G01_接口设计之美_代码造形的角色

1. 代码造形的历史：从函数、类到 EIT
2. 什么是代码造形？
3. 代码造形的用处
4. 大家熟悉的 2 种代码造形：函数和类
5. 介绍新的 EIT 代码造形
    - 5.1 接口是主角：EIT 呈现接口之美
    - 5.2 EIT 造形的重复组合
    - 5.3 为什么需要要有两个配角呢?

1. 代码造形的历史：从函数、类到 EIT

1970 年代的代表性语言就是 C 语言，主要的造形(Form)是”函数”(Function)；其意味着，世界上所有的软件都能以简单的函数形式表述出来，并基于简单规则而组合出复杂的系统。

1980 年代的代表性语言是 C++，当时人们找到了一个较大的造形，叫做“类”(Class)，因而引导整个软件产业走入面向对象(Object-Oriented)的技术潮流中。大家当时都相信所有的软件都可以用类来表示。其意味着，世界上所有的软件都能以简单的类形式表述出来，并基于简单规则而组合出复杂的系统。

到了 1995 年，人们又想找到更大的造形，当时找到的称为”设计模式”(Design Pattern)。只是，请留意，设计模式不是造形，它是 23 种设计模式，设计模式是有内涵的，但而形不能有 23 个。如果架构师要写 23 种模式，代码师要写 23 种代码。但这时候，代码怎么管理？做测试怎么测？以后改变的时候怎么处理？

于 2012 年，高焕堂找到了 EIT 码造形。它包含 3 个要素：基类是<E>，子类是<T>，抽象函数是<I>。全世界所有的代码都可以用这个来表示，就像 1980 年代，大家当时都相信所有的软件都可以用类来表示。现在我们也可以说，所有的软件都可以用 EIT 来表达。

2. 什么是代码造形？

顾名思义，代码造形就是代码层级的设计造形(Form)。代码造形就是开发者常用的词汇(Vocabulary)，其能直接对映(Map)到程序语言的基本结构，此结构大多定义成为关键词(Key word)。例如，指令(Instruction)、函数(Function)和类(Class)。

3. 代码造形的用处

- 因为代码造形能直接对映到程序语言的结构，具有高度的精确性(Precision)，架构师能准精确地传达设计的涵义。
- 因为代码造形是开发者的词汇，架构师以<代码造形>表述架构，基于共同词汇，提升了共识(Shared Understanding)，开发者很容易理解其架构的设计涵意。
- 所以，代码造形能大幅提升开发者的效率；而且迅速配合需求变更、架构创新(或重构)设计，大幅提升了整体团队的敏捷性。

- 架构师如同妈妈，使用 kid language 来与小孩交谈，非常有助于小孩语言天份的开发。同样地，架构师以<代码造形>来表述架构，来与开发者交谈；非常有助于开发者设计能力的提升。
- 架构师自由创意去思考架构设计(加法设计)，但是都以一致的<代码造形>来表述架构设计(减法设计)。就如同唐诗的<七言绝句>造形，李白、杜甫、白居易人人都能发挥创意、尽情思考，但都以一致的造形来表述(Representation)。不但没有伤害创意，而且还基于<诗同形>而相互激发创作的氛围。
- 即使架构設計尚未达到”美好”，只要能清晰而明确地表述，就能随着互相切磋着磨而人人都能進步神速，而邁向”美好”之境了。

4. 大家熟悉的 2 种代码造形：函数和类

在软件开发中，大家最熟悉的代码层级的基本造形有二：函数(Function)和类(Class)。

1970 年代的主要造形：函数(Function)

像 C 语言的代码基本结构就是函数，例如：

```
/*   C 语言程序代码   */
int function add( int x, int y)
        {   int sum;
            sum = x + y;
            return sum;
        }
int function mul( int x, int y)
        {   int sum;
            sum = x * y;
            return sum;
        }
int function exec( int a, int b)
        {
            int k = mul( add(a, b), 100);
        }
void main(){
            printf("%d", exec(3, 5));
        }
```

函数造形简单，其内部的组成要素是：指令(Instruction)，或称叙述(Statement)。也有简单的造形组合规律：线性排列，并相互调用(Function call)。

1980 年代的主要造形：类(Class)

自从 1980 年代到今天，软件开发的主要造形是：类(Class)。类造形并不难理解，它只是对函数造形加以扩大；也就是以函数为基础(保留了函数的各项功能)，扩大结合了属性(Attribute)；让开发者拥有更大的视野，具有更好的整体观。就如同太极图，引导人们掌握更宏大的整体观。

像 C++语言的代码基本结构就是类，例如：

```
//    C++程序代码
class Calculator {
        int x, y, value;
    public:
        void set(m, n){
            x = m;
            y = n;
          }
        void add() {
              value = x + y;
          }
        void mul() {
              value = x * y;
          }
        int get() {
              return value;
          }
}
//-----------------------------------------------
class Adder extends Calculator {
    public:
         int exec(int m, int n){
               set(m, n);
               add();
               set(get(), 100);
```

```
                mul();
                return get();
        }
}
//-----------------------------------------------
void main(){
      Adder adderObj = new Adder();
      printf("%d", adderObj.exec(3, 5);
 }
```

自从 1986 年 C++语言问世以来，类(Class)都是主要的软件代码造形(Form)，例如 C++、Objective-C、Java 和 C#等语言的主要代码造形就是类。类造形内含 2 个要素(更小的组成单位)：属性(Attribute)和函数(Function)。也有清晰的造形组合规律：定义了类与类之间的组合关系，例如上述范例里的"扩充(Extends)"关系等；并透过内含的函数来相互调用。

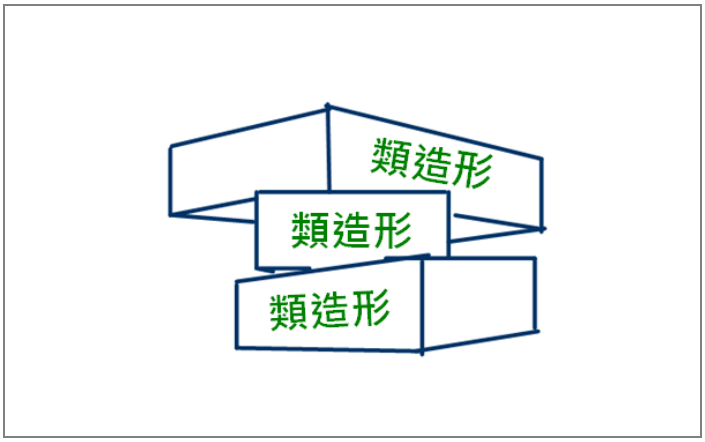
5. 介绍新的 EIT 代码造形

5.1 接口是主角：EIT 呈现接口之美

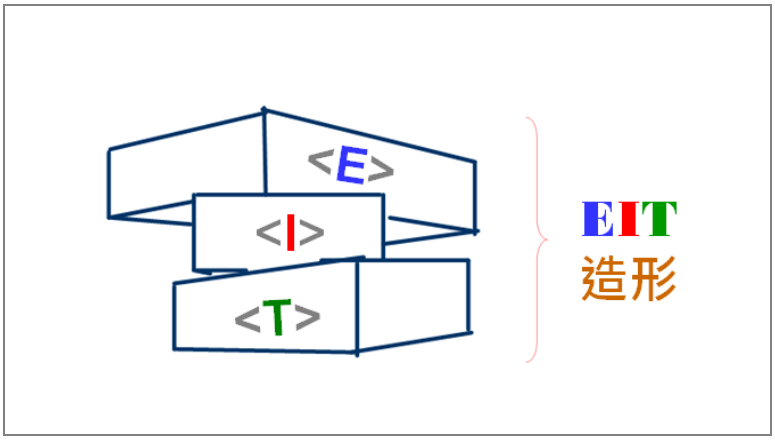
自从 1996 年 Java 问世之后，接口(Interface)成为 Java 语言的关键词(Key Word)。于是，<接口>的位阶已经提升了，其与<类>是同位阶了，而不再隐藏于类造形里。这意味着，我们可以设计一个更大的代码造形来包容类和接口两种元素。为了凸显接口角色，就得考虑两项特性：

- 为了清楚地定义一个接口(主角)，需要两个类来当配角。
- 此外，接口能实现为类(造形)。

于是，高焕堂 老师将 3 个<类造形>组合起来，成为一个更大的造形；就像生物 DNA 的螺旋结构，组合如下图：



在上图里，为于中间的类就是接口实现类。高老师将其命名为 EIT 造形：

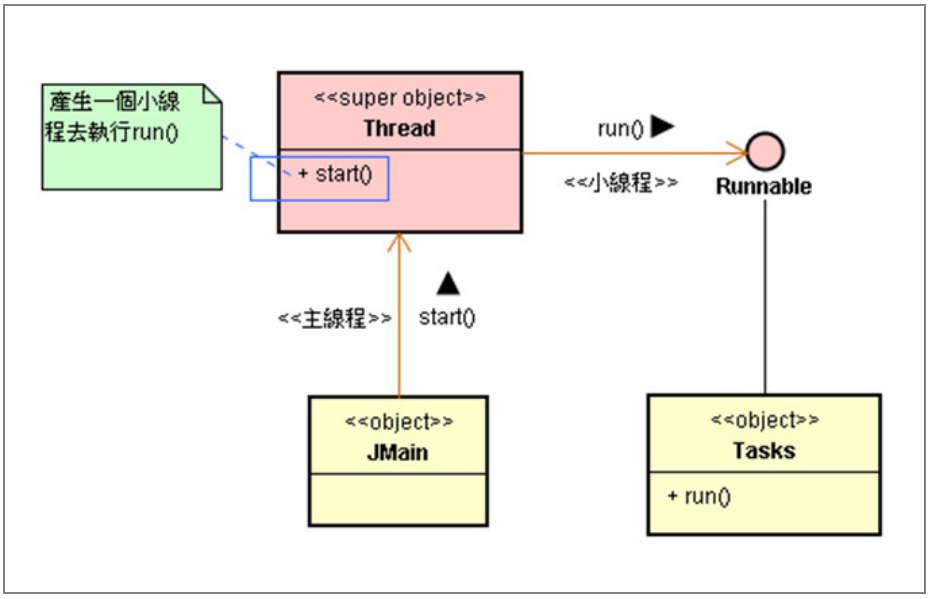


EIT 造形也不难理解，它只是对类造形加以扩大；也就是以类为基础(保留了类的各项功能)，将 3 个类结合在一起，各扮演不同角色；让开发者拥有更大的视野，具有更好的整体观。例如，Java 语言的程序：

```
class Tasks implements Runnable{
        public void run() {
                int sum = 0;
                for (int i = 0; i <= 100; i++)
                        sum += i;
                System.out.println("Result: " + sum);
        }}
// ---------------------------------------------------------
public class JMain {
        public static void main(String[] args) {
            Thread t = new Thread( new Tasks());
```

```
            t.start();
            System.out.println("Waiting...");
        }
}
```

其主要意图是：凸显出<接口>元素与类同位阶的角色。为了清楚地定义一个接口(主角)，需要两个类来当配角。例如，为了凸显 Runnable 接口，而且要精确地表述它，就需要 Thread 和 Tasks 两个类来陪衬，如下图：



由于接口定义是架构师的主要职责，所以 EIT 可以协助架构师清晰地定义接口，非常有助于清晰表达架构。

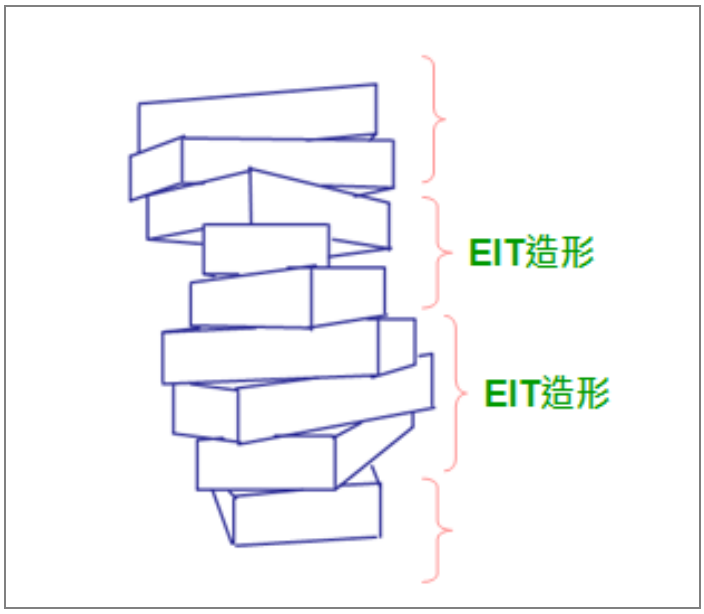
☆ EIT 造形的重复组合

前言：

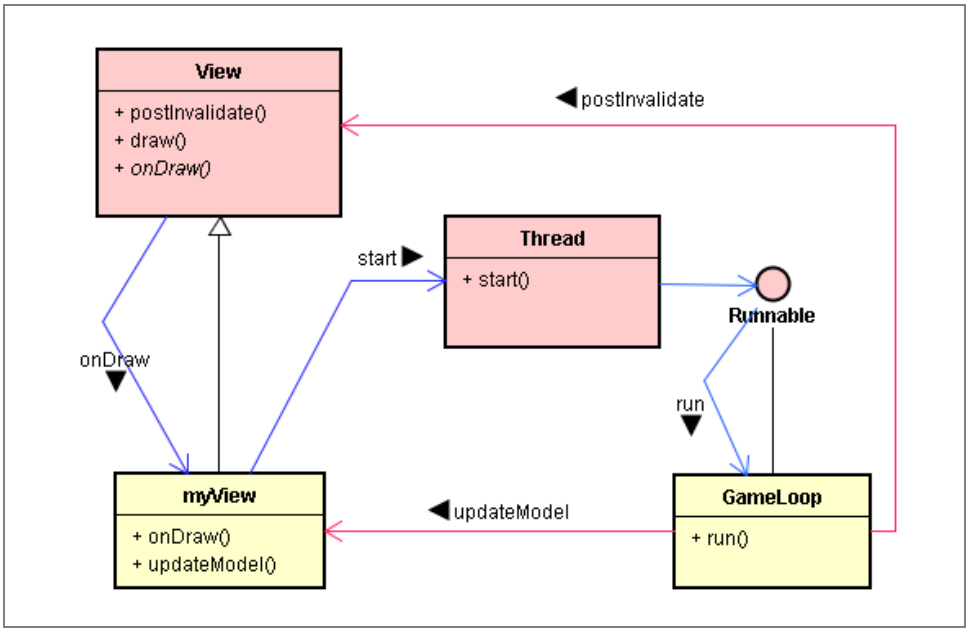
- 自从 1996 年 Java 问世之后，接口(Interface)成为 Java 语言的关键词(Key Word)。
- 我们可以设计一个更大的(EIT)代码造形来包容类和接口两种元素。
- 为了清楚地定义一个接口(主角)，需要两个类来当配角。
- EIT 造形是对类造形加以扩大；也就是以类为基础(保留了类的各项功能)，将 3 个类结合在一起，各扮演不同角色；让开发者拥有更大的视野，具有更好的整体观。

5.2 EIT 造形的重复组合

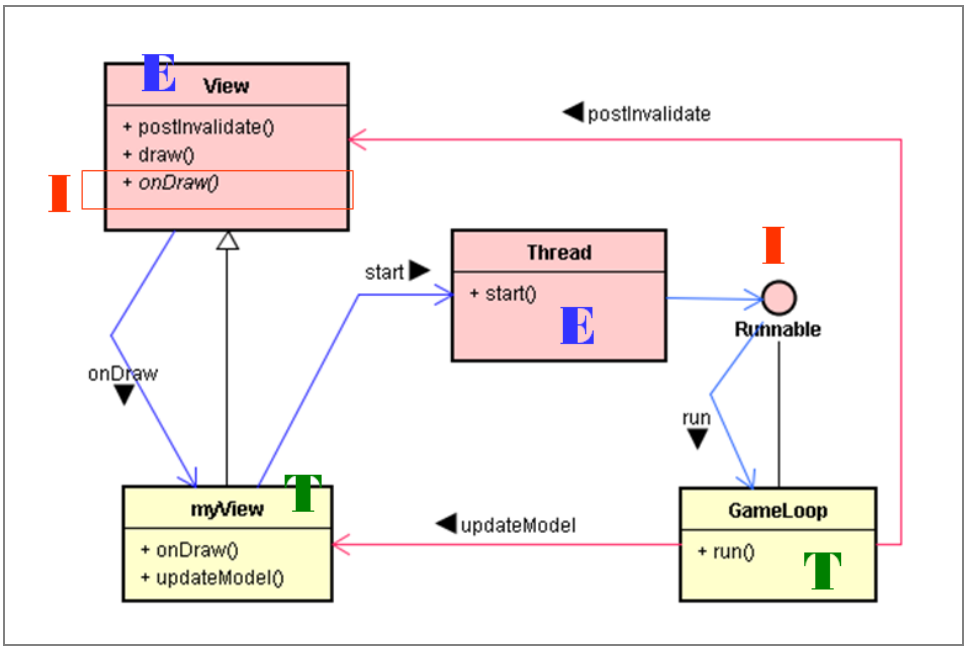
EIT 造形也内部结构简单，也能透过内含的类的组合关系，将 EIT 造形组合起来，轻易地组合出大型而复杂的系统。例如，EIT 造形能像 DNA 螺旋结构一样，组合起来：



EIT 造形提供更宏大的整体观，更易于重构，迅速从简单组合出复杂系统。



这是由两个 EIT 造形(即 Thread 造形和 View 造形)所组成的。



在游戏软件应用上，这个 Thread 造形里的小线程(由 UI 线程所诞生的)扮演一个特殊的角色：成为游戏的主控循环(Game Loop)，而 UI 线程则专注于响应 UI 的事件，创造出两个线程完美分工。由于这个线程专注于游戏主控循环，所以又称为游戏线程(Game Thread)。

游戏线程调用 postInvalidate()函数，间接触发 UI 线程去调用 invalidate()函数了，也触发 View 重新调用 App 子类的 onDraw()去重新绘图了。现在就将上图落实为 Android 程序码，如下：

```
// GameLoop.java
package com.misoo.pk001;

public class GameLoop implements Runnable {
        myView mView;

        GameLoop(myView v){
                mView = v;
        }
        public void run() {
                mView.doUpdate();
                mView.postInvalidateDelayed(1000);
        }
}
```

```
// myView.java
package com.misoo.pk001;
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.view.View;

public class myView extends View {
        private Paint paint= new Paint();
        private int x, y;
        private int line_x = 100;
        private int line_y = 100;
        private float count = 0;

        myView(Context ctx)
                {    super(ctx);    }
        public void doUpdate(){
              if( count > 12) count = 0;
              x = (int) (75.0 * Math.cos(2*Math.PI * count/12.0));
              y = (int) (75.0 * Math.sin(2*Math.PI * count/12.0));
              count++;
         }
         @Override protected void onDraw(Canvas canvas) {
                  super.onDraw(canvas);
               canvas.drawColor(Color.WHITE);
               paint.setColor(Color.BLUE);      paint.setStrokeWidth(3);
               canvas.drawLine(line_x, line_y, line_x+x, line_y+y, paint);
               paint.setStrokeWidth(2);            paint.setColor(Color.RED);
               canvas.drawRect(line_x-5, line_y - 5, line_x+5, line_y + 5, paint);
               paint.setColor(Color.CYAN);
               canvas.drawRect(line_x-3, line_y - 3, line_x+3, line_y + 3, paint);
               Thread gt = new Thread(new GameThread(this));
```

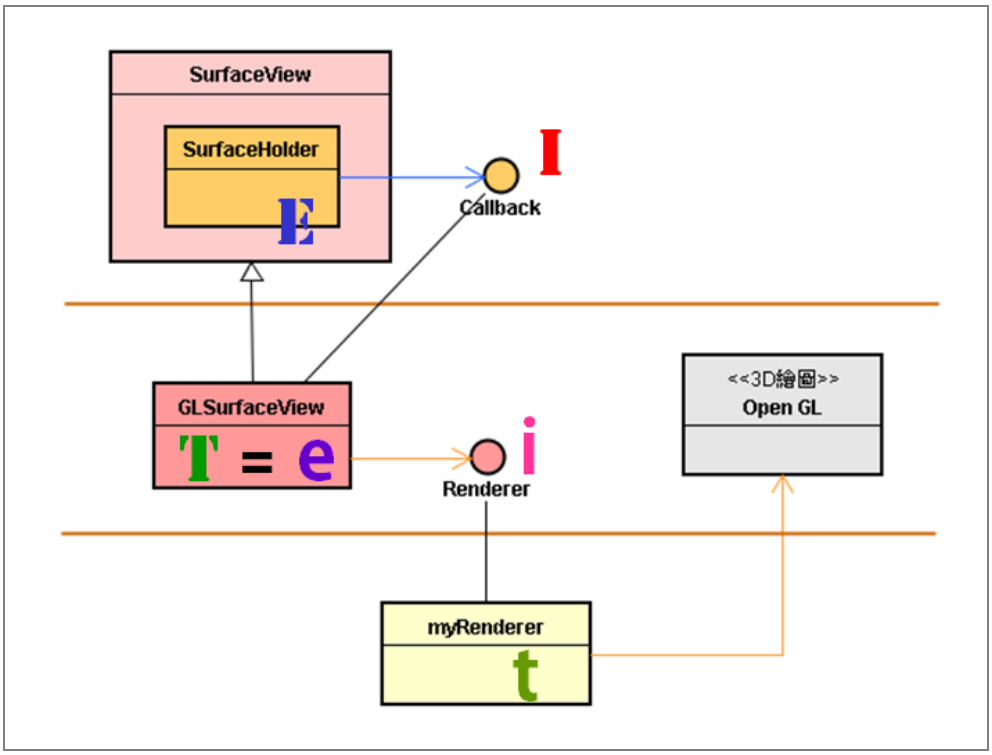
```
            gt.start();
    }}
```

```
// myActivity.java
package com.misoo.pk001;
import com.misoo.pk001.R;
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 myActivity extends Activity implements OnClickListener {
        private myView mv = null;
        private Button ibtn;

        @Override protected void onCreate(Bundle icicle) {
                super.onCreate(icicle);
                show_layout_01();
        }
        public void show_layout_01(){
                LinearLayout layout = new LinearLayout(this);
                layout.setOrientation(LinearLayout.VERTICAL);
                mv = new myView(this);
                LinearLayout.LayoutParams param =
                        new LinearLayout.LayoutParams(200, 200);
                param.topMargin = 10;
                param.leftMargin = 10;
                layout.addView(mv, param);
                //-----------------------------------------------
                ibtn = new Button(this);
                ibtn.setOnClickListener(this);
                ibtn.setText("Exit");
                ibtn.setBackgroundResource(R.drawable.gray);
                LinearLayout.LayoutParams param1 =
                        new LinearLayout.LayoutParams(200, 65);
                param1.topMargin = 10;      param1.leftMargin = 10;
                layout.addView(ibtn, param1);
                //-----------------------------------------------
                setContentView(layout);
         }
        public void onClick(View v)
                    {    finish();    }
        }
```

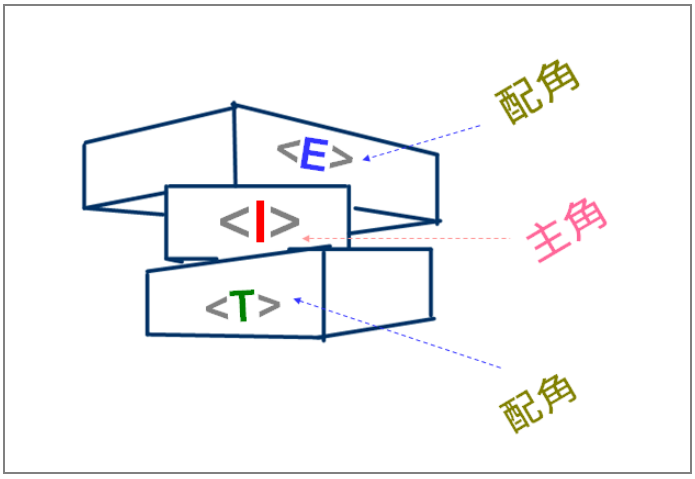
由于 EIT 造形只是对类造形加以扩大；也就是以类为基础(保留了类的各项功能)，将 3 个类结合在一起，各扮演不同角色。所以，只要利用类造形既有的组合机制，就能将 EIT 造形组合起来，成为复杂的系统了。例如，也能组合如下图：



这就是双层 EIT 造形的架构设计了。

5.3 为什么需要要有两个配角呢?

虽然从代码造形来看，<E>、<I>和<T>三者是同位阶的，但从架构师角度上，<I>属于主角，而<E>和<T>是配角。搭配两个配角，才能将<I>表述的完整而清晰。



搭配两个配角，就能将<I>表述的更完整而清晰。

拿英语来比喻

就如同英语，搭配了主词(Subject)和受词(Object)，就能够将动词(Verb)表述得更完整而清晰。例如，

- play

  ---- 没有主词和受词，动词<play>就显得意义不够清晰。

- 猫 玩(play) 绣球

  ---- 有了主词和受词，动词<play>就显得意义很清晰。

- 老师 弹(play) 钢琴

  ---- 有了主词和受词，动词<play>就显得意义很清晰。

拿厕所来比喻

依据传统的类造形，架构师会表述为：设计一个厕所类(对映到实际的厕所)，其提供接口(实际厕所的入口)给男生或女生使用。这很可能引导开发者先建置厕所，然后才提供入口给用户使用；经常是先类而后接口。

反之，如果使用 EIT 造形，则架构师就表述为：(女生<E>、入口<I>、厕所<T>)，以及<男生<E>、入口<I>、厕所<T>)。架构师考虑到女生和厕所，而定设计入口<Iw>，然后交给开发者去撰写<Tw>代码。同样地，架构师考虑到男生和厕所，而定设计入口<Im>，然后交给开发者去撰写<Tm>代码。架构师设计出来的<Iw>和<Im>是不一样的；同理，开发者撰写的<Tw>和<Tm>也是不一样的。架构师与开发者的沟通变得清晰而完整。这有效引导架构师先定义明确的<I>，然后提醒架构师搭配<E>和<T>来表述<I>。◆

~ End ~