，把不需要用到的 jar 檔從 build script 拿掉。
- 從 build script 中整理出 project dependency diagram。這個圖很重要，解決這兩個問題就靠它了。
- 在 VM 裡面裝一套 Jenkins，把 CruiseControl 上面所有 projects

在 Jenkins 裡面也建一份。

- 用 VM 裡面的 Jenkins 來 build projects，一直到可以產出 installer 為止。
- 停掉 CruiseControl，把 Jenkins 安裝在實體機器用以取代 CruiseControl。

團隊用了兩個方法來解決上述的那兩個問題：

- Jenkins 的專案可以設定 upstream projects，利用這個功能就可以建立起每個專案之間的 dependency。有了專案的 dependency 之後，某個專案 build 完成之後，可以設定自動去 build 相依於該專案的其他專案，如此一來就可以避免 installer 包到舊的 jar 檔的問題。設定 upstream projects 還有一個很重要的好處，容後再稟。
- 利用 SVN hook 將原本使用 pooling 來驅動一次 build 的方式改成 event-driven。只要有人將檔案 commit 到 SVN 中，就會觸發 SVN hook（一隻 script 程式），然後這個 SVN hook 就會呼叫 Jenkins 去 build 相對應的專案。

Jenkins 支援 concurrent builds 的功能，同時間可以有多個 build jobs 一起執行。使用者可以設定 Jenkins 同時間可以執行 build jobs 的數目。由於我們的 build server 有兩顆 Xeon 5580 CPUs，每顆有四個

cores，8 個 threads，一共有 16 個 threads，因此我們讓 Jenkins 同時間可以執行 10 個 build jobs，如此一來又可以有效的縮短整個產生 installer 的時間（假設有好幾個專案同時被修改）。

但是，concurrent builds 有一個問題需要解決，就是 project dependency 的問題。如果 project dependency 沒有設定正確，那麼同時間去 builds 好幾個專案，輕則會發生一個專案被重複 build 好幾次的問題，重則可能會發生 installer 包到舊的 jar 檔的問題。剛剛提到 Jenkins 的專案可以設定 upstream projects，利用這個資訊，在 concurrent build 的時候，Jenkins 會避免專案被重複 build 的問題。經過實際使用的結果，覺的該機制滿穩定的。相較於之前使用過 CruiseControl 的 concurrent builds 功能，只能用「慘不忍睹」來形容。

此外，Jenkins 有許多貼心的小功能，例如：

- 在畫面上可以自訂 tab 將眾多的專案加以分類。
- 可以顯示目前正在 build 的 projects 的進度（Jenkins 會參考上次 build 的時間來預估本次 build 尚須多少時間）。
- Jenkins 的 distributed builds 功能非常簡單易用。
- 支援 Jenkins 的 plug-ins 超多，擴充很方便。

***

結論是，SVN hooks + concurrent builds + upstream projects 讓整個 build 流程變得十分順暢。例如，Teddy 才從 Eclipse 把 code commit 到 SVN 中，正準備打開 Jenkins 網頁的時候，發現剛剛 commit 的 code 已經 build 好了，而 installer 也正在 build 當中。整個流程順暢到有時候 Teddy 都會有點小小的懷疑：這麼快，projects 到底有沒有被 build 到啊？

***

友藏內心獨白：軟體不如新，人不如故。

## 備註

1. CruiseControl
2. pooling
3. concurrent builds
4. SVN
5. SVN hooks
6. upstream projects
7. distributed builds

8. Xeon
9. Eclipse

## 参考資料

[1]

[2]

# 77 落實的能力

April 21 20:46~21:32

12/14 10:59-11:08

原文發表於 http://teddy-chen-tw.blogspot.com/2011/04/blog-post_21.html.

幾年前 Teddy 還在唸書的時後，看了一篇 *Continuous Testing* 的論文，大意是說，在 Eclipse 台上，利用現在個人電腦都跑很快的特點，當使用者在 coding 的時候，Continuous Testing 系統（一個 Eclipse plug-in）就偷偷在背景模式幫 programmers 執行 unit tests，如此可以縮短 coding →run unit testing 這個開發週期的時間。當時 Teddy 和某個學弟還改良這個想法，在 Eclipse 上面也開發了一個類似的 plug-in，雖然研究的性質居多，在實際軟體開發上面沒有真正派上什麼用場，但是覺的滿有趣的。

有在看 Teddy 部落格的鄉民們應該知道最近 Teddy 在想辦法縮短持續整合與建構的時間（請參考「Ten-Minute Build 」與「人客的要求：Ten-Minute Build 後續報導 」）， 雖然經過硬體升級之後，把產生安裝軟體的時間從原本的 15 分鐘縮短到 3 分鐘，但是 Teddy 有講過這個過程是不包含執行單元測試的時間。做持續整合沒有跑單元測試其實是不對的，但是以 Teddy 所遇到「現況」，雖然程式設計師都有寫單元測試，**但是有些單元測試的執行時間還滿長的**。最理想的作

法當然是要想盡辦法用 mock objects 這一套方法來讓單元測試跑的很快，但撇開這個不談，假設鄉民就是有一堆跑起來不是很快的單元測試，那麼在實施持續整合的時候到底要不要跑這些單元測試？請 鄉民們先想一想...

***

Teddy 目前的作法，單元測試只有程式設計師在開發環境（Eclipse）上執行，在持續整合系統上是完全不跑的（路人甲：持續整合不跑單元測試，好大的狗膽！）。**但是另外以 Robot 撰寫自動化功能測試來彌補**（Teddy 內心獨白：看完這句再罵也不遲）。這些自動化功能測試每天晚上會在所有我們所開發的軟體所支援的平台上面執行一遍（這句有點繞口...XD），早上的時候有一個 ~~地獄倒楣鬼~~ 專門技術人員會去看看有那些 Robot test cases 沒有過，然後去找出問題並加以解決。

如果你喜歡的話，這些自動化功能測試已經被搞成一包「自動化功能測試包」，可以用綠色軟體的方式隨時佈署到各個不同的待測平台上面執行功能測試。

***

因為最近升級了持續整合機器，先幫鄉民們複習一下，硬體設備如下：

- 主機：1U Server。
- CPU：Intel Xeon 5580 3.20 GHz * 2。
- RAM：DDR3 4G Reg. DIMM * 6 = 24 GB。
- 硬碟：2TB 3.5 吋 SATAIII 硬碟 * 4 做 RAID 10。
- OS：CentOS 5.x。

這台電腦有兩顆 Intel Xeon 5580 3.20 GHz 的 CPU，每一顆 CPU 有 4 個 core，可以跑 8 個 threads，只用來執行單一的持續整合工作太浪費了。**因此 Teddy 就想到，是不是可以在持續整合系統上面幫「執行單元測試」這件事專門建一個專案，然後當有需要的時候就去跑一下單元測試，如此一來便可同時兼顧「快速建 構」與「執行單元測試」這兩件事。**

其實每一次建構一個專案，除了基本的 compile 與 testing 以外，還有很多事可以做。例如，分析 test case coverage（會有點花時間），做靜態程式碼分析等等，這些都是以前很想做但是「沒時間」做的工作。現在把「執行單元測試（廣義的來講，應該是執行完整的整合工作）」這件事獨立成一個專案，這樣就有機會可以很頻繁的執行。你問為什麼？因為電腦跑很快啊，所以可以在電腦上面裝兩套持續整合系統，一套跑現有的工作（編譯並 產生安裝程式），另外

一套跑「完整的持續整合」，這樣就有點類似把 Continuous Testing 的概念套用在 Continuous Integration 上啦。

這樣講不知道鄉民們有沒有看懂...啊，Teddy 要準備去倒垃圾了，改天再聊。

***

友藏內心獨白：知道持續整合和落實持續整合中間的 gap 還是滿大的。

# 78 落實的能力（2）

April 22 21:20~22:18

12/14 11:31-11:41

原文發表於 http://teddy-chen-tw.blogspot.com/2011/04/2_22.html.

昨天寫到一半跑去到垃圾，當 Teddy 把垃圾放在電線桿旁邊，人站在離垃圾約 5 步之遙的位置，正在等待垃圾車到來時，突然發現有某位婦人正在偷偷地亂搞 Teddy...的垃圾。當下 Teddy 也沒想那麼多，直覺反應立刻跑過去說了一聲「**這是我的垃圾耶** （切，垃圾還怕人家搶啊...XD）」。原來那位婦人手上拿著兩包小垃圾，但是她沒有使用 ~~舊北市~~ 台北市政府所規定的垃圾袋，所以她就到處找「落單的垃圾袋」，試圖把她自己的垃圾「寄生」在別人的垃圾袋之中。

被 Teddy 一喊，這位婦人面露不好意思的表情，就帶著她的那兩小包垃圾，另外尋找合適的寄生對象。事後 Teddy 想一想，會不會太嚴厲了一點，也許有些人真的有經濟上的困難，雖然一般人可能覺的一包專用垃圾袋沒多少錢，但是對於經濟較不寬裕的人，長期省下來也是一筆不小的錢。至少這位婦人也沒把垃圾亂丟，還願意把自己的垃圾寄生在別人的垃圾袋裡面，也算是還有守法的精神了。和那些把垃圾隨地亂丟，或是硬塞到路邊垃圾筒的人相比，這位婦人算是有公德心的了。以後如果再遇到這種「垃圾寄生事件」，應

該要抱持著寬容一點的心去看待才是（Teddy 跟自己開了一個 3 分鐘的 retrospective meeting）。

***

昨天談到在持續整合系統（以下簡稱 CIS）上面如何兼顧「縮短整合執行時間」與「執行多樣化整合任務」這兩件事，最後得到**「安裝兩套 CIS，一套用來跑編譯與產生安裝程式的工作，另一套用來執行整的整合任務」**這樣的結論。

說真的 Teddy 之前沒想過這個問題，記得有快四年前有一次 Teddy 聽到有人在 CIS 上面只編譯程式而沒有執行單元測試，當場表現出「CI 不是這樣做滴」的鄙視態度。對於對方回答**「測試案例執行時間太久了啊，所以為了縮短整合的時間，只能選擇不去執行單元測試」**這樣的理由，當時 Teddy 是完全無法接受。為什麼？因為「書上說」單元測試就是要「想辦法」跑的很快啊，如果寫出來的單元測試跑的很慢，那一定是不認真沒有寫出「正港的單元測試 啦」。因為單元測試跑太久在 CIS 上面就不執行，如此因噎廢食的態度，實在是無法認同。

沒多久 Teddy 也面對到同樣的問題...結果...走上同樣的道路...**「測試案例執行時間太久，所以為了縮短整合的時間，只能選擇不去執行**

**單元測試」**。如此向下沉淪，實非 Teddy 行事之道。很遺憾，已經如此苟且偷生快三年了，難道這就是傳說中的「理論與實務之間的差距」？以 Teddy 的情況而言，會演變至此有以下幾個原因：

- 把 code 送交到 CIS 之前「基本上」都有在開發環境（Eclipse）上執行 JUnit，所以在 CIS 上面沒有跑 JUnit 的罪惡感就沒有那麼重...XD
- 每次把 code 送交到 CIS 之後，過不久就會產生一個 Installer 程式，而 Teddy 所屬團隊後來都有一個習慣，會直接去測試這個新產生的 Installer。由於我們有採行一些辦法讓測試整個 Installer 的流程比較順一點，所以慢慢也就覺的沒有在 CIS 上執行單元測試好像還過得去。
- 因為我們的程式需要**跨平台**且**與硬體設備相依性很高**，而 CIS 目前是安裝在 Linux 上面。以前剛開始還有在跑 JUnit 的時候，有一些與平台或是硬體相依的 test cases 會失敗，也沒有人去理它，**因為大家都知道此為正常現象，請安心服用...XD**。久而久之就沒有去注意 CIS 上面執行失敗的測試案例。既然沒人去看，為了省時間就乾脆廢掉算了。
- 當然最後總歸一句就是要省時間，趕快拿到新產生的 Installer。
- 再加上後來有採行每天晚上自動執行功能測試案例，自此之後在 CIS 上沒有執行單元測試這件事情就更加逐漸的被遺忘。

***

其實省時間的另外一個方法就是**升級硬體**，自從換了高檔的伺服器作為執行 CIS 的電腦之後，「手頭寬裕了不少」，因此開始想要儘量回歸到原本 CI 的精神，除了執行單元測試以外，test coverage 與靜態程式碼分析（FindBugs、PMD 這類的工具）也都要執行一下。今天在 sprint demo meeting 時，大家看了一下 PMD 的報表，就當場不小心找到一個 bug。雖然是一個小小的 bug，但是當下大家都認同這件事情應該要做。PMD 這個工具大概 5-6 年前 Teddy 就用過了，但是都是心血來潮時稍微用了一下下，從來沒有持續使用下去。可是只要是跟別人介紹 CI 觀念，都會提到 test coverage、PMD、FindBugs 這類的工具。為什麼不能持續使用的原因姑且先不談（因為打字打到手痛了），現在把 test coverage、PMD、FindBugs 放到 CIS 上面，有以下幾個好處：

- 整個 team 的人都看的到，**專案變的更透明了**。例如，當 Teddy 看到某個 class 的 test coverage 很低的時候，就有一股衝動會想要去多寫幾個 test cases。這難道是「便利商店集點活動症候群」發作嗎？
- 由於專案變透明了，如果自己寫的程式被這些工具找出太多明顯的 bugs，那就有點「太難看了」。所以 programmers 會更「自愛」一點。往正面思考，可以學到一些平常疏忽的 coding 技巧或是應該要避免的壞習慣。

- Scrum Master 如果沒事可看看報表，很容易可以找出一堆有待改善的項目。
- Test coverage 如果很高，可以拿到客戶面前去「炫耀」。

啊，衣服洗好了，Teddy 要去批衣服了。

***

友藏內心獨白：好忙的一天，為了九顆硬碟得罪了一堆人。

# 79 用 Robot 寫自動化功能測試到底有沒有用

November 04 21:43~20:50

12/10 23:37-23:50

原文發表於 http://teddy-chen-tw.blogspot.com/2011/11/robot.html.

慫恿團隊使用免費且開放原始碼的 Robot Framework 來撰寫自動化功能測試也已經有超過一年的時間了，Teddy 曾在「土炮跨平台自動化功能測試環境」與「Automated acceptance tests」提過若干使用的經驗，今天想談一個問題：「用 Robot 寫自動化功能測試到底有沒有用」？

先定義一下何謂「有沒有用」，**簡單的說，就是能不能在可接受的成本範圍之內，寫出測試案例並找到 bugs**。如果可以，那就認為這個方法是有用的。要如何找？一言以蔽之「**做 regression testing**」，如何做？不花腦筋的方法：「**每天晚上下班之後讓測試系統自動跑所有的 Robot test cases**」，隔天早上再來看有那些 test cases 失敗，然後探討失敗的原因，看看是不是因為系統 bugs 導致的，如果是，就加以修正。

以上程序有做過自動化功能測試的鄉民們( 無論是否使用 Robot 或是

其他工具）應該都很熟悉，也沒什麼特別好說明的。Teddy 想討論的第一件事：**「以 Robot 這種透過 UI 來做自動化功能畫測試的方法是否值得」**？

有許多開發人員認為透過 UI 來做自動化功能測試是一種不穩定的測試方法，因為 UI 可能會經常改變，導致測試案例失敗，所以常常花了很多時間才發現「原來不是程式有 bugs，而是測試案例有 bugs（因為 UI 修改所導致）」。再加上，許多 UI 操作的 turnaround time（下達指令之後到該指令完成所花的時間）可能差異很大（尤其是在分散式系統中），這樣的功能要透過 UI 來測試系統就常常可能因為 timeout 導致測試案例失敗，但是實際上有可能只是某次的執行時間花的比較久一點（例如後端的資料庫正在忙碌，或是網路速度剛好很慢）。最後，自動化功能測試只能知道「該功能是否成功或失敗」，當功能失敗的時候，不容易直接看出原因，所以後續可能需要花比較久的時間來找出錯誤原因。還有一個開發人員可能不願意透過 UI 來做自動化功能測試的原因，那就是「這種測試案例不是很好寫」。

***

與「自動化單元測試（automated unit testing）」相比，自動化功能測試的確比較不好寫，因為後者要測的範圍比較大（透過 UI 做 end-to-end testing）。在剛開始導入 Robot 的前 2-3 個月，Teddy 也一

直在想同樣的問題：「讓團隊成員花時間寫 Robot 測試案例是否值得（到底有沒有找到 bugs 啊）」？因為剛開始的時候大家對 Robot 還不是很熟悉，所以寫一個 Robot test case 花的時間比較長，而且這些 Robot test cases 常常會因為「測試環境」的問題導致執行失敗。例如剛剛 Teddy 說過的 timeout 的問題（timeout 時間設太短），再加上測試環境的準備以及一些跨平台測試的因素，所以剛起步的時候幾乎看不出所寫的 Robot test cases 可以找出什麼 bugs，只能求這些 Robot test cases 可以穩定的執行。

為了讓撰寫以及維護 Robot test cases 可以持續正常，每個 sprint 團隊都有一個 technical story 是用來至少確保 Robot test cases 可以正常執行，如果還有時間則每個 sprint 還會持續增加幾個（個位數）新的 Robot test cases。

看到這邊可能有鄉民會問，為什麼不把寫 Robot test cases 當作每一個 story 的驗收條件？這樣不是每個 story 完成之後就有現成的自動化功能測試了嗎？理想上這樣是很好，也許有人採用 Acceptance Test Driven Development (ATDD) 就是這樣做的，但是目前團隊並沒有採用 ATDD，每個 story 基本上會有單元測試以及手動的人工驗收測試，等 sprint demo 之後，確定沒有大的介面修改，才會在後續的 sprint 中找時間來「補寫」Robot test cases。

***

導入 Robot 到現在也超過一年了，團隊成員寫 Robot test cases 越來越快，除了對工具本身熟悉度變高以外，還有 team member 熱心的寫了很多好用的 keywords（擴充 Robot）來加速撰寫測試案例的時間。Robot test cases 越來越穩定，當 test cases 執行失敗時找出原因的時間也越來越短，投資總算慢慢看到具體的回報（因為當 Robot test cases 失敗的時候，真的有找到系統的 bugs）。

有些 bugs 在單元測試的層次是不容易找出來的，因為問題可能出在 UI 端與後端元件的溝通上面。或是當整個系統整合起來時，想透過 UI 來測試效能的問題。又或是要確定最後打包的安裝程式是否能夠正常的安裝並執行，確認網頁上面每一個 link 按下去都是正常的。以上種種情況都還是需要做 end-to-end testing。

當看到 JUnit 上面都是「綠燈」的時候，程式設計師的心情會變得比較好一點。當看到 Robot test cases 都通過的時候，每個人的心情都會變好。每個 sprint 投資個 10-20 小時在自動化功能測試上絕對是頗值的的一件事。

***

友藏內獨白：「傻的願意相信」是很重要滴，團隊是在寫 Robot 之後，才學會 Robot 。

## 備註

1. Robot Framework
2. Automated acceptance tests
3. regression testing
4. Acceptance Test Driven Development (ATDD)
5. end-to-end testing

## 參考資料

[1]

[2]

# 80 用 Robot 寫自動化功能測試到底有沒有用 (2)

November 26 08:15~10:02

12/11 11:23-11:52

原文發表於 http://teddy-chen-tw.blogspot.com/2011/11/robot-2.html.

上次 Teddy 談到「用 Robot 寫自動化功能測試到底有沒有用」這個問題，最近剛好把開發的系統做了一些「升級」，更是體會到「Robot... 不能沒有你」。

話說因為種種原因，最近將系統做了以下幾項「重大」升級：

- 將系統內建的資料庫換成 PostgreSQL：原本系統內建一個 pure Java database，但是最近經過一番效能測試發現這個 pure Java database 在資料筆數「很多」時執行效率不是很好，所以安排時間將內建資料庫換成 PostgreSQL。系統使用 Hibernate 連接到後端的資料庫，「理論上」換成另外一個資料庫「應該」不會有什麼重大的問題，但是...沒測過誰知道啊。
- 支援真正 64-bit 作業系統：雖然系統是使用 Java 開發，但是還是有部份 native code。在「系統功能不變（沒有增加新功能）」的前提下，要支援「真正 64-bit 作業系統」除了要將 JVM（JRE）

換成 64-bit 以外，native code 也要編譯成 64-bit。這些用 C/C++ 寫的 native code「理論上」編譯成 64-bit「應該」不會有什麼重大的問題，但是...沒測過誰知道啊。

- 升級 jQuery：為了解決某些 Web UI 的問題，將 jQuery 升級到最新版本。「理論上」升級 jQuery「應該」不會有什麼重大的問題，但是...沒測過誰知道啊。

***

各位吃「軟飯」的鄉民們應該會和 Teddy 一樣有相同的感慨，軟體開發久了，最後發現絕大部份的時間都花在「測試」上面。假設相同一隻程式（code）也許只要寫（改）個 10 次好了，光是測試可能就要 10 * N 次。這是什麼意思？假設你的程式要支援 8 個作業系統（Window XP、Windows 2003、Windows 2008、Windows 7、RHEL 6.x、SUSE 11.x、Ubuntu 11.x、CentOS 6.x），32-bit 和 64-bit，以及 3 種瀏覽器（IE、Firefox、Chrome），Teddy 數學不好，請鄉民們自行算一下有幾種測試環境的組合。

你可能會說：「**Developers 只要測試主要的開發平台（1-2 種平台），其他的找測試部門做平台相容性測試就好了啊**」。「理想上」是如此，但是，如果你的團隊沒有配合的「測試人員或是測試部門」，那怎麼辦？你可以選擇「**讓客戶幫你測 ...XD**」，或是認命一點，自己測。

假設鄉民們和 Teddy 一樣都是「有理想，有 ~~報復~~ 抱負的工程師」，但是又不能因為這麼多的測試排列組合把自己給累死，那只能想想自動化測試的方法，讓「電腦」來代勞。**寧可電腦累死，也總比你自己累死要來的好吧。**

***

但是要讓電腦累死之前自己可能會先累死，因為：

- 要寫自動化（功能）測試案例。以 Teddy 的例子，團隊持續花時間用 Robot 寫了許多自動化功能測試案例 。經過 Teddy 詢問 team members，**寫完一個 Robot test case 平均大概需要「四小時」（包含寫這個 test case 和測試這個 test case 的時間）**。
- 要準備測試環境。光有自動化功能測試案例，沒有測試環境也是白搭啊。以剛剛 Teddy 提到的那 8 個作業系統的例子，至少就要準備 16 個作業系統（8 個作業系統各 32-bit 和 64-bit 一共 16 個）。

看到這邊鄉民們可能會大喊一聲...等一下....「我們可是窮苦人家的小孩耶，去那邊生 16 台電腦出來安裝這 16 個測試環境？」。關於這一點請參考「土炮跨平台自動化功能測試環境」裡面介紹的方法。什

麼....你說:「用 VM 不就好了」...~~我還摻在一起做瀨尿牛丸勒~~....嗯嗯...這位「涉世未深」的鄉民，如果你的系統需要讀取硬體的資料，就不能用 VM 來當作測試環境啦。

如果鄉民們參考 Teddy 在「土炮跨平台自動化功能測試環境」裡面所建議採用「DRBL Server 管理測試影像檔」的方法，準備好這 16 個測試環境就沒事了嗎？當然不是，因為採用 DRBL Server 管理測試影像檔的方式主要是用來在下班之後跑 **nightly build**，每次 restore 一個測試影像檔可能需要數分鐘到 10 幾分鐘的時間，如果再加上執行 Robot test cases 的時間，一定無法滿足「Ten-Minute Build」的要求。如果要用這種方法在白天開發程式時拿來當作測試的環境，這種測試方法的 turnaround time 就稍嫌長了一點，程式設計師人員也不會想要去用（測試）。

那怎麼辦？兩種方法：

- 多增加若干台「實體機器」以執行與硬體相關的測試：這些給 developers 在「白天」用來測試的「實體機器」，可以是原本完整測試環境的「子集合」。以前面提到 8 個作業系統的例子，最少可以只準備一台 Windows 64-bit 電腦與一台 Linux 64-bit 電腦就好了。Developers 在本機上寫好的程式可以丟到這些實體機器上面測試。

- 增加一組或以上的 VM 以執行與硬體無關的測試：對於與硬體無關的功能，可以依據「公司口袋的深度 硬體測試資源的多寡」準備很多組 VM，例如準備上述 16 種完整的測試環境，或是完整測試環境的子集合。然後使用類似 Jenkins 這種「分散式持續整合系統」，當 developers 認為需要的時候，在 Jenkins 上面只要輕輕按下「建構按鈕」便可將所開發的軟體系統「**同時丟到這些 VM 上面跑 Robot 測試案例**」。那種感覺....很...過...癮...尤其是全部 test cases 都 pass 的時候....不過此種現象出現的機會大概跟 9 星連成一線一樣，百年難得一見....XD。

***

又扯了這麼多，你看的不累，Teddy 都寫到累了...XD。總結一下：

- **要持續寫 Robot test cases**：剛開始寫一個測試案例可能會花 8 小時以上，熟了之後應該可以在 4 小時完成一個 Robot 測試案例。想到這個測試案例可以一直使用，這 4 小時的投資算是滿划算的。
- **準備三種不同的測試環境**：（1）用 DRBL Server 管理給 nightly build 用的眾多實體測試環境之 OS images；（2）準備若干台實體機器（通常是完整測試環境的子集合）給 developers 白天開發軟體與測試之用；（3）依據公司口袋深度準備若干（越多越

好）VM，在白天上班時間由持續整合系統定期（例如每小時一次）或是由 developers 手動將自動化測試案例「同時」丟一組 VM 上執行。

- **找專人持續關注測試案例執行後的結果**：透過 UI 執行的自動化功能測試通常是很「脆弱」滴，很有可能因為「不明原因」執行失敗（有時成功，有時失敗）。因此，要指定 ~~地獄倒楣鬼~~ 專門技術人員，持續關注測試案例執行結果。看看測試案例執行錯誤原因是因為程式錯誤，測試環境問題，還是測試案例撰寫問題，然後加以排除。

***

什麼，你問 Robot test cases 有沒有找到什麼問題？有啊，舉的例子，最近換了新版的 jQuery，原本 UI 上面的某個 check box 突然消失了 。這個問題就被「檢查所有網頁連結」的 Robot test case 給找出來了。那麼現存的 Robot test cases 會涵蓋到所有的軟體功能嗎？別傻了...當然...還沒有（還沒寫到那個地步啊....），所以，人工測試還是有需要滴。當人工測試找到新的 bug 時，要記得為此 bug 加上一個新的 Robot test case，以確保日後如果相同的問題再次出現，可以被測試案例給逮住。這也算是符合 agile 精神... incremental growth... 持續做下去，就可以達到「顏回」不貳過的境界啦。

***

友藏內心獨白：禮拜六為什麼一大早就起床寫部落格...。

Todo

畫架構圖說明跨平台整合環境

## 備註

1. PostgreSQL
2. Pure java database
3. Hibernate
4. native code
5. 64-bit, 32-bit
6. JVM (JRE)
7. JQuery
8. DRBL Server
9. nightly build
10. Ten-Minute Build
11. 實體機器
12. 虛擬機器
13. incremental growth

# 81 誰 cover 誰

Jan. 14 23:00~ 15 00:12

12/18 18:39-18:45

原文發表於 http://teddy-chen-tw.blogspot.com/2011/01/cover.html.

話說當年 Teddy 在修「軟體測試與驗證」 這門課的時候，老師曾經花了不少時間教導如何判斷測試是否足夠的條件（test adequacy criteria）。總的來講（Teddy 內心獨白：劉老師，Teddy 對不起你。內容忘的差不多了，趕快偷看一下當年的上課投影片）program based structural test adequacy criteria 可分為兩大類：

- control-flow criteria
- data-flow criteria

其中 control-flow criteria 又包含：

- statement coverage
- branch coverage
- condition coverage
- relation coverage
- branch/condition coverage (BCC)
- compound condition coverage

- modified condition/decision coverage (MC/DC)
- path coverage

而 data-flow criteria 又包含：

- all-paths
- all-du-paths
- all-uses
- all-defs
- all-c-uses
- all-p-uses
- all-c-uses/some-p-uses
- all-p-uses/some-c-uses

正常的人類看到這邊差不多快吐了，如果你以為就只有上面這些 test adequacy criteria 那你就太單純了。所有的分類一定少不了「其他」這一類：

- function coverage
- linear code sequence and jump (LCSAJ) coverage
- call coverage
- object code branch coverage
- race coverage
- weak mutation coverage

- table coverage

套句電視模仿節目的台詞：「台灣媒體這麼多，新聞怎麼報？就亂報嘛！」...「Test adequacy criteria 這麼多，怎麼測？就亂測嘛！」當年 Teddy 聽完這一「拖拉庫」的方法之後心都涼了一半。如果是有規模且上軌道的軟體公司（很抱歉，這種公司好像大部分都不在台灣），上述方法當然是用來評估「測試是 否足夠」的重要參考。但是，在寶島台灣，也許 90% 以上的軟體專案人數都在 10 人以下，更別奢望有專屬的測試團隊來協助開發人員測試軟體。所以，假設鄉民們就是這「微型開發團隊的一 ~~元~~ 員」，那您要如何做測試？

Agile methods 有教，要寫 unit tests，做 TDD 或是 ATDD (Acceptance Test Driven Development)。問題是，常常寫了一堆 test cases 全部都通過，此時不禁懷疑這些「測試案例的有效性（能不能真的找到 bugs）」。

除了基本的單元測試以外，另外還有一招 Teddy 覺的是滿有用的，**就是當發生 bug 的時候，先寫（後寫也行啦，但是記得一定要寫）一個可以反應出該 bug 的測試案例，等修完 bug 之後這個測試案例就通過了**。 這樣可以確保已經發生過的 bug 萬一以後再發生的話有測試案例可以抓到它。聽起來很簡單也很有道理，不過說真的要持續做到不太容易。大體上 programmers 都會想趕快看 code 或用 trace

的方式來找出問題。一旦問題解決後，常常會懶的補測試案例。

最近 Teddy 發現，用 Robot 寫 acceptance tests 來確保功能錯誤不再發生是一個不錯的作法。開發團隊可以在每個 sprint 安排一個 story 專門為系統的某個（或某些）功能「補寫」acceptance tests。至於要寫哪些 acceptance test cases 可以用兩個很簡單的方式：

- 如果有軟體的使用手冊，就依據手冊上面操作軟體的方法來設計 acceptance tests。
- 找出所有使用者或是自己開發軟體時所發現的 bugs，寫 acceptance tests 來確認這些 bugs 已經消失了。

Teddy 認為以上作法對於「微型開發團隊」還滿有用的。不過，要真正落實這一點，有一個先決條件，就是要**建置一個「自動化測試環境」，並且要讓在這個環境中加入一個 acceptance tests 這件事變得很透明且很簡單**（這 是要花時間滴）。例如，如果要開發跨平台軟體，就要考慮到如何讓這些 acceptance tests 可以「自動的」被佈署到不同的平台上測試；如果要測試 Web 應用程式的介面，則可能要熟悉類似 Selenium 這一類的工具。能夠做到這一點，要持續增加新的 acceptance test cases 就變得比較可行。

結論：空有武林秘笈，沒有弟子來練功也是白搭。

***

友藏內心獨白：與其說把這些 coverage 忘的差不多了，還不如承認當年根本沒學好...Orz。

# 第九部 還少一本書

# 82 The Timeless Way of Building

02/06 23:33 ~ 02/07 02:19

12/15 12:11-12:27

原文發表於

http://teddy-chen-tw.blogspot.com/2010/02/timeless-way-of-building.html.

幾天前在 Facebook 上的「Scrum Community in Taiwan」看到十分熱心的柯兄轉貼的一篇文章 Top 20 Best Agile Development Books, Ever， 這 20 本書 Teddy 碰巧翻過 18 本 （請注意，翻過並不等於看完，更不等於看懂...只是... 翻過，沒什麼特別的意思...），這些書有空的話都應該要「翻」一下。不過，Teddy 對於原作者的排名，略有一點意見。引用一下原作者的資料，看一下前 10 名的排行榜：

1. Agile Software Development: Principles, Patterns and Practices by Robert C. Martin
2. Refactoring: Improving the Design of Existing Code by Martin Fowler
3. Agile Estimating and Planning by Mike Cohn
4. User Stories Applied: For Agile Software Development by Mike Cohn

5. The Pragmatic Programmer: From Journeyman to Master by Andrew Hunt and David Thomas
6. Agile Software Development: The Cooperative Game (2nd Edition) by Alistair Cockburn
7. Agile and Iterative Development: A Manager's Guide by Craig Larman
8. Extreme Programming Explained: Embrace Change (2nd Edition) by Kent Beck
9. Agile Project Management: Creating Innovative Products by Jim Highsmith
10. Continuous Integration: Improving Software Quality and Reducing Risk by Paul M. Duvall, Steve Matyas, and Andrew Glover

再怎麼輪，榜首也不應該是 Agile Software Development 這一本啊。Teddy 認為這本書寫得最棒的是專門講設計原則的章節（第 8 到 12 章以及第 20 章，*The Single-Responsibility Principle, The Open-Closed Principle, The Liskov Substitution Principle, The Dependency-Inversion Principle, The Interface-Segregation Principle, Principles of Package Design*）...自首無罪... 其實 Teddy 也只看了這幾章...不過這不是重點，重點是，雖然學會並實用這幾個設計原則就可以騙吃騙喝 ~~一輩子~~ 一陣子，但是很多原則並不是作者原創性的作品 。例如，The

Open-Closed Principle 和 The Liskov Substitution Principle 在更早之前 Bertrand Meyer 的名著 Object-Oriented Software Construction 這本書就有提到了（這本書很大一本，如果地震時不小心從書架掉下來打到頭絕對有把人打到腦震盪的實力。書中還有一狗票的設計原則等待鄉民們去挖寶）。

從抽象到具體的順序來看，Teddy 覺得第一名應該是 Kent Beck 寫的 Extreme Programming Explained: Embrace Change (第一版和第二版都應該看一下)，因為 Teddy 是讀了 Kent Beck 的書才接觸 agile methods 的，所以本書自然榮登 Teddy 心目中永遠的第一名（此時腦海中浮現出海綿寶寶的影像）。

***

依照往例，以上內容也不是本篇的重點，以下才是本篇的重點。

俗話說「**女人的衣櫃總是少一件衣服，女人的鞋櫃總是少一雙鞋子，男人的車庫總是少一部車子，小朋友的卡通總是少看一個小時，宅男的電腦總是少一顆硬碟**」。依此類推，「程式設計師的書櫃，應該總是少一本書」。今天 Teddy 要補上這一本書，一本 agile people 應該要看的書：*The Timeless Way of Building* （建築的永恆之道）。

看過 Design Patterns 的鄉民們應該有聽過 The Timeless Way of Building 這本書。Teddy 當年因為要研究 patterns 在其他領域的應用才去看 The Timeless Way of Building。在看這本書的當下，只知道它是啟發 software patterns 理論與應用的始祖，並不覺的它和 agile methods 有什麼特別的關係。簡單的說，作者在這本書對於「如何建構有活力，有生氣的建築物」闡述自己的一套理論，這個理論就是 pattern languages 理論。以 Teddy 有限的智慧，無法三言兩語說明清楚（ㄟ，其實 Teddy 也懷疑是否真的了解... 鄉民甲：你是來鬧的嗎？），直接看一段書上的文字比較快：

*It (i.e., the timeless way) is a **process** which brings **order** out of nothing but ourselves; **it cannot be attained, but it will happen of its own accord, if we will only let it.***

對應一下敏捷精神，敏捷方法是一種流程（process, 也許有些人不喜歡把敏捷方法看作是 process，不過這裡把 process 看成做事情的步驟或方法就好，不用想成重量級的大部頭流程），可以為開發團隊帶來某種「秩序」或「結構」。要變成一個敏捷團隊需要靠團隊自身的實踐，無法以強制規定或是工廠管理的方式來建立敏捷團隊。

*The prople can shape buildings for themselves, and have done it for centuries, by using languages which I call **pattern languages**. A pattern*

*language gives each person who uses it the power to create an infinite variety of new and unique buildings, just as his ordinary language gives him the power to create an infinite variety of sentences.*

對應一下敏捷精神，敏捷方法強調 no big upfront design、evolutionary design、simple design，除非開發團隊成員具有很強的 software patterns 基礎（當然還要有 testing 和 refactoring 等配套，等一下會提到），否則要滿足這些設計精神是很不容易達到的目標。

Once the building are conceived like this, they can be built, directly, from a few simple marks made in the ground---again within a common language, but directly, and without the use of drawings.

對應一下敏捷精神，XP 不是告訴我們，沒什麼事不要畫太多設計圖的啦 (source code is the design)。

*Several acts of building, each one done to **repair** and **magnify** the product of the previous acts, will **slowly generate a larger and more complex whole** than any single act can generate.*

對應一下敏捷精神，有沒有聞到 refactoring、iterative and incremental 的味道。

*Within the **framework** of a common language, millions of individual acts of building will together generate a town which is alive, and whole, and **unpredictable**, **without control**. This is the slow emergence of the quality without a name, as if from nothing.*

對應一下敏捷精神，Scrum 本身是一個 framework，採用 empirical management model 而不是用傳統的 defined processes 來管理負責專案。

寫到這邊 Teddy 和周公的約會已經遲到兩個多小時了，加上這本書的內容實在過於豐富又有點抽象，在沒有準備的情況下 Teddy 實在也掰不下去了。總之，這本帶點哲學味道的書，對於想要獲得「華山論劍」資格的 agile 鄉民們，是屬於不可不搶的「九陰真經上卷」，練完之後如果沒有走火入魔保證內力大增。

***

友藏內心獨白：Teddy，你不要看到英文單字一樣就隨便連連看，小心被抓包。

# 83 Smalltalk Best Practice Patterns

03/18 22:49~23:55

12/15 12:39-12:51

原文發表於
http://teddy-chen-tw.blogspot.com/2010/03/smalltalk-best-practice-patterns.html
.

連續寫了幾天的 exception handling，今天換個口味，介紹一本在台灣應該算是比較冷門的書：Smalltalk Best Practice Patterns, by Kent Beck.

雖然 Kent Bec 是 Teddy 的偶像，但是 Teddy 也不是「好野」到可以買下大師所寫得每一本書。當初看到書名想說這是一本談 Smalltalk 的書，而 Teddy 又完全不懂 Smalltalk，所以雖然書名也有 Patterns 這個字，但是也沒特別想買（在台灣好像找不到進口這本書的書商，當初是直接從 Amazon 買的）。那，為什麼最後還是買了，而且還要推薦？

記得當年 Teddy 在學 JUnit 設計的時候，看到 JUnit 使用到「Collecting Parameter」這個以前沒有看過的 pattern (大膽，居然使用了 GoF 之外的 pattern)，而這個 pattern 正是 Kent Beck 在這本書中

有介紹到的。此時 Teddy 猜想，這本書應該還有一些其他好料的，所以雖然看不懂 Smalltalk 還是買下來了。

先自首一下，這本書從 2002 年 5 月 17 號拿到手，到現在 Teddy 只看了 1/3。不過這不是重點，重點是，2003 年 Teddy 在學 Eclipse 設計的時候，居然發現 Eclipse 還有 SWT 也用到好幾個在這本書中有提到的 patterns。由於買了這本書，所以就比較容易了解 Eclipse 與 SWT 裡面一些 patterns 的用法 (廣義的來說，應該是 Eclipse 與 SWT 受到了 Smalltalk 的若干影響)。

舉一個例子，SWT 的 UI 實做了 **Variable State** 這個 pattern, Teddy 把書中關於這個 pattern 的說明引用一下：

***Problem**: How do you represent state whose presence varies from instance to instance?*

***Solution**: Put variables that only some instances will have in a Dictionary stored in an instance variable called "properties." Implement "propertyAt: aSymbol" and "propertiesAt: aSymbal put: anObject" to access properties.*

翻成白話文就是說，如果屬於同一個 class 的不同 instances 需要各自保有不同的狀態, 那麼請在該 class 中宣告一個 Dictionary(在 Java 裡面可以用 HashTable 之類的物件)的 data member(attribute)。每一個由這個 class 產生的不同 instances 就可以用這個 Dictionary 來存儲各自所需的資料（鄉民甲怒喊：這只是從英文的文言文，換成中文的文言文啊）。

不知道鄉民們是否有這樣的經驗，有一個現有的 class 雖然符合你的需要，但是你還需要增加一，兩個 data member 來儲存其他狀態。如果這個現有的 class 沒有實做 variable state pattern，那麼你可能就必須要利用「繼承」來將你所要增加的 data membe 寫在 subclass 裡面。鄉民們應該也都知道好的物件導向設計其實對於「繼承」的使用是很「節制」的，在上述例子中算是有點濫用繼承。舉的 SWT 的例子，在寫 GUI 程式的時候，有時候為了方便起見會把一些變數（或是物件）直接存在 UI 元件上面。假設你用了一個 Button 元件，因為否種原因你想把某個變數存在這個 Button 上面（至於是什麼原因，Teddy 也不知道..!$!@#%）。如果這個 Button 元件沒有實做 variable state pattern，那麼你就要寫一個 MyButton 的 subclass 來儲存這個變數，有點殺雞用火箭炮的感覺。

如果 Teddy 上面所講的鄉民們還是看不懂，沒關係。買一送一，再舉一個.NET 的例子。有一次 Teddy 在研究 exception handling 的時候

意外發現.NET 的 `Exception` 類別好像是從.NET 2.0 還是 3.0 之後，就實做了 variable state 這個 pattern。很可惜 Java 的 `Exception` 類別並沒有實做這個 pattern，所以假設你想要在 Java 的 `IOException` 上面多夾帶一些資料，以便於讓收到 exception 的人可以做進階的處理，很抱歉，你必須繼承 `IOException` 才行。

***

好像有點離題，變成在介紹 variable state pattern。總之，這本書中還有很多在其他地方比較看不到的資料，例如，如何幫 instance variable (i.e., data member)取名字和 pluggable selector 等等, 值得買來**珍藏**（就是擺著不看的意思…XD）。

最後，引用 Kent Beck 在書中所寫得一小段 Teddy 很喜歡的文字作為結尾。

*To me, development consists of two processes that feed each other. First, you figure out what you want the computer to do. Then, you instruct the computer to do it. Trying to write those instructions inevitably changes what you want the computer to do, and so it goes. (PS:這一段是起頭，後面那一段才是重點)*

*In this model, coding isn't the poor handmaiden of design or analysis. Coding is where your fuzzy, comfortable ideas awaken in the harsh dawn of reality. It is where you learn what your computer can do.* ***If you stop coding, you stop learning.***

***

友藏內心獨白： 據說"If you stop coding, you stop learning."是 Teddy 的座右銘。

# 84 Release It!

06/02 20:45~22:1106/02 20:45~22:11

12/16 10:41-10:49

原文發表於
http://teddy-chen-tw.blogspot.com/2010/06/release-it-design-and-deploy-production.html.

Teddy 寫了「600 多個 bugs 要怎麼修？」之後，突然想到之前看過的一本書：*Release It! Design and Deploy Production-Ready Software*，看書名就知道這本書的目的，的確，作者 Michael T. Nygard 也沒有唬爛讀者，這的的確確是一本值得掏錢出來買的好書。

想當年 Teddy 退伍後投入軟體業，也是很努力的開發了幾個系統，但是說實話一路上跌跌撞撞，總是有 de（修）不完的 bugs。為什麼開發出來的軟體就是無法有很好的品質？當然原因很多，資源不足是主要的原因，開發時程訂的很 ~~敢~~ 趕，也沒有專屬的測試人員，因此軟體普遍缺少完整的測試。不知道如何寫出穩定的軟體則是另外一個原因，其中包含 coding 功力不夠、exception handling 處理不良（有在處理嗎？）、軟體工程的 practices 做的不夠紮實等等。

正所謂「滿天全金條，麥殺 (要抓) 沒半條」，就算是 ~~買~~ 看了一堆

軟體開發的書還是不知道怎樣才能做出 Production-Ready Software，此時這本書便可派上用場。

這本書介紹了許多關於 **stability** 與 **capacity** 的 patterns 與 anti-patterns。看到 patterns 這個字不要被嚇一跳，本書 patterns 寫作的方式以文字說明為主，簡單易懂有又深度（深入淺出）。在這邊舉幾段書中的敘述給鄉民們參考一下：

**Chapter 5: Stability Patterns**

**5.1 Use Timeouts**

The **Timeouts** and **Fail Fast** patterns both address **latency problems**. The Timeouts pattern is useful when you need to protect your system from someone else's failure. Fail Fast is useful when you need to report why you won't be able to process some transaction. Fail Fast applies to incoming requests, whereas the Timeouts pattern applies primarily to outbound requests. They're two sides of the same coin.

這兩個 patterns（Timeouts 和 Fail Fast）Teddy 在開發系統的時候經常用到。先講 Fail Fast，這個 pattern 其實就是 Teddy 之前介紹過的 Exception Handling Robustness Level 裡面的 RL1 (robustness level

1)。意思是說，發生任何的 exceptions 都要立即回報，不可暗槓例外（do not ignore exceptions）。所以 Fail Fast 是有良心的 callee 要保護 caller，確保當 callee 有問題時會告知 caller，不會像有些人偷偷打一發魚雷然後又說不是自己幹的（發生 exception 又不承認，要找 bug 就很難了）。Timeouts 則是 caller 要保護自己，不會因為 callee 沒有回應而把自己給搞掛了。Timeouts 這個機制日常生活中人人都會用到，例如，鄉民們打電話給別人，如果對方沒接，總不可能讓電話一直響下去吧。一般正常人可能響個七~八聲沒人接就會把電話掛掉了。這個「電話響多久沒接就掛掉」就是 Timeouts，保護打電話的人不會一直空等。

鄉民甲：那我寫程式都沒有用到 Timeouts 耶。

Teddy：如果是寫單機版的程式，可能比較少用到 Timeouts。但是，只要寫網路應用程式或是資料庫程式就會經常遇到。例如，建立網路或資料庫連線可能會失敗，如果不設 Timeouts 那麼程式可能會一直等下去，讓 users 以為系統當機。執行外部程式則是另外一個例子，這些外部程式可能會「卡住」，如果不設 Timeouts 整個系統也會跟著卡住。但是設 Timeouts 也是有學問的，要長到可以讓大部分的工作都來的及完成，又不能長到讓 user 覺的等太久而產生系統當機的錯覺。

***

### 5.2 Circuit Breaker

**... It is a component designed to fail first, thereby controlling the overall failure mode.**

... More abstractly, the circuit breaker exists to allow one subsystem (an electrical circuit) to fail (excessive current draw, possibly from a short-circuit) without destroying the entire system (the house). Furthermore, once the danger has passed, the circuit breaker can be reset to restore full function to the system.

You can apply the same technique to software by wrapping dangerous operations with a component that can circumvent calls when the system is not healthy. This differs from retries, in that circuit breakers exist to prevent operations rather than reexecute them.

Circuit breakers are a way to automatically degrade functionality when the system is under stress.

上面的說明應該很清楚了，直接舉個例子。假設鄉民們寫了一隻網路程式，可以接受 client 透過 TCP/IP 連線。如果同時間有大量的 requests 連過來（生意太好或是被駭客攻擊）而你的程式沒有做任何保護，那麼可能會因為建了太多連線導致資源不足而讓程式當掉或是反應緩慢到接近當掉，總之就是無法 提供任何服務。如果套用 Circuit Breaker pattern，那麼當同時連線數目到達某一個數量之後，就暫時把接受新連線的功能關閉，等系統的負載降到某種程度之後再重新開啟接受新連線的功能。這樣 做當然會導致系統服務水準降低，但是至少可以讓系統提供某種程度的服務，至少不會接近死當。正所謂「好死不如賴活著」，就是這個道理。

其他更多更精彩的內容就請鄉民們自己去發掘了。

***

友藏內心獨白：有時候要看了好幾本爛書，才會看到一本真正的好書。

# 85 Managing the Software Process

Feb. 21 21:28~23:41

12/16 18:24-18:39

原文發表於

http://teddy-chen-tw.blogspot.com/2011/02/managing-software-process.html.

前一陣子 Teddy 看到一個消息，Watts S. Humphrey 這位軟工界的大師過世了(4 July 1927 - 28 October 2010)，這算是軟工界一個很大的損失 。Humphrey 博士寫了很多軟體工程方面的書籍與論文，對軟工領域可謂貢獻良多。大體上 Humphrey 博士的精神是**「一個流程如果無法測量，就無法改善」**，因此在他的書中可以看到很多制式的表格，提供軟體開發者一個「測量」的基準。這種精神反應到了極致，就是在軟工界鼎鼎有名的 CMM 以及後來進化版的 CMMI，因此有人稱呼 Humphrey 博士為 CMMI 之父。

今天要推薦的這本書就是 Humphrey 博士在 22 年前所寫得 *Managing the Software Process* 這本書。

路人甲：Teddy 你不是敏方法這一國的嗎，怎麼可以「為匪宣傳」？

路人乙：1989 年出版的「舊書」現在還在推薦？

***

還沒開始介紹就有鄉民們看的不爽了，請容 Teddy 娓娓道來。首先，Teddy 並沒有「討厭」或「反對」CMMI。而且 CMMI 並不是一種「流程」，而是一種「**評估流程成熟度」的框架（方法）**」，軟體團隊可以選用任何的流程來達到 CMMI 所要求的評估標準。你可以用 Waterfall、用 RUP、用 agile methods、或用吸星大法，只要符合 CMMI 對於每一個流程成熟度等級的要求便可（當然如果要獲得證書是需要花一筆錢滴）。

講的再白話一點，CMMI 就是一種「評鑑標準」，就好像唸國中的時候學校有「整潔競賽」一樣，整潔競賽的評分標準有「地上有沒有垃圾」、「抽屜有沒有髒東西」、「桌椅有沒有對齊」、「黑版有沒有擦乾淨」、「垃圾有沒有倒」、「門窗電燈有沒有關好」、「打掃用具有沒有歸位」等等囉哩八唆的一堆規定。打分數的老師就用這些「評鑑標準」來評量你們班級的「整潔成熟度等級」。至於要如何達到「高整潔成熟度」，則是不會被規範在「評鑑標準」中。有的班級可能會用「排值日生」的方式，有的人會要求「全班留下來打掃」，有些「富二代」的班級可以把打掃工作「外包」，有的班級可能會花錢買「iRobot」外加「好神拖」以提昇打掃效率（一個以 xxx 工具支援的

自動化打掃流程...XD）。總之不管是黑貓白貓，只要能通過「評鑑標準」的就是好貓。是不是這個意思？就是這個意思。

講了這麼多好像都不是重點，總之：

1. 介紹 Managing the Software Process 或是 CMMI 不算為匪宣傳。
2. 此書雖然已經出版 22 年了，但是如果你是那種「測量的擁護者」，那這本書還是十分值得一看。此外，本書還是有很多軟體工程觀念，作法與名詞介紹，歷久彌新。
3. 就算你是那種很不爽 CMMI 的人，還是要看。如果不「深入敵營」，如何打探消息並將敵人殲滅。

***

終於要講到書中內容，第一章開宗明義介紹 A Software Maturity Framework

*The objectives of software process management are to produce products according to plan while simultaneously improving the organization's capability to produce better products. The basic principles are those of* ***statistical process control****....*

*The basic principle behind statistical control is **measurement**.*

第一章提到了五個 process maturity levels:

- Initial
- Repeatable
- Defined
- Managed
- Optimizing

***

第二章介紹 The Principles of Software Process Change

- Major changes to the software process must start at the top. （就是說老闆要出來鎮壓反對勢力，至少不可以自己帶頭反對...XD）
- Ultimately, everyone must be involved. （就是說最終要把公司上上下下每個人都搞得很累...XD）
- Effective change requires a goal and knowledge of the current process. （減肥前要先拍照「證明」自己胖的跟條豬一樣，這樣減肥成功後才可以拿來跟前男友炫耀...XD）

- **Change is continuous.** (改善是沒完沒了滴，就是說顧問的錢要一直付下去不可以省...XD)
- Software process changes will not be retained without conscious effort and periodic reinforcement. (這一句也很重要，想一下，連 Scrum 裡面都有若干角色/機制可以呼應這一點。)
- Software process improvement requires investment. （Teddy 內心獨白：這一條的意思應該不是叫大家每次都花個 100 萬新台幣去換個 CMMI Level X 證書吧... XD）

***

有看電視的人都知道，醜女（男）在進行大改造之前，都要被現場來賓狠狠的批評一下。同理可套用在軟體流程改善上。第三章介紹 Software Process Assessment 的幾點原則。

- The need for a process model as a basis for the assessment.
- The requirement for confidentiality.
- Senior management involvement.
- An attitude of respect for the views of the people in the organization being assessed.
- An action orientation.

***

第四章談 The Initial Process，也就是 Level 1。一言以蔽之，Level 1 就是「亂七八糟」。接下來 5-8 章介紹 The Repeatable Process；9-14 章是 The Defined Process；15-16 章是 The Managed Process；17-20 章是 The Optimizing Process。在此列舉幾個大家比較熟悉的內容：

- 第六章 The Project Plan
- 第七章 Software Configuration Management (Part 1)
- 第八章 Software Quality Assurance
- 第十章 Software Inspections
- 第十一章 Software Testing
- 第十二 Software Configuration Management (Part 2)
- 第十七 Defect Prevention
- 第十八 Automating the Software Process

***

總之，這本書用「工程」的方法告訴鄉民們「軟體應該要這樣開發滴」，不管看完之後是否買帳，多了解一種看法也是不錯的。另外，這本書不會只告訴你要達到 Level x「應該」做這個，做那個，而且會實際告訴你「實做方法」。如果想導入 CMMI 的人看這本書 *CMMI: Guidelines for Process Integration and Product Improvement* 看的一頭霧水的時候，不仿先看看 *Managing the Software Process*。

***

友藏內心獨白：Humphrey 博士 2010 年 10 月逝世，而他的最後一本書 *Reflections on Management* 出版於 April 8, 2010。這豈不是說他老人家到生命的終點都還在寫書，真是太令人敬佩了。

# 86 The Unified Software Development Process

Feb. 22 22:41~Feb. 23 00:16

12/16 18:41-18:55

原文發表於

http://teddy-chen-tw.blogspot.com/2011/02/unified-software-development-process.html.

昨天介 紹了對於了解與實做 CMMI 有助益的 *Managing the Software Process* 這本書（Teddy 還是有照顧一下喜歡 CMMI 的朋友們喔），今天就順便介紹一下 *The Unified Software Development Process* 這一本。本書的作者也是鼎鼎大名的三巨頭 Ivar Jacobson、Grady Booch、James Rumbaugh。如果說昨天介紹的 Watts S. Humphrey 博士是 CMMI 之父，那麼今天介紹這三位 (Jacobson、Booch、Rumbaugh) 可以說是 UML 之父了。

這邊先澄清一個觀念，就好像常常有人會說：「我們團隊使用 CMMI **流程**來開發軟體」這種錯誤觀念（CMMI 並不是一種開發流程，原因請參考昨天內容），也常常會聽到有人說「我們使用 UML 來**分析**軟體專案」或是「我們使用 UML 來開發軟體」這樣的錯誤說法。哪裡錯了？

- UML 的全名是 Unified Modeling Language，這邊偷用一下 wikipedia 的解釋 「*UML includes a set of graphic notation techniques to create visual models of software-intensive systems.*」所以常常有人說 UML「只是」一種「畫圖（講好聽一點就叫做 visual modeling）」的表示法而已。了解這一些標準的「畫圖表示法」，例如 Class diagram、Sequence diagram、Activity diagram、Use case diagram 等等，並不能說你就知道要如何「分析」軟體。就好比知道各種幾何圖形並不表示你知道要如何畫出一幅好的圖畫一樣。這是兩件事情。
- 你可以用 UML 來表達軟體設計，但是你不能用 UML 來「開發軟體」。你可能會不服氣的說，可以啊，書上有說，UML 圖畫好之後按下「Code Generation」就把程式自動產生好了，還可以選要產生 Java、C#或是 VB.NET 各種不同的語言喔，連一行程式都不用。也許有一天大家真的可以用這種方式來「開發軟體」，但不是今天。

明明是要介紹 Unified Software Development Process(簡稱 Unified Process，UP) 為什麼 Teddy 又扯到 UML，因為 UP 和 UML 的作者都是同一批人，而且在看 UP 書籍或相關資料的時候，又經常看到一堆有的沒的 UML diagrams，所以鄉民們很容易就會 **UP, UML 傻傻分不清楚**。

再幫鄉民們問一個問題，那沒事幹麼學 UP，不是用 SCRUM 或 XP 這種 agile methods 就好了？答案也很簡單：

- 現實世界中，專案的大小，型態各異，多了解幾種軟體開發流程可以讓自己面對到不同的專案時，有多一點的選擇。就好像一個 programmers 不可能只會一種 programming language 一樣。就算你覺的 Java 是全世界最棒的語言，如果你真的傻傻的一輩子就只會 Java，那是很危險滴。
- 和別人（「兵」）吵架的時候，多知道一點東西比較不會被唬的一愣一愣的。有人說「知識就是力量」，Teddy 說，「**知識就是吵架本（吵架的本錢）**」。

***

Teddy 的部落格都是「當天現撈的」，常常一不小心寫太晚搞得 Teddy 晚上失眠。接下來以大易輸入法的方式快速介紹一下本書重點。

UP 的三大支柱：

- **Use-Case Driven**：這一點其實很有趣，如果了解 Scrum 的人把 Use-Case Driven 換成 Story Driven 這樣就可以了解這一點的含意。在 Scrum 中，軟體開發流程是在每一個 sprint 中挑選若干個 story 來開發，最後在第 N 個 sprint 的時候把軟體

做完。所以我們也可以說 Scrum 是 story driven（story「驅動」著 Scrum 流程的每一項活動）。在 UP 中，這樣的精神是很類似的，只是 UP 用來紀錄需求的單位是一個 Use Case。

- **Architecture-Centric**：這一條解釋起來比較難一點，大體的精神是在軟體開發的「較早階段」，利用先「實做」最 重要 5%～10% Use Cases 的機會建構起 software architecture 的「雛型」。日後隨著越來越多的 Use Cases 的內容被釐清與實做，這個 architecture 逐漸「~~轉大人~~ 成熟」。
- **Iterative and Incremental**：這一點就比較沒什麼好解釋，UP 和現代主流的軟體開發流程一樣，都是支援 IID 的流程 (Iterative and Incremental Development Process)。

其實這本書的重點 Teddy 已經講完了，了解這三點之後，是很有幫助的。有時候 Teddy 甚至懷疑自己在採行 Scrum 的時候都有被 UP「附身」而渾然不知的情況。鄉民們仔細想一下，想不起來的 Teddy 再幫各位複習一下，其實 Scrum 這個框架可以很簡單的用下面幾點表示：

- Role：Product Owner, Scrum Master, Team (Developer)
- Activity：Sprint Planning Meeting, Daily Scrum, Sprint Review Meeting, Retrospective Meeting
- Artifact：Story, Sprint Backlog, Sprint Backlog, Task Board, Release Plan

那...對於要如何「開發軟體（分析、設計、實做、測試、佈署...等等）」Scrum 是沒有規範的。**往好的方面來講這樣是「很有彈性」，從壞的方面來講這樣是「太有彈性以至於不知道要怎麼進行」（怎麼評價端看你是汎藍還是汎綠的...XD）**。 所以，無論是採用哪種流程，基本的「分析與設計能力」還是要有的。不管你使用傳統的「結構化分析與設計」或是目前主流的「物件導向分析與設計」，在不同的 流程中都是必備的技能。關於這些分析與設計的技巧，在 Scrum 是不管你的（Scrum 內心獨白：給你彈性啊），在 UP 中就提的比較多。所以，這本書介紹完 Use-Case Driven、Architecture-Centric、Iterative and Incremental 這三個觀念之後，緊接著就介紹 UP 所謂的「core workflows」：

- Requirements Capture
- Capturing the Requirements as Use Cases
- Analysis
- Design
- Implementation
- Test

鄉民可以把這些介紹 「core workflows」的章節當作學習某種「分析與設計方法」。說真的雖然當年 Teddy 為了考資格考不得已將這些「core workflows」很認真的讀了好幾遍，但是到現在早就忘的差不

多了。耶，Teddy 還是活的好好地，所以應該....沒看也沒關係吧。Teddy 覺的這本書介紹的「core workflows」過於「正規」了一點，**光是為了搞懂這些「表示方法」所花的時間與在實做上所帶來的真正好處相比，C/P 值似乎太低了一點**。 如果要學分析設計方法，還不如直接去看 Crain Larman 的 Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development 這本書就好了（書名還真是長）。

像是書中介紹 Use Case Realization 這個概念，Teddy 曾經聽說有公司在面試工程師的時候還拿來當考題...說真的，現在 Teddy 也答不上來。不過再強調一次，**Use-Case Driven、Architecture-Centric、Iterative and Incremental** 這三句話一定要熟記，非常有用的觀念。Teddy 在採用 Scrum 開發專案的時候，其實內心裡一直默背這三句「口訣」，尤其是 Architecture-Centric 的精神如果可以掌握，對於避免「不要因為採用逐步成長的開發流程就讓軟體長壞掉」這件事佔有重要的地位（當然 refactoring 與自動化測試也是必須的）。

***

友藏內心獨白：不要以為 Teddy 只會 Scrum 喔... 啊，還是寫太晚了。

# 87 Contributing to eclipse: Principles, Patterns, and Plug-ins

May 19 22:29~23:33

12/13 17:27-17:39

原文發表於
http://teddy-chen-tw.blogspot.com/2011/05/contributing-to-eclipse-principles.html.

瞎忙了好幾的月，有好一陣子都沒時間靜下來好好讀本書，在可預見的下半年大概也好不到哪理去。沒讀新書就少了寫部落格的「料」，只好繼續把以前的「存款」拿出來花。找了一本以前唸書的時候所讀過的書，拿出來「~~微波~~ 冷飯熱炒」一下也還不錯吃啦。

今天要炒的是 *Contributing to eclipse: Principles, Patterns, and Plug-ins* 這一本。有長期注意 Teddy 行蹤的鄉民們應該看過 Teddy 提到這本書好幾次了，前一陣子 Teddy 去參加 Scrum 經驗分享活動，也特別向圍觀的鄉民們推薦這一本書。尤其是想要成為 architect 的鄉民們，此書更是不可遺漏的秘笈。

雖然是 2003 年出版的「舊書」，但是這本書內容之豐富，絕對算是經典之作。先看一下作者是誰，哇賽，Erich Gamma and Kent Beck，兩位都是大師，就算看不懂內容，就衝著這兩位的名氣都應該要買一本回家供著。Teddy 敢大聲的說，要不是當年開發 Eclipse plug-ins 的經驗，再加上看過這本書，Teddy 的 architecture design 能力大概要少掉 1/3 ~ 1/2。

如果鄉民們恰巧有寫過 Eclipse plug-ins 的經驗，就應該可以體會到 Eclipse 設計偉大之處。幾乎所有的人都公認「整合」是一件很困難的事情，而 Eclipse 厲害之處，就是能夠把三教九流的 ~~大~~ 軟體全部給他整合到 Eclipse 平台之中（吸星大法和北冥神功都沒這麼厲害）。這樣的設計，趕快把他給偷...學起來，自用送禮兩相宜。日後無論是開發什麼軟體，都一輩子受用無窮。

講了這麼多廢話，到底這本書在講些什麼？以下是 Teddy 認為 Eclipse 幾個最基本也最重要的概念：

- Plug-ins, extension, and extension point.
- Menu and Action.
- Marker and Marker Resolution.
- Builder and Nature.
- View and Editor.
- Workspace and Resource

- Java Core

為什麼這些概念重要... 嗯... 這樣說好了，一般的應用程式都會有 GUI，學會了 Eclipse 的 Menu、Action、View、Editor 觀念，日後開發應用程式（無論是 desktop or web 應用程式）都有一定的幫助。

所有的應用程式都會有 model 這一層，學會 View 和 Editor，也就了解到 MVC 的概念，透過 data provider 提供資料給 UI 這一層來顯示。Marker 就好像「狗皮膏藥」一樣，讓你自己開發的 plug-ins 可以依據某些規則在 UI 上標記記號，用以提醒某種狀況。例如，在 Java Editor 中發生語法錯誤時貼一個小標籤在 Java Editor 左方。而 Marker Resolution 就是用來幫助使用者（自動或半自動）排除這些狀況的作法。

Builder 機制讓使用者（程式設計師）可以自行開發並外掛「處理資料的程式」。例如，你發明了一個超棒的演算法，可以找出 Java source code 的那一行會發生 bug，哪麼你就可以自己寫一個 Builder，掛到 Eclipse 裡面，當使用者存檔的時候，讓 Eclipse 自動呼叫你所寫的 Builder 去分析與處理 Java source code。如果你的 Builder 有支援 Marker 的功能，就可以自動貼一個 marker 在可能出問題的那一行上面。至於 Nature 的功能是用來幫專案貼上某種標籤，如此 Eclipse 才會知道，當不同的專案被開啟的時候，要呼叫哪些 Builders。例如，如果你的專案具有 Java 的 Nature，Eclipse 就不會去呼叫 CDT。

更棒的是，Eclipse 不只是設計來支援 Java 程式開發，只要針對不同的 Resource 建構自己的 AST (Abstract Syntax Tree)，就可以在 Eclipse 上提供一個類似 Java 的開發環境。根據 Teddy 所知，有些 IC 設計公司，除了提供自己的 IC（硬體）之外，也在 Eclipse 上開發自己的 compiler、editor、與 debugger，讓客戶可以有一個好用的整合環境來開發特殊的程式。

***

寫到這裡可能會有鄉民說，我們公司又不是要賣開發工具，學 Eclipse 有什麼用？其實 Eclipse 架構可以讓開發者做很多事，不是只能用來開發「開發工具」（這句有點饒口）。有興趣的人可以去看一下 Eclipse RCP (Rich Client Platform)。

最後提醒一下，建議先學好 design patterns 之後再來看這本書，不然可能會不太容易懂。最好能邊看邊些一些簡單的 Eclipse Plug-ins 來當作練習，以收事半功倍之效。

***

友藏內心獨白：學弟們，你們說學長講得對不對啊？

# 88 Software Build Systems: Principles and Experience

July 12 21:05~22:12

12/16 18:57-19:04

原文發表於

http://teddy-chen-tw.blogspot.com/2011/07/software-build-systems-principles-and.html.

上禮拜六晚上在南陽街附近吃完「本格派」的親子丼之後，順道去天瓏逛一下。去年到美國出差買回來的書都還沒看完，原本也沒打算要買新書，想說瞄一下看看天 瓏最近有沒有進什麼新書。看著，看著，Teddy 的眼睛突然被某本書的書名「刺」了 一下。是何書如此大膽，居然敢行刺 Teddy，原來是...

*Software Build Systems: Principles and Experience*

***

Teddy 在「CI 之看來看去少了一點什麼 」提到 Continuous Integration 這本書寫得雖好，但是看來看去覺的少了一點什麼，少了什麼？就是少了「**你打算要開發一個（中大型）軟體，要如何將該軟體放到 CI**

**系統上？**」的答案。其實 Teddy 這樣說也並不公平，在 Continuous Integration 第 94 頁 **Build System Components Separately** 這一小節中其實有提到：

*Sometimes integration builds take a long time to execute because of the time it takes to integrate the source code and other associated files. In this case, you can break apart the software into smaller subsystems (modules) and build each of the subsystems separately.*

算是對於上述問題有稍微「點到為止」指點了一個方向，但是關於要如何「實做」卻沒有提供答案。而今天 Teddy 要介紹的這本書，算是鉅細靡遺的將如何建構一個 build system 這件事從頭到尾清清楚楚的交代一遍。有些書，就算是從頭到尾讀上好幾遍，都無法給它按一個讚。有些書，看完前言就已經是讚讚讚讚讚.....「連讚不絕」啦。Software Build Systems 就是屬於後者。

這本書到底好在哪裡...簡單歸納以下幾點：

- **可以當 build systems 的教科書**：這算是優點嗎？是滴，Teddy 覺的作者有點像是在寫「Journal papers」的精神來寫這本書，不但介紹了許多 build systems 領域的術語，並且用了許多「歸納」的方法來介紹 build systems domain knowledge。舉個例子，作者在第 xxviii 頁提到兩種建構大型軟體產品的

方法，分別是 **Monolithic builds** 與 **Component builds**。說真的，這兩種方法其實 Teddy 早就知道了，但是就是沒辦法像本書作者一樣可以明白的告訴你這兩種方法的名稱與作法。以上特點，讓 Teddy 可以把這本書當作寫 papers 的參考資料，真是太讚了...找你好久了....。

- **用很簡單的方法介紹 make file 的寫法**：話說 Teddy 活到這把年紀，對於 C/C++的 make file 裡面一堆奇怪的符號總是搞不懂。本書裡面的例子，連 Teddy 都看得懂，相信各位鄉民們也一定看得懂。
- **介紹五種不同的 build tools**：包含 MAKE、ANT、SCons、CMAKE、ECLIPSE (Teddy 內心獨白：這個也算？)。個別去了解這些工具是很花時間的，有人幫你準備的好好地，又是一個讚。對 Teddy 這種需要開發跨平台軟體的人而言，能夠看到同時介紹 MAKE 與 ANT 算是滿實用的（Teddy 內心獨白：你不一定要會寫 build scripts，但是至少要看的懂。不會寫詩，至少也要會吟啊。
- **Advanced Topics 與 Scaling Up 更是精彩**：本書最後兩個部份介紹 **Dependencies**、Building with Metadata、**Software Packaging and Installation**、Version Management、**Build Machines**、Tool Management、Reducing Complexity for End Users、**Managing Build Size**, **Faster Builds**。這些內容，對於深入了解 build systems 以及 CI 都是非常實用的資訊。尤其書

> 中還有提到對於開發跨平台軟體需要注意的地方，這一點更是符合 Teddy 的口味啊，真是 ~~好吃~~ 好看。

介紹到這邊，書的內容還是要請有興趣的鄉民們自行挖掘。這本書寫得很好，還滿容易閱讀的。想買的鄉民們手腳要快，有人看到這一篇可能明天就衝去買了，晚到可是會搶不到滴。

***

友藏內心獨白：如果幫忙推薦書也有錢可以賺那該多好...XD。

# 89 零與無限大

April 26 22:02~23:20

12:10 22:40-20:53

原文發表於 http://teddy-chen-tw.blogspot.com/2011/07/blog-post_26.html.

Teddy 還真是個烏鴉嘴，幾天前在「一步到位還是一槍斃命？」才提到大陸高鐵，沒想到馬上發生了後車撞前車的「特大」意外事故。不過有一事 Teddy 不明，電視新聞報導時一直用「**動車**」這個名詞，這個詞翻成台灣的習慣用語，倒底是「火車」,「電車」,還是「高鐵」呢？

***

幾天前在 Facebook 上面的「Scrum Community in Taiwan」 社群中，有人問到「什麼時候不適合施行 Scrum 或 Agile」,Teddy 給了一個很簡單的答案：

關於「什麼時候不適合施行 Scrum 或 Agile」答案其實很簡單：「看看自己的老闆，公司文化，團隊現況」答案就出來了。我想台灣 99% 以上都屬於「不適合施行 Scrum or Agile 的公司」。以下是 Teddy 的想法：如果今天有人跑去天安門廣場大喊「XX 黨」下台，或是去利

比亞首都大喊「XX 費」下台，下場如何可想而知。公司老闆不支持，團隊成員也不想了解，這都是常見的現象。除非想要導入的「那個人」跟老闆很熟（老闆的兒子？）或是想當烈士，否則還是不要導入為佳。

說實話，以台灣一般公司的文化與主事者的心態，說要去導入 Scrum 是極為困難的一件事。Teddy 曾經聽過一個故事，某公司的高高層對一位新進員工（高階主管）說：

不要把你在以前公司所做的那一套帶到我們公司來，我們公司有自己的文化與做事的方法，你自己要想辦法融入，不要試圖去改變。

Teddy 相信很多公司都是抱持著這種「排斥改變」的心態：**好的，我了解了，你所建議的這個方法看起來似乎不錯，but I don't care。謝謝不聯絡。**

***

這幾天 Teddy 在讀許文龍先生的「**零與無限大**」這本書，書中描述了許文龍先生經營事業的作法，其中有很多作法都與一般熟知的經營方式不同。也就是因為許文龍先生有著打破既有框架的勇氣，所以獲得了成功。舉幾個例子：

- 一般的公司與它的原料供應商之間，存在的都是利益衝突的對立關係。公司的採購想辦法殺價同時拼命的拿回扣，而原料供應商則不停的派 ~~辣妹~~ 業務來推銷產品。許文龍則是一次直接跟原料供應商簽訂 10 年的合約，至於價格則是由雙方來談（細節請參考 pp. 38-39）。總之就是想辦法將與供應商共享利益，把原本的對立關係轉變成**非對立**。
- 在公司創造一個「**找答案，不找責任**」的環境（p. 63）。Teddy 相信台灣絕大部分的公司，都是「**找羔羊（代罪羔羊），不找答案**」。
- 早在 1985 年，奇美公司就實施週休二日。為了彌補減少的工時，奇美投入千萬元更新自動化設備來因應（p. 68）。
- 經營，就是一種「**適應環境的行為**」...什麼事情都要到「**現場**」去講才準...在戰爭中，指揮官一定要到前線，才能看到地圖上看不到的東西（pp. 127-128）。
- 奇美可以說是一個「無文字的社會」...我們人一生的時間實在有限，用這些時間來做事，比較實在。人家問說，為什麼奇美一個人可以做那麼多事？我說：**因為他們不用寫報告，可以一直做事**（p. 132）。Teddy 內心獨白：好香好濃的 agile 味道...XD。

這本書有 399 頁，整本書的內容都十分精彩，有興趣的鄉民們可以去找一本來看。

***

許文龍是受日本教育長大的，他在書中說他曾經讀過一本日本人翻譯的「**生態學**」的書，受這本書的影響很大。因此他的很多思考模式，是從「整個生態」的角度來看，格局也就變得比較大。Teddy 讀完本書之後，倒是看到很多 TPS（Toyota Production System），Lean 與 Agile 精神。天下的事，一事通，萬事通，也許這就是這個道理吧。

回到一開始的那個問題：「什麼時候不適合施行 Scrum 或 Agile」？Teddy 說，**什麼時候都不適合，什麼時候也都適合**。

***

友藏內心獨白：結尾那一句一定要搞的那麼玄嗎？

**備註**

1.
2.

## 参考資料

[1]

[2]

# 第十部 閒扯蛋

# 90 Teddy 的初衷

06/18 21:55~23:00

12/12 15:48-16:00

原文發表於 http://teddy-chen-tw.blogspot.com/2010/06/teddy.html.

Teddy 在 2008 年秋天畢業前夕，在 104 上登錄了履歷，收到四家公司的面試通知。這個故事就發生在 Teddy 到其中某家本土「純軟體公司」面試的時候。該公司主要業務為開發自有品牌的軟體（搭配硬體），在國內還做的不錯，頗有知名度。面試 Teddy 的人是該公司技術總監還有董事長。在面試的過程中，技術總監突然說：「Teddy 你的自傳寫得太好了，我一定要把這幾句念一下」。

有人曾問我為何想念博士班，我總是回答：**我希望改變人們在台灣開發軟體的方法**。因為了解到許多台灣的公司還是用非常傳統的方式在開發軟體，對於教科書上所談的軟體工程方法，以及許多在國外早已被證實非常有用的軟體開發實務作法，台灣的老闆、專案經理、或是軟體工程師總是能找出各種理由與藉口拒絕採用。我認為軟體開發應該是一件很愉快且有趣的工作與創作，因此在我求學與研究的這幾年，我試著將理論與實務加以結合。…

沒想到真的有人在看面試者的自傳...可是，搞到**當場朗讀**的地步，這位技術總監也算是**古今天下第一人吧**。當然 Teddy 當下有點小小的感動。什麼，你問面試結果如何，這...請參考古早某篇部落格文章。

***

Teddy 不是要炫耀自傳寫作，而是回想起 Teddy 唸博速班的這 N 年，人也老了、體力也衰退了、胃潰瘍也得了，在即將變成**「菜尾」**之前，回想起原始的**「初衷」**（PS：幾個月前看了「孫大偉的菜尾與初衷」這本書，寫得不錯值得一看）到底是什麼？是否還存在？

10 幾年前 Teddy 剛出社會，也做了不少專案，有的成功，但是失敗的好像更多。**明明每天做的累的跟狗一樣，有時候晚上還要煮稀飯、綠豆湯當宵夜跟同事一起分享，但是需求好像永遠都沒有「定下來」的一天，要把程式「寫完」好像是「煮沸海水」一樣的困難。**

啊，是 Teddy 軟工的書讀得不夠多吧。Teddy 五專是念電子科，因為軟體太強以至於硬體爛得一塌糊塗...XD。但是當時在學校所學的軟體課程，也只有 Pascal、Fortran、C/C++這些程式設計課程（不要提組合語言和 8051...我恨你們...XD），以及系統程式、作業系統、演算法而已。所以，只好在工作中不斷的看書。但是畢竟是沒有受過

「正統訓練」，經過幾年後，雖然累積了一些工作經驗，但是又經常會懷疑：「**軟體真的是這樣開發的嗎？**」Use Cases 拼命寫，UML diagrams 卯起來畫，好像挺沈重的。團隊成員明明只有小貓五、六隻，真的要搞到 RUP （Rational Unified Process）？可是書上都這樣寫啊，人家國外大廠都在用了，一定沒錯的啦。

有經驗，自己也看了許多書，不過還是缺少**獨立判斷的能力**。後來因緣際會之下， 不小心唸了博速班，在指導教授強力掃把助陣之下，勉強讓 Teddy 在 N 年後爬出校門。Teddy 要再強調，「博速」只不過是多唸了幾年書而已，真的沒什麼了不起的。Teddy 的指導教授曾經告訴 Teddy，**要有工專人的「黑手」精神**，Teddy 一直牢記在心，奉行不渝。

***

Teddy 沒事就在部落格上跟瘋狗一樣亂罵台灣的軟體開發現況，光是罵當然很簡單，但是也不能完全像 Teddy 的學弟 Lililala2 所說的：

不過當你處於一個快散掉的團隊(快沉的船)時,只需要確定兩件事:

1.確定自己有脫離火箭

2.確定點燃了脫離火箭後有地方可以著陸

其他什麼搶救團隊(搶救沉船)的事都是徒勞,多幹幾次之後就會想去賣雞排(或保險)

雖然 Teddy 有時候真想學日本電視節目「自給自足過生活」一樣躲起來（誰可以賞 Teddy 一枚脫離火箭加一塊農地啊），但是回想起當年還是年輕小伙子的「初衷」：

**我希望改變人們在台灣開發軟體的方法**

當年不知道這個方法應該是什麼，隨著年紀增長，輪廓漸漸清晰。

**Scrum + Lean + XP**

Teddy 的指導教授常常告訴修軟工課程的學生，**要「傻的願意相信書本裡所說的」，不要在尚未嘗試之前就先否定**。嘿，Teddy 也曾經被那個號稱「軟工界超級整人遊戲之 PSP (Personal Software Process)」搞得神經快錯亂，也是挺過來了。

覺的當前的方式不好，想辦法從小處著手，慢慢改變，總是會看到成果的。

***

友藏內心獨白：眼睛快瞎了，這種「勵志小品」挺噁爛的，不符合搞笑宗旨。

# 91 幸福有感

August 19 23:01~ August 20 00:0

12/10 22:23-22:27

上個禮拜五早上起床之後，「腰」覺的痛痛的，本來想說週末休息一下就好，結果到禮拜一還是沒有改善，只好在 Scrum Planning Meeting 之後下午請假去看復健科門診。到了醫院，復健科的「老師（復健治療師）」看到 Teddy 就說了一聲：「好久不見喔」。是啊，自從上次「二度」治療脖子痛到現在也已經過了 10 個月了。在醫院，就好像一個犯人出獄一樣，最不想說的兩個字，就是「**再見**」。雖然每個月繳了不少的健保費，但是沒事還是不要使用為妙。

本以為醫生會叫 Teddy 去照 X 光，結果並沒有。聽完 Teddy 的描述之後，醫生說先復健（深層熱療和電療），如果下次門診（就是今天啦）還沒改善再照 X 光。這可能是一個好徵兆，因為之前 Teddy 去看脖子痛時，第一次門診就去照了 X 光，結果是頸椎有問題，壓到神經。應該是長期姿勢不良加上電腦用太多了，可能低頭看書也有影響。結果一週復健六天，連續復健了 3 個月。那時候，Teddy 真的了解「小王」的辛酸，復健真的好累啊。而且聽復健治療師說，很多病人的問題都無法根治（有些是老化的自然現象，有些是因為沒有耐心持續復健下去），待症狀減緩之後，就停止復健，過一陣子問

題復發，又乖乖回來報到。所以在復健科常常可以見到「回鍋的熟客」。Teddy 脖子的問題，也是沒有根治，前後治療了兩輪，到後來只能靠減少一些電腦的使用以及注意看書的姿勢來避免復發。

Teddy 腰痛的問題，經過五次的復健，症狀從原本的「疼痛」，變成「酸」，已經算是有改善，醫生說要繼續復健。希望這次能夠在一個月之內痊癒。

***

前幾天腰痛的厲害，Teddy 不太敢用電腦，自然也沒寫部落格了。這一，兩天狀況稍微好點，Teddy 突然有所感慨。約五年多前，Teddy 身體狀況開始不太好，一開始是胃潰瘍，到現在只要喝含有咖啡因的飲料或是可樂，養樂多，胃就會不舒服。所以 Teddy 只好忍痛把喝咖啡的習慣給戒掉了。約兩年前脖子痛，Teddy 就慢慢減少寫程式的數量，偶爾手癢忍不住卯起來連續寫了幾天之後，脖子就會不太舒服，所以現在寫程式都要提醒自己「寫慢一點」。最近腰痛期間，最嚴重的時候連從椅子上面站起來腰都無法打直；**沒想到一個這麼簡單的動作，此時卻是如此的困難啊。**

古人說「能吃能睡就是福」，現在 Teddy 才深深體會到這個道理。對大多數的人來說，喝一杯熱熱的拿鐵咖啡，扭扭腰，擺擺頭，是多

麼簡單與自然的事，沒什麼了不起。對 Teddy 而言，能做到這幾件事，就是幸福。前幾天在雜誌上面看到「伊甸基金會」為「發展遲緩兒」募款的廣告，上面登了一張照片，照片中是一個坐在輪椅上的小女孩，她讓 Teddy 想起之前在復健科看到的另一個小女孩，因為某種原因不良於行，要靠復健來練習走路。對大多數的人來說，走路是多麼簡單與自然的事，對小女孩來說，能正常走路是幸福。

什麼是幸福？健康就是幸福，正常就是幸福，簡單就是幸福，吃的飽穿得暖就是幸福，做自己喜歡的事就是幸福，有能力助人就是幸福。錢，夠用就好。

路人甲：問題是我的錢永遠都不夠用啊。

***

友藏內心獨白：現在才體會到，「靠腰」真的很重要。

# 92 一步到位還是一槍斃命？

April 22 23:46~23:24

12/10 22:33-22:39

原文發表於 http://teddy-chen-tw.blogspot.com/2011/07/blog-post_22.html.

這幾年中國大陸的高鐵建設突飛猛進，從南到北蓋了有幾千公里的高鐵。從電視新聞的畫面看起來，感覺挺不錯的，平穩的車廂，高檔次的服務人員，以及還算合理的票價。但是，鄉親啊，電視是「聞不到味道滴」，此話怎說？

Teddy的一位友人最近剛從大陸回台灣，友人從大陸南方搭高鐵到武漢，再從武漢搭到上海，也算是繞了大半圈的大陸。據友人表示，大陸高鐵雖好，但有一個最大的缺點：「**煙味太重**」。

Teddy問：什麼，~~有人在高鐵上拜拜~~？ 大陸高鐵上可以抽煙啊？

友人說：不行，但是很多人跑到廁所抽煙。雖然車上一直廣播說不要在廁所抽煙，但是根本沒人理會。我搭的還是頭等車廂，如果是次一等的車廂，那味道可更多了。 在武漢的時候，還有人提著「鮮魚」上車，整個車廂搞得都是魚腥味。

Teddy內心獨白：經常在早上搭307公車經過果菜市場的人，就能夠體會到整車都是「新鮮食材」味道的那種感覺了。

硬體可以幾內年建設完畢，人的習慣...終生難改。這一代是不可能了，看看下一代會不會長進一點。有辦法把人的水準搞到一步到位Teddy就佩服你。

***

今天看到一則新聞，說台灣某大電腦公司斥資近百億元台幣併購位於外國的某雲端軟體公司，報導中說：

**...評估雖然可讓該公司在過去幾無著墨的雲端運算領域「一步到位」...**

看了這則報導怎麼好像有種看到大陸高鐵的感覺...

***

友藏內心獨白：**別鐵齒，誰說不會有人在「雲端上抽煙」**。

# 93 聞過則喜...誰說的？

July 17 15:21~16:06
19:28~20:19

12/10 20:56-23:14

原文發表於 http://teddy-chen-tw.blogspot.com/2011/07/blog-post_17.html.

最近罵人罵得太頻繁，但由於考慮到「人情世故」以及怕走在路上被「蓋布袋」，因此罵的方式很間接，覺的不太過癮。此時 Teddy 突然想起了一則親身經歷的小故事。

話說 2004 年九月 Teddy 到美國參加 PLoP 2004（Pattern Languages of Programs）研討會，這是 Teddy 第一次出國參加研討會，雖然有 Kay 同行陪伴，但是 Teddy 心裡還是滿緊張的。這個 PLoP 研討會鄉民們可能不太清楚，一般的研討會，輪到你報告論文的時間，大概也就是 15-20 分鐘。報告完畢之後，以台灣人這麼會善用時間的特性，一定是利用機會到當地去探訪一下風土民情，順便做一下國民外交（以上翻成白話文就是：到處觀光）。但是 PLoP 研討會是採用所謂的 **writers' workshop** 形式舉辦，有以下幾個特點：

- 投稿 papers 的作者，會被分成數個約 4-6 人的小組，研討會就是以小組討論的形式進行。
- 所有作者在參加研討會之前，就會得知自己被分配到那一個小

組，並知道小組中其它成員。

- 在出發參加研討會之前，要讀過自己那一個小組其他作者的論文（原因等一下就知道了）。
- 每次討論的 session，以 Teddy 當年的例子，長達 75 分鐘。在這個 session 中，只討論某一位作者的 paper。
- 討論會開始，由論文作者先念一小段自己論文裡面最喜歡的段落。之後作者就成為「**牆上的蒼蠅**（不可以出聲不然會被人類打死…XD）」，退居幕後聽其他的人如何評價你的論文。在這期間原作者不可以講話。
- 此時輪到其他人上場，大家先一起發表自己覺的該論文寫得好的部份，然後在說出覺的論文中需要改進的部份。

當年和 Teddy 同組的除了 Teddy 以外還有四個人，其中三位（兩男一女）都是美國大學的教授，另外一位男士是英國人。這位英國人後來將他的論文整理之後出了一本厚死人不償命的書：xUnit Test Patterns: Refactoring Test Code。現在想起來還有點好笑，因為當時有一位美國人還覺這位英國人「有些英文句子寫的有待改進」。

***

接下來的才是重點，這三位教授中，其中有一位就是寫 GoF Design Patterns 的第三作者 Ralph Johnson，此人就算是稱不上「大師」，但

也算是有名的大牌教授。這個故事就是發生在他的身上。

就在某次輪到要討論 Johnson 的論文時，我們這個小組突然多出了兩位圍觀的鄉民，在此以鄉民甲，鄉民乙稱之。鄉民甲看起來是西裔美國人，鄉民乙則是白人。在會議進行中，這位鄉民甲感覺起來就是來拍馬屁的，一直稱讚 Johnson 論文（但不知其動機為何）。重點在於這位鄉民乙，因為這位年輕人是來**踢館**的。鄉民乙舉出了幾點 Johnson 論文裡面的錯誤（Java 程式碼的語法錯誤以及一、兩個設計上的 issues），當場氣忿弄的有點僵。好在其他兩位教授以及那位英國人幫忙出來打圓場，說是「程式碼放到 word 排版很容易出錯」就這樣交代過去。雖然鄉民乙還想要繼續追打下去，但是畢竟勢單力薄，最後也只能見好就收。

那個 session 結束之後，Teddy 和 Kay 看到 Johnson 急急忙忙的跑去打電話，Teddy 猜想 Johnson 應該是打電話去罵他的學生吧...呵呵呵...這篇論文應該是他學生幫忙寫或是排版的吧...。

***

Teddy 很是佩服鄉民乙的行為，因為鄉民乙所舉出 Johnson 論文中的幾點錯誤，的確都是「**事實**」。就算「只是排版錯誤」，但對於學術論文而言，也是很丟臉的事情，不是小學生寫錯字回家罰寫三遍就

可以交代過去的。這種事要是發生在台灣...錯，假設不成立，根本不可能發生，台灣地狹人稠，大家動不動就說「**相遇得到（修都 A 丟）**」，誰敢那麼白目去揭穿「國王新衣」的秘密。

古人說，聞過則喜，但時代不同了，鄉民們不要傻傻相信。現在台灣普遍的現象則是：

- 官大學問大：反正當官的只要用一句「這是經過通盤考量之後的決定」就可以把所有的專業判斷全部打死。「**通盤考量**」翻成白話文就是說「老子就是要這樣幹，抗議無效」，是不是這個意思？就是這個意思....。
- 速食主義：凡事求快，要用最低成本達到短期目的。所以，新聞不必講求事實，只要報導那一家店在打折，那裡又推出新口味的火鍋，誰家的狗會唱歌，誰家的小強跌斷了腿又自行長了出來。其他內容只要把每天報紙上所寫的念一遍，加上從網路「翻拍」畫面加上配音說明，這樣就差不多了。**難道台灣的新聞系上課都在教這些？**
- 共犯結構：請自行發揮...
- 有關係就沒關係，沒關係就有關係。
- 為反對而反對：反正只要是非我族類，不管做的好不好，對不對，先罵了再說。可憐的楊大砲，沒事寫什麼書，最後落得一堆人想去告你。大陸有句順口溜：**不到北京不知道官小，不到**

**上海不知道錢少，不到海南島不知道自己身體不好，不到台灣不知道文革還在搞**。最後這句還挺貼切的。

寫到這邊有點離題了，不過反正本部落格一向以離題著稱...XD。昨天看了部電影，叫做「**我的名字叫可汗（My Name Is Khan）**」，片中男主角的媽媽告訴他，世界上只有兩種人，做好事的好人，和做壞事的壞人。不要分什麼回教徒，印度教徒，基督教徒，或是白人，黑人，本省人，外省人，台灣人，中國人。說得很好，但是有一個根本的問題電影中沒交代「**誰來決定誰是好人，誰是壞人？**」黑道漂白的例子可是屢見不鮮，好人做壞事的也是時有所聞。

***

看到有人用奇怪的方式去教導 Scrum，本和 Teddy 無關，反正害到別人也不會害到 Teddy。但是回頭想一想，如果悶不吭聲，Teddy 也很可能在其他自己不熟的領域，被其他人害到。台灣人這種「息事寧人」，「多一事不如少一事」的心態，要改過來還真不容易。連 Teddy 也變得有點「俗啊」，不敢指名道姓的點出有問題的文章（Teddy 又不是李大師）。

布袋戲有一句很有名的台詞「互向漏氣求進步」，和聞過則喜的意義差不多。在現今社會，能做到的又有幾人？

***

友藏內心獨白：年紀越大，膽子越小；薪水越高，膽子越小。此為正常現象，請安心度日。

# 94 白飯一碗

June 02 21:29~22:16

12/10 23:16-23:30

原文發表於 http://teddy-chen-tw.blogspot.com/2011/06/blog-post.html.

先聲明一下，本日主題與軟工完全無關（路人甲：YA....）。

今天晚上 Teddy 和 Kay 去住家附近的圖書館準備借一本預約的書，走到圖書館樓下才發現今天公休，於是直接到下一站：圖書館對面的「XX 起司」吃晚餐。這家店 Tedd 常來，幾乎是只要到圖書館借書都會來這裡用餐，價錢介於 89 ~ 160 之間，除了主餐以外，還有土司麵包吃到 ~~吐~~ 飽外加紅茶飲料，還有冷氣可以吹，算是滿平價又還乾淨的用餐環境。

就在 Teddy 剛點好晚餐的時候，隔壁桌來了一對父女，爸爸的腿行動不便，有點一拐一拐的，手也不是很靈活。女兒看起來大概是小三（小學三年級，別想歪了），天真可愛，看起來十分開心的樣子。他們只點了一份 89 元的咖哩飯，外加小朋友不點主餐的最低消費 30 元（麵包與飲料吃到飽的費用）。爸爸點餐的時候，向店員提出一個要求：

爸爸：我還要加一碗白飯。

店員：一碗白飯要 10 塊錢喔。

爸爸：以前不是都不用錢嗎？

店員：對啊，可是現在加一碗白飯要 10 塊錢。

***

爸爸跟店員盧了很久，還是不成功，臉上不禁露出失望的表情，但是他還是不放棄。就在老闆娘經過的時候....

爸爸：以前加白飯不是不用錢嗎？

老闆娘：不是不用錢，上次是因為你說小朋友吃不夠，加一小口，所以不用錢。我們加白飯是要收 10 塊錢的。

爸爸：可是小朋友吃不飽啊.....[喃喃自語]

老闆娘：現在物價一直上漲，米和麵包都漲價。像是米上個月就漲了三次。很對不起啦，沒辦法免費加白飯。

老闆娘：那你還要加白飯嗎？

爸爸：好吧....

***

此時女兒很高興的幫忙倒飲料回來，正準備去拿麵包。

女兒：拔拔，我去拿麵包。

爸爸：好，多拿一點.....

***

過一會咖哩飯和白飯送上來了，只見這位爸爸用湯匙舀了一些咖哩到那碗白飯上面，沒兩下子就把這碗白飯...應該說，淋了咖哩的白飯，給吃光光了。而一旁的小女孩，似乎什麼也沒察覺，還很高興的吃著她的咖哩飯，只是很天真的邊吃邊說「咖哩好辣喔」。

***

說真的，Teddy 很想偷偷跑去跟老闆娘講，以後這對父女不管加幾碗飯都算 Teddy 的。內心掙扎了好一會最後還是放棄（路人甲：你這個俗啊...XD）。Teddy 覺得很感慨，有的人努力爭取，就為了想吃一碗免費的白飯，而 Teddy 雖然是一位「涉世未深，口袋也不深的軟體從業人員」，但畢竟生活還算過得去。有時候生活上，工作上，為了一點小事就經常忿忿不平。想想別人，其實自己已經算是很幸福

了。

各位鄉民們，當你覺的有苦難申或是受到委屈的時候，就去吃碗白飯吧...記得要淋上咖哩...XD。

***

友藏內心獨白：切，1000 塊的路考保證班費用都可以吃 100 碗白飯了....

# 95 秀才遇到兵

Feb. 19 22:20~ Feb. 20 00:08

12/11 16:45-16:54

原文發表於 http://teddy-chen-tw.blogspot.com/2011/02/blog-post_19.html.

Teddy 發現這幾年陸陸續續所寫的近二百篇文章中，點閱率比較高的像是「600 多個 bugs 要怎麼修？」，「老闆，軟體不是這樣開發滴」以及「改行寫網路小說算了」，都有一個共通點，就是隱含著一種「**秀才遇到兵，有理說不清**」的無奈。相信這種感覺也是許多鄉民們內心小小的吶吼，所以才比較容易引起迴響。至於那些太技術性的文章，就當作 Teddy 自己寫爽的，充充業績之用。

秀才，當然就是咱們這些頑劣不冥的 ~~玫城屍~~ 工程師們，至於兵的「可能性」就很多了：

- **聖上**：豈只是「兵」，還且還是「天.....天都準時上班的...大兵」，一句話就可以讓你消失在地球表面，殺傷力 100%。
- **九品芝麻官**：天高皇帝遠，聖上殺傷力雖大，但總不會天天都給你一道聖旨，但九品芝麻官（直屬主管）就不同了。運氣好遇到那種每天盯死你的，讓你一想到要上班就憂鬱起來。遇到勤政愛民的九品芝麻官也就算了，但絕大部分都是那種善於揣摩上意，逢迎拍馬，找代罪羔羊的貨色。九品芝

麻官在「大清帝國」龐大的官僚體系中多官位雖小，但絕不可小覷。就好比一把只要三塊錢新台幣的「超級小刀」，每天只要被刺一刀縱使你有再多的血也不夠流啊。

- **PM**：俗話說「好的 PM 帶你上天堂，不好的 PM 讓你虛工做不完」。有些 PM 好像上班的時候忘了把大腦帶出們（**Teddy 內心獨白：沒有的東西要叫他去那裡生啊？**），遇到問題只會「轉寄 email」。把客戶的 email 轉給工程師，或是把工程師對於需求的問題直接轉給客戶，似乎已經變成這種 PM 的反射動作（還記得嗎，反射動作是不需要經過大腦思考滴）。如果 email「轉得好」也就算了，有時候把該給 testers 的 email 要求 programmers 去做，掀起一場無謂的「內鬥」，有這種 PM 公司還需要敵人嗎？殺傷力以「三國人物」來看的話，也算是「張飛」這種等級了。
- **大牌業務**：仗著「全公司都是靠我在養」的氣勢，大牌業務要你們這些手無縛雞之力的「秀才」作詩，你就得在「七步之內」吟出一首五言絕句（Teddy 內心獨白：我又不是曹植），叫你畫畫你就得在三分鐘內畫出一幅「春樹秋霜圖」（Teddy 內心獨白：先把秋香拿出來人家才要畫啦...XD）。要小心這些大牌業務都有「密奏之權」，可直達天聽，要是你好膽拒絕他們的要求，下場可是會「暗箭」穿心。
- **同事**：雖然「理論上」秀才的同事應該也是秀才啊，可是有時候你的秀才同事可能會「投筆從戎」，一不小心也變成小兵

了。

- **顧客**：出錢的是大爺，遇到這種「兵」只能當作「清冰」... 加減吃... 就算現在外面溫度只有攝氏 10 度加上細雨綿綿你還是得把這碗「御賜清冰」含著眼淚帶著微笑把它給「吞下去」。

但是，從「兵」的角度來看，這些「秀才」才是「不知天高地厚」的人。「兵」叫你作什你就做什麼，那來的那麼多意見。這個也不做，那個也不行，那你俸祿還要不要領？（兵內心獨白：大清國的國庫只剩下四百萬兩的壓庫銀了...）

***

有一次 Teddy 在和指導教授在討論某個研究題目的時候，他曾經說過類似的話（細節 Teddy 已經忘了，大意如下）：

美國（西方）雖然很強調科學，但是老美內心其實有一種反科學（瞧不起科學）的意識。你看很多老美的電影裡面，壞人在一個高科技的實驗室內正要做出危害全人類的勾當，此時類似「藍波」這種「頭好壯壯」的肌肉猛男在最後一刻及時趕到打倒了壞人。壞人警告肌肉猛男，千萬不要隨便亂碰任何按鈕，否則就會立刻爆炸。眼看剩下倒數 10 秒鐘壞人所設定的炸彈就要啟動了，「頭好壯壯」的肌肉猛男完全不鳥壞人的警告，二話不說，拿起手上的衝鋒槍朝這些「高

科技設備」一陣掃射.....沒事，狀況解除，肌肉猛男再次拯救世界。

***

如果把「肌肉猛男」換成「秀才」那就完蛋了，剩下 10 秒哪可能讓「秀才 」在那邊慢慢研究解除炸彈的密碼。有時候 Teddy 也會懷疑，學那麼多有的沒的真的有用嗎？也許真實世界需要的是「肌肉猛男」而不是「秀才」。

***

友藏內心獨白：這一篇只有有緣人才看的懂。

# 96 ISO 大戰乖乖

06/30 22:11~22:48

12/12 21:34-09:39

原文發表於 http://teddy-chen-tw.blogspot.com/2010/06/iso.html.

Teddy 今天從同事 A 先生那裡聽到一個真人真事的笑話。話說 A 先生在之前服務的公司有導入 ISO 的經驗，有一次某位來自香港的 ISO 稽核員到該公司的機房查核，看到裡面擺了一包「乖乖」。

稽核員：　機房裡面不是不可以有飲料和食物嗎，怎麼會有「乖乖」？

A 先生：　這不是拿來吃的。

稽核員：　那這是？

A 先生：　放「乖乖」可以保佑我們的伺服器電腦不出問題。

稽核員：　現在都科學時代了，還這麼迷信。

A 先生：　這是我們的「信仰」，請尊重。而且賣我們機器的供應商也建議這麼做。

稽核員：　遇到火星人（這一句沒說出口）。

稽核員：　就算你們不吃，食物還是會引來老鼠，危害資訊設備。

A 先生：　我們機房有防鼠設計，縫隙都用膠條密封起來，不會引來老鼠。

稽核員：　如果有員工偷吃呢？

A 先生：　我們機房有門禁和 24 小時錄影，不會有人在裡面偷吃東西。

就這樣一來一往過了 30 分鐘。

| | |
|---|---|
| 稽核員： | 好吧，我就不把這點記錄為主要缺失，只記為觀察項目（我累了…）。 |
| A 先生： | YES! (喊在心裡) 。 |

事後 A 先生把乖乖放在一個透明密封的盒子中，並在前方擺上一張「~~請勿餵食~~ 請勿食用」的告示牌，還有在文件中紀錄要定期更換乖乖等注意事項。

Teddy 問 A 先生，怎麼不把乖乖拿出來就好了，幹麼這個麻煩？基於兩個原因 A 先生說不行：

- 萬一拿出來之後真的出事，誰要負責？（那是銀行的機房，伺服器如果當機事情很大條）。
- 如果拿出來就承認這是一個「主要缺失」，事後的矯正措施要寫一堆文件，加上後續改善追蹤，很麻煩，當然要硬幺下去。

結論：一字曰之「猛」。

***

友藏內心獨白：原來 A 先生的 ~~嘴砲功~~ 幽默感是這樣訓練出來的，佩服，佩服。

# 97 一萬個小時的練習

Nov. 18 21:24~22:18

12/18 20:24-20:30

原文發表於 http://teddy-chen-tw.blogspot.com/2010/11/blog-post_18.html.

曾經有一陣子 Teddy 受人之託，積極的招募學弟~~妹~~們念博速班，但是卻都以失敗收場。分析其原因，可能有以下三點：

- Teddy 一念就念了 N 年才畢業，學弟們被嚇到了。
- 念完博士要那麼多年，投資報酬率不高，還不如早點投入職場。
- 裝孝帷，我都恨不得效法二分之一個秦始皇（焚書不坑儒），還叫我念。

博速班原本就是「有緣人」才可以念滴，強求不來。再說，念那麼多書有什麼好處，為何把人生最精華的 25-35 歲浪費在學校？

除了那一張薄的幾乎忘了它的存在的畢業證書以外，要如何具體且簡短的說明念博速班的優點，除了「興趣」二字以外，還真是不太容易。Teddy 想起去年天下雜誌的一篇文章「異數，作家葛拉威爾：創意來自一萬個小時的練習」，覺得這個葛拉威爾（Malcolm Gladwell）講得還真好。

該書的重點，一言以蔽之：**「不管哪一種專業，成功最大前提，都需要有一萬個小時的不斷練習。」**

假設一個博速生一天工作（上課、讀書、讀論文、找資料、討論、聽演講、做實驗、寫程式等等）6 小時，一年工作 300 天，那麼：

10000 / 6 / 300 = 5.56 年可以畢業，

如果拼一點，每天工作 10 小時，一年工作 300 天，那麼：

10000 / 10 / 300 = 3.3 年可以畢業 。

其實還滿準的耶。所以 Teddy 可以說，念博速班就是給自己一萬個小時的練習，從素人變成達人的一種訓練。Teddy 當年剛念博速班時，指導教授就說過「念個博速起碼要念個 200-400 篇論文」... 至於書就不用說了，多多益善。

路人甲： 現在網路那麼發達，任何問題 google 一下不就有答案了？！

Teddy： 講是這樣講，但是實際上同樣的「工具」不同的人來使用，就是有不同的效果。更不用提「速度」因素了。

> 在一萬個小時的練習之後，理解很多事情的能力都變強了，有點類似武俠小說所說得「有了九陽神功護體，學什麼都快」。

除了學東西快這個能力以外，決策的速度與品質也提昇了。做軟體的人都知道，舉凡從變數命名，程式對齊這種「小事」，到專案管理，資源安排，時程預估，需求分析，架構設計，唬爛老闆，找人吵架等等「大事」，都必需要在很短的時間內做出「相對合宜」的決斷，否則便會吃虧。**所以，學習與決策，速度與品質的提昇了，這些才是真正寶貴的地方**。

當然，這一萬小時的訓練也可在「職場」中進行，早點出社會，早點磨練也是不錯滴。但是要小心，確定自己有「磨練」到，而不是每天腦袋空空的卯起來加班，那就虧大了。

結論：一萬小時減掉你在某個領域已經持續練習的時間，就是你變成達人所需的時數。

***

友藏內心獨白：彷彿聽到某人說...哼，小 case，拎杯玩 game 早就超過一萬小時。

# 98 小朋友不可以說謊喔

August 24 20:36~21:14
Pacific Time (US & Canada).

12/19 11:13-11:21

原文發表於 http://teddy-chen-tw.blogspot.com/2010/08/blog-post_24.html.

常常會聽大人對小孩子說「小朋友不可以說謊喔」，以前 Teddy 不覺的這句話有什麼了不起的，最近 Teddy 赫然體會到這句話的真諦。這句話了不起的地方在於加了**「小朋友」**這三個字，限定了不可以說謊這個要求只適用於「小朋友」。至於大人呢？「善意的謊言」每天都在說。

老闆：　什麼時候才能讓第一個客戶使用我們的產品？

某高層：　三個月。

工程師：　（內心獨白）我怎麼不記得我曾經開發過任何可以用的東西？。

***

某高層：　我們的宇宙無敵，世界無雙之超級雲端儲存系統已經有客戶在試用了。

老闆：　　很好，很好。

工程師：　（內心獨白）你是說上個禮拜進貨的那 50 台硬碟？

***

老闆：　　產品要準備正式上市了，那我們的 beta 客戶反應如何？

某高層：　客戶很喜歡我們的產品。

工程師：　（內心獨白）我們的客戶是靈界人事嗎，怎麼不記得有人用過？

***

這個世界就是這樣，有人敢說，還真的就有人信耶。這種行徑跟詐騙集團好像沒什麼兩樣。

當年老蔣總統曾經說過：「一年準備，兩年反攻，三年掃蕩，五年成功」。還好老蔣總統已經不是小朋友了，講這樣激勵人心的話是很 OK 的。

***

有藏內心獨白：Teddy 的幽默感還在調時差，沒什麼搞笑的 fu。

# 99 需求分析書中最重要的資訊是什麼？

Sept. 21 23:04~ Sept. 22 00:14

12/22 17:01-

原文發表於 http://teddy-chen-tw.blogspot.com/2010/09/blog-post_21.html.

Teddy 這次到美國出差，利用 Amazon 在美國國內買書不用運費的優惠，一不小心買了 10 本書（標準的貪小便宜心態），等要回台灣打包行李時才發現，這還真有點快放不下（要不是幫剛做完月子沒多久的同事帶了六個奶瓶回來，可能會買更多書... XD）。不過這不是重點，今天的主題是 Teddy 要介紹一本此行出差所買的書：*Bridging the Communication Cap: Specification by example and agile acceptance testing*。

當初會買這本書是因為被書名副標題「Specification by example and agile acceptance testing」所吸引。記得當年 Teddy 還在念博速班的時候，看過 Phillip G. Armour 所寫的一篇論文，叫做 *The Case for a New Business Model: Is software a product or a medium?*, Communications of the ACM, pp. 19-22, August. 2000。這篇論文提到有五種儲存知識的媒介：

- DNA
- Brains
- Hardware
- Books
- Software

論文細節就容 Teddy 偷個懶，請鄉民們自行發掘。當年 Teddy 看完的感想是，一般的軟體需求都是以「文件」形式存在，屬於上述第四種知識儲存媒介（book）。但是軟體開發的最終產品卻是以第五種知識儲存媒介(software)存在(這算是所謂的「阻抗不匹配嗎？」)。所以，問題來了，軟體開發人員就需要確保這兩種不同的知識儲存媒介（想表達相同的事情---軟體功能或需求）是否同步（軟體工程裡面所謂的 traceability）。相信大家都知道軟體開發人員都很忙，沒有那個美國時間去更新需求文件，因此需求文件所記載的內容與程式碼實際完成的功能經常 有所出入也是很「正常」的現象。所以，到底要相信文件還是要相信程式碼，便成為許多軟體開發人員心中的痛（鄉民甲：其實... 兩者都不可信...XD）。

所以，Teddy 當時就在想，如果能夠將軟體需以 software 的形式表達，那麼需求與產品都是以第五種知識儲存媒介（software）來紀錄，是不是就可以減少「阻抗不匹配」的問題？此外，由於軟體具有「可執行」的這個特性，因此就有可能自動驗證「需求」與「軟

體系統」是否「同步」。

講了這麼多，還沒轉台的鄉民們，再看一次這本書的副標題：「Specification by example and agile acceptance testing」，其實是有類似的味道。許多做 agile testing 的人都知道所謂的 agile acceptance testing，在開發一個 story 的時候，先幫這個 story 寫一個 acceptance test case （當然此時這個 test case 一定會失敗，因為程式碼都還沒出生啊），中間經過一連串開發過程（細節跳過），最後如果這個 acceptance test case 通過，就代表這個 story 完成。從另一個角度來看，我們可以說這個 acceptance test case 紀錄著它所代表（測試）的那個 story 的知識。

***

講了這麼多，其實這全部都不是重點，回到本篇的重點：「需求分析書中最重要的資訊是什麼？」

答案：寫這本需求分析書的那個人的電話號碼。

來源請參考本書第 25 頁：

*Ron Jeffries said, during his session on the natural laws of software*

*development at Agile 2008, that the most important information in a requirements document are not the requirements, but the phone number of the person who wrote it.*

這本書目前只看了 20% 左右，有機會看完的話再跟鄉民們報告。

***

友藏內心獨白：親愛的鄰居們，夜深了，烤肉用具可以收起來了。搞得空氣中都是烤肉味道，怎麼睡覺啊。

# 第十一部 欲練神功

# 100 從 The Timeless Way of Building 學設計(1)

05/07 23:07~05/08 01:05

12/19 11:48-12:03

原文發表於
http://teddy-chen-tw.blogspot.com/2010/05/timeless-way-of-building-1.html.

有一陣子 Teddy 很熱心地不斷跟周遭認識的人「推銷」*The Timeless Way of Building* 這本書，告訴他們這本書有多好，多棒，看了之後可以深入了解 design patterns 背後的理論基礎 (鄉民甲：知道這個東東要幹嘛，能加薪嗎？)。有少數幾個意志不堅的朋友誤信 Teddy 之言買了這本書，但是好像都沒有人認真的去讀這本書。Teddy 當年很幸運有好學不倦的指導教授與 Teddy 一起看這本書，一起討論，所以讀起來不孤單。也許這些被 Teddy 拐騙的朋友們沒有那麼幸運能有讀伴，所以可能看了幾頁就看不下去（如果曾經有鼓起勇氣打開這本書的話...）。

也許光是靠一張嘴是沒有用的，必須要舉例子說明讀這本書真的可以幫助提昇軟體分析與設計的能力，這樣才會有誘因下定決心去讀這本書。Teddy 為了「生出」一篇部落格文章（奇怪，明明沒有料還

要硬擠），在此就舉個例子說明一下。

請翻到這本書第 19 章 **Differentiating Space**，接下來 Teddy 節錄本章的若干句子：

*Within this process, every individual act of building is a process in which space gets differentiated. It is not a process of addition, in which pre-formed parts are combined to create a whole: but a process of unfolding, like the evolution of an embryo, in which the whole precedes its parts, and actually gives birth to them, by splitting.*

看懂了，恭喜老爺，賀喜夫人。看不懂也沒關係，繼續看下去，看到最後再回頭看這一段。

***

*Design is often thought of as a process of synthesis, a process of putting together things, a process of combination.*

*According to this view, a whole is created by putting together parts. The parts come first: and the form of the whole comes second.*

*But it is impossible to form anything which has the character of nature by adding performed parts.*

作者的意思「應該」是說設計不是把「紅茶 + 牛奶 + 糖 = 奶茶」，或是說「客廳 + 房間 + 廚房 + 廁所 + 陽台 + 玄關 = 房子」的一種**「合成過程」**。作者在書中不斷強調「建築物要有生氣 (alive)」，不是買集合式住宅蓋好直接搬進去住就好了，因為集合式住宅的建築師根本不知道誰會搬進來住，因此不可能依據個別家庭的需求來建造與設計房屋。Teddy 「猜想」作者的想法應該比較接近日本電視節目「全民住宅改造王」那樣，要考慮到個別居民（家庭）的需求以及住宅地點特性來設計與建造。但是，要如何達到這樣的目的？請看下去。

***

*It is only possible to make a place which is alive by a process in which each part is modified by its position in the whole.*

*In short, each part is given its specific from by its existence in the context of the larger whole.*

***This is a differentiating process.***

設計應該是一種**「差異化」**（有人翻譯成「分化」...聽起來好像是要搞破壞...）的過程。什麼意思？上面這幾句應該很清楚了：每一個 part 要依據它所在位置的**「地形地物」(context)** 來調整，修正。這種調整，修正就是一種「差異化」的過程，使得這個 part 可以適應其所在的特定的「地形地物」。還是太抽象...Teddy 也知道...舉個例子，前一陣子 Teddy 看了另一個日本的節目，內容也是和蓋房子有關。一對夫妻在山坡地上買了一小塊地要蓋房子，建築師考慮到太太在廚房作家事時（站著）也可以和坐在客廳地板上 的先生在聊天時有「目光接觸」，而不是只聽到彼此的聲音而已，以便增加夫妻相處時間的感情。夫妻都是上班族，每天一大早就要出門搭很久的電車到公司，平常 在家裡相處的時間並不長。因此建築師特別做了兩項設計：

- 把廚房安排在客廳旁邊
- 廚房和客廳之間有若干**落差**（好像是廚房地面比客廳地面要矮 50 公分之類的）

一般正常情況下我們都不希望室內有任何落差，尤其是家裡有老人家或是行動不便的人。因為就算是幾公分的落差都可能造成不便或是不小心跌倒。在這個案例 中，建築師設計了 50 公分這麼高的落差，乍看之下覺的很「瞎」（有多瞎...大概和「熊貓人」差不多瞎），但是這樣的設計，卻是可以讓太太在廚房作家事的時候，眼睛就可

以直接 看到坐在客廳地板上的先生的眼睛 **(目光接觸)**，看到電視畫面之後覺的真的很完美。

因為屋主的錢大部分都花在買地上面了，剩下蓋房子的預算變得很少，因此設計師只能減少蓋房子的建地面積（就是蓋小一點的房子，例如本來可以蓋 50 坪現在只能蓋 30 坪之類的 ）。加上屋主夫妻的特別需求，因此出現了有「落差」的室內空間。這就是「差異化」的過程，這種差異化使得這棟房子和全世界其他地方的房子都不相同，住在屋 內的人也因為這樣的設計生活的很愉快。

***

書的內容就先講到這邊（因為 Teddy 已經超過就寢時間），那麼上面屁了這一大段，到底和「軟體設計」有何關聯？長話短說，用兩點說明：

- 許多 design patterns 的初學者經常犯的一個應用 patterns 錯誤就是以為只要直接把 pattern A + pattern B + pattern C = System X。這種設計思考模式通常做出來的 System X 會變成四不像，**雖然「設計圖上」看起來很熱鬧，這邊一個 pattern，那邊又一個 pattern，但是仔細一看又覺的這個設計怪怪的，但是卻又說不出為什麼怪怪的。**這就是看這本書的用途啦，

現在鄉民有幸讀到 Teddy 這一篇，就可以大聲說出：**光是用「合成」的方式是無法得到好設計滴。**

- 既然用合成方式不行，那麼到底要怎樣用「差異化」來設計軟體？簡單舉例，講錯不負責（周公在趕 Teddy 了，無法仔細思考）：首先，要有 whole 與 part 的概念，然後當逐一將 part 擺到 whole 裡面的時候，依據 part 所在的位置調整 part 使其可以完美的融合到地形地物之中，最後變成其中不可分割的一部分。還是有聽沒有懂...假設把 software architecture patterns 當成是 whole，design patterns 當成是 parts。今天有某個設計，預計採用 layered architecture (whole)，其中 UI layer 要用 MVC (part)，而 MVC 裡面用了 observer pattern (此時 MVC 變成 whole，observer 變成 part)，但是由於這個 AP 是一個網路分散式系統，因此 View 可能在網路的不同端，因此這個 observer pattern 就和 GoF 中標準的 observer 有點不同 (差異化)。仔細讀一下 design patterns 這本書，書中有很多 patterns 其實也都有提到差異化的作法，例如 proxy pattern 就可以依據應用環境再區分為 remote proxy、virtual proxy 與 protection proxy。

***

要做好差異化其實不是一件容易的事情，很多設計成功與失敗的地方，就在於差異化的好壞。大家平平都是套用差不多種類的 patterns，

為什麼有的人開發的系統就比較容易了解，開發與維護，有的人的系統就好像麵線一樣，一坨一坨黏在一起。

讀懂 GoF 的 *Design Patterns: Elements of Reusable Object-Oriented Software* 可以從物件導向的角度來解釋設計，讀了 *The Timeless Way of Building* 讓鄉民們可以用更一般性，更抽象的觀點來解釋設計，這也是讓自己與一般平民百姓有所區別的「差異化」過程喔。

***

友藏內心獨白：奇怪了，Teddy 又不認識作者也不是出版社的人幹麼一直推銷這本書？

# 101 從 The Timeless Way of Building 學設計(2)

5/13 23:29~5/14 00:20

12/19 12:14-12:26

原文發表於
http://teddy-chen-tw.blogspot.com/2010/05/timeless-way-of-building-2.html.

有些人在念研究所的時候研究 design patterns 相關的題目，指導教授可能叫你寫幾個 patterns 出來再舉一些例子就可以畢業了。不知道鄉民們有沒有想過這個問題：**「要如何生出一個新個 pattern」**？今天 Teddy 來談一下這個問題（醜話說在前面，Teddy 部落格上的內容純粹為 Teddy 個人狂想曲，產品成分可能包含重金屬與不明配方，鄉民們自行服用之後若有任何副作用 Teddy 恕不負責）。

***

Teddy 第一份工作做的是 e-learning，因此當 Teddy 辭掉工作去念博速班的時候，指導教授建議 Teddy 可以研究「E-learning patterns」。這個主意還不錯，因為 Teddy 對於 e-learning domain knowledge 與 design patterns 都滿熟的，又有 e-learning 實務經驗，於是 Teddy 便開始了三年 (還是四年，這麼快就忘了！) 的 e-learning pattern languages 研究生涯。

俗話說：「事情不是笨蛋想得那麼簡單」，真正要自己「生」出一套 patterns 出來，才發現箇中的難處。雖然 design patterns 還算熟，但是**「用 patterns」**和**「生出 patterns」**是兩回事，此時想到 GoF 的書裡面提到他們的研究是得到 *The Timeless Way of Building* 的啟發，指導教授與 Teddy 就想這本書裡面會不會有教導「如何生出 patterns」的方法，所以萌生看這本書的想法。

翻開課本第 14 章：Patterns which can be shared

*To work our way toward a shared and living language once again, we must first learn how to **discover** patterns which are deep, and capable of generating life.*

第一個重點來了，**patterns 不是被「創造」出來的，而是老早就存在，等著被「發現」**。就好像生物不是被「創造」出來的，而是被「發現」。（當然以現在的技術也許真的可以無中生有創造出一個全新的物種，但這畢竟是人工的，不是天然的。記得嗎...電視廣告有交代：「天然 A 尚好」）這也是為什麼有些 patterns 用起來會覺得很好，很自然，有些卻覺得很生硬。後者可能就是「基因改造」的 patterns。

*Each pattern is a three-part rule, which expresses a relation between a certain **context**, a **problem**, and a **solution**.*

*As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.*

上面這一段有學過 design patterns 的鄉民們應該就比較熟悉了，從結構上來說，patterns 的三大要素：context, a problem, and a solution。在 GoF 第三頁則說一個 pattern 有四個基本元素：pattern name, problem, solution, consequences，耶怎麼不一樣？基本上還是一樣的，pattern name 就不用提了，一定要有，至於 Alexander (*The Timeless Way of Building* 的作者) 所說的 context 和 GoF 所說的 consequences 從字面上來解釋只是套用 pattern「之前」和「之後」的差別（美容前，美容後），因此 consequences 也叫做 resulting context (美容後)。如果不要把 context 特別侷限在套用 pattern 之前，那麼 context 應該包含了 consequences。簡單的公式

context → applying a pattern →resulting context (consequences)

如果看不懂請跳過...Orz。

*As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.*

*The pattern is, in short, at the same time a **thing**, which happens in the world, and the **rule** which tells us how to create that thing, and when we must create it. **It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing.***

這邊有一個重點，pattern 是一個「東西」同時也是一個描述如何產生這個東西的「流程」。所以，如果鄉民們發現了一個新的 pattern 打算將它寫下來，而其他人在看過你所寫的 pattern 之後了解到這個 pattern 的確有意義，但卻無法如法泡製，那麼這可能表示你所描述的 pattern 只是一個「thing」，而還不具備「process」的特質。

***

講了這麼多，那到底要如何發現一個 pattern？如果是要寫 pattern，可能會以 context→problem→solution 的順序出現，但是發現 pattern 的順序剛好相反。

We must first define some physical feature of the place, which seems worth abstracting.

因為 Alexander 尋找的是建築領域的 patterns，所以他提到「define some physical feature（i.e., spatial relationships, 空間上的關係）of the place」。例如，你到了總統府旁邊的「台北賓館」，覺的這裡面怎麼那麼漂亮啊，雖然是老建築物，但是裡

面的一些設計用在當代還是很有用。因此，你想要 找出這些你覺的好的設計，這就是「尋找 pattern」的第一步，從好的建築物（或是軟體，或是其他領域裡面被認為是好的東西）中定義出 physical feature（如果是軟體的話，就可能包含靜態結構關係與動態互動關係）。

Next, we must define the problem, or the field of forces which this pattern brings into balance.

Finally, we must define the range of contexts where this system of forces exists and where this pattern of physical relationships will indeed actually bring it into balance.

***

最後 Teddy 要補充說明，找出 patterns 的研究是一種「**empirical study**」，就是說要從實務經驗中來尋找 patterns。就好像新物種幾乎都是在野外原產地被發現是一樣的道理，不太可能坐在冷氣房裡面理論推導一番就可以「發現」新物種。Alexander 和他的同事也是花了 10 幾年以上的時間來發現與實際驗證建築領域的 patterns，而 GoF 裡面的 patterns 也都有 Known Use 這的段落（說明那些知名的軟體也用了這個 design pattern）。而 Erich Gamma (GoF 的第一個人) 自己也是開發了很多軟體來驗證這些 design pattern，包含比較沒人知道的 HotDraw 以及很有名的 JUnit 還有 Eclipse。

結論就是，除非鄉民們對於某個領域很熟，也有多年的實務經驗，不然隨便亂找 patterns 可是會變成「誤入叢林的小白兔」，害人害己啊。

PS：Teddy 當年也很想研究 A Pattern Language for Computer Game...但是 Teddy 只會玩「青蛙過馬路」這一種等級的遊戲，所以只好放棄啦。當時真恨自己沒有從小卯起來玩 game...

***

友藏內心獨白：言語無法形容，只能用心體會。

# 102 從 The Timeless Way of Building 學設計(3)

05/17 23:19~05/18 00:14

12/19 12:36-12:41

原文發表於
http://teddy-chen-tw.blogspot.com/2010/05/timeless-way-of-building-3.html.

沒時間了，廢話不多說，翻到課本第 24 章：The Process of Repair

*Next, several acts of building, each one done to repair and magnify the product of the previous acts, will slowly generate a larger and more complex whole than any single act can generate.*

上面這一句先讀一次，看不懂最後回頭再看一次。

*No **building** is ever perfect.*

*Each **building**, when it is first built, is an attempt to make a self-maintaining whole configuration.*

*But our predictions are invariably wrong. **People** use **buildings** differently from the way they thought they would. And the larger the pieces become, the more serious this is.*

把上面這三句裡面的 building 用 software 替換，people 用 customers、clients 或是 users 替換，是不是也完全講的通？這一段的重點還沒完，繼續往下讀。

*The process of design, in the mind's eye, or on the site, is an attempt to simulate in advance, the feeling and event which will emerge in the real building, and to crate a configuration which is in repose with respect to these events.*

*But the prediction is all **guesswork**; the real events which happen there are always at least slightly different; and the larger the building is, the more likely the guesses are to be inaccurate.*

*It is therefore necessary to **keep changing the buildings**, according to the real events which actually happen there.*

*And the larger the complex of buildings, neighborhood, or town, the*

*more essential it is for it to be build up **gradually**, from thousands of acts, self-correcting acts, each on **improving** and **repairing** the acts of the others.*

看完上面這幾句，鄉民們應該有聞到「iterative and incremental development」和「refactoring」的味道吧（鄉民甲：我鼻子不通，沒聞到...）。傳統 waterfall 開發流程強調 big up-front design，希望藉此讓之後的實做變得很順利。因此也就造成有人認為軟開發只要經過架構師與分析師分析設計好之後就可以把設計文件丟給 programmers，然後把 programmers 當成生產線工人一樣「操」，之後系統就自然而言生出來啦。看官們，有可能嗎？看看第三句就知道了 (But our predictions are invariably wrong...)。

Alexander 認為建築物不是設計好，蓋好就沒事了，還需要依據實際使用的狀況去改善與修復。這個精神和 agile methods 所談的避免 big up-front design 改採用 evolutionary design，並且頻繁地套用 refactoring 來增進現有系統的品質是相似的（Teddy 不敢說完全一樣啦...）。這個也和 David Thomas 在 http://www.artima.com/intv/dry.html 所提到的：

**All programming is maintenance programming**

的精神很類似。Programmers 其實隨時隨地都處在**「維護模式」**，因為很 少真正的開發活動是「全新的」。就算是開發一個新的 story，也都是這邊寫一寫，那邊改一改，總是多多少少會和已存在的程式碼打交道。如果抱持著 all programming is maintenance programming 的心態，看到設計不良的地方，隨手做 refactoring，則系統的品質便可逐漸改善，這也使得未來增加新的功能變得更容易。但是如果沒有這種**「時時勤拂拭，勿使惹塵埃」**的精神（很遺憾，很多人都沒有...），認為**「幹嘛改，程式可以動就好了啊」**，那麼軟體便逐漸朝向**「比硬體還要硬的硬體」**邁進（這是什麼東東？）。

***

友藏內心獨白：救命啊，這一系列快掰不下去了啦。

# 103 從 The Timeless Way of Building 學設計(4)

6/10 21:47~22:22

12/19 13:13-13:23

原文發表於
http://teddy-chen-tw.blogspot.com/2010/06/timeless-way-of-building-4.html.

最近幾天 Teddy 真的是忙到靠北...邊走，但是為了鄉民們的福利，Teddy 還是拼著睡不著也要寫這一篇，這也是 *The Timeless Way of Building* 的核心精神。各位觀眾，掌聲歡迎：「**The Quality Without A Name (QWAN)**」。

翻開課本第二章

*There is a central quality which is the root criterion of life and sprint in a man, a town, a building, or a wilderness. This quality is objective and precise, but it cannot be named.*

這一張照片是 Teddy 在 2007 年到法國 ~~旅遊~~ 參加研討會，途經日內瓦，在列馬湖畔所拍攝的。要如何幫這張照片命名，才可以表達出列馬湖的「quality」呢？說「寬廣」，列馬湖是滿大的，但是又沒 有大到那種非.....常.....大...的大。說「漂亮」，是很漂亮，可是又沒有那種美到極致的感覺。說「高」，嗯，列馬湖的噴水柱是很 高，可是又沒有 101 來的高。說「平靜」，列馬湖是會讓人整個心情放鬆，可是湖邊人來人往，又不是真的那麼「平靜」。

路人乙：（路人甲出場太多次了，今天換人）啊....哇哩勒，Teddy 你是來亂的嗎？~~吃這個也癢，吃那個也癢。~~ 說這樣也不行，說那樣又不對。你到底是想怎樣？

你說對了，quality without a name 就是說這樣也不行，說那樣又不對。也可以說「說這樣也行，說那樣也行」**因為列馬湖同時都具有上述特質（quality）的某些部份，但是又沒有任何一個特質可以完全表達它。** 就是說「言語無法形容，只能用心體會」。如果鄉民們去過列馬湖（沒去過也沒關係，在腦袋中想像一下你曾經去過台灣最美的地方，可能是太魯閣、龜山島、新山夢湖等等）你就能感受到那種特質。這個特質的本身十分精確，但是卻無法用言語完整形容（無法命名）。但是，一旦你去到另外一個地方，如果這個地方也具有 與列馬湖（或是你心目中所想像的那個地方）相同的特質，你卻能夠立刻辨識出來。下面的內容應該就容易了解了。

*It is never twice the same, because it always takes its shape from the particular place in which it occurs.*

*The fact that this quality cannot be named does not mean that is is vague or imprecise. It is impossible to name because it is unerringly precise. Words fail to capture it because it is much more precise than any word. The quality itself is sharp, exact, with no looseness in it whatsoever. But each word you choose to capture it has fuzzy edges and extensions which blur the central meaning of the quality.*

再舉個例子，以「whole (完整)」這個字來看，完整是什麼意思？

路人乙：完整就是完整啊，這有什麼好講的。
Teddy：一整顆蘋果被咬一口，算不算完整。
路人乙：廢話，當然不算。

可是人家 Apple 的蘋果就是被咬一口啊，它卻是很「完整」的傳達了「人人看了 Apple 的產品都想咬一口的意念」。所以，這樣算不算「完整」。讀一下下面這句。

Imagine the quality without a name as a point, and each of the words which we have tried as an ellipse. Each ellipse includes this point. But each ellipse also covers many other meanings, which are distant from this point.

***

講到這邊，那這一堆有的沒的和軟體設計有何關係。這.....快想一個說法.....。當一個建築物，城鎮，都市，具有 QWAN，那生活在裡面的人就是很自然，幸福，舒服，有活力，有朝氣，頭好壯壯，長命百歲...。問題來了，既然這個建築物，城鎮，都市具有 QWAN，那就是說「言語無法形容，只能用心體會」，那怎麼可以重複蓋出（建

造出)具有 QWAN 的其他建築呢？總要有一個辦法吧！這就是本書作者 Alexander 在後續章節所要提出方法，就是用 pattern languages 來表達。

這...還是沒講到和軟體設計有何關係。...ㄟ， 鄉民們，如果有機會看到別人的軟體設計（軟體架構，~~一堆沒三小路用的~~ UML diagrams 或 source code 等），不知道是否曾經有那種「ㄟ，這個設計怎麼那麼好」或是「靠，這程式是誰寫的」這種感覺，可是有時候又說不清楚到底好在那裡，差在哪裡？如果鄉民們 學過了很多 patterns（不局限於 design patterns，也可以是 architecture patterns、coding patterns、exception handling patterns 等等）那麼，就可能「嘗試」（學過一堆 patterns 也不一定代表馬上就有能力把 patterns 串成 pattern languages，這兩者還是有所不同）從 pattern languages 的角度來解讀這個軟體設計。具有 QWAN 的軟體設計，不管它是哪個領域的軟體，用什麼語言開發的，你會覺得這個設計好棒，容易修改，擴充，測試，維護等等。沒有這種特質的，ㄟ...覺的好像住在 「鐵皮屋」，「頂樓加蓋」，或是「違章建築」裡面，總是覺的「鳥鳥的」。

***

友藏內心獨白：QWAN，我真是搞不懂你啊。

# 104 從 The Timeless Way of Building 學設計(5)

07/25 22:30~23:48

12/19 13:25-13:30

原文發表於
http://teddy-chen-tw.blogspot.com/2010/07/timeless-way-of-building-5.html.

在寫這一篇之前，Teddy 心中有一個疑問，**這一系列的文章，會不會太玄了一點，到底有沒有人在看啊？**Teddy 是不是要學一下電視台，對於收視率太低的節目讓它提早下檔...獨角戲唱久了也是會累滴。

請翻開課本第 526~527 頁：

*But then I read a passage in an ancient Chinese painting manual--the Mustard Seed Garden manual of painting--which made the situation clear* to me.

這句有點沒頭沒尾的，先解釋一下。Alexander 在他的書中經常會提起以前古老的建築物具有許多好的特質，而當今的許多建築物則是缺少了這樣的特質。久而久之 Alexander 不禁懷疑自己內心是否屬於

守舊派，而他的創作也只是在重複以前人們的作品而已。一直到他有一天讀了一本中國清朝初年的書，叫做「芥子園畫傳 （又稱為芥子園畫譜）」，他才釋疑。這本書的英文版在 Amazon 也買得到喔，真是太猛了。

接著看下去，有點長，請耐心看完。

*The writer of the manual describes how, in his search for a way of painting, he had discovered for himself the same central way that thousands of others like him had also discovered for themselves, throughout the course of history.* ***He says that the more one understands of painting, the more one recognizes that the art of painting is essentially one way, which will always be discovered and rediscovered, over and over again, because it is connected with the very nature of painting, and must be discovered by anybody who takes painting seriously. The idea of style is meaningless: what we see as a style (of a person or of an age) is nothing but another individual effort to penetrate the central secret of painting, which is given by the Tao, but cannot itself be named.***

上面用粗體字與底線標注的這幾句 Teddy 覺的很重要，繪畫（換成建築、廚藝、園藝、軟體開發都行）的藝術基本上只有一種方法，

而這種方法勢必被認真，用心投入繪畫的人不斷地發現與重複發現。至於 所謂的 style 則是不重要的，這只是某人或是某個時代來引導人們一窺繪畫秘密的一種手段而已。繪畫的核心秘密是「道」所賦予的，但是本身卻無法被命名。

還是看不懂，好吧，用現代人的講法：「只要有心，人人都可以是食神」。

為了還沒去買書的人，Teddy 就引用多一小段吧。這一段就不解釋了。

*The more I learn about towns and buildings, the more I feel the same thing to be true. It is true that many of the historic styles of building have some quality in common--they have it not because they are old, but because man has, over and over again, approached the secret which is at the heart of architecture. In fact, the principles which make a building good, are simple and direct--they follow directly from the nature of human beings, and the laws of nature--and any person who penetrates theses laws will, as he does so, come closer and closer to this great tradition, in which man has sought for the same thing, over and over again, and come always to the same conclusions.*

***

問題又來了，這些和設計有何關係？鄉民們先想一下...

講一個故事。Teddy 帶著實驗室的學弟們開發一個「持續整合」系統也已經有五年多的時間了，在這過程中，我們嘗試了許多 「styles」，例如：

- 利用「Javaspaces 分散式架構」與平行計算技術來加快持續整合的速度。
- 套用 Eclipse 中的 Builder 概念，讓使用者可以很容易的使用「單一整合工作」，而開發者也可以很容易的擴充各種新的整合工作。
- 基於上述兩點，讓系統達到「跨平台持續整合」的功能。
- 導入持續整合 workflows，讓使用者自行設定整合流程。
- 支援專案，相依專案，與函式庫專案的概念。

還有一堆有的沒的。這些手段，都只是我們試著去探求「持續整合核心秘密」的方法，而非最終目的。這個「持續整合核心秘密」本身很精確，但是卻無法被命名，**只有透過不斷地在軟體開發中親身徹底貫徹實施「持續整合」才可以慢慢地了解**。雖然 Teddy 經常提醒學弟們既然在研究持續整合，就要「用力的」在開發軟體的時候實踐它，如此才能了解「problem domain」的真正需求。很可惜大多數的人並沒有體會到這件事情的重要性，而比較關注在技術性的問題上（solution domain）。

***

結論，「**The art of painting must be discovered by anybody who takes painting <u>seriously</u>**」。做學問或做任何事情要升格到「達人」的境界，勢必要腳踏實地，逐步踏實。馬步蹲的穩，以後功夫學得自然快，學得深。

***

友藏內心獨白：一個 Teddy 兩個 teams，同樣的帶人方法進度為什麼差那麼多？

# 105 Quality Without A Name vs A Name Without Quality

6/12 20:27~21:12

12/19 13:33-13:37

原文發表於
http://teddy-chen-tw.blogspot.com/2010/06/quality-without-name.html.

不知道鄉民們有沒有搭過台北市的 307 公車，該公車行經台北縣市黃金路段，向來以**「班次多、車速快、司機狠」**等等特質聞名於世。307 公車現有「台北客運」與「大有巴士」兩家公司共同服務，「大有巴士」的服務品質不用多說，一句話：**「罄竹難書」**。 相較起來，「台北客運」就好很多，至少它們在轉彎的時候，都會禮讓行人。對於每天都要搭 307 上下班的 Teddy 來講，深深了解到 307 公車的「Quality」。不過，也不能一竿子打翻一船人，在「大有巴士」307 路線中，偶而也會讓 Teddy 遇到那 1%左右的好司機，而在「台北客運」307 中，也有 30%左右的司機會讓 Teddy 以為他們是從「大有巴士」借調而來的。總而言之，雖然司機有好有壞，整體而言 307 公車具有的「Quality Without A Name (QWAN)」只要搭過幾次的人永生難忘。

有一次，Teddy 和 Kay 到台北車站搭某一路公車要到臺北市立美術館，上車之後沒過多久，Teddy 和 Kay 互看了一眼，發出會心的一笑：**「這路公司具有 307 的 QWAN」**。

QWAN 雖然無法被精確的命名，但是一旦鄉民們在某處經歷過，而在另一處又重新出現之後，必定能夠很快的感受到。

***

在社會上也經常能夠發覺到 QWAN 的正面與反面例子。從 10 幾年前至今，Teddy 陸陸續續也認識到不少「博士」，在名片上要印上「博士」，在 e-mail 結尾要加上「博士」，在投影上要加上「博士」，在嘴上更是三不五時要表明自己是名校正港「博士」的身份。這些人也都具有相同的「Quality」，包括 「目空一切，自視甚高，舌燦蘭花，說謊不打草稿，眼高手低，號稱經驗與人脈豐富，喜歡拉關係，包打聽，etc」。實際接觸後，真是「好一個博士啊」。看過「海綿寶寶」嗎？在某一集的劇情中「派大星」也告訴別人**「請叫我派大星教授加博士」**，此時笑一笑就好，千萬別太認真。這種現象有另外一個名稱，叫做**「A Name Without Quality」**。

各位看官們，要注意區分好的和壞的「Quality Without A Name」，以及那些「A Name Without Quality」，才不會輕易被唬住了。

***

友藏內心獨白：老佛爺是要放在心裏尊重的，像你這樣整天掛在嘴邊講，毫無敬意，你是何居心？

# 106 有駕照不會開車

May 31 20:18~23:30

12/19 13:38-13:43

原文發表於 http://teddy-chen-tw.blogspot.com/2011/05/blog-post_31.html.

Teddy 是一個頭腦簡單，四肢不發達的人。距離感，方向感，還有手腳的反應很差，從小到大唯一學會的交通工具，就是騎腳踏車，而且還只能在河濱公園騎，不能騎到大 馬路上，否則是很危險滴（路人危險，Teddy 也危險）。前一陣子在某人一再的逼迫之下，Teddy 不得已只好以近 XX 的「高齡」在五月初繳了 1 萬元去報名了駕訓班（其實是 1 萬 1 千，因為上了一次課之後多交了 1 千塊參加了所謂的「路考保證班」...XD），昨天是筆試的日子，今天是路考，終於在 5 月的最後一天考上最簡單的自排車駕照。

到駕訓班上課之前一直聽說很多人考取駕照之後還是不會開車（不敢開車上路），Teddy 就覺得很納悶，不是都考到駕照了嗎，怎麼會不敢開車上路？等到 Teddy 親自去上課之後才發現，**切，這那是學開車啊，簡直是「李棠華特技團」訓練班嘛（老一輩的人才懂...XD）**！開車搞得好像以前上電子實習課的時候，在麵包板上面接電路一樣，「按圖施工，保證成功」。

每輛教練車都用立可白或與磁鐵在某些位置做上「記號」，請容許 Teddy 把整個過程做個抽象化的描述（做軟體的人就是喜歡 abstraction...XD）：在駕訓班學開車就是要學會如何利用這些「記號」與場地中的「固定參考點（可能是標竿，安全島，地上的黃線交叉點...）」的相對位置達到某種關係（某個角度）的時候，把「方向盤往左或是往右轉若干圈」。就這樣，敢問把車子真正開到路上的時候，到哪裡去找這些「參考點」，所以，按照這種訓練方式拿到駕照會開車應該算是天才了。

Teddy 也曾經想要試著按照自己的意思憑直覺隨便亂開，但是只要一沒開好被教練看到，就會 X@!#!@$... 總之教練**只管你有沒有拿到牌，不太在意你會不會開車。**

Teddy 在練車的時候，練著練著腦中突然浮現當年去資 X 會上 CMMI 課程時的光景（學費貴死人...還好是善心人士買單...XD）。在上 CMMI 課程之前，Teddy 自認為算是滿懂軟體開發這檔子事，上完課之後，Teddy 突然覺的不知道要如何開發軟體了，如果鄉民們知道「邯鄲學步」這句成語，那麼就可以體會 Teddy 當時的心情。

***

類似「A name without quality （拿到某某牌卻不會辦事）」的例子實在是太多了，拿最近最熱門的「塑化劑」來講，不是有某位議員

的太太，買了某 ISO 認證的健康食品，以為有 ISO 認證就有保障，實在是大錯特錯啊。ISO 認證這檔子事本身已經搞成一種「產業」了，至於品質提昇多少，就不容 Teddy 在此多做介紹了。

話說回來，台灣人真是太厲害了，只要有人敢「發牌」或「發照」，咱們台灣人就可以想出一套超級有效率又保證成功的方法來讓「有心人」可以拿到這些牌，照（真是皇天不負 ~~苦~~ 有心人啊...XD）。更強的是，在很多情況下 ~~輔導的人~~ 教你的人就是負責評鑑你的人。這樣的模式，要拿不到這些牌，照的人才算是異類阿（大概比在台灣考不上大學還要難...XD）。

***

話說 Teddy 拿到駕照之後還要再準備花一筆錢去找教練教「如何開車上路」，這種「兩階段學開車」的方式，還真是「台灣經濟奇蹟幕後的無名英雄」啊，真是忍不住想給它一個「讚」。

Teddy 曾經問過某位幫人家導入 CMMI 的顧問一個很直接的問題：

Teddy：我聽過若干 programmers 說，導入 CMMI 都在做文件，做假資料，都是老闆想要拿牌，對軟體開發沒有很大的幫助。請問以您的經驗，導入 CMMI 真的對於開發軟體有幫助嗎 ？

CMMI 顧問：也不能說完全沒有幫助，畢竟經過導入的過程，每家公司還是都學到了一些東西。

顧問不虧是顧問，回答的真好。

***

場景換到駕訓班。
Teddy：我聽過很多人說，到駕訓班學開車，拿到駕照之後還是不敢上路。請問以您的經驗，到駕訓班學開車真的有幫助嗎 ？

教練：也不能說完全沒有幫助，畢竟經過練車的過程，每個學員還是都學到了一些東西。

是啊，反正只要弄清楚，到駕訓班學開車的目的是「拿到駕照」，不是學會開車，這樣就 OK 了（這句話聽起來怎麼怪怪滴...XD）。
鄉民們有看到最近電視上狂播的那個「沒有」的廣告嗎？

又「沒有」人叫你去學...XD

***

友藏內心獨白：有些事，還是趁年輕去做比較好。

# 作者簡介

Teddy Chen，泰迪軟體創辦人，著有《笑談軟體工程：敏捷開發法的逆襲》與《笑談軟體工程：例外處理設計的逆襲》等書。

Teddy 畢業於台北科技大學機電科技研究所（資訊組）博士班，有二十年以上的軟體開發經驗。Teddy 曾擔任程式設計師、技術總監、專案經理、軟體架構師，目前為敏捷顧問與敏捷培訓課程講師，並在台北科技大學資工所擔任兼任助理教授，開設敏捷與精實軟體開發、軟體架構兩個課程。

聯絡 Teddy：
teddy@teddysoft.tw

部落格：
https://teddy-chen-tw.blogspot.com/

搞笑談軟工 FB 社群:
https://www.facebook.com/groups/teddy.tw/

This page is intentionally left blank