软件构造

Outline

- 概述
- 活动
- 实践方法
- Construction Idea

什么是软件构造?

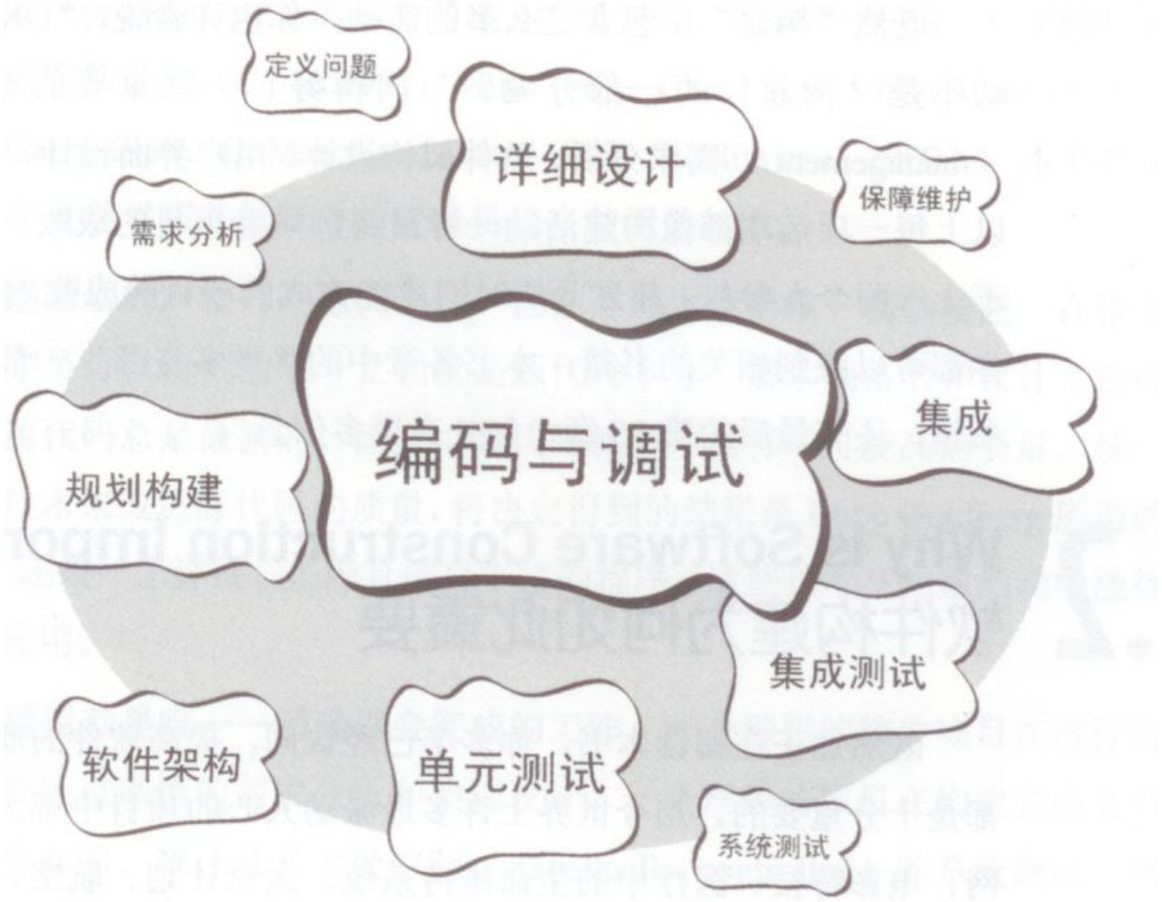
Construction is not Implementation

- Distinction Between Activities and Phases
  - Activity != Phase
  - Talking about “Construction” as an activity does not imply a distinct phase
  - Differentiating between kinds of activities is extremely helpful

[SWEBOK2004]

- 将“软件构造”定义为：通过编码、验证、单元测试、集成测试和调试等工作的结合，生产可工作的、有意义的软件的详细创建过程。

[McConnell2004]



认为软件构造除了核心的编程任务之外，还涉及详细设计（数据结构与算法设计）、单元测试、集成与集成测试、以及其他活动。

软件构造是设计的延续

- [Reeves1992]首先提出要仔细区分设计与实现的界限：
  - 设计是规划软件构建方案的过程，实现是依据规划的软件构建方案建造真正产品的过程；
  - 源程序是软件构建方案的最后一个规划，不是产品本身，真正的产品是运行于计算机上的由二进制代码组成的可执行程序；
  - 源程序的生产过程——编程，属于设计活动，编译器完成的编译和链接才是依据规划建造软件产品的实现活动。

Outline

- 概述
- 活动
- 实践方法
- Construction Idea

软件构造活动

- 详细设计；
- 编程；
- 测试；
- 调试；
- 代码评审；
- 集成与构建（Build）；
- 构造管理。

详细设计

- 有些项目会将主要的详细设计工作分配在软件构造阶段完成
- 不论是哪种项目，在软件构造阶段都不可避免的会涉及到详细设计的调整工作。因为编程语言是软件设计的一个重要约束，随着编程工作的进行和深入，人们可能会发现与预想不一致的情况和更多的约束，这个时候就需要在软件构造阶段修改详细设计方案。
- 软件构造阶段详细设计使用的方法与技术与软件设计阶段是一样的，只是应用在更小的规模上。

程序代码的典型质量

- 易读性。
    - 程序代码必须是易读的，看上去“显而易见是正确的”。易读性是编程最为重要的目标，它可以使得程序更容易开发，尤其是易于调试；可以使得程序更容易维护，减少理解代码的难度和成本；可以使得程序易于复用。
- 易维护性。
    - 除了易读之外，易维护性要求程序代码易于修改。
- 可靠性。
    - 程序代码必须是可靠的，要执行正确，并妥善处理故障。
- 性能。
    - 程序代码必须是高性能的，包括时间性能和空间性能，需要进行仔细的数据结构和算法设计。
- 安全性。
    - 不要遗留程序漏洞，不要出现重要信息的泄漏（例如内存数据区泄漏）。

编程的主要技术

- 构造可理解的源代码的技术，包括命名和空间布局；
- 使用类、枚举类型、变量、命名常量和其它类似实体；
- 使用控制结构；
- 处理错误条件——既包括预计的错误，也包括未预期的异常；
- 预防代码级的安全泄露（例如，缓冲区超限或数组下标溢出）；
- 使用资源，用互斥机制访问串行可复用资源（包括线程和数据库锁）；
- 源代码组织（组织为语句、例程、类、包或其它结构）；
- 代码文档；
- 代码调整。

测试

- Inspection and testing is concerned with establishing the existence of defects in a program
- 通常来说，程序员每修改一次程序就会进行最少一次单元测试，在编写程序的过程中前后很可能要进行多次单元测试，以证实程序达到了要求，没有程序错误。集成测试一般在单元测试之后，用来测试多个单元之间的接口是否编程正确。

调试

- Debugging is concerned with locating and repairing these errors
- 调试过程可以分为三个部分：重现问题、诊断缺陷和修复缺陷。

重现问题的方法

- 控制输入。
    - 找到相应的数据输入，能够重现绝大多数的问题。可以通过控制数据输入来重现问题意味着缺陷就发生在对该数据的处理代码之中。
    - 寻找能够重现问题的数据输入可以使用问题回溯推理、内存数据监控、记录输入数据日志等方法。
- 控制环境。
    - 有些问题是编译器、操作系统、数据库管理系统、网络管理系统等系统软件环境造成的，通过控制数据输入无法重现问题。这时就需要通过控制环境来重现问题。一定要记住的是，如果你进行各种手段诊断之后确信你的程序代码没有缺陷，就要警惕可能是软件环境造成了问题。
    - 控制环境以重现问题经常使用替换法，例如替换机器、操作系统、数据库管理系统等。

寻找和定位缺陷的方法

- 灵活使用编译器提示。
- 持续缩小嫌疑代码的范围。
- 检查刚刚修改过的部分。
- 警惕已出现缺陷和常见缺陷。
- 利用工具。

常见错误

- 内存或资源泄漏；
- 逻辑错误；
- 编码错误（例如条件判断不够充分）；
- 内存溢出（超出本身限制）；
- 循环错误（死循环或数目不合适）；
- 条件错误；
- 指针错误（超出范围，未赋值）；
- 分配释放错误（分配两次、未分配即释放、释放两次、分配未释放）；
- 多线程错误（同步）；
- 定时错误（没有考虑特殊情况）；
- 存储错误（考虑磁盘已满，文件不存在等特例）；
- 集成错误（相互之间的考虑不相容）；
- 转换错误（字符转换等出现问题）；
- 硬编码长度/尺寸；
- 版本缺陷（对以前的不兼容）；
- 不恰当重用带来的缺陷。

修复缺陷的注意点

- 一次只修复一个缺陷。
- 修改前保留旧版本的备份，如果项目使用了配置管理系统，这个工作会由配置管理工具完成，否则就需要由程序员手动完成。
- 使用测试和评审验证修复的有效性。
- 检查和修复类似的缺陷，这可以在代码搜索、程序切片等工具的帮助下进行。

代码评审

- 代码评审对代码的系统检查，通常是通过同行专家评审来完成的。通过评审会议可以发现并修正之前忽略的代码错误，从而同时提高软件的质量和开发者的技巧。
- 代码评审一般分为正式评审、轻量级评审和结对编程。

实践经验

- 就算不能评审全部的代码，最少也要评审一部分（20–33%）代码，以促使程序员编写更好的代码。
- 一次评审少于 200–400 行的代码。
- 目标为每小时低于 300–500 LOC 的检查速率。
- 花足够的时间进行正确缓慢的评审，但是不要超过 60–90 分钟/每次。
- 确定代码开发者在评审开始之前就已经注释了源代码。
- 使用检查列表，因为它可以极大地改进代码开发者和评审者的工作。
- 确认发现的缺陷确实得到修复了。
- 培养良好的代码评审文化氛围，在这样的氛围中搜索缺陷被看做是积极的活动。
- 采用轻量级，能用工具支持的代码评审。

集成与构建

- 在以分散的方式完成程序基本单位（例程、类）之后，软件构造还需要将这些分散单位集成和构建为构件、子系统和完整系统。
- 集成有大爆炸式集成和增量式集成两种方式。实践中增量式集成有着更好的效果。
- 构建将可读的源代码转换为标准的能在计算机上运行的可执行文件。构建过程需要配置管理工具的帮助。

构造管理

- 构造计划
- 度量
- 配置管理

Outline

- 概述
- 活动
- 实践方法
- Construction Idea

实践方法

- 重构
- 测试驱动开发
- 结对编程

重构

- 为什么要重构?
    - 1、因为无法预计到后续数年的修改，导致软件开发阶段的设计方案不能满足修改要求；
    - 2、随着修改次数的增多，软件设计结构的质量越来越脆弱，很难继续维持可修改性。
- 什么是重构?
    - 修改软件系统的严谨方法，它在不改变代码外部表现的情况下改进其内部结构。
- 重构的时机?
    - 增加新的功能时。需要注意的是重构发生在新功能增加完成之后，用来消除新功能所添加代码导致的坏味道；而不是发生在新功能添加之前，重构不改变代码外部行为，不是能够实现新功能添加的方法。
    - 发现了缺陷进行修复时。诊断缺陷时如果发现代码存在坏味道或者修复代码会引入坏味道，就需要进行重构。
    - 进行代码评审时。如果在评审代码时发现了坏味道，就需要进行重构。

代码的坏味道

- 太长的方法，往往意味着方法完成了太多的任务，不是功能内聚的，需要被分解为多个方法。[McConnell2004]认为如果方法代码长度超过了一个屏幕，就需要留心注意了。
- 太大的类，往往意味着类不是单一职责的，需要被分解为多个类。
- 太多的方法参数，往往意味着方法的任务太多或者参数的数据类型抽象层次太低，不符合接口最小化的低耦合原则，需要将其分解为多个参数少的方法或者将参数包装成对象、结构体等抽象层次更高的数据类型。
- 多处相似的复杂控制结构，例如多处相同类型的Case结构，往往意味着多态策略不足，需要使用继承树多态机制消除复杂控制结构。
- 重复的代码，往往意味着隐式耦合，需要将重复代码提取为独立方法。 一个类过多使用其他类的属性，往往意味着属性分配不正确或者协作设计不正确，需要在类间转移属性或者使用方法委托代替属性访问。
- 过多的注释，往往意味着代码的逻辑结构不清晰或者可读性不好，需要进行逻辑结构重组或者代码重组。

```
public class Member {
    private long id;
    private String name;
    private long point;
    ...
    public void update(){
        MapperService memberMapper = MemberMapper.getInstance();
        memberMapper.update(id, name, point);
    }
}
```

坏味道的代码

```
public class Member {
    private long id;
    private String name;
    private long point;
    …
    public void update(){
          MapperService memberMapper = MemberMapper.getInstance();
          MemberPO po = new MemberPO(id,name,point);
          memberMapper.update(po);
    }
}
```

```
public class MemberPO implements Serializable {
    private long id;
    private String name;
    private long point;
    public MemberPO(long i,String n, long p){
          id = i;
          name = n;
          point = p;

    }
    …
}
```

重构后的代码

测试驱动开发

- 测试驱动开发又被称为测试优先（Test First）的开发，随着极限编程方法的普遍应用而得到普及。
- 测试驱动开发要求程序员在编写一段代码之前，优先完成该段代码的测试代码。测试代码通常由测试工具自动装载执行，也可以由程序员手工执行。完成测试代码之后，程序员再编写程序代码，并在编程中重复执行测试代码，以验证程序代码的正确性。

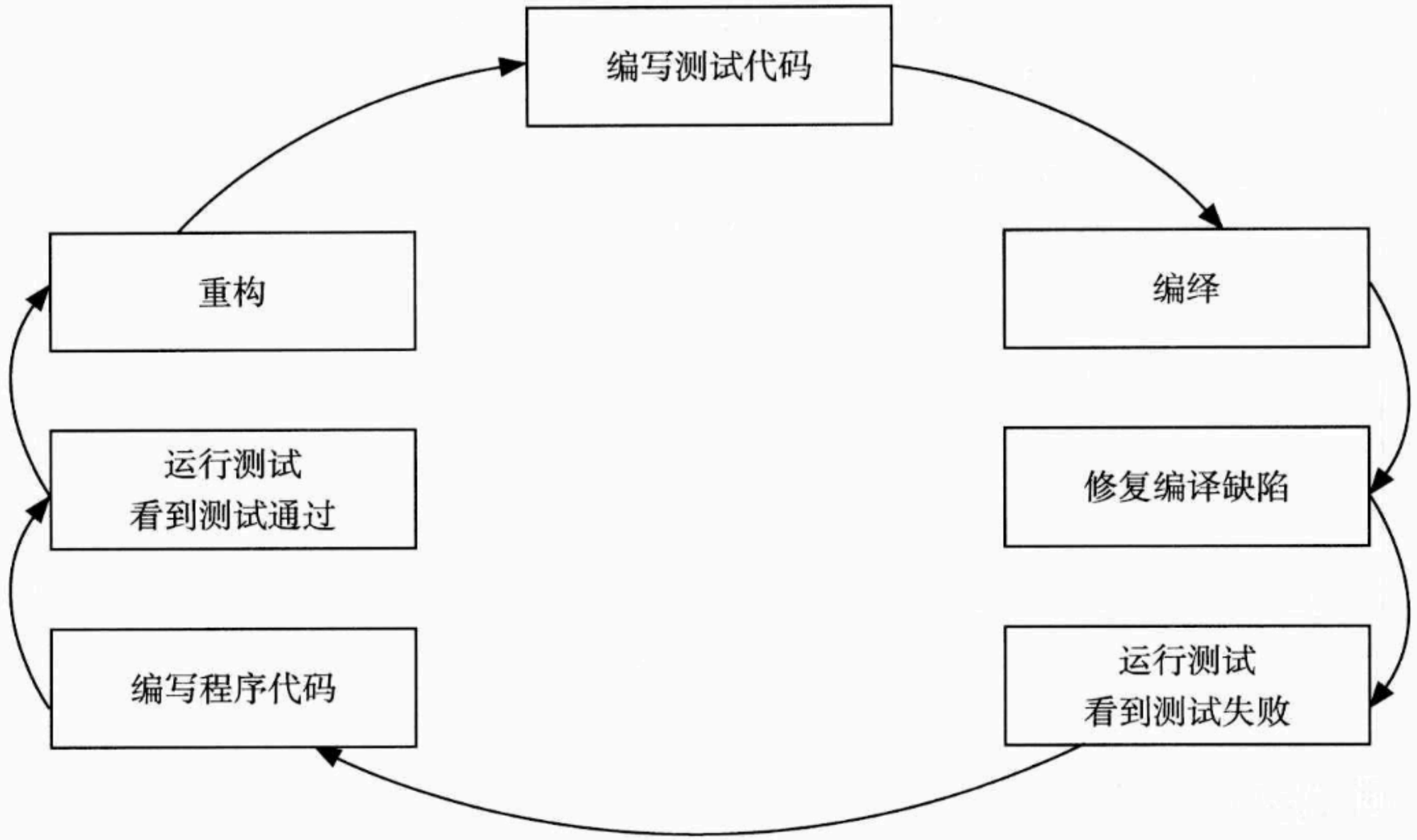


图 17-5　测试驱动开发过程

过程

表 17-1　getChange 方法的前置条件和后置条件

方法声明	public double getChange(double payment)
前置条件	payment>0 payment>= Sales.total
后置条件	return= payment− Sales.total

示例

表 17-2　getChange 方法的测试用例

ID	测试目的	输　　入	预期输出
1	参数不合法时异常	设置 Sales.total=90, payment=−100	异常：payment 应该大于 0；输出为 −1
2	参数不合法时异常	设置 Sales.total=90, payment=0	异常：payment 应该大于 0；输出为 −1
3	参数与 Sales 状态联合起来异常	设置 Sales.total=90, payment=50	异常：payment 应该大于 Sales.total；输出为 −2
4	正常功能	设置 Sales.total=90, payment=90	返回 0
5	正常功能	设置 Sales.total=90, payment=100	返回 10

```
public class GetChangeofSalesTester {
        public static void main(String[] args){
            Sales sale=new Sales();
            // 购买 2 个 ID=1 的商品，该商品的测试数据 Price=45
            sale.addSalesLineItem(1, 2)
            sale.total();
            double change;
            boolean passed=true;
            change= sale.getChange(-100);          //测试用例 1
            if (change!=-1 ) {
                passed=false;
                System.out.println("parameter exception not handled");
            }
            change= sale.getChange(0);             //测试用例 2
            if (change!=-1 ) {
                passed=false;
                System.out.println("parameter exception not handled");
            }
            change= sale.getChange(50);            //测试用例 3
            if (change!=-2 ) {
                passed=false;
                System.out.println("parameter&&attribute exception not handled");
            }
            change= sale.getChange(90);            //测试用例 4
            if (change!=0 ) {
                passed=false;
                System.out.println("when parameter=attribute, error");
            }
            change= sale.getChange(100);           //测试用例 5
            if (change!=10 ) {
                passed=false;
                System.out.println("when parameter>attribute,error");
            }
            if (passed==true)                      //所有测试用例通过
               System.out.println("None error found");
        }
}
```

```
@RunWith (Value = Parameterized.Class)
public class SalesTester {
        private double payment;
        private double change;

        @Parameters
        public static Collection<Double[]> getTestParameters (){
              return Array.asList ( new Double [] [] {
                                                    //payment, change
                                                      {-100, -1}, //测试用例 1
                                                      {0, -1}, //测试用例 2
                                                      {50, -2}, //测试用例 1
                                                      {90,0}, //测试用例 1
                                                      {100, 10}, //测试用例 1
                                                });
        }
        Public ParameterizedTest ( double payment, double change ){
              this.payment = payment;
              this.change = change;
        }


        @Test
        public void testChange () {
              Sales sale=new Sales();
              // 购买 2 个 ID=1 的商品， 该商品的测试数据 Price=45
              sale.addSalesLineItem(1, 2)
              sale.total();

              assertEquals (change, sale.getChange (payment) );
         }
}
```

结对编程

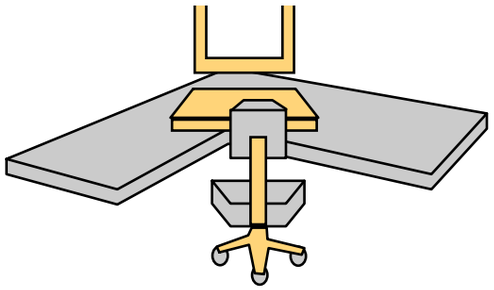
Pair Programming

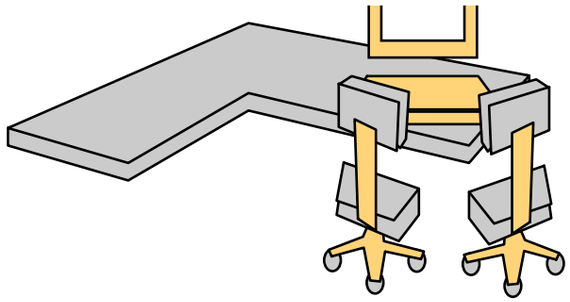
- Two programmers working side-by-side, collaborating on the same design, algorithm, code or test
- One programmer, the driver, has control of the keyboard/mouse and actively implements the program
- The other programmer, the observer, continuously observes the work of the driver to identify tactical (syntactic, spelling, etc.) defects and also thinks strategically about the direction of the work
- On demand, the two programmers can brainstorm any challenging problem
- The two programmers periodically switch roles, they work together as equals to develop software

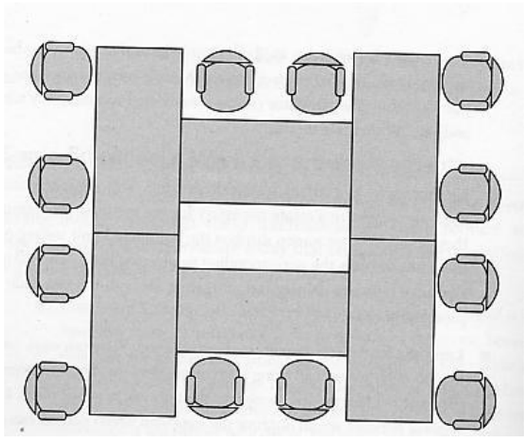
How does it work

- Pair-Pressure
    - Keep each other on task and focused
- Pair-Think
    - Bring different prior experiences to the task
- Pair-Relaying
    - Each, in turn, contributes to the best of their knowledge and ability Then, sit back and think while their partner fights on

- Pair-Reviews
    - Continuous design and code reviews
    - Ultimate in defect removal efficiency
    - Removes programmers distaste for reviews
        - 80% of all (solo) programmers don't do them regularly or at all
    - Defect prevention always more efficient than defect removal
- Pair Debugging
    - Talking about problem in a pair can lead to a solution becoming obvious
- Pair-Learning
    - Continuous reviews -> learn from partners techniques, knowledge of language, domain, etc.







Workplace Layout

Partner Selection

Pair Rotation

- Ease staff training and transition
- Knowledge management/Reduced product risk
- Enhanced team building

Outline

- 概述
- 活动
- 实践方法
- Construction Idea

Construction Ideas

- A Decade of Advances in Software Construction
- Ten Realities of Modern Software Construction
- Some of the Worst Construction Ideas of 1990s and 2000s

Construction Ideas

- A Decade of Advances in Software construction

1. Design has Been Raised a Level

- Programming has advanced through ability to create larger code aggregations
    - Statements
    - Routines
    - Classes
    - Packages
- Real legacy of OO might well be larger aggregations

2. Daily Build and Smoke Test

- Institutionalizes incremental integration
- Minimizes serious integration problems that used to be common
- Lots of other benefits, too

3. Standard Libraries

- Good programmers have always used libraries
- Now provided with languages (Java, C++, .NET)

4. Visual Basic

- Visual programming innovation
- The first development environment to make widespread use of COTS components
- Only language to learn Ada's syntax lessons (case statements, control statements, etc.)
- Highly integrated environment

5. Open Source Software

- Great aid to programmers during development
- Reduced barriers to making code available
- Opportunity to learn from available code
- Improved ability to read code
- Nice “community” of programmers

6. The Web, for Research

- FAQs
- Discussion groups
- Searchability in general

7. Widespread Use of Incremental Development

- Concepts were well known in 1990s
- Practice is well established in 2000s

8. Test-First Development

- Shortens time to defect detection
- Increases personal discipline
- Complements daily build & smoke test

9. Refactoring as a Discipline

- Provides a discipline for making changes
- Not so good as a total design strategy
- Good example of incrementalism

10. Faster Computers

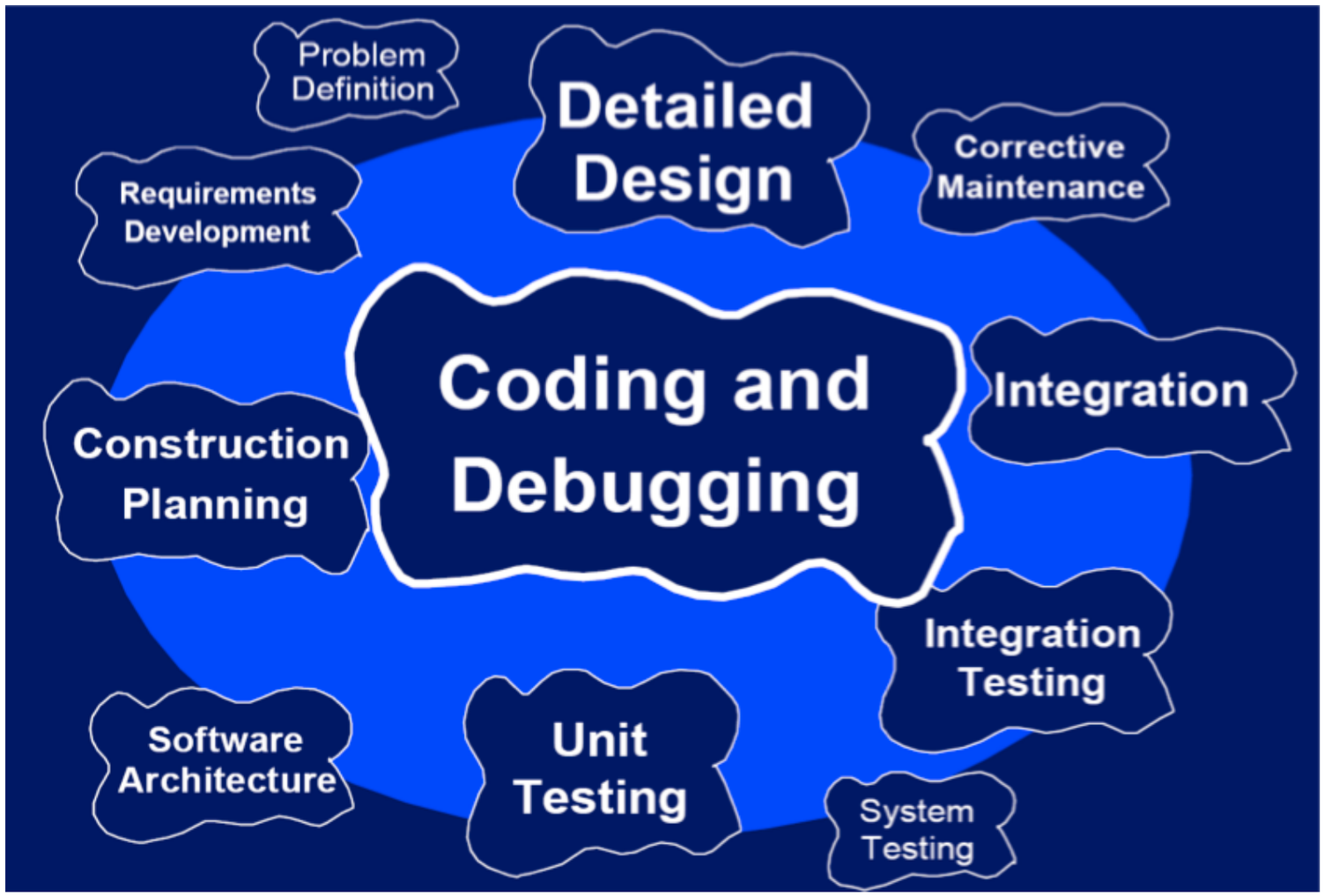
- Implications for optimization
- Implications for programming languages
- Implications for development

Construction Ideas

- Ten Realities of Modern Software Construction

-1-"Construction" is a Legitimate Topic

- Software "Construction" – Now Looks Like This



-2- Individual Variation Is Significant

- Where do Variations Exist?
- Researchers have found variations ranging from 10x to 28x in:
    - Coding speed
    - Debugging speed
    - Defect-finding speed
    - Percentage of defects found
    - Bad-fix injection rate
    - Design quality
    - Amount of code generated from a design
    - Etc.

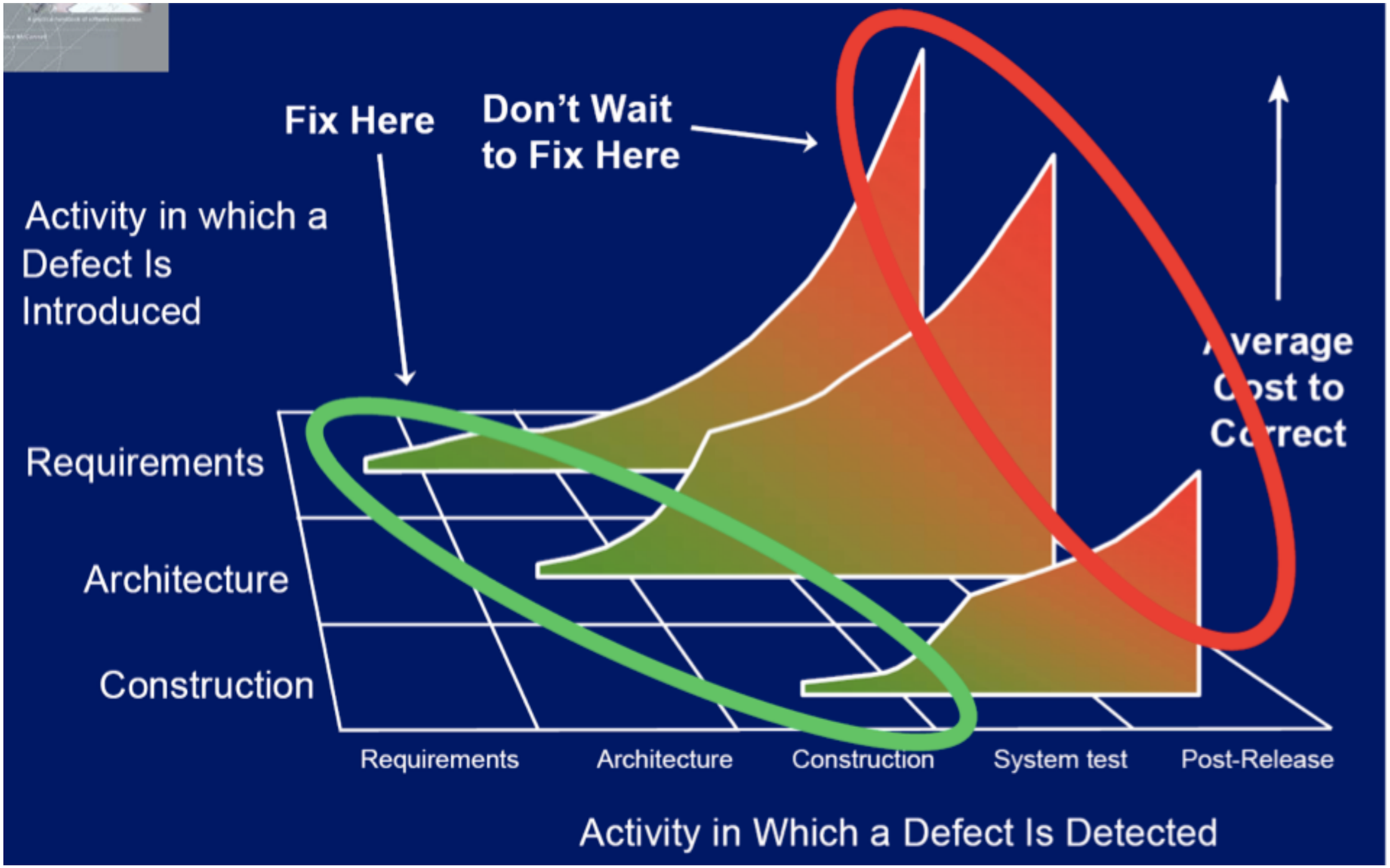
-3- Personal Discipline Matters

- Why Personal Discipline Matters
    - Being realistic about predicting the future
    - Areas where discipline matters
        - Refactoring
        - Prototyping
        - Optimization
        - Minimal-complexity designs specifically
        - Managing complexity generally
- Endpoints—Discipline and Courage
    - Humphrey on PSP
    - Beck on Extreme Programming

-4- A Focus on Simplicity Works Better than a Focus on Complexity

- Focus on read-time convenience, not write-time convenience

-5- Defect-Cost Increase is Alive and Well



-6- Importance of Design

- There are lots of valid points on the "no design"—"all design" continuum
- General Point: Extremes are Usually Not Productive
    - All design up front vs. no design up front
    - Entirely planned vs. entirely improvised
    - Pure iterative vs. straight sequential
    - All structure vs. all creative
    - Document everything vs. document nothing

-7- Technology Waves Affect Construction Practices

- Effect of Technology Waves on Construction
    - Definition of "technology wave"
        - Early-wave characteristics
        - Mature-wave characteristics
        - Late-wave characteristics
- Construction is affected by technology—more than I thought (doh!)
- Technology can be addressed in terms of general principles

-8-Incremental Approaches Work Best

- Perspective on Incrementalism
    - The pure waterfall model is not at all incremental or iterative—which is why it hasn't worked very well
    - Spiral development is highly incremental and iterative, which is part of why it does work well
    - All projects will experience iteration at some point
    - Think about where and when in your project you will get your incrementalism—cheaply, or expensively?

-9- The Toolbox Metaphor Continues to be Illuminating

- Toolbox Metaphor
    - What's best? Agile? XP? Scrum? CMM?
    - Toolbox explains there's no one right tool for every job
    - Different industry segments will have different tools and even different toolboxes
- What's in the Software Engineering Toolbox?
    - Best practices
    - Lifecycle models
    - Templates, checklists, patterns, examples
    - Software tools

-10- Software's Essential Tensions

- Software's essential tensions have remained unchanged for years:
    - Rigid plans vs. Improvisation
    - Planning vs. Fortune Telling
    - Creativity vs. Structure
    - Discipline vs. Flexibility
    - Quantitative vs. Qualitative
    - Process vs. Product
    - Optimizing vs. Satisficing
- Balance wavers, but basic tensions are constants

Construction Ideas

- Some of the Worst Construction Ideas of 1990s and 2000s

Worst Ideas, 1990s vs. 2000s

1990s	2000s
❖ Code & fix	❖ Code & fix
❖ "All design up front" programming	❖ "No design up front" programming
❖ Design for speculative requirements	❖ Planning to refactor later
❖ Components will solve all our construction problems	❖ Offshore outsourcing will solve all our problems
❖ Automatic programming	❖ Automatic programming
❖ Uninformed use of the waterfall model	❖ Uninformed use of Extreme Programming
❖ Calling everything "object oriented"	❖ Calling everything "agile"