G03_接口设计之美_五子棋框架设计范例

内容：

◇ 框架(Framework)：当今主流平台的幕后架构

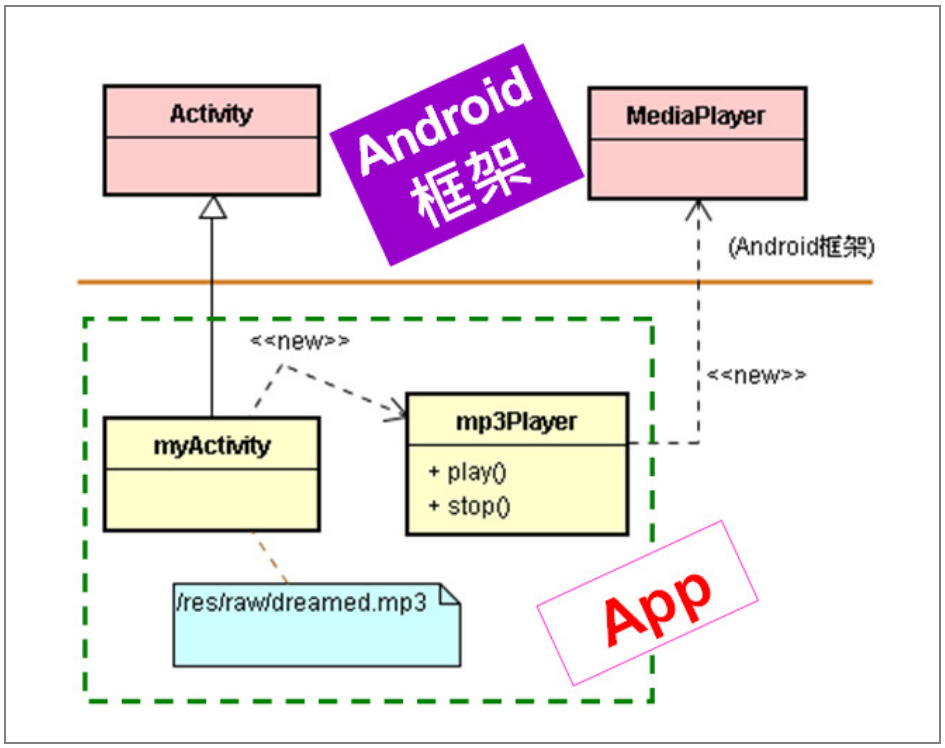
◇ 从 EIT 造形到框架(Framework)

前言：

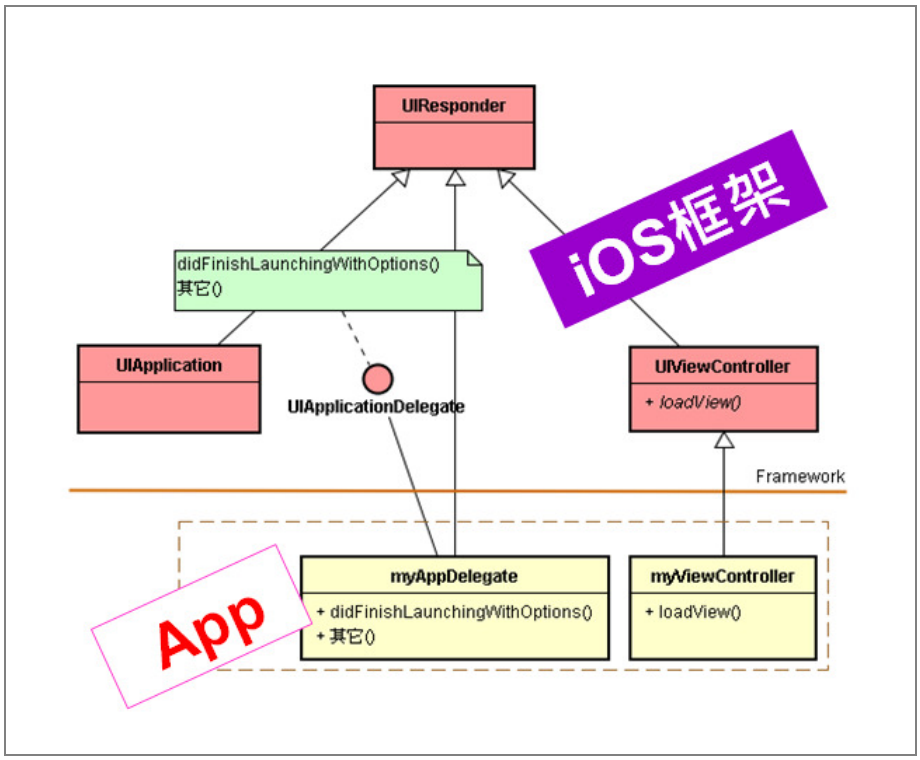
- 框架(Framework)是一个平台(如 Android 平台、iOS 平台等)，它提供 API(即接口)来与数十万支 App 对接。于是，<接口设计之美>对于框架开发是非常关键的了。
- 基于 EIT 造形去寻找接口、设计接口、表达接口，就能清晰定义框架的 API 了。
- 框架是当今主流平台的幕后架构，他强力支撑当今 Apple 和 Google 应用商店的运作。

1. 框架(Framework)：当今主流平台的幕后架构

框架(Framework)是一个平台(如 Android 平台、iOS 平台等)，它提供 API(即接口)来与数十万支 App 对接。之前，我們說過了，從架構設計應該迅速落实为(可执行的)代码。其中，可执行的代码有两种：框架和 App。由强龙开发框架代码；而由地头蛇开发 App 代码；这称为<强龙/地头蛇>分工模式。这就是当今 Apple 和 Google 应用商店的分工模式。例如：



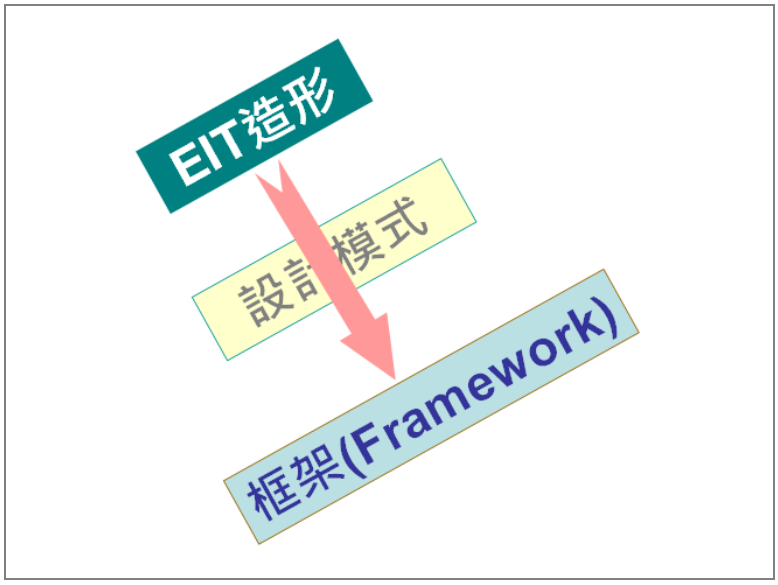
框架(Framework)是一个平台(如 Android 平台、iOS 平台等)，它提供 API(即接口)来与数十万支 App 对接。于是，EIT 造形与框架开发&设计就息息相关了。



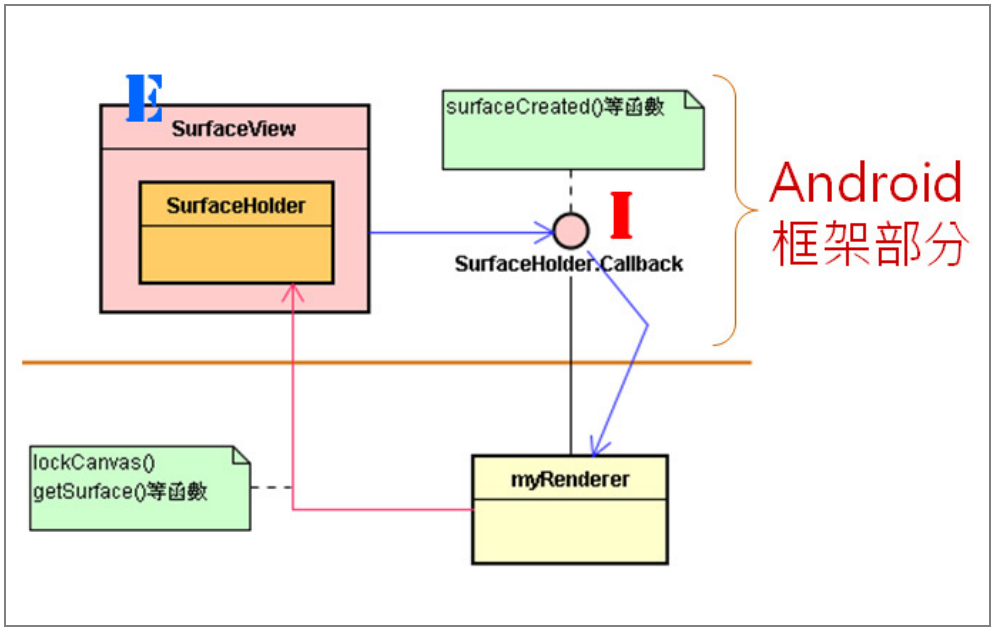
EIT 造形的焦点在于<I>，这个<I>恰好是框架与 App 的衔接点，也成为框架开发团队与 App 开发团队的分工界线。因此，架构师可以藉由 EIT 造形来清晰地表述框架与 App 的接口。然后，由框架开发团队(强龙)撰写<E>类代码，以及与<E>相关的其它类的代码，就成为软件框架了。并由 App 开发团队(地头蛇)撰写<T>类代码，以及与<T>相关的其它类的代码，就成为 App 软件了。

2. 从 EIT 造形到框架(Framework)

通常，一个框架含有许多接口<I>，亦即需要一群 EIT 造形来清晰表述这些<I>。其意味着，由一群 EIT 造形组合起来，成为框架的核心部分：与 App 的接口。



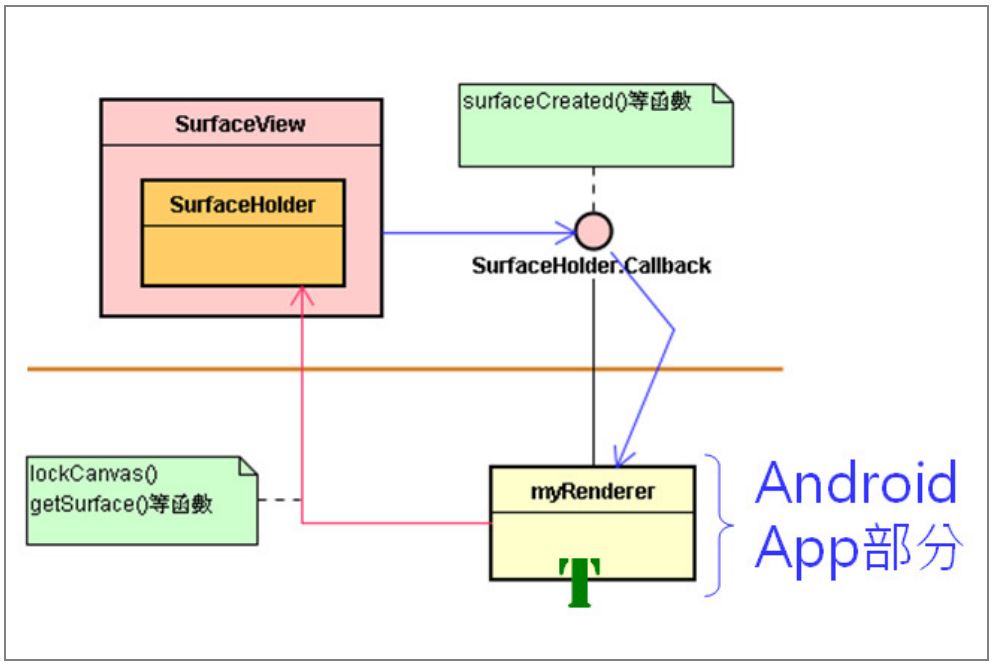
在特定领域(Domain)里，将 EIT 造形的<E&I>部份有意义地组合起来，就成为框架(Framework)的核心部分了。其中，包括将<E&I>组合起来成为框架本身；同时将<T>组合起来成为框架的应用(Application)软件。例如，Android 框架就包含了 SurfaceView 类，及其 SurfaceHolder.Callback 接口。



于此图里，SurfaceHolder.Callback 扮演<I>的角色，SurfaceView(含 SurfaceHolder)扮演<E>的角色，而 myRenderer 扮演<T>的角色。SurfaceView 引擎透过 Callback 接口，呼叫了 myRenderer 的 surfaceCreated()等函数。

虽然框架并没有包含<T>类，但是架构师必须思考整个 EIT 造形，将<T>

考虑进来，才能完整而明确地表述接口<I>。由于 EIT 造形能明确地将<I>表述出来，并清晰地传达给 App 开发者。于是，开发者就能开发<T>来实现<I>，并精确地搭配到<E>，就能与框架一起编译，成为可执行的 App 软件(如 Android 的 APK)了。




```
// myRenderer.java
// …….
class myRenderer implements SurfaceHolder.Callback {
        private SurfaceHolder mHolder;
        private DrawThread mThread;

        public void surfaceCreated(SurfaceHolder holder) {
                mHolder = holder;
                mThread = new DrawThread();    mThread.start();
           }
        public void surfaceDestroyed(SurfaceHolder holder) {
                mThread.finish();          mThread = null;
           }
        public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) { }
        // -----------------------------------------------------------------------
        class DrawThread extends Thread {
            int degree = 36;
            boolean mFinished = false;
            DrawThread() {    super();    }
           @Override public void run() {
                    Bitmap bmp = BitmapFactory.decodeResource(getResources(),
                                        R.drawable.x_xxx);
                    Matrix matrix;
                    degree = 0;
```

```
                while(!mFinished){
                        Paint paint = new Paint();
                        paint.setColor(Color.CYAN);
                        Canvas cavans = mHolder.lockCanvas();
                        cavans.drawCircle(80, 80, 45, paint);
                        //------ rotate ----------------------------
                        matrix = new Matrix();        matrix.postScale(1.5f, 1.5f);
                matrix.postRotate(degree);
                Bitmap newBmp = Bitmap.createBitmap( bmp, 0, 0,
                                bmp.getWidth(), bmp.getHeight(), matrix, true);
                cavans.drawBitmap(newBmp, 50, 50, paint);
                mHolder.unlockCanvasAndPost(cavans);
                degree += 15;
                try { Thread.sleep(100);
                } catch (Exception e) {}
            }}
        void finish() { mFinished = true;    }
    }}
```

```
// ac01.java
// ……..
public class ac01 extends Activity {
      private SurfaceView sv = null;
      @Override protected void onCreate(Bundle icicle) {
            super.onCreate(icicle);
            sv = new SurfaceView(this);
            myRenderer mr = new myRenderer();
            sv.getHolder().addCallback(mr);
             LinearLayout layout = new LinearLayout(this);
            layout.setOrientation(LinearLayout.VERTICAL);
            LinearLayout.LayoutParams param =
                  new LinearLayout.LayoutParams(200, 150);
            param.topMargin = 5;
            layout.addView(sv, param);
            setContentView(layout);
    }}
```

首先，SurfaceView 对象向 Android 的 WindowManagerService(和 SurfaceFlinger)系统服务取的一个 Surface，将它包装于 SurfaceView 里的 SurfaceHolder 对象里。

然后，透过 Callback 接口来呼叫 myRenderer 子类里的 surfaceCreated()函数，此时将该 SurfaceHolder 对象(的指针或参考)传递给 myRenderer 的对象。myRenderer 子类的对象才依循 SurfaceHolder 的指标而呼叫到 SurfaceHolder 的 lockCanvas()等函数。

以上說明了，Android 框架里含有一個 EIT 造形，就是：<SurfaceView, SurfaceHolder.Callback, myRenderer>造形的<E&I>部分。☆

G03_接口设计之美_五子棋框架设计范例

内容：

1. 框架(Framework)：当今主流平台的幕后架构
2. 从 EIT 造形到框架
3. 框架设计范例：以『五子棋』为例
   - 3.1 阶段一：从传统类(Class)造形设计出发
   - 3.2 阶段二：继续运用 EIT 造形设计

◇ 框架设计范例：以『五子棋』为例

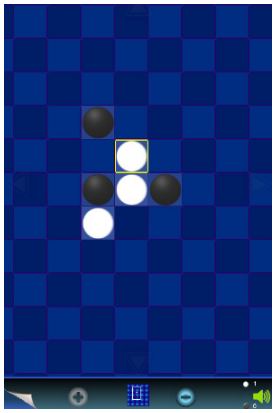
前言：

- 应用框架(Application Framework)是一个平台(如 Android 平台、iOS 平台等)，它提供 API(即接口)来与数十万支 App 对接。于是，EIT 造形与框架开发&设计就息息相关了。
- 于此，将以五子棋的架构设计为例，进行两阶段的分析与设计。
- 第 1 阶段：先依循传统的 OOAD(Object-Oriented Analysis & Design)来进行领域知识分析&设计。OOAD 就是基于类(Class)造形的分析&设计方法。这是目前业界的主流方法，其最主要的缺点是：没有清晰而明确地表述出接口。
- 第 2 阶段：延续上阶段 OOAD 的分析&设计结果，基于 EIT 造形去厘清接口<I>，因而清晰定义了框架的 API。
- 由于 EIT 造形是由类造形扩充而来，所以上述两阶段的衔接是很流畅的。

3. 框架设计范例：以『五子棋』為例

3.1 階段一：从传统类(Class)造形设计出发

『五子棋』 的 OOAD 分析与设计

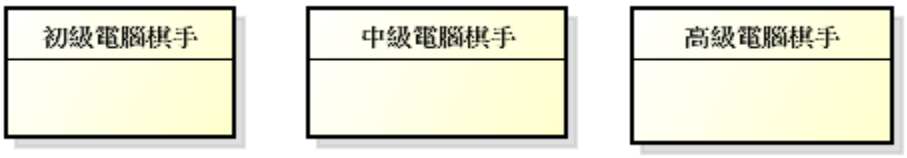


(图片来源：互动百科)

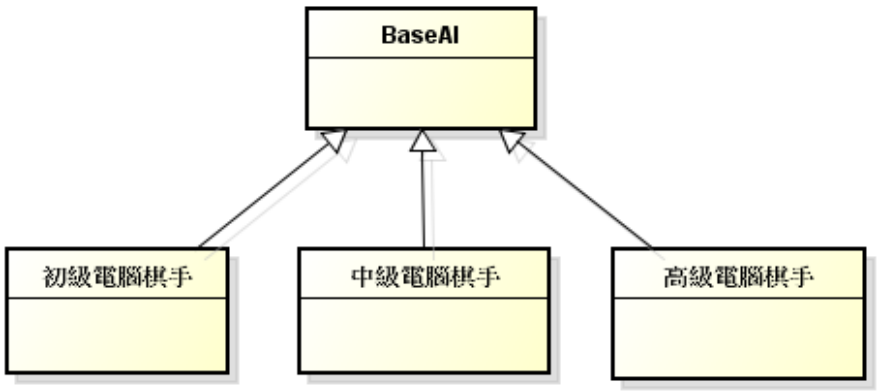
从传统类(Class)造形设计出发，针对『五子棋』进行传统的 OOAD 分析与设计(Object-Oriented Analysis & Design)。其分析步骤为：

Step-1: 找到主角，就是：棋手

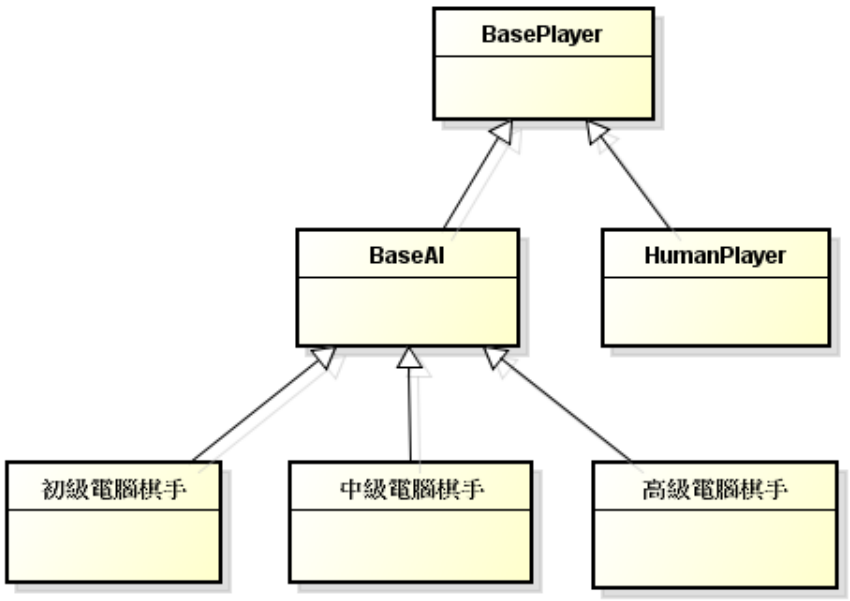
首先，寻找『五子棋』的核心概念，成为类造形的内涵。例如：五子棋游戏的主角是棋手(玩家)，棋手有两种：电脑和人；其中，电脑棋手又分为数个不同棋力等级，例如：



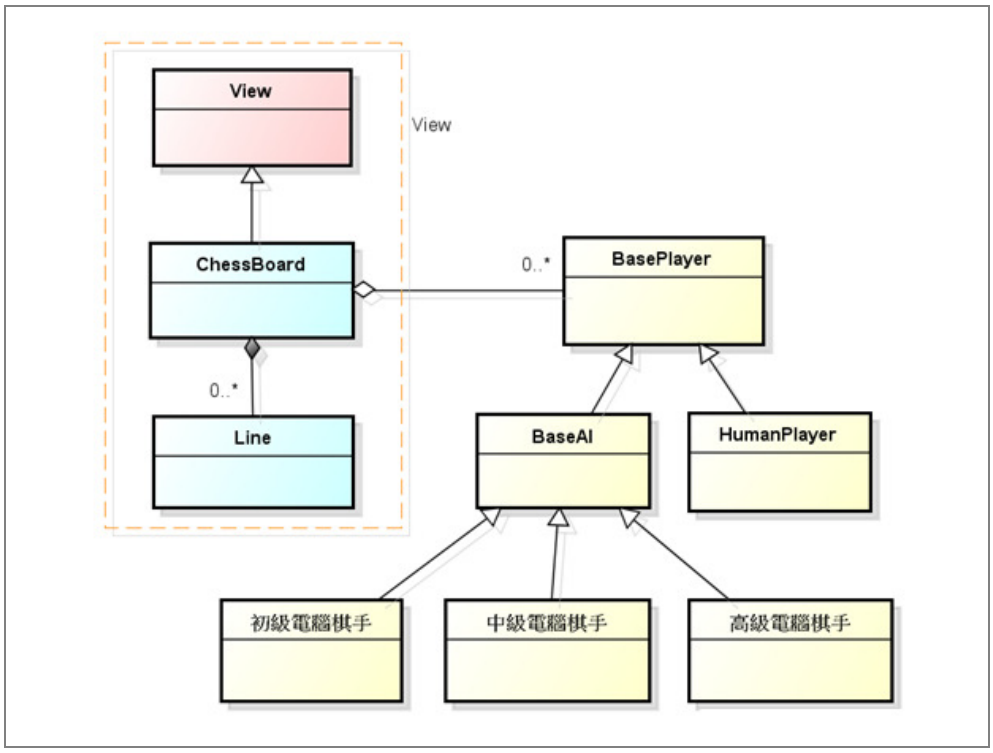
Step-2: 抽象出抽象类别(Super-class)，就是 BaseAI 類：



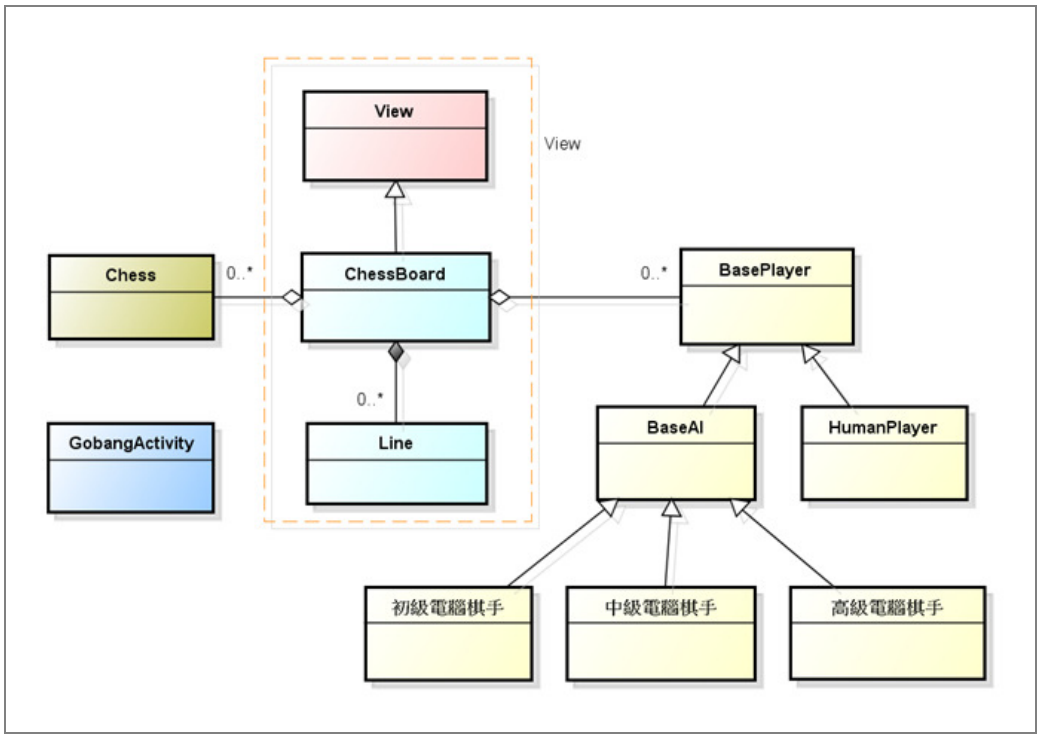
Step-3: 再增添一种棋手，HumanPlayer (人)，而且再度进行抽象，得到：



Step-4: 再联想到人之外的物——棋盘(Chess Board)，它必须呈现于 UI 画面上，所以设计成为 View 的子类别，得到：



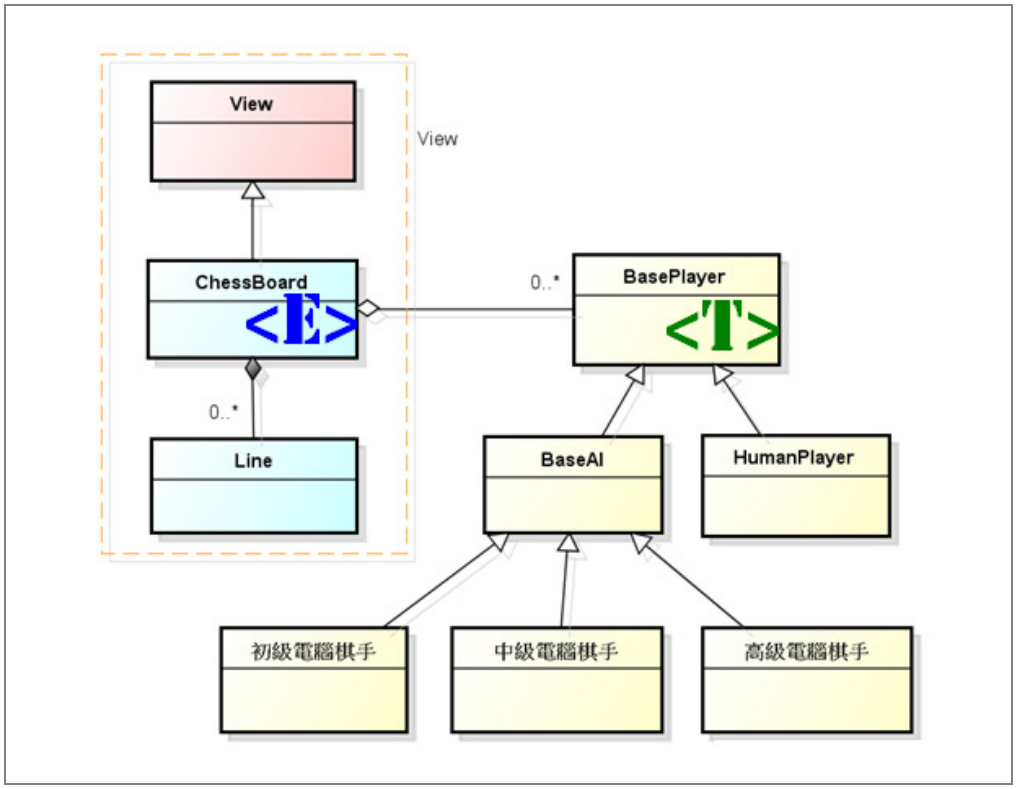
Step-5: 再从棋盘联想到相关的概念：棋(Chess)；以及用来控制 UI 显示的 GobangActivity 类别。如下图。



Step-6: 还可以继续联想下去，就更加完整了。

3.2 阶段二：继续运用 EIT 造形设计

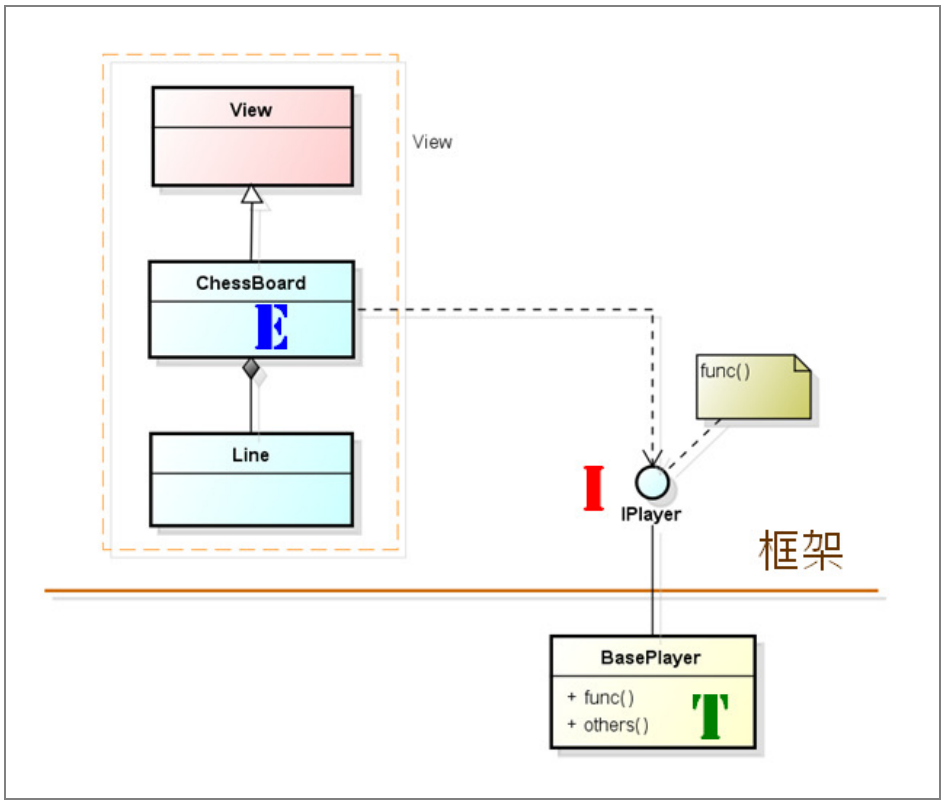
从上图里，可以看出来，传统基于类造形的分析与设计，只凸显了类(Class)和关系(Relationship)，而将接口(Interface)隐藏于类或关系里，此时 EIT 造形就派上用场了。例如，上图的”棋盘(Chess Board)”与”棋手(Player)”之间是 1:N 的组合关系，就隐含了一个重要接口：可让用户选择多位棋手。于是，藉由 EIT 造形的<I>来表述这个接口，而棋盘和棋手就是它的配角：棋盘扮演<E>角色，而棋手扮演<T>角色，如下图：



传统上，将<I>隐藏起来，常常带来许多缺点，例如：

- 架构师知道接口的存在，但没有途径去清晰地表述出来。
- 由于没有明确传达给开发者，徒增开发者的负担，也提高了失误的可能性。
- 由于没有凸显接口，无法协助项目经理(PM)掌握最佳的团队分工界线，例如框架开发与 App 开发的分际。
- 等等。

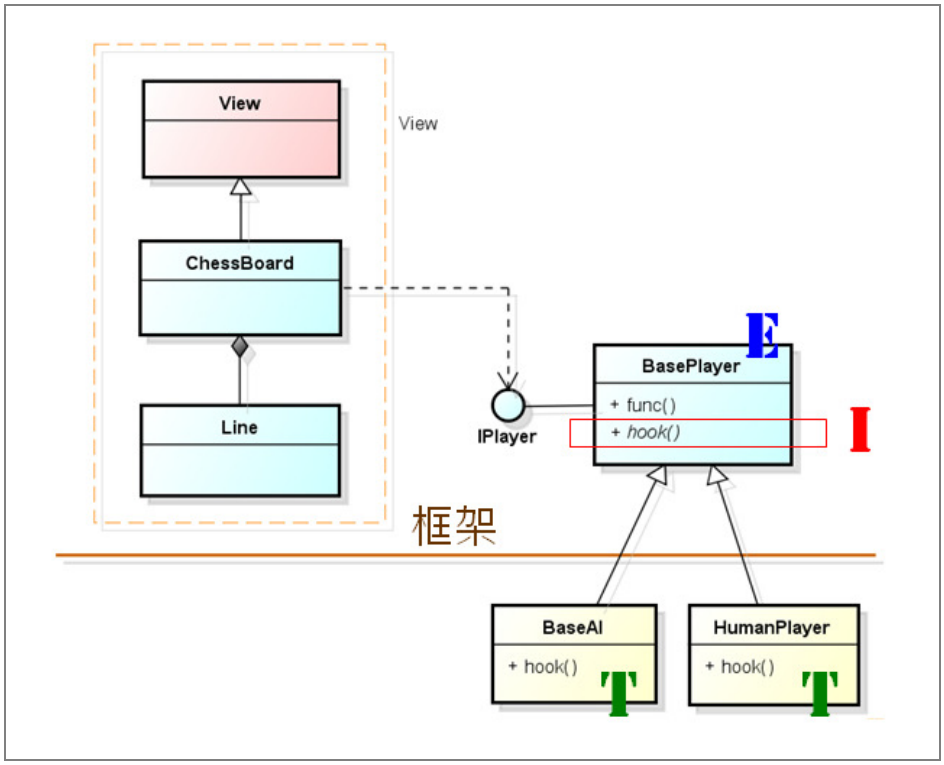
于是，可以藉 EIT 造形来凸显<I>，如下图：



这个 IPlayer 就成为框架与 App 的分工界线了。在买主还没来、或用户还没出现之前，框架团队就能先定义 IPlayer 接口，然后进行开发 Chess Board 和 Line 等类的代码，成为框架代码了。而等到买主来了、或用户出现之后，App 团队才开始动手设计棋手(即<T>的类体系)，成为 App 软件代码了。

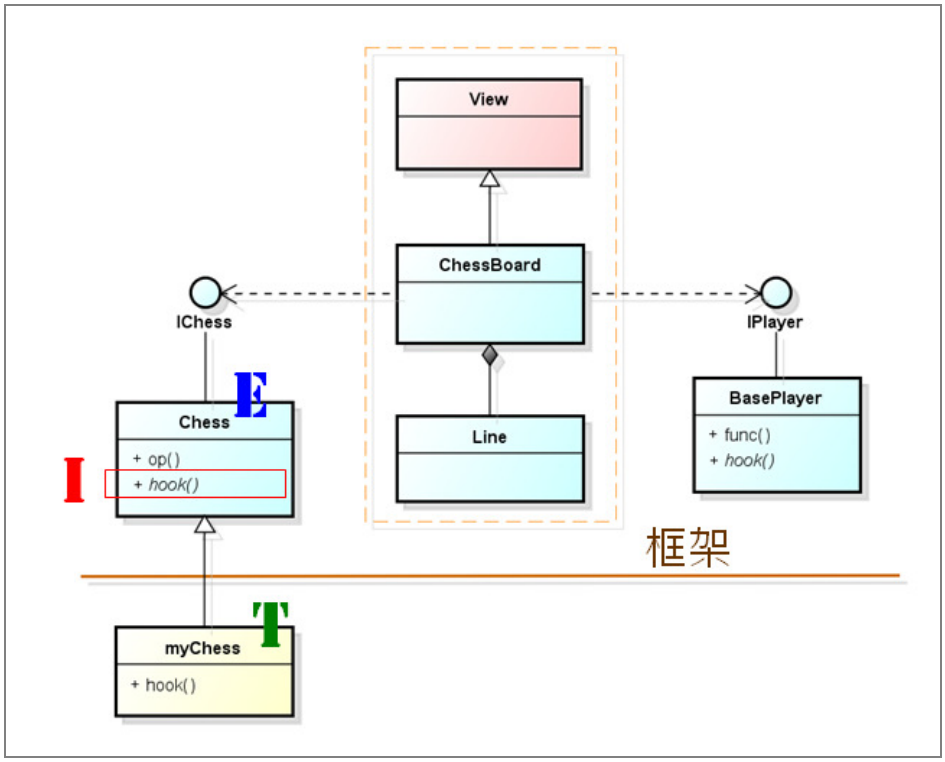
由于接口(如上图的 IPlayer)只能含有抽象函数，不能含有具象函数，所以这些抽象函数的实现代码都必须写在 App 的<T>类里。这会增加 App 开发团队的负担，延迟 App 开发交付的效率。

框架开发团队为了提供更多的默认行为(Default Behavior)来让开发团队可加以复用(Reuse)，就会设计抽象类(Abstract)来具象函数，来实现这些默认行为。例如下图：



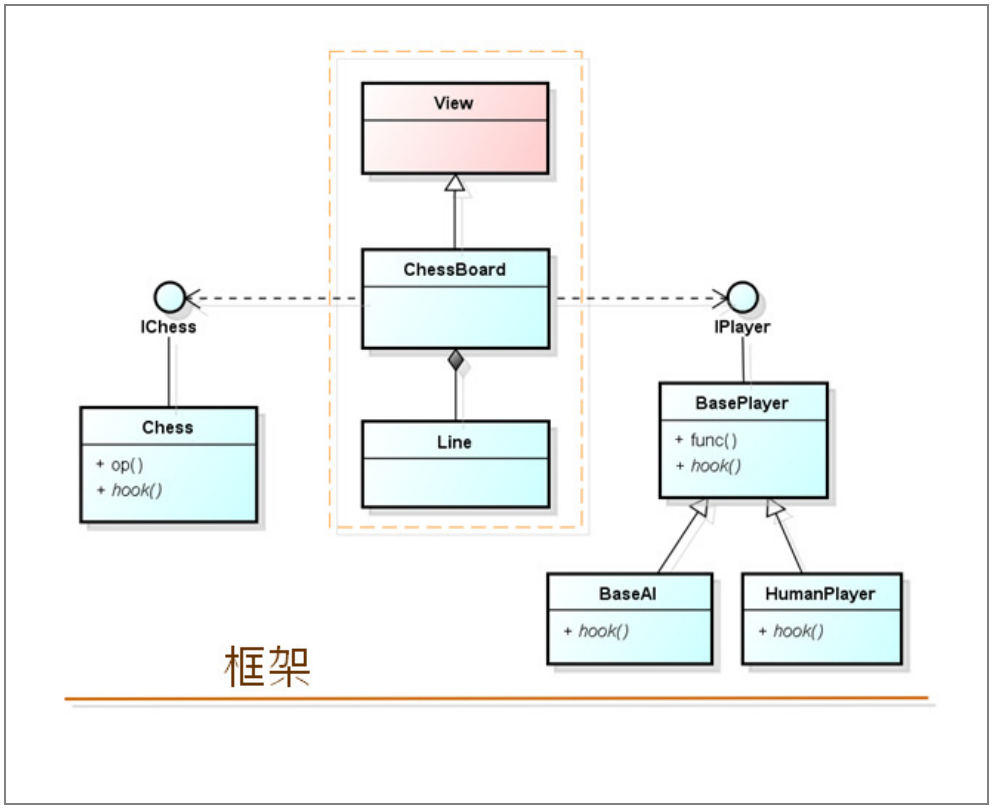
这个 BasePlayer 是一个抽样类，内含 func()等具象函数，可撰写代码来提供默认行为，让各<T>类来复用，能减轻 App 开发负担，提高开发交付效率。此时，BasePlayer 扮演两个角色：一方面扮演<Chess Board, IPlayer, BasePlayer>造形的<T>；另一方面又扮演一个新 EIT 造形的<E>，提供新的<I>，就是上图里的 hook()抽象函数，来让 App 的<T>类来撰写其实现代码。于是，这个新的<I>就封装了原来的 IPlayer 接口，变成框架与 App 的新接口，也是新的分工界线了。这个新界线的优点是：简清了 App 开发者的负担，因而能吸引更多 App 开发者来使用此软件框架了。

依样画葫芦，再依循 EIT 造形，继续设计一个 IChess 接口，以及设计一个 Chess 抽象类，来提供框架与 App 之间的<I>，如下图：



此时，Chess 也扮演两个角色：一方面扮演<Chess Board, IChess, Chess>造形的<T>；另一方面又扮演一个新 EIT 造形的<E>，提供新的<I>，就是上图 Chess 类里的 hook()抽象函数，来让 App 的<T>类(如上图的 myChess)来撰写其实现代码。

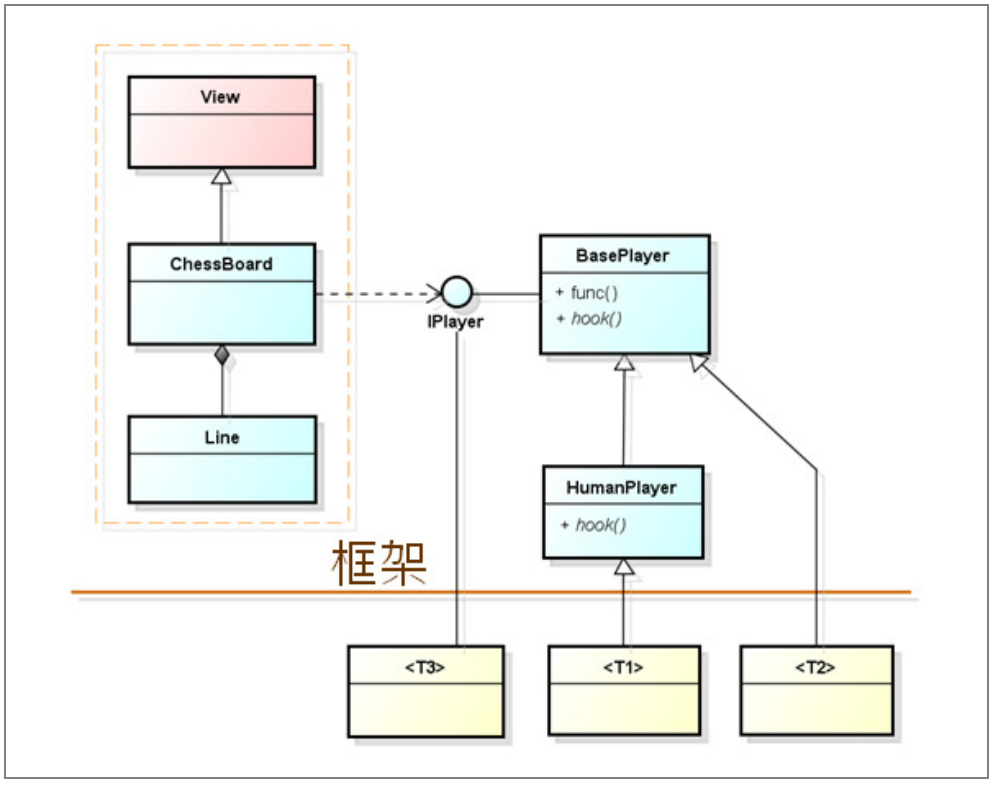
为了进一步减轻 App 开发者的负担，藉之吸引更多 App 开发者来使用框架，通常框架开发团队会持续增添更多的抽象类，来提供更贴心的默认行为。例如下图，架构师将 BaseAI 和 HumainPlayer 两个抽象类提升到框架里。



计算机棋手共享的默认行为，可以写在 BaseAI 抽象类里；而人员棋手的共享默认行为则写在 HumainPlayer 抽象类里。至于所有棋手都共享的默认行为则提升到最高层的 BasePlayer 抽象类里。如此，持续丰富框架的内涵，进一步减轻 App 开发者的负担，则框架的价值就愈来愈高了。

此外，也提供给 App 开发者更多的弹性选择空间；例如在上图里，App 开发者撰写<T>类时，就有多种选择了：

- 选择继承 BaseAI 或继承 HumainPlayer 抽象类：可复用(Reuse)这两类里的具象函数。如下图的<T1>类。
- 如果不适合继承上述两个类，可选择继承 BasePlayer 抽象类：可复用这个类里的具象函数。如下图的<T2>类。
- 如果不适合继承上述各个类，可选择直接实现 IPlayer 接口：必须自己实现 IPlayer 里所定义的各函数。如下图的<T3>类。


View
ChessBoard
Line
IPlayer
BasePlayer
+ func()
+ hook()
HumanPlayer
+ hook()
框架
<T3>
<T1>
<T2>

~ End ~