|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 起止时间 | 2020年6月22日至2020年7月10日 | | | | | | | | | |
| 选择参加的**专项实训**  在（ ）内打√ | 必修（1、2、3之中任选一并4、5） | | | | | | | | | 选修 |
| 1、社会实训  （） | 2、本学科专业  技能（ √） | | 3、科技创新  （ ） | | 4、学科前沿动态及讲座（ √） | | 5、经典原著与学科前沿文献阅读（ √） | | 6、其它学科特色课等（ √） |
| 指导教师 | 冯居易 | | 职称 | | 讲师 | | 教学单位 | | 信息学院 | |
| 实训内容及收获（后附实训报告） | 通过这次培训,我成长了很多。我在培训期间,我从实践中学到了许多,遇见了一些困难,也让我看到了我存在的很多问题。在考核上让自己的考核中把握得比较全面。总会出现有这样那样那样的问题,当前软件的功能也越来越繁琐,学得不到某种深度和宽泛的知识,很难在实践和操作中得到应付。所以,它直接反映了我们学习上的不足、欠缺和疏忽。我们需要认真地去钻研和学习,不断开拓视野,增强自身的实践技巧,才能顺利做好今后的考核。  培训过程中的收获主要有四个方面：一是通过间接参与企业的运营过程，我们在这次培训中学到很多关于培训的知识,同时也进一步地加深了我们的理论知识,增强了对理论与培训的基础知识,并且顺利地完成了本次培训的各项教学任务。第二、是增强实际从事工作的能力,为以后的就业及今后的工作中积累一些宝贵的技术培训经验。第三,小组在这次训练中初步建立,互相交流与沟通,彼此帮助,为我们达到目标而奋斗,让我清楚地看到了大家的劳动和工作热情,和大家的劳动和工作技巧。 | | | | | | | | | |
| 成绩  评定 | 综合评定成绩： 指导教师签字：  2020年 9月 11日 | | | | | | | | | |
| 系审核意见 | 签字（盖章）：  2020年 9月 14日 | | | | | | | | | |
| 学院审定 意见 | 签字（盖章）：  2020 年 9月15日 | | | | | | | | | |

《代码整洁之道》读后感

代码是我们用来表达需求的语言。对我们来说，代码将是一项无法避免的技能，需要好好学习。对于基本的代码编写，清洁将是提高编写、阅读和修改效率的好方法。事实上，对于我这个没有太多接触代码的初学者来说，在开始编写代码时，会有很多非标准的地方。如果在基本格式、命名等方面有一定的针对性要求，将对以后的写作和基本习惯的培养非常有帮助。

不规范的写作通常源于不耐烦，或是“忘了它，快跑！”有时候，这种想法往往是坏代码的开始，或者是养成好习惯的需要。混乱往往会造成混乱，降低效率。相反，整洁的代码倾向于减少人们的修改时间并提高效率。相反，在编写代码时，需要注意一些细节。简洁的代码力求集中精力。每个函数、类和模块都集中在一件事上，完全不受周围细节的干扰和污染。此外，还有许多具体要求。接下来，我们将以小部分的形式解释它们。

在编码过程中，命名是不可避免的。函数、变量等需要命名。一个好的名字可以让我们清楚地理解一个物体的功能等等。对于命名来说，最重要的是不辜负名称，也就是说，我们需要注意的问题不是代码的简单性，而是代码的模糊性和代码对应函数的准确性。在实际命名过程中，我们也需要注意一些问题。首先，我们需要避免误导，尽量不要使用非常相似或有特殊目的的名称。此外，对于一个类别的命名，有必要进行有意义的区分，而不是一些模糊的废话。简而言之，为了区分名字，我们应该以读者能够识别差异的方式来区分它们。此外，我们还需要注意命名可以读出和检索的名称，避免编码、思维映射、双关语和过于专业的单词。从一个词到另一个词，从一个词到另一个词，这是最完美的状态。最后，在命名时，我们需要添加有意义的上下文，避免添加一些无用的上下文。通常，在命名时，我们会一起命名一串单词。此时，我们常常忽略上下文。因为有些单词是连在一起的，我们可以很容易地想到它们的意思，但是当我们单独拿起它们时，通常很难阅读，所以我们需要添加一些必要的上下文。

函数在我们的代码编写中起着极其重要的作用。函数的规则是简短。此外，每个函数应该一目了然，只说一件事。此外，每个函数将按顺序转到下一个函数。这样的功能可以让您快速了解其对应的功能，并且具有很强的相关性，这样我们就可以了解接下来的步骤。用文本的话说，函数应该做一件事，做好它，并且只做它。对于特定功能的准备，还需要某些规范。首先，应该使用描述性名称。对于一类功能，命名方法应一致，并且应使用同一行中的短语、名词和动词。对于函数参数，最理想的状态是无参数，其次是一个参数和两个参数。尽量不要使用三个或更多参数。参数的命名也需要名副其实。因为函数应该做一件事，要么做某事，要么回答某事，所以我们需要将指令与查询分开。另一方面，我们需要使用异常而不是返回错误代码，这样错误处理代码就可以从主路径代码中分离出来。还有一点很重要，不要重复。

注释似乎是我们阅读代码的一个工件，但我们依赖注释从一开始就是错误的。真正好的代码不需要注释。当然，在某些情况下也需要适当的评论。但注释只是一种安慰，不能美化糟糕的代码。好的评论包括法律信息版权、关于提供信息的评论、意图、警告等。坏的评论通常提供冗余和误导性信息，或者太多的信息和缺乏密切联系导致我们在阅读过程中花费太多时间在评论上，完全忽略了代码本身。还有一个我们经常做的行为，注释掉代码，我们经常注释掉代码是为了方便，但是事实上，我们已经更新了一个版本，但是旧代码仍然存在，并且没有被删除，这已经成为一个障碍。

代码格式非常重要，需要认真对待。它主要分为两种类型：一种是垂直格式，另一种是水平格式。对于垂直格式，我们主要需要注意分离、距离和顺序。具体来说，我们需要注意我们编写代码的顺序以及每个区域的间隔和距离，这对我来说仍然比较常用。由于VHDL是一种并行语句，编写过程中的顺序和距离将非常有助于我们阅读代码，阅读后，如果我们按照一定的过程顺序写下来，我们就可以根据过程了解各个部分的功能以及实际输入输出和具体实际信号的连接。至于水平格式，我们需要注意水平对齐、缩进和空白范围，这将在我们的写作过程中对每一层都很清楚。此外，如果您在一个团队中，您需要制定一个团队规则，这样每个人编写的代码都可以更容易阅读和相互交流。

关于对象和数据结构，两者都有优点和缺点，这取决于我们的具体用途。对象暴露行为和数据隐藏使得在不修改现有行为的情况下添加新的对象类型变得容易，并且向现有对象添加新行为也很困难。数据结构公开数据时没有明显的行为。向现有数据结构添加新行为很容易，向现有函数添加新数据结构也很困难。它们之间存在反对称性，没有优劣之分，但可以根据使用场合灵活改变。

如果一个完整的程序考虑所有的情况，它通常需要考虑错误处理。错误处理是基于程序的，但它不能改变原始程序的逻辑。最好的情况是在具体的逻辑设计过程中考虑错误处理。我们只需要根据具体的错误上下文返回相应的异常值，就可以根据返回的异常值判断错误情况。此外，我们需要使用不可控制的异常，并且我们不能返回返回代码并返回和传播空值，这将使情况复杂化，但会添加不必要的部分，增加读取代码的难度并降低可读性。

我们经常需要使用第三方程序或与他人对接程序。在这种情况下，边界尤其重要。接口提供者和用户之间存在紧张关系。第三方软件包和框架提供商追求通用性，以便能够在多种环境中工作并吸引广泛的用户。用户希望专注于满足特定需求的界面。这种张力会导致系统边界出现问题。这要求我们浏览和了解边界，并使用学习测试来确保第三方软件包按照我们想要的方式工作。但有时我们会遇到接口尚未定义的情况。此时，我们可以使用不存在的代码。想象一个接口，从离接口最远的地方编写它。此外，接口可能会更改，这导致我们的程序需要更改。这就要求我们的边界是干净的，尽可能少地依赖边界的价值，尽可能多地依赖你能控制的东西，以免将来被它控制。

单元测试通常伴随着代码的相应部分，它们的使用频率几乎与代码的使用频率相同。因此，在编写单元测试的过程中，我们还需要要求代码是干净的。相对而言，如果测试代码不干净，我们需要花费大量时间更改测试代码，以便测试代码段的功能。干净的代码要求可读性，并且只测试一个带有断言语句的概念。测试的整洁性需要遵循一些规则：快速、独立、可重复、自给自足的验证（测试应具有布尔输出）、及时。

系统是一个程序的大框架，由许多部分组成。在具体系统的建设和使用中，我们需要将这两部分分开。具体方法之一是将整个构造过程移动到main或称为main的模块。在其他部分的建模和设计中,我们还需要假定所有的对象都已经被正确地进行了构造和设置,并在此基础上再次使用它。另一种方法可以进行构造和利用分离,其中控制逆向注入是一种重要的应用方式。它从该关注对象开始承担第二个责任,并将其关注点转移到另外一个关注该关心对象的关心物,因此遵守单一负责原则。在进行依赖性关系管理的场景中,对象本身不应该负责地实现对自己的依赖性关系。相反,它们更多地应该把这一权力和责任移交给其他"强大"的机制,以便于实现对其他控制人权力的逆转。由于初始化设置的容器是一个完整而不同的全局性问题,所以此次授权的机制一般都是一个具有特定目标或者具有特殊用途的容器。此外,在进行系统的设计时,我们根据现有容量进行设计。当我们达到一定的利用率时，我们会考虑产能扩张的问题，而不是一开始就把容量设置成一个大的值，这会浪费资源。此外，我们可以使用某些标准。这样，重用的思想可以提高一定的效率。

简单设计有四条规则：运行所有测试，不可重复，满足程序员的意图，并可能减少类和方法的数量。运行所有测试是所有这一切的前提。只有通过测试的系统才能朝着整洁的方向发展。在通过测试的基础上，我们可以使用以下三个规则进行重建，重新考虑设计是否落后，并逐步发展。此外，我们还需要实现最基本的不能重复的条件，明确表达作者的意图和少量功能。

并发编程有许多优点，但也有一些困难。并发可以解耦，这可以提高应用程序的吞吐量和结构。然而，这并不意味着并发编程可以优化程序。它还将带来一些开销和变化。在适当的场合下正确使用，可以起到相应的效果。但是，如果使用不当或程序过于简单，只会增加复杂性。在编程过程中，最重要的还是要遵循单一的权限和责任问题，加上多线程和数据共享，并配合设备代码。其中，最重要的一点仍然是代码的清洁度。

读了这本书后，我最大的收获是我需要保持代码干净，而不是我已经学会了保持代码干净。事实上，再读一遍之后，我仍然不理解很多概念，但我认为我在读了这本书之后有了实现这些事情的基本触感，当然，这种触感需要我通过后来的实际行动和具体学习来改善和加深我的印象。

事实上，对我来说，命名、格式、错误处理等模块在我实际的代码编写过程中都具有重要的实际意义。在我目前的学习和代码编写过程中，我对基本代码的清洁度也有一种基本的感觉。经过一段时间，我觉得我自己的程序有问题，但是在我仔细思考和具体测试的过程中，我发现没有问题。事实上，我以前考虑过一点，但在以后查看代码的过程中没有考虑到这一点，这会产生影响。相反，如果我在具体的书写过程中说出具体的数量，相应信号的功能会更加详细和准确。此外，可以避免添加注释。

保证源代码的品质十分必要。坏的代码就有可能会摧毁某企业。对于一个非常需要十分关心自己的代码质量的年轻人来说,留下一个只是专门关心自己的代码交付而不是专门关心自己代码质量的年轻人将会非常痛苦。什么样才是整洁编程语言代码,不同的时期都有着不同的定义界限。我们相信整洁的语言源代码必须要完全符合所需要使用语言的源代码规范;是可以再次重复使用;易于设备的安装与维修;内容简单。坏的代码将让你想做太多的事情或者东西,并且还会带有混乱的设计意图;干净源代码中的每一个函数、类和模块都必须专注于某一件事,而且并没有受到周围环境的影响。

"破窗理论":它指的是那些窗户打破了的建筑,它会让别人觉得好像没有什么人在照顾它,所以其他人不会去关心。他们让他们的窗户不断地被破坏,最后他们还是参与到了破坏的活动中,在墙上进行了涂鸦和垃圾的堆积。一扇碎玻璃窗把这栋建筑物引到了一条枯烂的路。在程序中亦是如此。在一个地方可能会有错误的代码,这很有可能导致整个项目的代码被打乱或者破坏。因此,当你要闻到代码中的臭味时,你就需要进行重构。我认为干净代码是重构的。

美国《童子军规则》中有一条规定“让地面比你来时更干净”。它映射到我们的代码，我们应该确保每天提交代码时代码是干净的，否则我们将重构它。好的代码是一点一点积累起来的。问题发现得越早，修改的成本就越低。

糟糕的代码究竟是如何会直接影响一个项目的?试着想象一下一份仅仅10行的代码却充满了 i 、 j 、 k 等各类变量的代码,读完也没有什么困难,或许其中的逻辑也不那么明确,但是好歹还是很容易地把代码写明了作者的思路和示范意图,但如果把自己的代码改变成1000行呢?或许这个模型中所用代码的字符串数量已经扩展到100行的话很多情况下就已经开始会导致让很多人都开始想做选择去舍弃,随意地进行代码的命名也会导致让很多人一头雾水,错综复杂地进行逻辑性混合。若还要再次在这种编程语言中进行继续叠加逻辑,这几乎都被认为实际上是一项无法执行的任务。讲述完如何使用一个保持整洁的源代码,如果我们碰巧遇到一团混乱的源代码,我们又什么需要如何把其他程序整理得井井有条呢?首先我们必须是否需要一个能够明确的去发现和找到混乱的位置,当然我们也很有可能一下子就会发现自己所有,不用太着急,我们就必须需要做到像是剥了洋葱一样一点一点的重新去拆卸和修复程序中的代码,小幅编译程序的重新调试和修复。每次修改,都会自动带上前边已经提及过的测试结果,以便于保证我们没有伤害现在的系统功能。当我们还是需要开始自己整理一大坨代码的时候,我自己的实践经验也就是,如果不知道怎么做一个改动的话时候,我们最好是可以先从最基本的一个角度出发入手,看懂一段简单的代码并且清晰而明确这个改动是在我们已经把自己做了一件好的事情之后,把其中一些代码都提取了起来,取一个很重要的一个具有表达和描述性质的名称。所以当将代码这样拆分成一个一个的编程方法后,结构就清晰了很多。之后我们会闻到许多码 smell ,发现许多的是重复。关于如何在网络上发现和使用 code smell ,可以通过一些参考《重构》一书,该系统中列举了许许多多的 code smell ,并且为大家提供了很好的解决办法。

根据我自己的经验，我几乎记不起代码的具体实现，尽管这是我编写的代码。每次我想自己开发一个新函数时,我都会想熟悉老代码的具体实现逻辑,然后我可以在它上面重新叠加一个新函数。一个简单的代码可以使得阅读它的程序员迅速理解其实现逻辑,而不会知道具体的实现过程和操作细节。如果一段代码需要求一个程序员充分理解所有代码的实现细节以及理解其代码的功能,那么它必然会导致程序员在编写过程中需要花费大量的时间来阅读一段和工作没有关联的代码。这样的坏代码投入得越多,程序员们所花费的精力和时间也就会越多,这正是为什么我在第一篇文章的开始说,坏代码对整个项目的影响往往是与整个项目的规模形成指数相关的。

正如在同一本书中提到的，整洁的代码似乎是多余的，但是这个动作确实是以后提高效率的关键。事实上，有时人们的迁就和拒绝会导致代码混乱。如果我们能在一开始就做好这些事情，我们就可以避免以后的重复工作。至少在基本层面上，代码可以是整洁的。虽然它不能像书中所说的那样好，但它对将来提高效率仍然有很大的帮助。