Code Structure View via UML Class Diagram

Shin-Jie Lee (李信杰)
Associate Professor
Dept. of CSIE
National Cheng Kung University
v2024-10-03

為何Code Structure View重要?

- 現實中常遇到需**理解既有程式碼(Legacy Code)**，但又缺乏設計文件，直接trace code常是件不容易的事
- 「Trace code」可說是個逆向工程過程，以理解系統的
  - Structure (靜態結構)
  - Behavior (動態行為)
- Code-to-UML Class Diagram這類的自動化同步工具可協助建構Structural View (即時且不會過時)，作為理解Behavior的基礎

利用ObjectAid來建構Code Structure View

- ObjectAid為Java-to-UML免費工具，安裝步驟請參考先前教材
- 此教材介紹如何利用ObjectAid來建構Java code structure view
- 實際專案可依程式語言、IDE工具來選擇專業的Code-to-Diagram自動化同步工具

首先，先熟練

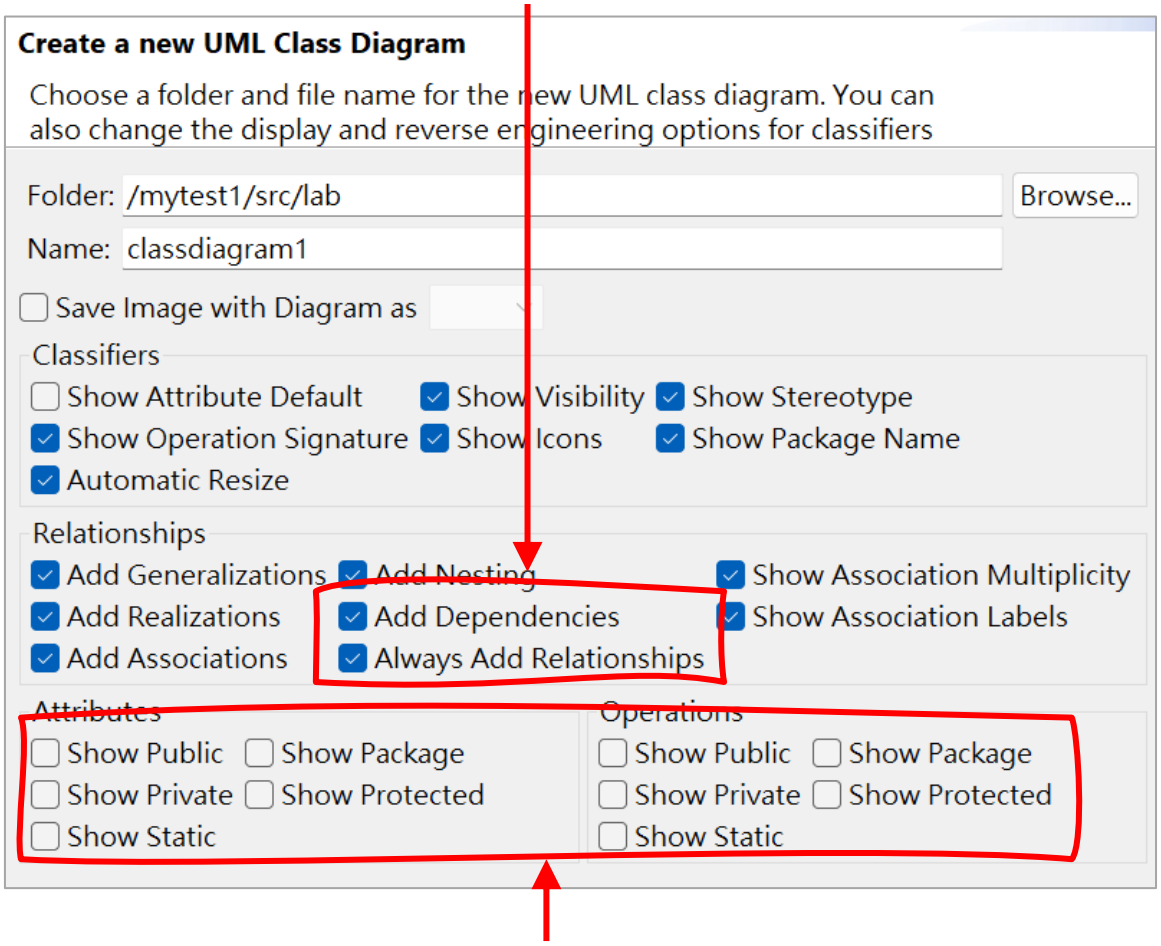
ObjectAid的起手式三步驟

ObjectAid的起手式三步驟

1. 設定New class diagram
2. 更新遺漏的dependency
3. 更新layout

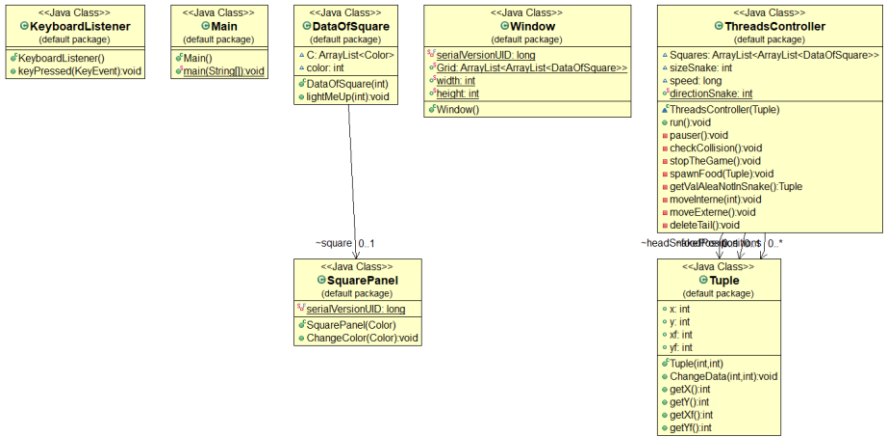
步驟1：設定New Class Diagram$_1$

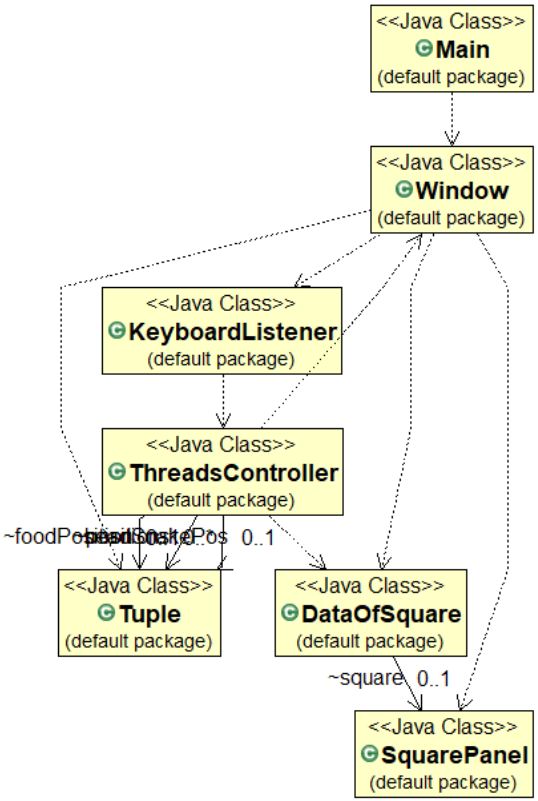
新增class diagram時，建議將此處打勾以呈現dependency虛線



建議將此處取消打勾，可先只聚焦在class name間的關係

步驟1：設定New Class Diagram$_2$



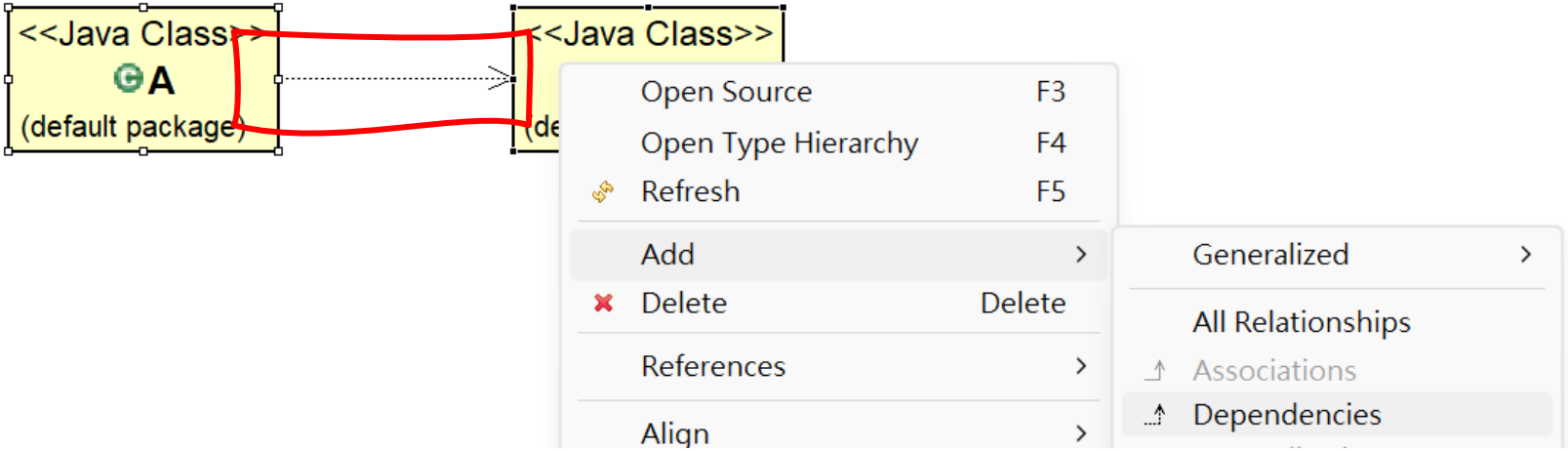


原本的設定較無法看出
class間的關係

調整後的設定較可看出
class間所有的關係

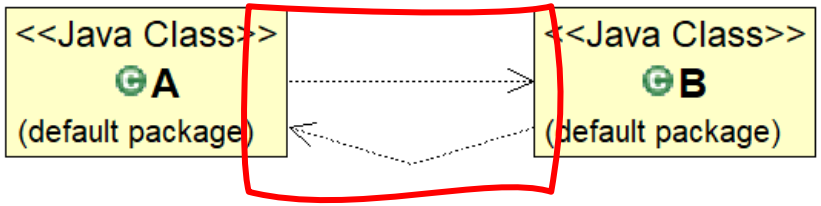
步驟2：更新遺漏的Dependency$_1$

原本應為雙向



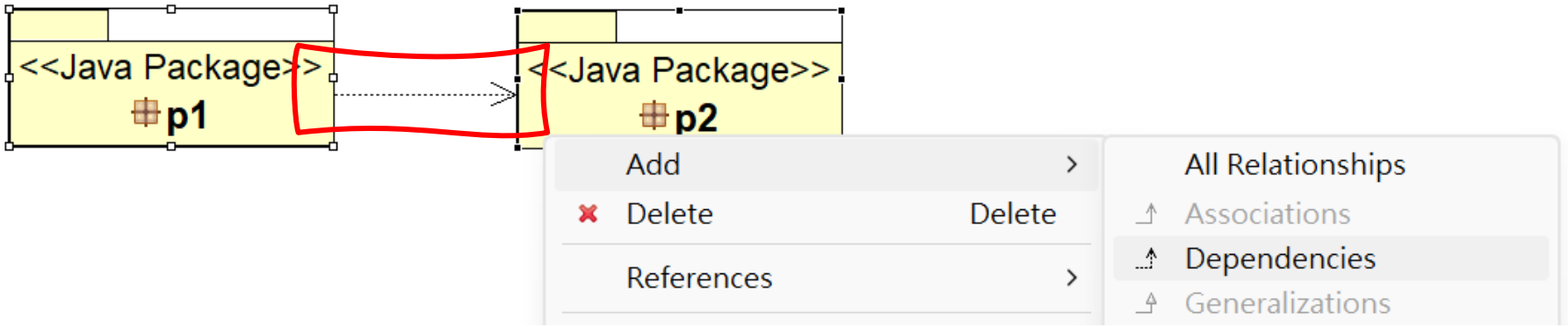
有時dependency會遺漏(或許為ObjectAid的bug)，修正如下【Ctrl + a 全選】➔【在任一class上右鍵】➔【Add】➔【Dependencies】

更新後即可正常顯示為雙向



步驟2：更新遺漏的Dependency$_2$

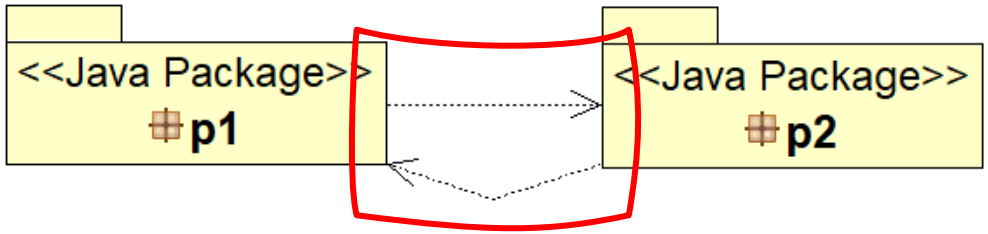
原本應為雙向



Package間的dependency也會被遺漏，【Ctrl+a 全選】➔【在任一Package上右鍵】➔【Add】➔【Dependencies】

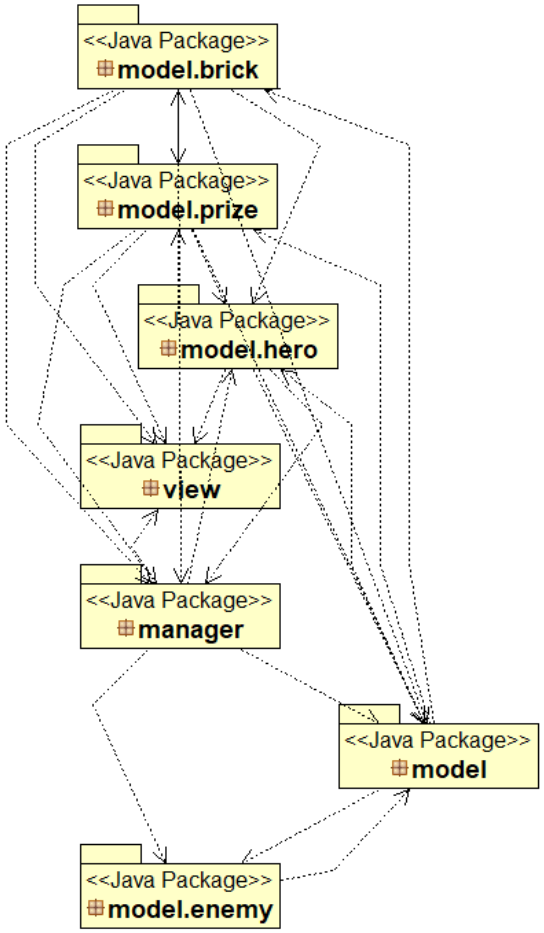
註：Package A若存在一個class參考到package B中一個class，即A depends on B

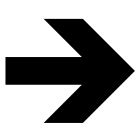
更新後即可正常顯示為雙向

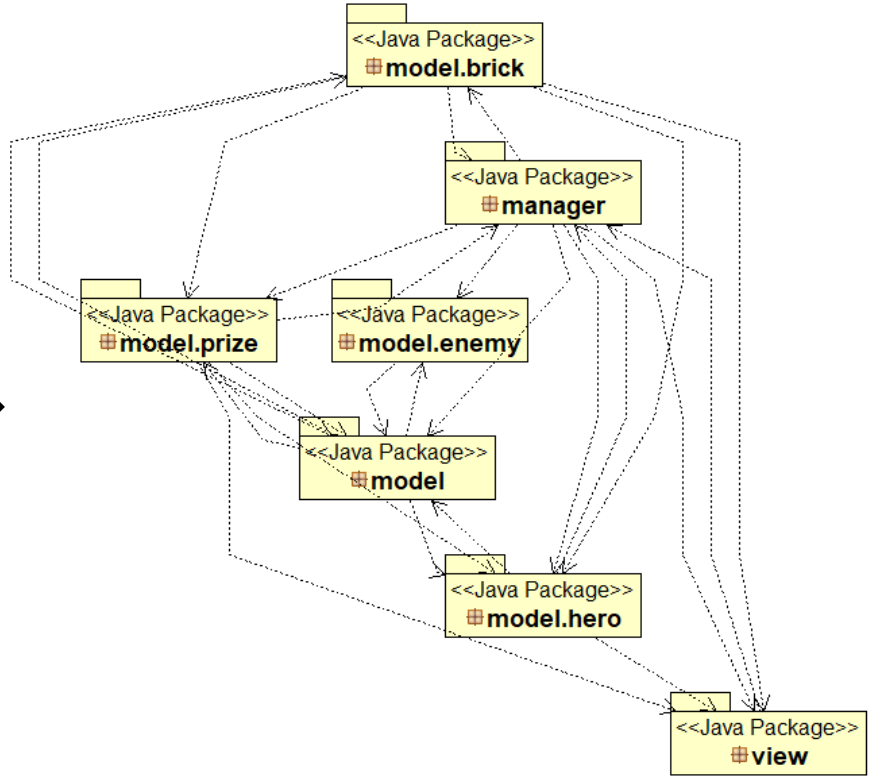


步驟3：更新Layout

接續上步驟後可在【畫面右鍵】➔【Layout Diagram】來更新Layout







上步驟dependency更新後有時會顯得凌亂

Layout更新後會較為整齊，接著可再手動調整

進入主題

Code Structure View

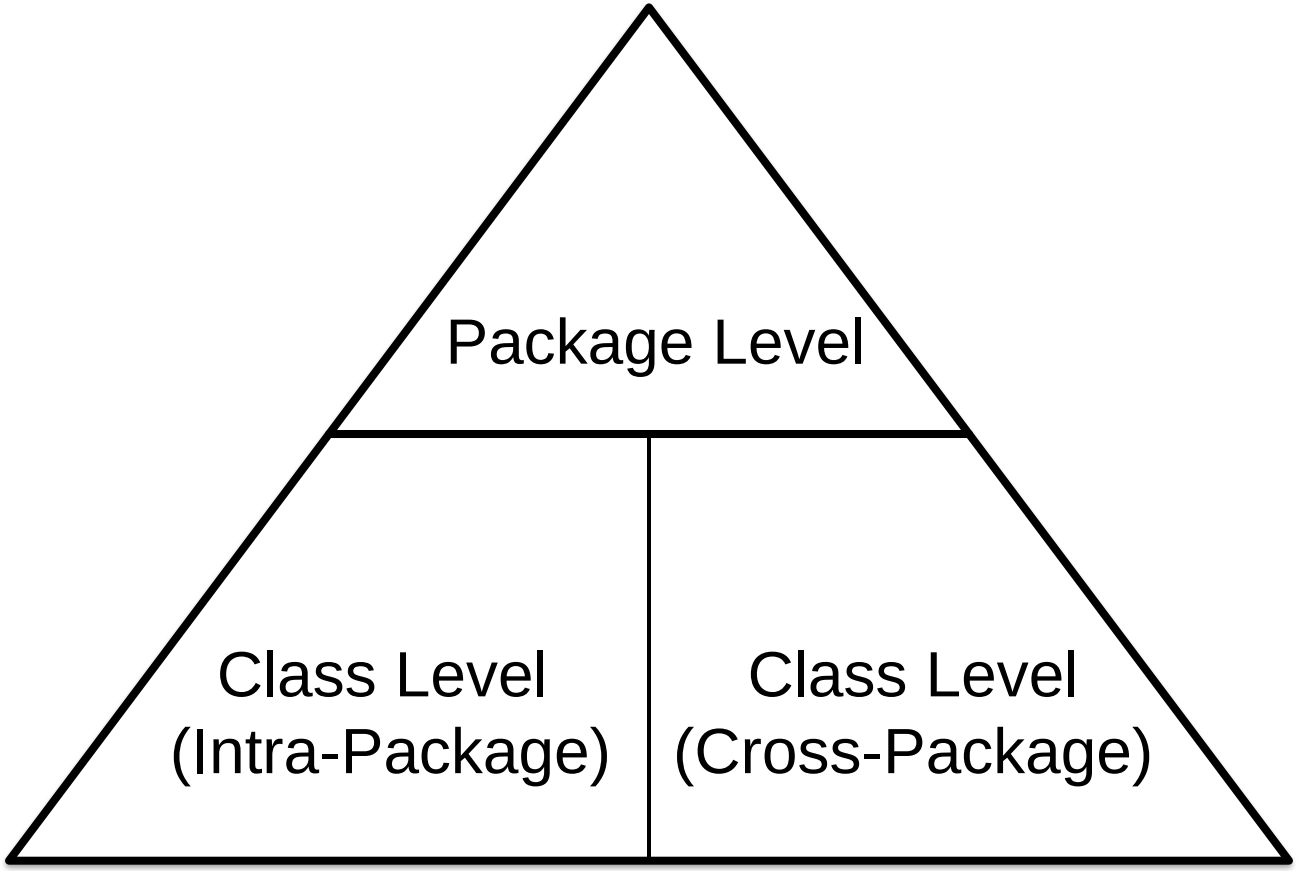
Levels of Abstraction

- Trace code 類似於在地圖上理解從 A 點到 B 點路徑的過程，反覆地進行 zoom in（查看局部細節）和 zoom out（把握全局視角），漸進地理解系統的運行方式。
  - 路徑 ➔ 系統動態Behavior
  - 地圖 ➔ 系統靜態Structure

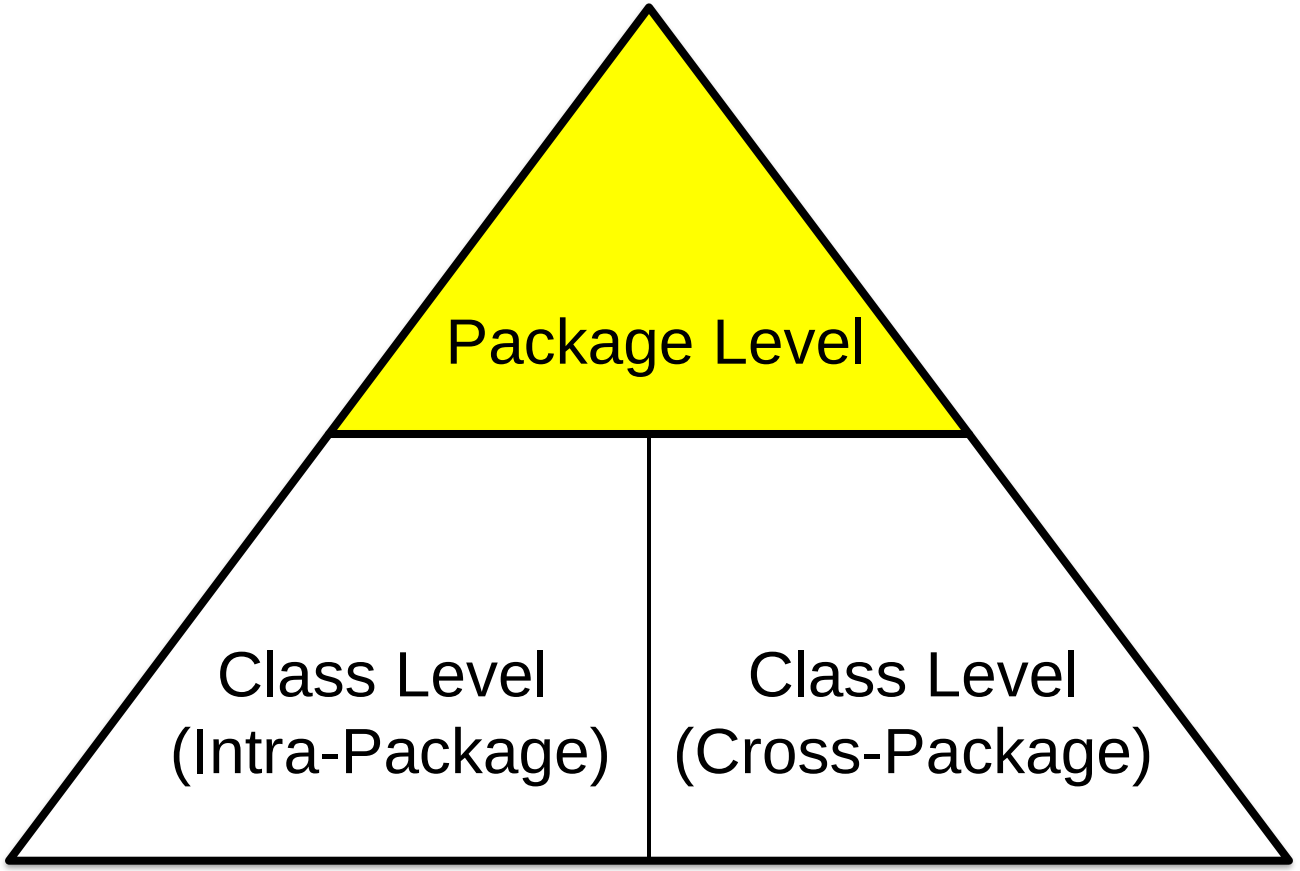
Levels of Abstraction in Maps

Levels of Abstraction (zoom in/out)

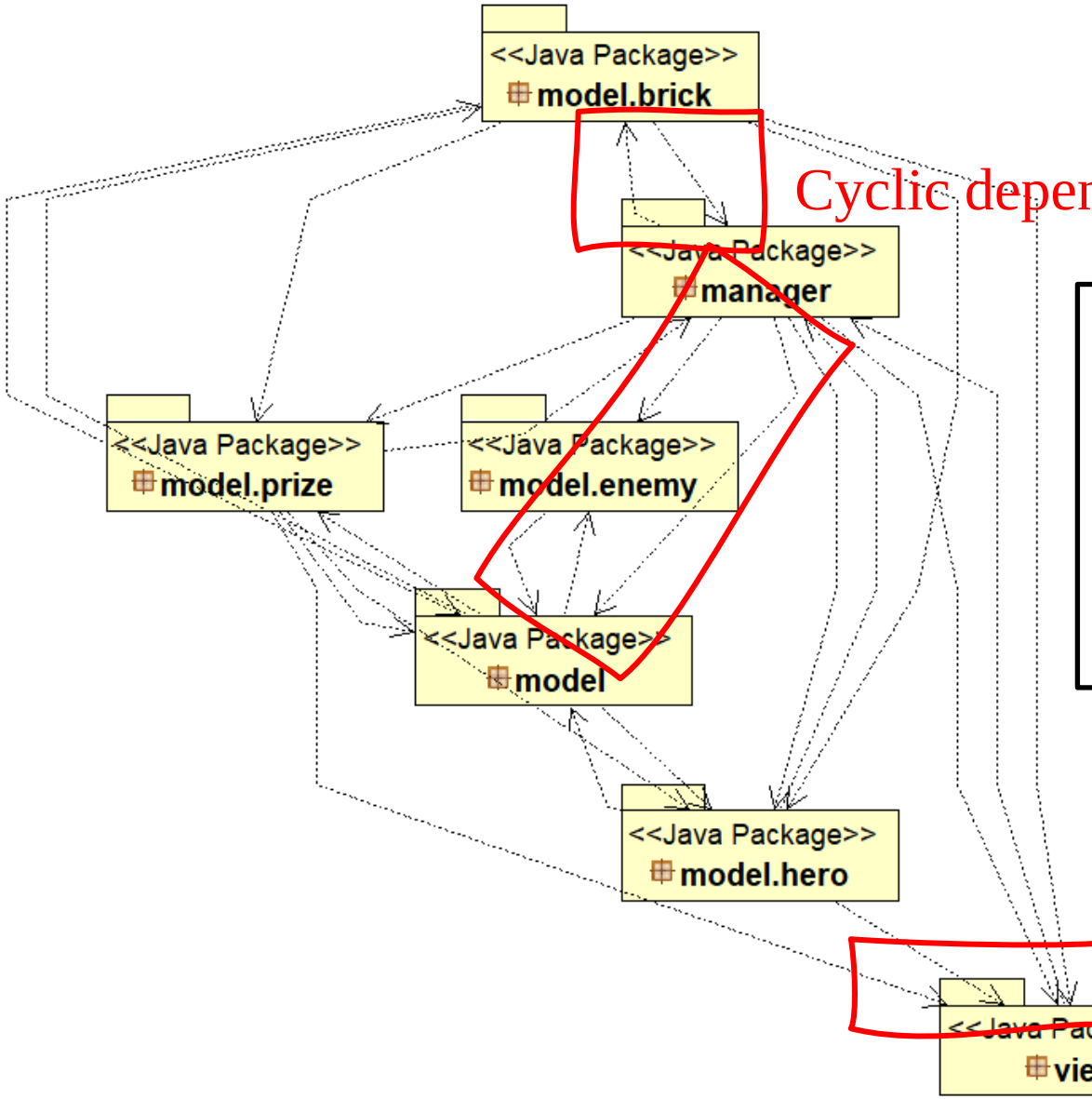
Levels of Abstraction in Java Code Structure



Levels of Abstraction



Package Level

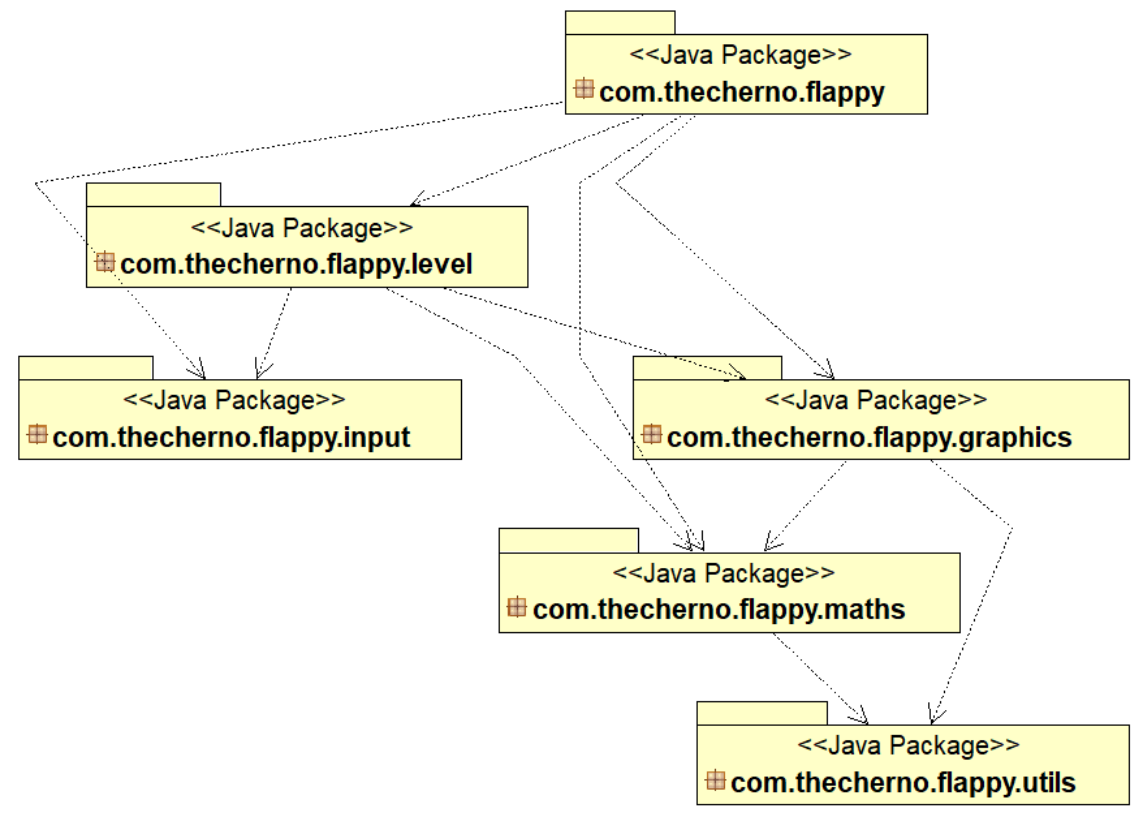


觀察重點

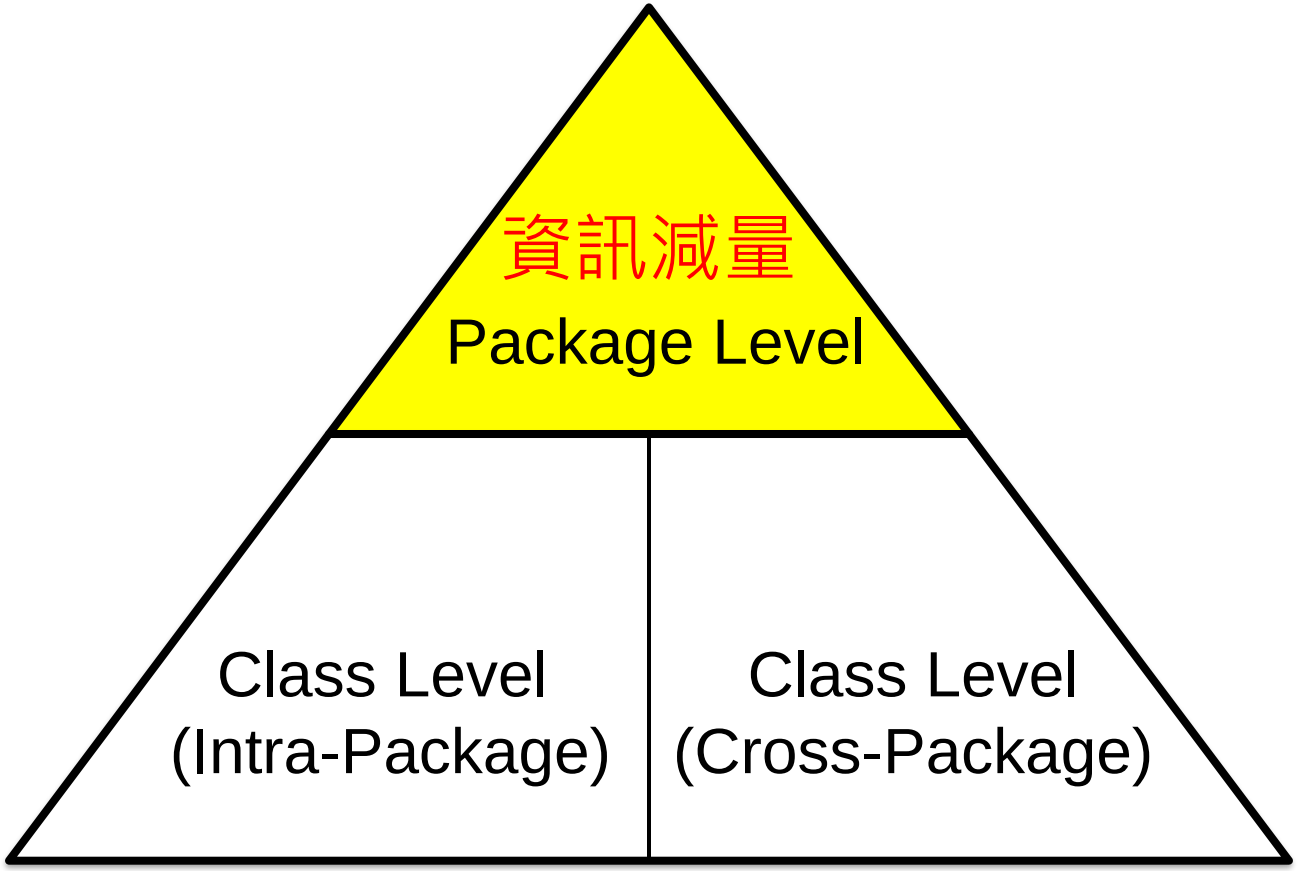
Package間若為單向依賴且無循環依賴，則耦合度較低，可提高維護性與延展性。

Lab (Flappy)

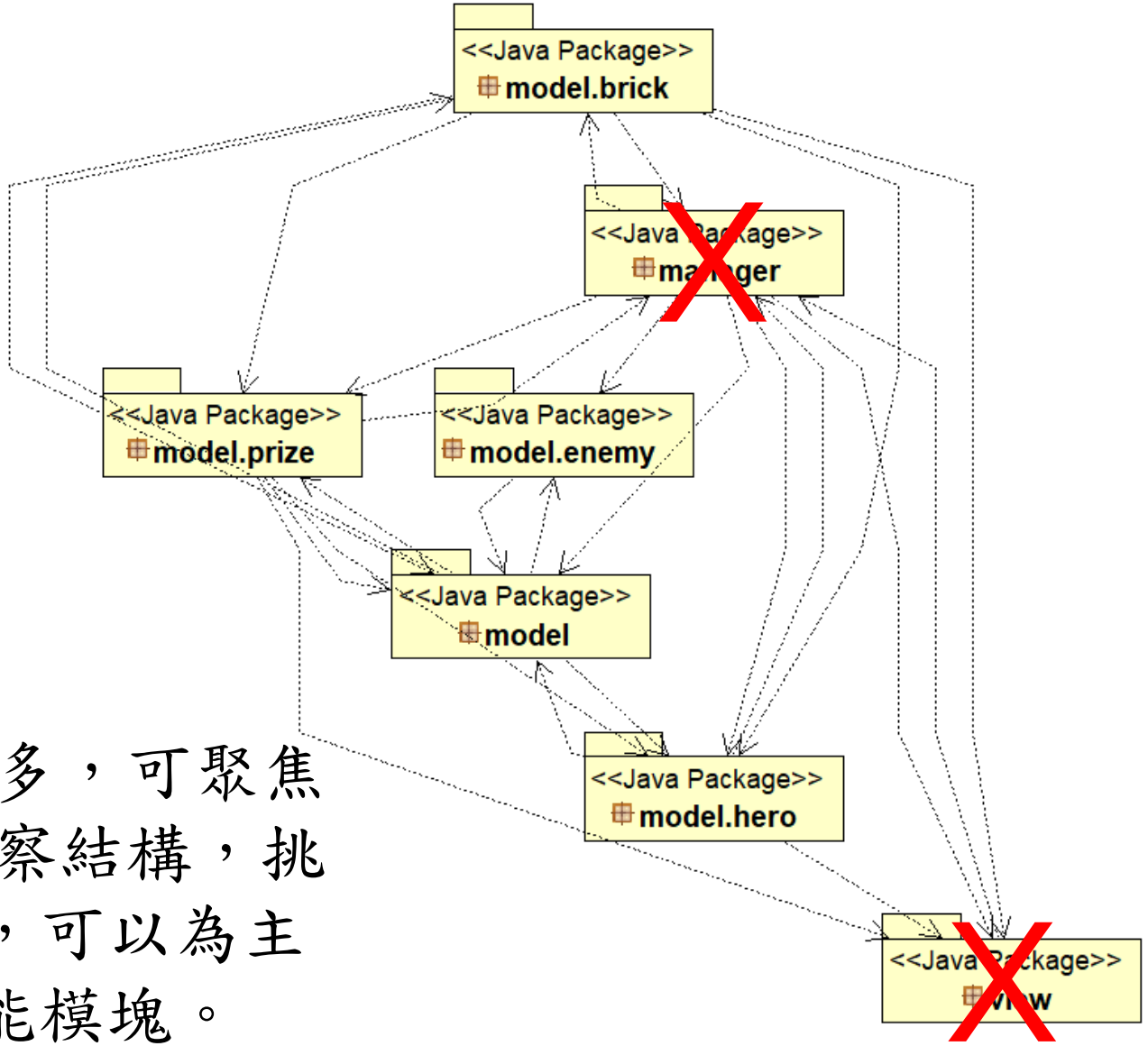
- 請繪製出以下project的package-level diagram，並識別是否存在cyclic dependencies
  - https://github.com/TheCherno/Flappy.git



Levels of Abstraction

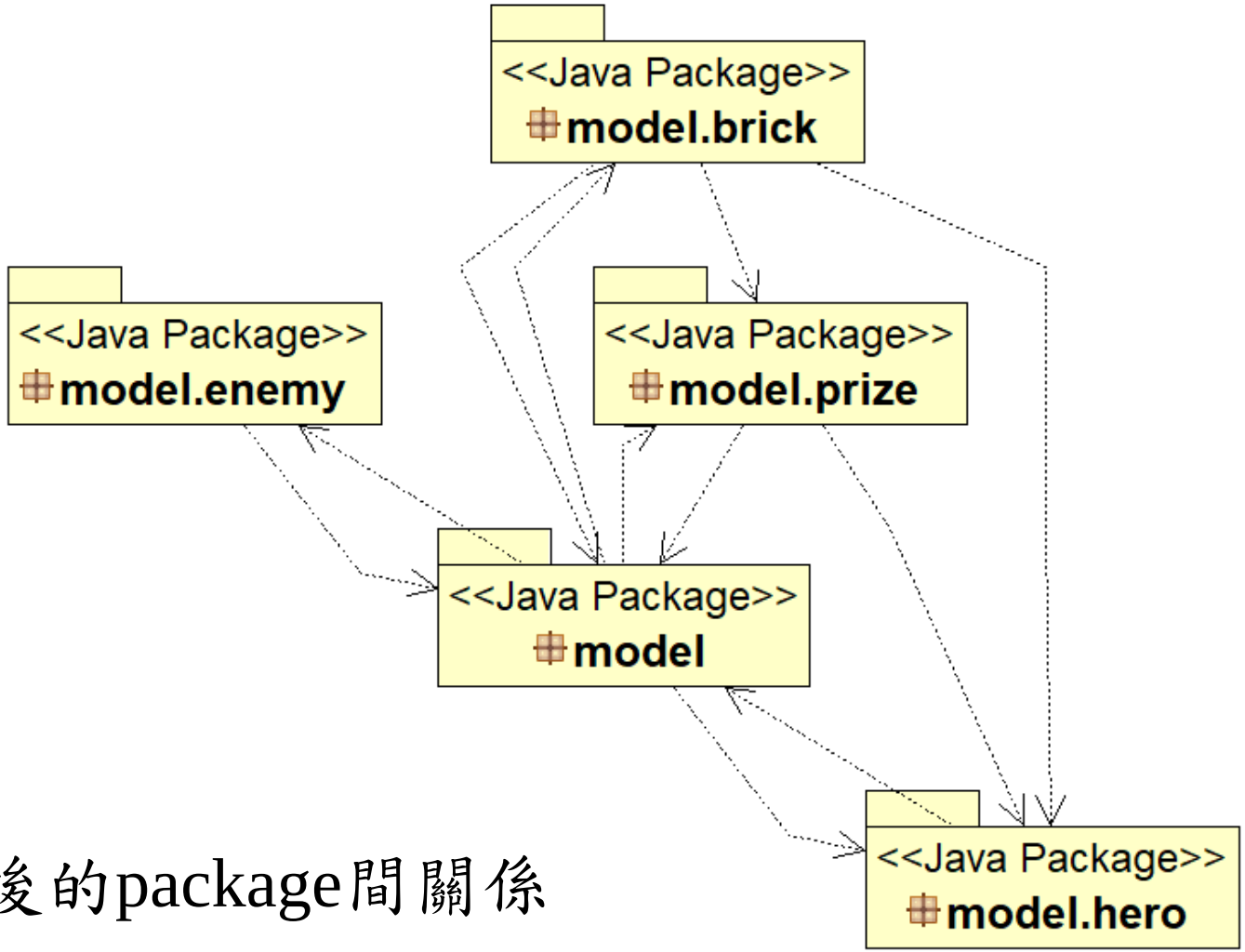


Package Level (減量)$_1$



若package數量太多，可聚焦部分package來觀察結構，挑選準則沒有標準，可以為主業務邏輯或某功能模塊。

Package Level (減量)$_2$

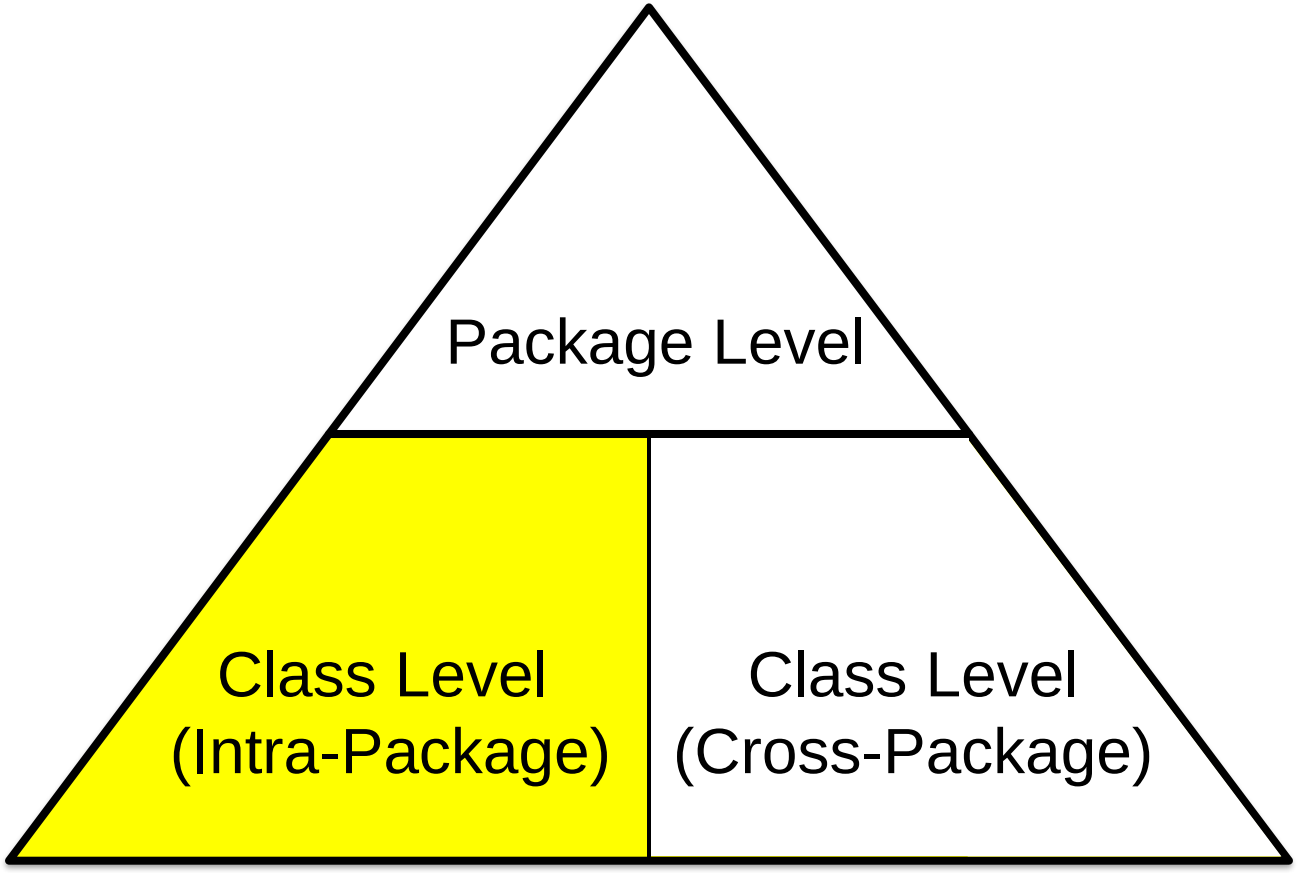


聚焦後的package間關係

Lab (Mario)

- 請繪製出上頁聚焦後的package間關係
  - https://github.com/ahmetcandiroglu/Super-Mario-Bros.git

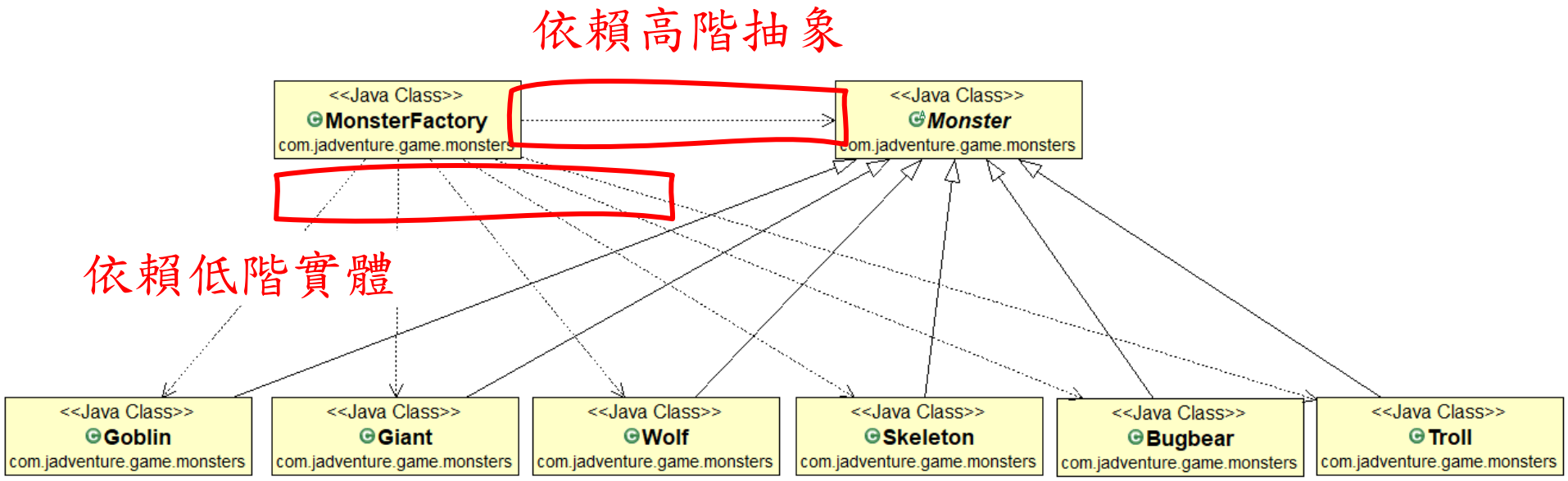
Levels of Abstraction



Class Level (Intra-Package)$_1$

觀察重點1

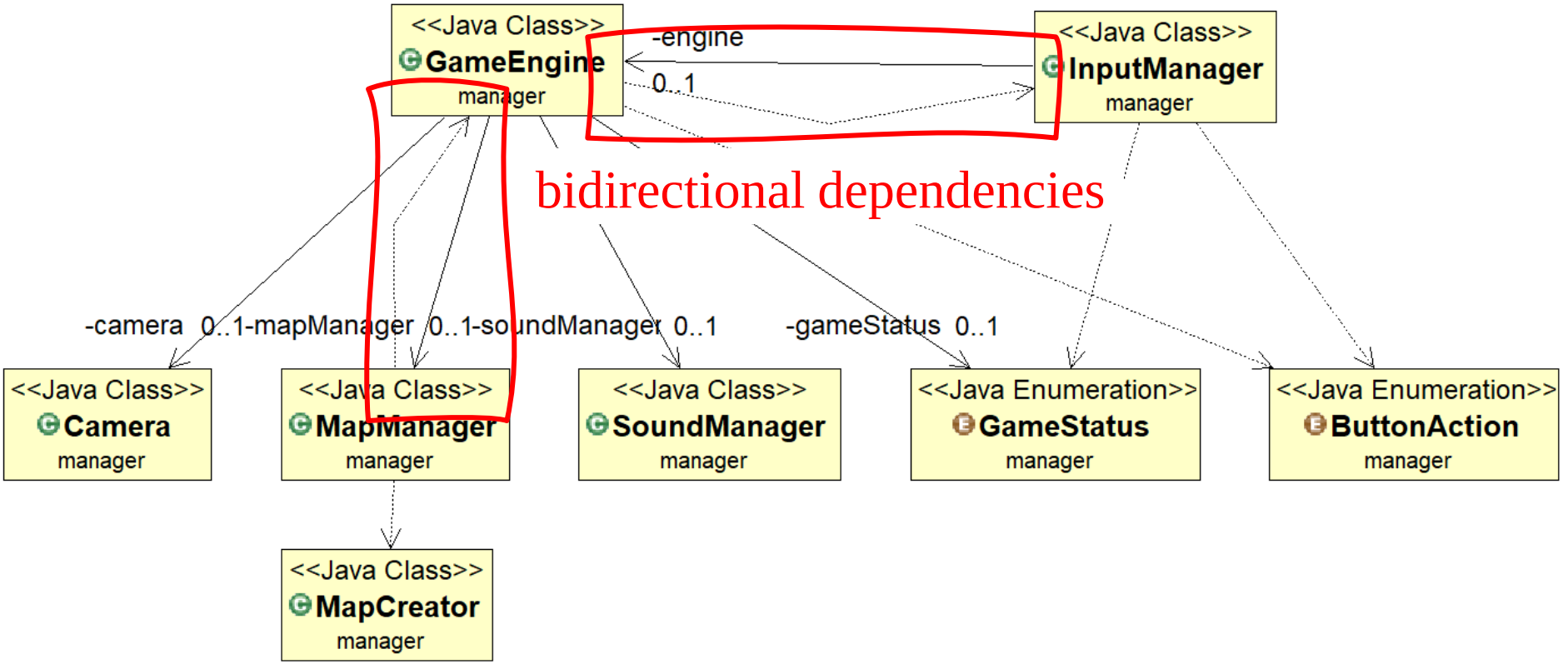
可關注依賴於高階抽象(interfaces、abstract classes)或低階實體(sub-classes)，在設計原則權衡下評估是否合適。



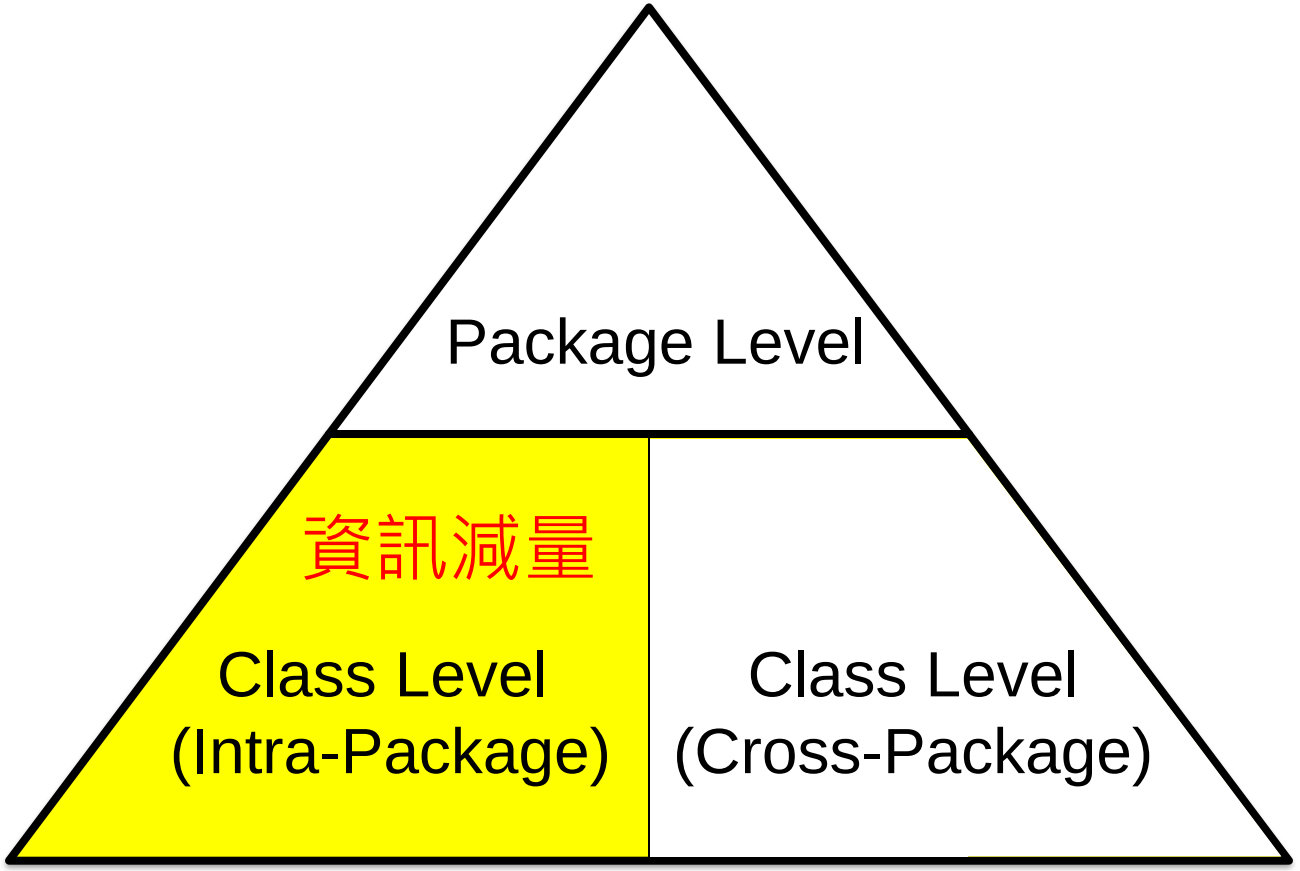
Class Level (Intra-Package)$_2$

觀察重點2

可關注bidirectional dependencies，在設計原則權衡下評估是否合適。

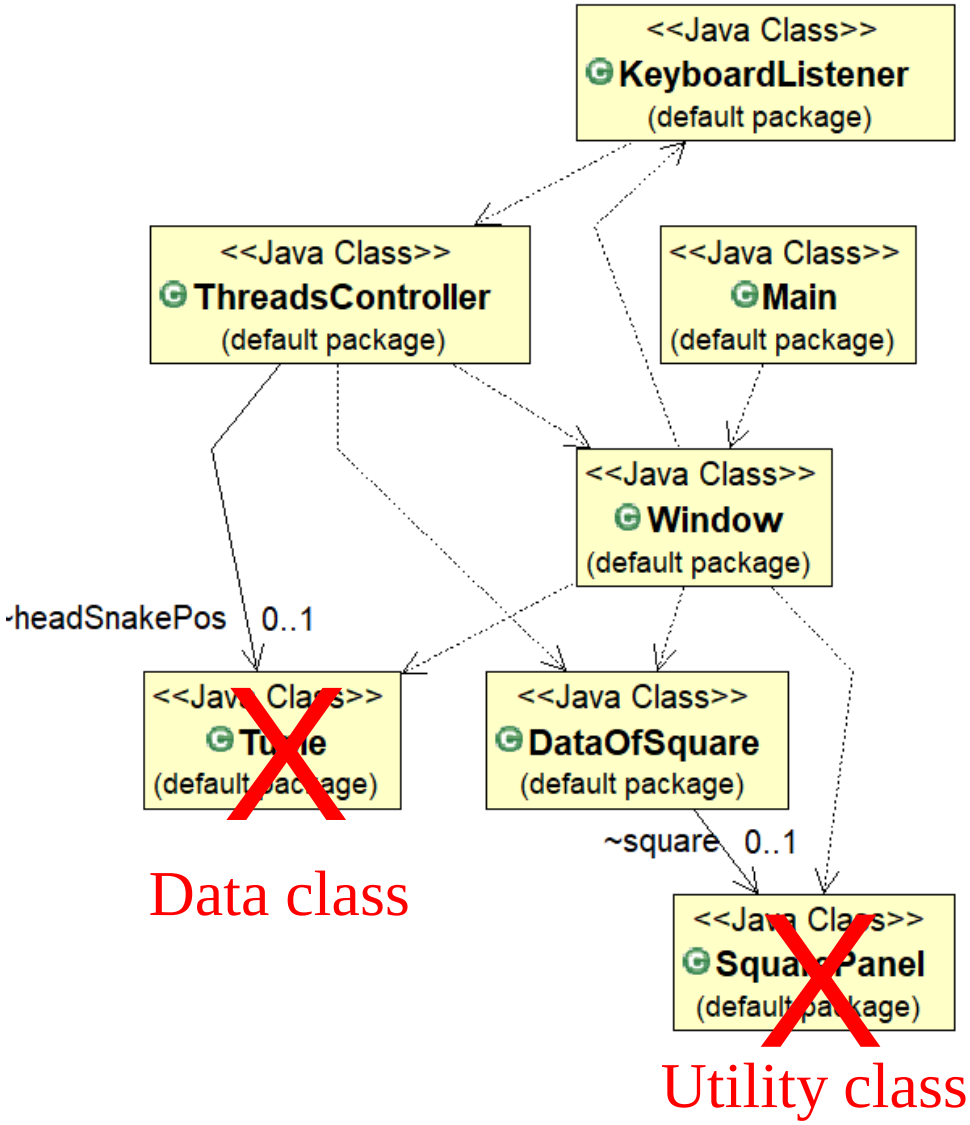


Levels of Abstraction



Class Level (Intra-Package) (減量)

- 可先聚焦在核心業務的classes，隱藏次重要的classes，例如data classes、utility classes、exception classes、enums、composition root、UI-layer classes
- 可隱藏dependencies，只顯示強烈關係(inheritance, implementation與association)

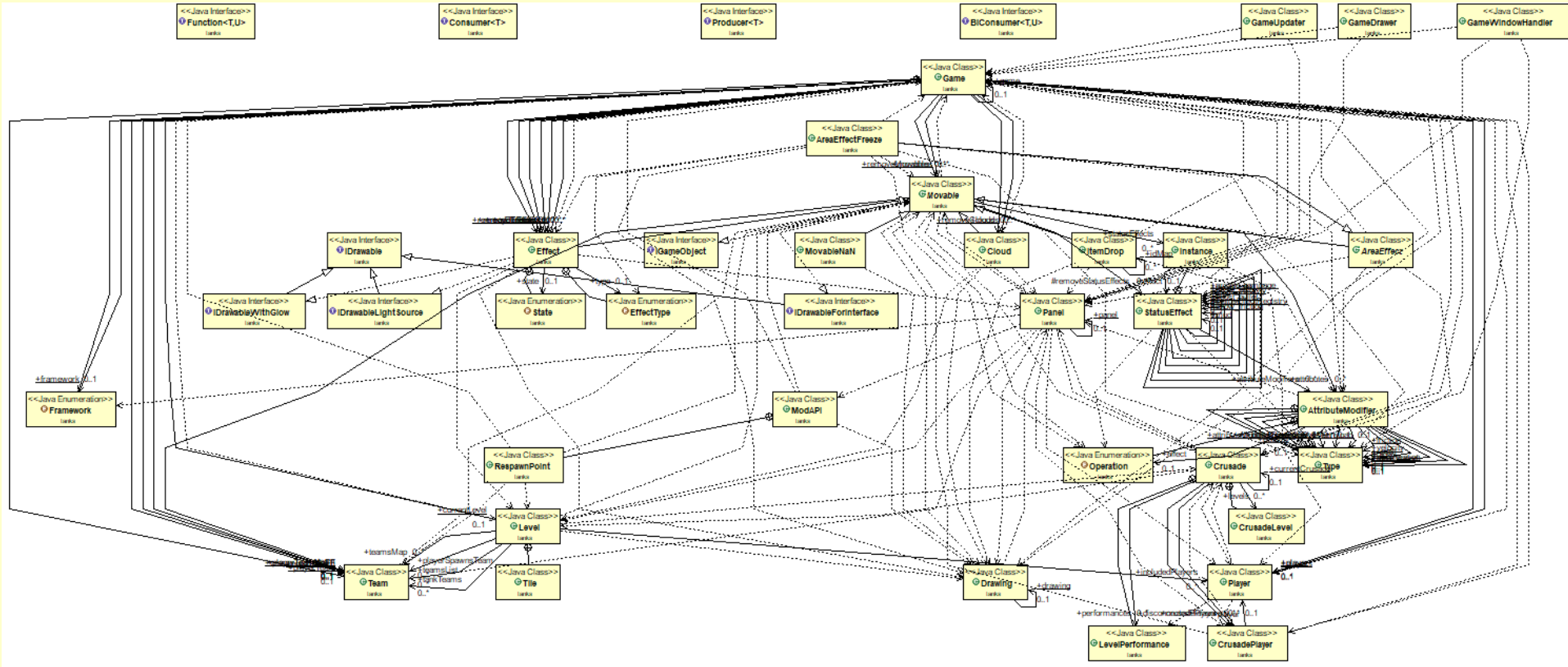


Lab (Tank)

- 請依據以下步驟畫出下面project中package `tank`內的class diagram
  - https://github.com/aehmttw/Tanks.git

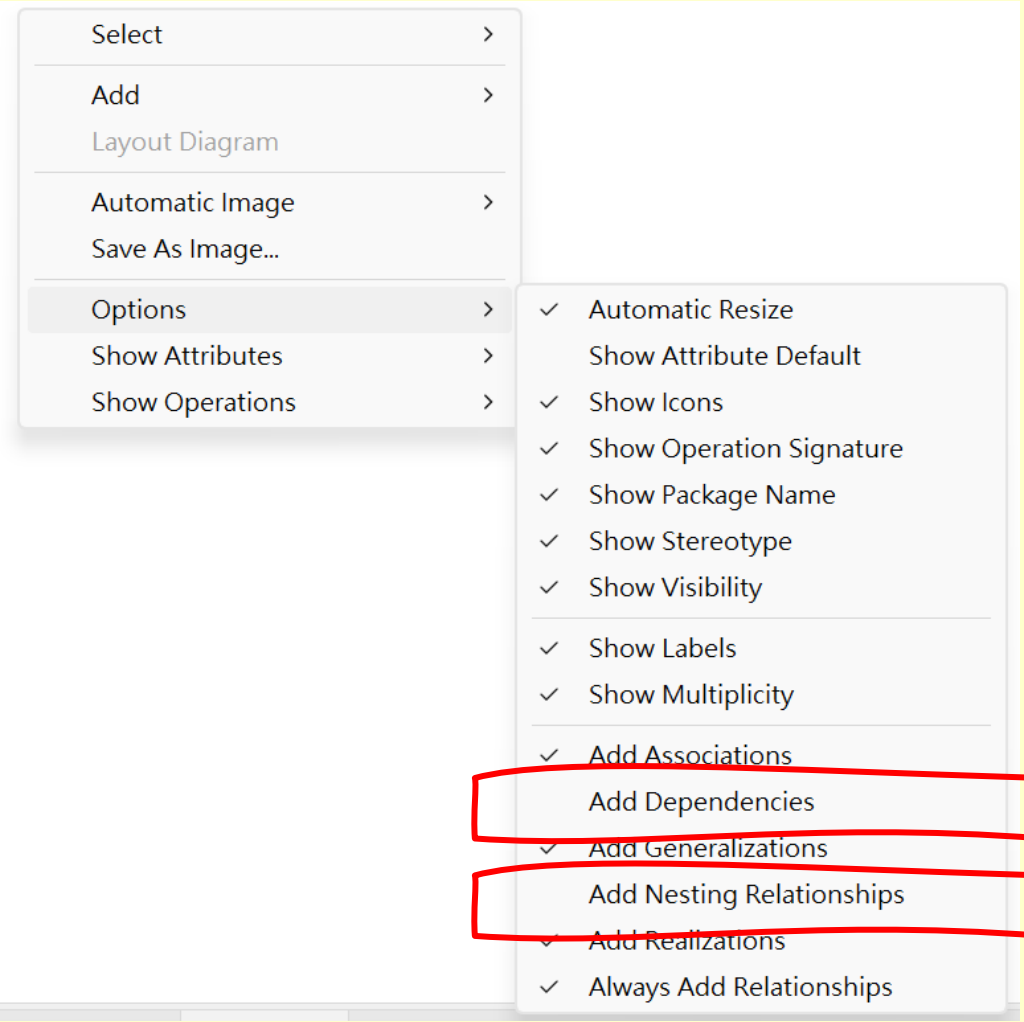
Lab (Tank)

- 步驟1：依據【起手式三步驟】畫出第一個版本的class diagram，結果layout會顯得相當複雜



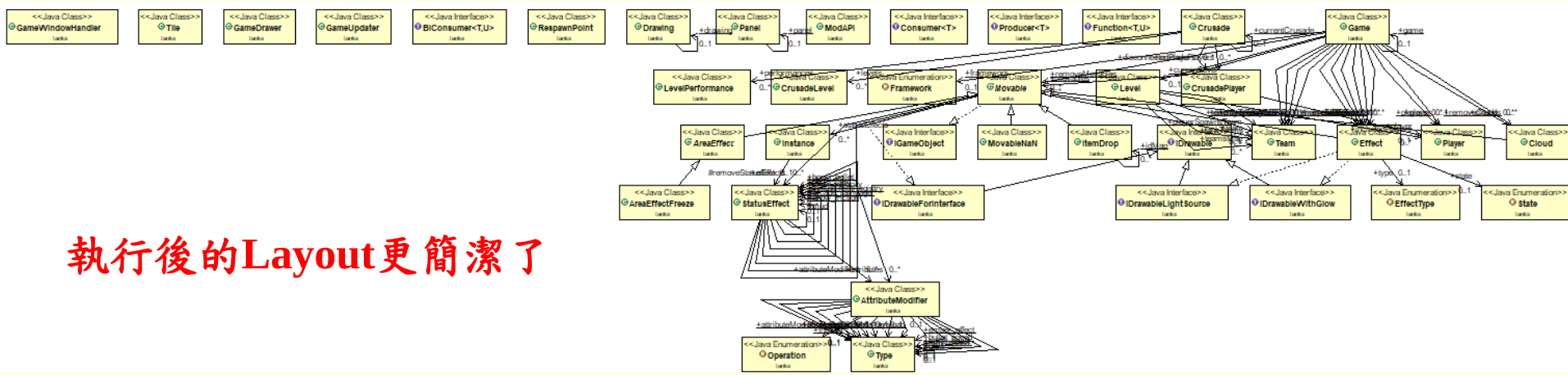
Lab (Tank)

- 步驟2：將diagram所有classes移除後，按右鍵取消打勾【Add Dependencies】與【Add Nesting Relationships】



Lab (Tank)

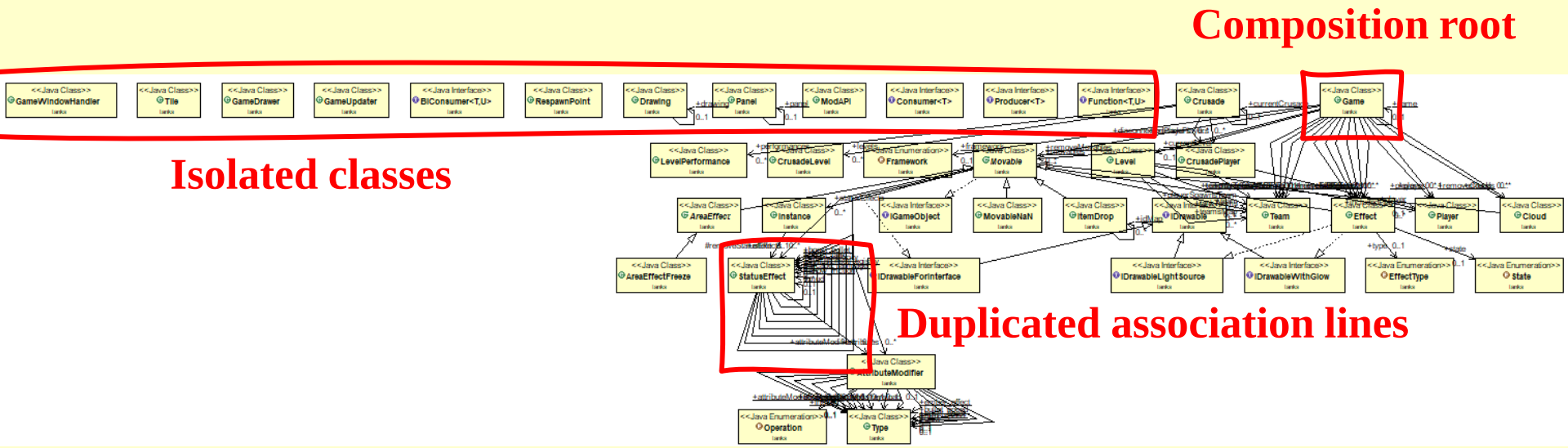
- 步驟3：再一次將package tank內所有的classes移入diagram，然後右鍵再【Layout Diagram】一次，即會只顯示較強烈的關係。



執行後的Layout更簡潔了

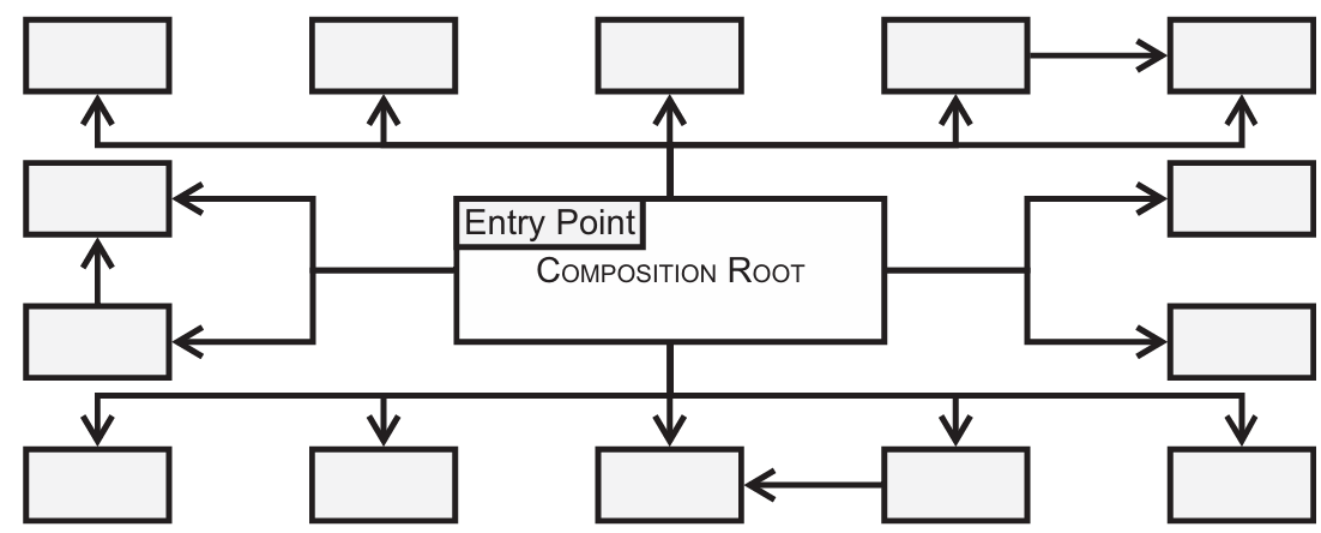
Lab (Tank)

- 步驟4：移除以下內容
  - Isolated classes (獨立的classes)
  - 重複的association lines (兩個class間的association一條即可)
  - Game class (此class與太多classes有關連，很可能為 **composition root**，可先移除，聚焦在核心classes)



Tips – 何謂Composition Root?

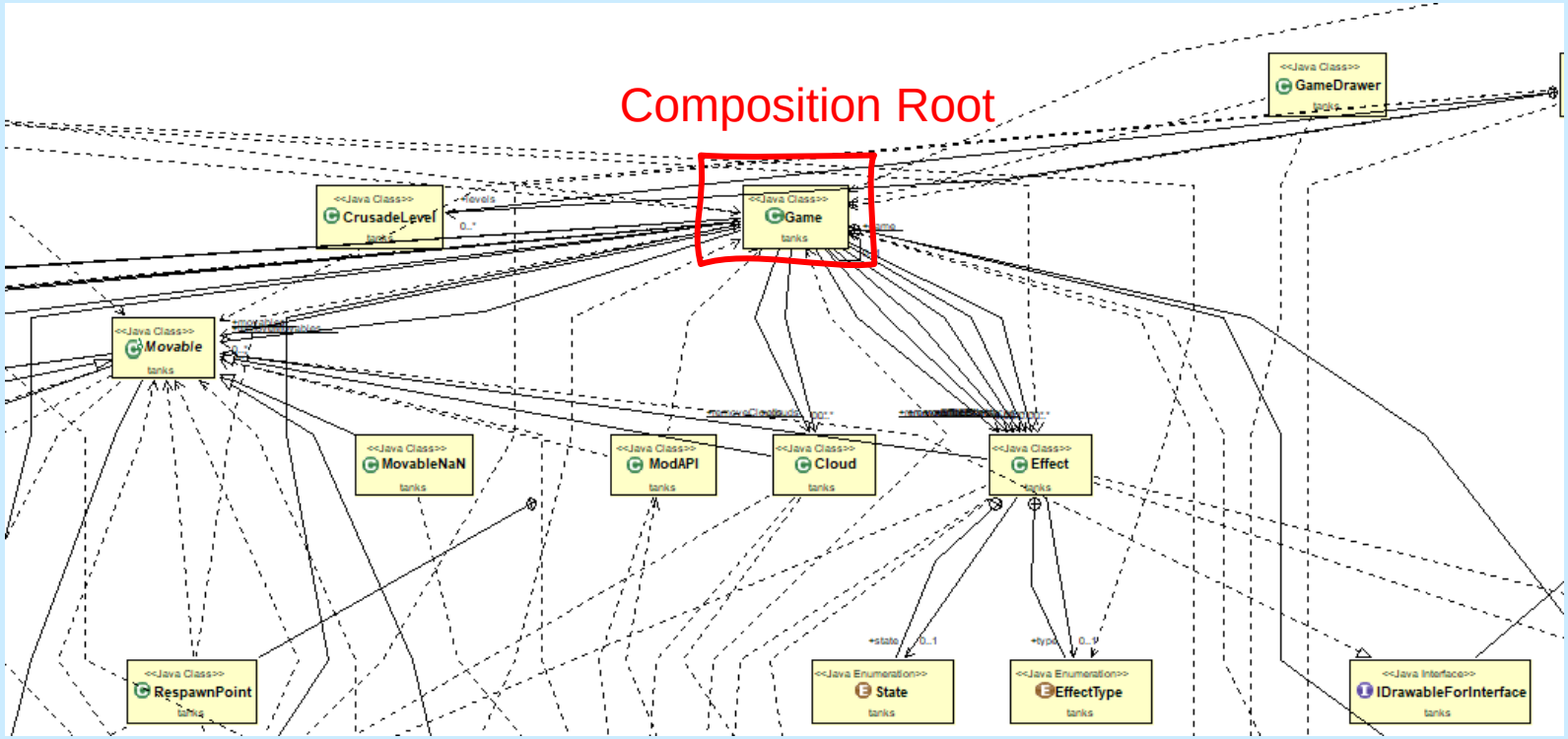
- 負責初始化和組合應用程式中所有相互依賴物件的類別
  A Composition Root is a single, logical location in an application where modules are composed together.
  - Close to the application's entry point
  - Takes care of composing object graphs of loosely coupled classes.
  - Takes a direct dependency on all modules in the system.



圖片來源: Book on "Dependency Injection Principles, Practices, and Patterns"

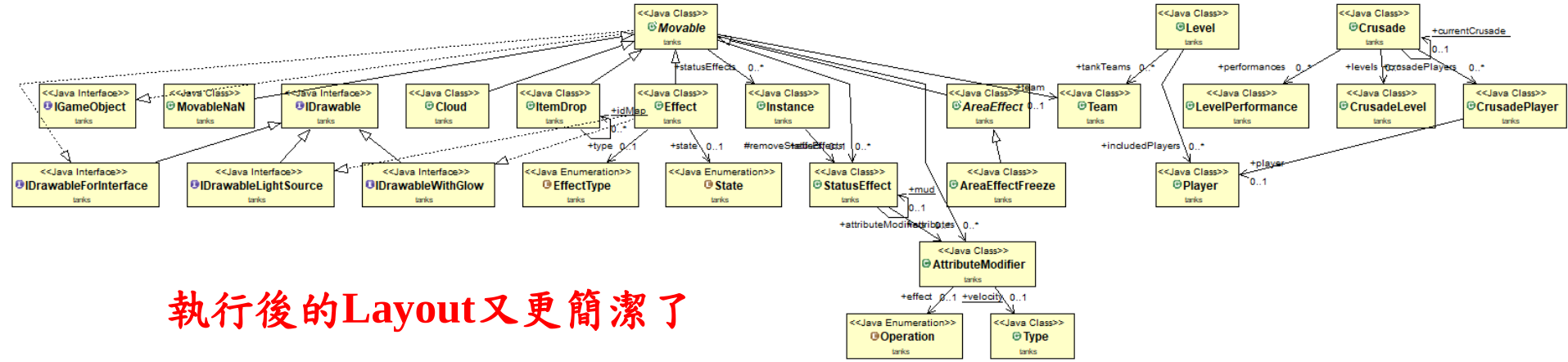
Tips – Composition Root對Code Structure View的影響

- Composition Root class(es)的特徵就是常有許多dependencies連至其他類別，牽制與混亂layout
- 在第一次code structure視覺化時，可考慮先移除它，專注在核心結構，之後再逐步加入它展開細節



Lab (Tank)

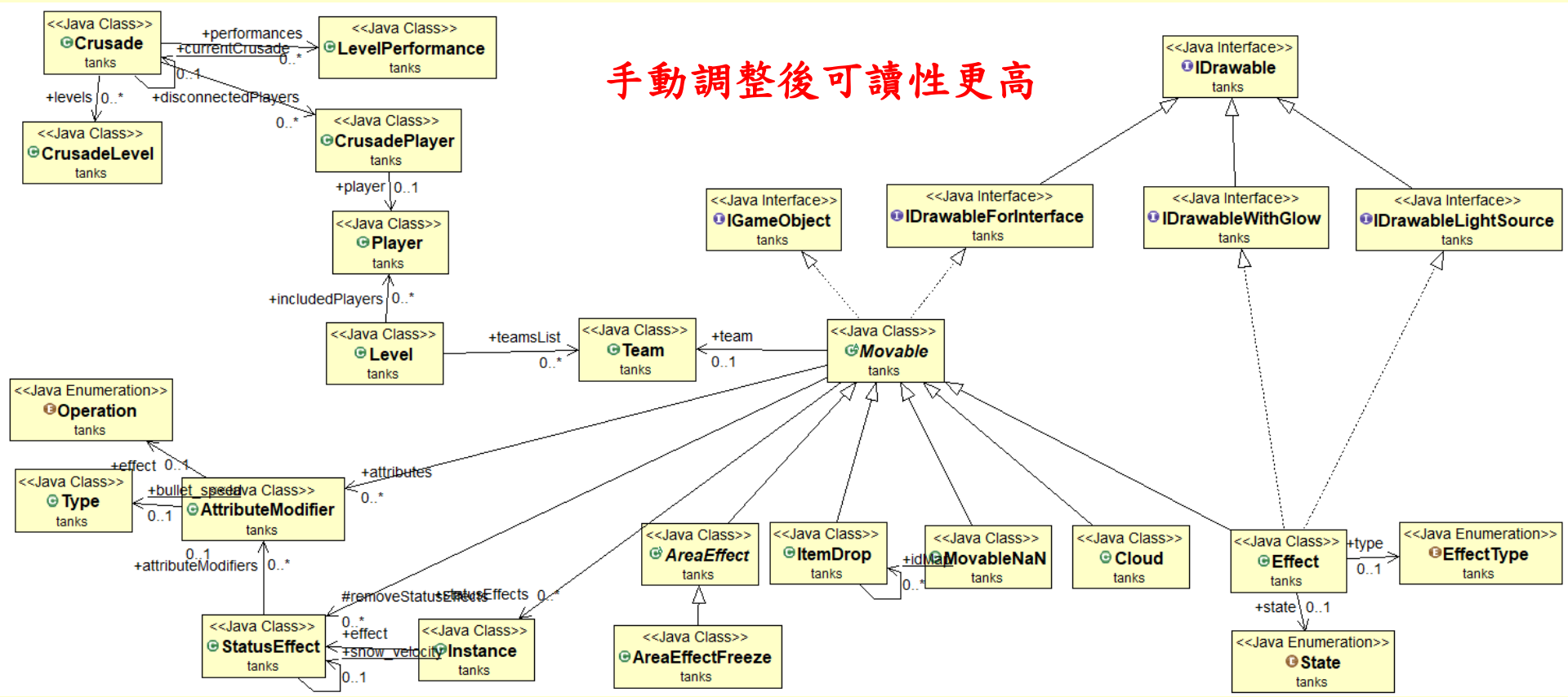
- 移除後，然後再【Layout Diagram】一次



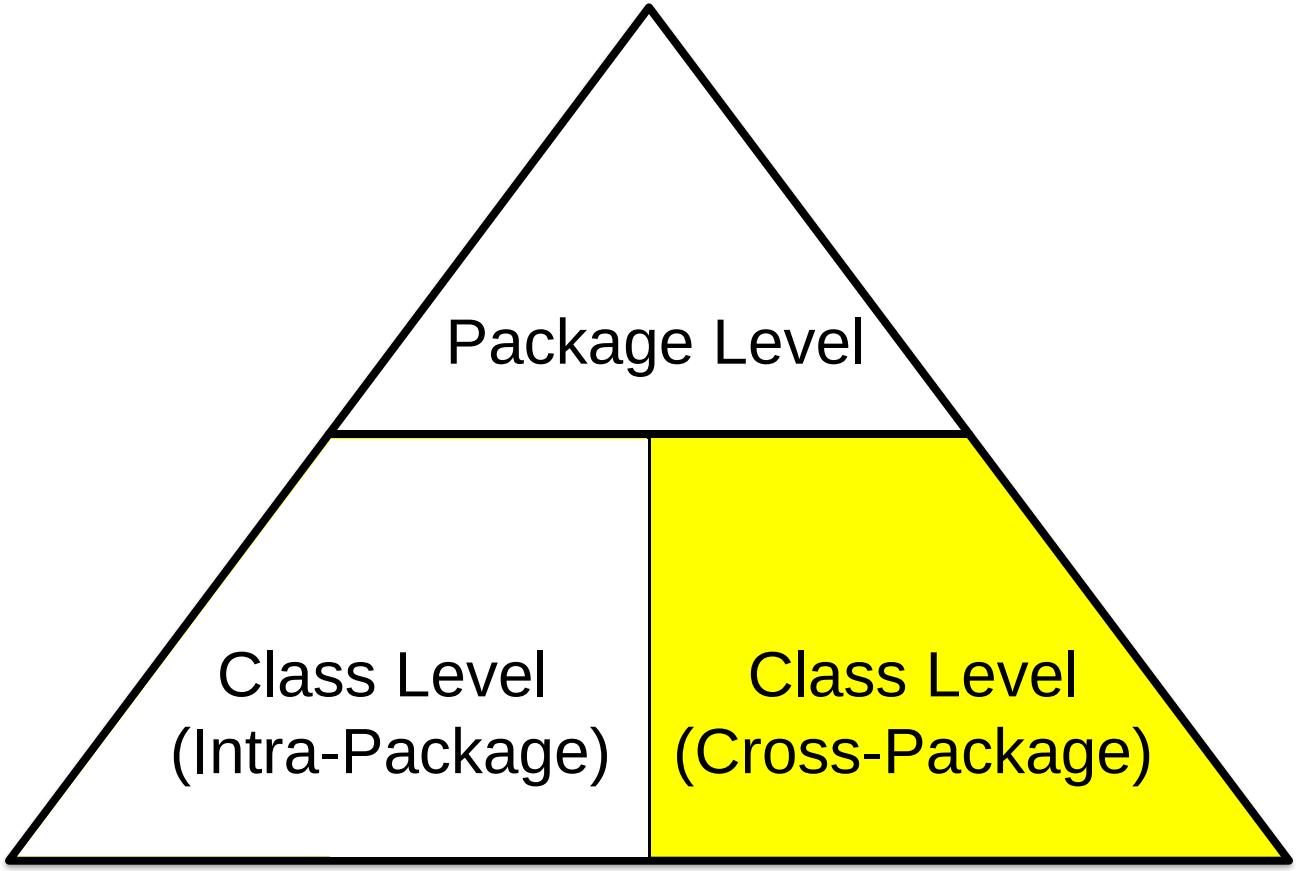
執行後的Layout又更簡潔了

Lab (Tank)

- 步驟5：手動調整layout，盡量維持【由上而下的樹狀結構】，並依耦合關係【群聚化】，最後的layout將變得較清晰易理解。

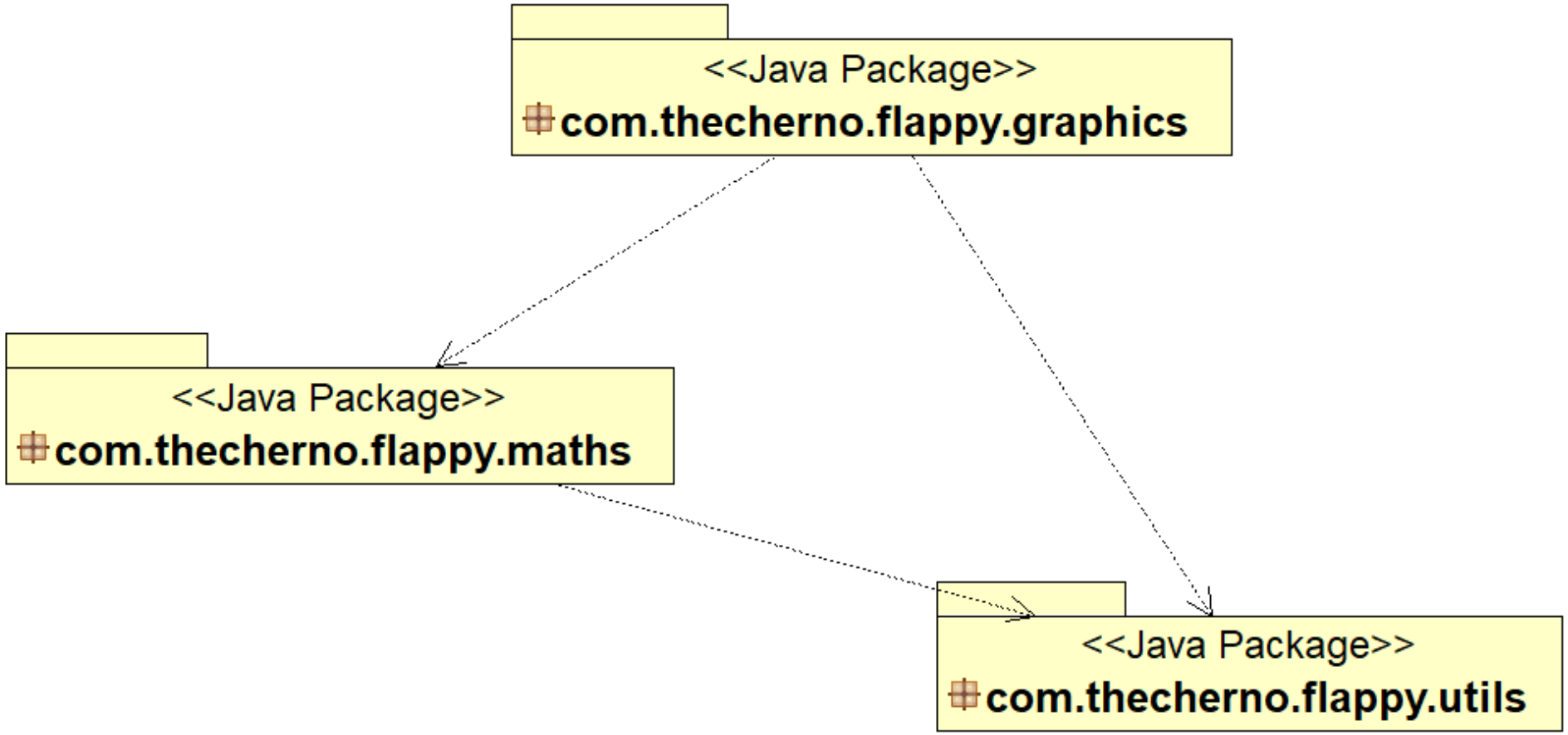


Levels of Abstraction



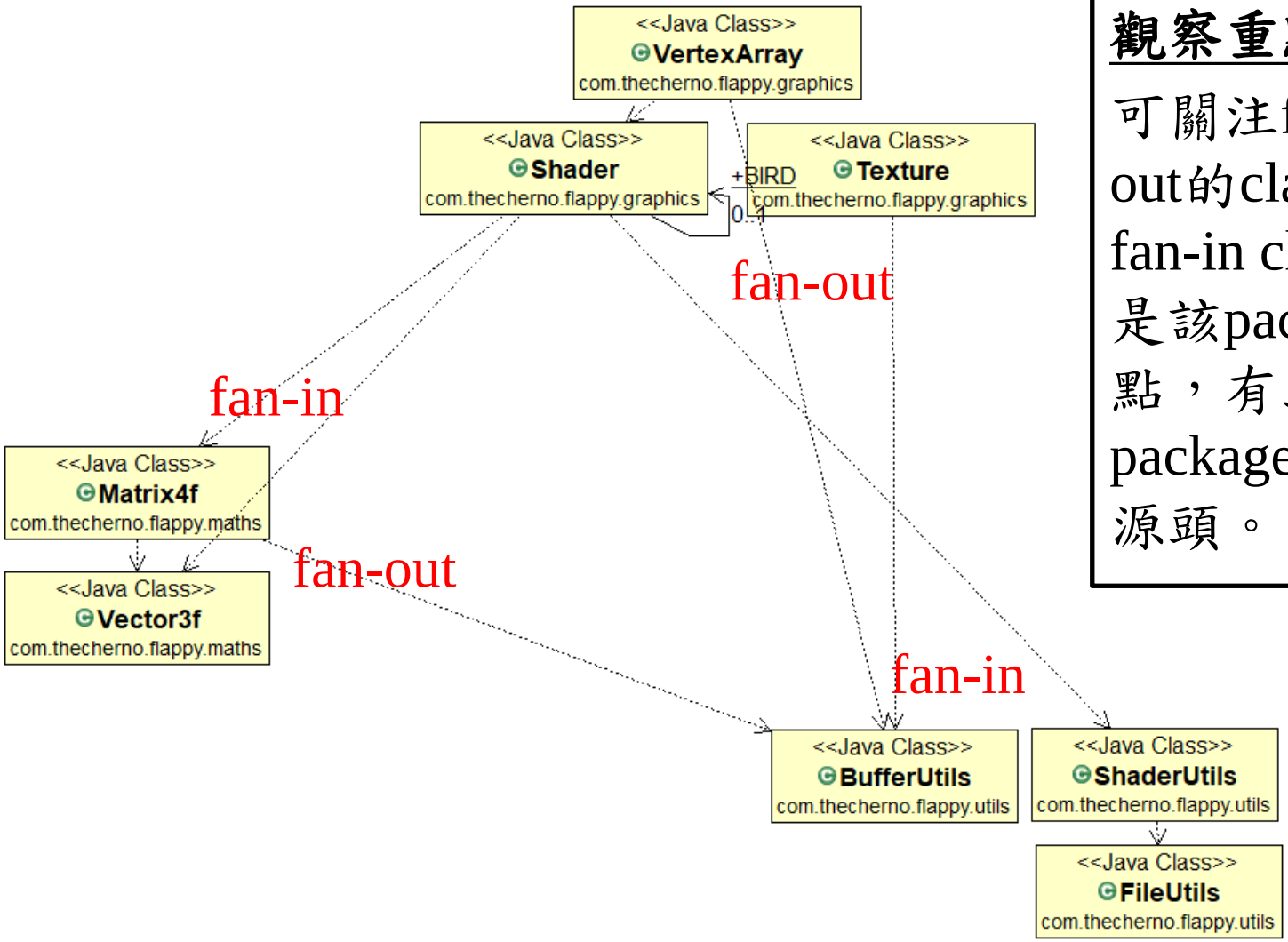
Class Level (Cross-Package)

- 假設我們要觀察這三個package內class間的關係



Class Level (Cross-Package)

- 依序將這三個package內的classes拖拉至diagram，彼此間關係即會顯示。



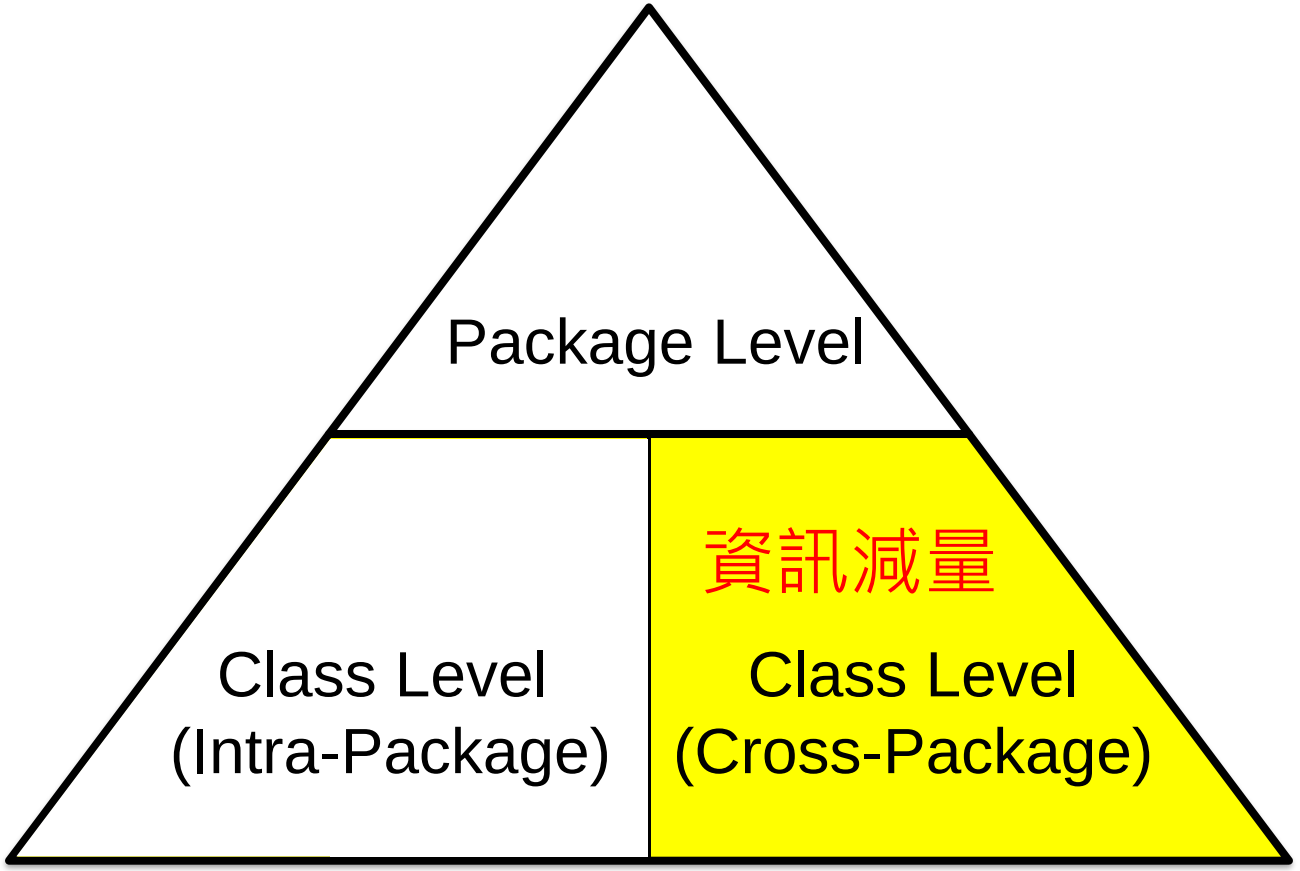
觀察重點

可關注fan-in與fan-out的classes。這些fan-in classes 通常是該package的進入點，有助於找到package行為的部分源頭。

Lab (Flappy)

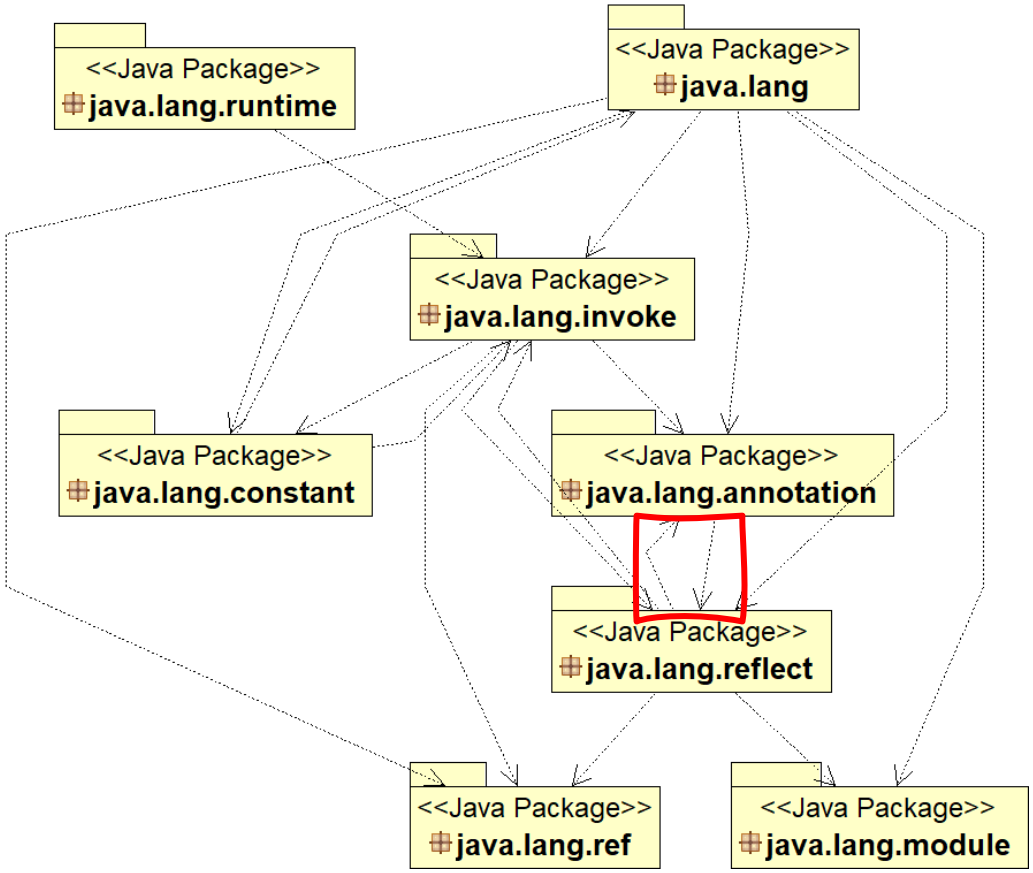
- 請繪製出上頁的diagram
  - https://github.com/TheCherno/Flappy.git

Levels of Abstraction



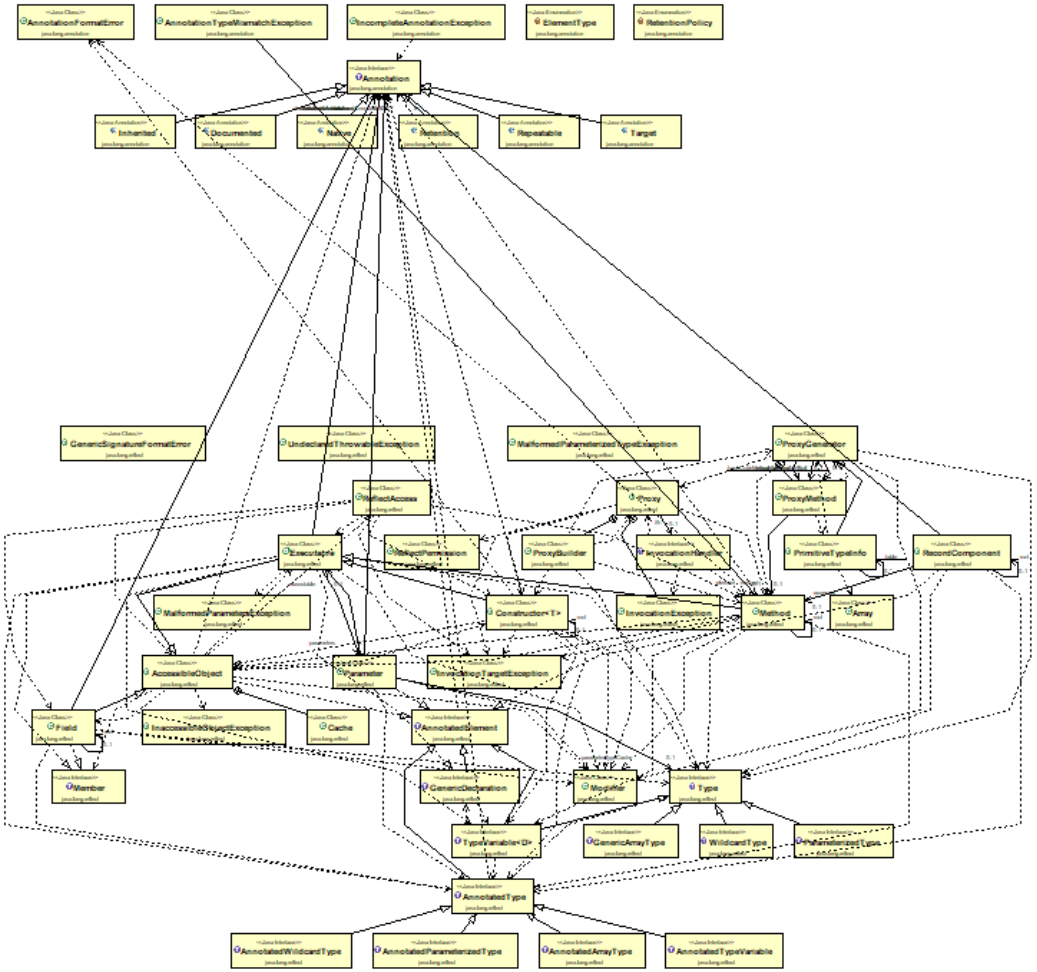
Class Level (Cross-Package) (減量)$_1$

- 但是，當多個package內的class數量太多時，diagram會過於複雜，造成不易觀察fan-in/out classes
- 例如，我們想觀察package `java.lang.annotation`與`java.lang.reflect`的雙向依賴關係



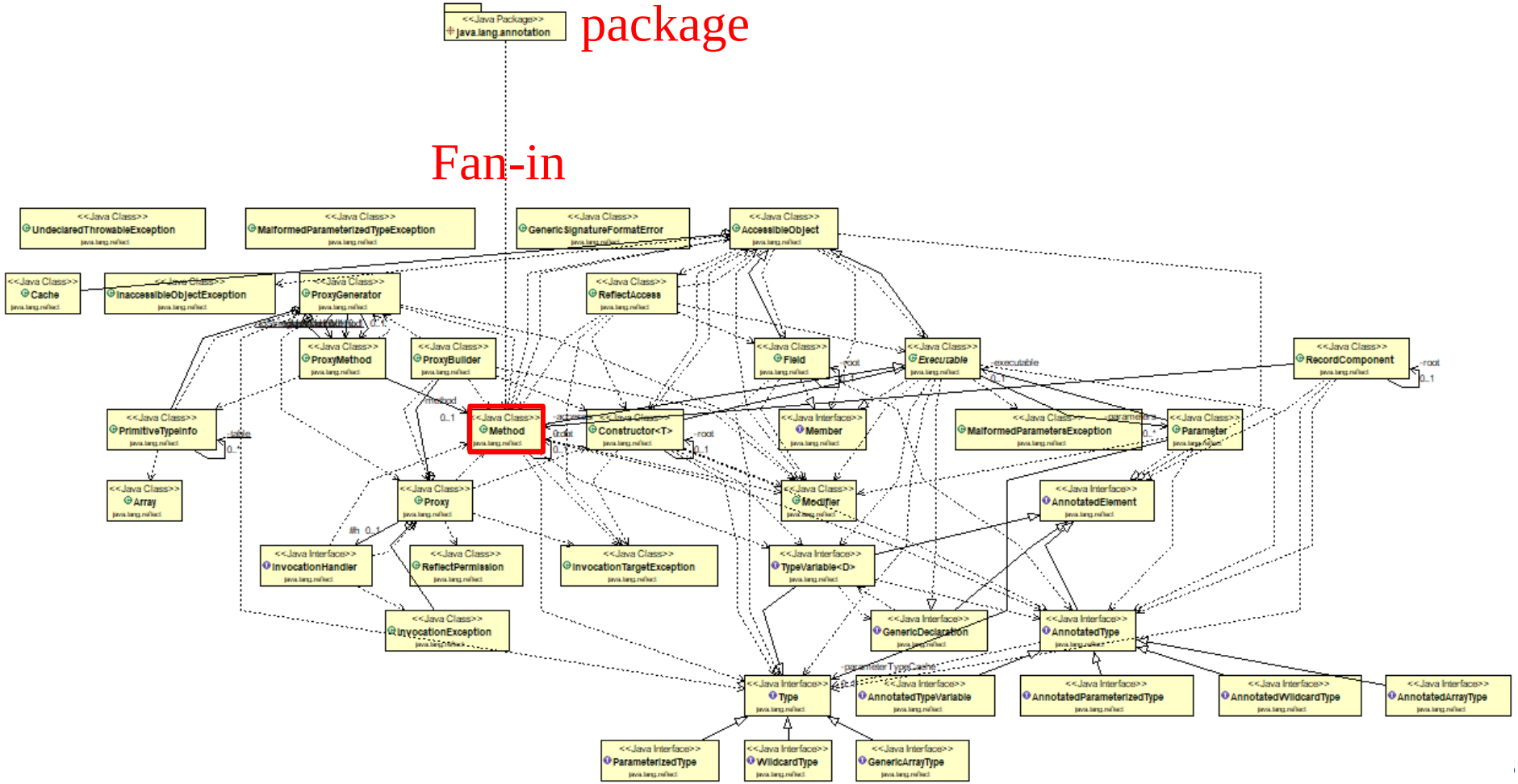
Class Level (Cross-Package) (減量)$_2$

- 若我們將這兩個package內所有class畫出，diagram會顯得複雜，造成fan-in classes不易識別。



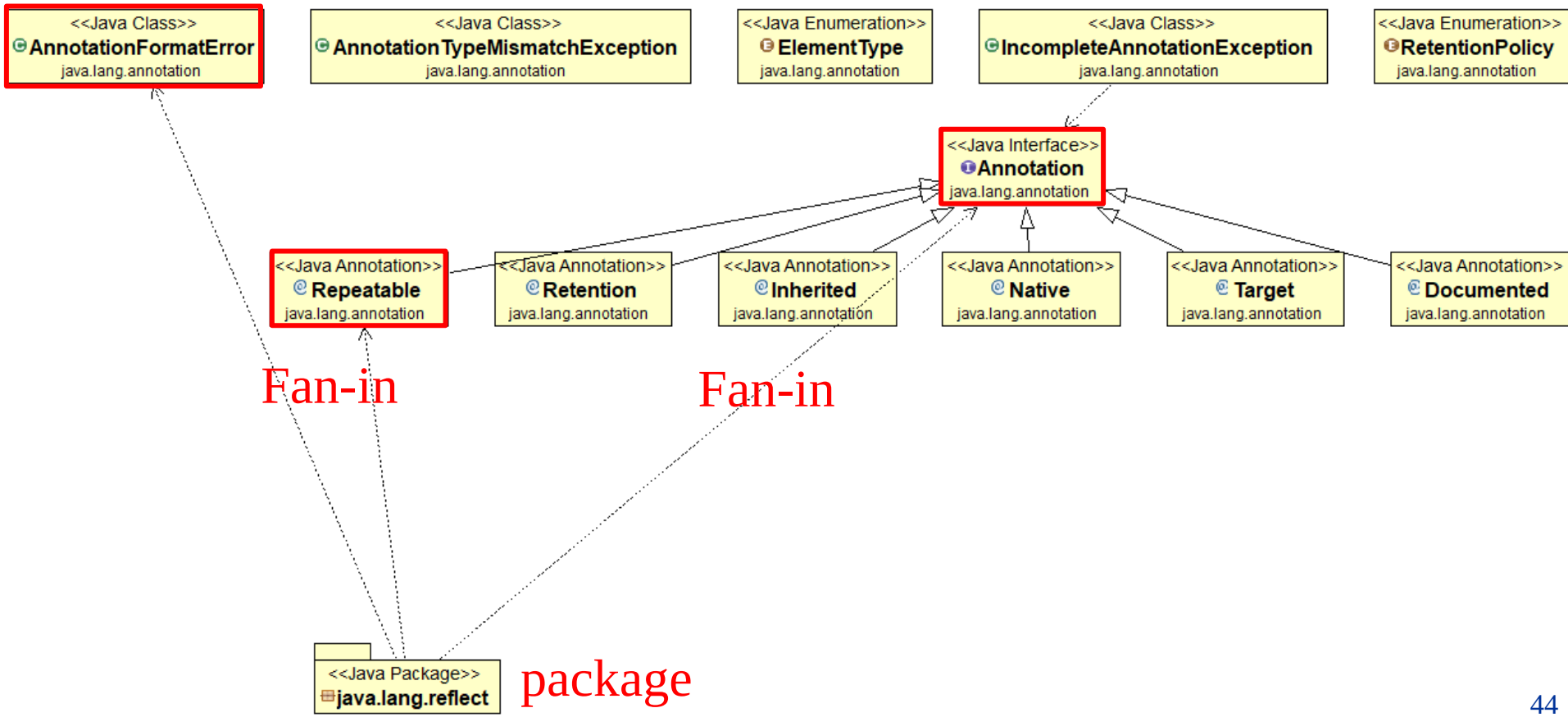
Class Level (Cross-Package) (減量)$_3$

- 因此，可採**packages與classes混搭**，將一個package內的classes與其他packages 放在一起進行佈局，便於找出fan-in classes。



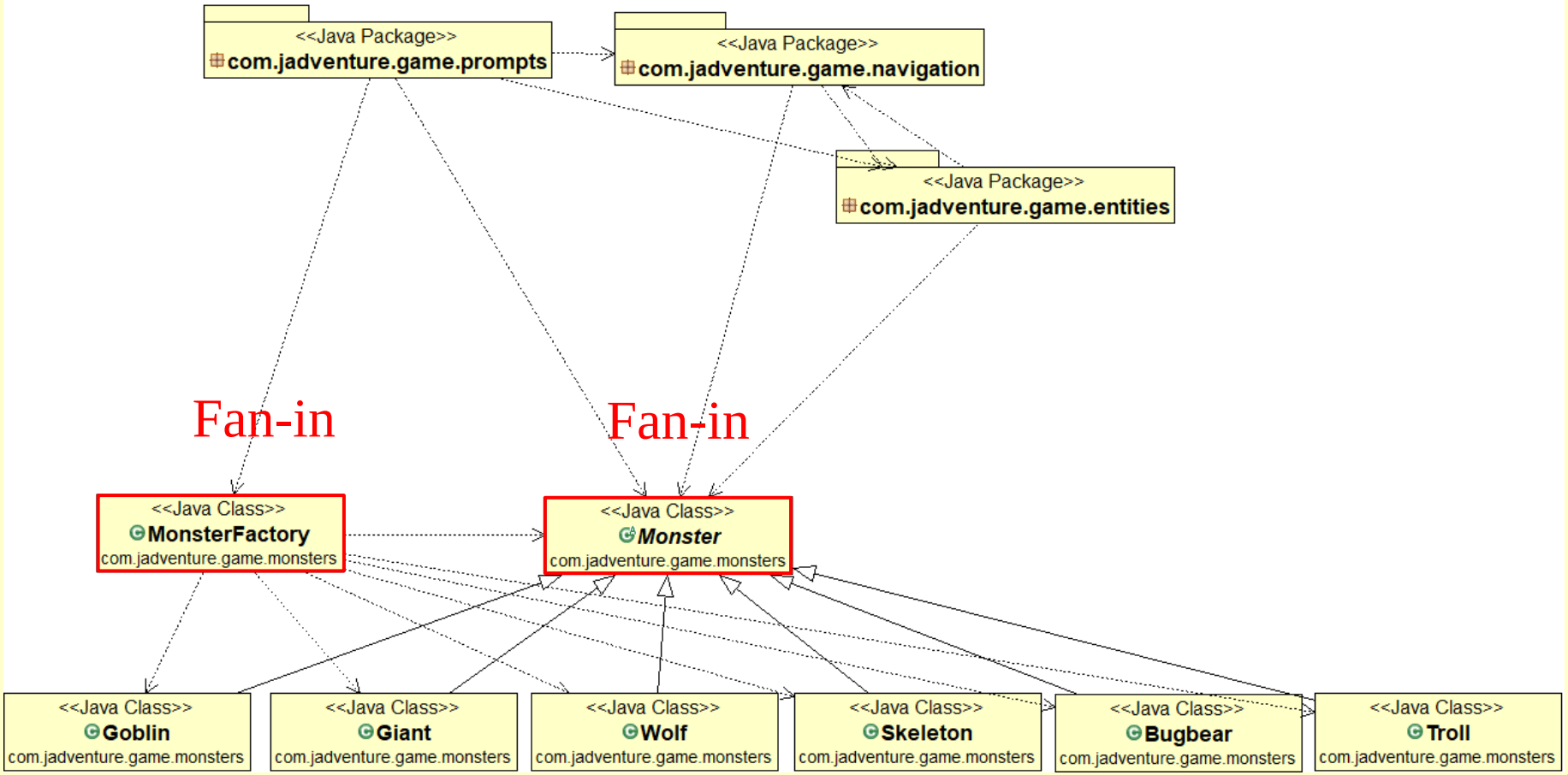
Class Level (Cross-Package) (減量)$_4$

- 相反地，亦可識別出由package `java.lang.reflect` fan-in的classes

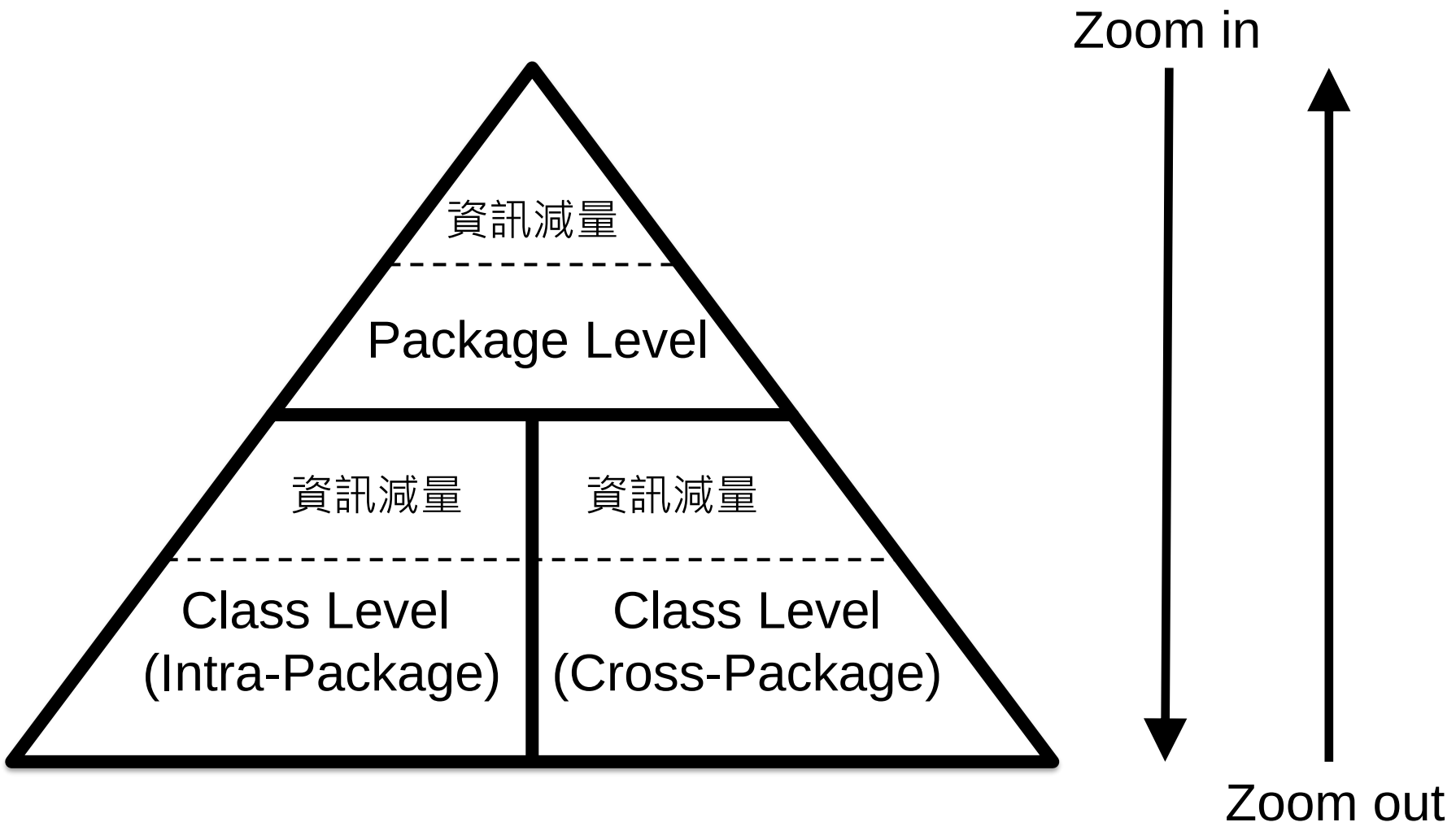


Lab (JAdventure)

- 請繪製並識別以下project的package `com.jadventure.game.monsters`內fan-in classes
  - https://github.com/Progether/JAdventure.git



Zoom In/Out Across Different Levels of Abstraction in Code Structure



Levels of Abstraction 摘要表

		觀察重點	資訊減量
Package Level		Package間若為單向依賴且無循環依賴，則耦合度較低，可提高維護性與延展性。	可聚焦部分package來觀察結構，挑選準則沒有標準，可以為主業務邏輯或某功能模塊。
Class Level	**Intra-Package**	1. 可關注依賴於高階抽象(interfaces、abstract classes)或低階實體(sub-classes)，在設計原則權衡下評估是否合適。 2. 可關注bidirectional dependencies，在設計原則權衡下評估是否合適。	1. 可先聚焦在核心業務的classes，隱藏次重要的classes，例如data classes、utility classes exception classes、enums、composition root、UI-layer classes 2. 可隱藏dependencies，只顯示強烈關係(inheritance, implementation與association)。
	Cross-Package	可關注fan-in與fan-out的classes，這些fan-in classes 通常是該package的進入點，有助於找到 package行為的部分源頭。	可採packages與classes混搭，將一個package內的classes與其他packages 放在一起進行佈局，便於找出fan-in classes。

總結

- 直接將大量所有package中的class關係結構全部一起視覺化會相當困難，遊走(zoom in/out)於levels of abstraction是個較可行的做法
- 透過各個level的觀察重點與資訊減量有助於理解code structure
- 視覺化後的code structure有時顯得複雜，但不代表就是不好的結構設計，需要搭配design principle進行評估與權衡
- 請注意，其他code structure visualization工具可能會與ObjectAid有差異，甚至有些程式語言的逆向工程工具不是產生UML Class Diagram，但都值得進一步了解