南京大學

NANJING UNIVERSITY

对质量属性的分析和讨论

Scalability & Sustainability, Flexibility & Reusability

学　　院：软件学院

创建时间：2021年5月1日

学生姓名：刘育麟

学　　号：181250090

教　　师：张贺、潘敏学

2021 年 5 月 25 日

1 Scalability & Sustainability

1.1 一般场景

1.1.1 Scalability

见表1。

表 1: Scalability的一般场景

Portion of Scenario	Possible Values
Source	开发者、维护人员、系统管理员
Stimulus	增加系统资源的负载或需求，如进程、I/O、或是储存的负载或需求。
Artifact	服务、代码、数据、接口、硬件
Environment	任何一个阶段，从系统需求开始被规划到系统上线后都可以。增加可以是暂时的、也可以是永久性的。
Response	系统提供一个新的资源去满足新的需求或是负载
Response Measure	负载增加时提供额外资源的时间、增加的成本与新资源能够提供的新价值的比率

1.1.2 Sustainability

见表2。

表 2: Sustainability的一般场景

Portion of Scenario	Possible Values
Source	开发者、维护人员、系统管理员、领域专家
Stimulus	软硬件版本的落后、运行平台的更新与迭代
Artifact	接口、代码架构、系统服务
Environment	长时间运行后的系统
Response	根据新的软硬件版本，设计新的接口与代码框架.对旧有设计进行修改或抛弃，能一样实现原来的用户需求
Response Measure	新的系统不会产生问题、能满足用户需求、修改和重构系统的成本是否低于新开发一个系统的成本

1.2 特别场景

1.2.1 Scalability

见图1、图2。

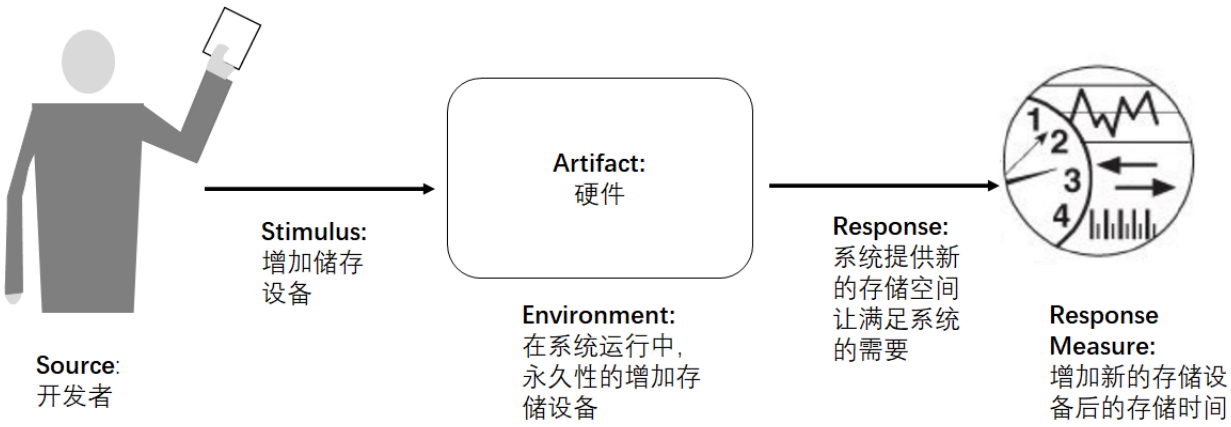

图 1: Scalability的特别场景-1

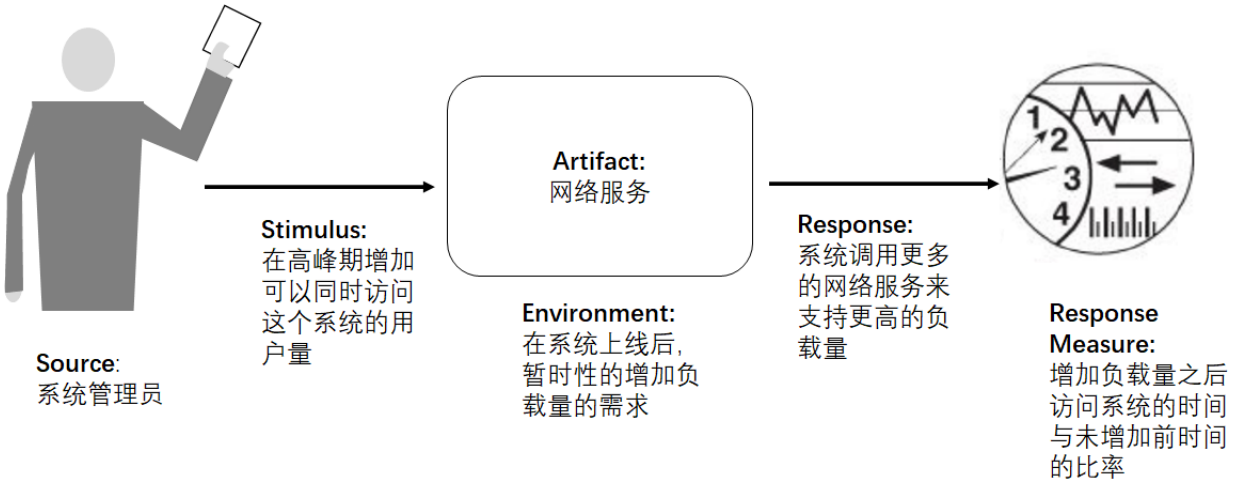

图 2: Scalability的特别场景-2

1.2.2 Sustainability

见图3、图4。

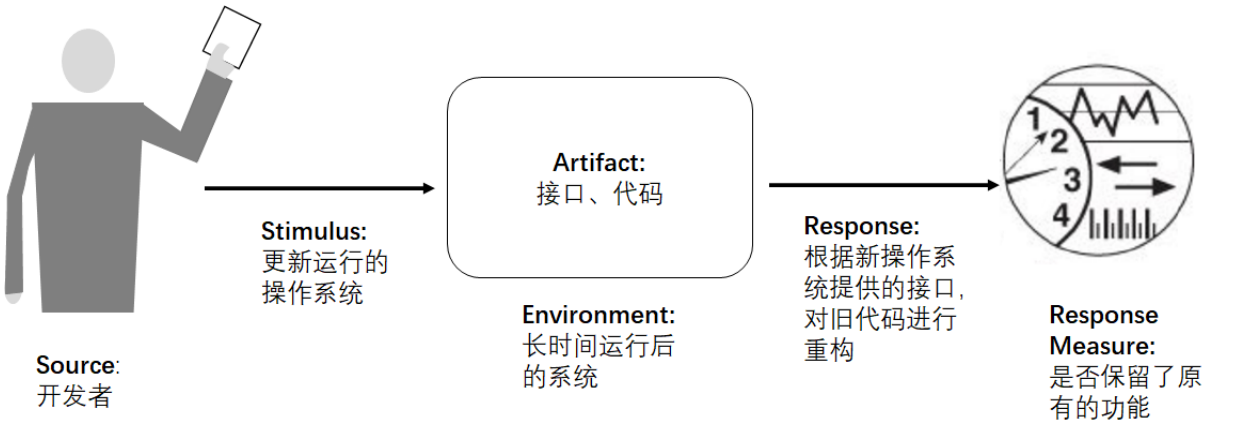

图 3: Sustainability的特别场景-1

1.3 关系&差异

1.3.1 关系

Salability用中文翻译是可扩展性，而Sustainability用中文翻译是耐久性，根据前面的分析，我们可以轻易的发现两者一个很重要的关联，要是可扩展性好，则在持久性方面，可以得到很大的提升。因为可扩展性关注的是增加一个资源来满足新的需求，如果这个系统运行很长时间，资源是一个新型号的资源，那么就需要进行持久性的度量，而度量的指标就是系统在引入新环境时的可扩展性，也就是说，引入新资源时，可以将这个资源看成是新的需求，那么，考虑如何将这个资源加进系统就是要

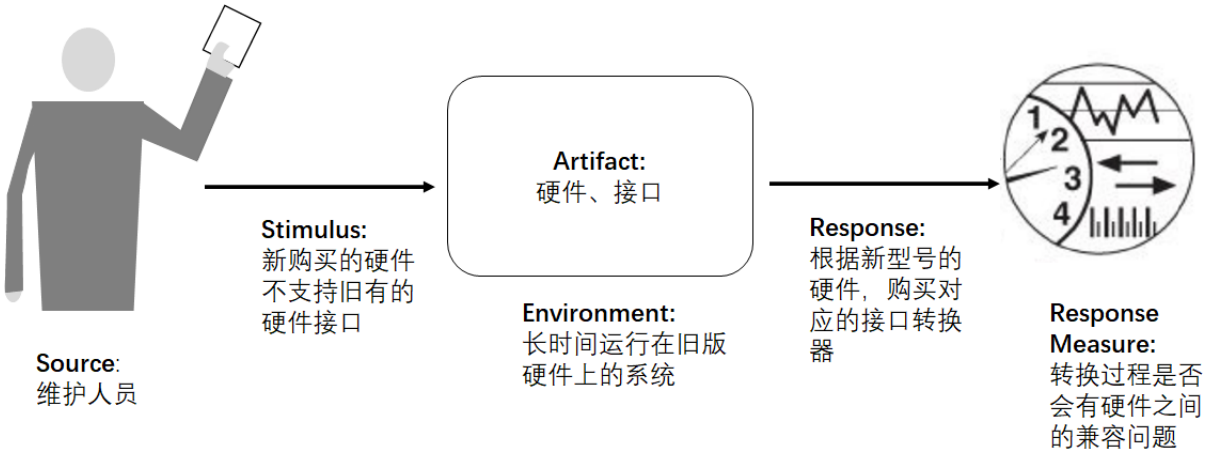

图 4: Sustainability的特别场景-2

判断该系统的可扩展性。可以说，这两个质量属性可以是相辅相成、彼此有交集的。

1.3.2 差异

可扩展性主要关注的是新功能的增加，而耐久性主要关注的是用新的功能代替旧的功能，一个需要长时间运行的系统，更需要关注的是耐久性的部分，而对于处在一个变化很快环境中的系统，这个系统会更多地关注可扩展性的部分，下面用一个例子来说明。

以一个超市的收银系统来说，在整个系统被规划出需求的那个阶段起，可扩展性就已经需要被考量。在规划需求的阶段，需求会不断的增加或减少，可扩展性差的系统就容易因为不断变化的需求而产生需要进行大量修改的问题，就像是如果我要在系统添加一种商品，可扩展性好的就不需要修改太多的东西，而可扩展性差的就需要对每个逻辑判断进行修改。可扩展性在系统上线后仍然存在，因为时代是与时俱进的，比如十年前的系统全是用硬币与钞票进行支付，而现在已经都是使用电子货币进行交易，那么，就如同前面提到的两者的关联，持久性也展现出来了。持久性确保整个系统有相应的接口或是架构能支持新的支付方式被引入，而这个引入的难易度主要体现在可扩展性上面，好的可扩展性可以更好的引入新的东西，也就是说这个系统的耐久性也相对的被提升了。但是，如果这个系统并不是增加新的需求，而是较为老的系统或是技术要被淘汰，譬如这个收银系统因为`Windows XP`不再被维护，要换新的操作系统，那么就不需要用可扩展性进行度量，而主要的是关注这个系统好不好被重构，重构花费的资源是否比直接建立一个新的系统来的少。

1.4 Strategy & Tactics

见表3。

表 3: Scalability和Sustainability的Strategy & Tactics

	Scalability	**Sustainability**
Strategy1	提高内聚	使用先进的软件开发技术和条件
Tatic1.1	模块遵循单一职责原则	敏捷开发
Benefit1.1	每个某块逻辑简单	能及时应对开发中的需求变更
Penalties1.1	过度的内聚增加系统中元素的依赖	不适用于大型团队
Tatic1.2	将任务分解成多个功能独立的子函数	面向对象开发
Benefits1	方便排查错误	容易进行代码的编写
Penalties1	过度的内聚增加系统中元素的依赖	程序的处理效率较慢
Strategy2	降低耦合	对系统进行维护
Tatic2.1	使用接口、封装等松耦合手段	定期维护系统并且替换老旧的组件
Benefit2.1	调用者不需要知道具体实现	提高系统的寿命
Penalties2.2	不利于深入调试	增加运维成本
Tatic2.2	严格限制依赖	定期对系统进行检查或测试
Benefits2	降低耦合，容易修改和扩展	保障运行阶段的质量
Penalties2	不利于深入调试	检查期间会导致系统需要暂时停止运行

2 Flexibility & Reusability

2.1 一般场景

2.1.1 Flexibility

见表4。

表 4: Flexibility的一般场景

Portion of Scenario	**Possible Values**
Source	开发者、维护人员
Stimulus	系统需要修改需求、系统需要更改运行环境
Artifact	代码架构、代码内容、系统服务、接口实现
Environment	任何一个阶段，从系统需求开始被规划到系统上线后都可以。
Response	按照需求的变更对系统进行相应的修改以适配新的需求
Response Measure	修改需求的成本、修改后的系统运行时间

2.1.2 Reusability

见表5。

表 5: Reusability的一般场景

Portion of Scenario	Possible Values
Source	开发者、维护人员
Stimulus	系统的某个功能需要被应用在不同地方
Artifact	代码内容、系统服务、接口
Environment	任何一个阶段，从系统需求开始被规划到系统上线后都可以。
Response	将旧有的部分代码使用接口调用运用在新代码上面
Response Measure	复用的代码被修改可能的问题、复用是否会有数据泄露的问题

2.2 特别场景

2.2.1 Flexibility

见图5、图6。

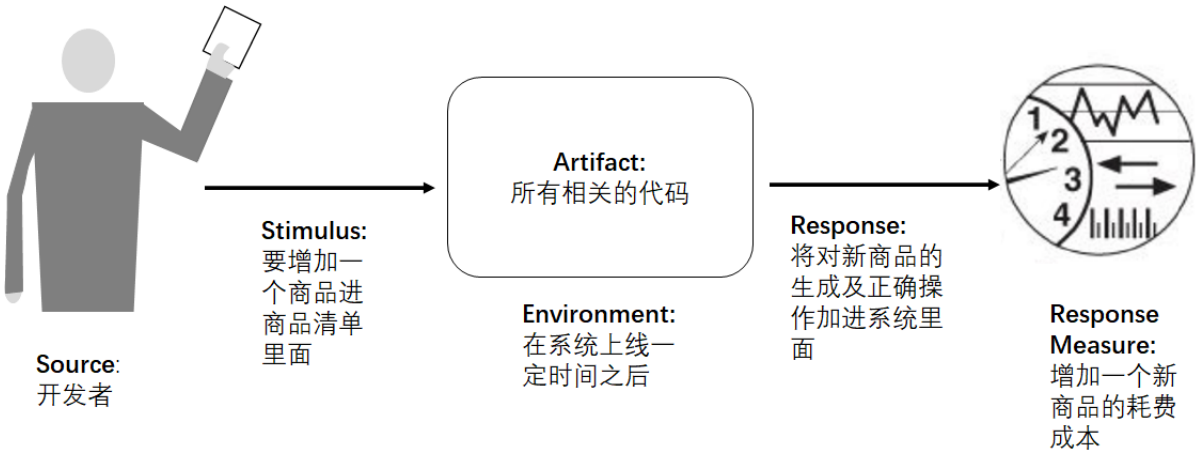

图 5: Flexibility的特别场景-1

2.2.2 Reusability

见图7、图8。

2.3 关系&差异

2.3.1 关系

Flexibility在中文的翻译是灵活性，Reusability在中文的翻译是可复用性。两者都是为了让代码能适应软件中的一个很重要的特征——可变性。软件是一个需要被一直迭代的产物，没有任何一个软件在需求确定下来开始开发后，就不需要再继续的修改或是增加需求。而软件的复杂性和不可见行决定了软件是一个不好被修改的东西，所以，系统中许多组件需要被复用或是泛化。灵活性主要就是检验该系统的泛化性，也就是在不同的需求或是获取到不同的数据时，系统能不能有相应的反应，而不需要再

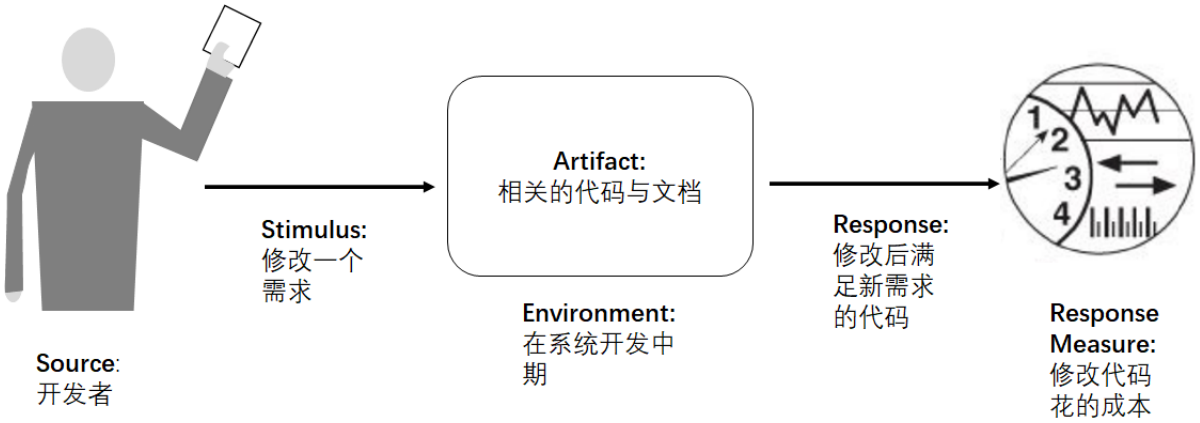

图 6: Flexibility的特别场景-2

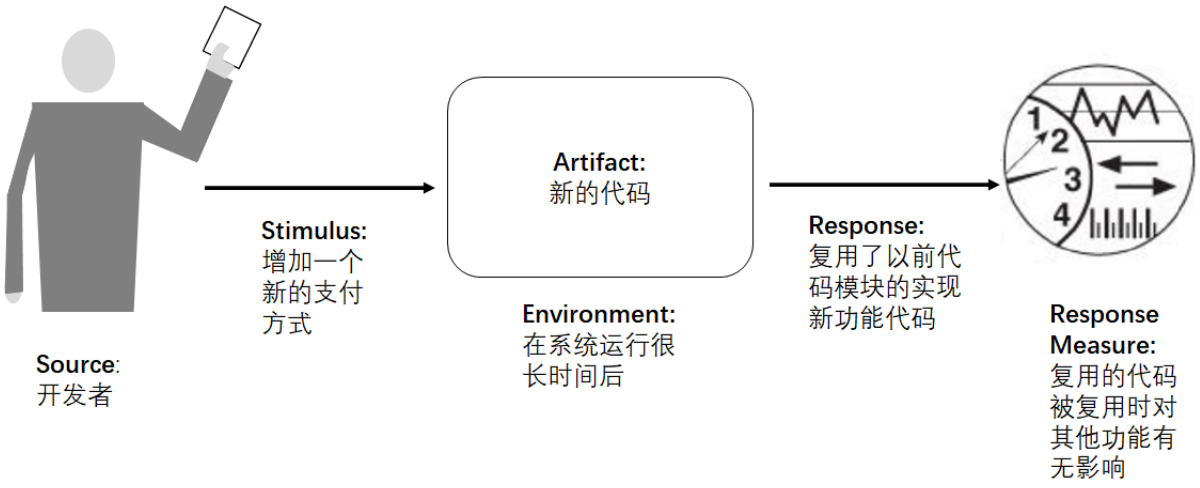

图 7: Reusability的特别场景-1

重新的构建整个系统。而可复用性就是检验系统的复用功能，在获取新的需求使，开发者能不能只需要根据原本的框架去进行代码的修改或是调用旧有代码实现新的功能，而不需要重写整个需求。

2.3.2 差异

虽然说两者的解决方法都是将代码分成各个模块进行调用，降低软件的修改困难，但是，两者的关注点是不同的。灵活性主要关注的是代码是否能适配环境或需求的变化，而可复用性更关注的是代码的复用以及简洁度，可以说，可复用性是灵活性的一个实现思想，他让代码变得可复用，也就相对的让代码更加弹性，能在新的需求或是新的环境中，能进行很小的修改或是不需要修改，就能适用当前的环境。

以商品交易系统为例，现在我需要设计一个系统能添加商品，那么灵活性更加关注的就是对于添加商品的形式以及数量上的限制，比如说我添加的商品的存储数量超过了电脑`int`类型能储存的数量，那么要进行什么操作能避免这种越界的问题，而如果

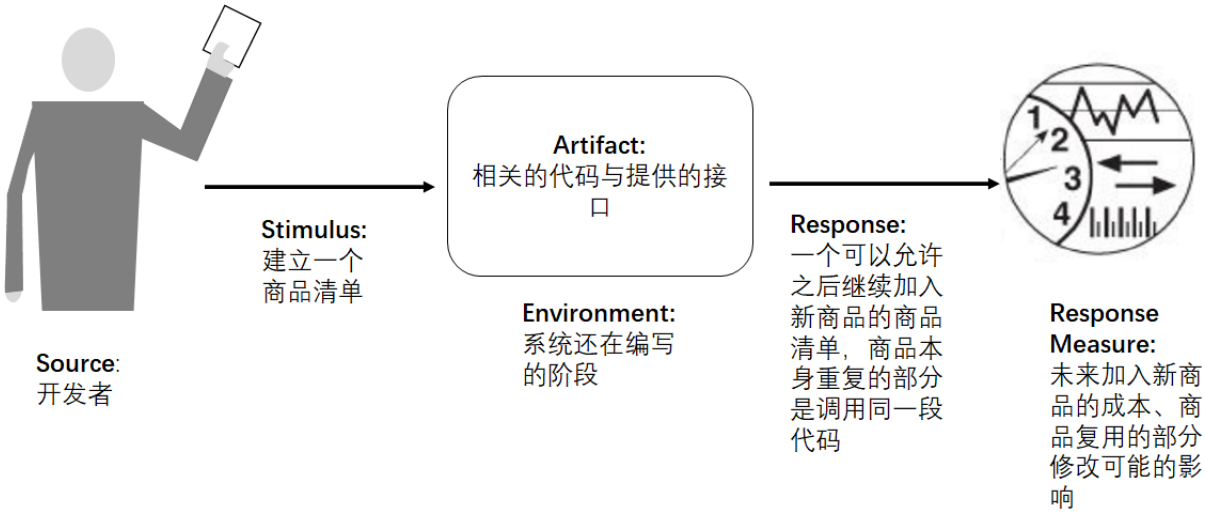

图 8: Reusability的特别场景-2

商品在添加的时候少了部分信息，会不会导致系统的崩溃，这就是灵活性要考虑并且极力避免的问题。而代码的复用能相对的减少这一部分的问题，因为已经被验证过不会导致系统崩溃的代码，应该能被复用于系统中其他部分，或是因为复用的某块代码因为某种原因出问题了，则修改对应部分的代码就好了，不需要考虑其他的部分，因为整个系统中不应该有实现重复逻辑的代码出现。就像是购买商品的流程中，每个商品都会有一个付款模块，如果要为每一个商品都写一个付款模块，那么当付款方式改变时，就要同时改很多代码，造成修改上的不必要浪费，而且里面和多重复的逻辑，如果代码中出现问题，无法精准的定位是哪一块代码除了问题。灵活性更看中的是系统的在不同环境的适应能力，可复用性则是看重系统模块间的泛用性，可以说一个关注的是整体，一个关注的是部分。

2.4 Strategy & Tactics

见表6。

表 6: Flexibility和Reusability的Strategy & Tactics

	Flexibility	**Reusability**
Strategy1	考虑系统运行的各种场景	使用较好的开发方法
Tatic1.1	进行不同的测试	结构化程序
Benefit1.1	可以增加系统在不同环境的适应能力	每个模块都只需要负责部分功能
Penalties1.1	消耗大量资源编写测试	一个模块的问题可能导致系统许多部分崩溃
Tatic1.2	代码中考虑各种可能发生的问题	面向对象开发
Benefits1	系统不容易崩溃	代码的封装性和编写效率高
Penalties1	大量判断导致运行缓慢	程序处理效率较慢
Strategy2	用简单构造复杂	每个功能切割成一个个小单元
Tatic2.1	将代码分模块编写	遵循单一职责原则
Benefit2.1	代码可以重复进行调用	模块可以重复利用
Penalties2.2	单一模块的问题可能影响多处	模块间依赖增加
Tatic2.2	提高模块的通用性	遵循开闭原则
Benefits2	程序容易修改和扩展	单元在确定可用之后就不需要进行修改
Penalties2	泛化性的代码运行速度低下	有问题没被发现的单元可能会影响系统很多地方