軟體型態管理

2005年版的課本

資科系
林偉川

型態管理的基本概念

- 引言
- 型態管理的定義
- 型態管理與軟體開發過程的關係
- 型態管理的目的
- 型態管理的標準
- 型態管理的重點：Manage "change"

2

引言

- 大部份的軟體開發工作是由一群擁有不同背景與專長的人共同合作完成，因此在進行軟體設計工作時可能會有下列情形發生：
  - 多人同時更新某一個軟體
  - 多人共同使用同一個程式碼，雖然部分程式已被更新，但仍有些人不知道(side effect!!) ➔ 使用版本不一致
  - 雖然發現程式的錯誤原因，但卻無法有效通知相關人員以避免錯誤的影響繼續擴大
  - 許多設計人員可以共用同一支程式，但設計者卻自行開發相同的程式

3

引言

- 在軟體發展的過程中，變更是不可避免的，因此有效的控制軟體的變更是軟體開發部門的重要工作項目
- 型態管理的重要問題如下：
  - 專案開發人員要如何瞭解現行發展的軟體版本？
  - 如何有效地控制軟體變更問題？
  - 軟體發生變更時，該如何通知相關人員？
  - 進行軟體更改時，應如何確保一定的軟體品質水準？
  - 如何評估軟體變更所造成的影響？

4

型態管理的定義

- 型態項目(Configuration Item, CI)
  - 軟體或硬體的組合元件能夠滿足使用者的需求，而且能夠應用在型態管理上之用途者(MIL STD 480)
- 型態管理(Configuration Management, CM)
  - 在整個軟體生命週期中，針對型態項目予以定義和控制各變更、紀錄與報告型態項目的實際狀況，以確保其完整性 (Integrity) 與可追蹤性 (Tractability) 的管理過程(IEEE STD 615-12)
- 但是可追蹤性的可行性很低
  - Requirement -> executable code ?

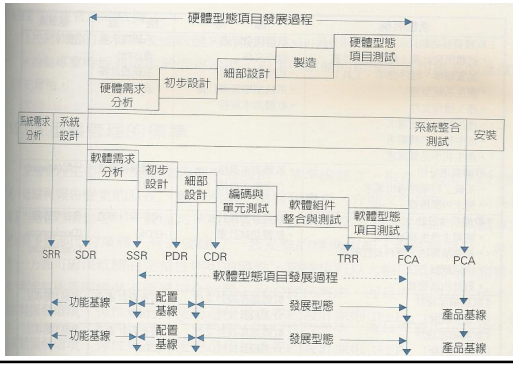
5

型態管理的定義

- 型態管理的主要工作：
  - 型態識別(Configuration Identification)
    - 定義與控制軟體型態的項目、基準線(Baselines)
    - 安排型態項目的發展時程
  - 型態變更控制(Configuration Change Control)
    - 對於型態項目的變更進行控制
  - 型態稽核(Configuration Auditing)
    - 對於軟體開發過程的基準線實施稽核工作，以確保軟體功能符合既定需求
  - 型態狀況報告(Configuration Status Report)
    - 提供軟體發展過程之記錄

6

型態管理與軟體開發過程的關係



型態管理與軟體開發過程的關係

- SRR：系統需求審查(System Requirement Review)
- SDR：系統設計審查(System Design Review)
- SSR：系統規格審查(System Specification Review)
- PDR：初步設計審查(Preliminary Design Review)
- CDR：嚴密設計審查(Critical Design Review)
- TRR：測試準備審查(Test Readiness Review)
- FCA：功能型態稽核(Functional Configuration Audit)
- PCA：實體型態稽核(Physical Configuration Audit)

8

型態管理的目的

- 型態管理是運用各種工具、方法與程序管制軟體發展的變更活動，使軟體品質維持良好的一致性與完整性，並使資源運用到最佳的效果
- 目的如下：
  - 降低軟體修改導致專案失敗的風險
  - 建立軟體發展過程中所有的記錄(程式碼、文件等)
  - 控制軟體的品質與改進軟體再用性(Reusability)
  - 控制專案時程與成本於既定水準

9

型態管理的主要工作項目

- 型態識別
  - Configuration Identification
- 型態變更管制
  - Configuration Change Control
- 型態稽核
  - Configuration Audit
- 型態狀況報告
  - Configuration Status Report

10

型態識別

- 型態識別主要的工作在於：
  - 訂定型態項目(Configuration Item, CI)
  - 訂定型態基準線(Configuration Baseline)
  - 製作型態項目的文件與編碼
- 型態識別的主要目的在於：
  - 建立及維持一套基準線，使其能在軟體生命週期內，對於每個型態項目做管制及狀態的彙報

11

型態識別

- 型態識別的過程：
  - 確認
    - 將軟體的類型(原始碼、資料、文件)、結構與基準線做適當的分類與確認
  - 命名
    - 應考慮命名的規則與習慣以及版本管理的需求
  - 取用
    - 取得、暫存、讀取、再應用
    - 應考慮執行效率與安全性
  - 存放
    - 資料庫管理 (長期的再應用)

12

型態變更管制

- 型態變更管制是針對型態項目中已建立的基準線
  - 提出修改
  - 進行評估、協調與核准
  - 對於已核准之修改加以執行
- 基準線為一個工作產品，指出某個活動的結束與另一個活動的展開之臨界點➔time
- 變更管理的實現，除確認每一個基準線文件(軟體工程指引、組態管理計劃、規格)，還要持續追蹤之後對於該基準線所做的任何變更
- 負責型態控制工作的主要單位：
  - 「型態控制委員會」(Configuration Control Board, CCB)

13

組態管理的活動

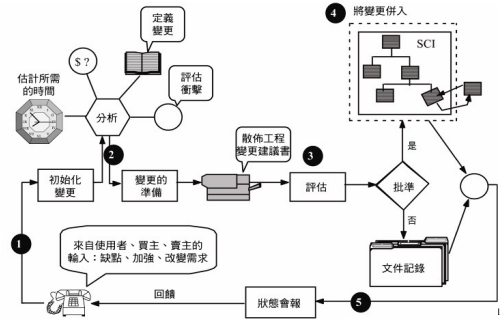
軟體組態管理活動
1.建立基準線*
  1.1 為每個組態項目（CI）指定獨特的識別碼（CI的例子：專案計劃、專案軟體工程指引、需要敘述、需求描述、架構的描述、原始碼、測試計劃）
  1.2 建立基準線文件
2.識別不同的內部發表版本（即同一個產品基準線的連續版本）
3.確保產品文件係根據完整且最新的技術。
4.實施標準
  軟體品質保證（SQA）
5.運用軟體組態管理，將每個組態項目由某個開發階段或測試，提升至另一個階段
6.利用軟體組態管理以確認顧客在內部（開發）基準線的參與及影響

*基準線（baseline）= 通過正式地複審並取得協議的工作產品，可作為進一步開發的基礎。唯有透過正式的變更控制程序，才能改變基準線。
基準線文件 = 指個別軟體文件或一組文件，它（們）完整地描述某個組態項目。

來自IEEE標準1042-1987

14

軟體組態控制程序



控制程序的基本5個步驟

- **初始化變更**
  - 對於SCI變更要求，由團隊成員與/或專案客戶提出
- **分析**
  - 針對變更的定義所需時間成本及對工作排程的影響，進行分析
- **變更的準備**
  - 將每個SCI的提議變更之決議，分發給團隊成員

16

控制程序的基本5個步驟

- **評估**
  - 評估SCI的變更提議，藉由自動化工具(PSLs)的支援，用來紀錄實現變更、文件版本維護與變更。遭否決的變更以文件紀錄，做為將來的參考
- **回饋**
  - 變更評估的結果，回饋至軟體開發者與客戶

17

軟體組態控制

- **工程變更建議書提供對於提議變更的描述，並明確指出該變更的發起者、基本理由，以及將會受影響的基準線，亦指出專案中會受到提議之變更影響的基準線及圖形，藉此可幫助預估完成該變更所需的時間及成本**
- **提案人所敘述之有關需要變更的理由，有助於評估變更對專案造成的衝擊，若完成某個變更的代價(成本)很高，則提案人應舉出強而有力的理由，做為決定是否實現該變更的判斷依據**

18

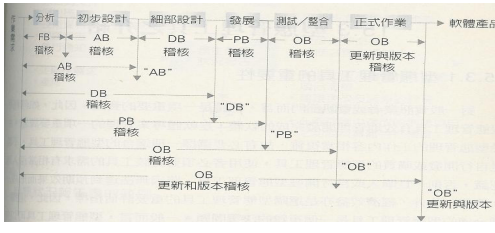
工程變更建議書表格樣本

工程變更建議書	提出日期	表單編號
提案組織之名稱及地址	理由代號	ECP編號/類型
變更影響的基準線	變更影響的圖形：gatt、pert..	
變更的描述：		
變更的需求：		
預估交付時程	預估變更所需的成本：	
☐同意　☐不同意	簽名：	決定日期：

19

型態稽核

- 型態稽核的目的是確保軟體能夠符合規格、標準、合約規定或是其他規定準則
- 型態稽核的主要活動：

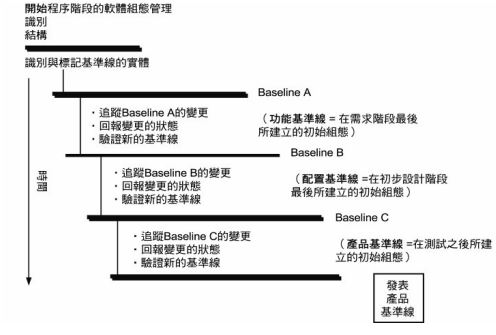


型態稽核

- FB: 功能基準線(Functional Baseline)
- AB: 分配基準線(Allocation Baseline)
- DB: 設計基準線(Design Baseline)
- PB: 產品基準線(Product Baseline)
- OB: 操作基準線(Operational Baseline)

21

變更管理模型



型態管理工具之介紹

- Version Control 的概念(Wincvs http://freefall.csie.isu.edu.tw/wincvs/wincvs.html)

23

Version Control: Introduction

- Version control tool provides some or all of the following basic features:
  - It provides a place to store your source code
  - It provides a historical record of what you have done over time
  - It can provide a way for developers to work on separate tasks in parallel, merging their efforts later
  - It can provide a way for developers to work together without getting in each others' way
- Version Control is also known as "Source Code Management" or SCM

24

From: http://www.ericsink.com/scm/source_control.html

Version Control: The Basics

- Two Important Terms:
  - Repository
    - a place to store your source code
  - Working folder
    - where each individual developer does their work
    - located on a desktop machine and accessed using a client

25

Version Control: The Basics

- The workflow of a developer is an infinite loop which looks something like this:
  - Copy the contents of the repository into a working folder
  - Make changes to the code in the working folder
  - Update the repository to incorporate those changes
  - Repeat

26

Version Control: The Basics

- Let's imagine for a moment what life would be like without this distinction between working folder and repository:
  - People store their code on a file server. Everyone uses Windows file sharing and edits the source files in place. When somebody wants to edit main.cpp, they shout across the hall and ask if anybody else is using that file. Their Ethernet is saturated most of the time because the developers are actually compiling on their network drives.

27

Version Control: The Basics

- There is a separation between working folder and repository
- Thus the most frequently used features of an SCM tool are the ones which help us move things back and forth between them
- Here are some commonly used terms:
  - Add
  - Get
  - Checkout
  - Checkin

28

Version Control: The Basics

- Add: A repository starts out completely empty, so we need to "Add" things to it. Using the "Add Files" command in Vault you can specify files or folders on your desktop machine which will be added to the repository.
- Get: When we copy things from the repository to the working folder, we call that operation "Get". Note that this operation is usually used when retrieving files that we do not intend to edit. The files in the working folder will be read-only.

29

Version Control: The Basics

- Checkout: When we want to retrieve files for the purpose of modifying them, we call that operation "Checkout". Those files will be marked writable in our working folder. The SCM server will keep a record of our intent.
- Checkin: When we send changes back to the repository, we call that operation "Checkin". Our working files will be marked back to read-only and the SCM server will update the repository to contain new versions of the changed files.

30

Version Control: The Basics

- Get vs. Checkout
  - Lock-Modify-Unlock vs. Copy-Modify-Merge (refer to SVN book, http://svnbook.red-bean.com/)
  - Lock-Modify-Unlock or Checkout-Edit-Checkin (Rules):
    - Files in the working folder are read-only unless they are checked out
    - Developers must always checkout a file before editing it. Therefore, the entire team always knows who is editing which files
    - Checkouts are made with exclusive locks, so only one developer can checkout a file at one time.

31

Version Control: The Basics

- Get vs. Checkout
  - Copy-Modify-Merge or Edit-Merge-Commit
    - Steps:
      - Edit the working file as needed
      - Merge any recent changes from the server into the working file
      - Commit the file to the repository
    - Rules:
      - Files in the working folder are always writable
      - Nobody uses checkouts at all, so nobody knows who is editing which files
      - When a developer commits his changes, he is responsible for ensuring that his changes were made against the latest version in the repository

32

Version Control: The Basics

- Your repository is more than just an archive of the current version of your code. Actually, it is an archive of every version of your code. Your repository contains history. It contains every version of every file that has ever been checked in to the repository. For this reason, a source control tool as a time machine.
  - One can easily retrieve an exact copy of the source code
  - If there is a piece of code looks strange and nobody can figure out why, it's handy to be able to look back at the history and see when and why a certain change happened

33

Version Control: Checkins

- Editing a single file
  - Checkout the file
  - Edit the working file as needed
  - Checkin the file back to repository

34

Version Control: Checkout

- ***Checking out*** a file has two basic effects:
  - On the server, the SCM tool will remember the fact that you have the file checked out so that others may be informed
  - On your client, the SCM tool will prepare your working file for editing by changing it to be writable

35

Version Control: Checkout

- Undoing a Checkout
  - Normally, a checkout ends when a checkin happens. However, sometimes we checkout a file and subsequently decide that we did not need to do so. When this happens, we "undo the checkout". Most SCM tools have a command which offers this functionality. On the server side, the command will remove the checkout and release any exclusive lock that was being held
  - (Note: Copy-Modify-Merge does things a little differently)

36

Version Control: Checkout

- The user has three choices for how the working file should be treated:
  - Revert: Put the working file back in the state it was in when I checked it out. Any changes I made while I had the file checked out will be lost
  - Leave: Leave the working file alone. This option will effectively leave the file in a state which we call "Renegade" (betray). It is a bad idea to edit a file without checking it out. When I do so, Vault notices my transgression (against law) and chastises (punish) me by letting me know that the file is "Renegade"
  - Delete: Delete the working file

37

Version Control: Checkins

- The process of a ***checkin*** isn't terribly complicated:
  - The new version of the file is sent to the SCM server where it is stored
  - The version number of the file in the repository is incremented by one
  - The file is no longer considered to be checked out or locked
  - The working file on the client side is made read-only again

38

Version Control: Checkins

- Checkins are additive
  - It is reassuring to remember one fundamental axiom of source control: Nothing is ever destroyed. Let us suppose that we are editing a file which is currently at version 4. When we checkin our changes, our new version of the file becomes version 5. Clients will be notified that the latest version is now 5. Clients that are still holding version 4 in their working folder will be warned that the file is now "Old"
  - But version 4 is still there. If we ask the server for the latest version, we will get 5. But if we specifically ask for version 4, and for any previous version, we can still get it.

39

業界 "Best Practice"

- Microsoft's "Daily Build and Smoke Test"
  - Build daily
    - "Treat the daily build as the heartbeat of the project. If there's no heartbeat, the project is dead."
    - Some organizations build every week, but the problem is that if the build is broken, it may be weeks before the next good build
- Check for broken builds
  - If the software isn't usable, the build is considered to be broken and fixing it becomes top priority
  - Strict enough to keep showstopper defects out but lenient (generous) enough to ignore trivial defects (an undue attention to which could paralyze (degrade) progress)

40

From: http://www.stevemcconnell.com/ieeesoftware/bp04.htm

業界 "Best Practice"

- Microsoft's "Daily Build and Smoke Test" (cont.)
  - Smoke test daily
    - A "smoke test" is a relatively simple check to see whether the product "smokes" when it runs
    - The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly
    - The daily build has little value without the smoke test
    - The smoke test must evolve as the system evolves
  - Establish a build group
    - On Windows NT 3.0, for example, there were four full-time people in the build group

41

From: http://www.stevemcconnell.com/ieeesoftware/bp04.htm

業界 "Best Practice"

- Microsoft's "Daily Build and Smoke Test" (cont.)
  - Add revisions to the build only when it makes sense to do so
    - Individual developers usually don't write code quickly enough to add meaningful increments to the system on a daily basis, they should only integrate consistent collection of code
  - Create a penalty for breaking the build
    - Make it clear from the beginning that keeping the build healthy is the project's top priority
    - If the build is broken too often, it's hard to take seriously the job of not breaking the build
    - lollipops (candy) for "suckers"; pagers to be called in

42

From: http://www.stevemcconnell.com/ieeesoftware/bp04.htm

業界 "Best Practice"

- Microsoft's "Daily Build and Smoke Test"
  - Build and smoke even under pressure
- BENEFITS
  - It minimizes integration risk
  - It reduces the risk of low quality
  - It supports easier defect diagnosis
  - It improves morale
    - knowing that the code is always in a working/workable state everyday!

43

From: http://www.stevemcconnell.com/ieeesoftware/bp04.htm

業界 "Best Practice"

- Requirements?
  - 需要比較大的支援團隊
    - 可能可以跟測試團隊結合？
  - 可能比較適合於非常大型的專案(Windows, Office!)
  - 但是也許可以引用一些重點的概念

44

結論

- Configuration Management is "Change Management"
  - Instead of managing all the details of change and tracing them from requirements to executable, we can use some tricks to just manage the results
  - Version Control (both of source code and document)
  - Daily Build and Smoke Testing to ensure working code
- Naming Conventions(命名/編碼原則)
- Coding Conventions

45