**Architectural design：**軟體架構是組織的商業目標與實現該目標的軟體系統之間的橋樑。軟體架構包含結構及結構間的關係，以及軟體與非軟體之間的配置關係。

**Software Architecture vs. Software Design：**

Software architecture負責軟體的框架和high-level infrastructure

Software Design負責code level design，例如每個module的工作、classes scope、functions purposes等。

**Architectural abstraction：**

1. Architecture in the small：將individual programs分解為components的方式。
2. Architecture in the large：包含其他系統、programs、program components的複雜企業系統的架構。

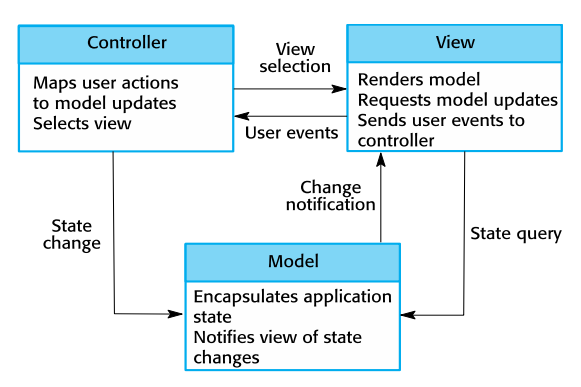
**Advantages of explicit architecture：**Stakeholder communication、System analysis、Large-scale reuse

**4 + 1 view model of software architecture：**架構可以從幾個不同的視角中紀錄，例如logical view、process view、development view (implementation view)、physical view (deployment view)、use cases or scenarios(+1)。

**Model-View-Controller (MVC) pattern：**將presentation和interaction與system data分開。該系統被建構為相互溝通的三個logical components。

Model component：管理system data以及對該data的相關操作。

View component：定義和管理如何將數據呈現給用戶。

Controller component：管理user interaction，並將這些interactions傳遞給View和Model。

**Layered architecture pattern：**將系統功能分成好幾個層級，一層只依賴緊鄰的下一層所提供的服務。

Advantages：只要maintain interface，就可以替換整個層。

Disadvantages：

1. 通常很難在層之間提供清晰的分隔，並且高層可能必須直接與較低層interact。
2. 性能較差，因為服務請求需要多層處理。

**Repository pattern：**系統中的所有數據都在central repository中進行管理，所有系統components均可存取該central repository。

Advantages：

1. components可以是獨立的，它們不需要知道其他component的存在。
2. 一個component所做的更改可以傳播到所有components。
3. 可以對所有數據進行統一管理。

Disadvantages：

1. Repository中的問題會影響整個系統。
2. 資料存取效率較低。
3. 在數台computers之間分佈repository可能很困難。

**Client–server pattern：**系統的功能被組織為服務，每個service從separate server交付。Client是這些services的用戶，透過access servers以使用services。

Advantages：servers可以分佈在整個網絡中。

Disadvantages：

1. 每個服務都是single point of failure，因此容易受到拒絕服務攻擊或服務器故障的影響。
2. 性能可能是不可預測的，因為取決於網絡以及系統。
3. 如果servers由其他組織擁有，則可能有管理問題。

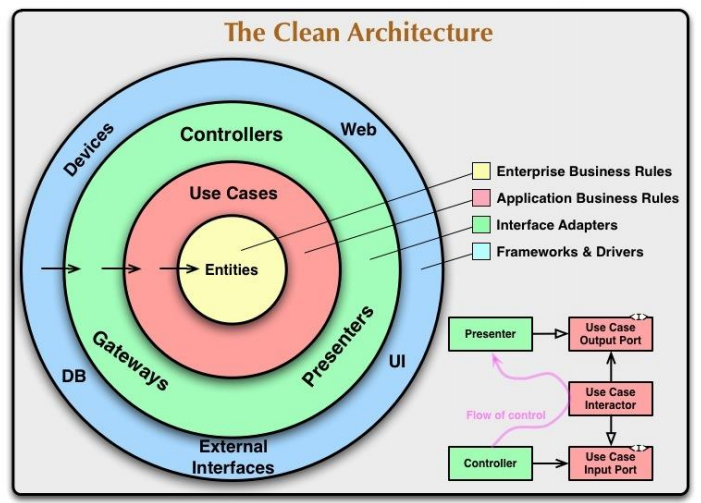
**The pipe and filter pattern：**提供一個處理資料流的架構，每一個處裡步驟被封裝在filter元件中，資料則是透過pipe在兩個相鄰的filter之間傳遞。透過重組filter可以建構出不同功能的系統。

Advantages：

1. 易於理解並支持transformation reuse
2. 工作流樣式與許多業務流程的結構相匹配。
3. 通過添加轉換來進行演化很簡單。

Disadvantages：數據傳輸的格式須達成共識。

**Clean architecture：**



要實作出 Clean Architecture，需要完成三項規則：

1. 分層規則：將系統依照重要順序以及離 IO 的遠近度來做分層。

| 層 | 職責 | 例子 |
| --- | --- | --- |
| Entity | 負責保管業務概念、業務狀態以及業務規則。本層是軟體的核心。不能依賴任何外部套件。 | 含有業務邏輯的 Class |
| Use Case | 定義軟體要完成的任務(使用案例)，並指揮 Entity 層的物件來實現業務邏輯的計算。 | 所有滿足使用者需求的使用案例 |
| Adapter | 將內外的資料轉換成合適的格式輸出或輸入。 | Controller, Gateway, Presenters |
| Framework & Driver | 為上面各層提供框架或技術能力 | 任何框架、Database、外部服務 |

1. 相依性規則：外圈是機制，內圈是策略。原始碼依賴關係只能指向內部，朝向更高層級的策略。內圈不能出現任何外圈的知識，包含變數、類別、函式。若內圈知道了外圈的知識，那只要外圈的元件有更動，那內圈相關的元件就得跟著修改。
2. 跨層原則：當物件跨層時(物件由內層向外層傳)，通常使用Dependency Inversion Principle來解決這種明顯的矛盾。圖的右下方是越過圓邊界的示例。它顯示了Controller和Presenter在下一層與Use case進行通信。它從Controller開始，遍歷Use case，然後在Presenter中結束執行。他們的依賴性都指向Use case。

**5 key activities in an object-oriented design processes：**

1. Define the context and modes of use of the system：

為了決定如何提供所需的系統功能以及如何建構系統以與其環境進行溝通，了解正在設計的軟體與其外部環境之間的關係是很重要的。了解context還可以使您確定系統的boundary。設置系統的boundary可幫助您確定正在設計的系統中實現了哪些功能以及在其他關聯系統中具有哪些功能。

系統context模型和interaction模型提供了系統與其環境之間關係的互補視圖，系統context模型是一種結構模型，用於演示正在開發的系統環境中的其他系統（以簡單的框圖表示），interaction模型是一種動態模型，它顯示了系統在使用時如何與其環境進行互動（用例模型表示）。

1. Design the system architecture：確定組成系統的components以及他們的interaction，然後使用architectural pattern組織這些components。
2. Identify the principal system objects：Identifying object classes通常是object oriented design的困難部分。從非正式系統描述中設計出的Object classes、attributes、operations可作為initial objects，再加上application domain knowledge或scenario analysis等信息(可以從需求文件、與用戶的討論、對現有系統的分析中收集)，即可refine and extend the initial objects。
3. Develop design models：設計模型顯示了objects、object classes、entities之間的關係。設計模型分為兩個種類：

Structural (static) models：根據object classes、relationships描述系統的靜態結構。

Dynamic models：描述objects之間的dynamic interactions。

Develop design models過程中的重要一步是確定所需的設計模型以及這些模型所需的詳細程度。

1. Specify object interfaces：必須Specify object interfaces，以便可以並行設計objects和其他components。Interface design 意味著定義object或a group of objects提供的signatures and semantics of the services。objects可能具有多個interfaces，這些interfaces是所提供的方法的觀點。UML使用class diagram進行interface specification，但也可以使用Java。

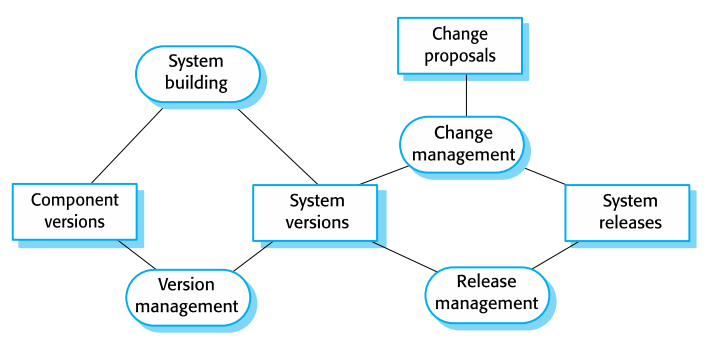
**SOLID：**

|  |  |  |  |
| --- | --- | --- | --- |
| **Initial** | **Stands for** | **Name** | **Concept** |
| **S** | **SRP** | Single Responsibility Principle | 一個類別只有一個單一目的、責任、變更的理由。 |
| **O** | **OCP** | Open Closed Principle | 一個類別應該對擴展開放，但對於修改封閉。 |
| **L** | **LSP** | Liskov substitution Principle | 子類別應該始終擴展其父類別，而不改變其行為。 |
| **I** | **ISP** | Interface Segregation Principle | 針對不同的客戶，只開放其所需要的介面讓他使用。 |
| **D** | **DIP** | Dependency Inversion Principle | 高層次的模組不應該依賴於低層次的模組，且兩者都應該依賴於抽象介面。  抽象介面不應該依賴於具體實現。而具體實現則應該依賴於抽象介面。 |

**Observer pattern：**

Description：將display of the state of object與object本身分開。當object狀態更改時，所有displays將自動被通知並更新以反映更改。

Problem description：This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

**configuration management (CM)：**

Version management：持續追蹤system components的多個版本，並確保不同開發人員對components所做的更改不會相互干擾。

System building：組裝program components、data、libraries，然後編譯它們以創建可執行系統。

Change management：持續追蹤客戶和開發人員對軟件更改的請求，確定更改的成本和影響，並決定是否更改。

Release management：準備software的release，並持續追蹤已發行供客戶使用的系統版本。

**Domain-Driven Design (DDD)：**

Domain：是應用程序邏輯所圍繞的知識和活動範圍。

DDD：是Domain概念的擴展和應用，因為它適用於軟體開發。它旨在通過將軟體的相關部分連接到一個不斷發展的模型中來簡化複雜應用程序的創建。

**building blocks of DDD**：

1. Entities：與傳統objects（由attributes定義）相反，該object由其一致的連續性consistent thread。
2. Value objects：具有attributes但不具有唯一identity的immutable (unchangeable)object。
3. Domain Event：一個object，用於記錄與系統內的模型活動相關的離散事件。儘管可以追蹤系統中的所有事件，但僅針對域專家關心的事件類型創建域事件。
4. Aggregate：在group中具有定義boundaries的一組entities和value objects。不是讓每個entities或value objects自己執行所有操作，而是為the collective aggregate of items分配了singular aggregate root item。
5. Services：本質上，service是一種不適合object領域的操作或業務邏輯形式。 換句話說，如果某些功能必須存在，但不能與Entities或Value objects相關，則可能是一項service。
6. Repositories：是一項使用global interface來提供對特定aggregate collection內的所有Entities和Value objects的訪問的服務。 應該定義方法以允許創建、修改、刪除aggregate中的objects。 但是，通過使用repository service進行數據查詢，其目的為從object模型的業務邏輯中移除此類數據查詢功能。
7. Factories：如design patterns中所述，DDD建議使用factory，factory封裝了創建複雜objects和aggregates的邏輯，從而確保客戶端不了解objects的內部工作。

**Program testing goals：**

1. 向開發人員和客戶證明該軟體符合其需求。.
2. 發現軟體行為不正確或不合需要或不符合其規格的情況。

**Validation testing：**

* 1. 向開發人員和系統客戶證明軟件符合其要求
  2. 測試成功表示系統按預期運行。

**Defect testing：**

1. 在軟體行為不正確或不符合規範的情況下發現軟體中的故障或缺陷
2. 測試成功代表系統無法正常運行並因此暴露系統缺陷。

**Verification vs validation (V&V)：**

Verification：該軟體應符合其specification。

Validation：該軟體應滿足用戶的實際需求。

目的為建立的系統是適合目標的信心。

**Stages of testing：**

Development testing：包含由開發系統的團隊執行的所有測試活動。

1. Unit testing：測試program units或object classes。單元測試應著重於測試objects或methods的功能。
2. Component testing：為多個individual units集成以創建components。Component testing應著重於測試component interfaces。
3. System testing：整合系統中的某些或所有組件，並對系統進行了整體測試。System testing應著重於測試component interactions。

Release testing：測試打算在開發團隊之外使用的系統，目的是說服系統供應商使用它。Ex：Scenario testing、Performance testing

User testing：用戶或客戶提供有關系統測試的意見和建議。Ex：Alpha testing、Beta testing、Acceptance testing

**Acceptance testing：**客戶對系統進行測試，以確認是否準備好被系統開發人員接受並部屬在客戶環境中。**Stages in the acceptance testing process**：Define acceptance criteria、Plan acceptance testing、Derive acceptance tests、Run acceptance tests、Negotiate test results、Reject/accept system。

**Black-box testing technique：**測試案例的設計是依靠系統的輸入與輸出行為資訊

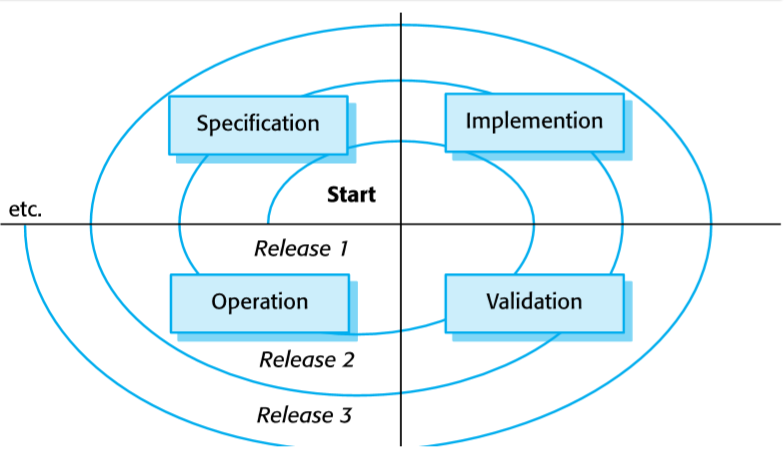
**White-box testing technique：**測試案例的設計是依靠程式設計與程式碼資訊

**Test-driven development (TDD)：**強調先寫測試再寫程式。TDD幫助programmer闡明他們對code實際上應該做什麼的想法。

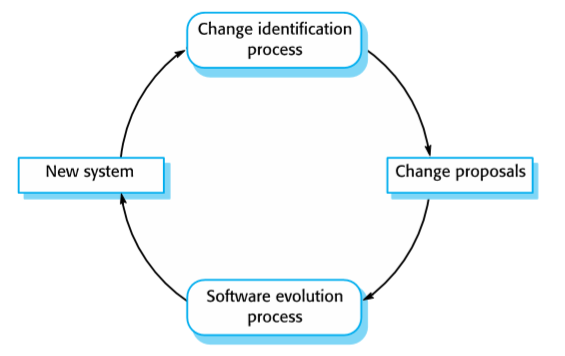
**Benefits of test-driven development：**Code coverage、Regression testing、Simplified debugging、System documentation

**Regression testing：**regression test隨著program的開發而逐步開發。始終可以運行regression test來檢查程序更改是否未引入新的錯誤。

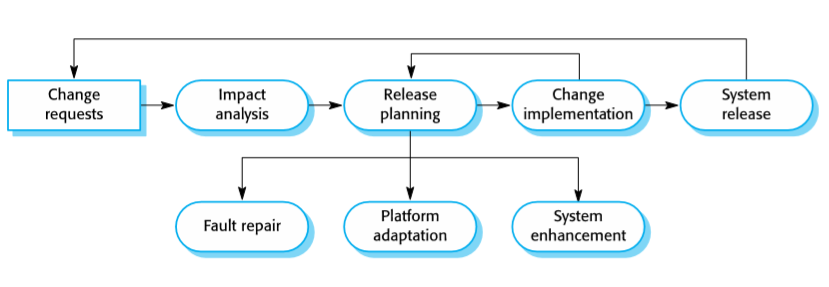
**A spiral model of development and evolution：**



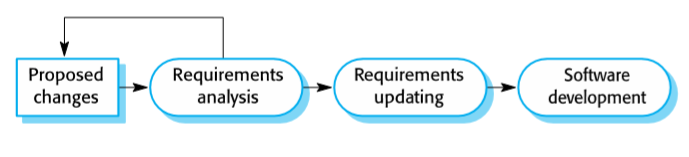
**Change identification and evolution processes：**



**The software evolution process：**



**Change implementation：**



**Legacy systems：**使用過時的軟體和硬體技術開發的較舊的軟體系統，對企業仍然有用。

**System quality assessment：**Business process assessment、Environment assessment、Application assessment

**Business process assessment：**使用viewpoint-oriented的方法並從stakeholders那裡尋求答案

**Types of maintenance：**

1. Fault repairs：為Corrective maintenance，更改系統以解決bugs / vulnerabilities並以符合其要求的方式糾正deficiencies。
2. Environmental adaptation：為Adaptive maintenance，藉由維護使軟體能適應不同的操作環境 (computer、OS等)。
3. Functionality addition and modification：為Perfective maintenance，修改系統以滿足新需求。

**Maintenance cost factors：**Team stability、Contractual responsibility、Staff skills、Program age and structure

**Advantages of reengineering：**Reduced risk、Reduced cost

**Reengineering process activities：**Source code translation、Reverse engineering、Program structure improvement、Program modularisation、Data reengineering

**Refactoring and reengineering：**兩者都是為了使軟件更易於理解和更改。

Refactor是在整個開發和演化過程中不斷改進的過程。旨在避免structure和code降級，從而降低維護系統的成本和難度。為preventative maintenance。

Reengineering是在系統維護了一段時間並且維護成本不斷增加之後進行的。可以使用自動化工具來處理和重新設計舊系統，以創建更易於維護的新系統。

**Bad smells：**相同功能的實作上，可有各種程式碼的組織方式，而這些組織方式有令程式碼難以維護的特徵，此特徵即為bad smells，影響程式碼品質、設計。