|  |
| --- |
| 模式名稱 |
| 狀態模式State Pattern |
| 使用情境 |
| 在不同狀態下，外部使用者不需改變使用方式，就可達成各種狀態的變換  避免if-else和switch不好維護的問題。  例如外部使用者呼叫 英雄.攻擊();  英雄類別的狀態，如果是普通狀態，將使用普通攻擊；如果是憤怒狀態，將使用重擊，外部使用者不須更改寫法 |
| 使用方法 |
| 至少會有三個class  Context：供外部呼叫和切換狀態的class  State：供Context呼叫的介面或interface，會有個欄位儲存Context的參考，實作時才能找到要變換狀態的目標  ConcreteState：實作該State要做的事 |

|  |
| --- |
| 模式名稱 |
| 外觀模式Facade Pattern |
| 使用情境 |
| 假設一個駕駛要駕駛一輛汽車，踩下油門會牽動 引擎、傳動裝置、儀表板等複雜的系統，這些系統應該包裝成一個被整合過的類別(Facade)讓使用者使用，以降低耦合度及出現BUG的機會  使用者踩下油門應該是 油門.踩下();  而非  引擎.運作();  傳動裝置.運作();  儀錶板.運作();  這樣會使客戶端的程式碼過於複雜，如果任意調換程式順序可能導致嚴重問題，且會有高耦合度，造成程式碼難以維護  綜上，有下列好處：   1. 易於切分程式，分工開發 2. 增加對系統的安全性(客戶端無法隨意更動運作順序或程式碼) 3. 降低程式碼與客戶端的耦合度 |
| 使用方法 |
| 把  引擎.運作();  傳動裝置.運作();  儀錶板.運作();  寫進油門類別  讓油門.踩下()  執行以上程式碼，如此一來，使用者只要執行油門.踩下()即可 |

|  |
| --- |
| 模式名稱 |
| 單例模式Singleton Pattern |
| 使用情境 |
| 軟體設計上有一些物件有唯一性的需求  例如：伺服器端的程式只能連接到一個資料庫、只能有一個日誌產生器、遊戲只能同時有一個關卡正在進行、只能操作一個角色等 |
| 使用方法 |
| 需要程式語言的支援，必須具備下列條件的語言才可使用：   1. 靜態類別屬性 2. 靜態類別方法 3. 重新定義類別建構者存取層級(將建構子設為private)   用靜態方法回傳物件，假如不存在就產生一個  建構子用private層級使其無法初始化，以免跑出多個物件  連線類型的遊戲，可用單例來限制連線數  注意：不要過度使用單例，因為它有點像全域變數，過多會造成濫用，要仔細規劃 |