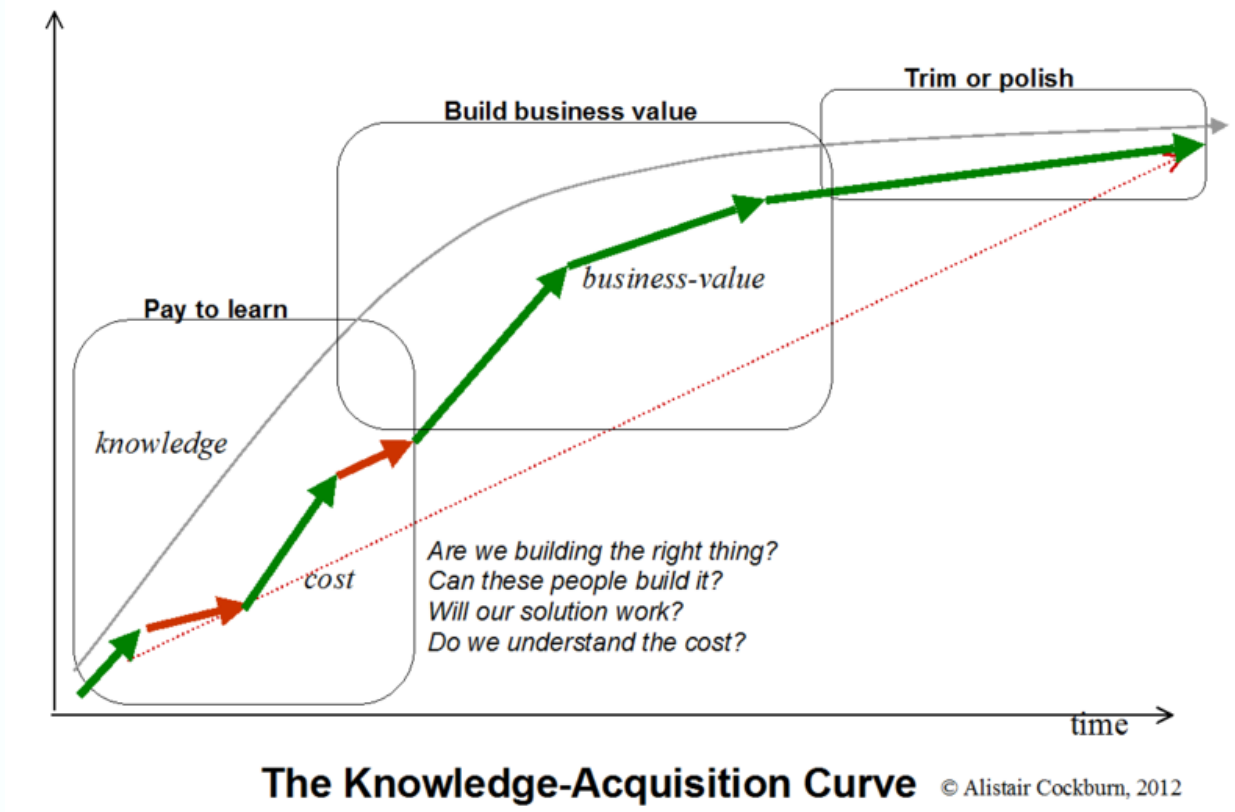
**问题的分类**

最初在 1999 年被 Dave Snowden 开发出来的 **Cynefin 框架**尝试把世界上的问题划分到了 5 个域中（大类）：

- 简单（Simple）问题，该域中的因果关系非常明显，解决这些问题的方法是 感知–分类–响应（Sense-Categorise-Respond），有对应的**最佳**实践
- 复合（Complicated）问题，该域中的因果关系需要分析，或者需要一些其他形式的调查和/或专业知识的应用，解决这些问题的方法是感知–分析–响应（Sense-Analyze-Respond），有对应的**好的**实践
- 复杂（Complex）问题，该域中的因果关系仅能够从回顾中发现，解决这些问题的方法是探索–感知–响应（Probe-Sense-Respond），我们能够感知**涌现**实践（emergent practice）
- 混乱（Chaotic）问题，该域中没有系统级别的因果关系，方法是行动–感知–响应（Act-Sense-Respond），我们能够发现**新颖**实践（novel practice）
- 失序（Disorder）问题，该域中没有因果关系，不可感知，其中的问题也无法被解决

显然，软件开发过程更多地是一个复杂（Complex）问题。在一个产品被开发出来之前，不确定性非常高，团队（包括业务人员和技术人员）对产品的知识也是最少的，而且需要大量的学习和尝试才可以明确下一步可能的方向。不幸的是，很多时候我们需要在一开始（不确定性最高的时候）就为项目做计划。这种从传统行业中非常适合的方法在软件开发领域不再适用，这也是敏捷开发、精益等方法论在软件开发

中更加适合的原因。




正因为软件开发事实上是一个学习的过程，我们学习到的新知识反过来会帮助我们对问题的定义，从而带来变化。这里的变化可能来自两个方向：

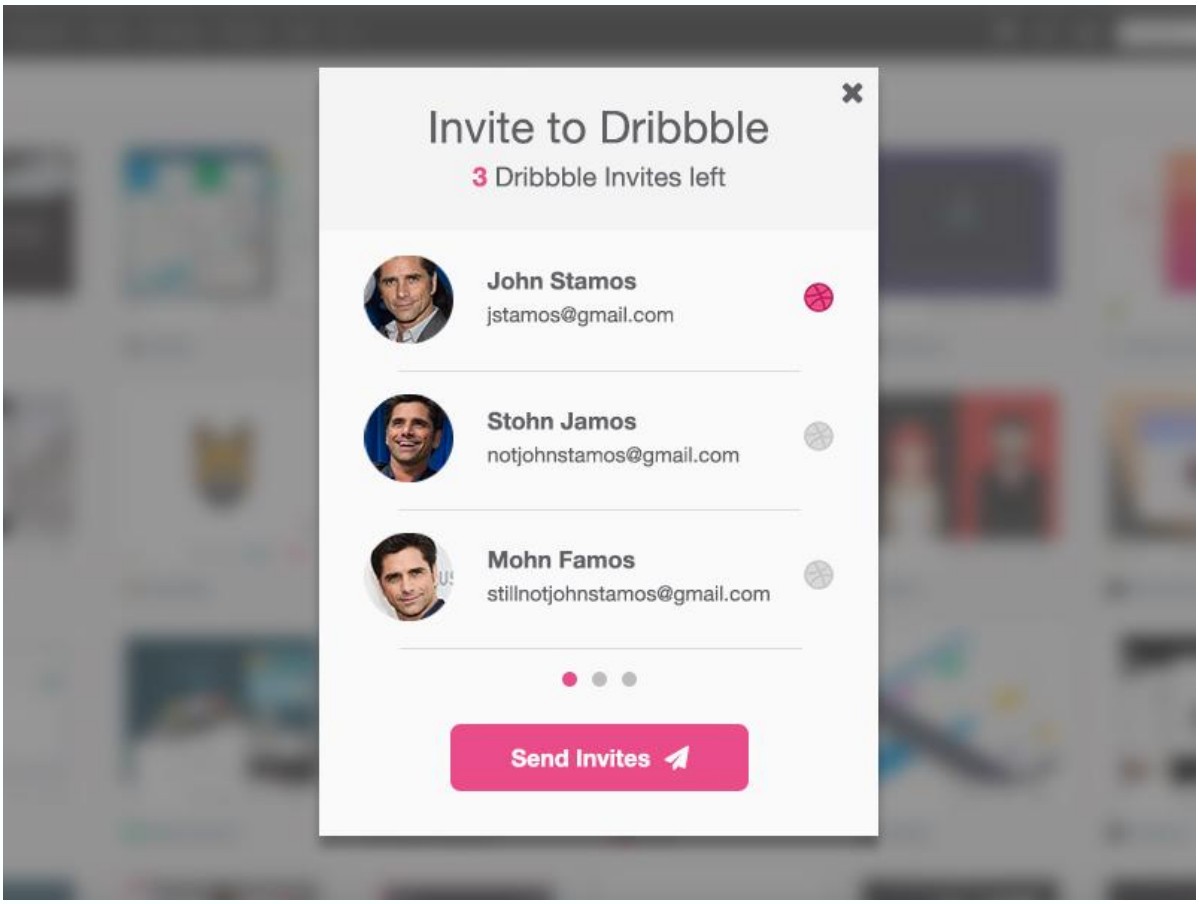
- 功能性
- 非功能性

功能性的变化指随着对业务的深入理解、或者已有业务规则为了匹配市场而产生的变化。比如支付方式由传统的货到付款变成了网银付款，又变成了微信支付、支付宝扫码等等。一个原始的电商平台仅仅提供基本的购物服务，但是后来可以根据已有数据产生推荐商品，从来带来更大的流量。这些变化需要体现在已有的代码中，而对代码的修改往往是牵一发而动全身。

非功能性的变化是指随着业务的发展，用户规模的增加，数据量的变化，安全认知的变化等产生的新的需求。比如 100 个用户的时候无需考虑性能问题，但是 100 万用户的时候，性能就变成了必须重视的问题。天气预报应用的数据安全性和网络银行的数据安全性要求也大不相同。

而在业务提出一个需求的时候，往往只是一个简化过的版本。

**非功能性复杂性**




这是一个经过设计师精确设计的界面，在它被设计出来之前，用户事实上无法准确的描述出它。设计过程中经历了很多的诸如：

- 线框图
- 颜色的确定
- 交互的动画
- 信息层次

往复多次之后，界面确定了。在没有仔细思考使用场景的时候，开发会误以为这个功能非常简单。但是如果你是一个有经验的开发者，很快会想到的一些问题是：

- 在宽屏下如何展示
- 在平板上如何展示
- 在手机上如何展示
- 即使仅仅支持桌面版，跨浏览器要考虑吗？支持哪些版本？
- 有些 UI 效果在低版本的浏览器上不工作，需要 Shim 技术

除此之外，依然有大量的其他细节需要考虑：

- 性能要求是什么样的？
- 安全性要考虑吗？
- 在网络环境不好的时候，要不要 fallback 到基础视图？
- 既然涉及发送邀请函，送达率如何保证

- 与外部邮件服务提供商集成时的工作量

等等。这些隐含的信息需要被充分挖掘出来，然后开发者才能做一个合理的评估，而且这还只是开始。一旦进入开发阶段，很多之前没有考虑到的细节开始涌现：字体的选用，字号，字体颜色，元素间的间距等等，如何测试邮件是否发送成功，多个角色之间的 conversation 又会消耗很多时间。

**需求的变化方向**

作为程序员，有一天你被要求写一段代码，这段代码需要完成一件很简单的事：

1. 打印”Hello, world”5 次

很容易嘛，你想，然后顺手就写下了下面这几行代码：

```
print("Hello, world")
print("Hello, world")
print("Hello, world")
print("Hello, world")
print("Hello, world")
```

不过，拷贝粘贴看起来有点低端，你做了一个微小的改动：

```
for(var i = 0; i < 5; i++) {
  print("Hello, world")
}
```

看起来还不错，老板的需求又变成了打印”Goodbye, world”5 次。既然是打印不同的消息，那何不把消息作为参数呢？

```
function printMessage(message) {
  for(i = 0; i < 5; i++) {
      print(message);
  }
}

printMessage("Hello, world")
printMessage("Goodbye, world")
```

有了这个函数，你可以打印任意消息 5 次了。老板又一次改变了需求：打印”Hello, world”13 次（没人知道为什么是 13）。既然次数也变化了，那么一个可能是将次数作为参数传入：

```
function printMessage(count, message) {
  for(i = 0; i < count; i++) {
      print(message);
  }
}

printMessage(13, "Hello, world");
printMessage(5, "Goodbye, world");
```

完美，这就是抽象的魅力。有了这个函数，你可以将任意消息打印任意次数。不过老板是永远无法满足的，就在这次需求变化之后的第二天，他的需求又变了：不但要将”Hello, world”打印到控制台，还要将其计入日志。

没办法，通过搜索 JavaScript 的文档，你发现了一个叫做高阶函数的东东：**函数可以作为参数传入另一个参数！**

```
function log(message) {
  system.log(message);
}

function doMessage(count, message, action) {
  for(i = 0; i < count; i++) {
      action(message);
  }
}

doMessage(5, "Hello, world", print);
doMessage(5, "Hello, world", log);
```

这下厉害了，我们可以对任意消息，做任意次的任意动作！再回过头来看看那个最开始的需求：

1. 打印”Hello, world”5 次

稍微分割一下这句话：**打印，”Hello, world”，5 次**，可以看到，这三个元素最后都变成了可以变化的点，软件开发很多时候正是如此，需求可能在任意可能*变化*的方向上变化。这也是各种软件开发原则尝试解决的问题：如何写出更容易扩展，更容易响应变化的代码来。

**小结**

软件的复杂性来自于大量的**不确定性**，而这个不确定性事实上是无法避免的，而且每个软件都是独一无二的。另一方面，软件的需求会以各种方式来变化，而且往往会以开发者没有预料到的方向。比如上面这个小例子中看到的，最后的需求可能会变成将消息以短信的方式发送给手机号以 185 开头的用户手机上。