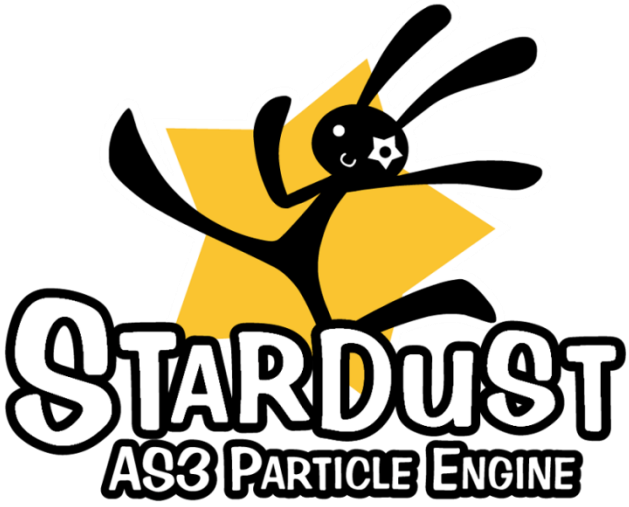
軟體設計模式期末專題

星塵粒子引擎



組別： 16

系級： 電機四

組員： 周明倫(B95901008)

林昭瑋(B95901038)

指導教授： 陳俊良

目次

Stardust Particle Engine 的由來

星塵粒子引擎 (Stardust Particle Engine) 是我 (周明倫) 於 2009 年暑假自修完 Head First Design Patterns 與 GoF Design Patterns 之後，為了自我練習設計模式而開發的實驗性質作品。在這之前我已經有開發另外一套粒子系統 Emitter，由於當時沒有很強的設計模式知識與觀念，類別與介面設計不佳，導致維護極度不容易，Stardust 也可以算是 Emitter 的後繼者吧。

以我個人的定義來說，粒子就是大量「行為、外觀相似」但「互相存在差異」的個體。粒子特效即是利用粒子製作的視覺特效，在電影、電玩是不可或缺的特效；例如火焰、煙霧、雨滴、雪花等，都算是粒子特效。現在網路上 Flash 網站與遊戲已非常流行，Flash Player (Flash 的虛擬機器)全球的普及率超過 95%，是一塊非常值得投資的領域。自從開發 Stardust 以後，我便一直以把 Stardust 推廣到全世界為目標，希望大家能夠接受 Stardust，並且使用 Stardust 來開發 Flash 網站與遊戲的視覺特效。

粒子引擎最重要的就是效能，因為要即時處理大量的粒子物件、更新它們的屬性與參數，並且將其視覺化於螢幕上。另外，粒子特效也需要一些行為控制，例如亂流、群體移動、重力模擬等。Stardust 將繁瑣的演算法包裝起來，提供一套簡單上手的介面，讓程式設計師不用費心去撰寫粒子特效相關的底層記憶體管理與演算法等，而能夠專注於粒子行為的設計上。

這學期上了設計模式的課，讓我們對改善 Stardust 架構躍躍欲試，於是我們這學期決定好好研究一下 Stardust 的架構、對其架構作優化、盡量把我們這學期所學到的知識應用在上面。

▲一位日本 ActionScript 3.0 程式設計師用 Stardust 製作的蝴蝶粒子特效 (ActionScript 3.0 為 Flash 的程式語言)

Stardust 目前的發展狀況

Google Code Project

目前 Stardust 已經發展成一個稍具規模的 Project，目前使用 Google Code 提供的 SVN Repository 版本控管系統來儲存所有相關檔案。

Stardust 首頁：http://code.google.com/p/stardust-particle-engine
首頁上提供各版本的原始檔、範例、編譯過的 Bytecode 下載。

PDF Manual & Video Tutorials

為了幫助大家學習 Stardust，首頁上除了提供用 ASDoc 自動生成的 Documentation 以外，還提供了 PDF 格式的英文自修手冊；另外，於 YouTube 也有提供一系列的 Stardust 教學影片。這些教學資源於 Stardust 專案首頁上皆可找到連結。

Feedbacks

Stardust 剛推出的時候使用者並沒有很多，但是最近開始有人於 Twitter 與 WonderFL 上流傳 Stardust 的消息，於是 Stardust 的使用者人數開始快速增加。已經有一些網友於 Blog 上面撰寫 Stardust 的介紹與教學，Stardust 也被 WonderFL 收錄為可用 API。

WonderFL 首頁：http://wonderfl.net

※ WonderFL 為一個日本 ActionScript 3.0 專業交流網站，提供一個簡單的文字編輯器和線上編譯服務，讓大家可以從沒有 Flash 編譯器的電腦直接於網頁上撰寫 Flash 程式、即時觀看編譯後的結果、以及分享和觀賞其他使用者的作品；目前全世界已經有許多知名的 ActionScript 3.0 程式設計師加入 WonderFL 的社群。WonderFL 收錄許多第三方 API，讓程式設計師撰寫程式的時候有更多的 API 選擇，而 Stardust 也有被收錄其中。

Stardust Particle Engine 的特色

擴充容易

Stardust 將粒子的初始化與行為委派給 **`Initializer`** 和 **`Action`** 兩個類別，將視覺化處理委派給 **`Renderer`** 類別，並且已經有非常多的內建 **`Initializer`**、**`Action`**、與 **`Renderer`** 類別供選擇。如果使用者想要自己擴充出自己的自訂粒子初始化、行為、或者視覺化方式，可以自行繼承這些類別來辦到。

支援 2D 與 3D 特效

Stardust 的粒子模擬系統有兩套 model，一套是 2D 粒子特效，而另外一套是 3D 的。使用者可以視需求採用 2D 或 3D 的 model 來作粒子模擬，然後用 **`Renderer`** 類別將此 model 視覺化。Stardust 提供內建的 2D 與 3D Renderer，也提供支援一些其他的 ActionScript 3.0 第三方 3D 引擎的 Renderer，例如 ZedBox(我自己寫的 2.5D 引擎)、Papervision3D、與 ND3D。

可設計複雜的粒子行為

Action Trigger 是一種特別的 Action，同時也是一個 Composite Action，它能夠指定其 Component Action 在「特定條件」狀況下觸發，藉此設計行為更加複雜與多樣化的粒子特效。例如「當一個粒子死亡時，原地生出多個粒子」可以製作煙火特效，以集「當兩個粒子互相碰撞實，蹦出火花粒子」可以製作物體撞擊火花特效。

支援 XML 序列化

Stardust 最重要的特色之一，就是它支援 XML 序列化。Stardust 是現行的 ActionScript 3.0 粒子引擎中唯一支援 XML 序列化功能的。藉由序列化，使用者可以將一個粒子系統的行為設計與參數轉換成一個外部 XML 檔案，以供 Flash 應用程式動態載入；這個功能在開發大型 Flash 應用程式的時候很有用：只要修改並且儲存該外部 XML 檔案，然後重新執行 Flash 應用程式即可套用新的粒子特效參數，而不用重新編譯。

光碟附加檔案說明

檔案與資料夾結構

files ┬ ***Flash Player.exe***
- ├ src (Stardust 原始檔)
  - │ └ ***Stardust.as3proj***
- ├ FlashDevelop
  - │ └ ***FlashDevelop.exe***
- ├ docs (Stardust documentation)
  - │ └ ***index.html*** (documentation 首頁)
- ├ manual (Stardust PDF 英文說明書)
  - │ ├ ***Stardust Particle Engine Manual.pdf***
  - │ └ examples
    - │ └ 說明書範例
- └ demos
  - ├ ***Stardust Demos.as3proj***
  - ├ 2D (2D demos)
    - │ └ <任意資料夾>
      - │ ├ FLA 檔
      - │ ├ AS 檔
      - │ └ SWF 檔
  - └ 3D (3D demos)
    - └ <任意資料夾>
      - ├ FLA 檔
      - ├ AS 檔
      - └ SWF 檔

★ src 資料夾中包含完整的 Stardust project 原始檔，其中的 Stardust.as3proj 是免費第三方編輯器 FlashDevelop 的專案檔。

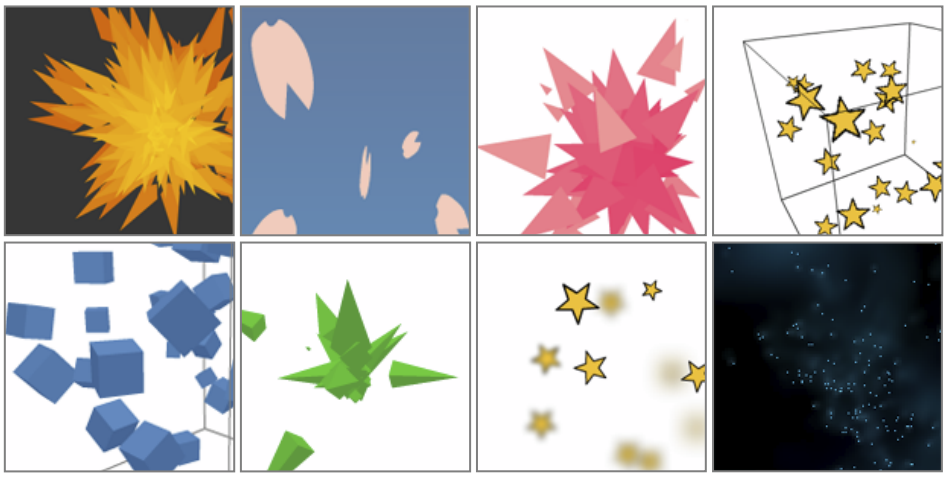
★ FlashDevelop 資料夾中的 FlashDevleop.exe 為 FlashDevleop 的 stand-alone 版本主程式，可以用來開啟 AS3PROJ 專案檔。請注意，FlashDevelop 需要於支援.NET Framework 2.0 的執行環境下才可執行。

★ docs 資料夾中包含透過 Adobe 提供的 documentation 生成器 ASDocs 所自動產生的 API 說明文件。

★ manual 資料夾中有我為 Stardust 撰寫的 PDF 英文使用手冊，是於 Stardust 首頁下載得到的。

★ demos 資料夾中是一些 Stardust 的特效展示，包含原始檔與編譯過後的 SWF 檔案，執行方式請見下頁的檔案說明。

以下為一些 Demo 的截圖：



檔案類型說明

★ FlashDevelop 為一個使用.NET Framework 的免費 ActionScript 編輯器，其功能完整、使用者眾多，是最熱門的免費 ActionScript 編輯器。

★ AS3PROJ 檔為 FlashDevleop 的 ActionScript 3.0 Project 專案檔，用 FlashDevleop 開啟，即可於 Project Panel 中瀏覽專案。

★ Flash Player.exe 是 Flash Player 的 stand-alone 版本，是可獨立執行的虛擬機器，可以用來開啟 SWF 檔。

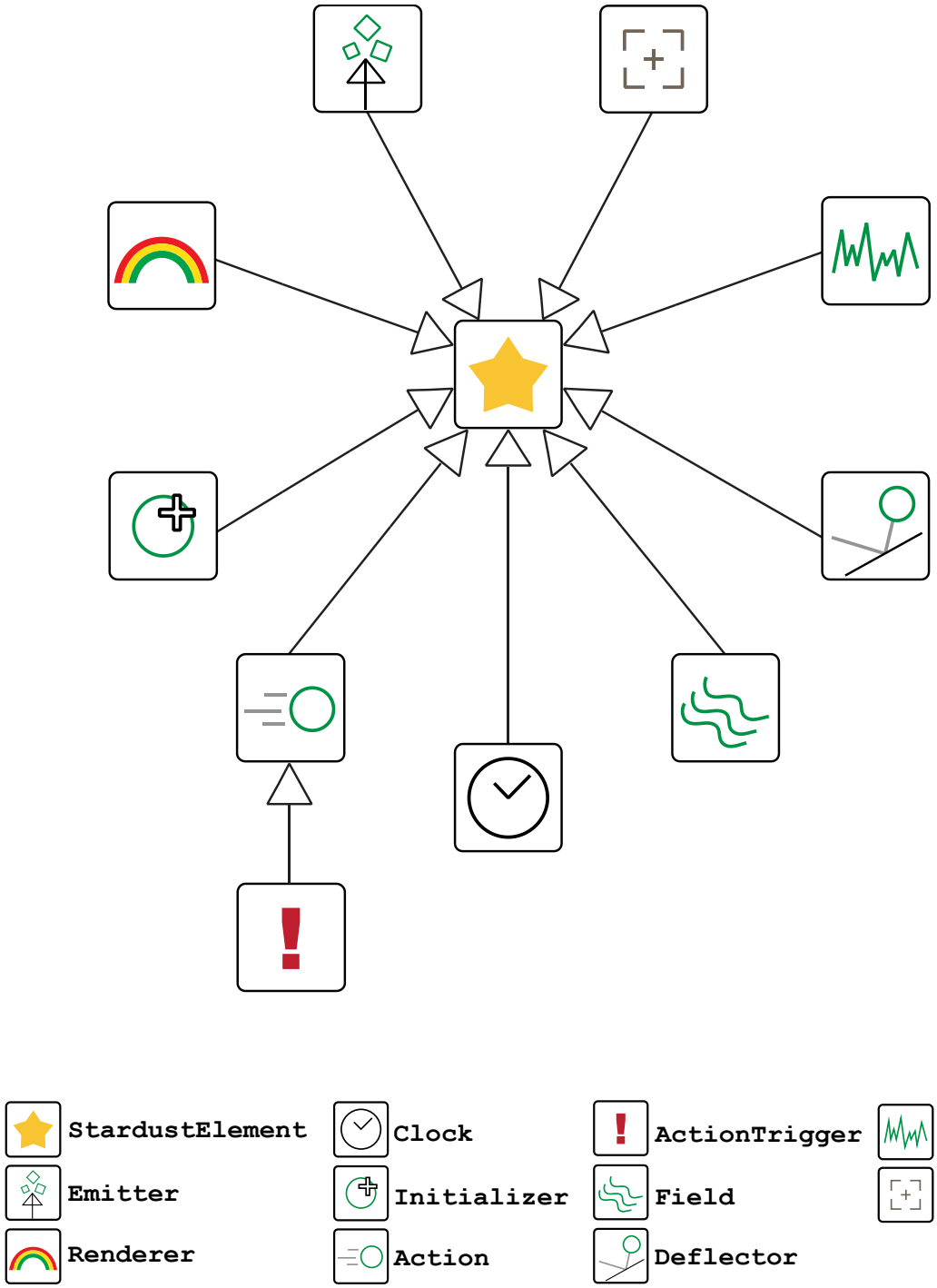
★ FLA 檔是 Flash IDE 的原始檔，若沒有 Flash IDE 無法開啟該檔案並無所謂，因為這些 FLA 檔只包含視覺元件的外觀設計資訊，真正的程式碼都寫在 AS 檔裡。

★ 一個 AS 檔相當於一個 JAVA 檔，其中包含一個 class 或者 interface 的定義，與 FLA 檔同名的 AS 檔為 main class。

★ SWF 檔相當於 Java 的 CLASS 檔，是 FLA 檔與 AS 檔編譯之後產生的 Bytecode 檔，將其拖曳到 Flash Player 視窗中即可執行。

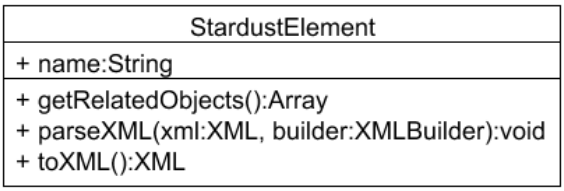
Stardust 類別繼承關係

為了不要讓畫面充滿了擁擠的 UML diagram，我先直接將 Stardust 的 PDF 英文說明書中的插圖在此重現，之後再用 UML diagram 詳細介紹各個類別。



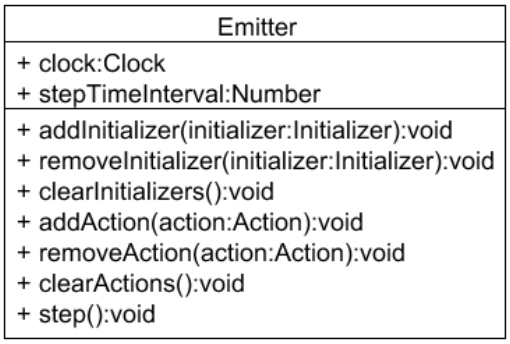
Stardust 元素

`StardustElement` 類別



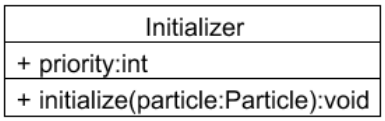
除了 Value Object (VO)類別以外，Stardust 的所有類別都繼承自 **`StardustElement`** 這個類別。這個類別中定義了 XML 序列化所需的 method 與 property，諸如 **`name`**、**`toXML()`**與 **`parseXML()`**。

`Emitter` 類別



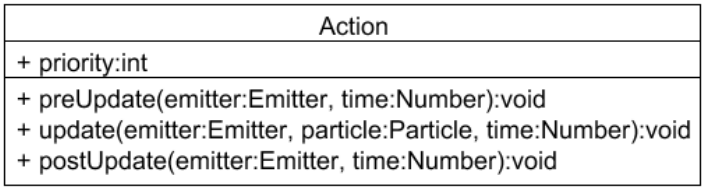
Emitter 是粒子系統的噴發器，新粒子的生成與初始化、粒子的行為都是由它管理。使用者透過 **`addInitializer()`**與 **`addAction()`**兩個 method 來賦予 Emitter 想要使用的粒子初始化與粒子行為。**`step()`**為 Stardust 的主迴圈，使用者需持續呼叫這個 method 以維持粒子模擬的運作。**`stepTimeInterval`** 的數值是呼叫一次主迴圈所模擬的時間間隔，例如 2 即代表粒子模擬的速度為兩倍。**`clock`** 決定每一次呼叫主迴圈，Emitter 生成多少新的粒子。

Initializer 類別



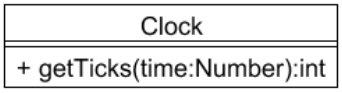
此類別是用來初始化新生粒子的，新生成的粒子物件會被當作參數傳入 Initializer 的 **initialize()** method 而被初始化。**priority** 是 Initializer 的執行優先順序。

Action 類別



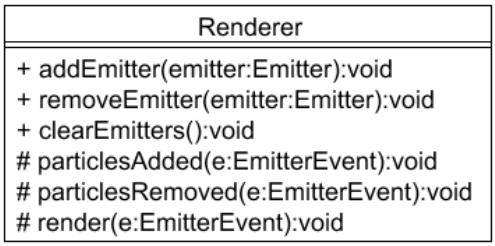
此類別是用於每一次呼叫主迴圈時，更新粒子物件的屬性用的，例如 Move Action 會依照粒子物件的速度來更新位置。**preUpdate()** 會於開始更新粒子時呼叫一次，用以設定好會用到的資源；**update()** 會用於更新所有的粒子，每個粒子都會被當作參數傳入該 method 一次；**postUpdate()** 則是於更新完粒子屬性之後被呼叫一次，是用來釋放資源用的。**priority** 是 Action 的執行優先順序。

Clock 類別



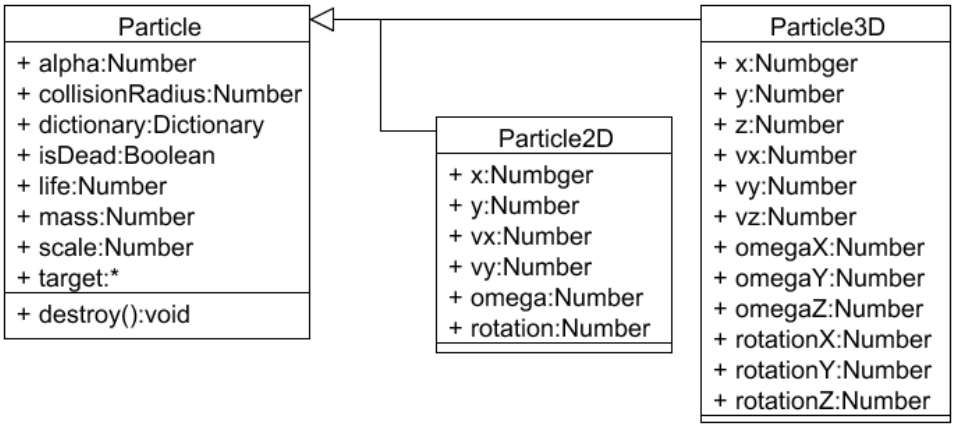
Clock 類別用於決定 Emitter 於一次主迴圈中，會生成多少個新粒子物件：的回傳值即代表新粒子的生成個數。

Renderer 類別



Renderer 負責將 Emitter 的粒子 model 視覺化，透過 **addEmitter()** 可以開始監控 Emitter 的粒子生成、死亡、與主迴圈的呼叫。**particlesAdded()** 會於新粒子生成時被觸發，**particlesRemoved()** 會於死亡粒子從模擬中移除時被觸發，**render()** 則會於 **Emitter.step()** 呼叫完成時觸發，以將主迴圈結束時的 model 視覺化。

Particle 類別



此為 VO 類別。一個 **Particle** 物件代表一個粒子，此類別定義了 2D 與 3D 粒子共用的屬性，2D 與 3D 粒子特有的屬性則分別定義於 **Particle2D** 與 **Particle3D** 類別中。

Stardust 使用流程

生成 Emitter 物件

Emitter 是 Stardust 的根本，要創造粒子特效就需要生成 Emitter 物件。生成 Emitter 物件時可從其 Constructor 傳入一個 Clock 物件，以指定生成粒子的頻率。

以下程式碼會生成一個 **Emitter** 物件，並且指派一個 **SteadyClock** 物件，表示每一次呼叫主迴圈時，創造三個新的粒子。

```
var emitter:Emitter= new Emitter2D(new SteadyClock(3));
```

※ **Emitter2D** 類別是用來創造 2D 粒子特效的 **Emitter** 子類別，而 **Emitter3D** 則用來創造 3D 粒子特效。

指派 Initializer 物件

Initializer 用來初始化新生粒子的屬性；使用者可依自己需求，指派想要使用的 Initializer 物件給 Emitter 物件。

以下程式會碼指派初始化粒子尺寸的 **Scale** Initializer 物件，並且於其 Constructor 中傳入一個 **UniformRandom** 物件，導致新生粒子的尺寸位於 2±0.5 的範圍內。

```
emitter.addInitializer(
                new Scale(new UniformRandom(2, 0.5)));
```

指派 Action 物件

Action 用來於主迴圈中更新粒子的屬性；同 Initializer，使用者可以依自己的需求，指派要使用的 Action 物件給 Emitter 物件

以下程式碼會指派一個 **Move** Action 物件給 **Emitter** 物件，導致每一次呼叫主迴圈時，粒子都會依照其速度而更新位置座標。

```
emitter.addAction(new Move());
```

生成 Renderer 物件

以上的步驟都將是只會修改與更新粒子模擬的 Model 而已，要將其以視覺呈現在畫面上，需要使用 Renderer 物件。視覺呈現的方式有很多種，使用者可依自己的需求使用不同的 Renderer。

以下程式碼會生成一個 **DisplayObjectRenderer** 物件、指定該 Renderer 視覺化的目標 Container、並且指派一個 Emitter 物件給它，每一次呼叫完主迴圈，Renderer 會自動更新畫面。

```
var renderer:Renderer
                = new DisplayObjectRenderer(container);

renderer.addEmitter(emitter);
```

持續呼叫主迴圈

以上皆為前置作業，要開始模擬粒子特效，需要持續呼叫主迴圈函式 **Emitter.step()**。以下範例程式碼監聽 Flash 每一次更新 Frame 的事件，於這個時候呼叫主迴圈。

```
addEventListener(Event.ENTER_FRAME, mainLoop);

function mainLoop(e:Event):void {
    emitter.step();
}
```

Stardust 主迴圈運作流程

以下行為依照一次呼叫主迴圈的發生順序列出：

生成新粒子

Emitter 會呼叫其被指派的 Clock 物件的 **`getTicks()`** 函式，該函式的回傳值決定本次主迴圈生成的新粒子數量；另外，新粒子的各項參數將會被 **`Initializer.initialize()`** 函式初始化。初始化完畢後，Emitter 將會通知 Renderer 作新生粒子的相關處理。

更新粒子

每一個粒子的各項參數將會被 **`Action.update()`** 函式更新。於這些 Action 的更新流程中，某些 Action 物件有可能會把粒子標示為「死亡」 (將 **`Particle.isDead`** 屬性設為 **`true`**)，這些死亡的粒子接下來會被移除。

移除死亡粒子

根據 **`Particle.isDead`** 屬性，Emitter 會將死亡的粒子從模擬中移除，並且通知 Renderer 做死亡的相關處理。

視覺化呈現

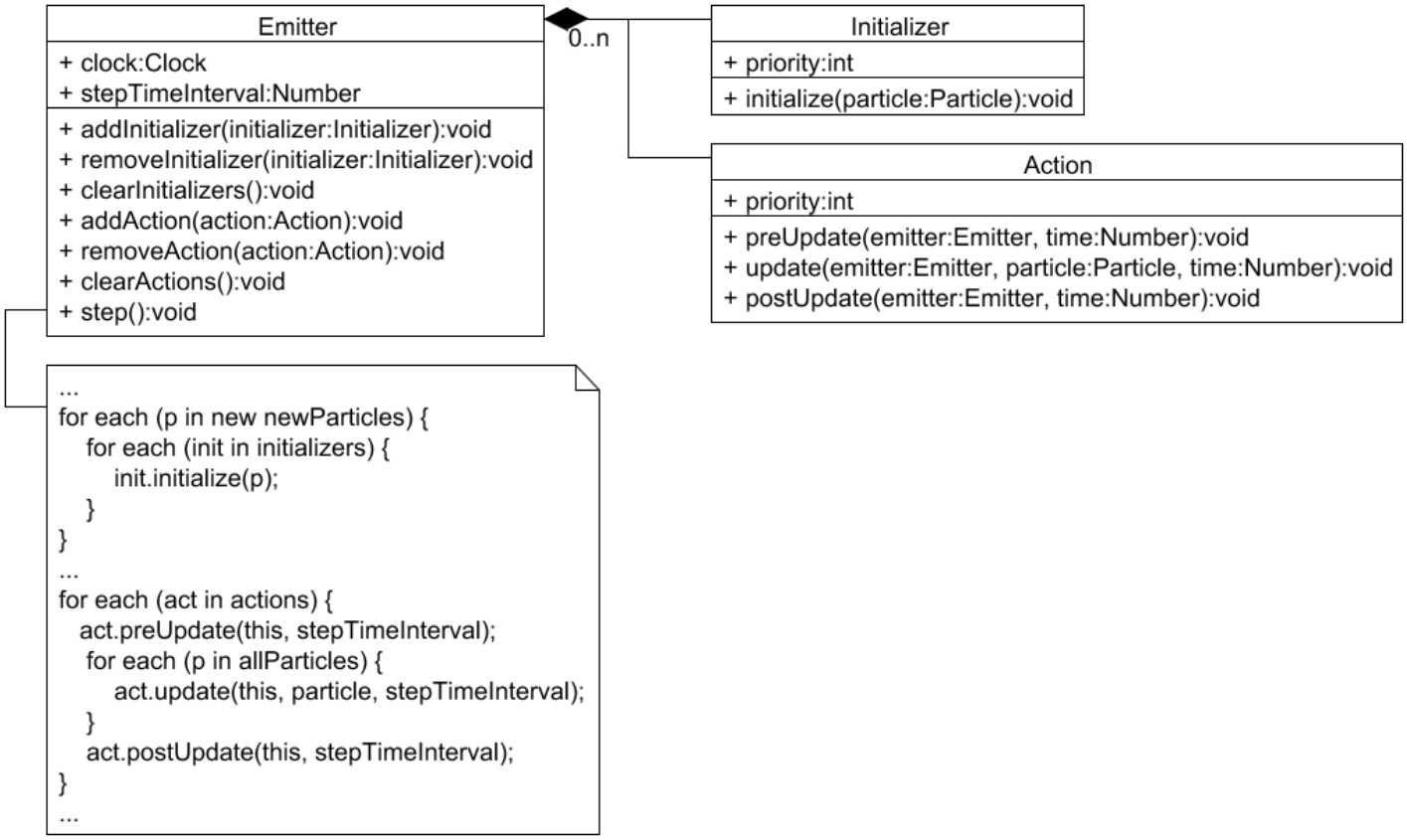
主迴圈中修改 Model 的部分到此為止，接下來 Emitter 會通知 Renderer 依照主迴圈結束前的 Model 來視覺化呈現粒子特效。

Stardust 與設計模式

使用自己想要的 Initializer 與 Action

Command Pattern

使用者可依照自己需求使用不同組合的 Initializer 與 Action 物件，此為 Command Pattern。Emitter 只管呼叫 **`Initializer.initialize()`** 來初始化新生粒子，以及呼叫 **`Action.update()`** 來更新粒子屬性；真正的 **`initialize()`** 與 **`update()`** 行為是由 Initializer 與 Action 子類別來 Override。**`Initializer.initialize()`** 與 **`Action.update()`** 即相當於 Command Pattern 中 Command Object 的 **`execute()`** 函式。

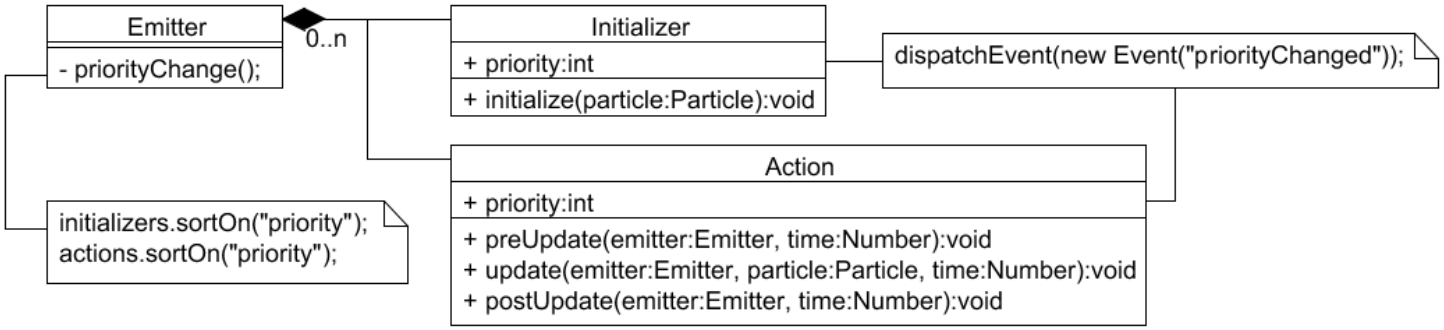


Initializer 與 Action 的執行優先順序變更

Observer Pattern

Initializer 與 Action 都有定義一個 **`priority`** 屬性，代表這些物件的使用優先順序，Emitter 於使用 Initializer 與 Action 物件時，將會先使用 **`priority`** 值較大者。當一個 Initializer 或 Action 物件的 **`priority`** 屬性變更時，將會透過 Observer Pattern 通知 Emitter，導致 Emitter 重新依照優先順序來排序物件；Initializer 與 Action 物件為 Subject，Emitter 的角色則是 Observer。

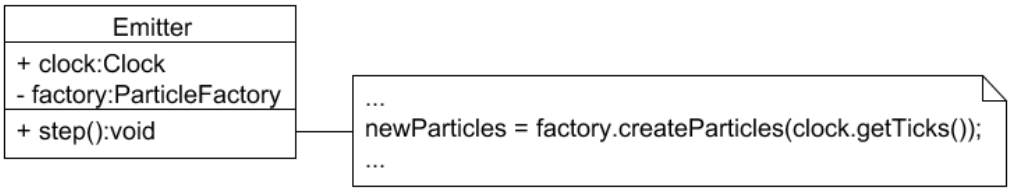
(此功能使用 ActionScript 3.0 的 Event API 來實作 Observer Pattern)



利用 Clock 來決定粒子生成頻率

Strategy Pattern

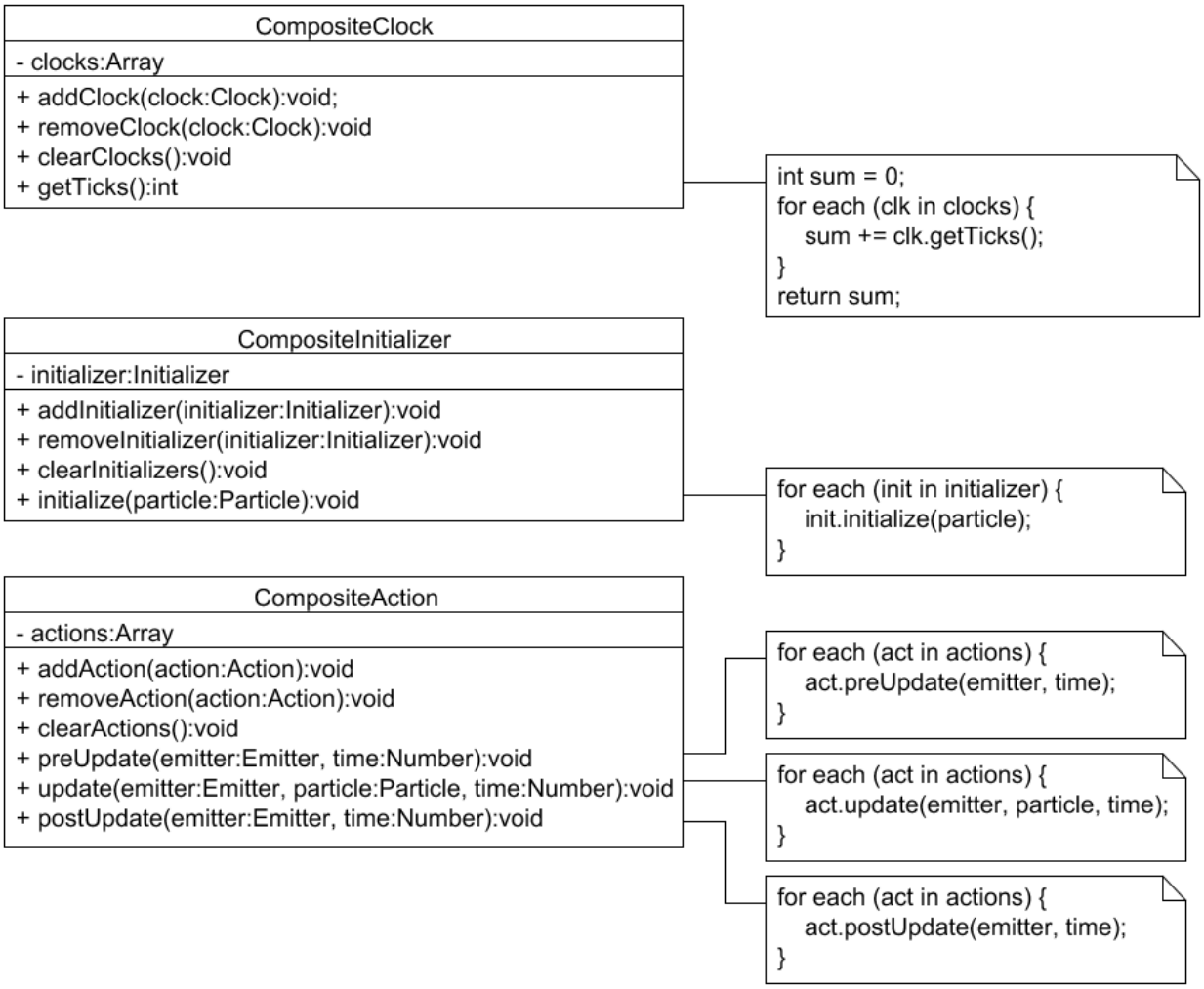
指派給 Emitter 不同的 Clock，即可有不同的粒子生成頻率，例如 **`SteadyClock`** 代表穩定生成頻率，**`RandomClock`** 代表隨機生成頻率。此為 Strategy Pattern，Clock 物件即為 Strategy/Behavior Object。



將多個元素打包

Composite Pattern

使用者可以使用 **`CompositeClock`**、**`CompositeInitializer`**、和 **`CompositeAction`** 類別，分別將多個 Clock、Initializer、或 Action 物件打包成集合，以方便管理，此為 Composite Pattern。Clock、Initializer 和 Action 為 Component，**`CompositeClock`**、**`CompositeInitilizer`** 和 **`CompositeAction`** 則是 Composite。

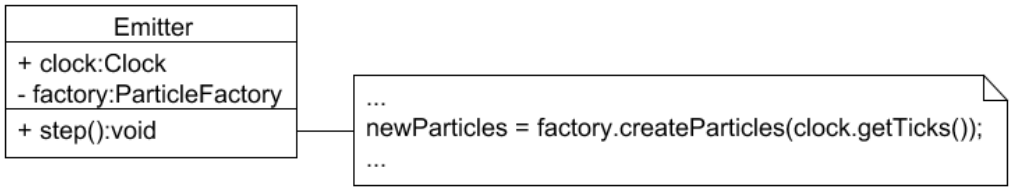


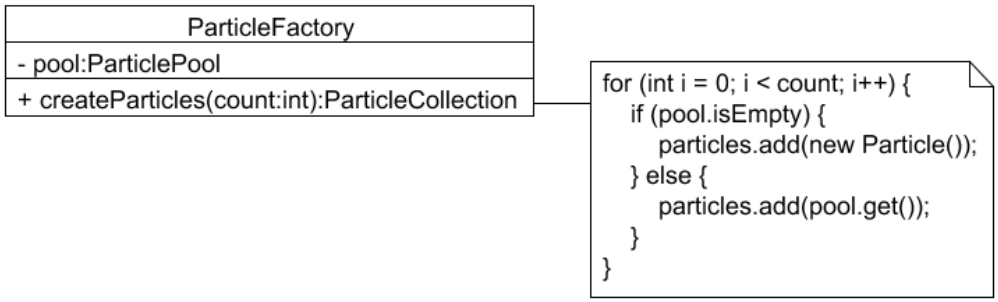
粒子物件生成

Factory Method Pattern

為了增進 Stardust 效能，代表死亡粒子的 Particle 物件會從 Emitter 被移除，然後放到一個暫存物件池(Object Pool)；當需要生成新的 Particle 物件時，若 Object Pool 還有庫存，則會直接從庫存中取得一個 Particle 物件，以省去生成新 Particle 物件的資源消耗。

每一個 Emitter 本身都含有一個 **`ParticleFactory`** 物件，該物件負責生成 Particle 物件交給 Emitter 加入模擬。若 Object Pool 尚有 Particle 物件庫存，則 Factory 會從庫存中取得 Particle 物件；反之，則會生成新的 Particle 物件。Emitter 透過 **`ParticleFactory.createParticles()`** 函式取得粒子物件，此為 Factory Method Pattern；**`ParticleFactory`** 類別為 Factory，粒子物件則是 Product。

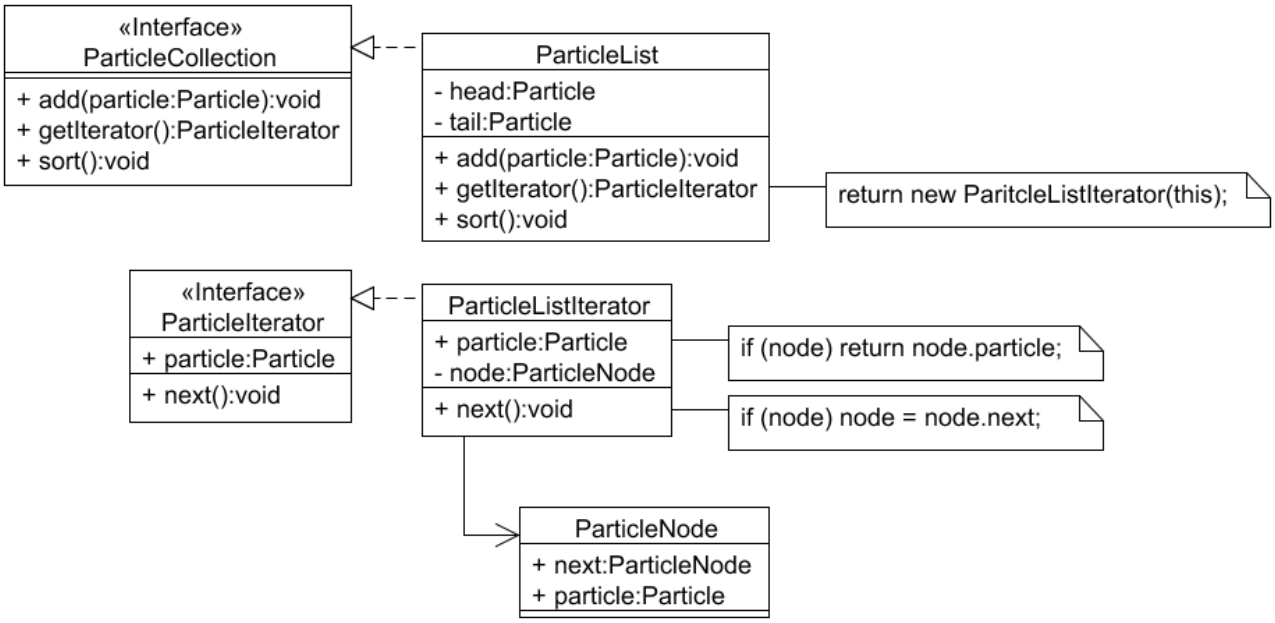




粒子集合 ParticleCollection 類別

Iterator Pattern

實作 **ParticleCollection** 介面的類別可以用來儲存 Particle 物件集合，並且使用 Iterator Pattern 來提供使用者 Traverse 此集合的途徑。**ParticleList** 為一實作 **ParticleCollection** 介面的類別，其背後使用的資料結構是 Linked-List。另外尚有一個 **ParticleArray** 類別，使用的資料結構為單純的陣列。



以下程式碼會將一個集合中的所有粒子位置重設至原點(0, 0)。

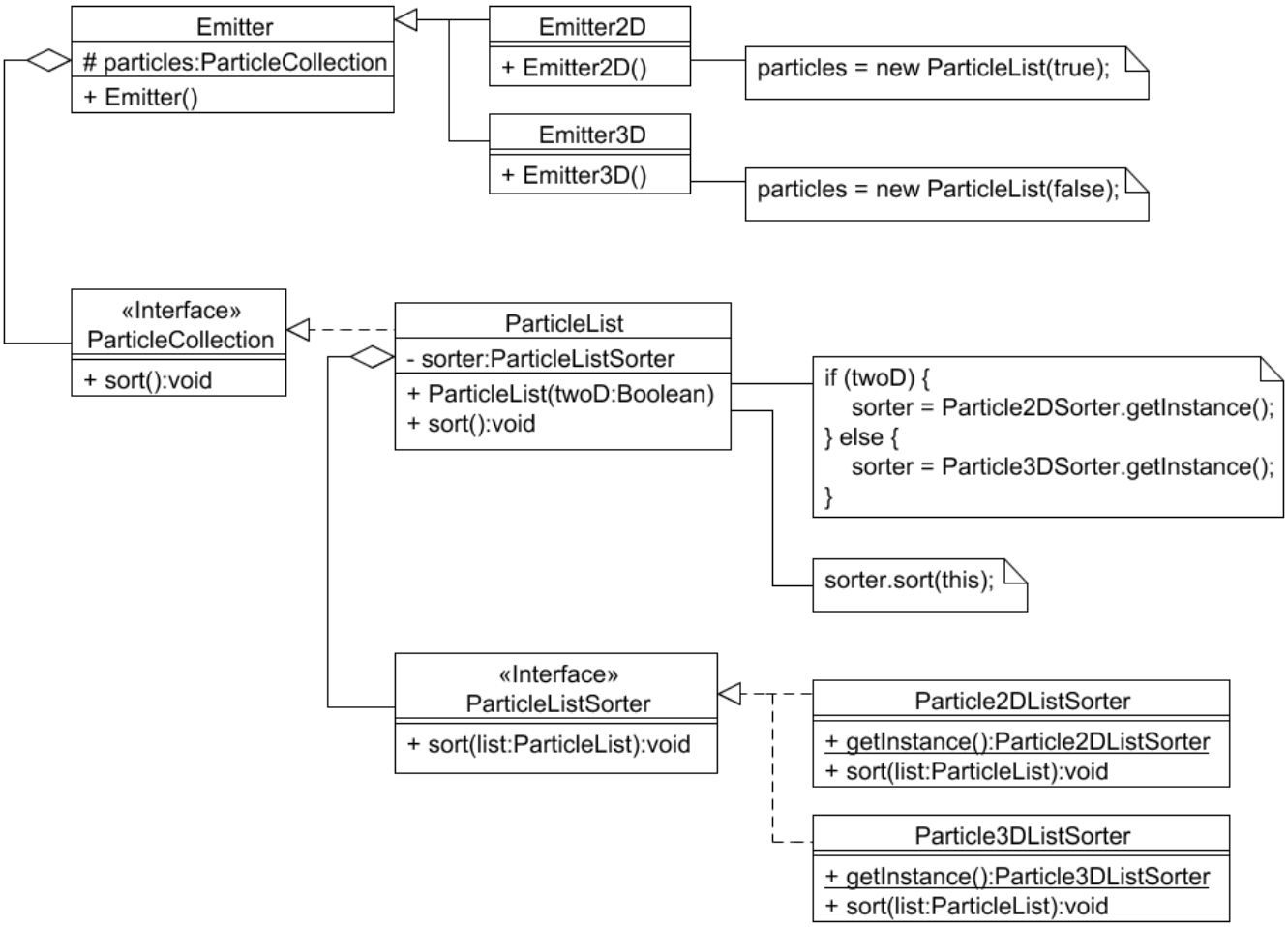
```
var iter:ParticleIterator = particles.getIterator();
var p:Particle2D;
while (p = Particle2D(iter.particle)) {
    p.x = p.y = 0;
    iter.next();
}
```

Strategy Pattern

某些 Action 會需要依照 X 軸排序完畢的 **ParticleCollection**，以增加粒子模擬效率，例如 **Collide** Action 模擬粒子間的撞擊，就需要粒子以 X 軸排序。**Emitter2D** 與 **Emitter3D** 使用不同的粒子類別(**Particle2D** 與 **Particle3D**)，它們使用的 **ParticleList** 物件則利用不同的 **ParticleListSorter** 物件來作排序，此為 Strategy Pattern。**ParticleListSorter** 的角色為 Strategy/Behavior Object。

Singleton Pattern

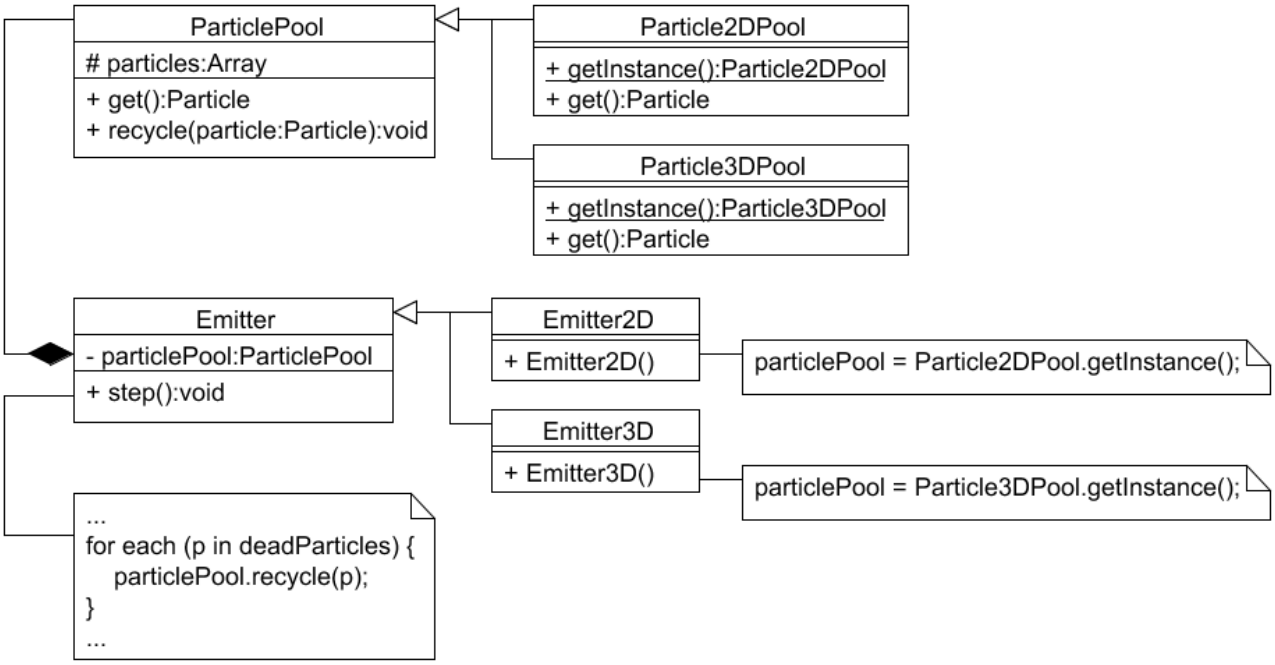
Particle2DSorter 與 **Particle3DSorter** 作的事情都是依照粒子 X 軸座標排序，沒有物件之間的差異，所以各只要一個物件就夠了，此兩個類別都有 **getInstance()** method 可以呼叫，以取得唯一的物件，此為 Singleton Pattern。



從 Particle Pool 中提取粒子物件

Singleton Pattern

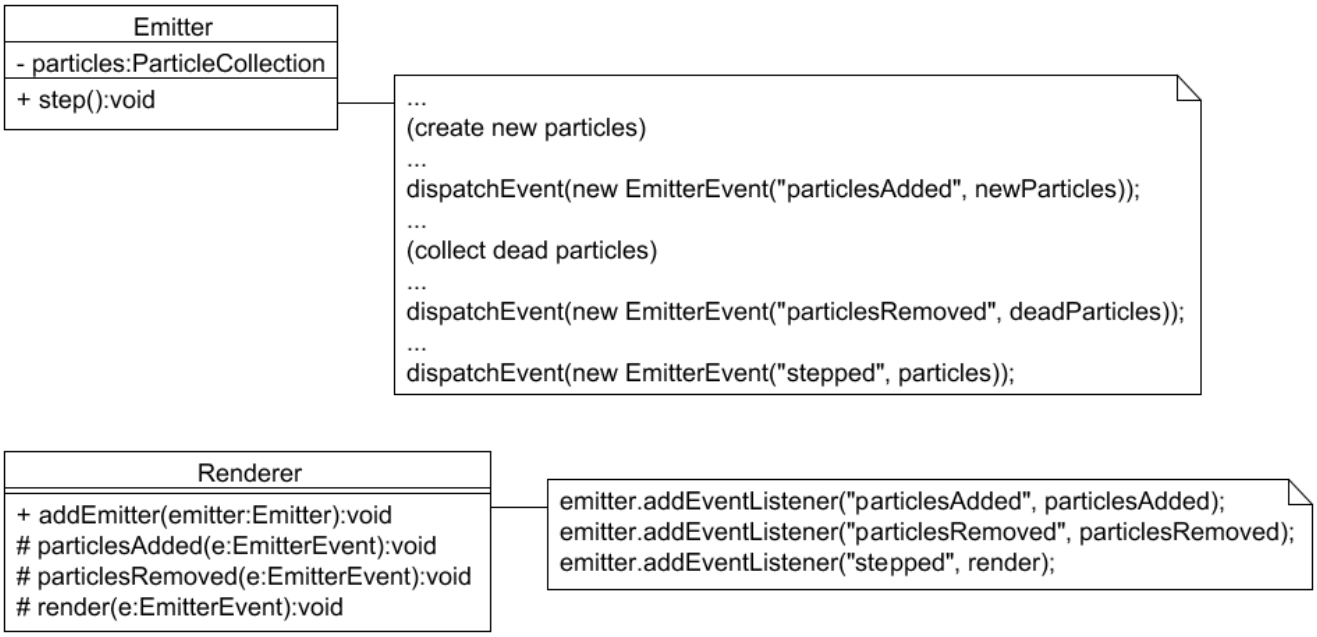
粒子引擎通常都會需要用到大量的粒子物件，為了省去 Particle 物件生成時的運算資源消耗，死去的粒子會被存放到 Particle Pool 以供未來重複利用。Stardust 中回收 Particle2D 與 Particle3D 物件的 Pool 都只有單一個物件來集中管理 Particle2D 與 Particle3D 物件，此為 Single Pattern。



Renderer 的視覺呈現

Observer Pattern

主迴圈 **`Emitter.step()`** 在生成新粒子、移除死亡粒子、和粒子更新完畢時，都會通知所有監聽此 Emitter 的 Renderer，好讓 Renderer 作出相對應的處理。此為 Observer Pattern，Emitter 的角色是 Subject，而 Renderer 則是 Observer。



MVC Pattern

Stardust 整體的視覺化加上 Flash 的互動，可以算是使用 MVC Pattern。Emitter 內部保有的 **`ParticleCollection`** 包含了所有的 **`Particle`** 物件，是屬於存放數值資料的 Model；當 Model 資料更新的時候，Renderer 則會將畫面更新，此為 View。剩下的 Controller 部分，則不包含在 Stardust 中，Stardust 當初就是為了 Flash 互動特效設計的，所以 Controller 的部分，是每一個使用 Stardust 的 Flash 應用程式中的互動：例如「滑鼠左鍵的按下觸發煙火特效」、「滑鼠軌跡上噴發出星星」等。

Stardust 缺乏了 MVC Pattern 中的 Controller，因為 Controller 是要交給 Flash 應用程式設計者來實作的。所以為了展示完整的 MVC Pattern，以下附上一段範例程式碼，展示一個「星星噴發位置跟著滑鼠跑」的效果。

主程式

```
var emitter:StarEmitter
          = new StarEmitter(new SteadyClock(1));

var renderer:Renderer
          = new DisplayObjectRenderer(container);

renderer.addEmitter(emitter);

addEventListener(Event.ENTER_FRAME, mainLoop);
function mainLoop(e:Event):void {
    //更新 point 位置 (Controller)
    emitter.point.x = mouseX;
    emitter.point.y = mouseY;

    //主迴圈
    emitter.step();
}
```

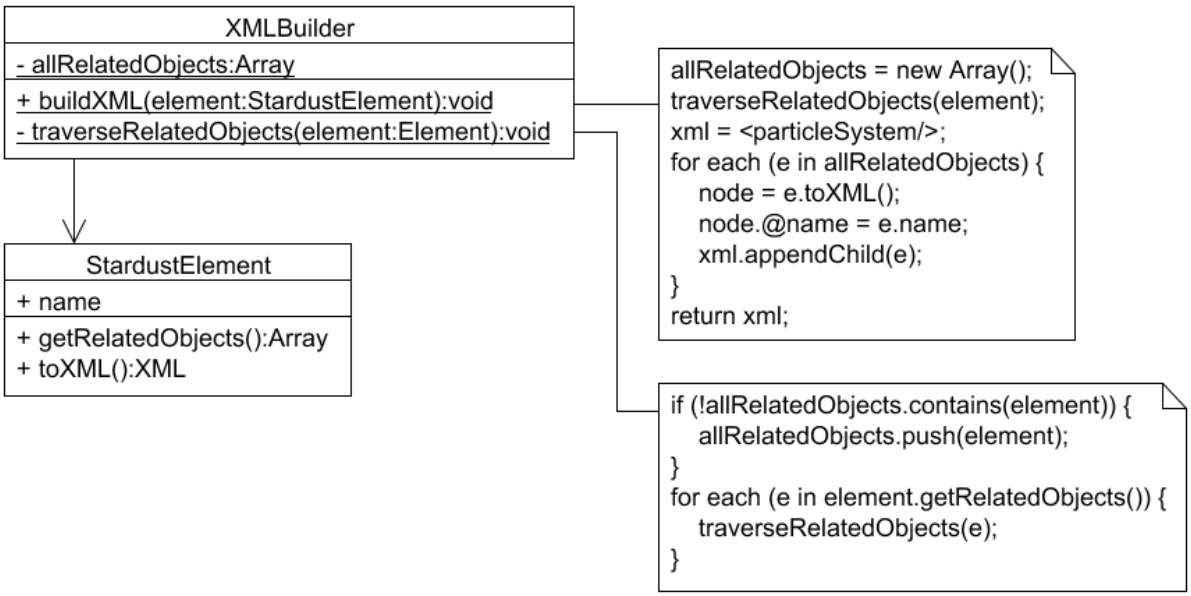
StarEmitter 類別

```
//繼承 Emitter2D
class StarEmitter extends Emitter2D {

    //給主程式使用的 Reference
    public var point:SinglePoint = new SinglePoint();

    public function StarEmitter() {
        //initializers
        //噴出星星
        addInitializer(new DisplayObjectClass(Star));
        //生命長度介於 50±10 範圍內
        addInitializer(new Life(
                  new UniformRandom(50, 10)));
        //位置等於 point 的(x, y)座標
        addInitializer(new Position(point));

        //actions
        //老化
        addInitializer(new Age());
        //生命歸零時死亡
        addInitializer(new DeathLife());
    }
}
```

XML 序列化

Visitor Pattern

要將一個粒子系統序列化為一筆 XML 資料 (每一個 XML 結點代表一個元素)，必需取得跟此系統相關的所有其他元素的參考，這就是 **`StardustElement.getRelatedObjects()`** 函式的目的：它會回傳包含跟這個元素相關的元素陣列，而 **`XMLBuilder`** 類別扮演 Visitor 的角色，以遞迴的方式走遍所有元素，呼叫它們的 **`toXML()`** 函式來取得序列化後的 XML 結點。



Stardust 效能檢討

Stardust 顧及到系統維護的容易性以及未來擴充性，大量使用 Design Pattern，類別數量非常多，分工非常細；然而另一方面，卻也因為類別間需要頻繁地相互溝通，導致 Function Call 次數頗多：例如若有 1000 個粒子和 5 個 Action 物件，每一次呼叫 **`Emitter.step()`** 就會呼叫 5000 次 **`Action.update()`**；這對使用虛擬機器的語言而言(如 ActionScript、Java、C#)，是拖垮效能的致命傷之一。

我(周明倫)曾於 WonderFL 上看到有人將所有函式功能展開寫死在一個 Function Call 中，而得以流暢地模擬 25000 個粒子；相較之下，若使用 Stardust 製造同樣的特效，一次只能模擬 1000 個粒子，效能差距非常大。我曾經試過把 Stardust 功能展開寫死成單一 Function，果然效能就大幅上升了。不過我不認為這是一個程式正確的寫法，一個大型程式專案中如果使用這種方法撰寫程式，儘管可以得到不錯的效能，未來維護起來卻將會極度繁瑣與困難；這是一體的兩面，我覺得犧牲程式效能，換極容易維護的程式，是非常值得的。其實 Stardust 的效能，在實際粒子特效應用上不是什麼大問題，因為一般的 Flash 粒子效果用到 150 個粒子就算非常多、非常華麗了，離 1000 還有很大一段差距，執行起來是非常順暢的。

現在我專注的，將是如何把 Stardust 因為大量 Function Call 而造成的效能減損降低。例如在 **`Emitter.step()`** 中要利用 **`Action.update()`** 來更新所有粒子，會需要雙迴圈；原本的雙迴圈程式碼長這樣：

```
for each (var a:Action in actions) {
    var iter:ParticleIterator = particles.getIterator();
    var p:Particle2D;
    while (p = Particle2D(iter.particle)) {
        a.update(emitter, p, time);
        iter.next();
    }
}
```

但我後來發現 Iterator 迴圈的速度比 for…each 迴圈的速度慢，於是我把程式碼改成這樣：

```
var iter:ParticleIterator = particles.getIterator();
var p:Particle2D;
while (p = Particle2D(iter.particle)) {
    for each (var a:Action in actions) {
        a.update(emitter, p, time);
    }
    iter.next();
}
```

將 Iterator 迴圈與 for…each 迴圈內外對調，讓前者只走完一次，效能就改善不少了。

或許我還可以把一些常用的 Action 類別的程式碼展開寫死、包裝成單一 Action 類別，理論上就可以把最大模擬粒子數推至 5000 以上。

Stardust 未來的改善空間還很大(當然嘛，跟展開寫死的程式效能差了至少 25 倍)，我會繼續尋找效能改善的可能性，然後繼續推廣 Stardust。

課程心得

周明倫

Design Pattern，不愧是眾 OOP 程式設計師長久以來的智慧結晶；在上一個暑假自修完 Head First Design Patterns 與 GoF Design Patterns 之後，原本對 Interface 扮演的角色不甚了解的我，忽然有種豁然開朗的感覺：各種機制存在的目的都說得通了。

我對 Design Pattern 躍躍欲試，於是放棄原本的 Emitter 專案，重新打造一個 Stardust 粒子引擎。在這其間的許多大小專案，我也將 Design Pattern 的所學應用上去，成就感真是非常大。

這學期的設計模式課程，讓我首次認識了 NetBeans IDE，並且令我重拾對 Java 視窗應用程式開發的興趣與熱情。教授的課程，讓我對原本不是很了解的 Java API 設計有了新的認知；例如大一下的 Java 課，我對於教授提供的 Sample Code 中寫到的這段程式碼，一直無法了解其背後意義：

```
new BufferedReader(new InputStreamReader(System.in));
```

學完了 Decorator Pattern 之後，才真正瞭解這樣子的寫法意義為何，以及當初 Java API 為什麼要這樣設計。

另外，隨著這學期課程的進展，我也漸漸認識到，自己以前撰寫程式的時候，其實已經不知不覺用到了不少 Pattern；當時只是覺得這樣子寫程式似乎比較好管理與擴充，結果沒想到竟然無意間使用到了 Design Pattern，心裡又驚又喜。在我了解到我這些程式的撰寫方式是 Design Pattern 之後，我就會開始養成習慣多留意能夠應用 Pattern 的地方，讓我撰寫程式的時候更能得心應手，並且會更加考慮到未來的可維護性與擴充性。

這學期初我修習系上的網路與多媒體實驗時，助教要我們製作一個小畫家程式，以當作開始正式做驗之前的練習。我發現幾乎所有實驗室的人，都把程式碼塞在一個 JFrame 類別中，近千行的程式碼看起來頗不容易維護；而我因為暑假已經有自修 Design Pattern，就想辦法將 Pattern 應用在這個專案中。我利用了 Command Pattern 把線段、圓形、矩形、橡皮擦、檔案讀寫等功能分離成單一 Command 類別，又用 Memento Pattern 作出 Undo/Redo 的功能：其實就是對一個 Stack 執行 Command 物件的 Push 與 Pop 而已。其

他同學遇到一個問題，就是由於一條短直線線段是一個 Command，每一次 Undo 都只會將一個小線段移除掉，而不是把滑鼠畫出來的一整個筆劃都消除;這個問題雖然讓我稍為煩惱了一下,但是我馬上就想到可以用 Composite Pattern 將一筆畫的多個線段 Command 包裝成一個 Composite Command,再把這個 Composite 給 Push 到 Stack 中，如此一來一次的 Undo 就可以將整個筆劃完全消除。相較於其他同學需要在近千行的程式碼中勉強湊出解決方法，我只需要再寫一個 Composite Command 類別就可以了，Design Pattern 的威力與實用性真是不容小覷。

很謝謝教授這學期開了這門課,幫助我釐清了一些自修時還不是很了解的觀念。像是一語道破 Flyweight Pattern 本質上就是 Singleton Pattern 的變體,還有 Visitor Pattern 為什麼破壞了 Information Encapsulation。這門課對我未來開發的程式專案，真是助益匪淺！

林昭瑋

老實說，在修習教授開的軟體設計模式前，我是一個從來沒接觸過 Object-Oriented 語言的新手，只有接觸過 C++，具備一般的程式語法能力而已。為了進入 OO 的世界，我去買了 Head First 系列的 JAVA 入門書，

我覺得 OO 的世界真的很有趣，它使我能夠將自己的想法變成各種 Object，然後進行排列和溝通，而 Design Pattern 則是提供了我在不同目的下如何布好各個 Object 的樣板，使 program 之後的擴充性和可維護性大大提升，並具有結構性的規劃，使 Program 中不會有一堆 Reuse 的程式碼，讓 Programmer 之後在看自己的 Program 時，能有更深刻的了解。

從一開始的 Template Method Pattern、Strategy Pattern、Visitor Pattern、Factory Pattern 的各種變體、Builder Patten 到最近的 Adapter Pattern、Façade Pattern、Mediator Pattern 等，我覺得每一個 Pattern 都很有味道，切合他本身的 Intent，雖然有些 Pattern 的結構上來說是類似，甚至是相同的，但其 Intent 的分歧，讓每個 Pattern 都具有自己的特色。

譬如說 Strategy Pattern 和 State Pattern，他們在結構上根本是一樣的，但由於 Strategy Pattern 的目的是將一些同樣種類的 Algorithms 包裝起來,使其能夠輕易的改變使用的 Algorithm，但 State Pattern 的目的是要使 Object

可以根據內部 State 的改變而讓 Behavior 改變。 這兩者的差別在於前者使用者了解目前我要哪一種 Behavior，像 Head First 的例子，我們知道 **RubberDuck** 的飛法是 **FlyNoWay**，**RedHeedDuck** 是 **FlyWithWings**，並且我們能夠在 Runtime 時依照需求去改變他的行為，新增的 **FlyWithRocket** 即為一個例子，而後者以 **GumMachine** 為例，我們並不知道目前的 State 在哪裡，只要他會產生對應的 Behavior 就好。

像這種 Pattern 之間的比較，教授在授課時講了不少，在 Head First 的 Design Pattern 一書中每一章節的結尾附近也會出現，這種比較不僅讓我了解兩個(或多個)Pattern 之間的差異，也讓我更清楚個別 Pattern 的 Intent，我認為這種比較是這門課不可或缺，可以說是很重要的一環，也讓未來在實際 Coding 時，在決定使用哪種 Pattern 能有更明確的考量。

相關連結

Stardust Particle Engine 首頁
http://code.google.com/p/stardust-particle-engine

FlashDevelop 編輯器
http://www.flashdevelop.org/community

Adobe Systems 官方網頁
http://www.adobe.com

CJ's Blog (含 Stardust 開發現況與開發日誌)
http://cjcat.blogspot.com

參考資料

[1] Flint Particle System 粒子引擎
http://flintparticles.org

[2] Box2D 物裡引擎
http://www.box2d.org

[3] ***ActionScript 3.0 Design Patterns: Object Oriented Programming Techniques*** (2007). Adobe Developer Library. William Sanders, Chandima Cumaranatunge.

[4] ***Foundation ActionScript 3.0 Animation: Making Things Move!*** (2007). Friends of ED. Keith Peters.

[5] ***Essential ActionScript 3.0*** (2007). Adobe Dev Library. Colin Moock.

[6] ***Head First Design Patterns*** (2004). O'Reilly Media. Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra.

[7] ***Design Patterns: Elements of Reusable Object-Oriented Software*** (1994). Addison-Wesley Professional. Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides.