

极限编程

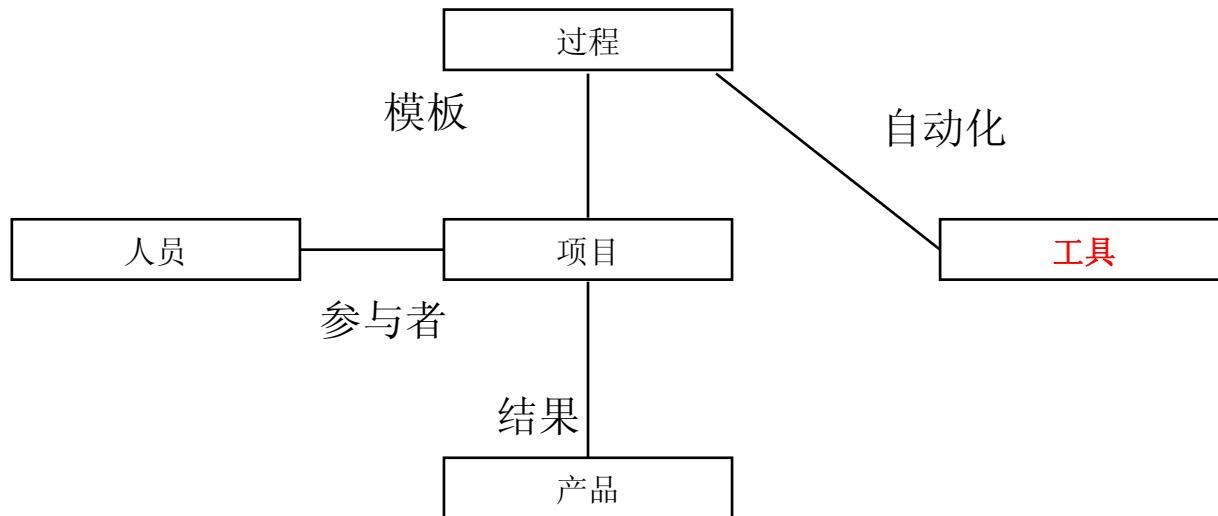
(Extreme Programming)



-yf@itechs.iscas.ac.cn

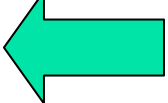


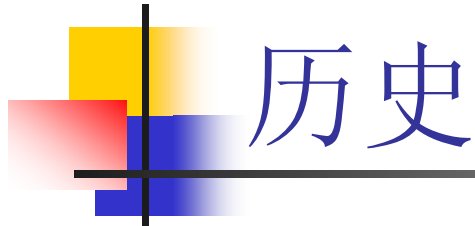
软件开发的要素





OUTLINE

- 历史 
- 定义
- 内容
- 应用
- 讨论（设计已死? etc.）
- XP的未来



■ Kent Beck -XP

- 1996, Chrysler, C3项目
- 1997 , 《 Extreme Programming Explained – Embrace Change 》

■ 17个大牛—Agile

- 2001.2, Utah, Agile Software Development Alliance
- XP, SCRUM, DSDM , ASDD, Crystal ,FDD.....



敏捷宣言 (<http://www.agilemanifesto.org/>)

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

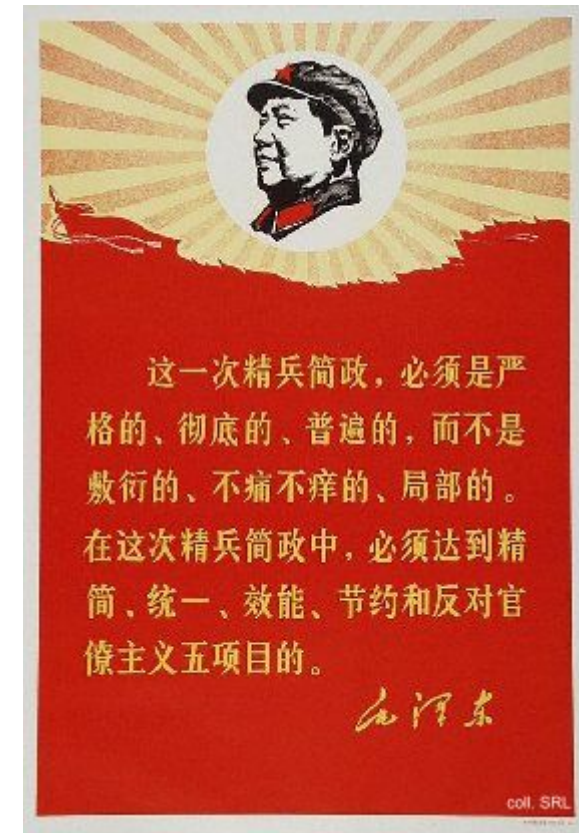
Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler

James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick

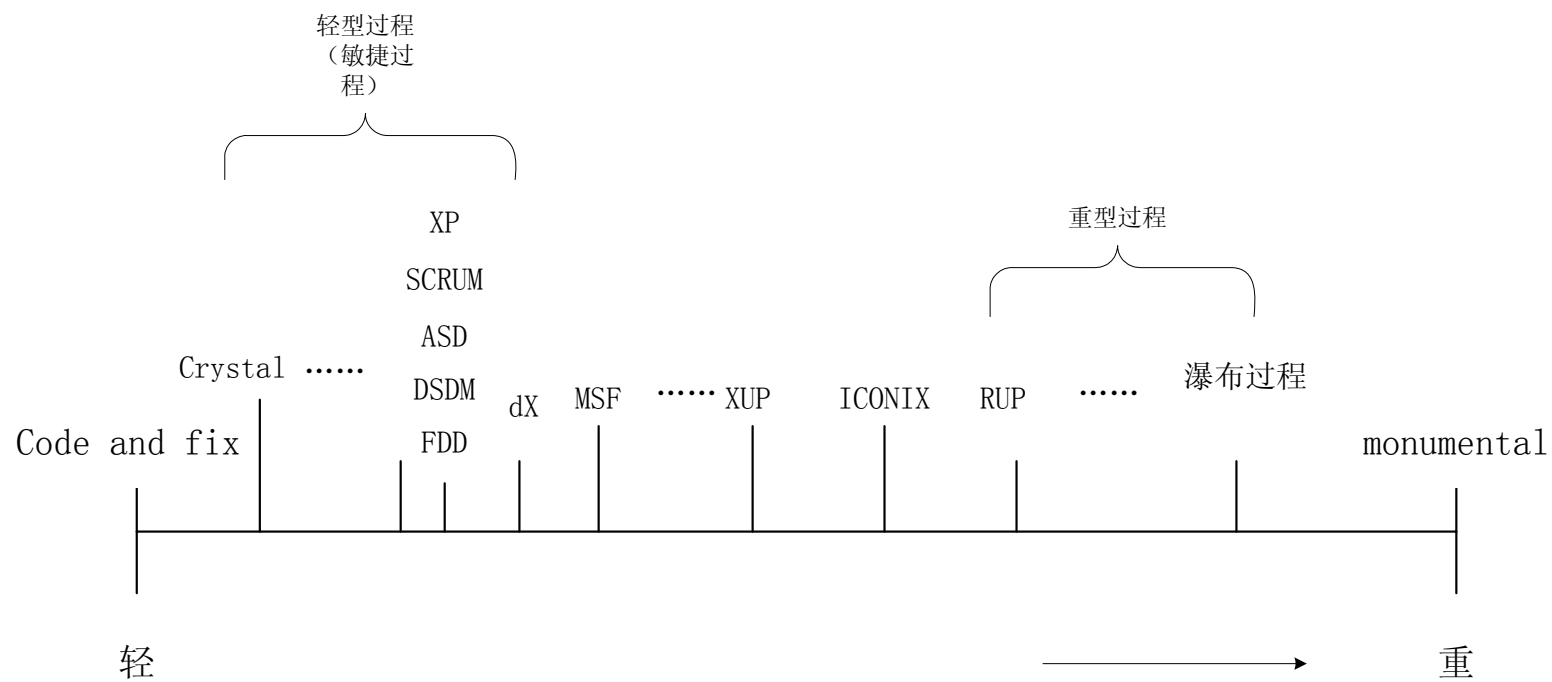
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

历史的车轮

- 软件危机
 - -> 软件工程
 - -> 重型工程不能承受之变
- **From Nothing,
to Monumental,
to Agile**
 - <new methodology>



过程





对变化的不同态度

- 将变化遏制在摇篮中

---- 重型工程支持者

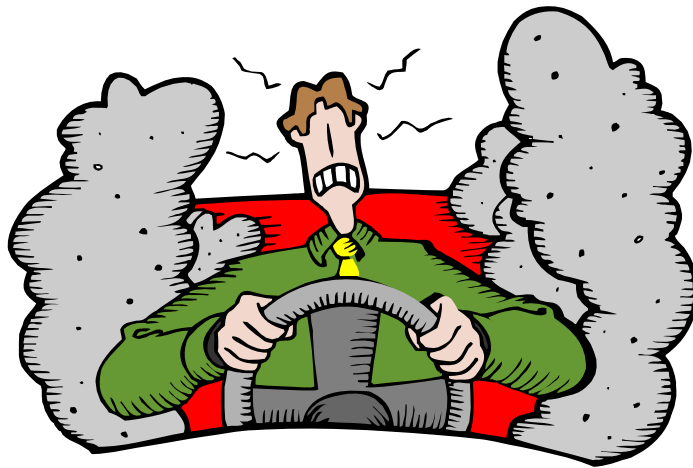
- 拥抱变化

---- XP支持者

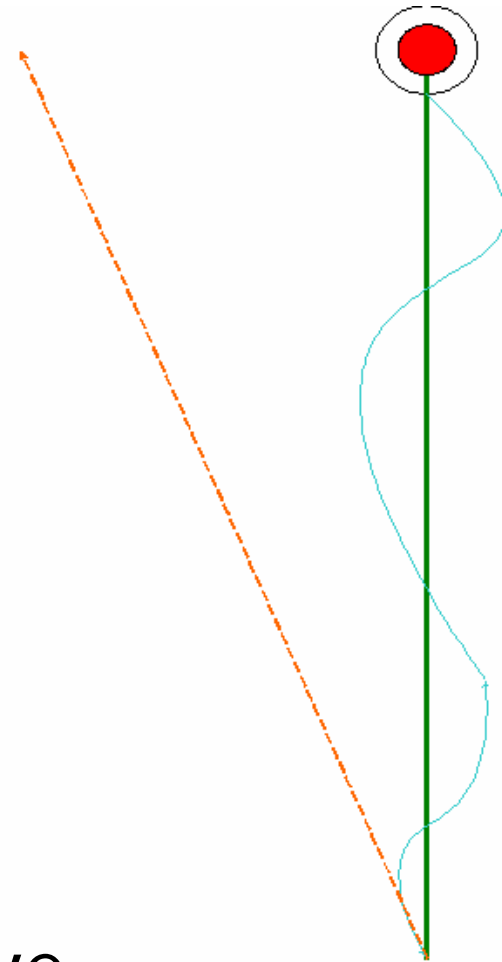


软件开发之路

直中靶心 **vs** 随时调整

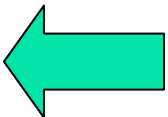


Adaptive, **NOT** Predictive





OUTLINE

- 历史
- 定义 
- 内容
- 讨论（设计已死?etc.）
- XP的未来



What is XP?

- 一种开发过程？
- 一种方法论？
- 一系列的最佳实践（best practice）？



What is XP?

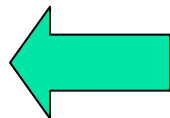


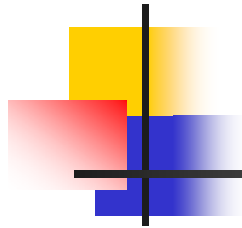
- Extreme Programming (XP) is actually a deliberate and disciplined approach to software development.
- XP is successful because it stresses customer satisfaction.
- 更多: <http://www.extremeprogramming.org/what.html>



OUTLINE

- 历史
- 定义
- 内容
- 讨论（设计已死? etc.）
- XP的未来

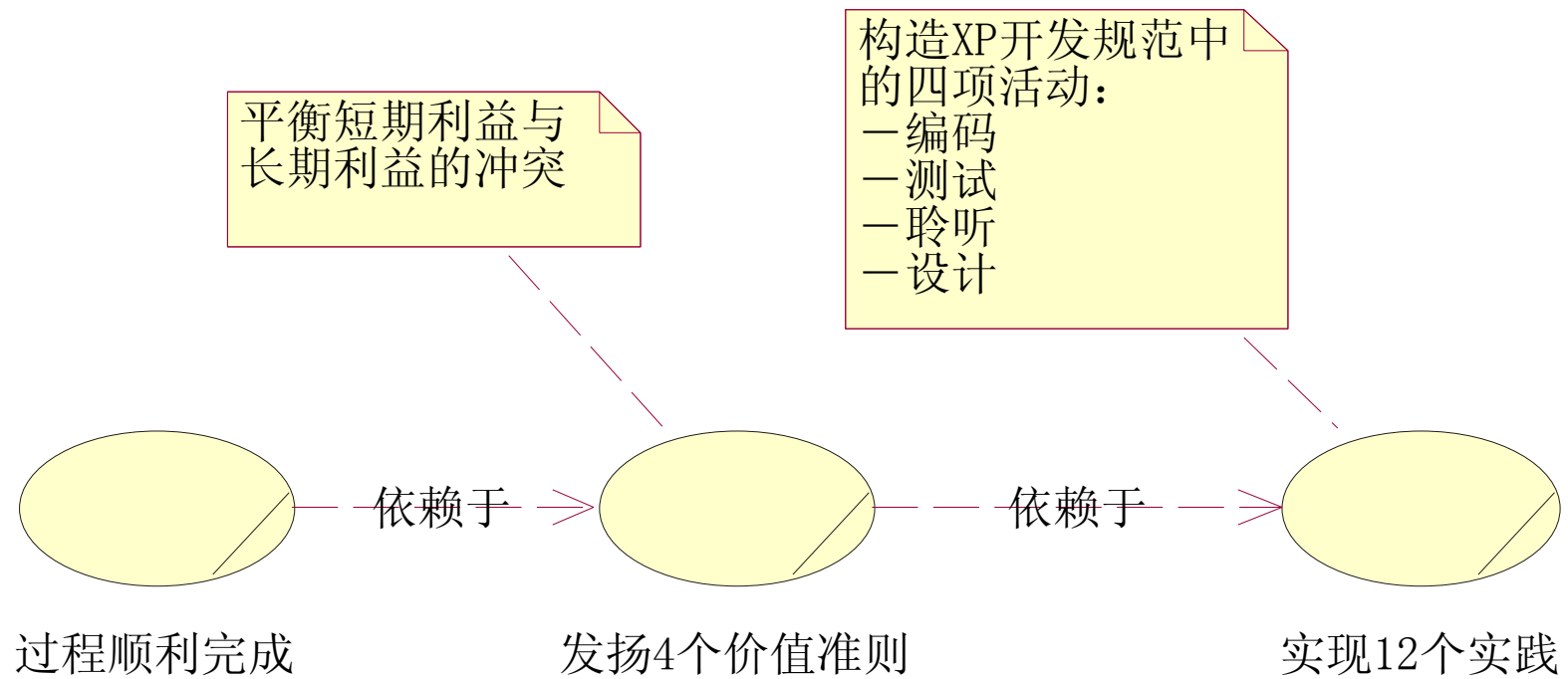




XP的口号

- Do The Simplest Thing that Could Possibly Work
- YAGNI(You Aren't Going to Need It)
- Once and Only Once
- DRY(Don't Repeat Yourself)

XP的内容

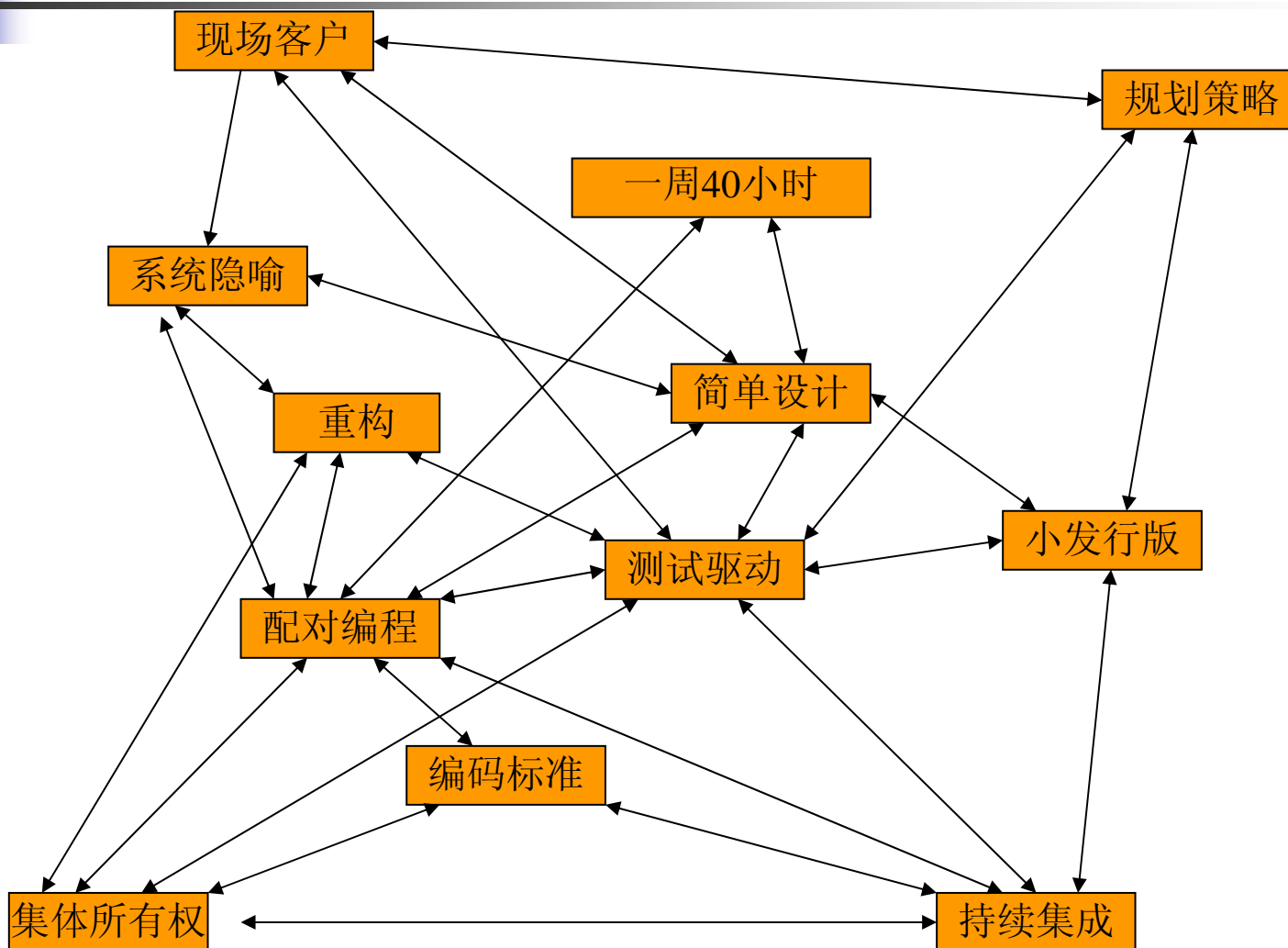




12个核心实践

- 计划博弈(Planning game)
- 成对编程(pair programming)
- 系统隐喻(System Metaphor)
- 测试驱动(Test-driven)
- 代码重构 (Refactoring)
- 小型发布(Small releases)
- 简单设计(Simple design)
- 集体代码所有权(Collective ownership)
- 持续集成(Continuous integration)
- 现场客户(On-site customer)
- 代码规范(Coding standards)
- 每周40小时工作制 (40-hour week)

实践之间的互相支持





计划博弈 (Planning Game)

- 同样有计划，随时改变计划
- 结合项目进展和技术情况，确定下一阶段要开发与发布的系统范围。
- Adaptive



成对编程 (Pair Programming)

- 浪费 Vs 提高效率

- Martin Fowler 说:

当别人说PP会降低生产力时，我回答，如果大部分编程的过程都是在打字的话，那么PP确实会降低效率。



PP的好处

- 所有设计决策都牵涉到至少两个人。
- 至少有两个人熟悉系统的每一部分。
- 几乎不可能出现两个人同时疏忽测试或其它任务。
- 改变各对的组合在可以在团队范围内传播知识。
- 代码总是由至少一人复查。
- 心理学..... ;)



PP的变种

- Pair Solve
- Pair Debug
- 赶场的技术高手



系统隐喻 (System Metaphor)

- XP通过一个简单的关于整个系统如何运作的隐喻性描述（**story**）来知道全部开发。
- 隐喻可以看作是一种高层次的系统构想。通常包含了一些可以参照和比较的类和模式，它还给出了后续开发所使用的命名规则。XP不需要事先进行详细地架构设计。



系统隐喻 (System Metaphor)

- Xpers认为这就是系统的架构.....
- “不断地细化架构” -----Kent Beck
- 但并不是推迟架构设计的时机。
- Is it enough?



测试驱动 (Test-driven)

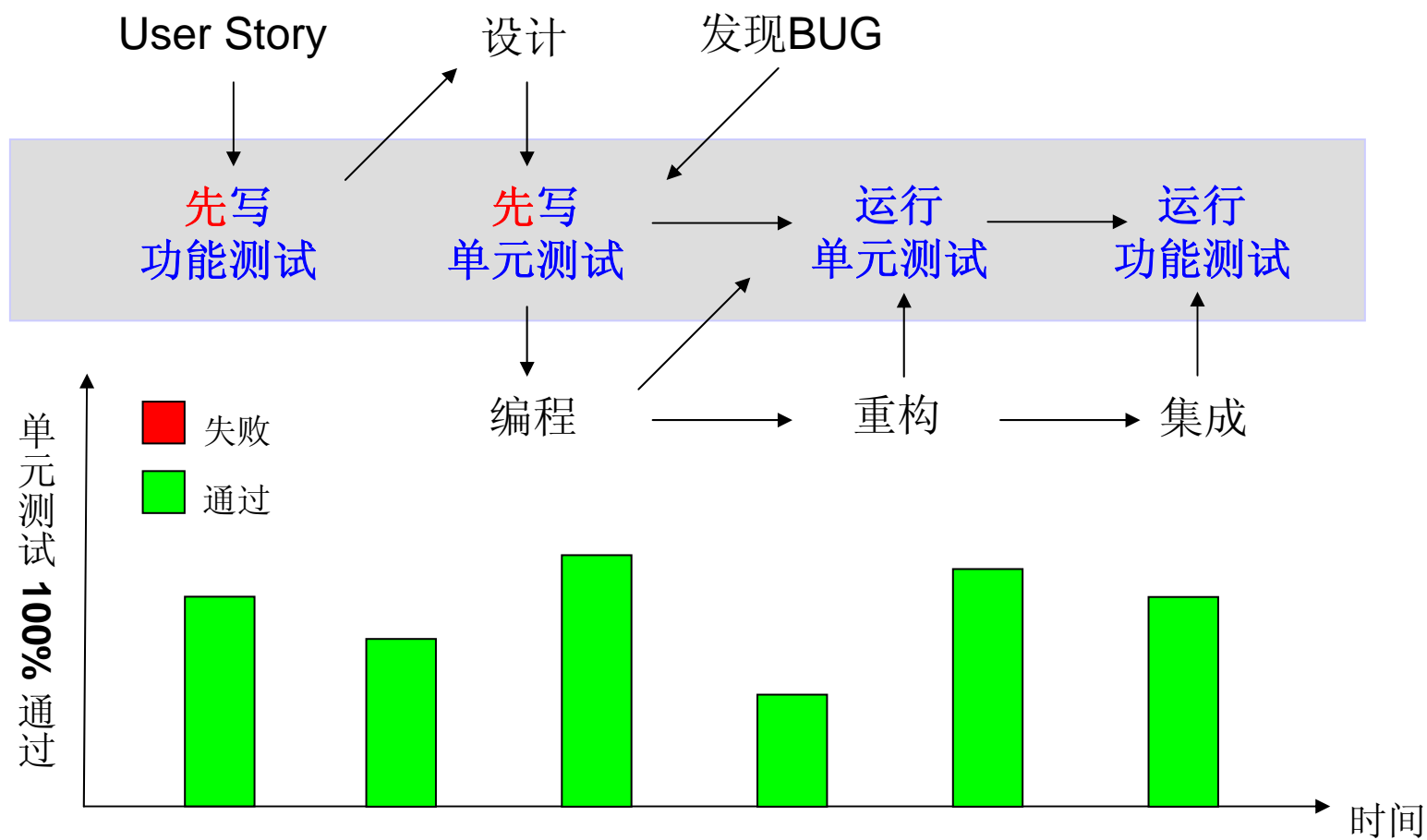
- 先测试后编码
- 开发人员必须保证单元测试和集成测试始终运行无误，甚至现场客户代表也要能够编写功能测试程序。
- 无论怎么强调测试的重要性都不为过。



测试驱动 (Test-driven)

- XP 有两种测试方法:
 - 1. 单元测试
 - 2. 验收测试(满意度测试)
- 研发人员在编程的同时也撰写单元测试。
- 顾客在确定功能之后就撰写验收测试。

测试驱动开发





代码重构 (Refactoring)

- 重构是在不更改功能性的前提下对代码加以改进。XP小组在进行重构时毫不手软。
- 开发人员重构有两个重要时机：实现特性之前和之后。开发人员尝试确定更改现有代码是否可以让新特性的开发更容易。他们查看刚刚写好的代码，看是否有方法可以对它进行简化。例如，如果他们认为有抽象的机会，就会进行重构来从具体实现中除去重复的代码。



代码重构 (Refactoring)

- 重构不是XP所特有的行为，在任何的开发过程中都可能并且应该发生。
- 重构就是设计
 - 正： 重构改善代码结构.....在改进中实现设计
 - 反： 重构不能取代设计。



小型发布 (Small Release)

- 非常短的周期内以递增的方式发布新版本，从而可以很容易地估计每个迭代周期的进度，便于控制工作量和风险；同时，也可以及时处理用户的反馈。
- 小型发布突出体现了敏捷方法的优点。**RUP**强调迭代式的开发，对于系统的发布并没有作出过多的规定。用户在提交需求后，只有在部署时才能看到真正的系统，这样就不利于迅速获得用户的反馈。如果能够保证测试先行、代码重构、持续集成等最佳实践，实现小型发布也不是一件困难的事情，在有条件的组织可以考虑使用。



简单设计 (Simple Design)

- 代码的设计应该尽可能的简单，只要满足当前功能的要求，不多也不少。
- **XP** 说设计不应该只做一次，甚至希望永不更改。**XP**其实认为设计非常重要，也是一种例行工作。我们总是随时做能够达到目的而且最简单的设计，然后依据现实情况做适当修改。
- **YANGI**



简单设计 (Simple Design)

- Kent Beck认为对于XP来说，简单设计应该满足以下几个原则：

1. 成功执行所有的测试；
2. 不包含重复的代码；
3. 向所有的开发人员清晰地描述编码以及其内在关系；
4. 尽可能包含最少的类与方法。

对于国内大部分的软件开发组织来说，应该首先确定一个灵活的系统架构，而后在每个迭代周期的设计阶段可以采用XP的简单设计原则，将设计进行到底。



集体代码所有权(Collective ownership)

- 小组中的任何人都应该有权对代码进行更改来改进它。每个人都拥有全部代码，这意味着每个人都对它负责。
- But, Xpers:自己的玩具自己收！



持续集成 (Continuous Integration)

- 提倡在一天中集成系统多次，而且随着需求的改变，要不断的进行回归测试。因为，这样可以使得团队保持一个较高的开发速度，同时避免了一次系统集成的恶梦。
- 持续集成也不是XP专有的最佳实践，著名的微软公司就有每日集成 (Daily Build) 的成功实践。但是，要注意的是，持续集成也需要良好的软件配置变更管理系统的有效支持。



现场客户 (On-site Customer)

- 要使功能最理想，XP 小组需要在现场有一位客户来明确素材，并做出重要的企业决策。开发人员是不允许单独做这些事情的。让客户随时在场可以消除开发人员等待决策时出现的瓶颈。



代码规范 (Code Standards)

- 如果没有编码标准，重新划分代码会更加困难，按应当的频度交换对更困难，快速前进也更困难。目标应该是团队中没有人辨认得出是谁写的哪一部分代码。以团队为单位对某一标准达成协议，然后遵守这一标准。目标不是创建一个事无巨细的规则列表，而是提供将确保您的代码可以清晰交流的指导方针。编码标准开始时应该很简单，然后根据团队经验逐步进化。不要预先花费太多时间。创建能够工作的最简单标准，然后逐步发展。
- 集体代码拥有的基础



每周40小时工作制 (40-hour Week)

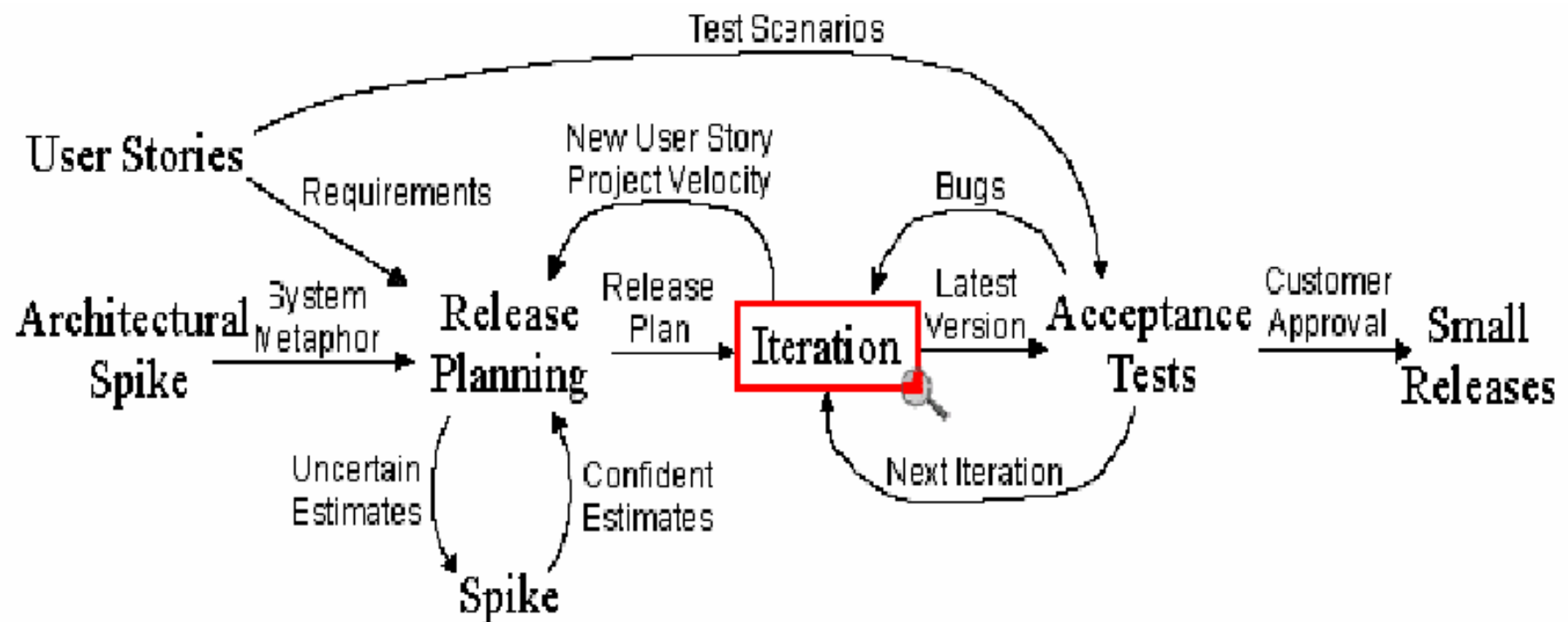
- 项目团队人员每周工作时间不能超过40小时，加班不得连续超过两周，否则反而会影响生产率
- “以人为本”



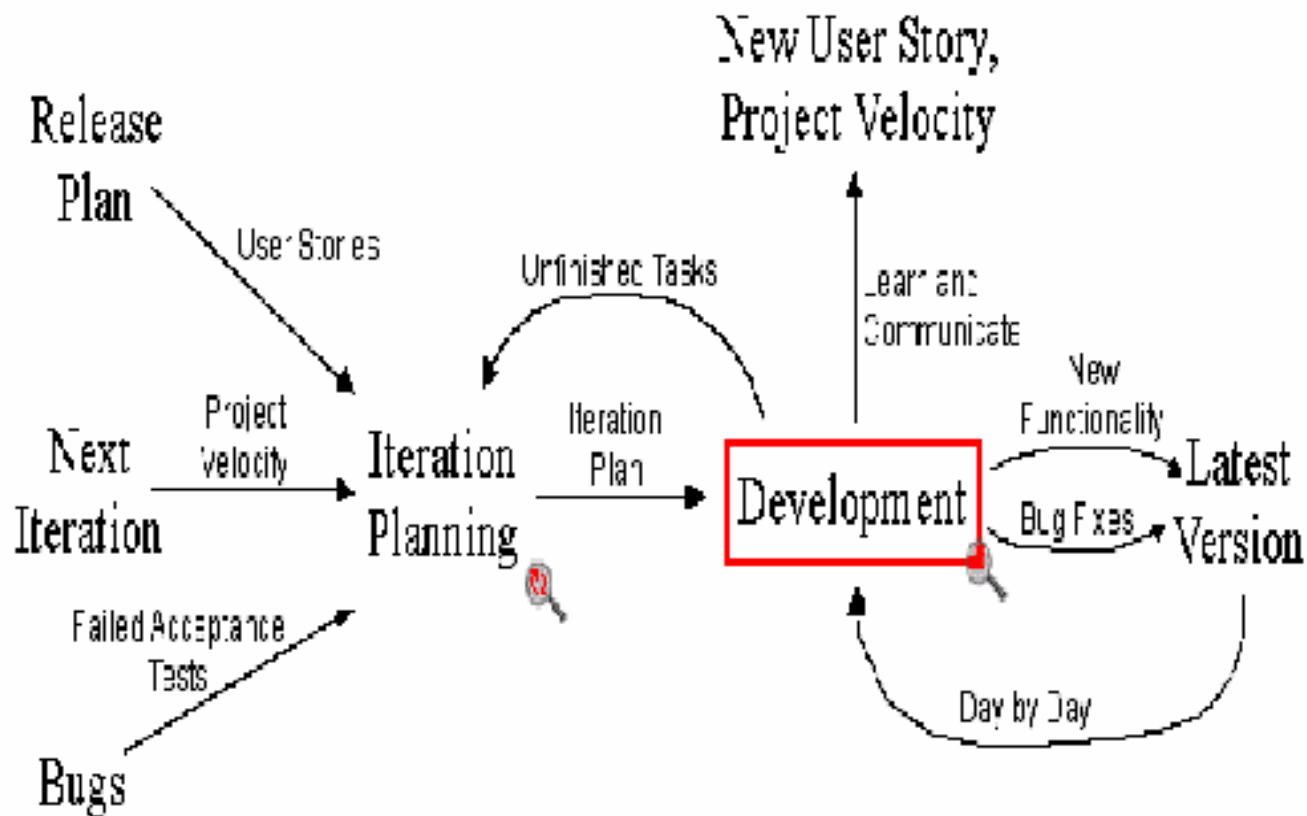
XP体现四个价值目标

- 沟通（communication）
- 简化（simplicity）
- 反馈（feedback）
- 勇气（courage）

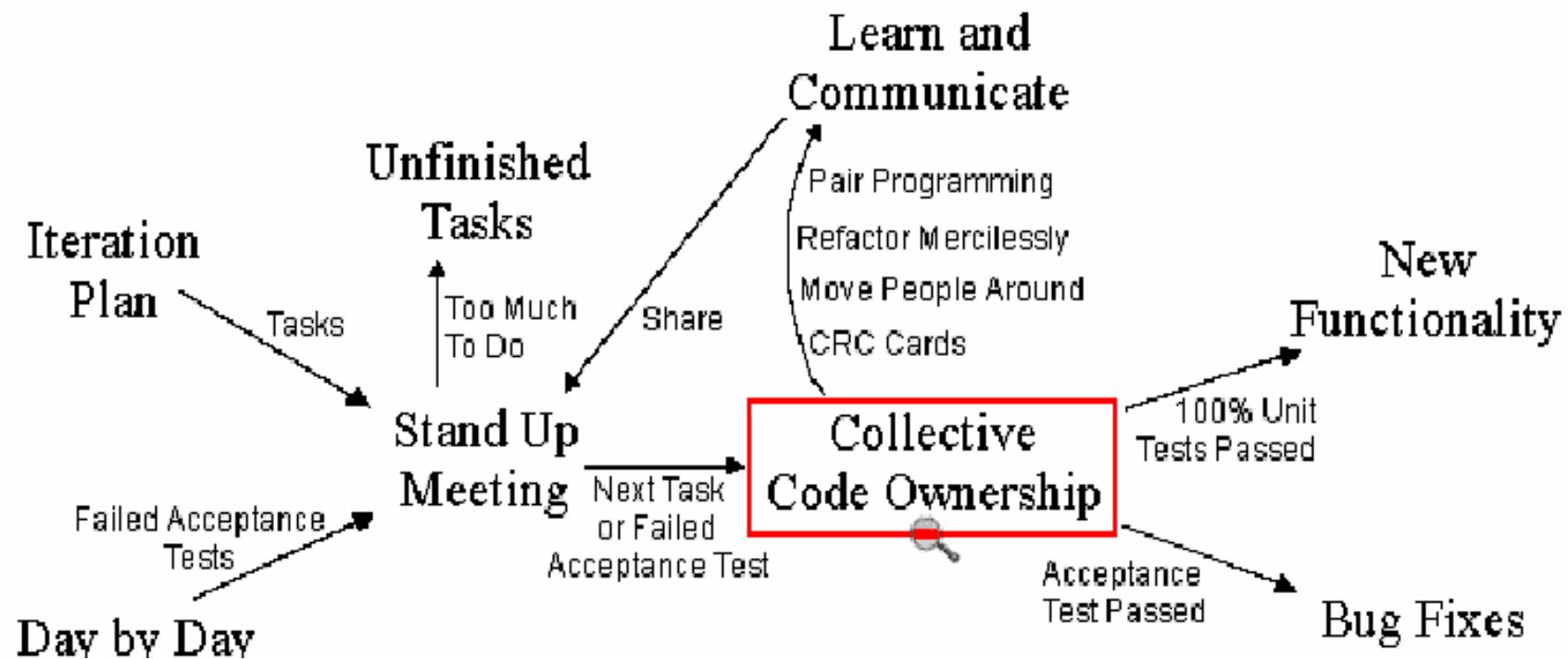
XP的流程1



XP的流程2



XP的流程3

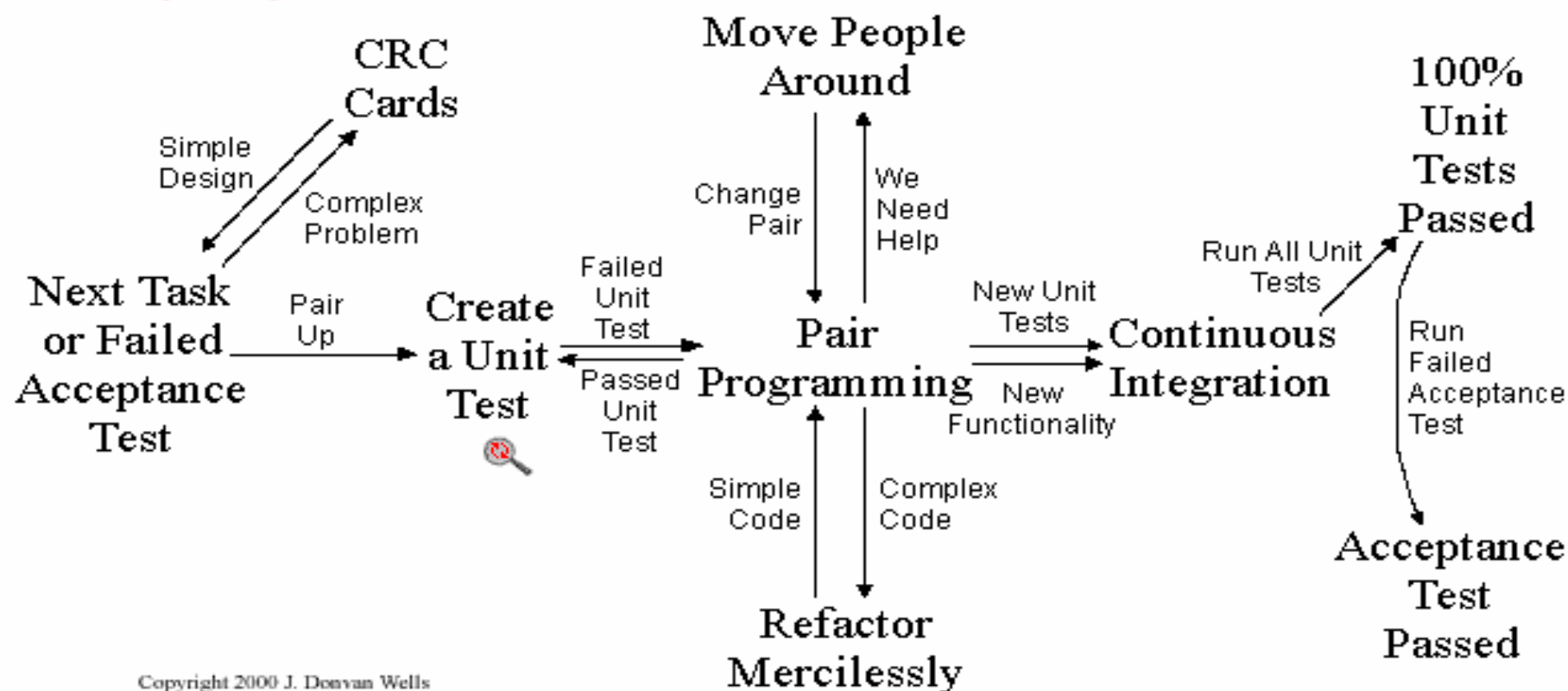


XP的流程4



Collective Code Ownership

Zoom Out





XP的文档

- **USER STORY**

记录了用户的需求，简单清晰是它的特征。

- **CRC卡**

设计阶段产生的东西。这个东西也不一定能维护得很好，比如改动了一个类，程序员可能不会重新画一个**CRC卡**。



XP的文档

- 单元测试

这些是测试程序，而不是打印在纸上的东西。到了项目后期和维护阶段，花在这个文档上面的时间可能比花在其他文档上的时间多。因为，每发现一个**BUG**，就应该补充一个相应的单元测试。这个是XP中最强调的文档，对于程序员来说，这个也是程序，不象现在很多项目中要求那些文档那么枯燥。



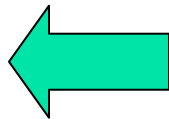
XP的文档

- 其他根据实际需要而灵活规定的一些文档，比如反映项目进度和开发速度的文档、资源调配的汇报等等。关于用户手册等，或者如果是用户指定要求的某些文档，可以把当作一个 **USER STORY**写下来，评估并安排在 **ITERATION**中。



OUTLINE

- 历史
- 定义
- 内容
- 应用
- 讨论
- XP的未来



环境

- 既有无隔墙隔板的工作场地，也又单独的工作间；
- 一个足够宽敞的地方供大家开会；
- 足够大的白板；
- 足够长的电脑桌，可以让两个人并排坐在同一台电脑前面；
- 每一个人都能很容易地看到其他人并和他们交流。
- 一些白纸或者卡片。

更理想的条件：

- POP
- 电视机
- Video Game
- 落地玻璃窗



开发

产生Story

评估Story

开始提炼
Metaphor

发布
计划

迭代计划

CRC卡设计

分解Task

承担Task

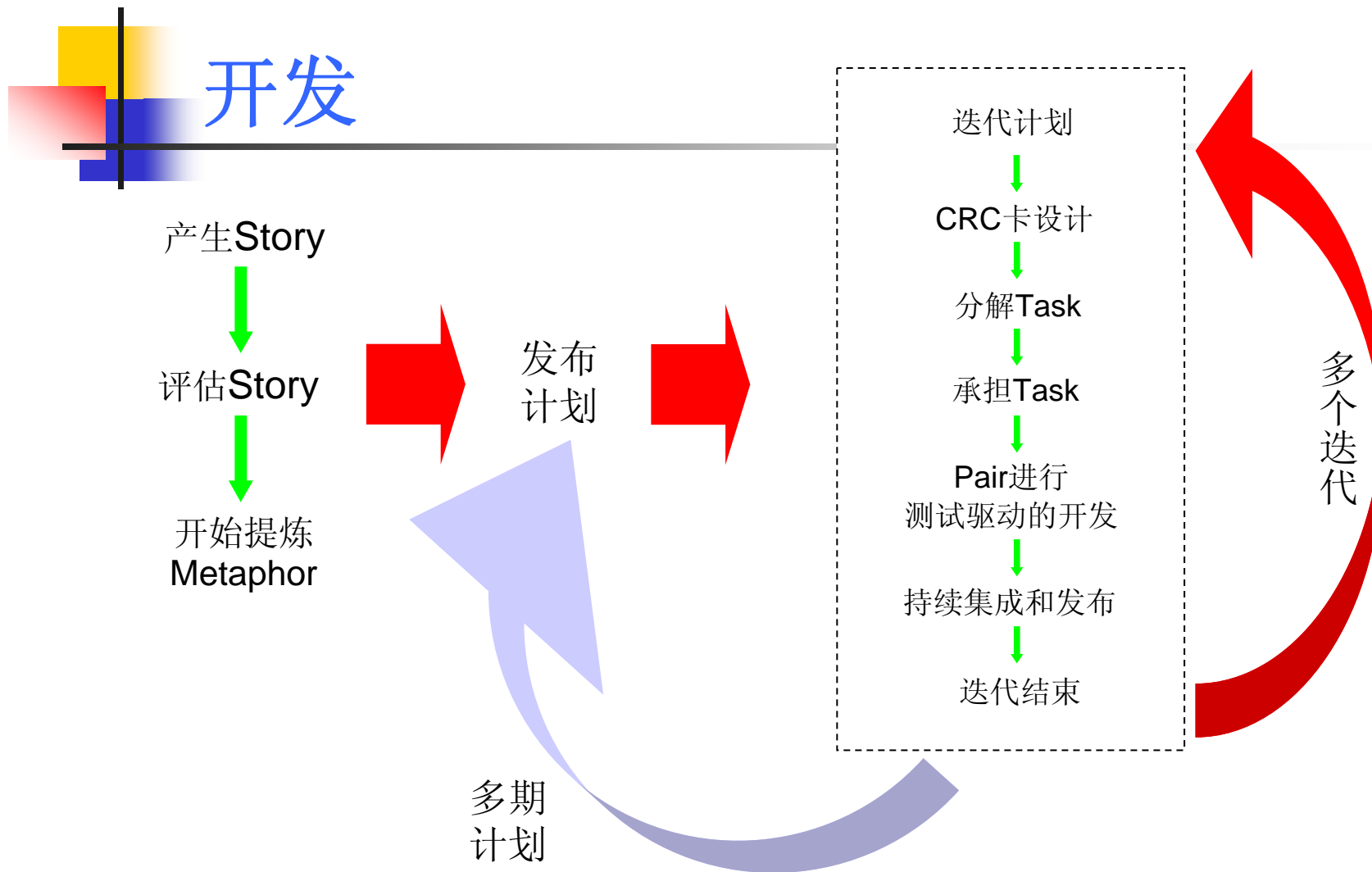
Pair进行
测试驱动的开发

持续集成和发布

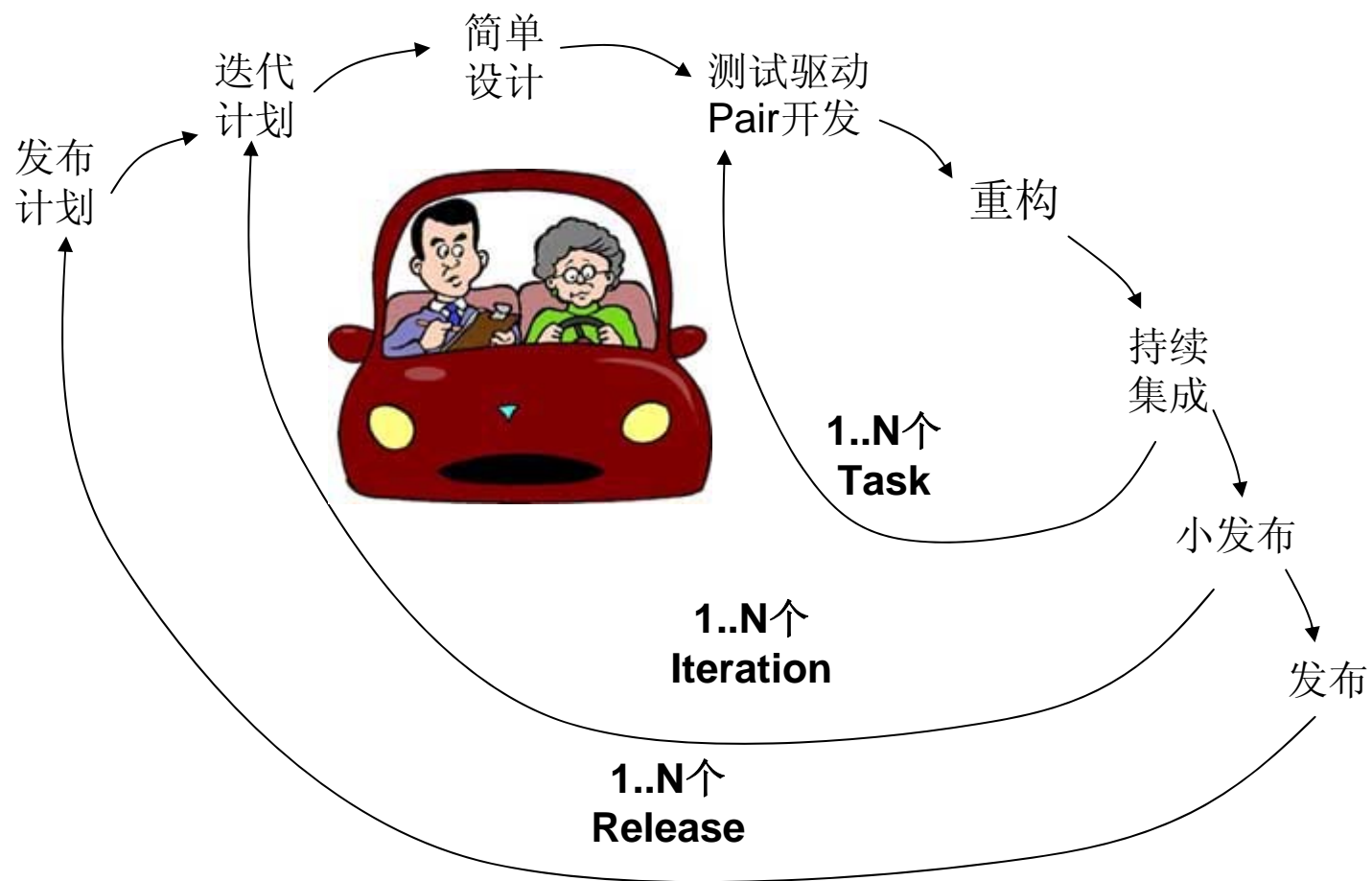
迭代结束

多个
迭代

多期
计划



XP的增量过程



XP过程



1、准备开发环境。

2、Exploration Phase

(1) 程序员和用户一起把需求分解为User Story，用户写Story卡。

(2) 用户把在Story卡上标上商业优先级(1, 2, 3)。

(3) 程序员看卡片，估计所需要的理想开发周，把估计的时间写在Story卡上(1, 2, 3)。

(3.1) 由一个Senior估计时间。

(3.2) 如果估计不出来，做Spike。

(3.2) 如果时间大于三周，分解Story，撕掉旧Story卡，写下新的Story卡。回到(3)，重新估计。

(3.3) 由一个Junior估计时间。衡量二者的差异，取平均值。

(3.3) 如果时间大于三周，回到(3.2)。

(3.4) 如果时间小于一周，尝试和其它Story合并。合并后，撕掉原来的Story卡，写下新的Story卡。

(4) 程序员看卡片，估计开发风险。把风险级别写在Story卡上(1, 2, 3)。

(4.1) 程序员将所有卡片放在一起，然后阅读。如果认为风险低，就放在第二堆，否则不移动。

(4.2) 程序员阅读第二堆卡片。如果认为风险低，就放在第三堆，否则不移动。

(4.2) 对三堆卡片分别标记。



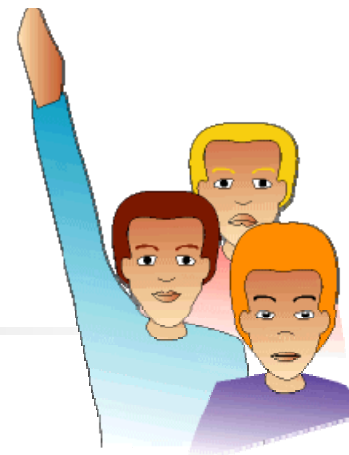
XP过程

3、Release Plan

- (1) 确定该Release中Iteration的长度(例如3周)。计算，开发总时间/迭代长度 = 迭代数。
- (2) 计算可完成的开发量，告诉客户。
 - (2.1) 根据经验，估计程序员的平均开发速度(例每个Iteration完成1个理想周)。
 - (2.2) 计算，平均开发速度 x 程序员数 x 迭代数 = 可完成的工作量。
- (3) 客户挑选Story卡，累计这些卡片上的估计开发时间，不要超过宣布的可完成工作量。挑选的原则为商业价值优先。
- (4) 把要完成的Story卡贴在墙上或白板上，要让所有的人都能容易地，清楚地看到。
- (5) 整个Team，包括客户，在一起做快速设计，并想出Metaphor。



XP过程



4、Iteration Plan

- (1) Tracker宣布该Iteration的计划速度。
 - (1.1) 如果是第一个Iteration, 则计划速度 = 程序员数 x 1。
 - (1.2) 否则, 计划速度 = 上一个Iteration实际完成的理想开发周数目。
- (2) Team挑选要实现的Story, 累计这些Story卡上的估计开发时间, 不要超过计划速度。客户挑选商业价值最高的User Story, 商业价值相同时挑选开发风险最高的。
- (3) 客户宣读并解释User Story卡。
- (4) Team用CRC卡或者其它方式设计该Story, 并将其分解成Engineering Task, 记录在Task卡上。
- (5) Tracker宣布各程序员在该Iteration的计划速度。
 - (5.1) 如果是第一个Iteration, 则计划速度 = 5 理想开发天。
 - (5.2) 否则, 计划速度 = 上一个Iteration实际完成的理想开发天数目。
- (6) 每个程序员挑选一个Task, 并估计所需要的理想开发天。如果大于三天就分解Task, 小于1天就和其它的Task合并。
- (7) 把天数和程序员的名字写在Task卡上。
- (8) 每个程序员重复(6)的过程, 直至所有Task的理想开发天的总和等于自己的计划速度。
- (9) 留下的Task卡可以放在一个Iteration, 或者程序员提前开发完后承担。

XP过程

4、Iteration Development

- (1) 客户为每个User Story定义功能测试。
- (2) 每天早晨开一个简短的Standup Meeting。每个人简要讲述昨天做的工作，今天准备做的工作，和碰到哪些困难。
- (3) 程序员进行Pair Programming
 - (3.1) 根据难度和重要性决定Senior，Intermediate和Junior
 - (3.2) Pair自由决定先开发谁的Task。
 - (3.3) Pair开发时，自由决定谁是Driver，谁是Navigator
- (4) 设计该项Task。
- (5) 为该设计的所有类和方法写Unit Test。
- (6) 写程序实现该设计的所有类和方法。
- (7) 运行所有的Unit Test，在100%通过后Check In。
- (8) 重构。完成后执行(7)。



XP过程

4、Iteration Development

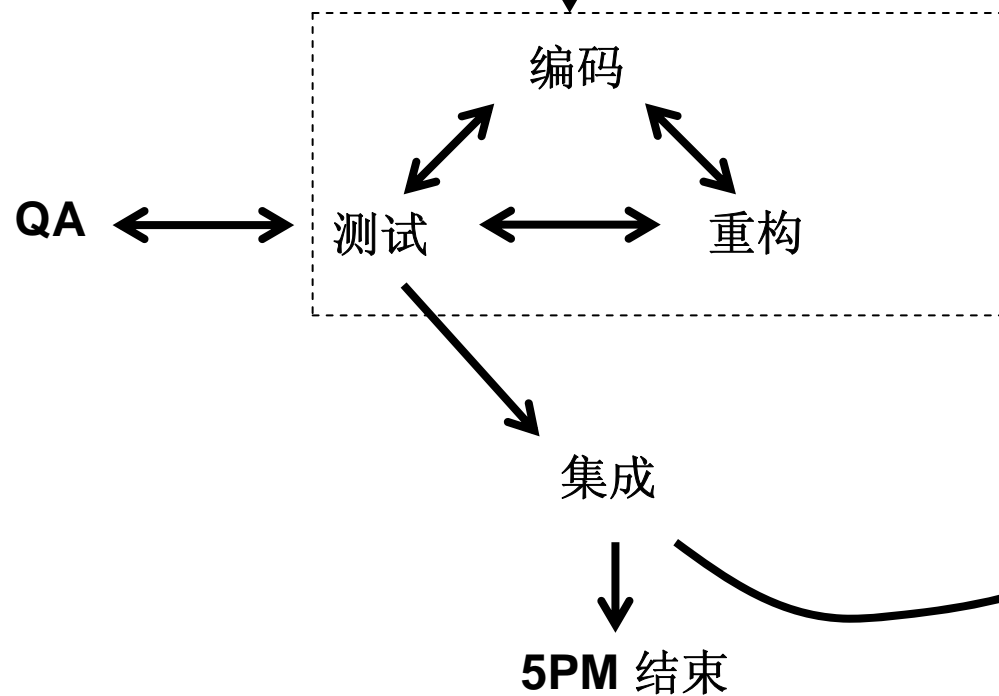
- (9) Build系统，只放入一个Pair完成的功能。
 - (10) 运行功能测试，通过则进入(11)，否则如下：
 - (10.1) 如果失败或者发现BUG，找到原因后要先写Unit Test。
 - (10.2) 写程序。通过所有Unit Test后Check In。
 - (10.3) 重新执行(9)。
 - (11) 进行小发布。
 - (12) 客户使用，给予反馈。
 - (12.1) 如果发现BUG，进入步骤4-10.1。
 - (12.2) 在此基础上产生的新需求或变化，做为一个Story或Task。
 - (12.2.1) 客户和Team协商何时评估该Story/Task，以及放在那个Iteration中。
 - (12.2.1) 评估过程参考2。
 - (13) Tracker跟踪统计Story和Task的完成情况。
- 5、开始下一个Iteration，重复4。



程序员的一天

9AM Standup Meeting

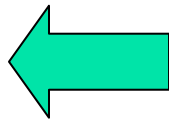
Pair Up





OUTLINE

- 历史
- 定义
- 内容
- 讨论
 - XP中的pattern
 - XP中的设计
- XP的未来





Pattern和XP并不冲突

- **patterns** 被滥用了。但并不表示 **patterns** 是不足取的，问题在于你要怎么运用它。
- **XP** 只是与一般使用 **patterns** 的方式不同而已，并没有抹煞它的价值
- **XP** 也许是开发的一种流程，但 **patterns** 可是设计知识的骨干，不管是哪种流程这些知识都是很有用的。不同的流程使用 **patterns** 的方式也就不同，**XP** 强调等到需要时才使用 **patterns** 以及透过简单的实作逐步导入 **patterns**。所以 **patterns** 仍然是一种必须获得的关键知识。



XP中使用 patterns 的建议— martin Fowler

- 花点时间学习 patterns。
- 留意使用 patterns 的时机 (但是别太早)。
- 留意如何先以最简单的方式使用 patterns，然后再慢慢增加复杂度。
- 如果用了一种 pattern 却觉得没有多大帮助—不用怕，再次把它去掉。



XP中的设计

- Jim Highsmith 写了一篇很棒的文章 "summary of XP", 他把 **planned design** 和 **refactoring** 放在天秤的两端。大部份传统的做法假设构想不变, 所以 **planned design** 占优势。而当你的成本越来越不允许变更, 你就越倾向于采用 **refactoring**。Planned design 并不是完全消失, 只是将这两种做法互相搭配运用取得平衡。对我来说, 在设计进行 **refactoring** 之前, 总觉得这个设计不够健全



XP中的设计

- Continuous integration、testing 和 refactoring 这些有效的实作方法让 evolutionary design 看似很有道理。但是我们尚未找出其间的平衡点。我相信，不论外界对 XP 存有什么印象，XP 不仅仅是 testing、coding 和 refactoring。在 coding 之前还有 design 的必要。部份的 design 在 coding 之前准备，大部份的 design 则发生在实作每一项详列的功能之前。总之，在 up-front design 和 refactoring 之间可以找到新的平衡。

——— 《设计已死》



So is Design Dead?- martin的 答案

- 没什么原因，只是设计的本质已经改变。XP的设计追求以下的技巧：
- 持续保持清爽的程序代码，越简单越好。
- 重构的技巧，所以当你觉得必要的时候都可以有信心的动手。
- 具有 **patterns** 的知识：不只是照它的解法，更要感觉何时可以应用，或是如何导入 **patterns**。
- 知道如何将设计说给必要的人了解[译注8]，用程序代码、或是图形、或上述所有的工具：交谈。



OUTLINE

- 历史
- 定义
- 内容
- 讨论
 - XP中的pattern
 - XP中的设计
- XP的未来 



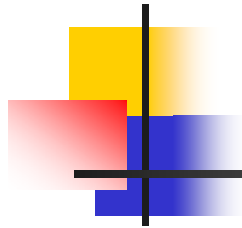
C3项目的失败

- 客户报告对**C3**项目的失败要承担主要的责任之一，该项目是采用**XP**的标志之一。当其关键客户代表离开项目时，她也同时带走了大量的领域内知识。



XP的未来

- XP meet CMM
- XP meet UML
- XP and RUP → XRUP, dX



有关网站

- <http://www.agilechina.org/>
- <http://www.softwarereality.com/>
- <http://www.extremeprogramming.org/>
- <http://pairprogramming.com/>
- <http://xprogramming.com/>
- <http://www.martinfowler.com/>



spike solutions

- 创建关键问题解决方案，解决关键的技术和设计问题
- 大多数 spikes可能都不足以得到保持，有可能被丢弃. 但制作SPIKE的目标降低技术风险；
- 一旦技术困难对系统的开发造成了威胁，立即派一对开发人员关注于该问题一至两周，以降低潜在的风险