如何通过技术手法避免进度风险

前言

上一场Chat，我们聊过了产品前期的需求管理过程（收集、分析、估算、计划），然而并不是做了计划就完事大吉。本场Chat和大家讲一下Audio UI团队的以下三个改进内容：

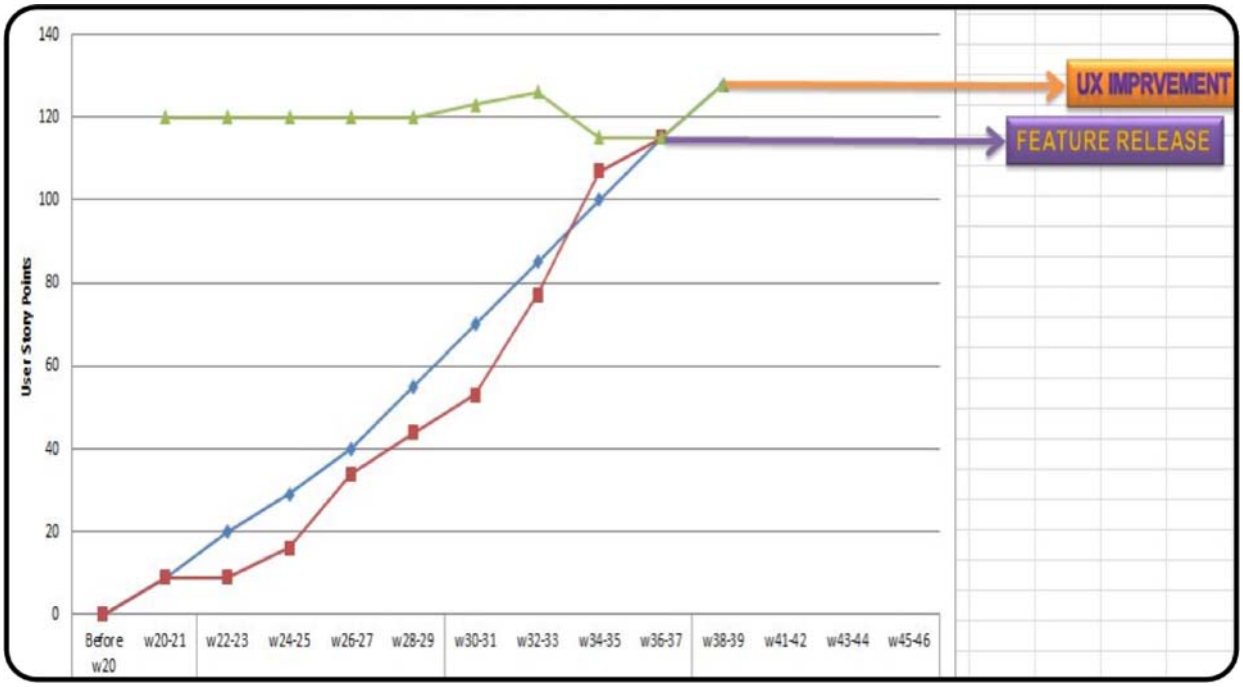
- **故事墙的变迁**：如何利用“故事墙（看板）”发现常见却不易察觉的“坏味道”。
- **单元测试与测试前置**：虽然有生产成本，但收益更大。
- **主干开发的动机**：放弃分支开发的恶习，消除不必要的浪费。

当我们做完工作量估算，并分析完依赖后，做了一份项目计划。这份计划的决策点有以下几个：

- 团队对这种新的迭代工作方式需要一个熟悉的过程。
- 项目过程中会从Synergy切换成Git（除我以外，团队中没人用过Git）。
- 在这个项目中，我们要写单元测试（除我以外，团队中没人写过单元测试）。
- 我们团队要在自己的开发分支上，使用持续集成实践，需要在项目开始时准备环境。
- 虽然对其它团队有依赖，但是已与该团队沟通，应该不会影响我们之间的集成进度。

Tech Lead评估后认为，没有改变开发模式时，正常开发速度为每个迭代完成13个故事点。但前两个迭代需要学习新工作模式，因此工作产出应该少于13个故事点，约为9个故事点，相当于原来产出的70%。但后期集成测试阶段的时间会缩短，可以弥补这个产出损失。

所以整个项目计划如下图中的蓝色曲线（本图是整个项目整个执行过程的完整Burnup曲线）。



- 绿色曲线：需求范围
- 蓝色曲线：项目计划
- 红色曲线：实际执行

图有你会发现：

1. 没有最后的团队集成测试迭代。
2. 产品经理在过程中修改过需求范围。
3. 产品经理在最后增加了需求。

在这个项目之前，Nokia功能手机部门所有团队的故事墙（白板）都是类似下面的样子。只有三列，**ToDo-Doing-Done**。每个人员占用一行，这一行中是他在本次迭代中的工作任务。

从上面这个两周迭代的白板中，我们能发现下面两个事实：

- 为期两周的迭代，只有两个工作量很大的用户故事Story1和Story2。
- 每个Story的任务分解非常模板化：
  - 设计模块。
  - 修改模块。
  - 对模块修改进行code review。
  - 写Story的测试用例。
  - 对Story进行测试。

这就能闻出“坏味道”啦：

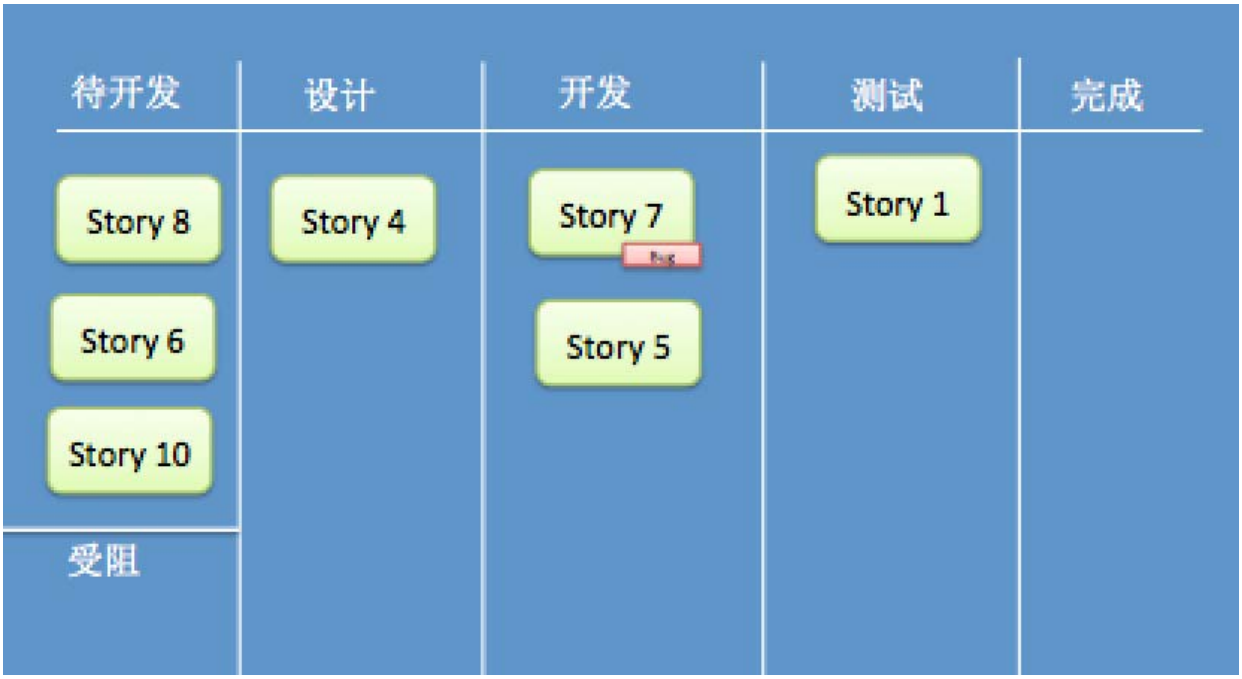
1. **Story太大。**因为只有到了本次迭代的末期，才能将开发人员的代码集成在一起，完成对Story的测试，集成时间太靠后，质量在过程中不可控。
2. **仅仅按工作活动拆解。**

一、故事墙的变迁

变化一：以价值流动为核心的故事墙

在这个项目中，我们将开发者的工作活动做为每一列的列头，这就是精益管理中价值流分析的雏形。而每个小粒度的用户故事都是一个价值项，它们从白板的左侧向右侧流

动。每当一个故事走到最后一列“Done”，就代表着价值的交付，因为这意味着我们生产的代码写完且达到质量，验证通过了。而团队在每时每刻都可以看到真实的项目（需求）进展。如下图所示：



细心的同学可能会注意到在“开发”一栏中，Story7卡片上有一个红色小□纸条，写着“Bug”。这表明，Story7已经开发完成了，但在测试时发现了Bug。测试人员会在Story7贴上一个小纸条，写上Bug内容，并把它放回“开发”一栏，由开发人员修复。

测试同学的疑问：

写测试用例这个活动不用体现在这个白板上吗？”

我的回答：

并不是说写测试用例这个活动没有价值，而是因为这个活动并不在这块白板上体现。因为当Story被放入到Backlog时，就已经应该写好验收条件或测试用例了。我们应该在每个迭代开始前，都尽可能为这个迭代中的每个需求准备好测试用例。

变化二：明示“完成”的定义（每个环节的质量标准）

在迭代执行过程中，我们发现，开发人员并按我们设想的方式工作，而是在开发完成之后，补写设计文档，所以基本动作都是直接将开发完成的需求移动到了“测试”一栏。在迭代回顾会议上，我们重新强调了这个要求，而且明确了每次挪动Story时必须完成的动作。很多开发同学是新手，开发之前要做模块设计review，再开始写代码，写完代码之后要做Code Review。为了时刻提醒每个人，我们还把它打印出来，贴在了每个栏目之间，这就是Defination of Done，简称DoD。

- **待开发—>设计**：验收条件（或测试用例）必须写好，并经过产品人员、开发人员和测试人员的review，没有异议。
- **设计—>开发**：完成设计文档的更新；设计review完成。

- **开发—>测试**：编写对应用的单元测试，确保所有单元测试用例成功通过；做完codereview.
- **测试—>完成**：全部测试用例都能通过，所有已发现的Bug被修复。

变化三：On-board Design

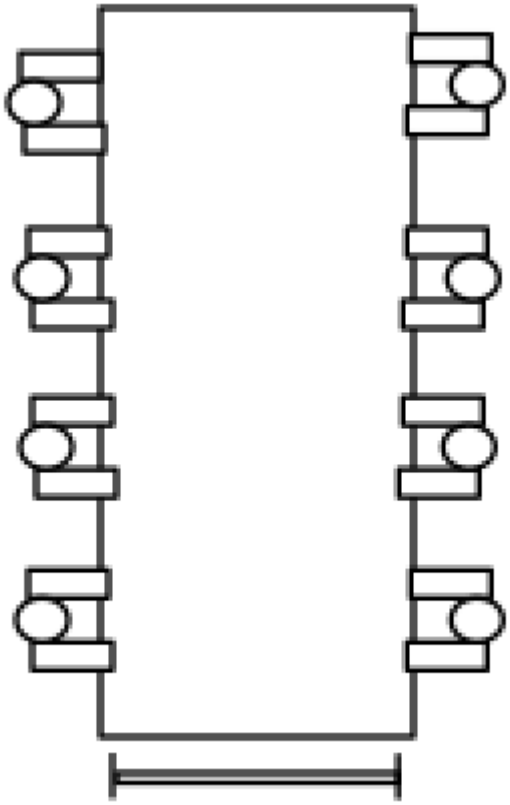
我们在每个需求卡片上记录了它在每个阶段停留的时间，这个时间以天为单位，D代表在“设计”一天，C代表在“开发“一天，T代表在“测试”一天，如下图所示。

两周后，发现了一个有趣的现象。那些实现上相对简单的需求与相对复杂的需求相比，在设计时间上并没有太大的差异。这是为什么呢？原来，每个开发人员在拿到一个新Story时，习惯性的行为是：

1. 自己先在Word文档中编写相关模块设计。
2. 再发邮件给其他开发人员Review（大部分是新兵，经验少，而且Nokia有一个自己开发且有点复杂的代码框架）。
3. 等对方给他回复邮件后，再根据邮件中的内容进行修改Word文档，修改完成后才算设计完成。

整个流程并没有什么太大的问题，问题在于工作的方式。

当一开始就使用Word文档进行编写时，开发人员会花大量的时间在文档的美化上了（这是一个自然而然的事情，谁都不想自己画出来的设计图不好看）。然而，在没有确定这个设计有效前，这种让文档整洁的美化工作就是一种浪费。因此，我们改变了一下要求，让大家做“On-board Design”。



我们的工位排列如下图所示，在桌子的一头有个大白板。正面用于展示我们的“故事墙”，反转后，在背面就可以写写画画，用于讨论问题。

所谓“On-board Design”，分下面几个步骤。

1. Jojo拿到一个Story后，自己做初步设计（不限于用Word文档，白纸上也行）。
2. Jojo约几个同事到一起到白板前，一边在白板上画，一边给大家讲（最多5分钟）。
3. 大家给出反馈，和改进建议。
4. Jojo根据讨论结果，编写设计文档，并提交，设计结束。

这样，一个需求从开始设计到更新完文档的过程很快就会完成了。

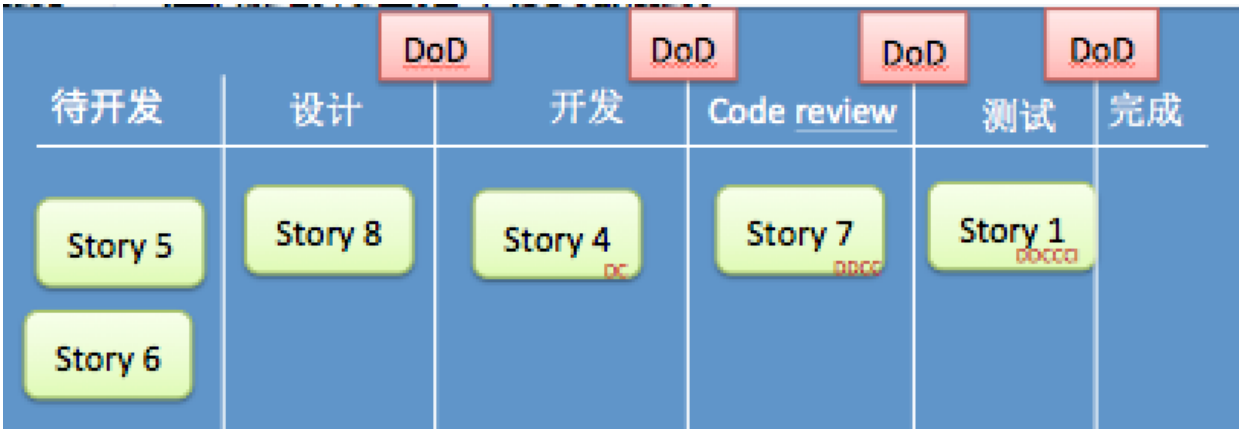
变化四：先code review，后开发

虽然团队约定，当一个需求开发完成以后，需要做Code Review。然而，在故事墙的“开发”一栏中，经常有需求卡片堆积现象。

原来大家把手上的Story开发完成以后，都会发邮件给reviewer，并当面提示一下对方。然后就再拿一个新的需求开发。而reviewer自己也在开发需求，可能还很棘手。这时，Reviewer通常会想尽快完成自己手中的工作，然后再Review别人的代码。每个人都有这种倾向，所以会产生卡片在“开发”一栏上堆积。

于是，我们将故事墙增加了一栏，名字“code review”。当开发人员完成手中需求的开发，并发出review邮件后，就可以将这个需求移动到“code review”一栏中。这样，我们就可以清楚地知道，哪些需求是在开发中的，哪些需求是在code review中。

与此同时，我们还约定，当发出review 申请时，需求通知reviewer。如果Reviewer在两个小时内还没有回复的话，开发者需要再次提醒reviewer。**即在整个流程中，越是接近于最后交付的环节，Story越快收获价值，要让所有需求快速流动起来。**



变化五：保障交付质量，避免不必要的任务切换

经过两个迭代后，我们又发现了一个现象。开发人员开发人员提交测试后，测试人员经常会发现一些很明显的问题，需要返工，由开发人员修复这些明显Bug。而且，开发人员习惯于先忙完手中的新工作，再来修复这些Bug。这是一直以来开发人员养成的坏习惯。

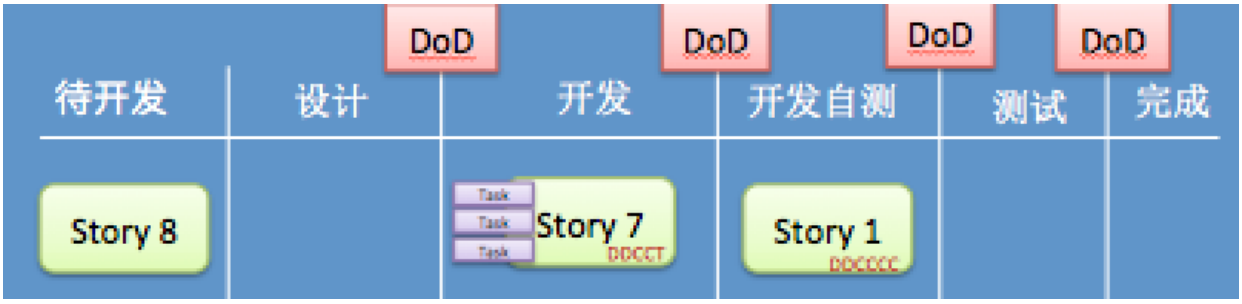
根据项目前的团队约定，开发人员要根据每个需求的验收条件，进行开发自测，确保没有明显问题。所以，我们的白板增加了“开发自测”一栏，如下图所示（*由于上次的改进，对需求进行Code Review的时间大幅度缩短，我们将其从故事墙上删除*）。同时，也定义了从“开发”到“开发自测”的DoD，即：根据验改条件或测试用例，完成自测，确保没有低级Bug。

变化六：没有反馈，就是风险

尽管在前期需求拆分过程中，我们会尽可能让每个需求都是小粒度的。但总是有个别需求无法拆成小（强行拆分可能也无法验收，或者验收成本过高）。

此时，我们使用弱反馈方式，即：每个需求还是可以拆分成多个开发任务，虽然每个开发任务无法由测试人员进行验收，但是可以通过code review来确定其完成。所以，我们要求将这类需求分解成多个开发任务，每个任务由code reviewer来确认其完成。

在故事墙上，将拆分出来的任务写在小卡片上，与该需求放在一起。每确认完成一个任务，就将该任务移除。故事墙如下图所示。



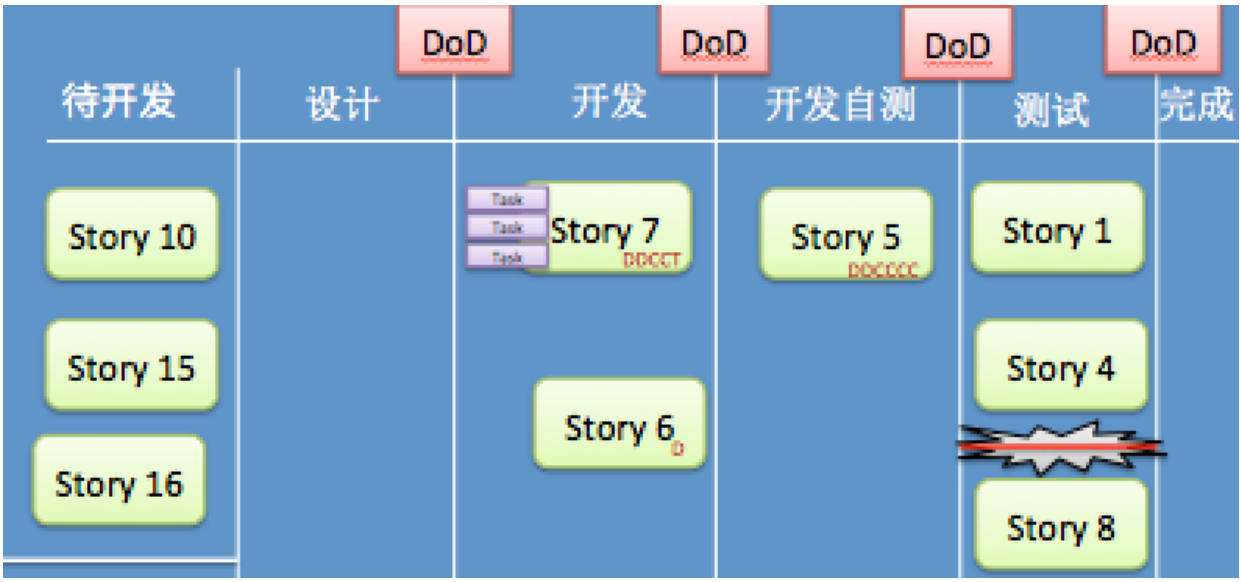
变化七：限制在制品数量

> “在制品（Work InProgress，简称WIP）”这个概念来自于精益生产管理理论，它是指尚在生产流程中未交付的半成品。精益生产管理理论认为，库存不增加价值，而只增加成本。在精益生产体系中，库存分为原材料库存、在制品库存（WIP）、成品库存三种类型。

随着项目的进展，故事墙上又出现了“堆积”症状。在“测试”一栏中的需求会越来越多。原因其实很简单：系统增加的功能越来越多，需要测试的内容比项目前期要多。另外，开发完成后，验收出来的缺陷数量也有所增加。虽然能够可以比较快速地修复这些缺陷，但是我们只有一名测试人员，来不及验证这些被修复的缺陷。

怎么办呢？我们采用了精益管理理论中的“**限制在制品数量**”的策略，但并没有严格指定在“测试”环节可以有多少个需求。我们在每日早上的站会时，由测试人员评估是否当天能够完成测试环节中所有需求的验收。如果她能够完成这些验收工作，那么一切工作正常进行；假如她认为无法当日完成这些验收工作，Tech Lead会指定当天需求开发的人员不再领取新的开发任务，而是测试人员的指导下，帮助她进行需求验收工作，直致她评估可以完成当日验收工作为止。

这就相当于在“测试”环节，我们限定了最高带宽，一旦超过了测试人员的生产力，那么就停止前面开发环节的生产，扩大“测试”环节的产能，如下图所示。



故事墙改进小结：

至此，我们制定出来的团队约定，可以保证所有的用户需求在整个研发流程中平滑流动，在每个环节上都不会出现工作量大多，或者工作量太少的情况。总结一下到目前为止，我们用到的一些精益理论和质量管理小技法：

1. Value Stream Mapping（故事墙是“以价值为核心的工作流程”的体现）
2. 减少单位工作的批量大小（将大需求拆分成更小的需求）；
3. 持续改进：（不断观察工作流程，并发现问题，改进工作方式）；
4. 限制在制品的数量（测试人员每日评估产能）；
5. 质量内建（每个环节都定义了该环节的完成标准，避免不必要的任务切换）。

二、单元测试与测试前置动作

在这个项目中，我们引入了单元测试。引入的动机相对简单，公司鼓励大家写单元测试，但前提是不影响项目的交付。另外，团队成员希望学习尝试单元测试。团队中除了我之外，没有人接触过单元测试，更不用说单元测试框架的了解和熟悉了。所以，在第一个迭代期间内，我花了三个小时间，教大家使用CppUnit框架，并做了一个小练习，用的例子是火星探测车那道题目。

由于项目没有前期积累，要从头开始搭建一些辅助项目本身的基本测试类，所以测试代码量相对较大。项目结束后，做了代码量的统计。产品代码与测试代码的比例为1:1.2，即每100行产品代码平均对应120行单元测试代码。收到的益处有三个直接反映：

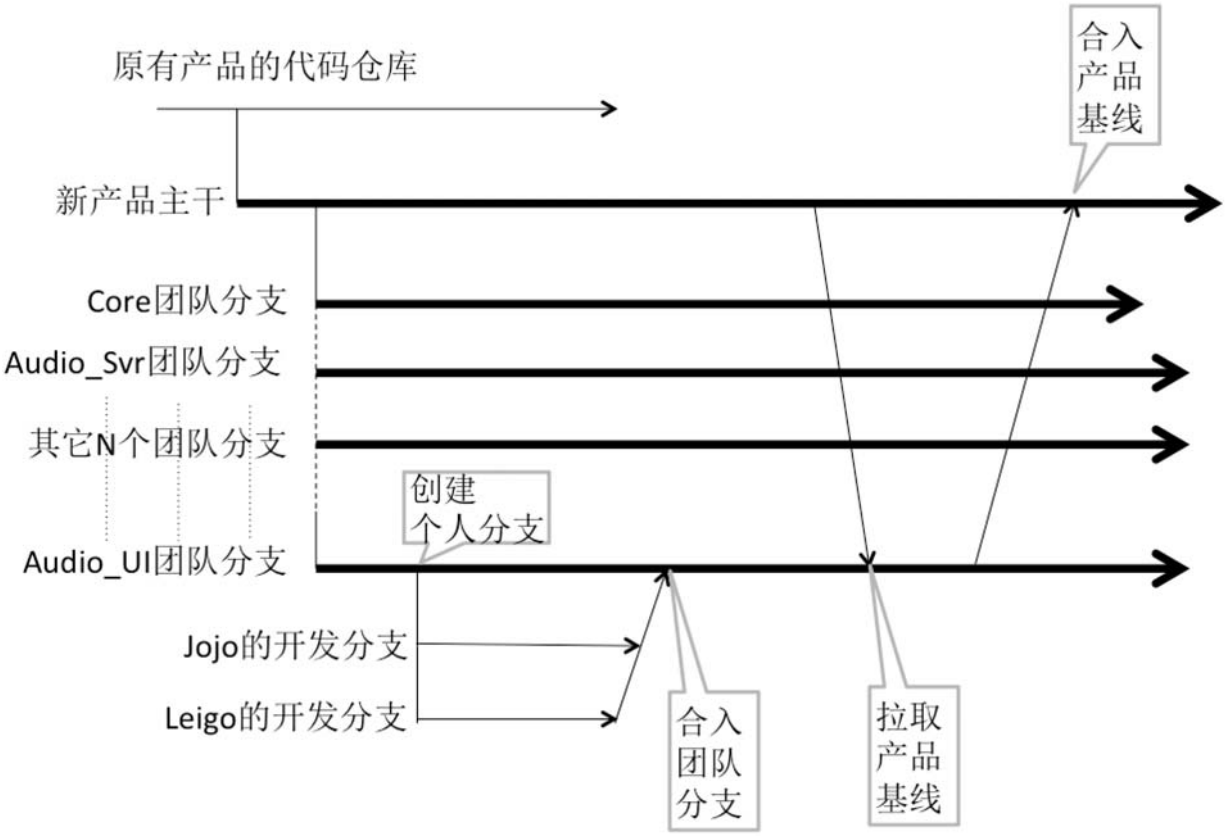
1. 团队提交代码比较有信心，参见下面一节。
2. Bug修复比较快。
3. 在这个项目之后的另一款产品开发中，团队成员能够很快增加并验证新功能。

测试前置（在腾讯被称为测试前移）是指测试人员在项目前期就深入参与相关活动，并提供相关信息输入，还有相关产出。有如下几个表现：

1. 参与拆分需求，并确认拆分后的需求可测试。
2. 每个需求在进入开发前，写好验收条件，甚至是测试用例。
3. 每个需求开发完成后，立刻进入测试，验证其开发质量，而不是在迭代后期一起测试。

三、主干开发的动机

在讨论切换主干开发的动机之前，我们要先说明一下Nokia功能手机部门的产品开发分支策略，方便大家理解。分支工作方式如下图所示：



分支策略解析：

- 新产品主干
  - 团队保证一个Feature已完成且达到要求的开发质量，才能合入该主干。
  - 该分支在每天晚上有一次自动化构建和测试，如果失败，团队必须优先修复。
- 团队分支
  - 各团队成员将代码提交到该分支，进行团队集成。
  - 某个功能开发完成后，与产品主干代码合并，自行验证通过后，再合入到产品主干。
- 个人分支
  - 每个人自己的开发分支。

原有提交习惯：每个团队都习惯于经过一段时间的开发后，再统一一次性提交代码到产品主干。因为这种习惯，所以每次从产品主干上拉取代码合并时，都会有比较大的合并及验证工作量，因此，每个团队都倾向于尽可能少地合入代码，即在最后时刻再合入代码。同样的道理，每个人为了减少代码合并的工作量，也都是尽可能少地将代码合入团队分支。

在项目初期（前两个迭代），Audio_UI团队也使用这样的策略。但是，在我的鼓励下，他们在第二个迭代时做了一次向产品主干的合并，负责合并的同学感到非常痛苦，要做

两次Merge操作（他要把他的代码合并到团队分支，然后再合并到产品主干）。在迭回顾会议上，我们讨论了这个问题。大家一致认为，我们应该删除团队分支。但还是有两个担心：

1. 我们自己代码的质量不好怎么办？可是看看我们自己的工作流程，马上就消除了这个担心。

2. 别人代码质量不好怎么办？结论是：我们的设计与其它团队只有一个接口，只要这个接口可以正常工作，那么对我们就不会有影响。而这个接口相对稳定，只有少量修改。

因此，从第三个迭代开始，我们就删除了团队分支，所有人直接向主干提交代码。

值得注意的是，公司刚好在此时将代码管理工具从IBM的Rational Synergy切换到了Git，并且提供了gitflow和code review的工具（谢天谢地，估计很多人都不知道Synergy这个工具，那他们真是太幸运了）。将代码提交到产品主干，实际上系统平台会自动创建一个分支，在这个分支上会有一系列检验，这些检验全部通过后，才会真正合入产品主干，这些检验如下：

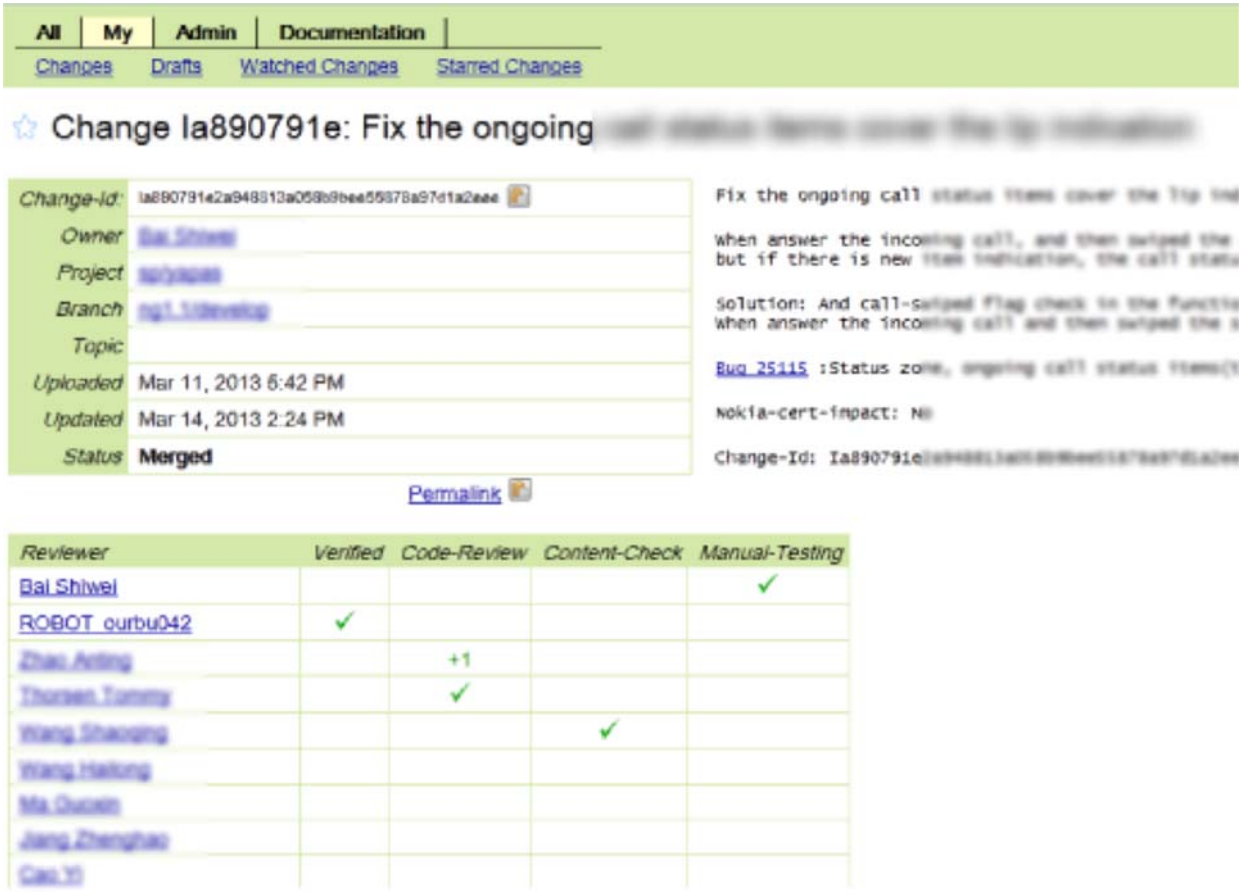
1. 会有一个自动创建的测试任务（ROBOT_task）必须执行成功（我们也要求负责配置这个自动化测试任务的团队将我们团队的单元测试用例也放入其中了）。

2. 每次提交的code review，必须得分在2分以上。

3. content Check必须通过。

4. 手工测试必须通过。

如下图所示：



到此，这个案例的主要改进就讲完了。整个项目的工作改变包括如下一些内容，有一些内容并没有在这两篇文章中讲到。

Task 拆分	→	Story 拆分
人天估算	→	排序法估算
拍脑袋的计划	→	基于显式PRIAD的发布计划
强调Sprint的成功	→	弱化Sprint成功的概念
分支开发	→	主干开发
基于Story的代码提交	→	基于开发任务, 每人每日提交
打补丁式的开发	→	通过设计，分离关注点
形同摆设的编码规范	→	团队认可并严格遵循的编码规范
最后集成测试	→	实时测试
Synergy	→	GIT
	→	单元层次的集成测试
	→	单元测试