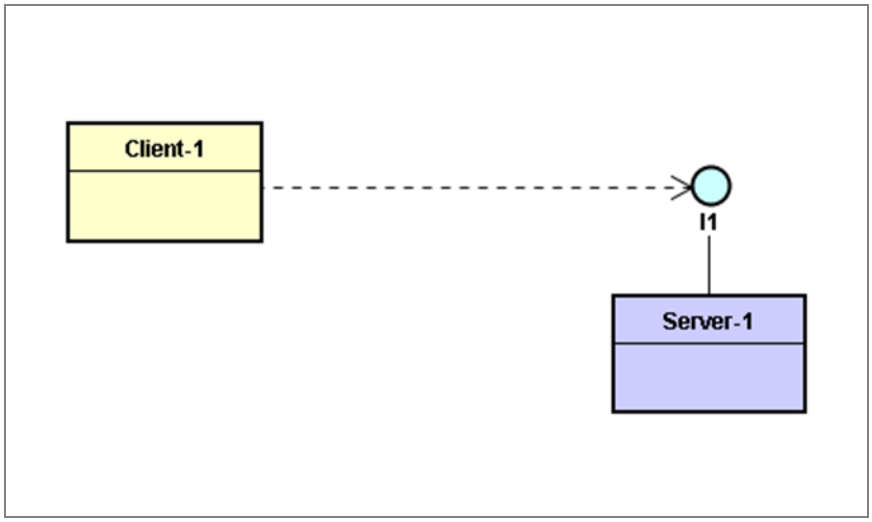
G07_接口设计之美_通用性接口设计范例

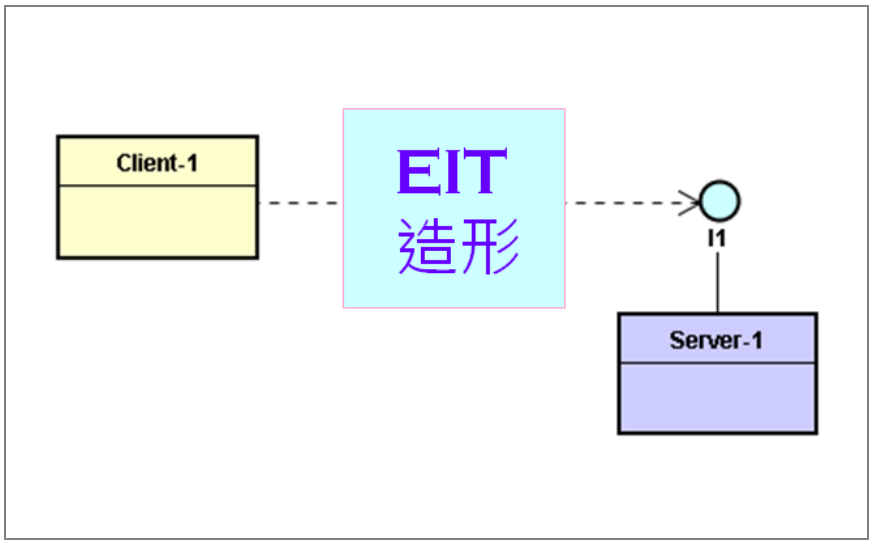
内容：

1. 复习：通用性<I>设计

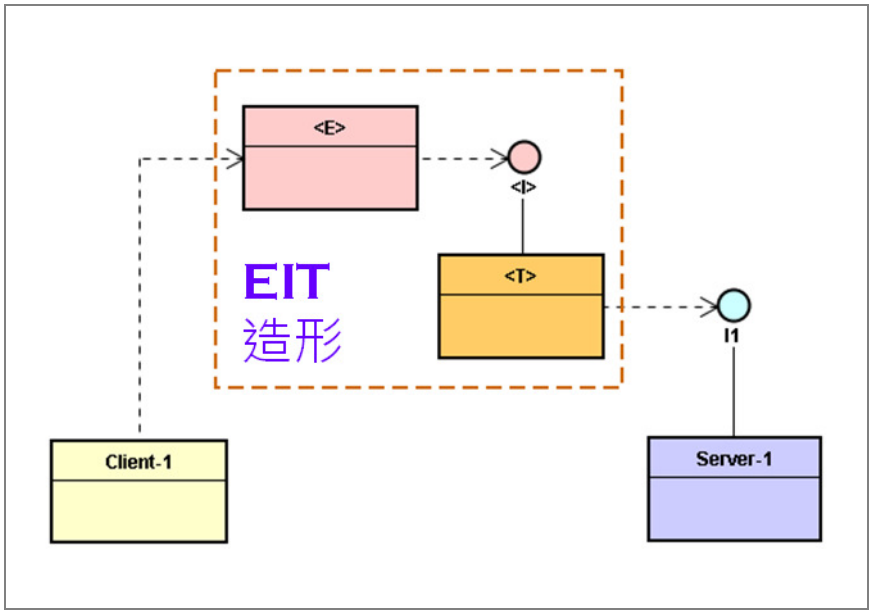
首先，看看一个 Client/Server 结构：



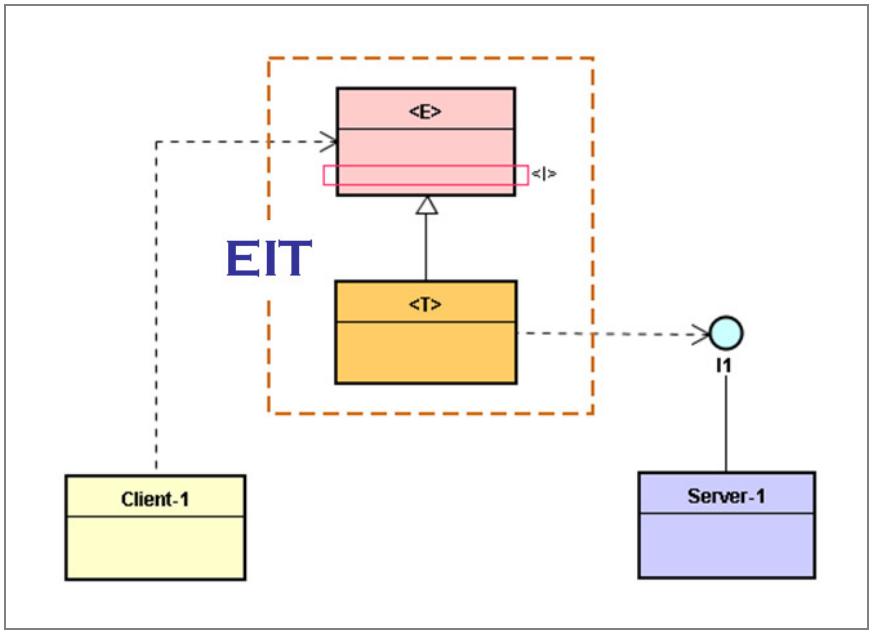
兹增添一个 EIT 造形，如下：



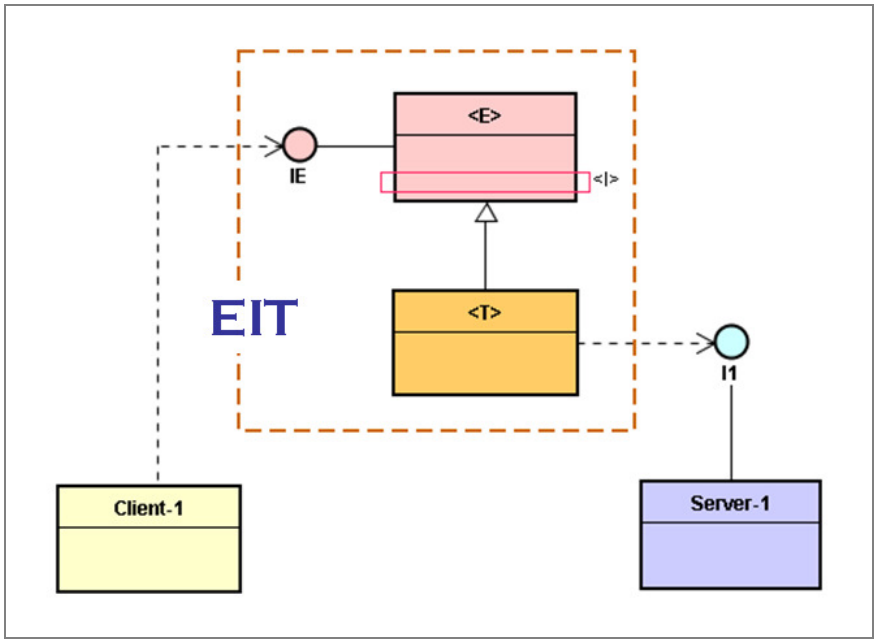
此 EIT 造形的内部结构，如下：



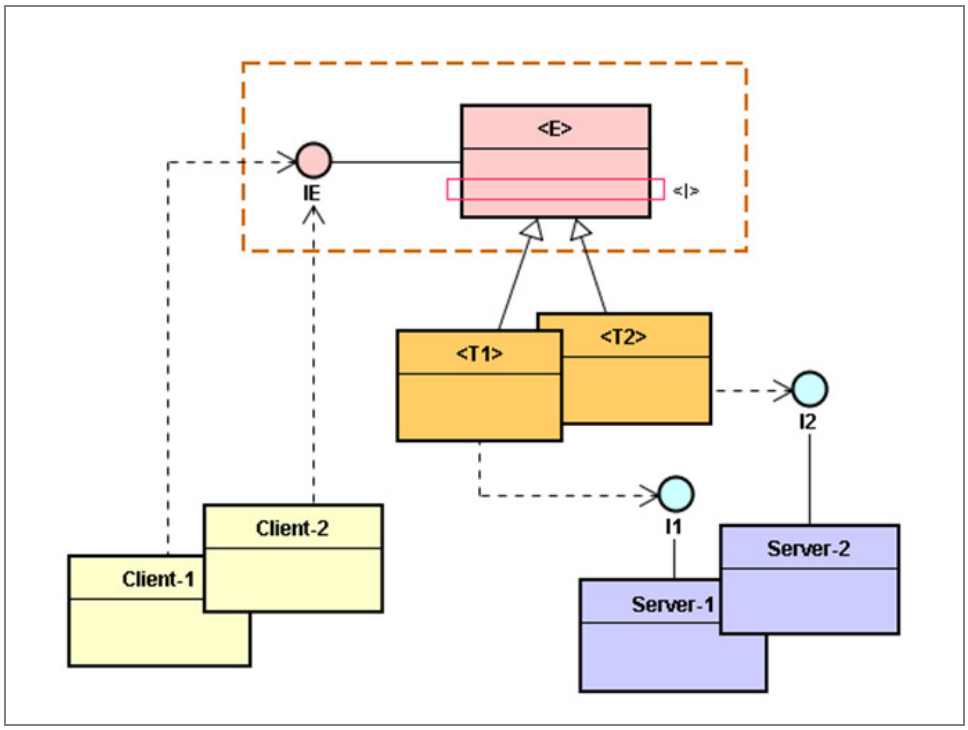
EIT 造形有两种主要变形，兹采取另一个变形，上图就相当于：



这个<E>可提供一个接口<IE>给 Client 来使用，上图也相当于下图：



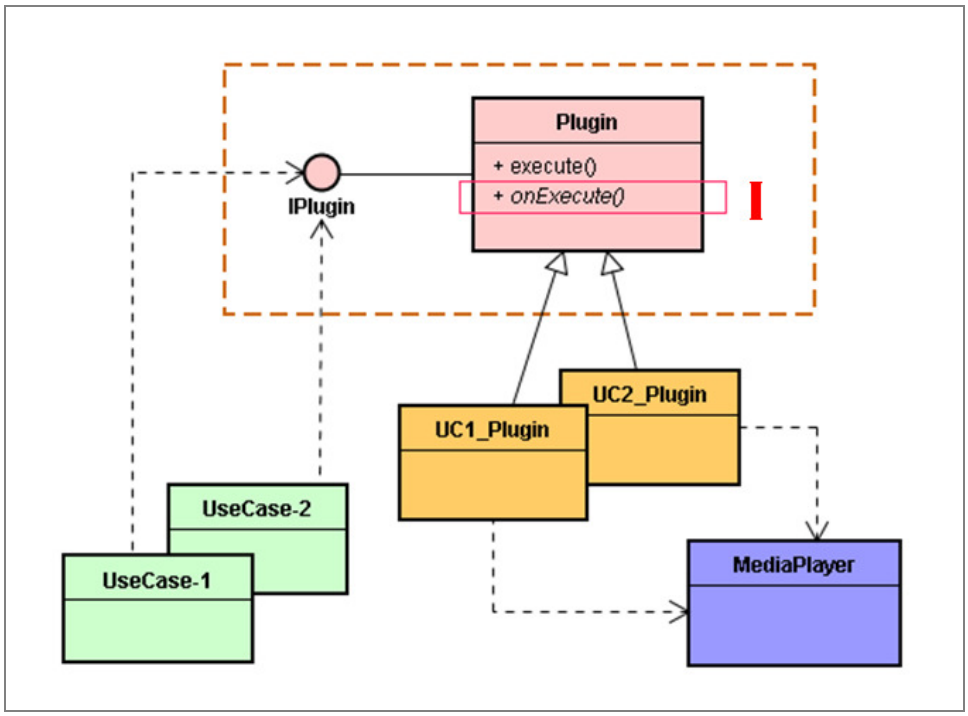
基于此架构，兹再增添一组 Client-2/Server-2 结构，他们共享同一组<E&I>了；此<I>和<IE>就成为通用性接口了。如下图：



其中的 {Client-1, T1, Server-1} 与 {Client-2, T2, Server-2} 都可独立开发，只是共享<IE>和<I>通用性接口而已。这种独立性，可搭配敏捷迭代过程而独立开发与产出。亦即，{Client-1, T1, Server-1} 与 {Client-2, T2, Server-2} 可各归属于不同跌代的开发&产出标的。

2. 通用性<I>支撑敏捷用例(Agile-Use Case)

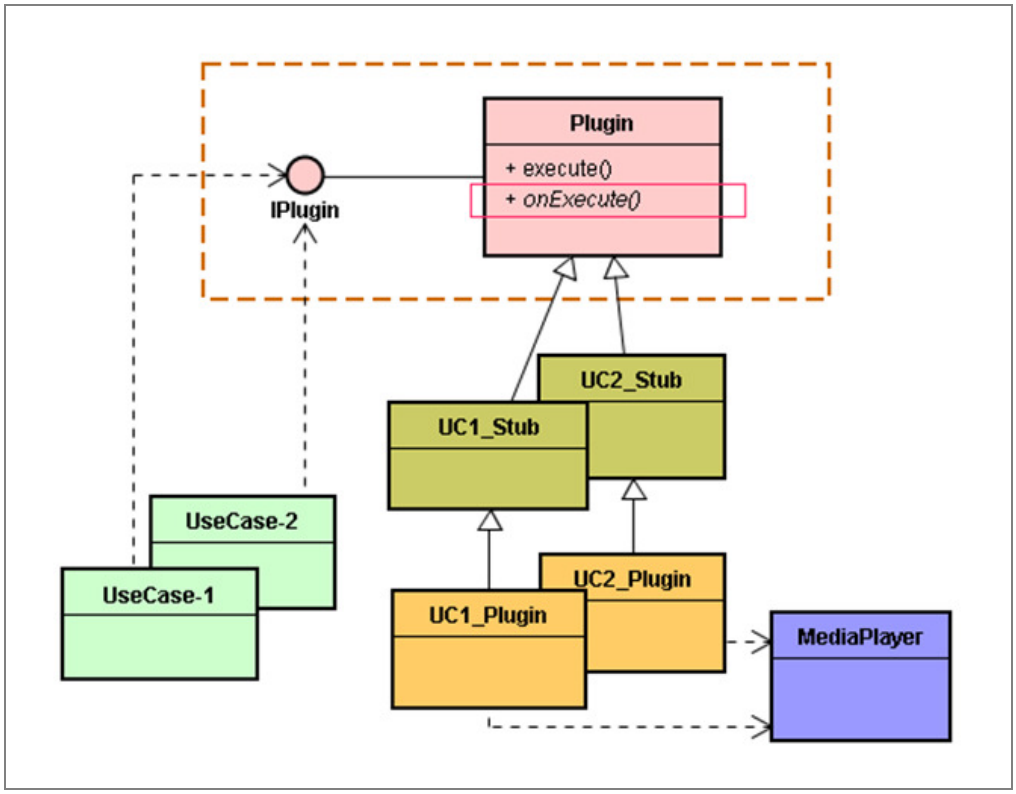
也就是基于上述的特性，常常可拿来搭配敏捷用例(Agile-Use Case)，因为在敏捷开发思维里，需求(Requirement)分析可以从”用户故事(User Story)”所叙述的愿景(Vision)出发，随着敏捷跌代过程，逐渐收集更多需求细节，并表述于用例图(Use Case Diagram)和用例叙述(Use Description)文件里。由于一个用例就是系统提供给用户的一项完整服务(Service)，其幕后隐藏了一个<I>。那么，一群用例就能共享一个通用性<I>了。于是，将用例内涵与上图的基本架构结合起来了。就成为下图：



此架构图展现了两个优越的特性：

- 各组用例之间都是独立的。例如，{UseCase-1, UC1_Plugin} 与 {UseCase-2, UC2_Plugin} 两者是独立的，可以分别归属于不同跌代的工作产出目标。虽然各组用例共享了通用性接口，但都能用有自己的特殊性接口。
- 透过 EIT 造形的转换接口之后，Client **部分与 Server** 部分是独立的，创造了两端各自的变动自由度。例如，UseCase-1 的 UI 画面的改变，不会受制于 MediaPlayer 的接口(函数)；此外，MediaPlayer 的接口改变了，只需改变 UC1_Plugin，而不会波及各 UseCase-1 或 UseCase-2 等。

接着，还可以添加上曹操类(Stub)，它能将通用性的<I>，转换成特殊性的接口。如下图：



之后，基于此项结构，UseCase 个数都可以无限增加，而且都透过通用性的 IPlugin 接口的 exectute()函数和 Plugin 类的 onExecute()函数来做为通信渠道。于是，实现了各用例的通用性接口设计了，也实现了敏捷用例(Agile-Use Case)的目标。☆

G07_接口设计之美_通用性接口设计范例

内容：

◇ 用例(Use Case)概念与需求表述

前言：

- Client-Server 是一个大家熟悉的架构，其 Client 与 Server 之间隐藏了一个接口<I>。
- 然而，{Client-1, Server-1}、{Client-2, Server-2}和{Client-n, Server-n}所隐藏的<I1>、<I2>和<In>可能都是不一样的。
- 那么，我们又如何藉由 EIT 造形来定义出统一而共通性的<I>呢？基于这项通用性<I>，就能将敏捷迭代的产出，持续整合起来。
- 接下来，就拿敏捷需求分析中常见的”敏捷用例”(Agile-Use Case)来做范例，基于这项通用性<I>，就能将敏捷迭代所产出的 Use Case 相关代码，持续整合起来。

3. 用例(Use Case)概念与需求表述

3.1 用例概念的起源

建置系统时，无论是软件系统或其它领域的系统，所面对的第一关就是系统需求(Requirement)。需求就是用户(User)所期待于系统者，也是开发者想藉之而讨好客户的东西。然而用户所期待的，常常远超出开发者的最大能力之所及。由于这种落差是常态，所以我们需要一种有效的方法来让双方逐渐地磨合，进而达成共识(Agreement)。自从 1992 年 Jacobson 提出”用例(Use Case)概念以来，它逐渐成为萃取和磨合双方的最常用途径。

采取”用例”的第一个步骤是，拿它来表达从用户脑海所萃取的需求知识。这像刚采撷出来的钻石一般很宝贵但没有光彩。因之必需进行第二步骤，就是表达开发者构想中能力所及又创意的卖点。接着进入第三步骤，让两者展开知识与构想的交流，促进磨合，逐渐呈现光彩夺目的好钻石。

3.2 Why Use Case?

用例在确保系统的可用性(Usability) 上，是个强有力的工具。它擅长于表述用户为什么(Why)要使用(Use)系统，或如何(How)使用系统。想设计出更适用的软件系统时「使用」才是最重要的问题所在。包括表达出许多「为什么」(Why)。例如，为什么需要这软件？为何用户要去接触这软件？用户欲达成什么事？

当我们用心探索围绕在「使用」的一连串「为什么」问题之后，就能导出高

度可用性的软件了。在这过程中，就已厘清了用户的外部行为(External Behavior)，以及其跟系统的交互情形(Interaction)，也因而定义出系统应有的外部行为，以协助用户来完成其工作。

于是，可知在我们弄清用户为什么「使用」这系统时，其实就已定义出用户对这系统的需求了。用例能描述用户的外部行为及其与系统的互动情形，也因而表达出了系统的责任(Responsibilities)，亦即是用户对系统的需求了。

总之，用例是个强有力的工具让软件人员暂时不考虑软件系统内部的行为和结构，而专注于厘清用户「为什么」去「使用」这系统，搭配敏捷迭代过程，充分正确地掌握用户的最新需求，然后重构系统设计，并迅速落实为代码。

3.3 认识 Use Case 及其剧景(Scenario)

用例描述人们使用某系统时，其使用途径。每个人去使用系统时，其目的是期望系统提供服务或产品。当系统在提供完整的服务或产品的过程中，会执行一连串的小活动：在这活动之中，也常会跟用户沟通，取得用户的指示而调整其活动或顺序。例如，人们去麦当劳餐厅买汉堡时，柜台人员会向顾客寻问是「外带」还是「内用」而决定其包装程序。因之，每个人去买汉堡时，其使用「麦当劳服务系统」的途径，包括用户与服务人员的对话(Dialog)过程都可能不一样(例如有人先问价钱，有人先点餐等)。其中，每个人使用系统的途径(或其对话过程)就是个特殊的实例(Instance)，特别称为剧景(Scenario)或实景。

虽然每个人使用系统的剧景会有些差异，但是若用户的目标(Goal)是相同的，则其剧景常会极为相似。那么这些类似的剧景的集合就是个类(Class)，这个类就特别称为"用例(Use Case)"，而其实例就称为"剧景(Scenario)"了。

3.4 需求的表述(Representation)：基于 Use Case 模型

Use Case 模型包括：用例叙述(Use Case Description)和用例图(Use Case Diagram)。

用例叙述(Use Case Description)

用例叙述是针对用户与系统之间的对话(Dialog)过程，以文字描述出来。根据 Wirfs-Brock 的建议，可只描述用户与系统之间的对话如下：

Use Case 叙述

UC:買漢堡

{

user action	*system response*
指定購買漢堡並付款	
	收錢、做漢堡 給收據和漢堡
拿走漢堡	

}

这就是「用例叙述」(Use Case description)了。

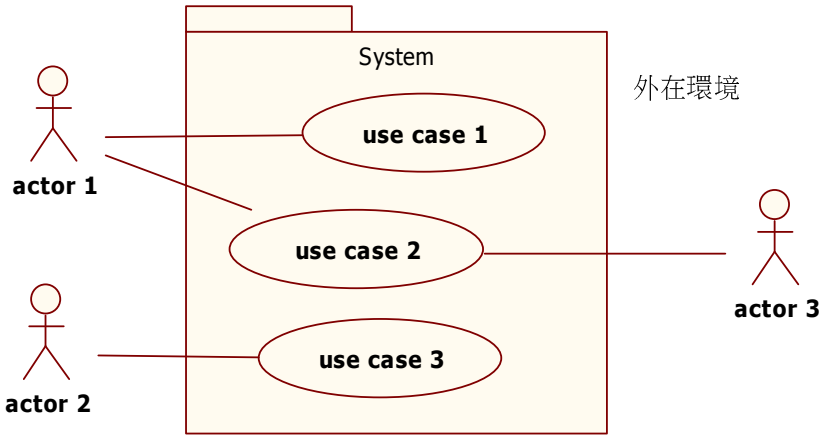
用例图(Use Case Diagram)

系统含有许多用例，而环境里的人或物（即用户）在不同时刻，可能扮演不同角色（怀有不同的目的）来使用系统中的不同用例，以取得系统的不同服务。「角色」(Role)代表一群对系统怀有相同兴趣或目的人或物。当这些人或物每次使用系统时，系统就执行一个特定的用例来为其服务，协助其达成目的。角色含有特定的目的，而其目的决定系统中的用例内涵，亦即决定系统应有的功能。

role —→ goal or goals —→ use cases

在电影上，我们称扮演某特定角色的人或物为「演员」(Actor) ，于是，在 Use Case 模型(包含用例叙述和用例图)也称扮演共同角色的用户们为 Actor。这 Actor 代表「一群」怀有共同目的的用户，在 Use Case 模型必须表达出 Actor 与用例之间的关系，才能让人们去了解与检验用例。

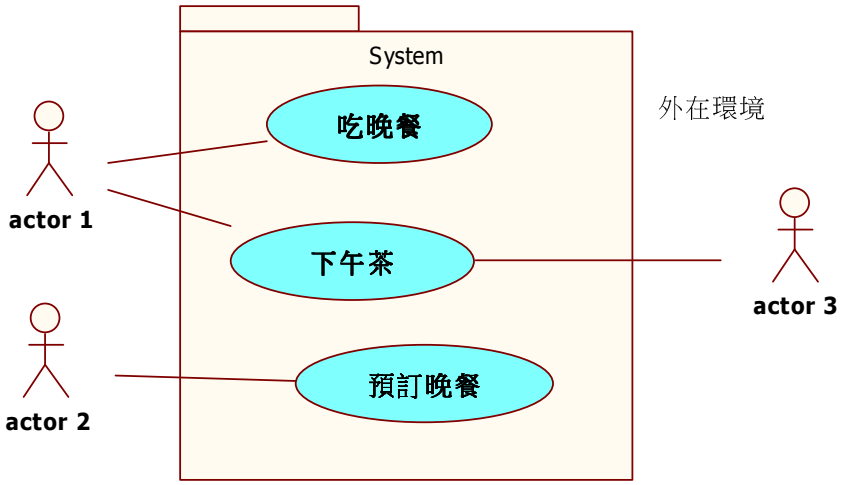
例如在下图里，当您扮演 actor 2 时（怀有一个目的），就可使用系统，此时系统就执行 use case3 来协助您完成目的。同样地当您扮演 actor1 时，就可使用 use case 1 或 use case 2 来达成您的目的。所以，用例图含有良个基本元素：Actor 及 Use Case。如下图：



这就称为用例图了，其表示了，系统将提供 use case 1 来协助 actor 1 达成其目的，也提供 use case 2 来达成 actor1 的另一个目的；此外，也提供 ues case 3 来替 actor 2 达成目的。

以一家餐厅为例

一家餐厅的用例图，可能如下：



有了用例图之后，还可以搭配用例叙述，来表述 Actor 使用系统之途径，以及系统应有的行为。在此用例叙述里，只描述系统的外部行为，及其与 Actor 之互动而已，并不描述系统的内部行为。例如，针对上述的用例图里的 3 个用例，可各写一个用例叙述来表述之。例如：

用例叙述

UC: 吃晚餐

{

user action	*system response*
点餐	
	上开胃菜
	上主餐
请上点心	
	上点心
买单	
	结账
	开收据
拿走收据	

}

这就表示出系统的应用功能和行为了，也就是表述了系统的一项需求了。依样画葫芦，可继续针对另外两个用例：<UC：下午茶>和<UC：预定晚餐>，来撰写它们的用例叙述。☆

G07_接口设计之美_通用性接口设计范例

内容：

◇ 「播放 MP3 音乐」范例的 Agile-Use Case 模型

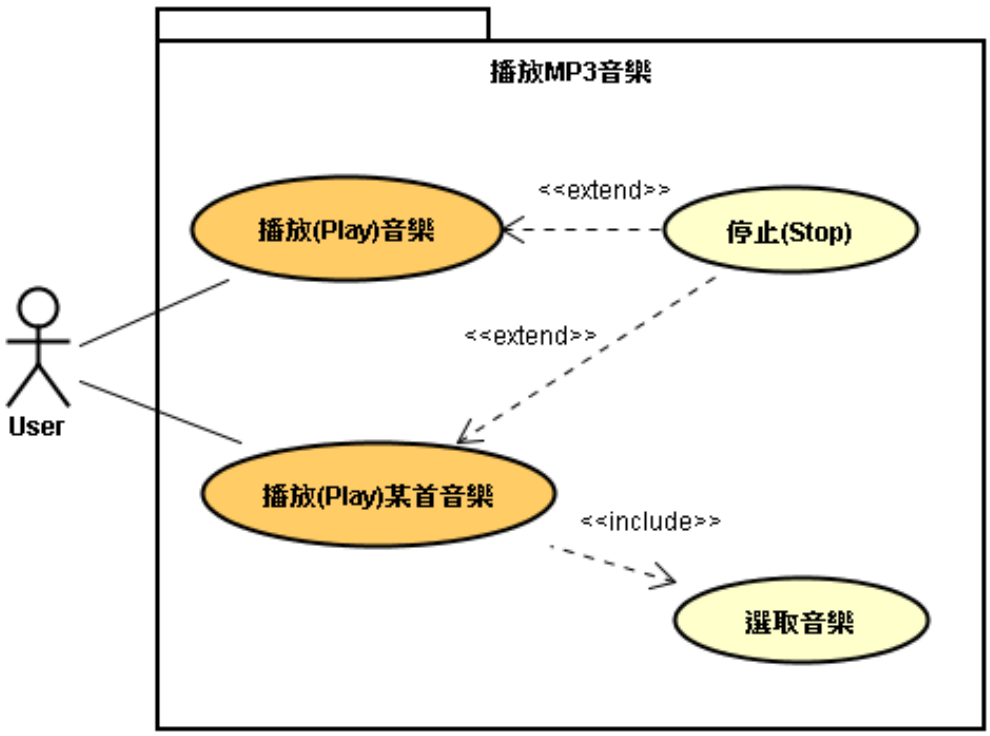
前言：

- Use Case 模型偏于需求的表述(Representation)，其主要的表述形式有二：用例叙述(Use Case Description)和用例图(Use Case Diagram)。
- 随着敏捷迭代过程，Use Case 的数量会逐渐增加，其表述内容也会逐渐细腻而完整。

4. 「播放 MP3 音乐」范例的 Agile-Use Case 模型

4.1 用例图

茲绘制其用例图，如下：



从这用例图里，可以看出来，目前含有两个用例：

- UC-001：播放(Play)音乐。
- UC-002：播放(Play)某首音乐。

从上图可以看到，用例之间有两种关系，分别是「包含」(include)和扩充(extend)关系。

「包含」(include)关系

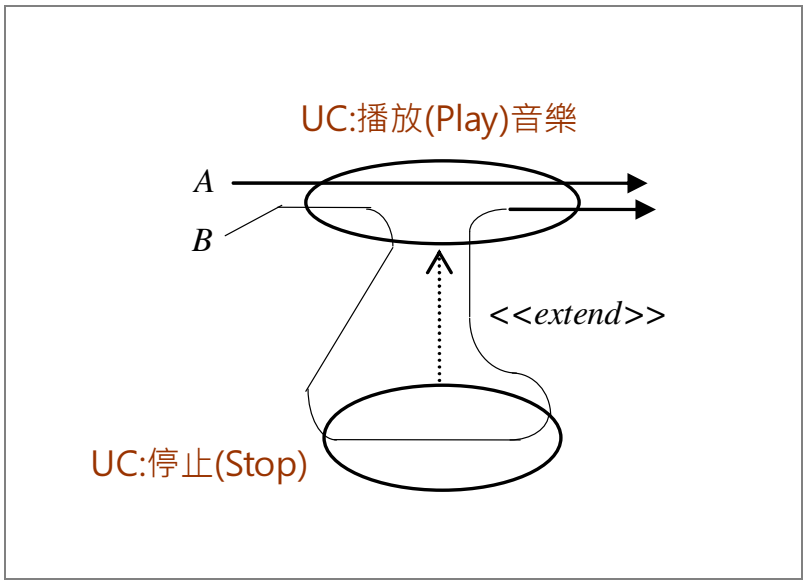
两个用例之间可以有「包含」(include)关系，用以表示某一个用例的对话(Dialog)流程中，包含着另一个用例的对话流程。一旦出现数个用例都有部份相同的对话流程时，将相同的流程记录在另一个用例中，前者称为「基础用例」(Base Use Case)，后者称为「内含用例」(Inclusion Use Case)。如此，这些基础用例就可以共享内含用例，而且未来其它的用例只要建立包含关系，就可以立即享用已经在其它用例定义好的相同对话流程了。

简而言之，<<include>>关系是来自于抽象的动作(Abstraction)，即从数个不同的用例之中，抽象出共同的部分，而成为可以重用的「内含用例」。「内含用例」是「基础用例」对话流程中的必备部分。例如，上图里的<UC：选取音乐>用例是<UC：播放(Play)某首音乐>用例对话流程中”不可或缺的”一部分。

扩充(extend)关系

两个用例之间可以有「扩充」(extend)关系，用以表示某一个用例的对话流程，可能会依条件而临时插入到另一个用例的对话流程中;前者称为「扩充用例」(Extension Use Case)，后者称为「基础用例」(Base Use Case)。有了扩充关系后，便可以将特定条件下才会引发的流程表述于扩充用例中。当执行基础用例时，可以单纯地执行基础用例的对话流程；但是在特定条件发生时，则会额外插入并执行扩充用例所表述的对话流程。例如，上图里的<UC：停止(Stop)>用例是<UC：播放(Play)某首音乐>的扩充用例，它是整个对话流程中”可选择性的”部分。

这<<extend>>关系来自于用例内执行活动的过程有分为主要过程(Main Course) 及替代过程(Alternative Course)。例如，<UC：播放(Play)某首音乐>的过程中，如果用户不想听时，会选择按下<Stop>按钮而结束播放该曲音乐。



所以<UC：播放(Play)音乐>描述着 A 这项 Scenario。而<UC：播放(Play)音乐>的用例叙述加上<UC：停止(Stop)>的用例叙述，共同描述 B 这项 Scenario。图中的 A 是个正常的情境(Normal Scenario)，而新附加的用例表达出特殊或替代情境(Alternative Scenario)。

4.2 用例叙述

基于上述的用例模型概念，兹分别撰写其用例叙述，如下：

用例叙述

Use Case ID	UC-001
Use Case 名称：	播放(Play)音乐
目的：	用户听一首默认(Default)的 MP3 音乐
系统名称：	“播放 MP3 音乐” 系统
Client 种类	人
主要程序：	

user action	*system response*
1. 按下< Play>按钮	
	2. 播放默认的 MP3 音乐
3. 按下<Exit>按钮	

	4. 结束此用例
替代程序：	
2a. 播放时，如果 User 按下<Stop>按钮，就结束播放该首音乐	

用例叙述

Use Case ID	UC-002
Use Case 名称：	播放(Play)某首音乐
目的：	用户选取一首 MP3 音乐，然后听该首音乐
系统名称：	“播放 MP3 音乐” 系统
Client 种类	人
主要程序：	
user action	***system response***
1. 选取某首音乐 2. 按下< Play>按钮	
	2. 播放该首 MP3 音乐
3. 按下<Exit>按钮	
	4. 结束此用例
替代程序：	
2a. 播放时，如果 User 按下<Stop>按钮，就结束播放该首音乐	

☆

G07_接口设计之美_通用性接口设计范例

内容：

◇ 第 1 次跌代：实践<UC-001：播放(Play)音乐>

前言：

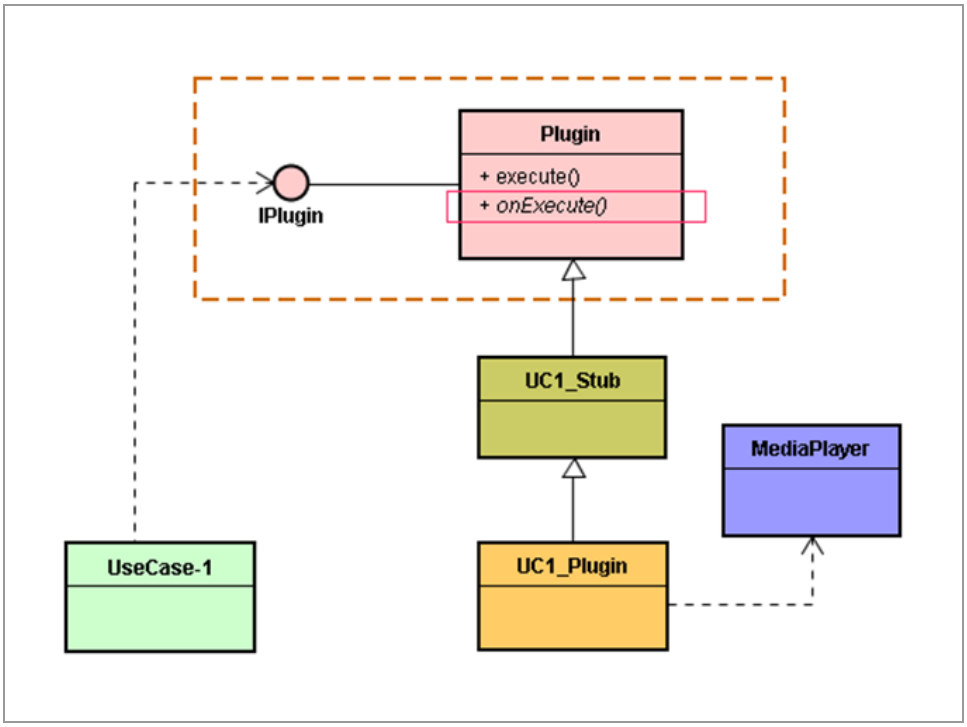
- Use Case 模型偏于需求的表述(Representation)，其主要的表述形式有二：用例叙述(Use Case Description)和用例图(Use Case Diagram)。
- 随着敏捷迭代过程，Use Case 的数量会逐渐增加，其表述内容也会逐渐细腻而完整。此外，Use Case 所落实出来的代码也愈来愈多、愈来愈细腻。
- 因之，我们可藉由 EIT 造形来定义出统一而共通性的<I>呢?基于这项通用性<I>，就能将敏捷迭代的产出代碼，持续整合起来。

5. 「播放 MP3 音乐」范例的实现代码

5.1 第 1 次跌代：实践<UC-001：播放(Play)音乐>

架构设计

此范例展示如何建立一个框架(Framework)：包含一个 IPlugin 接口和一个 Plugin 基类。由它来提供统一的通信接口，来整合各敏捷跌代过程中所开发的各用例代码。例如，在敏捷的第 1 回跌代时，可先产出 UC-001 部分，其架构图为：

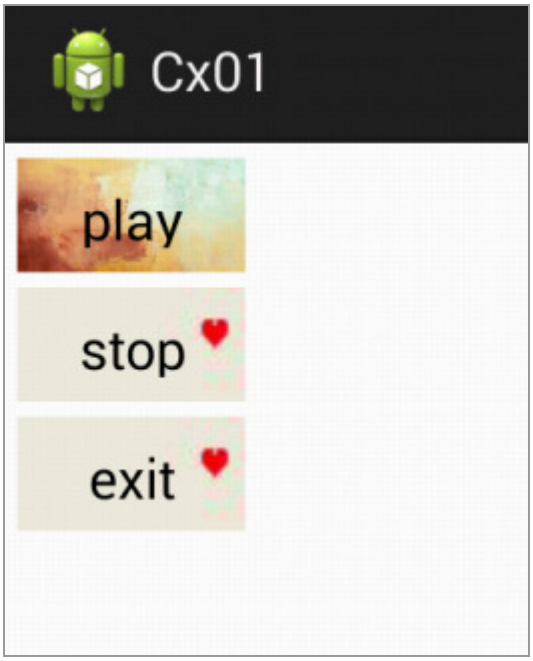


前面已经提到过，这项架构设计的优点是：

- 各组用例之间都是独立的，可做为不同跌代的工作产出目标。
- 虽然各组用例共享了通用性接口，但都能用有自己的特殊性接口。
- 透过 EIT 造形的转换接口之后，Client 部分与 Server 部分是独立的，创造了两端各自的变动自由度。例如，UseCase-1 的 UI 画面的改变，不会受制于 MediaPlayer 的接口(函数)；此外，MediaPlayer 的接口改变了，只需改变 UC1_Plugin，而不会波及各 UseCase-1 或 UseCase-2 等。
- 掌握框架者，就能掌控一切。例如，可以在 Plugin 基类的幕后，可设计一只看门狗(Watch Dog)来监控各 Use Case 的通信。则框架主人透过统一接口的制定，来达到全面监控的目标。

UI 画面设计

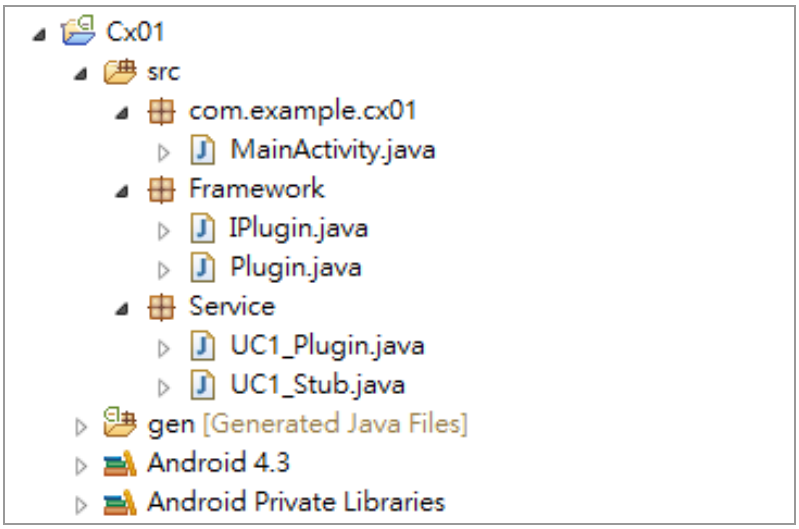
由于 EIT 造形的转换接口，Use Case 的 Client 部分与 Server 部分是独立的，创造了两端各自的变动和成长自由度。首先，敏捷地设计了简单的 UI 画面如下：



按下<play>按钮，就会播放预设(Default)的 MP3 音乐，按下<stop>就停只播放，按下<exit>就结束此程序了。

Android 开发项目

在 Android 平台上来实践上述范例的架构，兹建立 Android 开发项目(Project)如下：



代码范例：Framework 部分

// IPlugin.java 接口

```
package Framework;

public interface IPlugin{
        boolean execute(int code, int arg);
}
```

// Plugin.java 类

```
package Framework;

public abstract class Plugin implements IPlugin{

        @Override public boolean execute(int code, int arg) {
                return onExecute(code, arg);
        }
        protected abstract boolean onExecute(int code, int arg);
}
```

代码范例：App 部分

// UC1_Stub.java 类

```
package Service;
import Framework.Plugin;
public abstract class UC1_Stub extends Plugin {

        @Override protected boolean onExecute(int code, int arg) {
                if(code == 0)    play(arg);      //播放目前音樂
                else if(code == 1)    stop();    //停止播放
                return false;
        }
        public abstract void play(int arg);
        public abstract void stop();
}
```

这曹操类的主要任务是：

- 实现了天子的接口，也就是 onExecute()抽象函数。
- 提供了自己的新接口，也就是 play()和 stop()两个函数。

// UC1_Plugin.java 类

```
package Service;
import android.content.Context;
import android.media.MediaPlayer;
import android.util.Log;

public class UC1_Plugin extends UC1_Stub{
        private MediaPlayer mPlayer = null;
```

```
    private Context ctx;

    public UC1_Plugin(Context context){   ctx = context;     }
    @Override public void play(int arg) {
        if(mPlayer != null) return;
        mPlayer = MediaPlayer.create(ctx, arg);
        try {  mPlayer.start();
        } catch (Exception e) { Log.e("StartPlay", "error: " + e.getMessage(), e);   }
    }
    @Override public void stop() {
        if (mPlayer != null) {
            mPlayer.stop();       mPlayer.release();       mPlayer = null;
        }
}}
```

这个 App 类(及子民类)只需要实现曹操的接口就行了，也就是实现 play()和 stop()函数。

// MainActivity.java 类

```
package com.example.cx01;
import Framework.IPlugin;
import Service.UC1_Plugin;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.LinearLayout;

public class MainActivity extends Activity implements OnClickListener {
     private Button btn, btn2, btn3;
    private IPlugin ip;

    public void onCreate(Bundle icicle) {
                    super.onCreate(icicle);
                    LinearLayout layout = new LinearLayout(this);
                    layout.setOrientation(LinearLayout.VERTICAL);
                    btn = new Button(this);          btn.setId(101);
                    btn.setBackgroundResource(R.drawable.paint);
                    btn.setText("play");             btn.setOnClickListener(this);
                    LinearLayout.LayoutParams param =
                        new LinearLayout.LayoutParams(150, 75);
                    param.topMargin = 10;          param.leftMargin = 10;
                    layout.addView(btn, param);
                    btn2 = new Button(this);       btn2.setId(102);
                    btn2.setBackgroundResource(R.drawable.heart);
                    btn2.setText("stop");          btn2.setOnClickListener(this);
                    layout.addView(btn2, param);
                    btn3 = new Button(this);       btn3.setId(103);
                    btn3.setBackgroundResource(R.drawable.heart);
                    btn3.setText("exit");           btn3.setOnClickListener(this);
                    layout.addView(btn3, param);
                    setContentView(layout);
                    //------------------------------
```

```
                    ip = new UC1_Plugin(this);
                    }
            public void onClick(View v) {
                  switch(v.getId()){
                        case 101:    play(); break;
                        case 102:    stop(); break;
                        case 103:    stop();    finish();    break;
                   }
             }
            public void play(){    ip.execute(0, R.raw.jp_song);          }
            public void stop(){    ip.execute(1, -1);    }
}
```

G07_接口设计之美_通用性接口设计范例

内容：

◇ 第 2 次跌代：实践<UC-002：播放(Play)某首音乐>

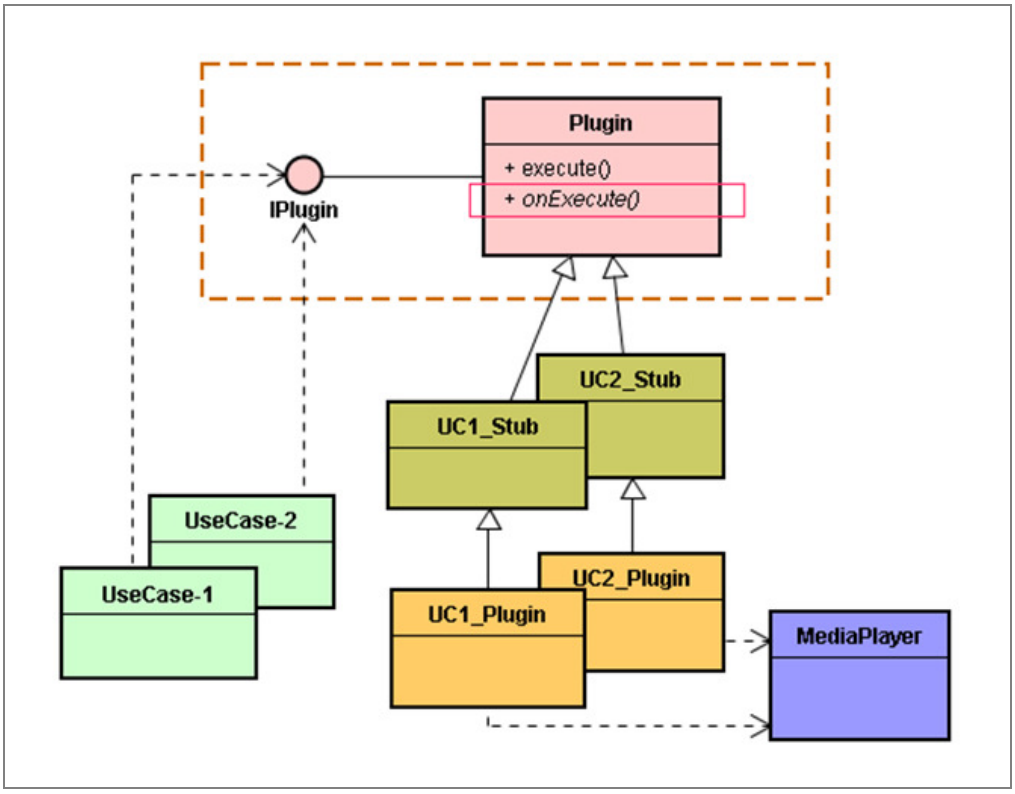
前言：

- Use Case 模型偏于需求的表述(Representation)，其主要的表述形式有二：用例叙述(Use Case Description)和用例图(Use Case Diagram)。
- 随着敏捷迭代过程，Use Case 的数量会逐渐增加，其表述内容也会逐渐细腻而完整。此外，Use Case 所落实出来的代码也愈来愈多、愈来愈细腻。
- 因之，我们可藉由 EIT 造形来定义出统一而共通性的<I>呢?基于这项通用性<I>，就能将敏捷迭代的产出代碼，持续整合起来。

5.2 第 2 次跌代：实践<UC-002：播放(Play)某首音乐>

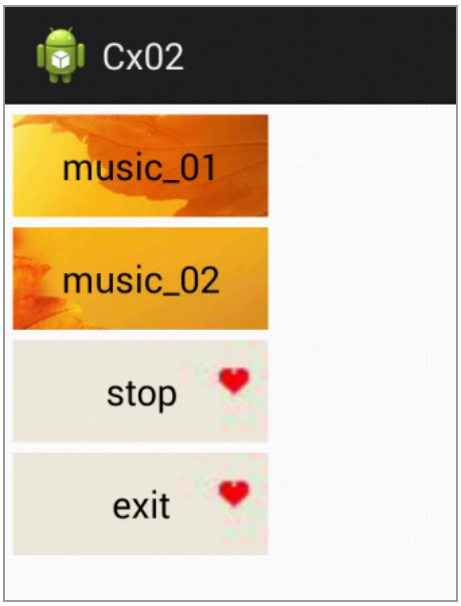
架构设计

在刚才的第 1 次敏捷跌代里，已經建立了一个框架，提供统一的接口来整合各回合跌代所开发产出的 UseCase 实现代码。现在，基于此项架构，可以继续配合敏捷跌代而开发更多的 UseCase，它们都透过通用性的 IPlugin 接口的exectute()函数和 Plugin 类的 onExecute()函数来做为通信渠道。于是，实现了各用例的通用性接口设计了，也实现了敏捷用例(Agile-Use Case)的目标。例如，可以增添新的 UseCase-2，如下图：



UI 画面设计

由于 EIT 造形的转换接口，Use Case 的 Client 部分与 Server 部分是独立的，创造了两端各自的变动和成长自由度。首先，敏捷地设计了简单的 UI 如下：

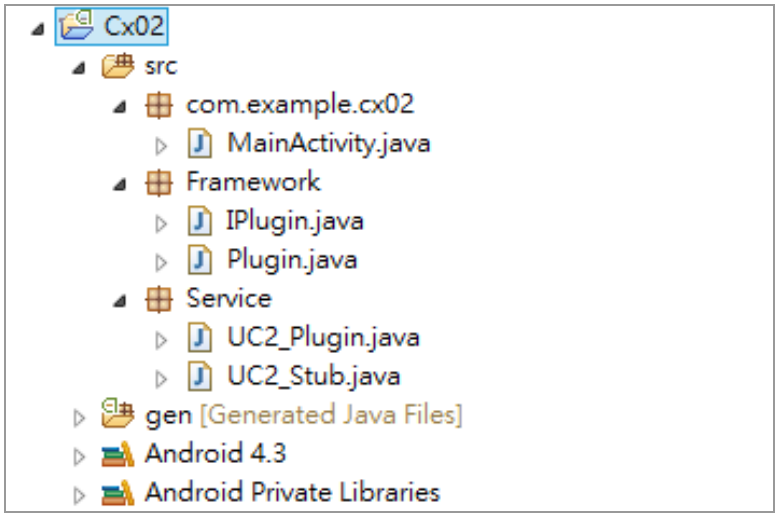


按下<music-01>按钮，就选择第 01 首 MP3 音乐，并开时播放。播放时，如果按下<stop>就停只播放。按下<music-02>按钮，就选择第 02 首

MP3 音乐，并开时播放，此时如果按下<stop>就停只播放。若按下<exit>就结束此程序了。

Android 开发项目

在 Android 平台上来实践上述范例的架构，兹建立 Android 开发项目(Project)如下：



代码范例：Framework 部分

// IPlugin.java 接口

```
package Framework;

public interface IPlugin{
        boolean execute(int code, int arg);
}
```

// Plugin.java 类

```
package Framework;
public abstract class Plugin implements IPlugin{

        @Override public boolean execute(int code, int arg) {
                return onExecute(code, arg);
        }
        protected abstract boolean onExecute(int code, int arg);
}
```

代码范例：App 部分

// UC2_Stub.java 类

```
package Service;
```

```
import Framework.Plugin;

public abstract class UC2_Stub extends Plugin {
       private int curr_music;
        @Override
        protected boolean onExecute(int code, int arg) {
                if(code == 0)                   play(curr_music);     //播放目前音樂
                else if(code == 1)              stop();               //停止播放音樂
                else if(code == 2)              curr_music = arg;     //設定目前音樂
                return false;
        }
        public abstract void play(int arg);
        public abstract void stop();
}
```

// UC2_Plugin.java 类

```
package Service;
import android.content.Context;
import android.media.MediaPlayer;
import android.util.Log;

public class UC2_Plugin extends UC2_Stub{
        private MediaPlayer mPlayer = null;
        private Context ctx;

        public UC2_Plugin(Context context){    ctx = context;      }
        @Override public void play(int arg) {
                if(mPlayer != null) return;
                mPlayer = MediaPlayer.create(ctx, arg);
                try {
                        mPlayer.start();
                } catch (Exception e) { Log.e("StartPlay", "error: " + e.getMessage(), e);   }
        }
        @Override public void stop() {
                if (mPlayer != null) {
                        mPlayer.stop();
                        mPlayer.release();
                        mPlayer = null;
                }
  }}
```

// MainActivity.java 类

```
package com.example.cx02;
import Framework.IPlugin;
import Service.UC2_Plugin;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.LinearLayout;

public class MainActivity extends Activity implements OnClickListener {
        private Button btn, btn2, btn3, btn4;
```

```
    private IPlugin ip;

    public void onCreate(Bundle icicle) {
                    super.onCreate(icicle);
                    LinearLayout layout = new LinearLayout(this);
                    layout.setOrientation(LinearLayout.VERTICAL);
                    btn = new Button(this);       btn.setId(101);
                    btn.setBackgroundResource(R.drawable.leaf);
                    btn.setText("music_01");   btn.setOnClickListener(this);
                    LinearLayout.LayoutParams param =
                          new LinearLayout.LayoutParams(250, 100);
                    param.topMargin = 10;       param.leftMargin = 10;
                    layout.addView(btn, param);
                    btn2 = new Button(this);     btn2.setId(102);
                    btn2.setBackgroundResource(R.drawable.leaf2);
                    btn2.setText("music_02");   btn2.setOnClickListener(this);
                    layout.addView(btn2, param);
                    btn3 = new Button(this);     btn3.setId(103);
                    btn3.setBackgroundResource(R.drawable.heart);
                    btn3.setText("stop");          btn3.setOnClickListener(this);
                    layout.addView(btn3, param);
                    btn4 = new Button(this);          btn4.setId(104);
                    btn4.setBackgroundResource(R.drawable.heart);
                    btn4.setText("exit");               btn4.setOnClickListener(this);
                    layout.addView(btn4, param);
                    setContentView(layout);
                    //-------------------------------
                    ip = new UC2_Plugin(this);
                    }
          public void onClick(View v) {
                  switch(v.getId()){
                       case 101: set_music(R.raw.jp_song); break;
                       case 102: set_music(R.raw.test_cbr); break;
                       case 103: stop(); break;
                       case 104: stop(); finish(); break;
                  }
          }
          public void play(){     ip.execute(0, -1);     }
          public void stop(){     ip.execute(1, -1);   }
          public void set_music(int song_id)
                              { stop();   ip.execute(2, song_id);   play(); }
    }
}
```

6. 结语

在 UseCase-1 里，其与 Server 的特殊性接口，包括有两项功能：

```
public void play()
      {   ip.execute(0, R.raw.jp_song);      }
public void stop()
      {   ip.execute(1, -1);   }
```

其将 play()函数编号为 0，并将 stop()编号为 1；然后调用通用性的 IPlugin 接

口的 execute()函数。

而在 UseCase-2 里，其与 Server 的特殊性接口，包括有三功能：

```
public void play()
      {    ip.execute(0, -1);      }
public void stop()
      {    ip.execute(1, -1);    }
public void set_music(int song_id){
            stop(); ip.execute(2, song_id); play();
      }
```

其将 play()函数编号为 0，将 stop()编号为 1，并将 set_music()编号为 2；然后调用通用性的 IPlugin 接口的 execute()函数。

所以 UseCase-1、UseCase-2 和 UseCase-n 都共享通用性接口，又能提供自己特殊化接口。基於其通用性接口，來整合眾多的用例；同時，基於其特殊化接口，來滿足各用例的特殊功能。

~ End ~