嵌入式系统软件架构设计

目录

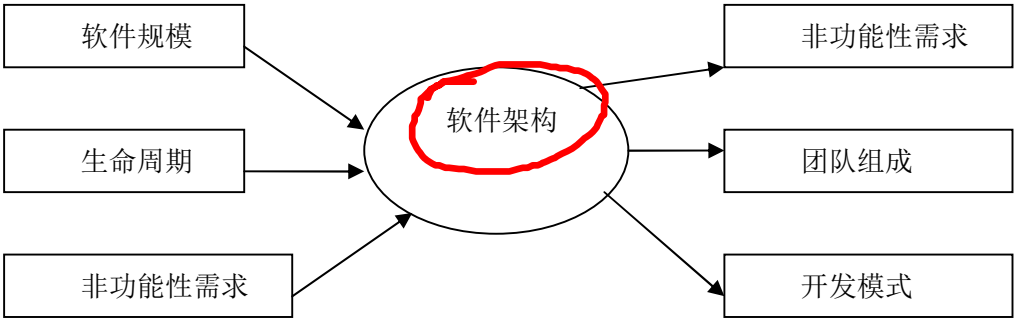
1. 前言

嵌入式是软件设计领域的一个分支，它自身的诸多特点决定了系统架构师的选择，同时它的一些问题又具有相当的通用性，可以推广到其他的领域。本课程试图从嵌入式软件架构设计出发，启发大家对软件架构设计的理解。本课程的很多内容是对谢老师课程在嵌入式领域的具体阐述。

提起嵌入式软件设计，传统的印象是单片机，汇编，高度依赖硬件。传统的嵌入式软件开发者往往只关注实现功能本身，而忽视诸如代码复用，数据和界面分离，可测试性等因素。从而导致嵌入式软件的质量高度依赖开发者的水平，成败系之一身。随着嵌入式软硬件的飞速发展，今天的嵌入式系统在功能，规模和复杂度各方面都有了极大的提升。比如，Marvell 公司的 PXA3xx 系列的最高主频已经达到 800Mhz，内建 USB,WIFI,2D 图形加速,32 位 DDR 内存。在硬件上，今天的嵌入式系统已经达到甚至超过了数年前的 PC 平台。在软件方面，完善的操作系统已经成熟，比如 Symbian, Linux, WinCE。基于完善的操作系统，诸如字处理，图像，视频，音频，游戏，网页浏览等各种应用程序层出不穷，其功能性和复杂度比诸 PC 软件不遑多让。原来多选用专用硬件和专用系统的一些商业设备公司也开始转换思路，以出色而廉价的硬件和完善的操作系统为基础，用软件的方式代替以前使用专有硬件实现的功能，从而实现更低的成本和更高的可变更，可维护性。

2. 决定架构的因素和架构的影响

架构不是一个孤立的技术的产物，它受多方面因素的影响。同时，一个架构又对软件开发的诸多方面造成影响。



下面举一个具体的例子。

摩托车的发动机在出厂前必须通过一系列的测试。在流水线上，发动机被送到每个工位上，由工人进行诸如转速，噪音，振动等方面的测试。要求实现一个嵌入式设备，具备以下基本功能:

1. 安装在工位上，工人上班前开启并登录。
2. 通过传感器自动采集测试数据，并显示在屏幕上。
3. 记录所有的测试结果，并提供统计功能。比如次品率。

如果你是这个设备的架构师，哪些问题是在设计架构的时候应该关注的呢？

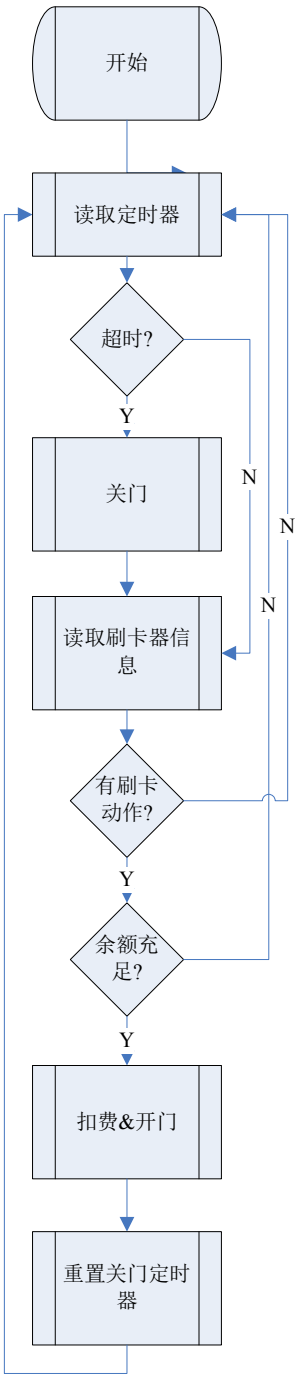
2.1. 常见的误解

2.1.1. 小型的系统不需要架构

有相当多的嵌入式系统规模都较小，一般是为了某些特定的目的而设计的。受工程师认识，客户规模和项目进度的影响，经常不做任何架构设计，直接以实现功能为目标进行编码。这种行为表面上看满足了进度，成本，功能各方面的需求，但是从长远来看，在扩展和维护上付出的成本，要远远高于最初节约的成本。如果系统的最初开发者继续留在组织内并负责这个项目，那么可能一切都会正常，一旦他离开，后续者因为对系统细节的理解不足，就可能引入更多的错误。要注意，嵌入式系统的变更成本要远远高于一般的软件系统。好的软件架构，可以从宏观和微观的不同层次上描述系统，并将各个部分隔离，从而使新特性的添加和后续维护变得相对简单。

举一个城铁刷卡机的例子，这个例子在前面的课程中出现过。

简单的城铁刷卡机只需要实现如下功能：



一个 While 循环足以实现这个系统，直接就可以开始编码调试。但是从一个架构师的角度，这里有没有值得抽象和剥离的部分呢？

1. 计费系统。计费系统是必须抽象的，比如从单次计费到按里程计费。
2. 传感器系统。传感器包括磁卡感应器，投币器等。设备可能更换。
3. 故障处理和恢复。考虑到较高的可靠性和较短的故障恢复时间，这部分有必要单独设计。

未来很可能出现的需求变更：

1. 操作界面。是否需要抽象出专门的 Model 来？以备将来实现 View。

2. 数据统计。是否需要引入关系型数据库？

如果直接以上面的流程图编码，当出现变更后，有多少代码可以复用？

不过，也不要因此产生过度的设计。架构应当立足满足当前需求，并适当的考虑重用和变更。

2.1.2. 敏捷开发不需要架构

极限编程，敏捷开发的出现使一些人误以为软件开发无需再做架构了。这是一个很大的误解。敏捷开发是在传统瀑布式开发流程出现明显弊端后提出的解决方案，所以它必然有一个更高的起点和对开发更严格的要求。而不是倒退到石器时代。事实上，架构是敏捷开发的一部分，只不过在形式上，敏捷开发推荐使用更高效，简单的方式来做设计。比如画在白板上然后用数码相机拍下的 UML 图；用用户故事代替用户用例等。测试驱动的敏捷开发更是强迫工程师在写实际代码前设计好组件的功能和接口，而不是直接开始写代码。

敏捷开发的一些特征：

1. 针对比传统开发流程更大的系统
2. 承认变化，迭代架构
3. 简洁而不混乱
4. 强调测试和重构

3. 嵌入式环境下软件设计的特点

要谈嵌入式的软件架构，首先必须了解嵌入式软件设计的特点。

3.1. 和硬件密切相关

嵌入式软件普遍对硬件有着相当的依赖性。这体现在几个方面：

1. 一些功能只能通过硬件实现,软件操作硬件，驱动硬件。
2. 硬件的差异/变更会对软件产生重大影响。
3. 没有硬件或者硬件不完善时，软件无法运行或无法完整运行。

这些特点导致几方面的后果：

1. 软件工程师对硬件的理解和熟练程度会很大程度的决定软件的性能/稳定性等非功能性指标，而这部分一向是相对复杂的，需要资深的工程师才能保证质量。
2. 软件对硬件设计高度依赖，不能保持相对稳定，可维护性和可重用性差
3. 软件不能离开硬件单独测试和验证，往往需要和硬件验证同步进行，造成进度前松后紧，错误定位范围扩大。

针对这些问题，有几方面的解决思路：

1. 用软件实现硬件功能。选用更强大的处理器，用软件来实现部分硬件功能，不仅可以降低对硬件的依赖，在响应变化，避免对特定型号和厂商的依赖方面都很有好处。这在一些行业里已经成为了趋势。在 PC 平台也经历了这样的过程，比如早期的汉

卡。

2. 将对硬件的依赖独立成硬件抽象层，尽可能使软件的其他部分硬件无关，并可以脱离硬件运行。一方面将硬件变更甚至换件的风险控制在有限的范围内，另一方面提高软件部分的可测试性。

3.2. 稳定性要求高

大部分嵌入式软件都对程序的长期稳定运行有较高的要求。比如手机经常几个月开机，通讯设备则要求 24*7 正常运行，即使是通讯上的测试设备也要求至少正常运行 8 小时。为了稳定性的目标，有一些比较常用的设计手段：

1. 将不同的任务分布在独立的进程中。良好的模块化设计是关键
2. Watch Dog, Heart beat，重新启动失效的进程。 看门狗、心跳包
3. 完善而统一的日志系统以快速定位问题。嵌入式设备一般缺乏有力的调试器，日志系统尤其重要。
4. 将错误孤立在最小的范围内，避免错误的扩散和连锁反应。核心代码要经过充分的验证，对非核心代码，可以在监控或者沙盒中运行，避免其破坏整个系统。
   举例，Symbian 上的 GPRS 访问受不同硬件和操作系统版本影响，功能不是非常稳定。其中有一个版本上当关闭 GPRS 连接时一定会崩溃，而且属于 known issue。将 GPRS 连接，HTTP 协议处理，文件下载等操作独立到一个进程中，虽然每次操作完毕该进程都会崩溃，对用户却没有影响。
5. 双备份这样的手段较少采用

3.3. 内存不足

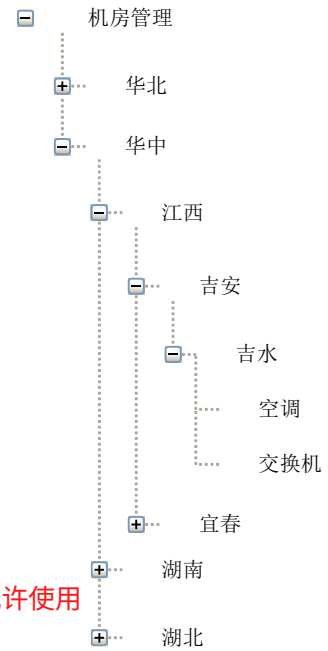
虽然当今的嵌入式系统的内存比之以 K 计数的时代已经有了很大的提高，但是随着软件规模的增长，内存不足的问题依然时时困扰着系统架构师。有一些原则，架构师在进行设计决策的时候可以参考：

3.3.1. 虚拟内存技术

有一些嵌入式设备需要处理巨大的数据量，而这些数据不可能全部装入内存中。一些嵌入式操作系统不提供虚拟内存技术，比如 WinCE4.2 每个程序最多只能使用 32M 内存。对这样的应用，架构师应该特别设计自己的虚拟内存技术。所谓的虚拟内存技术的核心是，将暂时不太可能使用的数据移出内存。这涉及到一些技术点：

1. 引用计数，正在使用的数据不能移出。
2. 使用预测，预测下一个阶段某个数据的使用可能性。基于预测移出数据或者提前装入数据。
3. 占位数据/对象。
4. 高速缓存。在复杂数据结果下缓存高频率使用的数据，直接访问。
5. 快速的持久化和装载。

下图是一个全国电信机房管理系统的界面示意图：



用户点击哪个节点，就加载哪个节点。
一开始时，只加载前几个节点省份，
这样节省内存。
如果一下子，把所有省份的数据都加载到内存，
内存占用率就太大了。并且操作系统对单个进程允许使用的最大内存数有限定，会导致申请内存失败！

每个节点下都有大量的数据需要装载，可以使用上述技术将内存占用降到最低。

3.3.2. 两段式构造

在内存有限的系统里，对象构造失败是必须要处理的问题,失败的原因中最常见的则是内存不足（实际上这也是对 PC 平台的要求，但是在实际中往往忽略，因为内存实在便宜）。两段式构造就是一种常用而有效的设计。举例来说：

CMySimpleClass:

```
class CMySimpleClass
  {
  public:
    CMySimpleClass();
    ~CMySimpleClass();
    ...
  private:
    int SomeData;
};
```

CMyCompoundClass：

```
class CMyCompoundClass
{
      public:
        CMyCompoundClass();
        ~CMyCompoundClass();
       ...
      private:
        CMySimpleClass* iSimpleClass;
};
```

在 CMyCompoundClass 的构造函数里初始化 iSimpleClass 对象。

```
CMyCompoundClass::CMyCompoundClass()
{
      iSimpleClass = new CMySimpleClass;
}
```

当创建 CMyCompoundClass 的时候会发生什么呢？

```
CMyCompoundClass* myCompoundClass = new CMyCompoundClass;
```

1. 为 CMyCompoundClass 的对象分配内存
2. 调用 CMyCompoundClass 对象的构造函数
3. 在构造函数中创建一个 CMySimpleClass 的实例
4. 构造函数结束返回

一切看起来都很简单，但是如果第三步创建 CMySimpleClass 对象的时候发生内存不足的错误怎么办呢？构造函数无法返回任何错误信息以提示调用者构造没有成功。调用者于是获得了一个指向 CMyCompoundClass 的指针，但是这个对象并没有构造完整。

如果在构造函数中抛出异常会怎么样呢？这是个著名的噩梦，因为析构函数不会被调用，在创建 CMySimpleClass 对象之前如果分配了资源就会泄露。关于在构造函数中抛出异常可以单讲一个小时，但是有一个建议是：尽量避免在构造函数中抛出异常。

所以，使用两段式构造法是一个更好的选择。简单的说，就是在构造函数避免任何可能产生错误的动作，比如分配内存，而把这些动作放在构造完成之后，调用另一个函数。比如:

```
AddressBook* book = new AddressBook()
If(!book->Construct())
{
  delete book;
  book = NULL;
}
```

在使用Qt(C++)编程的时候，也一定要使用二段式构造法，因为构造函数无法返回错误值。

这样可以保证当 Construct 不成功的时候释放已经分配的资源。

在最重要的手机操作系统 Symbian 上，二段式构造法普遍使用。

3.3.3. 内存分配器

不同的系统有着不同的内存分配的特点。有些要求分配很多小内存，有的则需要经常增

长已经分配的内存。一个好的内存分配器对嵌入式的软件的性能有时具有重大的意义。应该在系统设计时保证整个系统使用统一的内存分配器，并且可以随时更换。

3.3.4. 内存泄漏

内存泄漏对嵌入式系统有限的内存是非常严重的。通过使用自己的内存分配器，可以很容易的跟踪内存的分配释放情况，从而检测出内存泄漏的情况。

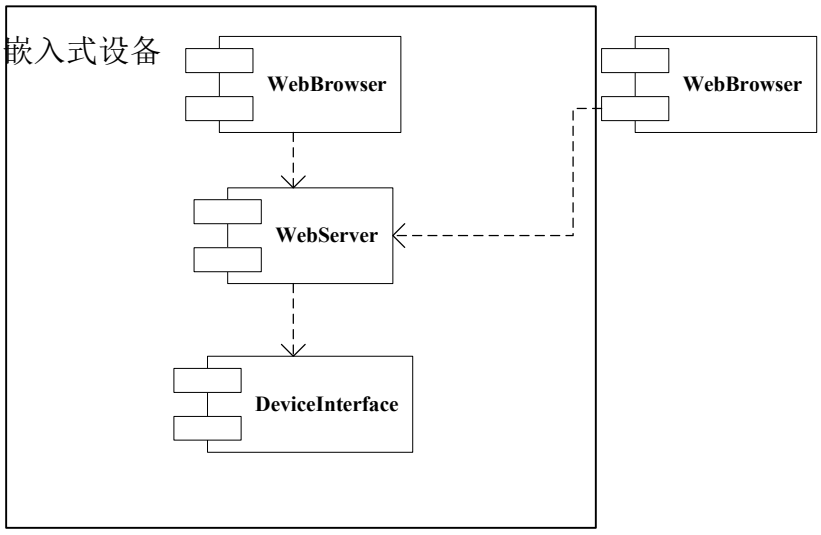
3.4. 处理器能力有限，性能要求高

这里不讨论实时系统，那是一块很大的专业话题。对一般的嵌入式系统而言，由于处理器能力有限，要特别注意性能的问题。一些很好的架构设计由于不能满足性能要求，最终导致整个项目的失败。

3.4.1. 抵御新技术的诱惑

架构师必须明白，新技术常常意味着复杂和更低的性能。即使这不是绝对的，由于嵌入式系统硬件性能所限，弹性较低。一旦发现新技术有和当初设想不同之处，就更难通过修改来适应。比如 GWT 技术。这是 Google 推出的 Ajax 开发工具，它可以让程序员像开发一个桌面应用程序一样开发 Web 的 Ajax 程序。这使得在嵌入式系统上用一套代码实现远程和本地操作界面成为了很容易的一件事。但是在嵌入式设备上运行 B-S 结构的应用，性能上是一个很大的挑战。同时，浏览器兼容方面的问题也很严重，GWT 目前的版本还不够完善。

事实证明，嵌入式的远程控制方案还是要采用 Activex VNC 或者其他的方案。



3.4.2. 不要有太多的层次

分层结构有利于清晰的划分系统职责，实现系统的解耦，但是每多一个层次，就意味着性能的一次损失。尤其是当层和层之间需要传递大量数据的时候。对嵌入式系统而言，在采

用分层结构时要控制层次数量，并且尽量不要传递大量数据，尤其是在不同进程的层次之间。如果一定要传递数据，要避免大量的数据格式转换，如 XML 到二进制，C++结构到 Python 结构。

嵌入式系统能力有限，一定要将有限的能力用在系统的核心功能上。

3.5. 存储设备易损坏，速度较慢

受体积和成本的限制，大部分的嵌入式设备使用诸如 Compact Flash, SD, mini SD, MMC 等作为存储设备。这些设备虽然有着不担心机械运动损坏的优点，但是其本身的使用寿命都比较短暂。比如，CF 卡一般只能写 100 万次。而 SD 更短，只有 10 万次。对于像数码相机这样的应用，也许是足够的，但是对于需要频繁擦写磁盘的应用，比如历史数据库，磁盘的损坏问题会很快显现。比如有一个应用式每天向 CF 卡上写一个 16M 的文件，文件系统是 FAT16，每簇大小是 2K，那么写完这个 16M 的文件，分区表需要写 8192 次，于是一个 100 万次寿命的 CF 实际能够工作的时间是 1000000/8192 = 122 天。而损坏的时候，CF 卡的其他绝大部分地方的使用次数不过万分之一。

除了因为静态的文件分区表等区块被频繁的读写而提前损坏，一些嵌入式设备还要面对直接断电的挑战，这会在存储设备上产生不完整的数据。

3.5.1. 损耗均衡

损耗均衡的基本思路是平均地使用存储器上的各个区块。需要维护一张存储器区块使用情况的表，这个表包括区块的偏移位置，当前是否可用，以及已经擦写地次数。当有新的擦写请求的时候，根据以下原则选择区块：

1. 尽量连续
2. 擦写次数最少

即使是更新已经存在的数据，也会使用以上原则分配新的区块。同样，这张表的存放位置也不能是固定不变的，否则这张表所占据的区块就会最先损坏。当要更新这张表的时候，同样要使用以上算法分配区块。

如果存储器上有大量的静态数据，那么上述算法就只能针对剩下的空间生效，这种情况下还要实现对这些静态数据的搬运的算法。但是这种算法会降低写操作的性能，也增加了算法的复杂度。一般都只使用动态均衡算法。

目前比较成熟的损耗均衡的文件系统有 JFFS2、和 YAFFS。也有另一种思路就是在 FAT16 等传统文件系统上实现损耗均衡，只要事先分配一块足够大的文件，在文件内部实现损耗均衡算法。不过必须修改 FAT16 的代码，关闭对最后修改时间的更新。

现在的 CF 卡和 SD 卡有的已经在内部实现了损耗均衡，这种情况下就不需要软件实现了。

3.5.2. 错误恢复

如果在向存储器写数据的时候发生断电或者被拔出，那么所写的区域的数据就处于未知

的状态。在一些应用中，这会导致不完整的文件，而在另一些应用中，则会导致系统失败。所以对这类错误的恢复也是嵌入式软件设计必须考虑的。常用的思路有两种：

1. 日志型的文件系统

这种文件系统并不是直接存储数据，而是一条条的日志，所以当发生断电的时候，总可以恢复到之前的状态。这类文件系统的代表如 ext3。

2. 双备份

双备份的思路更简单，所有的数据都写两份。每次交替使用。文件分区表也必须是双备份的。假设有数据块 A，A1 是他的备份块，在初始时刻和 A 的内容是一致的。在分区表中，F 指向数据块 A，F1 是他的备份块。当修改文件时，首先修改数据块 A1 的内容，如果此时断电，A1 的内容错误，但因为 F 指向的是完好的 A，所以数据没有损坏。如果 A1 修改成功，则修改 F1 的内容，如果此时断电，因为 F 是完好的，所以依然没有问题。

现在的 Flash 设备，有的已经内置错误检测和错误校正技术，可以保证在断电时数据的完整。还有的包括自动的动态/静态损耗均衡算法和坏块处理，完全无须上层软件额外对待，可以当作硬盘使用。所以，硬件越发达，软件就会越可靠，技术不断的进步，将让我们可以把更多的精力投入到软件功能的本身，这是发展的趋势。

3.6. 故障成本高昂

嵌入式产品都是软硬件一起销售的给用户的，所以这带来了一个纯软件所不具备的问题，那就是当产品发生故障时，如果需要返厂才能修复，则成本就很高。嵌入式设备常见有以下的几类故障：

a) 数据故障。由于某些原因导致数据不能读出或者不一致。比如断电引起的数据库错误。
b) 软件故障。软件本身的缺陷，需要通过发布补丁程序或者新版本的软件修正。
c) 系统故障。比如用户下载了错误的系统内核，导致系统无法启动。
d) 硬件故障。这种故障只有返厂，不属于我们的讨论范围。

针对前三类故障，要尽可能保证客户自己，或者现场技术人员就可以解决。从架构的角度考虑，如下原则可以参考：

a) 使用具备错误恢复能力的数据管理设计。当数据发生错误时，用户可以接受的处理依次是：
    i. 错误被纠正，所有数据有效
    ii. 错误发生时的数据（可能不完整）丢失，之前的数据有效。
    iii. 所有数据丢失
    iv. 数据引擎崩溃无法继续工作

    一般而言，满足第二个条件即可。(日志，事务，备份，错误识别)
b) 将应用程序和系统分离。应用程序应该放置在可插拔的 Flash 卡上，可以通过读卡器进行文件复制升级。非必要的情况不要使用专用应用软件来升级应用程序。
c) 要有“安全模式”。即当主系统被损坏后，设备依然可以启动，重新升级系统。常见的 uboot 可以保证这一点，在系统损坏后，可以进入 uboot 通过 tftp 重新升级。

4. 软件框架

在桌面系统和网络系统上，框架是普遍应用的，比如著名的 ACE, MFC, Ruby On Rails 等。而在嵌入式系统中，框架则是很少使用的。究其原因，大概是认为嵌入式系统简单，没有重复性，过于注重功能的实现和性能的优化。在前言中我们已经提到，现在的嵌入式发展趋势是向着复杂化，大型化，系列化发展的。所以，在嵌入式下设计软件框架也是很有必要，也很有价值的。

4.1. 嵌入式软件架构面临的问题

前面我们讲到，嵌入式系统软件架构所面临的一些问题，其中很重要的一点是，对硬件的依赖和硬件相关软件的复杂性。还包括嵌入式软件在稳定性和内存占用等方面的苛刻要求。如果团队中的每个人都是这些方面高手的话，也许有可能开发出高质量的软件，但事实是一个团队中可能只有一两个资深人员，其他大部分都是初级工程师。人人都去和硬件打交道，都负责稳定性，性能等等指标的话，是很难保证最终产品质量的。如果组件团队时都是精通硬件等底层技术的人才，又很难设计出在可用性，扩展性等方面出色的软件。术业有专攻，架构师的选择决定着团队的组成方式。

同时，嵌入式软件开发虽然复杂，但是也存在大量的重用的可能性。如何重用，又如何应对将来的变更？

所以，如何将复杂性对大多数人屏蔽，如何将关注点分离，如何保证系统的关键非功能指标，是嵌入式软件架构设计师应该解决的问题。一种可能的解决方案就是软件框架。

4.2. 什么是框架

框架是在一个给定的**问题领域内**，为了**重用**和应对未来需求**变化**而设计的**软件半成品**。框架强调对特定领域的抽象，包含大量的专业领域知识，以缩短软件的开发周期，提高软件质量为目的。使用框架的二次开发者通过重写子类或组装对象的方式来实现特殊的功能。

4.2.1. 软件复用的层次

复用是在我们经常谈到的话题，“不要重复发明轮子”也是耳熟能详的戒条。不过对于复用的理解实际上是有很多个层次的。

最基础的复用是复制粘贴。某个功能以前曾经实现过，再次需要的时候就复制过来，修改一下就可以使用。经验丰富的程序员一般都会有自己的程序库，这样他们实现的时候就会比新的程序员快。复制粘贴的缺点是代码没有经过抽象，往往并不完全的适用，所以需要进行修改，经过多次复用后，代码将会变得混乱，难以理解。很多公司的产品都有这个问题，一个产品的代码从另一个产品复制而来，修改一下就用，有时候甚至类名变量名都不改。按照“只有为复用设计的代码才能真正复用”的标准，这称不上是复用，或者说是低水平的复用。

更高级的复用是则是库。这种功能需要对经常使用的功能进行抽象，提取出其中恒定不变的部分，以库的形式提供给二次开发程序员使用。因为设计库的时候不知道二次开发者会

如何使用，所以对设计者有着很高的要求。这是使用最广泛的一种复用，比如标准 C 库，STL 库。现在非常流行的 Python 语言的重要优势之一就是其库支持非常广泛，相反 C++一直缺少一个强大统一的库支持，成为短板。在公司内部的开发中总结常用功能并开发成库是很有价值的，缺点是对库的升级会影响到很多的产品，必须慎之又慎。

框架是另一种复用。和库一样，框架也是对系统中不变的部分进行抽象并加以实现，由二次开发者实现其他变化的部分。典型的框架和库的最大的区别是，库是静态的，由二次开发者调用的；框架是活着的，它是主控者，二次开发者的代码必须符合框架的设计，由框架决定在何时调用。

举个例子，一个网络应用总是要涉及到连接的建立，数据收发和连接的关闭。以库的形式提供是这样的：

```
conn = connect(host,port);
if(conn.isvalid())
{
  data = conn.recv();
  printf(data);
  conn.close();
}
```

框架则是这样的：

```
class mycomm:class connect
{
  public:
     host();
     port();
     onconnected();
      ondataarrived(unsigned char* data, int len);
      onclose();
};
```

有点类似于回调函数的味道

框架会在“适当”的时机创建 mycomm 对象，并查询 host 和 port，然后建立连接。在连接建立后，调用 onconnected()接口，给二次开发者提供进行处理的机会。当数据到达的时候调用 ondataarrived 接口让二次开发者处理。这是好莱坞原则，“不要来找我们，我们会去找你”。

当然，一个完整的框架通常也要提供各种库供二次开发者使用。比如 MFC 提供了很多的库，如 CString，但本质上它是一个框架。比如实现一个对话框的 OnInitDialog 接口，就是由框架规定的。

4.2.2. 针对高度特定领域的抽象

和库比较起来，框架是更针对特定领域的抽象。库，比如 C 库，是面向所有的应用的。而框架相对来说则要狭窄的多。比如 MFC 提供的框架只适合于 Windows 平台的桌面应用程

序开发，ACE 则是针对网络应用开发的框架，Ruby On Rails 是为快速开发 web 站点设计的。

越是针对特定的领域，抽象就可以做的越强，二次开发就可以越简单，因为共性的东西越多。比如我们上面谈到嵌入式系统软件开发的诸多特点，这就是特定领域的共性，就属于可以抽象的部分。具体到实际的嵌入式应用，又会有更多的共性可以抽象。

框架的设计目的是总结特定领域的共性，以框架的方式实现，并规定二次开发者的实现方式，从而简化开发。相应的，针对一个领域开发的框架就不能服务于另一个领域。对企业而言，框架是一种极好的积累知识，降低成本的技术手段。

4.2.3. 解除耦合和应对变化

框架设计的一个重要目的就是应对变化。应对变化的本质就是解耦。从架构师的角度看，解耦可以分为三种：

1. 逻辑解耦。逻辑解耦是将逻辑上不同的模块抽象并分离处理。如数据和界面的解耦。这也是我们最常做的解耦。
2. 知识解耦。知识解耦是通过设计让掌握不同知识的人仅仅通过接口工作。典型的如测试工程师所掌握的专业知识和开发工程师所掌握的程序设计和实现的知识。传统的测试脚本通常是将这二者合二为一的。所以要求测试工程师同时具备编程的能力。通过适当的方式，可以让测试工程师以最简单的方式实现他的测试用例，而开发人员编写传统的程序代码来执行这些用例。
3. 变与不变的解耦。这是框架的重要特征。框架通过对领域知识的分析，将共性，也就是不变的内容固定下来，而将可能发生变化的部分交给二次开发者实现。

4.2.4. 框架可以实现和规定非功能性需求

非功能性需求是指如性能，可靠性，可测试性，可移植性等。这些特性可以通过框架来实现。以下我们一一举例。

性能。对性能的优化最忌讳的就是普遍优化。系统的性能往往取决于一些特定的点。比如在嵌入式系统中，对存储设备的访问是比较慢的。如果开发者不注意这方面的问题，频繁的读写存储设备，就会造成性能下降。如果对存储设备的读写由框架设计，二次开发者只作为数据的提供和处理者，那么就可以在框架中对读写的频率进行调节，从而达到优化性能的目的。由于框架都是单独开发的，完成后供广泛使用，所以就有条件对关键的性能点进行充分的优化。

可靠性。以上面的网络通讯程序为例，由于框架负责了连接的创建和管理，也处理了各种可能的网络错误，具体的实现者无须了解这方面的知识，也无须实现这方面错误处理的代码，就可以保证整个系统在网络通讯方面的可靠性。以框架的方式设计在可靠性方面的最大优势就是：二次开发的代码是在框架的掌控之内运行的。一方面框架可以将容易出错的部分实现，另一方面对二次开发的代码产生的错误也可以捕获和处理。而库则不能代替使用者处理错误。

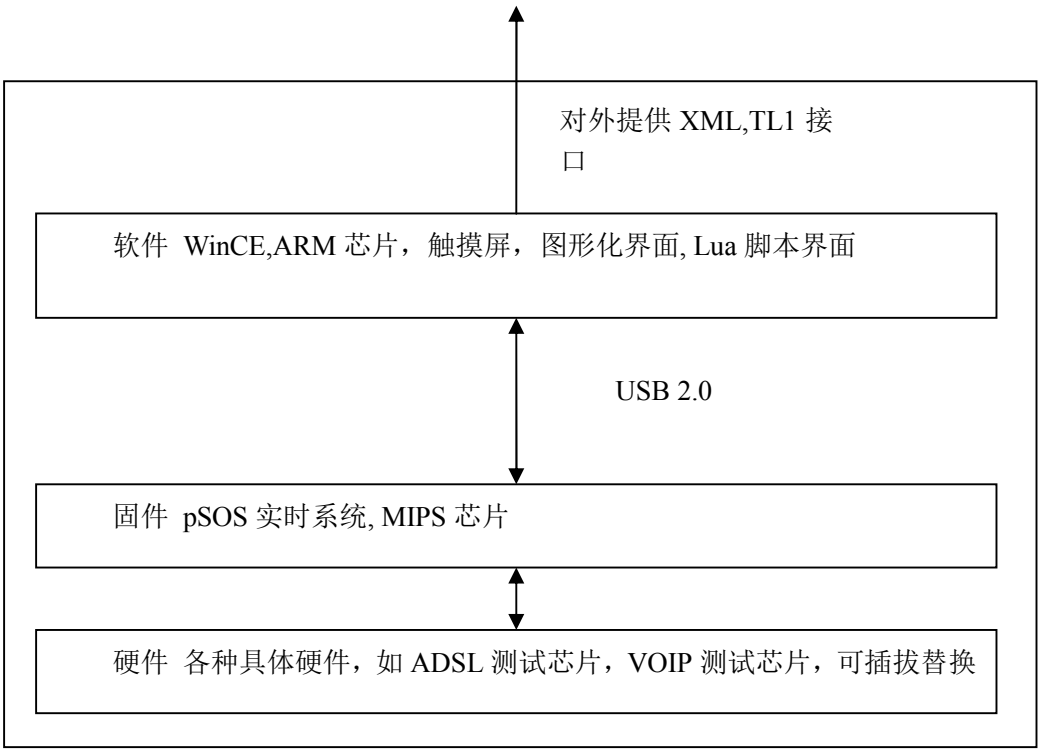
可测试性。可测试性是软件架构需要考虑的一个重要方面。下面的章节会讲到，软件的可测试性是由优良的设计来保证的。一方面，由于框架规定了二次开发的接口，所以可以迫使二次开发者开发出便于进行单元测试的代码。另一方面，框架也可以在系统测试的层面上提供易于实现自动化测试和回归测试的设计，例如统一提供的 TL1 接口。

可移植性。如果软件的可移植性是软件设计的目标，框架设计者可以在设计阶段来保证

这一点。一种方式是通过跨平台的库来屏蔽系统差异，另一种可能的方式更加极端，基于框架的二次开发可以是脚本化的。组态软件是这方面的一个例子，在 PC 上组态的工程，也可以在嵌入式设备上运行。

4.3. 一个框架设计的实例

4.3.1. 基本架构

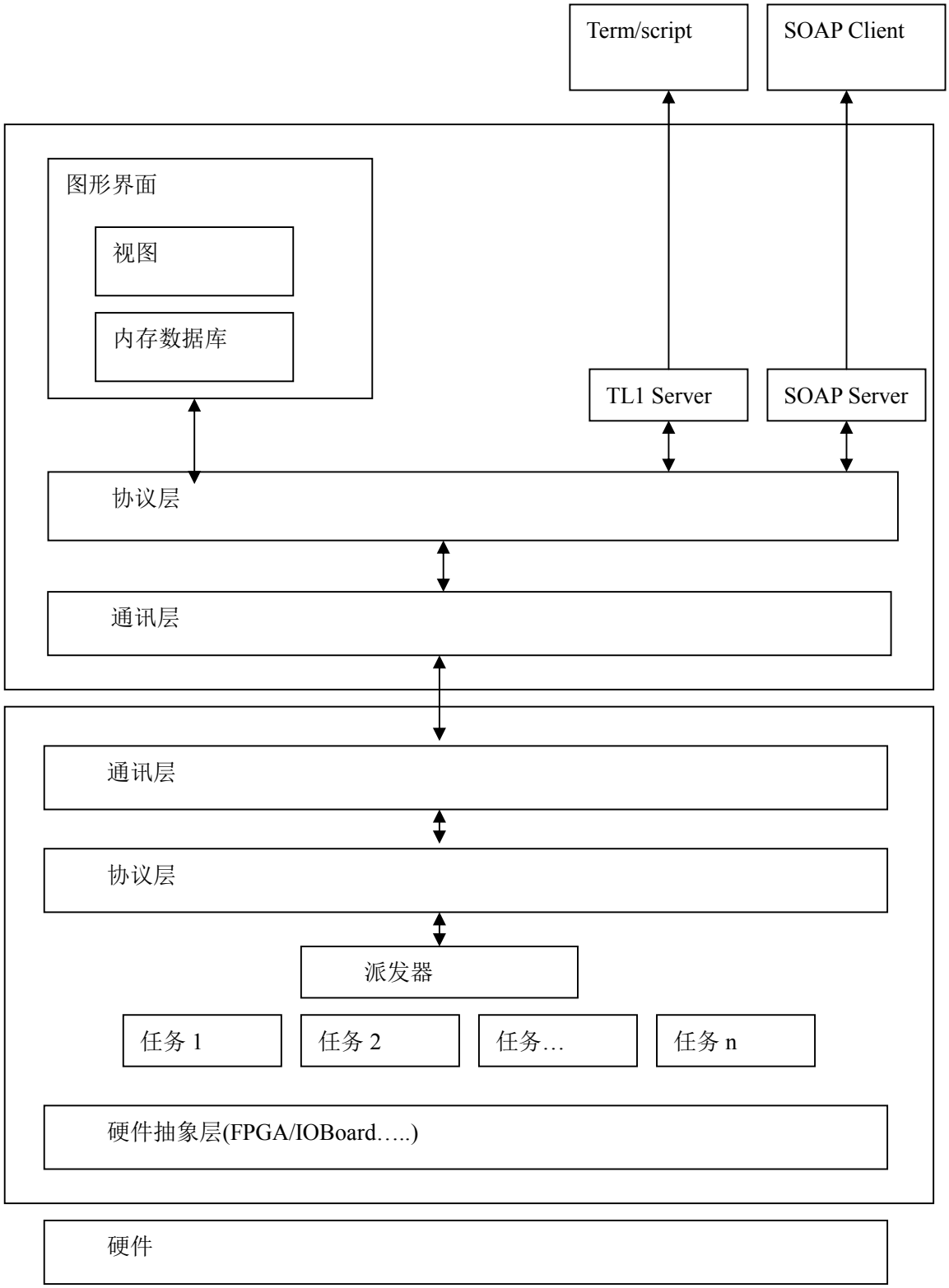


4.3.2. 功能特点

上面是一个产品系列的架构图，其特点是硬件部分是模块化的，可以随时插拔。不同的硬件应用于不同的通讯测试场合。比如光通讯测试，xDSL 测试，Cable Modem 测试等等。针对不同的硬件，需要开发不同的固件和软件。固件层的功能主要是通过 USB 接口接收来自软件的指令，并读写相应的硬件接口，再进行一些计算后，将结果返回给软件。软件运行在 WinCE 平台，除了提供一个触摸式的图形化界面外，还对外提供基于 XML(SOAP)接口和 TL1 接口。为了实现自动化测试，还提供了基于 Lua 的脚本语言接口。整个产品系列有几十个不同的硬件模块，相应的需要开发几十套软件。这些软件虽然服务于不同的硬件，但是彼此之间有着高度的相似性。所以，选择先开发一个框架，再基于框架开发具体的模块软件成了最优的选择。

4.3.3. 分析

软件部分的结构分析如下：


Term/script
SOAP Client
图形界面
视图
内存数据库
TL1 Server
SOAP Server
协议层
通讯层
通讯层
协议层
派发器
任务 1
任务 2
任务…
任务 n
硬件抽象层(FPGA/IOBoard…..)
硬件

系统分为软件，固件和硬件三大块。软件和固件运行在两块独立的板子上，有各自的处理器和操作系统。硬件则插在固件所在的板子上，是可以替换的。

软件和固件其实都是软件，下面我们分别分析。

软件

软件的主要工作是提供各种用户界面。包括本地图形化界面，SOAP 访问界面，TL1 访问界面。

整个软件部分分为五大部分：

通讯层

协议层

图形界面

SOAP 服务器

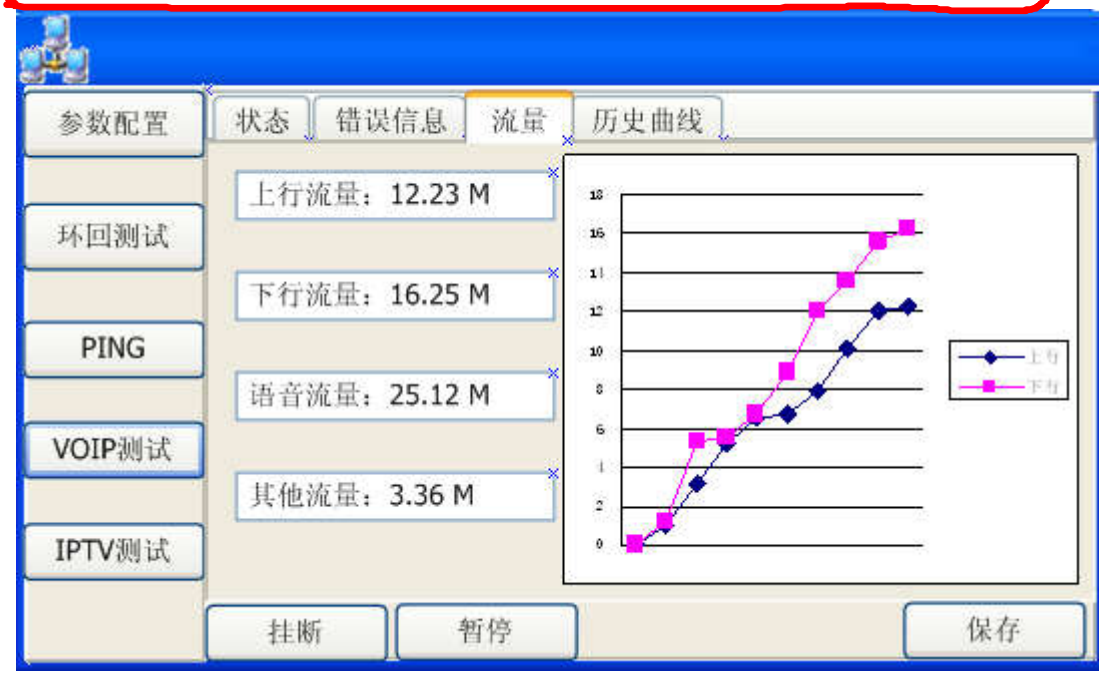
TL1 服务器

通讯层要屏蔽用户对具体通信介质和协议的了解，无论是 USB 还是 socket，对上层都不产生影响。通讯层负责提供可靠的通讯服务和适当的错误处理。通过配置文件,用户可以改变所使用的通讯层。

协议层的目的是将数据进行编码和解码。编码的产生物是可以在通讯层发送的流，按照嵌入式软件的特点，我们选择二进制作为流的格式。解码的产生物是多种的，既有供界面使用的 C Struct，也可以是 XML 数据，还可以是 Lua 的数据结构(tablegt)。如果需要，还可以产生 JSON,TL1,Python 数据,TCL 数据等等。这一层在框架中是通过机器自动生成的，我们后面会讲到。

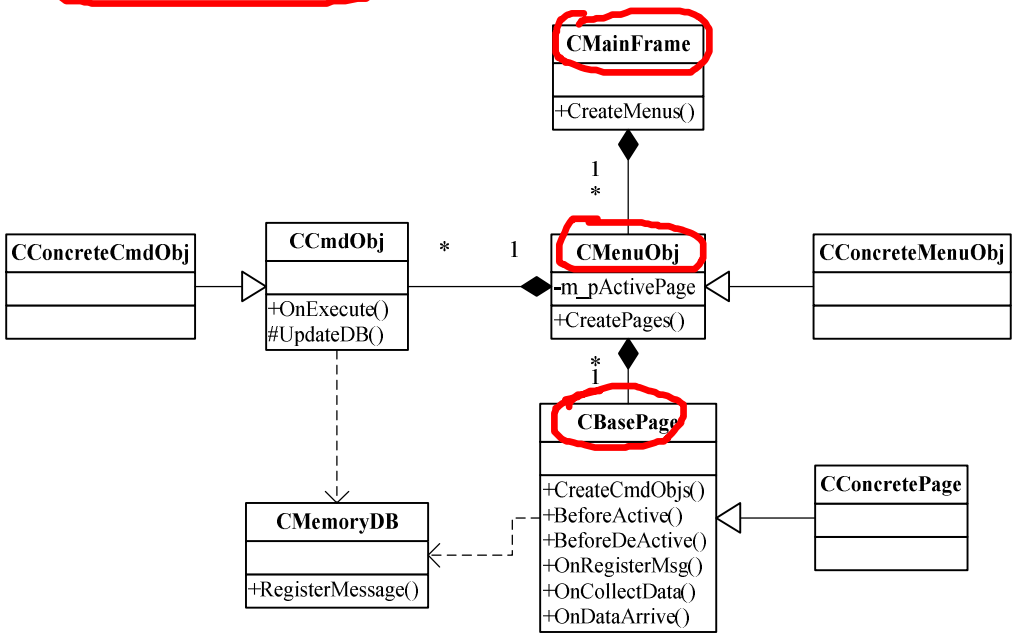
内存数据库，SOAP Server 和 TL1 Server 都是协议层的用户。图形界面通过读写内存数据库和底层通讯。

图形界面是框架设计的重点之一，原因是这里工作量最大，重复而无聊的工作最多。

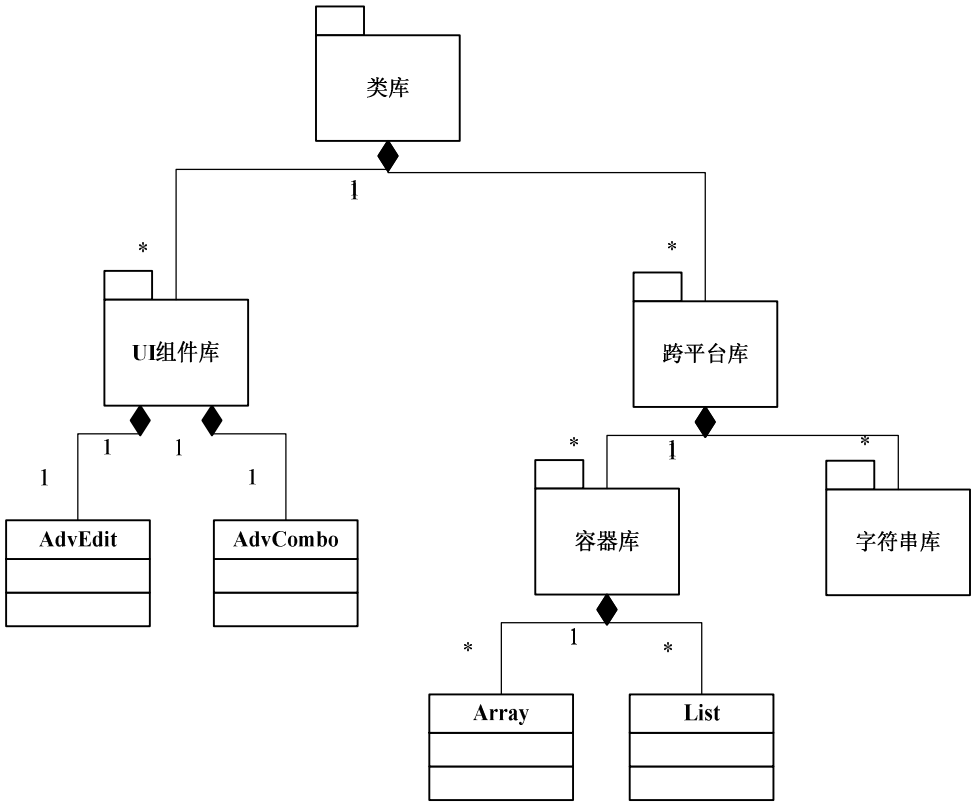


让我们分析一下在图形界面开发工作中最主要的事情是什么。

1. 收集用户输入的数据和命令
2. 将数据和命令发给底层
3. 接收底层反馈
4. 将数据显示在界面上



同时有一些库用来进一步简化开发：



这是一个简化的例子，但是很好的说明了框架的特点：

1. 客户代码必须按照规定的接口实现
2. 框架在适当的时候调用客户实现的接口
3. 每个接口都被设计为只完成特定的单一功能
4. 将各个步骤有机的串起来是框架的事，二次开发者不知道，也无须知道。
5. 通常都要有附带的库。

固件

固件的主要工作是接受来自软件的命令，驱动硬件工作；获取硬件的状态，进行一定的计算后返回给软件。早期的固件是很薄的一层，因为绝大部分工作是由硬件完成的，固件只起到一个中转通讯的作用。随着时代发展，现在的固件开始承担越来越多原来由硬件完成的工作。

整个固件部分分为五大部分：

硬件抽象层，提供对硬件的访问接口

互相独立的任务群

任务/消息派发器

协议层

通讯层

针对不同的设备，工作量集中在硬件抽象层和任务群上。硬件抽象层是以库的形式提供的，由对硬件最熟悉，经验最丰富的工程师来实现。任务群则由一系列的任务组成，他们分别代表不同的业务应用。比如测量误码率。这部分由相对经验较少的工程师来实现，他们的主要工作是实现规定的接口，按照标准化文档定义的方式实现算法。

任务定义了如下接口，由具体开发者来实现：

```
OnInit();
OnRegisterMessage();
OnMessageArrive();
Run();
OnResultReport();
```

框架的代码流程如下：（伪代码）

```
CTask* task = new CBertTask();
task->OnInit();
task->OnRegisterMessage();
while(TRUE)
{
  task->OnMessageArrive();
  task->Run();
  task->OnResultReport();
}
delete task;
task = NULL;
```

这样，具体任务的实现者所关注的最重要的事情就是实现这几个接口。其他如硬件的初

始化，消息的收发，编码解码，结果的上报等等事情都由框架进行了处理。避免了每个工程师都必须处理从上到下的所有方面。并且这样的任务代码还有很高的重用性,比如是在以太网上还是在 Cable Modem 上实现 PING 的算法都是一样的。

4.3.4. 实际效果

在实际项目中，框架大大降低了开发难度。对软件部分尤其明显，由实习生即可完成高质量的界面开发，开发周期缩短 50%以上。产品质量大大提升。对固件部分的贡献在于降低了对精通底层硬件的工程师的需要，一般的工程师熟知测量算法即可。同时，框架的存在保证了性能，稳定和可测试性等要素。

4.4. 框架设计中的常用模式

4.4.1. 模板方法模式

模板方法模式是框架中最常用的设计模式。其根本的思路是将算法由框架固定，而将算法中具体的操作交给二次开发者实现。例如一个设备初始化的逻辑，框架代码如下：

```
TBool CBaseDevice::Init()
{
        if ( DownloadFPGA() != KErrNone )
        {
            LOG(LOG_ERROR,_L("Download FPGA fail"));
            return EFalse;
        }
        if ( InitKeyPad() != KerrNone )
        {
             LOG(LOG_ERROR,_L("Initialize keypad fail"));
             return EFalse;
         }
        return ETrue;
}
```

DownloadFPGA 和 InitKeyPad 都是 CBaseDevice 定义的虚函数，二次开发者创建一个继承于 CBaseDevice 的子类，具体来实现这两个接口。框架定义了调用的次序和错误的处理方式，二次开发者无须关心，也无权决定。

4.4.2. 创建型模式

由于框架通常都涉及到各种不同子类对象的创建，创建型模式是经常使用的。例如一个绘图软件的框架，有一个基类定义了图形对象的接口，基于它可以派生出椭圆，矩形，直线各种子类。当用户绘制一个图形时，框架就要实例化该子类。这时候可以用工厂方法，原型

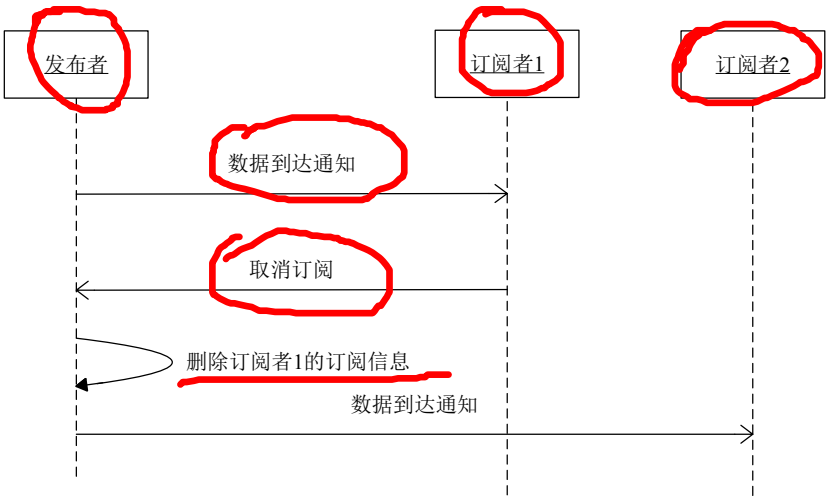
方法等等。

```
class CDrawObj
{
      public:
            virtual int DrawObjTypeID()=0;
            virtual Icon GetToolBarIcon()=0;
            virtual void Draw(Rect rect)=0;
            virtual CDrawObj* Clone()=0;
};
```

4.4.3. 消息订阅模式

消息订阅模式是最常用的分离数据和界面的方式。界面开发者只需要注册需要的数据，当数据变化时框架就会将数据“推”到界面。界面开发者可以无须关注数据的来源和内部组织形式。

消息订阅模式最常见的问题是同步模式下如何处理重入和超时。作为框架设计者，一定要考虑好这个问题。所谓重入，是二次开发者在消息的回调函数中执行订阅/取消订阅的操作，这会破坏消息订阅的机制。所谓超时是指二次开发者的消息回调函数处理时间过长，导致其他消息无法响应。最简单的办法是使用异步模式，让订阅者和数据发布者在独立进程/线程中运行。如果不具备此条件，则必须作为框架的重要约定，禁止二次开发者产生此类问题。

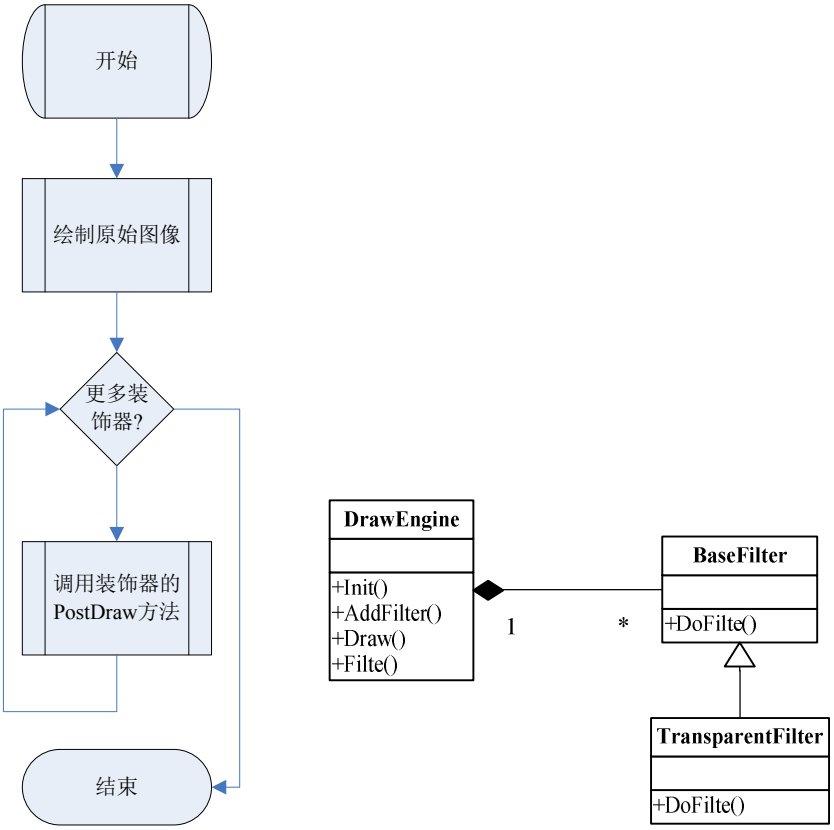


4.4.4. 装饰器模式

装饰器模式赋予了框架在后期增加功能的能力。框架定义装饰器的抽象基类，而由具体的实现者实现，动态地添加到框架中。

举一个游戏中的例子，图形绘制引擎是一个独立的模块，比如可以绘制人物的静止，跑动等图像。如果策划决定在游戏中增加一种叫“隐身衣”的道具，要求穿着此道具的玩家在

屏幕上显示的是若有若无的半透明图像。应该如何设计图像引擎来适应后期的游戏升级呢？



当隐身衣被装备后，就向图像引擎添加一个过滤器。这是个极度简化的例子，实际的游戏引擎要比这个复杂。装饰器模式还常见用于数据的前置和后置处理上。

4.5. 框架的缺点

一个好的框架可以大大提高产品的开发效率和质量，但也有它的缺点。

1. 框架一般都比较复杂，设计和实现一个好的框架需要相当的时间。所以，一般只有在框架可以被多次反复应用的时候适合，这时候，前提投入的成本会得到丰厚的回报。
2. 框架规定了一系列的接口和规则，这虽然简化了二次开发工作，但同时也要求二次开发者必须记住很多规定，如果违反了这些规定，就不能正常工作。但是由于框架屏蔽了大量的领域细节，相对而言，其学习成本还是大大降低了的。
3. 框架的升级对已有产品可能会造成严重的影响，导致需要完整的回归测试。对这个问题有两个办法。第一是对框架本身进行严格的测试，有必要建立完善的单元测试库，同时开发示例项目，用来测试框架的所有功能。第二则是使用静态链接，让已有产品不轻易跟随升级。当然，如果已有产品有较好的回归测试手段，就更好。
4. 性能损失。由于框架对系统进行了抽象，增加了系统的复杂性。诸如多态这样的手段使用也会普遍的降低系统的性能。但是从整体上来看，框架可以保证系统的性能处于一个较高的水平。

5. 自动代码生成

5.1. 机器能做的事就不要让人来做

懒惰是程序员的美德，更是架构师的美德。软件开发的过程就是人告诉机器如何做事的过程。如果一件事情机器自己就可以做，那就不要让人来做。因为机器不仅不知疲倦，而且绝不会犯错。我们的工作是让客户的工作自动化，多想一点，就能让我们自己的工作也部分自动化。极有耐心的程序员是好的，也是不好的。

经过良好设计的系统，往往会出现很多高度类似而且具有很强规律的代码。未经良好设计的系统则可能对同一类功能产生很多不同的实现。前面关于框架设计的部分已经证明了这一点。有时候，我们更进一步，分析出这些相似代码之中的规律，用格式化的数据来描述这些功能，而由机器来产生代码。

5.2. 举例

5.2.1. 消息的编码和解码

上面关于框架的实例中，可以看到消息编解码的部分已经被独立出来，和其他部分没有耦合。加上他本身的特点，非常适合进一步将其“规则化”，用机器产生代码。

编码，就是把数据结构流化；解码反之。以编码为例，代码无非是这样的：(二进制协议)

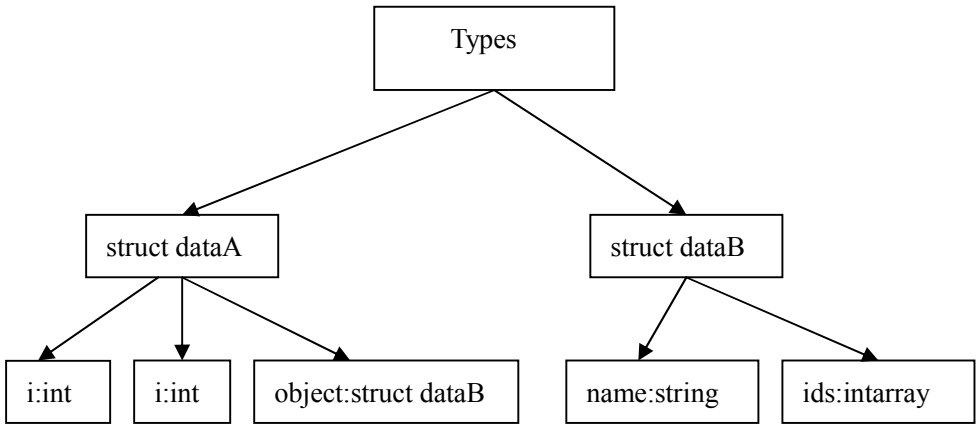
```
stream << a.i;
stream << a.j;
stream << a.object;
```

(为了简化，这里假设已经设计了一个流对象，可以流化各种数据类型，并且已经处理了诸如字节序转换等问题。)

最后我们得到一个 stream。大家是否已经习惯了写这种代码？但是这样的代码不能体现工程师任何的创造性，因为我们早已经知道有 i, 有 j, 还有一个 object，为什么还要自己敲入这些代码呢？如果我们分析一下 a 的定义，是不是就可以自动产生这样的代码呢？

```
struct dataA
{
  int i;
  int j;
  struct dataB object;
};
```

只需要一个简单的语义分析器解析这段代码，得到一棵关于数据类型的树，就可以轻易的产生流化的代码。这样的分析器用 Python 等字符串处理能力强的语言不过两百行左右。关于数据类型的树类似下图：



只要遍历这棵树，就可以生成所有数据结构的流化代码。

在上一个框架所举例的项目中，为一个硬件模块自动产生的消息编码解码器代码量高达三万行，几乎相当于一个小软件。由于是自动产生，没有任何错误，为上层提供了高可靠性。

还可以用 XML 或者其他的格式定义数据结构，从而产生自动代码。根据需要，C++/Java/Python，任何类型的都可以。如果希望提供强检查，可以使用 XSD 来定义数据结构。有一个商业化的产品，xBinder，很贵，很难用，还不如自己开发。(为什么难用？因为它太通用)。除了编码为二进制格式，还可以编码为任何你需要的格式。我们知道二进制格式虽然效率很高，但是太难调试（当然有些人看内存里的十六进制还是很快的），所以我们可以在编码成二进制的同时，还生成编码为其他可阅读的格式的代码，比如 XML。这样，通讯使用二进制，而调试使用 XML，两全其美。产生二进制的代码大概是这样的：

```
Xmlbuilder.addelement(“i”,a.i);
Xmlbuilder.addelement(“j”,a.j);
Xmlbuilder.addelement(“object”,a.object);
```

同样也很适合机器产生。同样的思路可以用来让软件内嵌脚本支持。这里不多说了。(内嵌脚本支持最大的问题是在 C/C++和脚本之间交换数据，也是针对数据类型的大量相似代码。)

最近 Google 发布了它的 protocol buffer，就是这样的思路。目前支持 C++/Python，估计很快会支持更多的语言，大家可以关注。以后就不要再手写编码解码器了。

5.2.2. GUI 代码

上面的框架设计部分，我们说到框架对界面数据收集和界面更新无能为力，只能抽象出

接口，由程序员具体实现。但是让我们看看这些界面程序员做的事情吧。(代码经过简化，可以看作伪代码)。

```
void onDataArrive(CDataBinder& data)
{
  m_biterror.setText(“%d”,data.biterror);
  m_signallevel.setText(“%d”,data.signallevel”);
  m_latency.setText(“%d”,data.latency”);
}

Void onCollectData(CDataBinder& data)
{
  data.biterror = atoi(m_biterror.getText());
  data. signallevel = atoi(m_ signallevel.getText());
  data. latency = atoi(m_ latency.getText());
}
```

这样的代码很有趣吗？想想我们可以怎么做？（XML 描述界面，问题是对于复杂逻辑很难）

5.2.3. 小结

由此可见，在软件架构的过程中，首先要遵循一般性的原则，尽量将系统各个功能部分独立出来，实现高内聚低耦合，进而发现系统存在的高度重复，规律性很强的代码，进一步将他们规则化，形式化，最后用机器来产生这些代码。目前这方面最成功的应用就是消息的编解码。对界面代码的自动化生成有一定局限，但也可以应用。大家在自己的工作中要擅于发现这样的可能，减少工作量，提高工作效率。

5.2.4. Google Protocol Buffer

Google 刚刚发布的 Protocol Buffer 是使用代码自动生成的一个典范。

Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use **special generated source** code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the "old" format.

你要做的首先是定义消息的格式，Google 指定了它的格式：

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Once you've defined your messages, you run the protocol buffer compiler for your application's language on your `.proto` file to generate data access classes. These provide simple accessors for each field (like `query()` and `set_query()`) as well as methods to serialize/parse the whole structure to/from raw bytes – so, for instance, if your chosen language is C++, running the compiler on the above example will generate a class called `Person`. You can then use this class in your application to populate, serialize, and retrieve `Person` protocol buffer messages. You might then write some code like this:

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

Then, later on, you could read your message back in:

```
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

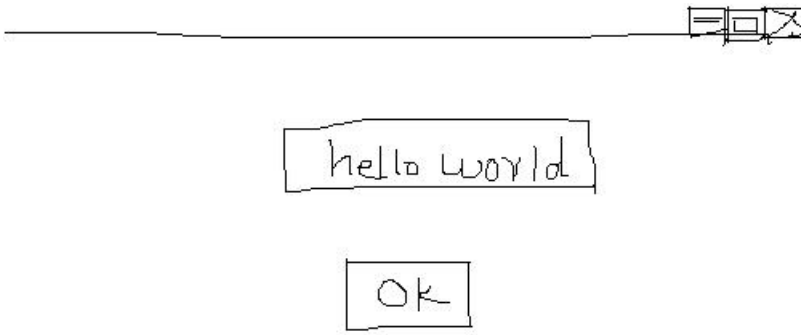
Protocol Buffer 的编码格式是二进制的，同时也提供可读的文本格式。效率高，体积小，上下兼容。目前支持 Java Python 和 C++，很快会支持更多的语言。

6. 面向语言编程(LOP)

6.1. 从自动化代码生成更进一步

面向语言编程的通俗定义是：将特定领域的知识融合到一种专用的计算机语言当中，从而提高人与计算机交流的效率。

自动化代码生成其实就是面向语言编程。语言不等于是编程语言，可以是图，也可以是表，任何可以建立人和机器之间交流渠道的都是计算机语言。软件开发历史上的一次生产率的飞跃是高级语言的发明。它让我们以更简洁的方式实现更复杂的功能。但是高级语言也有它的缺点，那就是从问题领域到程序指令的过程很复杂。因为高级语言是为通用目的而设计的，所以离问题领域很远。举例来说，要做一个图形界面，我可以跟另一个工程师说：这里放一个按钮，那边放一个输入框，当按下按钮的时候，就在输入框里显示 Hello World。我甚至可以随手给他画出来。



对于我和他直接的交流而言，这已经足够了，5 分钟。但是要让转变为计算机能够理解的语言，需要多久？

如果是汇编语言？(告诉计算机如何操作寄存器和内存)

如果是 C++? (告诉计算机如何在屏幕上绘图，如果响应鼠标键盘消息)

如果有一个不错的图形界面库？(告诉计算机创建 Button,Label 对象，管理这些对象，放置这些对象，处理消息)

如果有一个不错的开发框架+IDE? (用 WYSIWYG 工具绘制，设计类，类的成员变量，编写消息响应函数)

如果有一门专门做图形界面开发的语言？

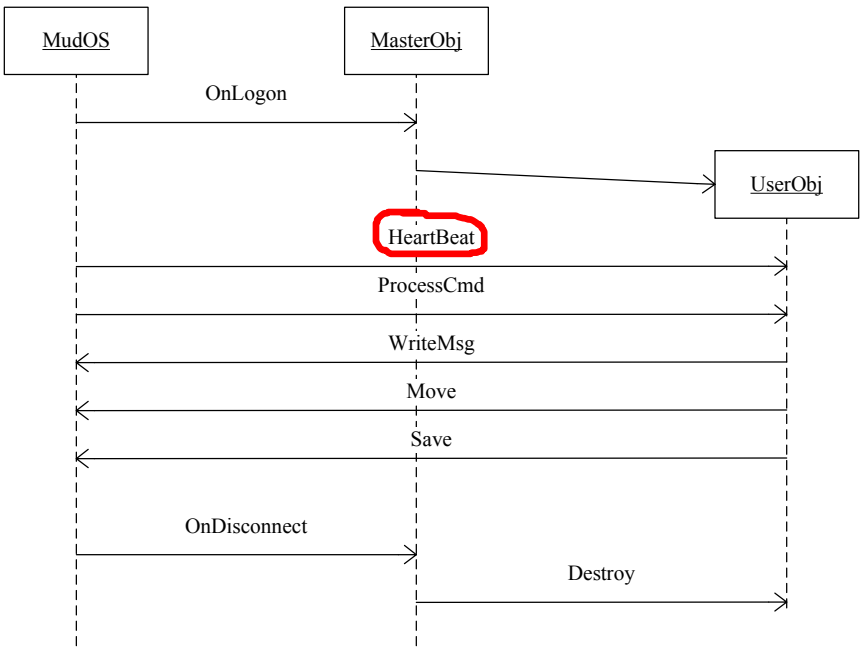
可以是这样的：

```
Label l {Text=””}
Button b{Text=”ok”,action=l.Text=”hello world”}
```

通用的计算机语言是基于变量，类，分支，循环，链表，消息这些概念的。这些概念离问题本身有着遥远的距离，而且表达能力非常有限。自然语言表达能力很强，但是歧义和冗余太多，无法格式化标准化。传统的思想告诉我们：计算机语言就是一条条的指令，编程就是写下这些指令。而面向语言编程的思想是，用尽量贴近问题，贴近人的思维的办法来描述

问题，从而降低从人的思想到计算机软件转换的难度。

举一个游戏开发的例子。现在的网络游戏普遍的采用了 C++或者 C 开发游戏引擎。而具体的游戏内容，则是由一系列二次开发工具和语言完成的。地图编辑器就是一种面向游戏的语言。Lua 或者类似的脚本则被嵌入到游戏内部，用来编写武器，技能，任务等等。Lua 本身不具备独立开发应用程序的能力，然而游戏引擎的设计者通过给 Lua 提供一系列的，各种层次上的接口，将领域知识密集的赋予了脚本，从而大大提高了游戏二次开发的效率。网络游戏的鼻祖 MUD 则是设计了 LPC 来作为游戏的开发语言。MUD 的引擎 MudOS 和 LPC 之间的关系如图：



用 LPC 创建一个 NPC 的代码类似如下：

```
inherit NPC;
void create()
{
        set_name("菜花蛇", ({ "caihua she", "she" }) );
        set("race", "野兽");
        set("age", 1);
        set("long", "一只青幽幽的菜花蛇，头部呈椭圆形。\n");
        set("attitude", "peaceful");
        set("str", 15);
        set("cor", 16);
        set("limbs", ({ "头部", "身体", "七寸", "尾巴" }) );
        set("verbs", ({ "bite" }) );
        set("combat_exp", 100+random(50));
        set_temp("apply/attack", 7);
```

```
        set_temp("apply/damage", 4);
        set_temp("apply/defence",6);
        set_temp("apply/armor",5);
        setup();
}
void die()
{
        object ob;
        message_vision("$N 抽搐两下，$N 死了。\n", this_object());
        ob = new(__DIR__"obj/sherou");
        ob->move(environment(this_object()));
        destruct(this_object());
}
```

LPC 培养了一大批业余游戏开发者，甚至成为很多人进入 IT 行业的起点。原因就是它简单，易理解，100%为游戏开发设计。这就是 LOP 的魅力。

6.2. 优势和劣势

LOP 最重要的优点是将领域知识固化到语言中，从而：

1. 提高开发效率。
2. 优化团队结构，降低交流成本，领域专家和程序员可以更好的合作。
3. 降低耦合，易于维护。

其次，由于 LOP 不是通用语言，所涉及的范围就狭窄很多，所以：

4. 更容易得到稳定的系统
5. 更容易移植

相应的，LOP 也有它的劣势：

1. LOP 对领域知识抽象的要求比框架更高。
2. 开发一门新的语言本身的成本。幸好现在设计一门新的语言不算太难，还有 Lua 这样的“专用二次开发”语言的支持。
3. 性能损失。不过相比开发成本的节约，在非性能核心部分使用 LOP 还是很值得的。

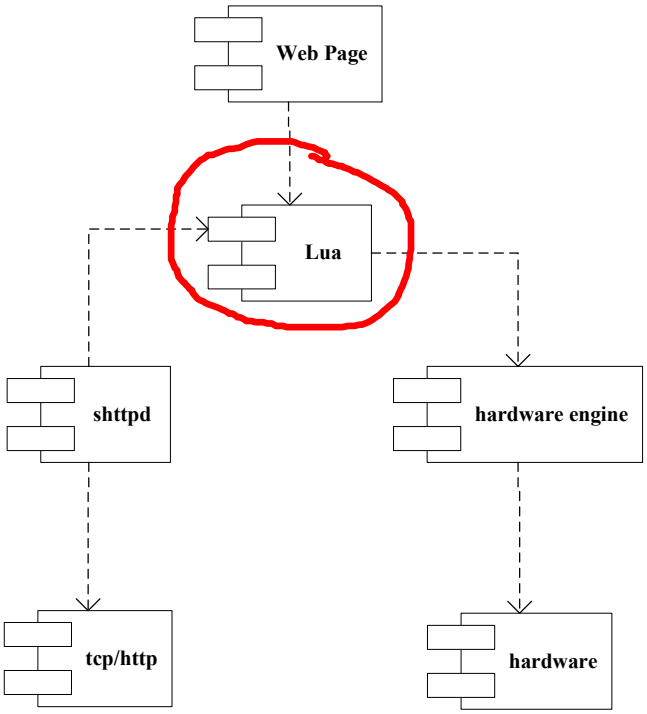
6.3. 在嵌入式系统中的应用

举例，嵌入式设备的 Web 服务器。很多设备都提供 Web 服务用于配置，比如路由器，ADSL 猫等等。这种设备所提供的 web 服务的典型用例是用户填写一些参数，提交给 Web 服务器，Web 服务器将这些参数写入硬件，并将操作结果或者其他信息生成页面返回给浏览器。由于典型的 Apache,Mysql,PHP 组合体积太大且不容易移植，通常嵌入式系统的 Web 服务都是用 C/C++直接写就的。从 socket 管理，http 协议到具体操作硬件，生成页面，都一体负责。然而对于功能复杂，Web 界面要求较高的情况，用 C 来写页面效率就太低了。

shttpd 是一个小巧的 web 服务器，小巧到只有一个.c 文件，4000 余行代码。虽然体积

很小，却具备了最基本的功能，比如 CGI。它既可以独立运行，也可以嵌入到其他的应用程序当中。shttpd 在大多数平台上都可以顺利编译、运行。lua 是一个小巧的脚本语言，专用于嵌入和扩展。它和 C/C++代码有着良好的交互能力。

将 Lua 引擎嵌入到 shttpd 中，再使用 C 编写一个（一些）驱动硬件的扩展，注册成为 Lua 的函数，形成的系统结构如下图：



这样的应用在嵌入式系统中是有一定代表性的，即，以 C 实现底层核心功能，而把系统的易变部分以脚本实现。大家可以思考在自己的开发过程中是否可以使用这种技术。这是 LOP 的一种具体应用模式。（没有创造一种全新的语言，而是使用 Lua）

7. 测试

7.1. 可测试性是软件质量的一个度量指标

好的软件是设计出来的，好的软件也一定是便于测试的。一个难于测试的软件的质量是难以得到保障的。在今天软件规模越来越大的趋势下，以下问题是普遍存在的：

1. 测试只能手工进行，回归测试代价极大，实际只能执行点测，质量无法保证
2. 各个模块只有集成到一起后才能测试
3. 代码不经过任何单元测试就集成

这些问题的根源都在于缺乏一个良好的软件设计。一个好的软件设计应该使得单元测试，模块测试和回归测试都变得容易，从而保证测试的广度和深度，最终产生高质量的软件。除了功能，非功能性需求也必须是可测试的。所以，可测试性是软件设计中一个重要的指标，是系统架构师需要认真考虑的问题。

7.2. 测试驱动的软件架构

这里谈的是测试驱动的软件架构，而不是测试驱动的开发。TDD(Test Driven Development) 是一种开发方式，是一种编码实践。而测试驱动的架构强调的是，从提高可测试性的角度进行架构设计。软件的测试分为多个层次：

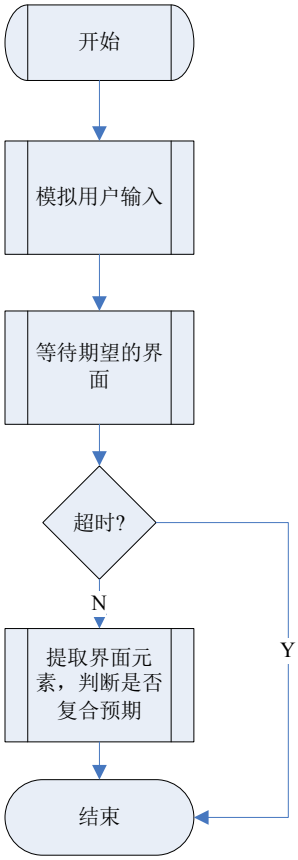
7.3. 系统测试

系统测试是指由测试人员执行的，验证软件是否完整正确的实现了需求的测试。这种测试中，测试人员作为用户的角色，通过程序界面进行测试。在大部分情况下这些工作是手工完成的。在规范的流程中，这个过程通常要占到整个软件开发时间的 1/3 以上。而当有新版本发布的时候，尽管只涉及了软件的一部分，测试部门依然需要完整的测试整个软件。这是由代码“副作用”特点决定的。有时候修改一个 bug 可以引发更多的 bug，破坏原来工作正常的代码。这在测试中叫回归测试(Regression test)。对于规模较大的软件，回归测试需要很长的时间，在版本新增功能和错误修正不多的情况下，回归测试可以占到整个软件开发过程了一半以上，严重影响了软件的交付，也使软件测试部门成为软件开发流程中的瓶颈。测试过程自动化，是部分解决这个问题的办法。

作为架构师，有必要考虑如何实现软件的可自动化测试性。

7.3.1. 界面自动化测试

在没有图形化界面以前，字符方式的界面是比较容易进行自动化测试的。一个编写良好的脚本就可以实现输入和对输出的检查。但是对于图形化的界面，人的参与似乎变得不可缺少。有一些界面自动化的测试工具，如 WinRunner， 这些工具可以记录下测试人员的操作成为脚本，然后通过回放这些脚本，就可以实现操作的自动化。针对嵌入式设备，有 Test Quest 可以使用，通过在设备中运行一个类似远程桌面的 Agent，PC 端的测试工具可以用图像识别的方法识别出不同的组件，并发送相应用户的输入。此类工具的基本工作原理如图：



但是这个过程在实际中存在三个问题：

1. 可靠性差，经常中断运行。要写一个可靠的脚本甚至比开发软件还要困难。比如，按下一个按钮，有时候一个对话框立刻就出现，有时候可能要好几秒，有时候甚至不出现，操作录制工具不能自动实现这些判断，而需要手动修改。
2. 对操作结果的判断很困难，尤其是非标准的控件。
3. 当界面修改后，原有代码很容易失效

要应用基于图形界面的自动化测试工具，架构师在架构的时候应该考虑：

1. 界面风格如何保持一致。应当由架构，而非程序员决定架构的风格。包括布局，控件大小，相对位置，文字，对操作的响应方式，超时时长，等等。
2. 如何在最合适测试工具的界面和用户喜欢的界面之中折中。比如，Test Quest 是基于图像识别的，那么黑白两色的界面是最有利的，而用户喜欢的渐进色就非常不利。也许让界面具备自动的切换能力最好。

对于已经完成的产品，如果架构没有为自动化测试做过考虑，所能应用的范围就非常有限，不过还是有一些思路可以供参考：

1. 实现小规模的自动化脚本。针对一个具体的操作流程进行测试，而不是试图用一个脚本测试整个软件。一系列的小测试脚本组成了一个集合，覆盖系统的一部分功能。这些测试脚本可以都以软件启动时的状态作为基准，所以在状态处理上会比较简单
2. ”猴子测试”有一定的价值。所谓猴子测试，就是随机操作鼠标和键盘。这种测试完全不理解软件的功能，可以发现一些正常测试无法发现的错误。据微软内部的资料，微软的一些产品 15%的错误是由“猴子测试”发现的。

总的来讲，基于界面的自动化测试是不成熟的。对架构师而言一定要避免功能只能通过

界面才能访问。要让界面仅仅是界面，而软件大部分的功能是独立于界面并可以通过其他方式访问的。上面框架的例子中的设计就体现了这一点。

思考：如何让界面具有自我测试功能？

7.3.2. 基于消息的自动化测试

如果软件对外提供基于消息的接口，自动化测试就会变得简单的多。上面已经提到了固件的 TL1 接口。对于界面部分，则应该在设计的时候，将纯粹的“界面”独立出来，让它尽可能的薄，而其他部分依然可以基于消息提供服务。

在消息的基础上，用脚本语言包装成函数的形式，可以很容易的调用，并覆盖消息的各种参数组合，从而提高测试的覆盖率。关于如何将消息包装为脚本，可以参考 SOAP 的实现。如果使用的不是 XML，也可以自己实现类似的自动代码生成。

这些测试脚本应该由开发人员撰写，每当实现了一个新的接口（也就是一条新的消息），就应该撰写相应的测试脚本，并作为项目的一部分保存在代码库中。当需要执行回归测试的时候，只要运行一遍测试脚本即可，大大提高了回归测试的效率。

所以，为了实现软件的自动化测试，提供基于消息的接口是一个很好的办法，这让我们可以在软件之外独立的编写测试脚本。在设计的时候可以考虑这个因素，适当的增加软件消息的支持。当然，TL1 只是一个例子，根据项目的需要，可以选择任何适合的协议，如 SOAP。

7.3.3. 自动化测试框架

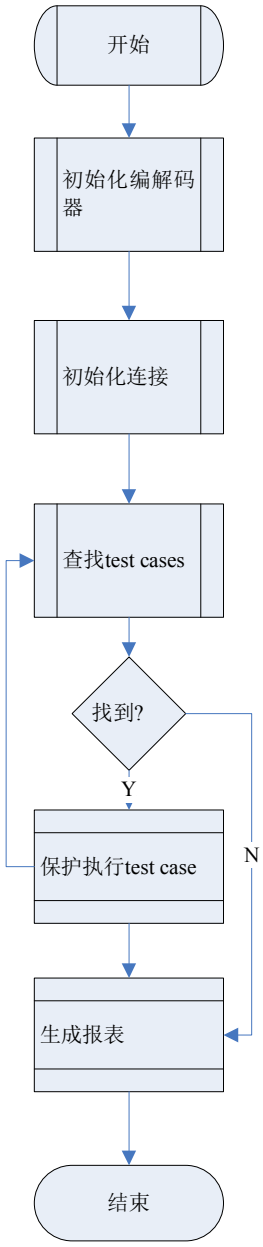
在编写自动化测试脚本的时候，有很多的工作是重复的，比如建立 socket 连接，日志，错误处理，报表生成等。同时，对于测试人员来说，这些工作可能是比较困难的。因此，设计一个框架，实现并隐藏这些重复和复杂的技术，让测试脚本的编写者将注意力集中在具体的测试逻辑上。

这样一个框架应该实现以下功能：

1. 完成连接的初始化等基础工作。
2. 捕获所有的错误，保证 Test Case 中的错误不会打断后续的 Test Case 执行。
3. 自动检测和执行 Test Case。新增的 Test Case 是独立的脚本文件，无须修改框架的代码或者配置。
4. 消息编解码，并以函数的方式提供给 Test Case 编写者调用。
5. 方便的工具，如报表，日志等。
6. 自动统计 Test Case 的运行结果并生成报告。

自动化测试框架的思路和一般的软件框架是一致的，就是避免重复劳动，降低开发难度。

下图是一个自动化测试框架的结构图：



每个 Test Case 都必须定义一个规定的 Run 函数，框架将依次调用，并提供相应的库函数供 Test Case 用来发送命令和获得结果。这样，测试用例的编写者就只需要将注意力集中在测试本身。举例：

```
def run():
    open_laser()
    assert(get_laser_state() == ON)
    insert_error(BIT_ERROR)
    assert(get_error_bit() == BIT_ERROR)
```

测试用例的编写者拥有的知识是“必须先打开激光器然后才能向线路上插入错误”。而架构师能提供的是消息收发，编解码，错误处理，报表生成等，并将这些为测试用例编写者隔离。

问题: open_laser, get_laser_state 这些函数是谁写的？

问题：如何进一步实现知识的解耦？能否有更方便的语言来编写 TestCase?

7.3.4. 回归测试

有了自动化的测试脚本和框架，回归测试就变得很简单了。每当有新版本发布时，只需运行一遍现有的 Test Case，分析测试报告，如果有测试失败的 Case 则回归测试失败，需要重新修改，直到所有的 Case 完全通过。完整的回归测试是软件质量的重要保证。

7.4. 集成测试

集成测试要验证的是系统各个组成模块的接口是否工作正常。这是比系统测试更低层的测试，通常由开发人员和测试人员共同完成。

例如在一个典型的嵌入式系统中，FPGA，固件和界面是常见的三个模块。模块本身还可以划分为更小的模块，从而降低复杂度。嵌入式软件模块测试的常见问题是硬件没有固件则无法工作，固件没有界面就无法驱动;反过来，界面没有固件不能完整运行，固件没有硬件甚至无法运行。于是没有经过测试的模块直到集成的时候才能完整运行，发现问题后需要考虑所有模块的问题，定位和解决的代价都很大。假设有模块 A 和 B，各有十个 bug。如果都没有经过模块测试直接集成，可以认为排错的工作量相当于 10*10 等于 100。

所以，在设计一个模块的时候，首先要考虑，这个模块如何单独测试？比如，如果界面和固件之间是通过 SOCKET 通信的，那么就可以开发一个模拟固件，在同样的端口上提供服务。这个模拟固件不执行实际的操作，但是会响应界面的请求并返回模拟的结果。并且返回的结果可以覆盖到各种典型的情况，包括错误的情况。使用这样的技术，界面部分几乎可以得到 100%的验证，在集成阶段遇到错误的大大减少。

对固件而言，因为处于系统的中间，所以问题复杂一些。一方面，要让固件可以通过 GUI 以外的途径被调用；另一方面则要模拟硬件的功能。对于第一点，在设计的时候，要让接口和实现分离。接口可以随意的更换，比如和 GUI 的接口也许是 JSON，同时还可以提供 telnet 的 TL1 接口，但是实现是完全一样的。这样，在和 GUI 集成之前，就可以通过 TL1 进行完全的测试固件。对于第二点，则应该在设计的时候提取出硬件抽象层，让固件的主要实现和寄存器，内存地址等因素隔离开来。在没有硬件或者硬件设计未定的时候实现一个硬件模拟层，来保证固件可以完整运行并测试。

7.5. 单元测试

单元测试是软件测试的最基本单位，是由开发人员执行以保证其所开发代码正确的过程。开发人员应该提交经过测试的代码。未经单元测试的代码在进入软件后，不仅发现问题后很难定位，而且通过系统测试是很难做到对代码分支的完全覆盖的。TDD 就是基于这个层次的开发模式。

单元测试的粒度一般是函数或者类，例如下面这个常用函数：

```
int atoi(const char *nptr);
```

这是一个功能非常单一的函数，所以单元测试对它非常有效。可以通过单元测试验证下列情况：

1. 一般正常调用,如”9”,”1000”,”-1”等

2. 空的 nptr 指针
3. 非数字字符串,”abc”,”@#!123”,”123abc”
4. 带小数点的字符串, “1.1”,”0.111”,”.123”
5. 超长字符串
6. 超大数字，”99999999999999999999999999”
7. 超过一个的-号和位置错误的-号,”—1”,”-1-“,”-1-2”

如果 atoi 通过了以上测试，我们就可以放心的将它集成到软件中去了。由它再引发问题的概率就很小了(不是完全没有，因为我们不能遍历所有可能，只是挑选有代表性的异常情况进行测试)。

以上的例子可以说是单元测试的典范，但实际中却常常不是这么回事。我们常常发现写好的函数很难做单元测试，不仅工作量很大，效果也不见得好。其根本的原因是，函数没有遵循好一些原则：

1. 单一功能
2. 低耦合

反观 atoi 的例子，功能单一明确，和其他函数几乎没有任何耦合（我上面并没有写 atoi 的代码实现，大家可以自己实现，希望是 0 耦合）。

下面我举一个实际中的例子。

这是一个简单的 TL1 命令发送和解析软件，功能需求描述如下：

- 通过 telnet 与 TL1 服务器通讯
- 发送 TL1 命令给 TL1 服务器
- 解析 TL1 服务器的响应

TL1 是通讯行业广泛使用的一种协议，为了给不熟悉 TL1 的朋友简化问题，我定义了一个简化的格式：

```
CMD:CTAG:PAYLOAD;

CMD - 命令的名字，可以是任意字母开头，由字母和下划线组成的字符串
CTAG - 一个数字，用于标志命令的序号
PAYLOAD - 可以是任意格式的内容
; - 结束符
```

相应的，TL1 服务器的回应也有格式：

```
DATE
CTAG COMPLD
PAYLOAD
;
DATE – 日期和时间
CTAG – 一个数字，和 TL1 命令所携带的 CTAG 一样
COMPLD – 表明命令执行成功
PAYLOAD - 返回的结果，可以是任何格式的内容
; - 结束符
```

举例：

```
命令：GET-IP-CONFIG:1:;

结果：
2008-7-19 11:00:00
1 COMPLD
ip address: 192.168.1.200
gate way: 192.168.1.1
dns: 192.168.1.3
;

命令：SET-IP-CONFIG:2:172.31.2.100,172.31.2.1,172.31.2.3;

结果：
2008-7-19 11:00:05
2 COMPLD
;
```

软件的最上层可能是这样的：

```
Dict* ipconf = GET_IP_CONFIG();
ipconf->set(“ipaddr”,”172.31.2.100)
ipconf->set(“gateway”,”172.3.2.1”)
ipconf->set(“dns”,”172.31.2.1”)
SET_IP_CONFIG(ipconf);
```

以 GET_IP_CONFIG 为例，这个函数应该完成的功能包括：

- 建立 telnet 连接，如果连接尚未建立
- 构造 TL1 命令字符串
- 发送
- 接收反馈
- 解析反馈，并给 IP_CONF 结构复制
- 返回

我们当然不希望每个这样的函数都重复实现这些功能，所以我们定义了几个模块：

1. Telnet 连接管理
2. TL1 命令构造
3. TL1 结果解析

这里我们来分析 TL1 结果解析，假设设计为一个函数，函数的原型如下：

Dict* TL1Parse(const char* tl1response)

这个函数的功能是接受一个字符串，如果它是一个合法且已知的 TL1 回应，则将其中的结果提取出来，放入一个字典对象中。

这本来会是一个很便于进行单元测试的例子：输入各种字符串，检查返回结果是否正确即可。但是在这个软件中，有一个很特殊的问题：

TL1Parse 在解析一个字符串时，它必须要知道当前要处理的是哪条命令的回应。但是请注意，在 TL1 的回应中，是不包括命令的名字的。唯一的办法是使用 CTAG，这个命令和回应一一对应的数字。Tl1Parse 首先提取出 CTAG 来，然后查找使用这个 CTAG 的是什么命令。这里产生了一个对外调用，也就是耦合。

有一个对象维护了一个 CTAG 和命令名字对应关系的表，通过 CTAG，可以查询到对应的命令名，从而知道如何解析这个 TL1 response.

如此一来，TL1Parse 就无法进行单元测试了，至少不能轻易的进行。通常的桩函数的办法都不好用了。

怎么办？

重新设计，消除耦合。

将 TL1Parse 拆分为两个函数：

```
Tl1_header TL1_get_header(const char* tl1response)
Dict* TL1_parse_payload(const char* tl1name ,const char* tl1payload)
```

这两个函数都可以单独进行完整的单元测试。而这两个函数的代码基本就是 TL1Parse 切分了一下，但是其可测试性得到了很大的提高，得到一个可靠的解析器的可能性自然也大大提升了。

这个例子演示了如何通过设计来提高代码的可测试性—这里是单元测试。一个随意设计，随意实现的软件要进行单元测试将会是一场噩梦，只有在设计的时候就考虑到单元测试的需要，才能真正的进行单元测试。

7.5.1. 圈复杂度测量

模块的复杂度直接影响了单元测试的覆盖率。最著名的度量代码复杂度的方法是圈复杂度测量。

计算公式：V(F)=e-n+2。其中 e 是流程图中的边的数量，n 是节点数量。简单的算法是统计如 if、while、do 和 switch 中的 case 语句数加 1。适合于单元测试的代码的复杂度一般认为不应该超过 10。

7.5.2. 扇入扇出测量

扇入是指一个模块被其他模块所引用，扇出是指一个模块引用其他模块。我们都知道好的设计应该是高内聚低耦合的，也就是高扇入低扇出。一个扇出超过 7 的模块一般认为是设计欠佳的。扇出过大的模块进行单元测试不论从桩设置还是覆盖率上都是困难的。将系统的传出耦合和传入耦合的数量结合起来，形成另一个度量：不稳定性。

不稳定性 = 扇出 /(扇入 + 扇出)

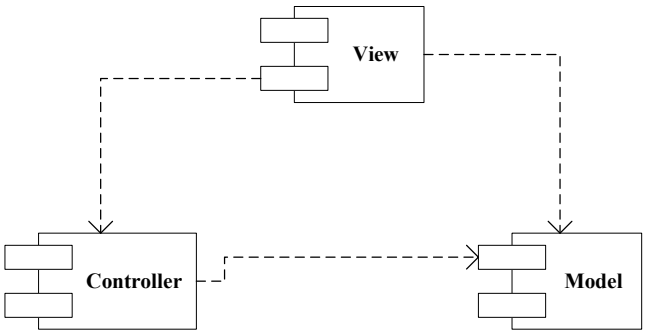
这个值的范围从 0 到 1。值越接近 1，它就越不稳定。在设计和实现架构时，应当尽量依赖稳定的包，因为这些包不太可能更改。相反的，依赖一个不稳定的包，发生更改时间接受到伤害的可能性就更大。

7.5.3. 框架对单元测试的意义

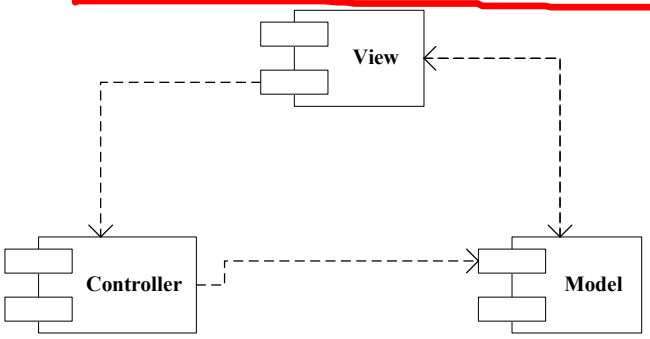
框架的应用在很大程度上可以帮助进行单元测试。因为二次开发者被限定实现特定的接口，而这些接口势必都是功能明确，简单，低耦合的。之前的框架示例代码也演示了这一点。这再次说明了，由高水平的工程师设计出的框架，可以强制初级工程师产生高质量的代码。

8. 维护架构的一致性

在实际的开发中，代码偏离精心设计的架构是很常见的事情，比如下图示例了一个嵌入式设备中设计的 MVC 模式：



View 依赖于 Controller 和 Model, Controller 依赖于 Model，Model 作为底层服务提供者，不依赖 View 或者 Controller. 这是一个适用的架构，可以在相当程度上分离业务，数据和界面。但是，某个程序员在实现时，使用了一个从 Model 到 View 的调用，破坏了架构。



这种现象通常发生在产品的维护阶段，有时也发生在架构的实现阶段。为了增加一个功能或者修正一个错误，程序员由于不理解原有架构的思路，或者只是单纯的偷懒，走了“捷径”。如果这样的实现不能及时发现并纠正，设计良好的架构就会被渐渐破坏，也就是我们常说的“架构”腐烂了。通常一个有一定年龄的软件产品的架构都有这个问题。如何监视并防止这种问题，有技术上的和管理上的手段。

技术上，借助工具，可以对系统组件的依赖进行分析，架构的外在表现最重要的就是各个部分的耦合关系。有一些工具可以统计软件组件的扇入和扇出。可以用这种工具编写测试代码，对组件的扇出进行检测，一旦发现测试失败，就说明架构遭到了破坏。这种检查可以集成在一些 IDE 中， 在编译时同步进行，或者在 check in 的时候进行。更高级的工具可以对代码进行反向工程生成 UML，可以提供更进一步的信息。但通常对扇入扇出做检查就可以了。

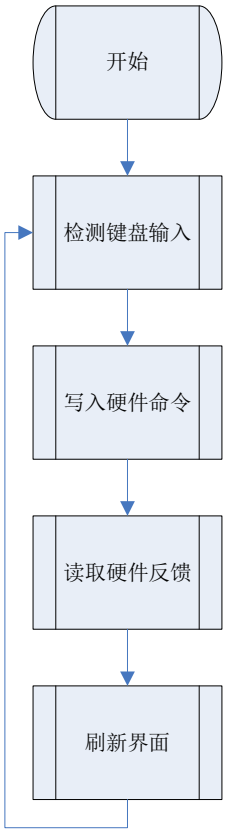
通过设置代码检视的开发流程，对程序员 check in 的代码进行评审，也可以防止此类问题。代码检视是开发中非常重要的一环，它属于开发后期阶段用来防止坏的代码进入系统的重要手段。代码检视通常要关注以下问题：

1. 是否正确完整的完成了需求
2. 是否遵循了系统的架构
3. 代码的可测试性
4. 错误处理是否完备
5. 代码规范

代码检视通常以会议的形式进行，时间点设置在项目阶段性完成，需要 check in 代码时。对于迭代式开发，则可以在一个迭代周期结束前组织。参与人员包括架构师，项目经理，项目成员，其他项目的资深工程师等。一般时间不要太长，以不超过 2 个小时为宜。会议前 2 天左右发出会议通知和相关文档代码，与会者必须先了解会议内容，进行准备。会议中，由代码的作者首先讲解代码需要实现的功能，自己的实现思路。然后展示代码。与会者根据自己的经验提出各种问题和改进意见。这种会议最忌讳的是让作者感到被指责或者轻视，所以，会议组织者要首先定义会议的基调：会议成功与否的标准不是作者的代码质量如何，而是与会者是否提供了有益的建议。会后由作者给与会者打分，而不是反之。

9. 一个实际嵌入式系统架构的演化

上世纪九十年代，互联网的极速发展让通讯测试设备也得到了极大的发展。那个年代，能够实现某种测量的硬件是竞争的核心，软件的目的仅仅是驱动硬件运行起来，再提供一个简单的界面。所以，最初的产品的软件结构非常简单，类似前面的城铁门禁系统。

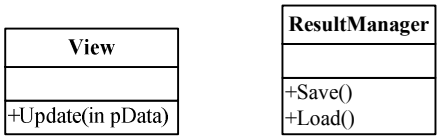


优点：程序简单明了的实现了用户的需求，一个程序员就可以全部搞定。

缺点：完全没有划分模块，底层上层耦合严重。

9.1. 数据处理

用户要求能将测量结果保存下来，并可以重新打开。数据存储模块和界面被独立出来。



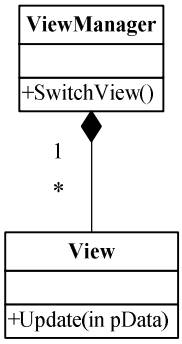
依然保持上面的主逻辑，但是界面部分不仅可以显示实时的数据，也可以从ResultManager 中读取数据来显示。

优点：数据和界面分离的雏形初步显现

缺点：ResultManager 只是作为一个工具存在，负责保存和装载历史数据，界面和数据的来源依然耦合的很紧。不同的界面需要的不同数据都是通过硬编码判断的。

9.2. 窗口管理

随着功能不断复杂，界面窗口越来越多，原来靠一个类来绘制各种界面的方式已经不能承受。于是窗口的概念被引入。每个界面都被视为一个窗口，窗口中的元素为控件。窗口的打开，关闭，隐藏则由窗口管理器负责。

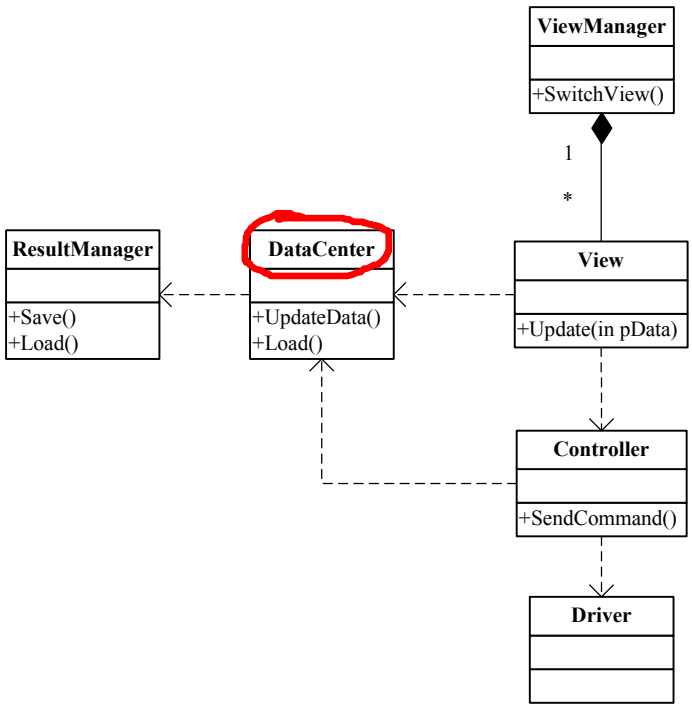


优点：界面功能以窗口的单位分离，不再是一个超大的集合。

缺点：虽然有了窗口管理器，但是界面依然是直接和底层耦合的，依然是大循环结构。

9.3. MVC 模式

随着规模进一步扩大，最初的大循环结构终于无法满足日益复杂的需求了。标准的 MVC 模式被引入，经历了一次大的重构。



数据中心作为 Model 被独立出来，保存着当前最新的数据。View 被放在了独立的任务中执行，定期从 DataCenter 轮询数据。用户的操作通过 View 发送给 Controller，进一步调用硬件驱动执行。硬件执行的结果从驱动到 Controller 更新到 DataCenter 中。界面，数据，命令三者基本解耦。ResultManager 成为 DataCenter 的一个组件，View 不再直接与其通讯。

MVC 模式的引入，第一次让这个产品了有真正意义上职责明晰，功能独立的架构。

9.4. 大量类似模块，低效的复用

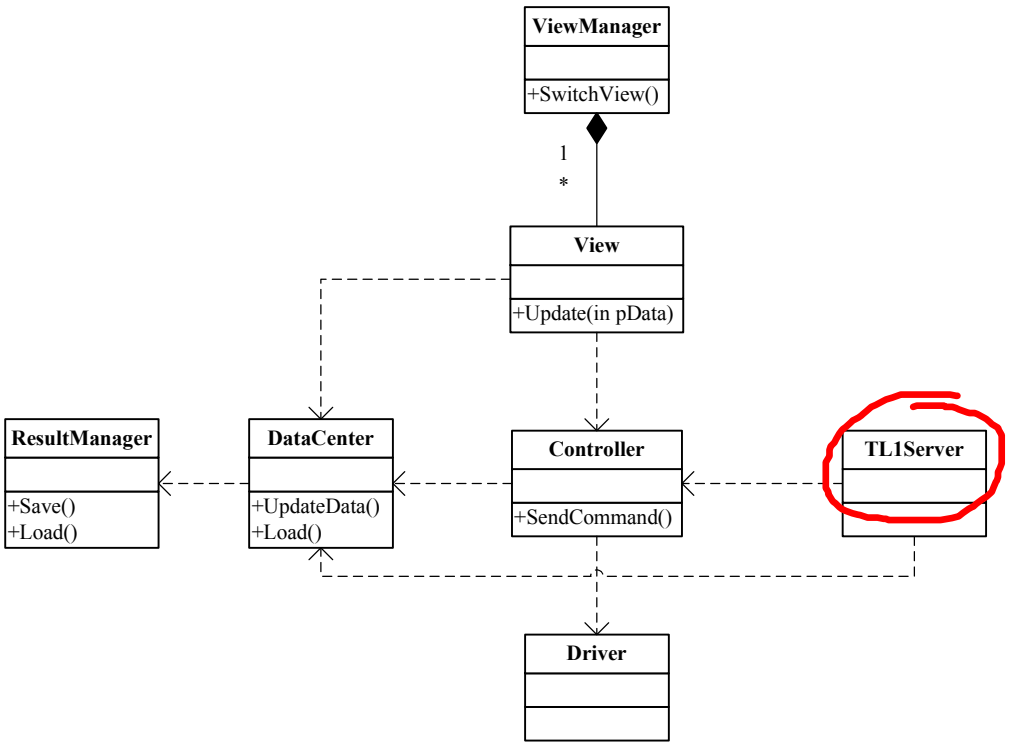
到上一步，作为一个单独的嵌入式设备，其架构基本可以满足需求。但是随着市场的扩展，越来越多的设备被设计出来。这些设备虽然执行的具体测量任务不同，但是他们都有着同样的操作方式，类似的界面，更主要的是，它们面临的问题领域是相同的。长期以来，复制和粘贴是唯一的复用方式，甚至类名变量名都来不及改。一个错误在一个设备上被修正，同样一段代码的错误在其他设备上却来不及修改。而随着团队规模的扩大，甚至 MVC 的基本架构在一些新设备上都没能遵守。

最终框架被引入了这个系列的产品。框架确定了如下内容：

1. MVC 模式的基本架构
2. 窗口管理器和组件布局算法
3. 多国语言方案（字符串管理器）
4. 日志系统
5. 内存分配器和内存泄露检测

9.5. 远程控制

客户希望将设备固定安放在网络的某个位置，作为“探针”使用，在办公室通过远程控制来访问这个设备。这对于原本是作为纯手持设备设计的系统又是一个挑战。幸运的是，MVC 架构具有相当的弹性，早期的投入获得了回报。



TL1 Server 对外提供基于 Telnet 的远程控制接口。在系统内部，它的位置相当于 View，只和原有的 Controller 和 DataCenter 通讯。

9.6. 自动化的 TL1 解释器

由于 TL1 命令相当多，而 TL1 又往往不是客户的第一需求，很多设备的 TL1 命令开始不完整。究其原因，还是手写 TL1 命令的解释器太累。后来通过引入 Bison 和 Flex，这个问题有所改善，但还是不足。自动化代码生成在这个阶段被引入。通过以如下的格式定义 TL1，工具可以自动生成 TL1 的编码和解码器代码。

```
CMD_NAME
{
  cmd = “SET-TIME-CONFIG::<ctag>::<year>,<month>,<day>,<hour>,<minute>,[<second>]”
  year = 1970..2100
  month = 1..12
  day = 1..31
  hour = 0..23
  minute = 0..59
  second = 0..59
}
```

9.7. 测试的难题

经过数十年的积累，产品已经成为一个系列，几十种设备。大部分设备进入了维护期，经常有客户提一些小的改进，或者要求修正一下缺陷。繁重的手工回归测试成为了噩梦。

基于 TL1 的自动化测试极大的解放了测试人员。通过在 PC 上运行的测试脚本，回归测试变得简单而可靠。唯一不足的是界面部分无法验证。

基于 Test Quest 的自动化工具需要在设备运行的 pSOS 系统上开发一个类似远程桌面的软件，而这在 pSOS 上并非易事。不过好消息是，由于框架固定了界面的风格和布局算法，基于 Test Quest 的自动化工具会有很高的识别效率。

9.8. 小结

从这个实际的嵌入式产品重构的历程可以看出，第三步引入 MVC 模式和第四步的框架化是非常关键的。成熟的 MVC 模式保证了后续一系列的可扩充性，而框架则保证了这个架构的在所有产品中的准确重用。

10. 总结

本课程是针对嵌入式软件开发的特点，讨论架构设计的思路和方法。试图给大家提供一种思想，启发大家的思维。框架，自动化代码生成和测试驱动的架构是核心内容，其中框架又是贯穿始终的要素。有人问我，什么是架构师，怎么样才能成为架构师？我回答说：编码，编码，再编码；改错，改错，再改错。当你觉得厌烦的时候，停下来想想，怎么才能更快更好的完成这些工作？架构师就是在实践中产生的，架构师来自于那些勤于思考，懒于重复的人。

2020/3/24 zhangshaoyan read end.