第 50 期（2005 年 7 月）

募捐

X-Programmer 非程序员
软 件 以 用 为 本

联系：think@umlchina.com

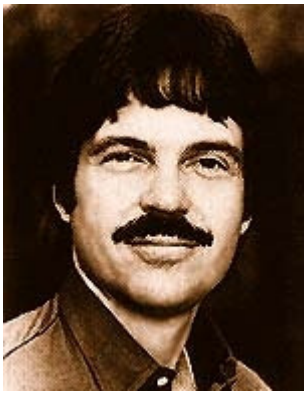
http://www.umlchina.com/

惠普裁员，Alan Kay 离开

[2005/7/22]

惠普试验室发言人 Dave Berman 昨天说“一些项目不再继续下去并不是因为他们没有价值，当公司对资源进行优先安排时，总希望将资源放到能对公司产生最大影响的方面”

作为惠普大重组，全球削减 14,500 个工作职位的一部分，惠普公司将停止惠普试验室的四个研发项目，其中包括科技先驱 Alan Kay 领导的项目。



全面重组

星期二，惠普总裁 Mark Hurd 表示在 18 个月内惠普将裁去相对于 151,000 名全球员工总数的一定比例的员工。

具体有关研发的裁员在昨天的 San Jose Mercury News 上首次被报道，其中有一封惠普试验室总监 Dick Lampman 给内部员工的电子邮件。

Herd 在周二时说过将尽量减少重组对研发的影响。被停掉的研发项目包括 Alan Kay 领导的高级软件研发团队。这个项目正致力开发一个新的互联网应用系统。Kay 在上个世纪 70 年代在施乐研究实验室工作期间曾以图形用户界面的研究而闻名。Kay 还是现代编程语言的先驱。

Kay 昨天并未立即回复电子邮件做出任何评论。他于 2002 年加入惠普任高级名士。

和 Kay 的高级软件研发团队同时被砍掉的其他研究项目是：位于马萨诸塞州剑桥的惠普剑桥实验室，这个实验室主要从事与健康医疗技术相关的研究；惠普消费者应用和系统实验室和新兴技术实验室。除了剑桥的实验室，有三个被砍的实验室位于加州的帕洛阿图市。

建立优先级

惠普试验室发言人 Dave Berman 称：公司正在调整研发目标，集中到最有可能对惠普发生长期影响力的方面。

Berman 昨天说到“一些项目不再继续下去并不是因为他们没有价值，当公司对资源进行优先安排时，总希望将资源放到能对公司产生最大影响的方面”

惠普的其他研发项目还将继续，包括它著名的纳米技术及量子计算技术研究。另外，惠普将继续其在企业计算技术、影像和打印技术、IT 服务等业务领域的研究。

Berman 说“所以这些都将继续进行下去，现在我们将能够更集中地使用资源。”

（自 technewsworld，Sicilia 摘译，不得转载用于商业用途）

关于Alan Kay>>

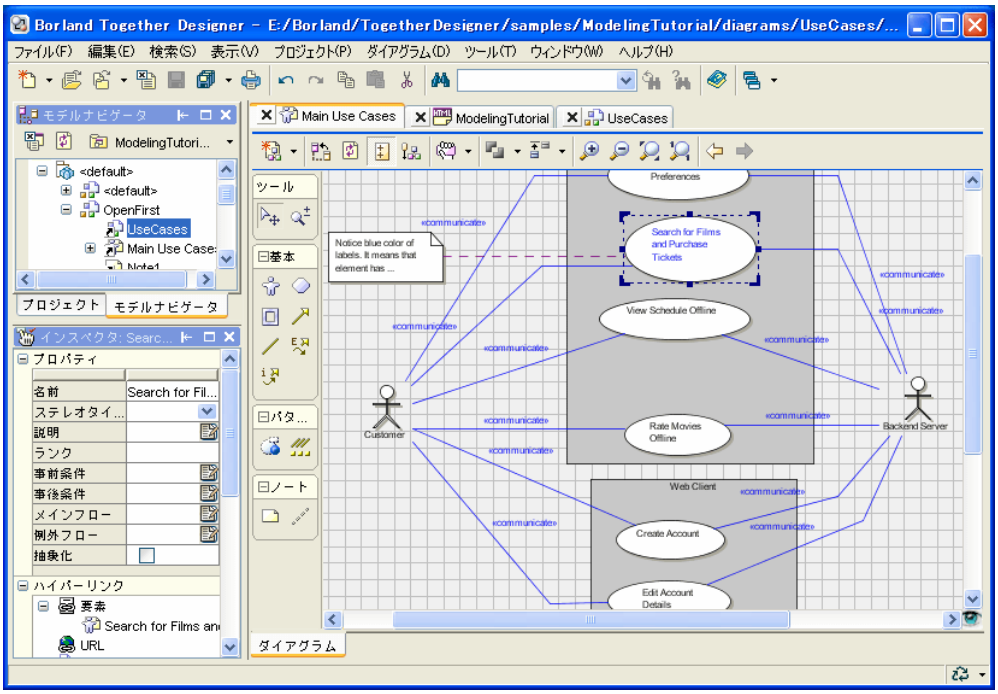
征稿

http://www.umlchina.com/xprogrammer/xprogrammer.htm

Borland 发布 Together2006

[2005/7/7]

Borland 公司发布了 Borland Together 2006,这是对其旗舰产品 Together 的一次重要升级。Together2006 致力于通过公共的可视化语言消除业务涉众（business stakeholders）、分析师和软件开发人员之间的鸿沟。



Together 2006 支持业务过程建模，并通过使能模型到模型的转换支持模型驱动架构（MDA）的设计和开发。据 Borland 的说法,这也是他们为开发人员提供的第一个可以在 Eclipse 开发框架下用到 Together 所有功能的版本。

“作为 IT 和业务用户联结的主要机制，业务过程出现了”，Gartner 主管 Internet 和 e-Business 技术的副总裁，Michael Blechar，在 Borland 的一次新闻发布会上指出，“这有助于保证软件开发工作基于正确的业务功能模型。另外，也为业务分析师提供了用他们熟悉的术语和工具描述过程的途径，其收益将在他们工作的一致性上表现出来”。

“在帮助软件组织克服障碍、开发高质量软件方面，建模扮演了枢纽角色。”Borland 负责产品的副总裁 Boz Elloy 在一个声明中指出，“新的 MDA 功能可以显著地加强架构演进过程中软件的弹性，通过引入这些功能以及业务建模能力和其它一些新功能，Borland 在单一易用的工具集中为顾客提供了接触最新的建模革新技术的入口。”

Together 2006 通过公共的可视的需求、架构和设计帮助分析师、架构师和开发人员保持同步，不论他们是否正在定义或加强业务过程、创建数据库结构或正在创建新的应用。Together 提供传统的 Together 技术，例如通过物理和逻辑数据模型支持数据建模设计，通过 use－case 分析帮助 IT 和应用分析人员定义和设计需求，帮助开发团队应用模型驱动开发的最佳实践。另外，Together 也提供了一些新的功能，包括全面的利用 Eclipse 集成框架、支持业务过程建模、模型驱动架构支持，以及模型和代码质量保证。

在 Borland 的记者发布会上，Modus Operandi 的工程副总裁 Alan McCutchen 指出，“作为在软件开发中应用 MDA 方法的 Together 用户，我们对 Borland 在 Together2006 中所做的增强感到非常振奋。通过整合 MDA 和 EII，我们看到了从垂直 Web Services 的扩展编码到关注于数据建模之上层次的面向对象集成方法的范式转换。这其中，我们的顾客得到的好处包括更快的 time-to-value，更强的跨越多个应用的潜在重用性，以及维护费用的下降。”

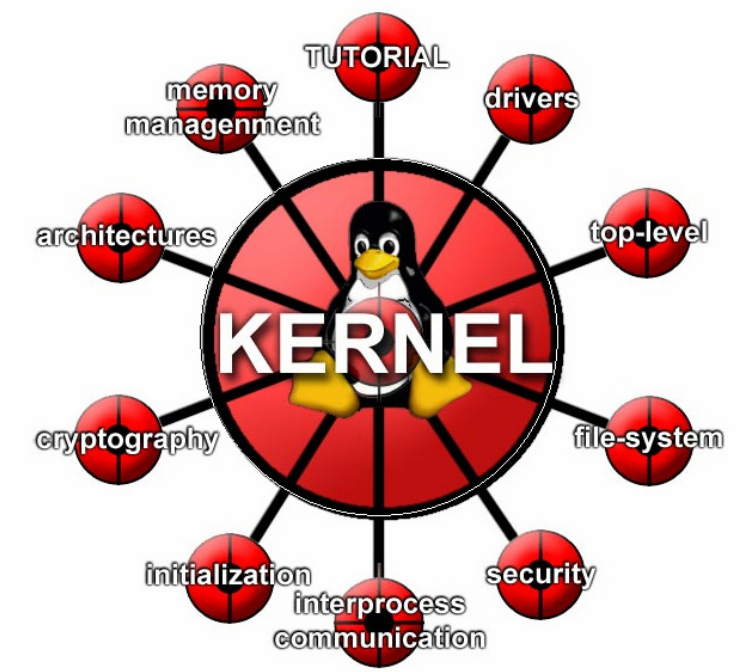
Together 2006 包括 Together Architect 2006 for Eclipse、Together Designer 2006 for Eclipse 和 Together Developer 2006 for Eclipse。

（自 integratedmar，袁峰 摘译，不得转载用于商业用途）

用 UML2.0 对 Linux 2.6 kernel 建模

[2005/6/30]

Software Revolution 公司应用 UML2.0 建模得到了 Linux 2.6 kernel 模型，这个模型包括各种图形以及文本，对 kernel 的各个主要子系统都进行了建模表示，包含了所有源代码的域和功能。



Software Revolution 认为这个模型是已有的 Linux kernel 模型中最完整和细致的模型，符合并超过 UML2.0 的要求。

Software Revolution 的 Linux 模型用到了 SVG（scalable vector graphics）来建模控制流图（control flow graph）、构造图（structure charts）、状态机、action diagram、data element table、call charts, complexity metrics, business rules, decision tables 和 data elements。图中包含了到源码表中相关行和功能的超链接。源码表也建立了索引和关联。

Software Revolution的设计级Linux Documentation文档可以在这里得到，同时提供有FAQ，bugzilla和教程。

Software Revolution 公司致力于将使用遗留语言书写的应用和方法转换为使用现代的支持 web 形式软件的形式。

（自 linuxdevices，袁峰 摘译，不得转载用于商业用途）

OBJECT-ORIENTED SOFTWARE CONSTRUCTION

SECOND EDITION

CD-ROM INCLUDED

- *The* Most Comprehensive, Definitive O-O Reference Ever Published
- An O-O *Tour de Force* by a Pioneer in the Field
- CD-ROM Includes Complete Hypertext Version of Book AND Object-Oriented Development Environment

BERTRAND MEYER

《面向对象软件构造》中译本，UMLChina 辅助教材

即将由 机械工业出版社 出版

The Unified Modeling Language Reference Manual, Second Edition

JAMES RUMBAUGH
IVAR JACOBSON
GRADY BOOCH

Covers UML 2.0

UMLChina 译
（王海鹏、李嘉兴）

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

CD-ROM included

《UML 参考手册》2.0 版中译本
即将由机械工业出版社华章分社出版

Domain-Driven DESIGN

Tackling Complexity in the Heart of Software

众多问题根源在于
领域模型没有捕获到
业务的深层内涵

Eric Evans

Foreword by Martin Fowler

《领域驱动设计》中译本

即将由清华大学出版社出版，UMLChina 审稿

模型驱动软件开发模式（下）

Markus Völter、Jorn Bettin 著，徐异婕 译

讨论

构建工作流（P）

要将开发过程规范化，工作流是个不错的范型 [注：RUP 中有需求工作流、测试工作流等] 。应使它可改变和敏捷。文档化的工作流有助于团队例行公事。没有什么比彻底常规化更有助于保证进度和交付质量的了。

2、领域建模

元模型形式化**

你正在为某软件系统族开发一个 MDSD 基础设施。你想利用合适的领域特定语言(DSL)来定义你的应用（即应用族成员）。

☆ ☆ ☆

一个领域通常包括领域特定的抽象概念、思想、以及正确性约束，这些都是用 DSL 开发领域应用是所必须的。如何确保你的 DSL、以及利用 DSL 来定义的应用模型对于特定领域是正确的呢？

DSL 可以以 UML 辅以 profile 为基础，profile 中定义了如何用经过版型定制的 UML 类（stereotyped UML classes）来描述领域概念。虽然 UML 允许定义在任何模型类之间的任何种类的关联，但它可能对领域毫无意义。只有在几种特定概念（经过版型定制的类）之间才可以允许某几种关联。为了得到正确的模型，必须考虑这些领域相关的约束。

因此：

应该采用形式化的方式描述元模型。必须明白无误地描述元模型，并使它可为“实现元模型（IMPLEMENT THE META MODEL）”模式所用，从而实际检查用 DSL 描述的应用模型。形式化的元模型应该作为“以元模型作为交流语言（TALK META MODEL）”模式的基础，这样才能在使用中检验它。

☆ ☆ ☆

有一些符号对定义元模型很有帮助。MOF 是很受流行的一种，在使用基于 UML 的 DSL 时尤为如此。如果 DSL 是基于扩展的 UML（例如使用 Profile），应尽量将元模型的范围限制在你希望明确支持的概念，而去掉所有没有用的 UML 的缺省特性。另一项有用的元建模技术是以特性建模（feature modeling）为基础的，当你主要根据特性配置应用时尤为有用。

要提取优秀的领域特定元模型和 DSL，领域经验不可或缺。然而，在许多组织中，模型驱动方法主要用于自动生成在特定技术平台之上运行业务逻辑所必须的“胶水代码（glue code）”。这样一来，你可能希望使用“以架构为中心的元模型（ARCHITECTURE-CENTRIC META MODEL）”。

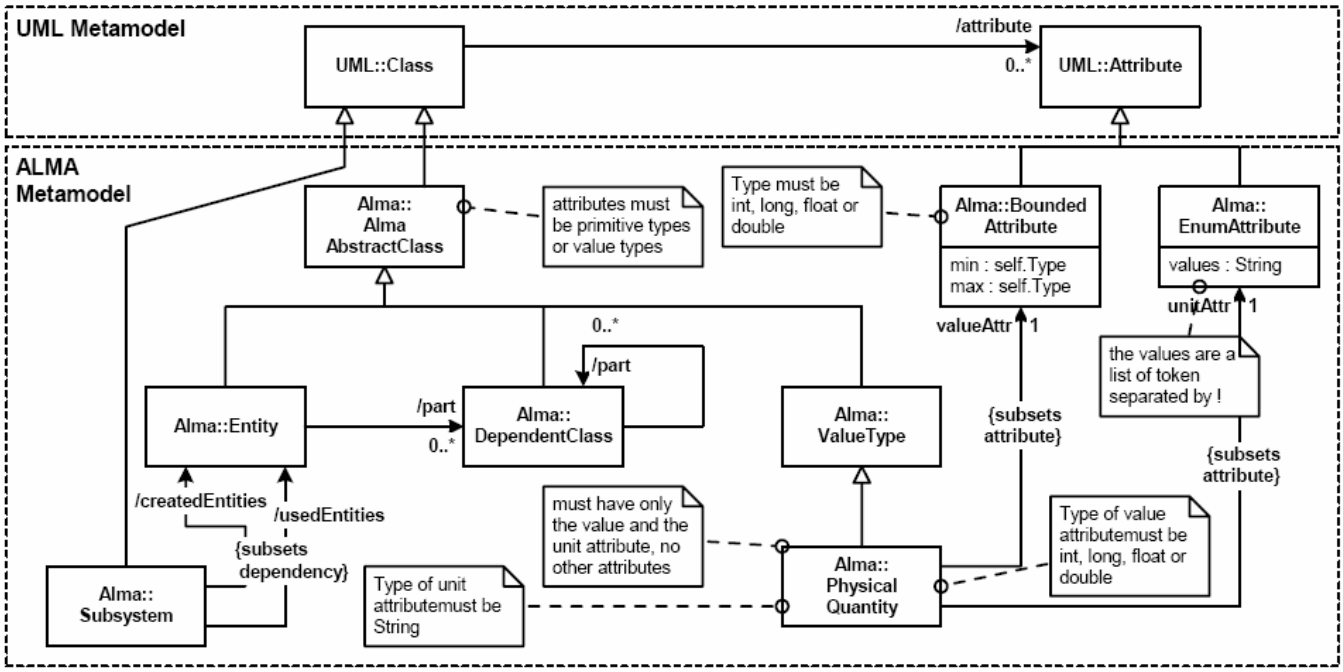
如果采用“以架构为中心的元模型”的方法，元模型的主要目的是加强架构相关的约束，并为设计者表示规范提供一种有效机制，这样可以摆脱具体实现语言的语法和执行平台所依赖的设计模式的束缚。

当构建元模型时必须理解领域。构建一个术语表或本体描述，可以作为第一步。当然，元模型是采用“迭代的双路开发模式（ITERATIVE DUAL-TRACK EVELOPMENT）”递增定义的。

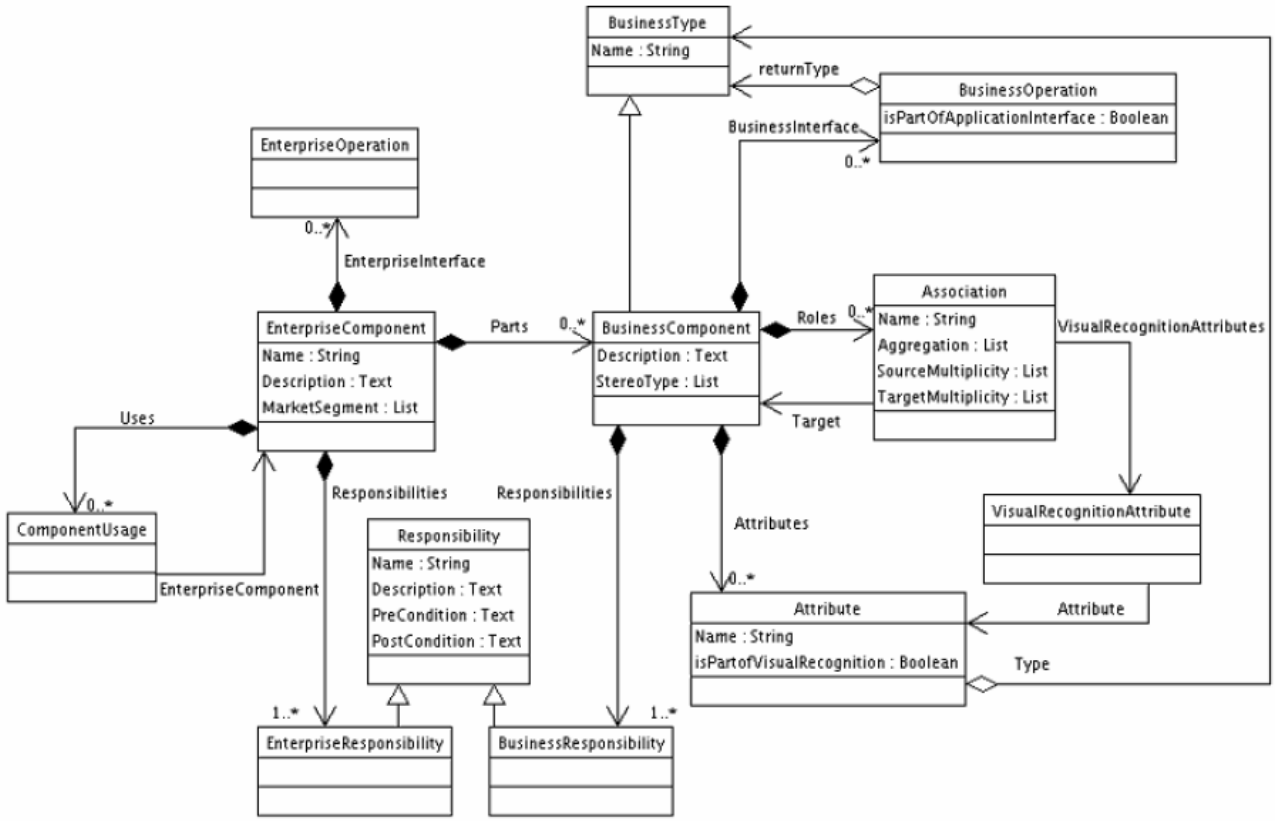
注意，“元模型形式化”是提取正确的 DSL 的重要前提；同时，它是“实现元模型”模式的基础；而且，它本身还是领域特定工具的基础。然而，你仍然需要确认元模型实际描述的真实领域是正确的，你可能希望用“以元模型作为交流语言（TALK META MODEL）”模式来完成这些工作。

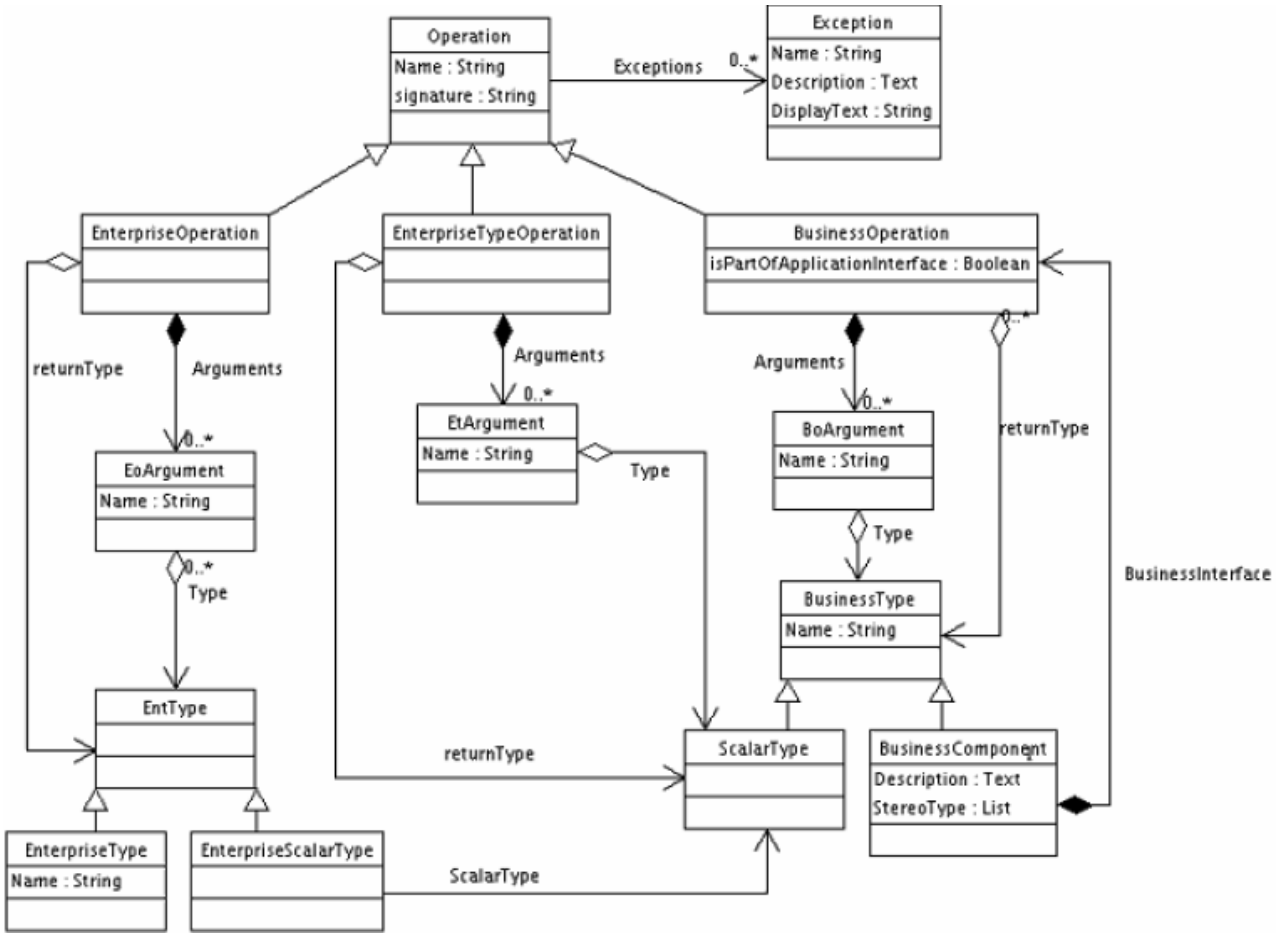
☆ ☆ ☆

ALMA radio astronomy 项目利用下面定义的元模型来定义它的（部分）数据模型。我们以迭代方式开发元模型，并在最后用下面的格式来正式归档。代码自动生成工具也可以实现元模型。



下面两张图描述了一个为大型分布式商业应用精心制作的元模型。从[Bettin 2003]可以了解更多的信息。

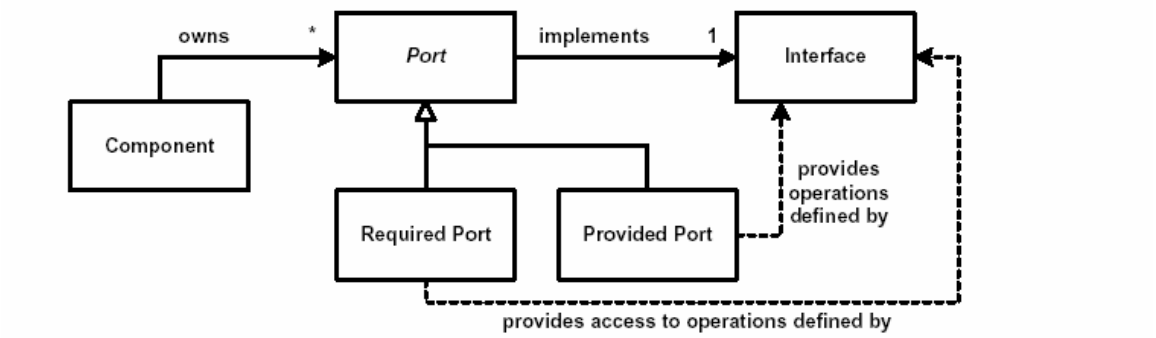




当你写一本关于模型驱动开发的书时，形式化的元模型也很有帮助。我们想比较 MDSD、MDA、GP 以及“以架构为中心的 MDSD”等方法的相同点和不同点。这个重大成果会在为 MDSD 领域提取形式元模型时给出。

以元模型作为交流语言(P)

持续改善和验证一个领域的“形式化元模型”非常重要。领域专家和开发团队都应使用它。为此，和涉众讨论期间以元模型作为交流语言是个好办法。



考虑上面例子中的元模型，和涉众讨论时，应该使用类似下面的句子：

- 一个组件有多个端口。
- 每个端口明确实现一个接口。
- 有两种端口：所需端口和提供端口
- “提供端口”提供由它的接口定义的操作。
- “所需端口”提供由它的接口定义的操作访问点。

一旦你发现基于元模型的句子无法表达某些事物时，或许：

a） 你必须换一个句子；

b) 这个语句是错的；

c) 必须升级元模型。

上述技术是一个著名的领域建模技术，和 Eric Evan 的[Evans 2000]一书中所述的“普适语言（ubiquitous language）”模式有关。这本书中包含了许多其它有用的技术，毕竟，元模型也只是建模中的一部分。

以架构为中心的元模型(P)

万一你不能得到一个精密复杂的领域特定元模型，你可以直接用元模型来表示技术平台的概念。这样一来，成了以架构为中心的模型驱动开发，再就是转换相对简单。随着时间的推移，越来越多的领域特定概念被添加到元模型中，模型更加易于表达，更少的依赖技术平台，但需要越来越复杂的转换。

3、工具架构

实现元模型**

你有一个领域的形式化元模型。

☆ ☆ ☆

在你的领域中已经存在一个定义好形式化元模型，这是件好事情；但是，在开发已定义的产品线中的部分应用时，需要高效的使用它。如果元模型归档后就成了尘封的纸张，那么它的作用就消失殆尽了。

用手工方式检查元模型是错误的，而且很乏味。而使用现成的建模工具也是徒劳，因为它不能理解你的元模型(这种情况可以随时间推移发生改变)。即使 UML 工具理解你的元模型，也只能检查“基于模型的 UML(UML based models)”。

然而，为了确保自动生成器能够产生并且配置应用，必须确保你的模型“满足”元模型定义。

因此：

应该用一些工具来实现元模型，可以读取模型并可以对照元模型来检查它。检查涉及包括声明约束的每件事物。确保仅当模型符合元模型时，才进行模型的转换。

☆ ☆ ☆

这种方法是符合 MDSD 的，只要元模型不只是张“图片”，而是 MDSD 流程中有用的东西。你可以使用 MDSD 方法从原模型本身产生元模型的实现，或者手工实现它。

元模型的实现是转换引擎（transformation engine）或代码产生器的典型部分，因为正确的模型是成功转换的前提。

☆ ☆ ☆

B+m 自动生成器架构[GenFW]支持使用 Java 类来实现领域元模型。每个元模型元素作为一个 Java 类来实现。当模型被读取，模型被描绘成元模型元素的实例。因为元模型类可以实现一个被架构调用的 CheckConstraints() 操作，这样可以很容易实现约束检查。自动生成器缺省使用 UML 元模型，开发者可以对这个元模型扩展或替换。

LANSA RUOM 不仅提供 OO 建模的能力，还提供了一定的元建模能力，主要集中在架构约束的定义上。LANSA RUOM 支持不同用户定义（用户可定义）类型的组件之间的可允许的依赖，这样就防止了在 RUOM 建模工具中定义不合乎规定的依赖。这种方法显然不同于标准 UML 建模工具的方法，标准 UML 的方法事实上没有约束检查，只有是否符合 UML 语法的检查。注意[Bettin 2001]的一个例子，它符合标准 UML 的语法，但妨碍了真实的表达架构。

来自 Gentastic 的 eGen 自动生成器支持可视化的元建模，并且自动生成一个设计的入口，它可以限制约束必须与元模型中关系一致。这种方法对于领域特定元模型来说是完美的。在这个特定例子中，主要的缺点是自动生成的设计入口的质量。GMT Fuutje 工具的当前版本支持沿着 UML 标记值这条线进行元建模，元建模只需进行很少的编码。也就是说，元模型元素可以用附加的元素来进行扩展。更广泛的元模型变化需要以 java 代码的形式来实现，也许和利用 b+m 自动生成器的方法有点类似。

Codagen Architec 自动生成器依赖 UML 元模型，并且依赖标记值从标准 UML 工具到自动生成器被传递。这种方法不考虑在设计时期用 UML 工具进行任何约束检查，仅在自动生成时期检查 UML 模型中“不正确的”标记值。

忽略具体语法**

你希望转换模型或从模型自动生成代码。

☆ ☆ ☆

每个模型必须用具体的语法来描述（例如：用 XMI 描述基于 MOF 的模型）。但是，当你真正想转换元模型的实例时，根据模型具体语法细节来定义转换，却使得转换笨拙并且混乱。如何能确保转换不依赖具体的语法呢？

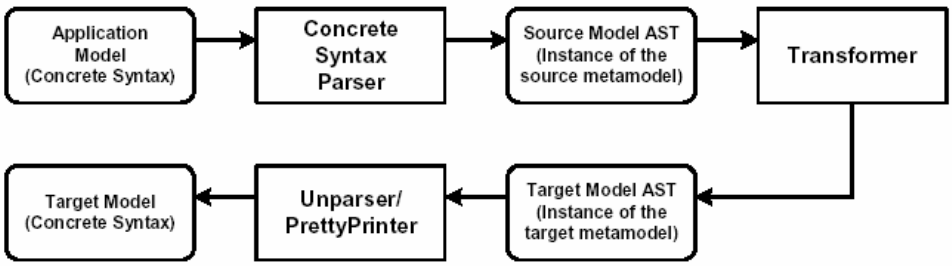
典型情况下，基于具体语法来定义转换很可能造成效率低下。假设使用 XMI；XMI 语法非常复杂，基于 XMI（也可能是 XSLT）来定义转换不是非常高效。

同样，在许多情况下，对相同的元模型会有多种具体语法可用。例如，如果你为不同“技术子域（TECHNICAL SUBDOMAINS）”使用了不同的 DSL，那么基于具体语法定义转换未必要将一个特定具体的语法绑定在这个转换上。

因此：

应该基于源（source）和目标（target）元模型定义转换。转换器应使用三个工作阶段的方法：

- **首先，将输入模型解析（parse）成一些元模型的内存中的表示（特别的是一个对象结构）。**
- **然后，将输入模型转换成输出模型（仍作为一个对象结构）。**
- **最后，将目标模型“解析回（unparse）”成具体的语法。**

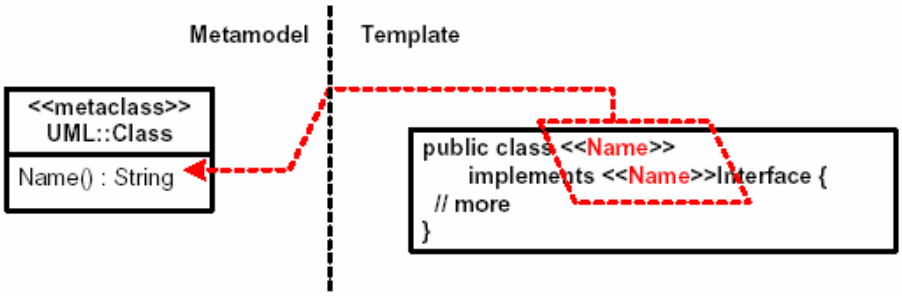


☆ ☆ ☆

采用这种方法说明转换，要高效得多。它也使得转换器更加灵活，因为它可以和任何种类的具体语法一起工作。对于基于 XMI 的具体语法时，它非常重要，因为 XMI 的细节会根据 UML 工具不同而发生变化。你并不希望将转换绑定到特定工具上（甚至工具的特定版本）。

代码自动生成器（将模型转换成代码）经常不采用完整的三个阶段方法，而是直接从输入模型实例自动生成原文的输出；产生输出 AST 实例将极度的复杂。相反，被使用的模板接近“源元模型”。注意：这种方法和“实现元模型（IMPLEMENT THE META）”模式是“好搭档”。如果做得对，为了双方的目的，将采用同样的实现。然后模板可以直接接近元对象；元对象的属性可以被用来为模板赋值提供数据，在下面的插图中显示。

另外，值得说明的是，编译器使用该方法已经由来已久了。编译分为几个阶段，第一个阶段解析具体的语法，并在内存中构建一个抽象语法树，在随后的阶段操作语法树。



☆ ☆ ☆

上图对于大多数基于生成器或转换器的模板语言来说，非常有代表性。

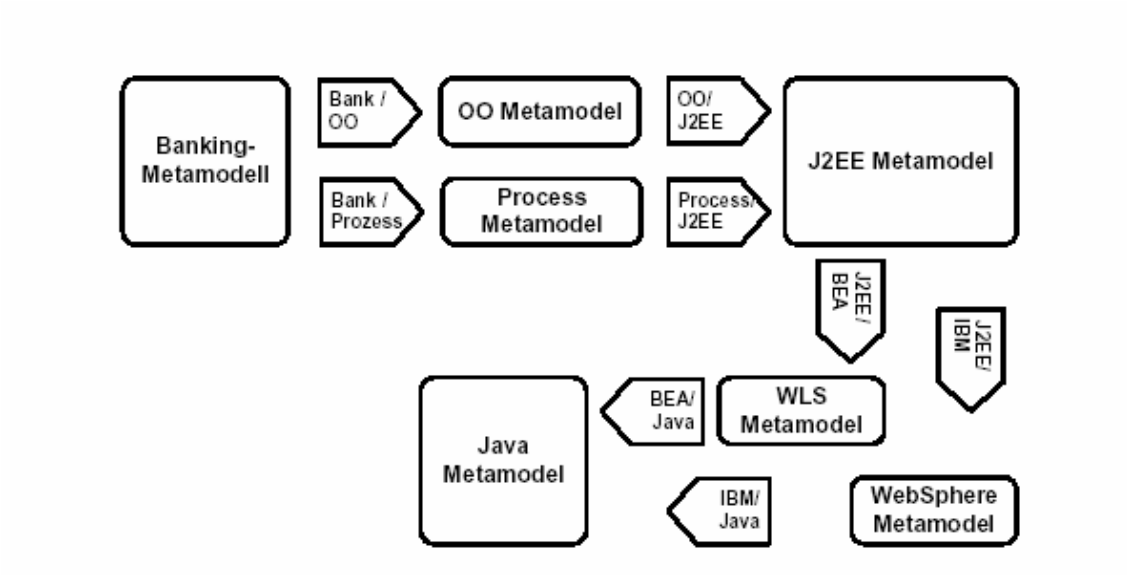
再次研究 b+m 自动生成器框架，我们已经看到它用 Java 类来描绘应用元模型。这个转换是基于模板的（因为它直接自动生成代码）。这个模板可以包含涉及元模型和它的属性的声明。你看不到模型的任何具体的语法。前后端负责为解析具体语法和实例元模型元素。

例如 B+m generator、Fuutje GMT、Codagen Architect、eGen 和 LANSA RUOM 等，都使用模板语言来为用户屏蔽模型的具体语法。在 LANSA RUOM 中，模板语言十分的薄弱，这个模板语言通过完全支持用户可定义的预模板处理器来补偿，在代码自动生成时期，在具体的模型语法和模板变量之间履行转换的角色。在 eGen 中，可用的模板变量（template variables）和导航（navigation）是直接通过对用户可定义元模型（user definable meta model）的直接反射（reflection）完成的。

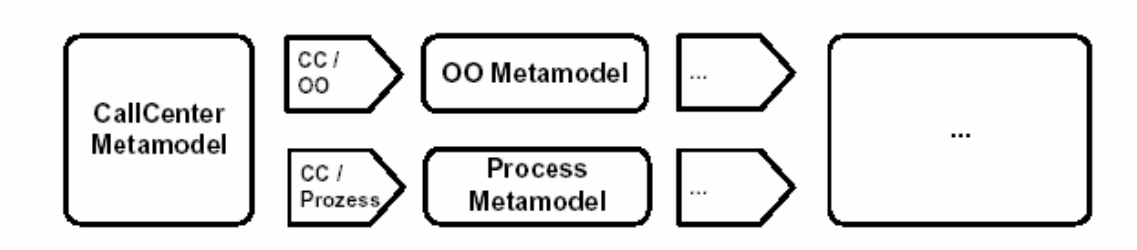
就我们所知，大部分现今的 MDA 工具依赖不标准的模板语言，用来将模型转换为文本工件，比如代码。这些模板语言的局限性和被提议的可选择方法被粗略的补充在[Bettin 2003b]。

模块化、自动化转换(P)

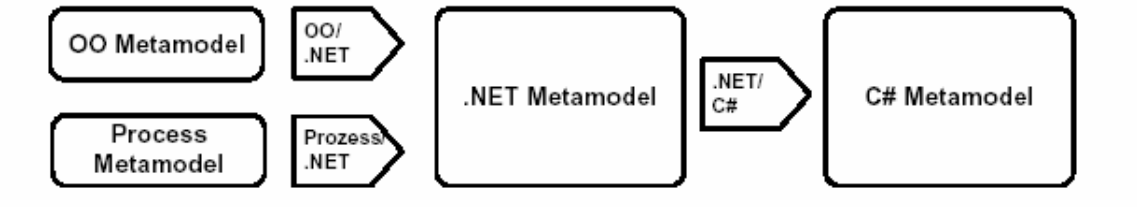
为了便于重用转换中的部分功能，将转换模块化是一个好注意。考虑下面的例子，我们从一个银行特定模型中自动生成 J2ee 代码。



现在设想我们想为呼叫中心应用自动生成 J2EE 代码。因为转换是模块化的，我们只需要交换第一部分。所有进入后来转换的工作能够保持不变。



最后，如果我们想把软件系统家族迁移到.NET 上，我们只需要替换后面部分即可。



如果你只有一个单独的直接转换，重用就不大可能。注意和 OMG 形成对比的是，我们不推荐考虑改变和标记中间的模型。在转换之中，它们仅仅是为改变数据的一种标准格式。如果我们需要模型提高控制或配置部分或全部转换的程度，需要“外部模型标记（EXTERNAL MODEL MARKINGS）”模式的帮助。

转换作为一等公民（P）

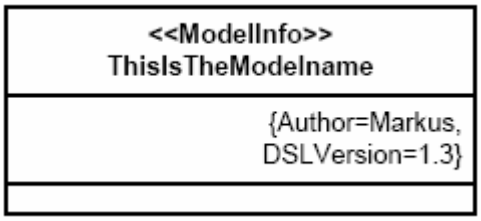
在 MDSD 中，转换是重要的内容。它不应作为“二流观念”。你必须对它们进行重构、归档、标记版本、并考虑如何将不同的分支进行合并，等等。简而言之，你应该考虑转换作为“一等公民”并真正重视它们。

面向方面的元模型（P）

如果你实现元模型，千万不要将问题领域元模型和解决方案空间的功能混为一谈。使用 AO（ASPECT-ORIENTED）技术将它们区分开来。

模型中的叙述性信息（P）

为了便于版本化和多版本自动生成，你必须确保：自动生成器知道一个特定模型 DSL 的哪个版本要被构建。为此，应该将叙述信息嵌入模型之中，比如作者、版本状态等等。在 DSL 中使用定制语法，增加这些信息并不费事；但在 UML 中，这样做却开销较大，至少如果象保持这种方法可移植并不容易。下面的方法兼容所有工具：为类增加预先定义的、由标记值表示的构造型（stereotype）。下面是一个例子。



4、应用平台开发

两个阶段构建*

你的工作内容和软件系统族相关，并且需要设计一个模型驱动的自动生成器。

☆ ☆ ☆

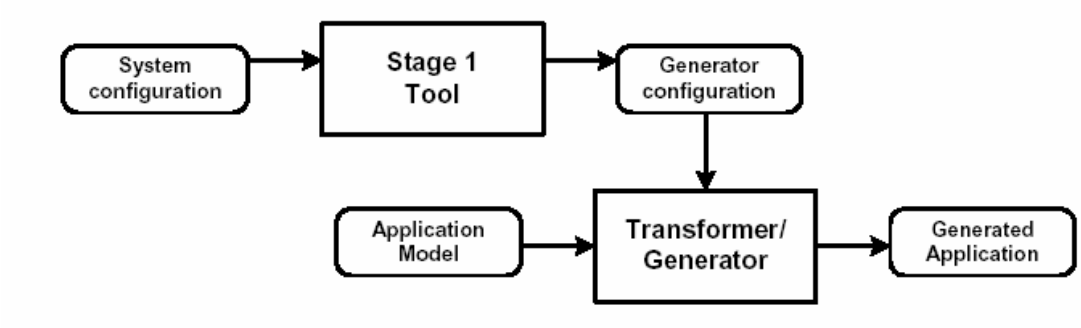
将所有的产品配置步骤合并到一个转换中，通常会非常复杂。不同特性彼此依赖。系统的不同部分被指定采用不同方法。如何能够构建一个简单、可持续的、可以修改的转换过程呢？

考虑目标平台的选择。执行不同的转换应当视平台而定。因而，平台的选择决定哪个转换被执行。将所有选择合并到一套转换中，它可以照顾到所有的可能性，这非常困难。

另外，还存在库或横切关注点选择问题。

因此：

将代码自动生成分为两个阶段：第一个阶段是读取某些配置并为核心转换准备实际的自动生成器。第二个阶段是执行转换并利用在第一阶段中准备好的内容。



☆ ☆ ☆

在许多情况下，第一个阶段使用不同的工具（例如一个批处理文件或一个脚本）来准备自动生成器本身。同样，虽然第二阶段的模型经常描述应用功能和结构，但对第一阶段的说明实际上更多的是工具配置的活动。

这样一来，就没有一个放之四海而皆准的转换或模板代码管理的范型（paradigm），而是每个工具各不相同。通常，相对于领域架构的影响，工具架构对方法的影响更大。因为许多工具采用基于文件的方法，下面给出的例子具有代表性。

注意：这种方法也可用于开源代码的分发，只须在构建应用的第二步用 make install 来准备 makefile。

☆ ☆ ☆

例如，在小型项目中基于 XML 的规范用于定义目标平台。基于此情况，ant 工具可用于准备自动生成器运转的环境：特别是，它将一套可应用的模板文件，从生成器装载它们的位置拷贝到本地。在第二个阶段，自动生成器本身处理模板，并因此从应用模型中自动生成代码，这是个 UML 和 XML 的组合。

正如前面谈到的，在 LANSA RUOM 中，模板处理前，有处理器读取模型，为每个模板搭建模板执行环境。前处理器也是模型驱动整合的地方，因为他们可以根据需要访问与先前系统相关的元信息，并使得这些信息对 RUOM 模板可用。LANSA RUOM 另外的有趣的特性是，模板处理之后的后处理器，它支持利用任意的代码对用户可定义符号（预模板处理器命令）进行置换。这些符号不一定需要在模板中表示，但可能作为自动生成代码的一部分——通过模板变量内容的估计产生。这样，在 LANSA RUOM 中 build 包含三个主要阶段。完全支持递归模板运行这个特性在模板语言中不是必需的。总体说来，在工具准备和调整模板运行方面有重要的差别。例如在 eGen 中，连接模板的代码与模板代码从物理上来说是分离的。在 Codagen 的架构中，模板执行顺序和模板之间的关系被间接的指定，经由模板的分类表和模板列表的排序。所有工具的公共主题是“多阶段”构建。

隔离生成和非生成代码**

你可以自动生成应用的大部分内容了，可你仍然必须要手工编写一些方面的代码。

☆ ☆ ☆

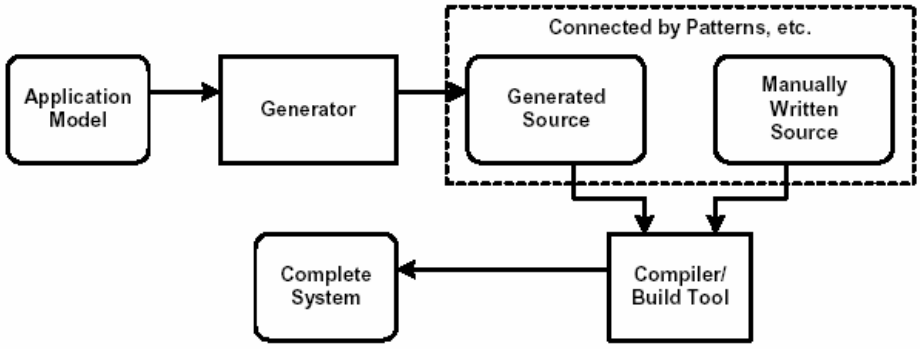
如果只是应用的部分是自动生成的，那么“其余”部分必须由手工编程来填满。可是，通过增加非自动生产代码来修改自动生成的文件，会带来一致性、build 管理、以及代码重组时人工代码的版本化和写的过多的问题。

如果自动生成的代码从不需要修改，需要时整个自动生成结果可以简单的被删除和更新。如果代码被修改，必须有一个保护区域，当更新代码时，自动生成器不能删除这个区域的代码；这要求在更新之前，自动生成器实际上来重读自动生成的代码，并保护受保护的区域。一致性的问题将会出现（当非自动生成部分出现矛盾，导致模型发生改变时）。

另外，版本管理是非常复杂的，因为从模型中手工编码和自动生成代码是用同一个文件，尽管他们将分别的进行版本控制。

因此：

应该将自动生成和非自动生成代码保存在不同的文件中。永远不要修改自动生成的代码。设计一个框架来清楚的定义哪些工件是自动生成的，哪些不是。使用合适的设计方法，来“连接” 自动生成和非自动生成代码。接口和设计模式诸如工厂、策略、桥接或模板方法都是一个好的起点(see [GHJV95])。



☆ ☆ ☆

使用这些模式的结果是：应用被迫使用一个好的设计，这样可用清楚的区分不同的方面。自动生成代码被作为“可抛弃工件”，甚至不需要对它进行版本控制。一致性问题因而不再出现。

本模式很关键，还因为：当执行一个产品或产品线家族中的一个新的变更时，通常需要改编手工代码（实际上不是自动生成的）。因而，对于有效的管理变更和帮助识别变更点，区分自动生成代码和非自动生成代码是必需的。

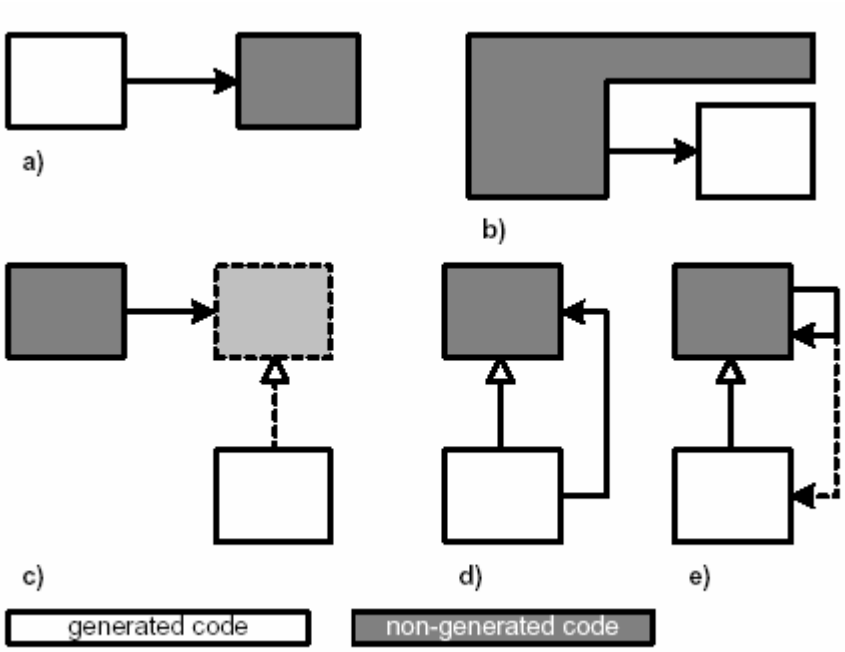
至于缺点，这种方法可能需要多一点精心设计，或者多一些（手工）编程。

有时，由于执行的原因，直接将手工代码插入自动生成代码中，这种情况是不可避免的，这样一来，就必须引入保护区。

在许多情况下，这种模式非常普遍，你有几种生成器自动生成全部系统的不同部分，例如在“技术子域（TECHNICAL SUBDOMAINS）”的上下文中。手工编写的代码可以看作有非常特殊的生成器（编程人员 ）产生的，架构显然要迎合这一不同的方面。

☆ ☆ ☆

下图显示了如何使用上文中提到的模式，将自动生成代码和非自动生成代码组合在一起。



首先，生成代码可以调用包含在库中的非自动生成代码（情况 a 中）。这个用法很重要，因为它主要的告诉你要生成尽可能少的代码，并且依靠被自动生成代码使用的预实现组件。正如 b 中所示，相反的过程也是可能发生的。非自动生成的框架可用调用自动生成的部分。为了能行的通，非自动生成源码可对照由自动生成代码实现的抽象类或接口来编程。工厂方法被用于“插件”——自动生成的积木块中，c 图中举例说明。

自动生成类也可以成为非自动生成类的子类。这些非自动生成基础类可以包含有用的普通方法，它们从自动生成子类中被调用（d 中所示）。这个基类也可以包含它所调用的抽象方法，它们被自动生成子类（在 e 中所示的模板方法模式）实现。此外，工厂方法对插件实例化也很有益。

功能丰富的领域相关平台**

你正从领域相关模型中产生代码。

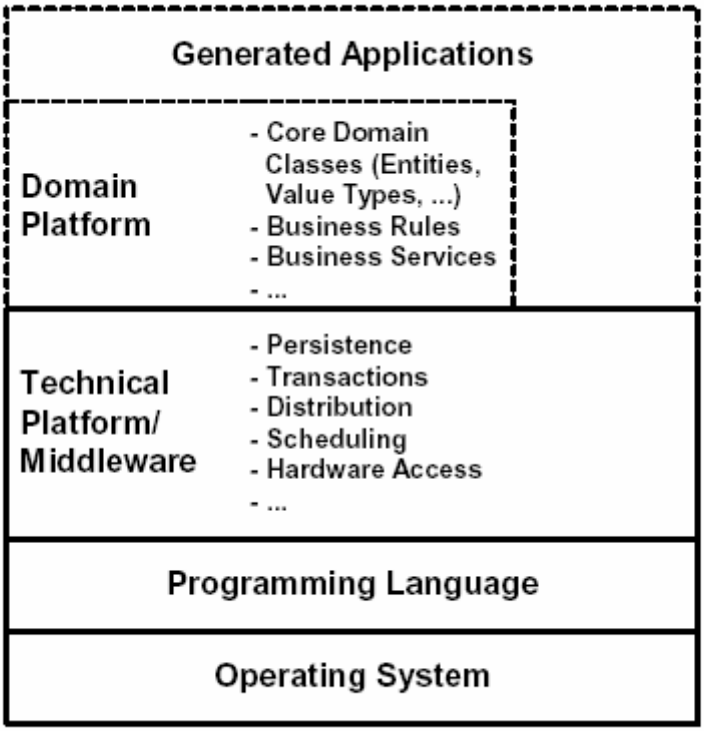
☆ ☆ ☆

最终，应用模型一定被转换成一个用来执行的确定的目标平台。领域概念和目标平台的差距越大，转换的过程就越复杂。就目前的工具而言，这还是个问题。

转换应该简单易懂才能切实可行。这主要是因为现在支持“传统”开发的开发环境（集成开发环境、向导、调试工具等。）比过去精良得多。更多的工作可以按“常规方法”完成，并且完成得更好。这里有一个关于这个问题的很好的例子，那就是相关对象的映射工具。OO 方法和相关数据模型之间的不匹配产生的阻力是主要的问题，尽管这个问题刚刚被提出，它现在已经得到了很好的解决。

因此：

应该定义一个包含库、框架、基础类、解释器等在内的功能丰富的领域相关平台。转换将为领域相关应用平台“自动生成代码”。



☆ ☆ ☆

自动生成的代码不只包含“真正的代码”，还包含配置文件，部署信息和其它被用于领域相关平台中的工件。随着你对领域认识的深入，以增量的方式增强领域相关平台(frameworks， libraries)的能力。这降低了需要被自动产生（有时甚至会手工完成）的“框架完成代码”的数量和复杂度，转换变得简单一些了，这正是我们希望的，毕竟当前的工具有太多限制。

注意：不能过度使用模式，否则，我们将退回到一般的开发。你应该用合适的 DSL 来尽可能的模拟你的商业逻辑，并生成实现。另外，针对模型化应用的所有胶水代码或配置数据都将被生成；使用模型（LEVERAGE THE MODEL）!

一旦应用平台逐渐变得和用 DSL 描述的概念足够接近，转换的复杂度将降低。生成器被限于生成重复的胶水代码。只有工具的支持仍然有限，这种方法就会有实际意义。最后，这种方法将支持你使用“以架构为中心的元模型（ARCHITECTURE-CENTRIC META MODEL）”模式。

应用平台设计的关键是：通过“迭代的双路开发（ITERATIVE DUAL-TRACK DEVELOPMENT）”模式，来进行迭代和增量的开发。事先将框架设计得过细，一向是要失败的。相反，结合代码自动生成的小型框架为迭代开发提供了扎实的基础。当自动生成对实现有困难时，通常通过改进框架来解决。相反，当实现框架特性有困难时，通常自动生成技术可以提供很好的解决方案。随着你对于系统的理解加深，你将在 DSL/generator 和平台之间来回地重构。

注意：作为结果，你可以在领域元模型中应用平台的核心概念。大体上来说，DSL 和 framework/platform 会被作为一个问题的两个方面来考虑：framework 提供核心概念和功能，反之，DSL 通常“利用”特殊应用场景中的这些概念。

☆ ☆ ☆

出自 SoftMetaWare [BH 2003]的 Time Conscious Objects (TCO)工具包是一个关于框架和自动生成技术互补的很好的案例。其中，框架为“time”概念提供支持，在元模型的支持下，通过使用不同的“时间意识（time consciousness）”级别，“time conscious classes”可以被标记为很高的抽象度。

以架构为中心的 MDSD，受到 b+m 和另外一些 MDSD 务实派的提倡，并明确针对领域元模型来描绘平台架构的核心概念。

提供了一个领域特定语言的例子，这个例子严重依赖[Bettin 2002]中提供的“功能丰富的领域相关框架”。在这个例子中，DSL 是一个对于面向对象用户界面行为说明的可视化符号。

在“Zemindar”项目中，本文作者之一运用 DSL 实现了在运行期间终端用户的复杂算法和统计函数的设计。在这个例子中，DSL 不必发明创造，并且第三方的现货供应的 Java 电子表格组件被作为 DSL 在项目中应用。这个项目也用到了模型驱动产生器，但重新从头实现一个电子表格功能的框架是很不切实际的——甚至是更加糟糕的，应该尽量“产生”这样的功能。

技术子域**

你正在使用 MDSD 构建一个大型复杂软件系统族。

☆ ☆ ☆

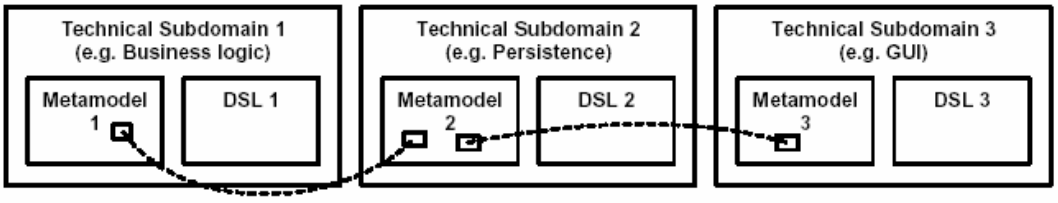
大型系统由它们涵盖的多个方面（aspect）组成。用一个全面的模型来描述所有的方面是个非常复杂和让人忘而却步的任务。这个模型将变成复杂和充满多个方面细节的。同样，用 DSL 可以描述一个方面，但可能对描述其它方面并不合适。

考虑使用基于 UML 的 DSL 来描述应用功能（业务逻辑）。另外，你现在也只能描述“持久化方面（persistence aspects）”、以及 GUI 设计和布局。你将只能简要说明在 DSL 中持久项，比如：被生成的持久使用的适当的代码和表结构。将所有的放进同一个模型中是非常困难的。基于 UML 的语言典型的不适合其他的方面。试图用 UML 来为 GUI 布局建模简直不可能。

另外，将所有的内容放进一个模型会使维持复杂化，阻碍为不同小团队进行工作包的直接分离。

因此：

将系统放在不同的技术子域中。每个子域都应该有它的元模型，特别是有它自己的 DSL。定义少数“网关元类（GATEWAY META CLASSES）”，它们会出现在多个元模型中，从而帮助你将不同方面（aspect）连接在一起。



☆ ☆ ☆

如果你在转换引擎中采用“忽略具体语法”模式，这种模式尤其有用；这是因为，虽然在转换器内部的描述是相同的（并因此提供了不同元模型天然的集成），但你可以使用不同领域的具体语法描述网关元模型元素。

注意：这个模式将系统的几个部分安排到几个技术子域中，并没有将整个系统构建到不同的功能包中。后者当然也很有用并应当被采用。

“技术子域”的一个特殊情况是“模型驱动的集成（MODEL DRIVEN INTEGRATION）”。要指定映射和包装规则，可以采用适当的 DSL 来详细地给出规约。“基于产生器的 AOP（GENERATOR-BASED AOP）”模式可以作为处理交叉“技术子域”的方法。

☆ ☆ ☆

出自 SoftMetaWare [BH 2003]的 Time Conscious Objects (TCO)工具包被明确设计成，将现存架构作为一个技术子域来对待。TCO 假定已存在系统是基于任意 OO 建模语言（UML 或普通 Java 代码），并且，TCO 非常简洁的 DSL 支持用户使用对象的“时间意识”级别信息来为模型增加注释。

有个小的组件项目，使用基于 UML 的 DSL 来规约接口、依赖、操作和组件。其中，定义系统配置、组件实例位置和诸如远程调用（remoting）中间件的技术方面配置，使用了完全不同的基于可匹配 XML DTD 的 DSL。

DSL 可以用于规约面向对象的用户界面中的行为[Bettin 2002]，这样一来，可以结合诸如标准 UML 的其他建模语言来规约对象结构。

模型驱动的集成*

你需要将已存在系统和基础设施，与你的以模型驱动方式开发的软件集成。

☆ ☆ ☆

从零开始开发的项目很少，大多数新软件都是在已存在一到多个运行已久的系统的情况下开发的。另外，也常希望以增量的方式，用更符合业务需要的、采用当前技术的软件，逐步淘汰遗留系统。于是，集成不同的、或新或旧的系统成了许多项目的一部分，无论它是不是模型驱动的。

根据集成策略不同，代码可能针对当前技术产生，也可能针对被集成系统采用的较旧的技术产生。产生的工件还包括进行数据转换的一次性脚本。典型的，集成要采用系统化的方法考虑 API 映射。

因此：

应该根据要集成的软件系统的领域，扩充模型驱动的软件开发范型。当采用模型信息是，在系统直接进行信息映射非常有价值。为“模型驱动的集成”定义“技术子域”；如果太复杂，就考虑每个系统的“技术子域”。定义这些领域的 DSL，这样你就可以表达业务领域与遗留系统直接相关元素之间的映射。然后就可以自动切断遗留系统。

☆ ☆ ☆

与现存系统集成，是模型驱动方法的强项——有时被误认为弱项。

就集成两个不同的模型驱动系统而言，将两个系统之间的集成代码隔离，会使模板定义不必重复。

对简单的集成问题，采用“技术子域”模式似乎小题大做了，用 UML 标记值或 DSL 中的对等概念捕捉两个系统之间相关元素的映射就足够了。仅当信息并不打乱领域模型，或者是不准备淘汰的遗留系统或子系统之间的集成，才采用此方法。

特别地，如果遗留系统计划逐步淘汰，应保证集成代码在不需要时能容易地去除，否则随着时间的推移，这些无用的代码会拖累系统架构。可使用[Evans]所述的“防腐层（Anticorruption Layer）”技术。使用“外部模型标记（EXTERNAL MODEL MARKINGS）”模式规约映射。

想象一下，要将遗留系统的逐步淘汰自动化，到达到使用相关子域的 DSL 标识“淘汰部分”的程度。要成为好公民并使下一代人生活幸福——应切记可能三年以后才淘汰遗留系统的最后一部分，那时人们对你的集成代码可能一无所知。

☆ ☆ ☆

多年以前，本文作者之一使用该模式进行遗留基础设施的集成，为安全起见还结合了 LANSA RUOM。那次，RUOM 的模板前和模板后处理器是关键。

基于产生器的 AOP*

你正使用 MDSD 技术开发软件系统（族）。你通过代码产生器产生实现代码。

☆ ☆ ☆

很多应用中，横切关注点必须始终如一地以局部化的方式处理。编程语言不提供如何将这些关注点模块化，而增加其他的工具（例如方面织入工具）又常常因为支持不足、工具能力不足、或开发者技能不够，而不能成功。那么你又如何以一致的方式处理横切关注点呢？

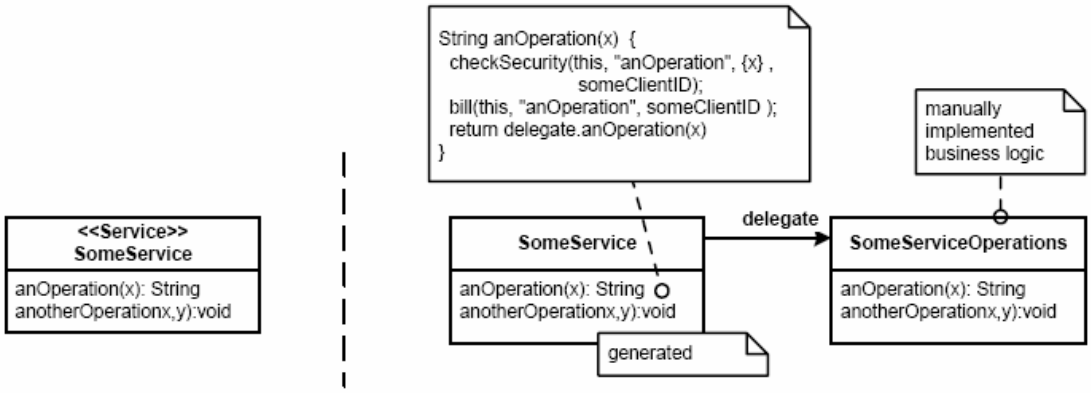
若业务应用需要对多客户有效时，常需要记录每个客户的使用情况，从而需要记录每个操作的执行并判断每次调用的开销。

另一种横切的方面需要在这种场景下——如同很多其他场景——被授权（检查客户是否有权访问某功能或读、改、删某信息）。可能仅允许客户查看它“拥有”的数据，也可能限制客户仅使用应用所有功能的一个子集。

为保证一致性，你希望确保这些方面（aspect）无需由应用开发者手工处理，而是某种形式的 AOP 来集中处理横切关注点。

因此：

依靠生成器来实现横切关注点处理。你要么利用生成器的部分特征（例如借助转换或模板，用生成器产生一个元模型元素的很多实例），要么利用生成器实现代理、解释器等支持 AOP 的设计模式。假设横切关注点是“技术子域”并提供了合适的 DSL。



☆ ☆ ☆

采用本模式，你不必使用另外的工具（如 aspect weaver）就能处理横切关注点。当然，处理所有种类的横切关注点是不可能的；诸如 AspectJ 之类的 aspect-weaver 在语言级进行处理，更强大些，也更通用些。然而，许多情况下基于生成器的方法就足够了。另外，你保持了调整生成器结构的自由（并且可能可以调节应用平台架构），从而处理你要求的方面。

☆ ☆ ☆

该方法在一个 EJB 项目中用于产生上述那种代理，实现了动态安全性检查，有不错的权限审核和日志记录功能。

尽可能产生易读代码**

你从模型产生应用代码。

☆ ☆ ☆

很多时候，开发者不看产生的代码是不切实际的。尽管他们不改这些产生代码，但当调试应用或确认转换引擎配置时可能要读这些程序。你如何保证开发者切实理解了产生的代码，而不是惧怕这件事情呢？

有种偏见很流行：不必阅读、处理、调试生产的代码。有时这种偏见甚至就是人们采用代码生成、而不采用模型驱动开发的原因。所以，克服这种偏见很关键。

因此：

尽可能产生易读代码！设计代码生成模板时，也须时时考虑开发者必须——至少某种程度上——处理这些产生代码。

☆ ☆ ☆

你可以关注下列事情，来使产生的代码易读：

- 产生注释。在模板中，即使不注释所有信息，也应该尽量多地提供信息。典型地，你甚至调整产生代码的注释。
- 既然典型而言很多工具支持的“空白管理”功能并不充分，所以你必须决定让你的模板易读、还是让产生的代码易读。保证模板易读并使用优良的打印或格式化工具在代码产生之后调整格式，是个好办法。该打印工具对每种编程语言都是有的，比如 XML 等。
- 第三种方法是模板或转换产生的代码的　位置字符串　。它描述了模型元素从产生的代码的哪个特定节开始。这是个好方法，特别是调试时会获益匪浅，并包含了模板或转换的名字和“上次改变”时间戳。例如：SomeOperationStereotype [2003-10-04 17：05：36] FROM MODEL ELEMENT aPackage：：aClass：：SomeOperation()。

采用该模式有个不小的困难，它会使产生的代码和代码格式密切相关。特别是缩进很关键。

另外，将不错的原型模板化时总应随手加注释。

该模式应该切实用于产生的代码和手写的代码。实际上，对代码进行太多手工改造会很杂乱。注意：有时需要进行代码优化，结果可能不大易读，这时应显式标识和描述，并与其他代码分离。

☆ ☆ ☆

希望你不仅仅是说“哦，我们也这样做吗？”

描述型元对象**

你正使用模型驱动技术开发软件系统（族）。你通过代码产生器产生实现代码。

☆ ☆ ☆

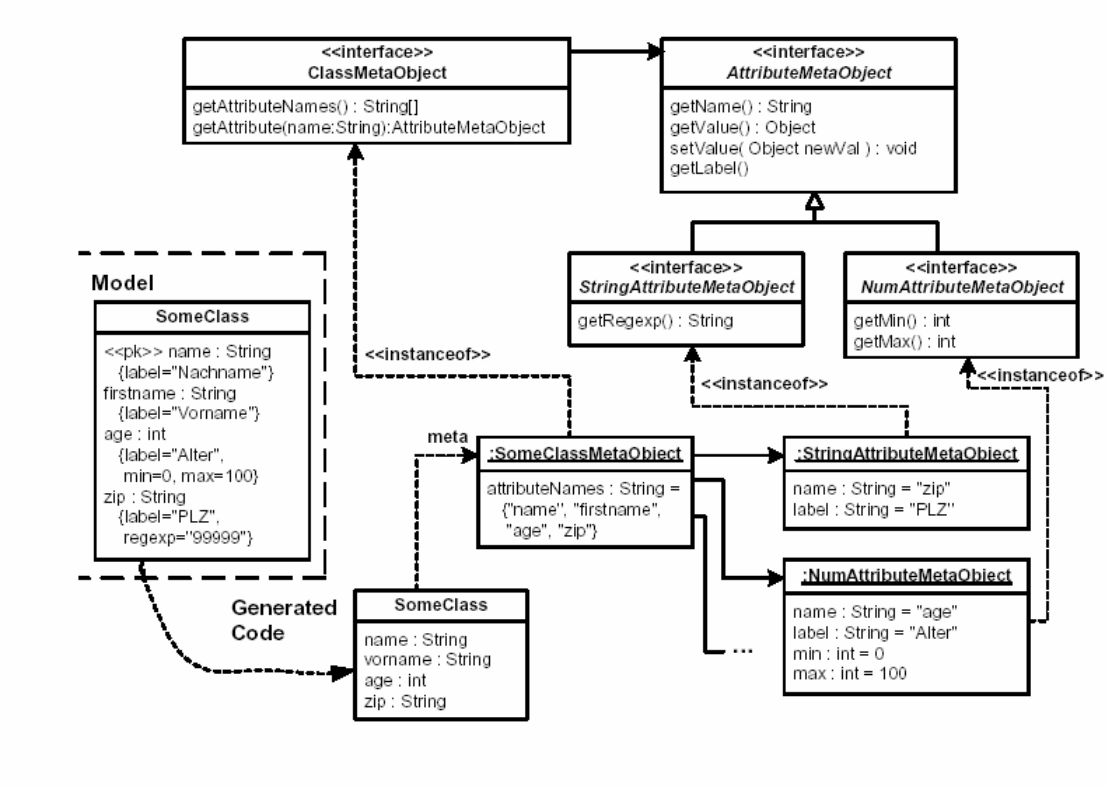
在模型驱动开发中采用富领域特定平台，应用通常需要一些信息，这些信息是在运行期用来控制应用平台各个方面的模型元素。如何使得模型信息在运行期可用，并且能够和自动生成的工件组合在一起？如何搭建自动生成代码和架构部分的桥梁？

假设要搭建的系统必须能够提供领域特定的日志机制。应用务必能够将自动生成类的属性值输出到日志文件中。为了使之成为可能，日志中需要明确每个类的所有属性的名字和值。尤其在不支持反射机制的语言中，实现该机制并不容易。

另一个问题是，你为对象属性提供附加信息的注释，例如明确的标记、形式描述、或数值属性的值域。在运行时刻——例如动态构建用户界面时，你需要能够访问这些信息。在编程语言自身的类中插入这些信息并不方便。

因此：

使用在产生代码时可用的描述待产生工件的元对象信息。提供一种方法来使待产生工件和它的元对象关联在一起。确保元对象有统一的可被“功能丰富的领域相关平台”接口。



☆ ☆ ☆

本模式使模型的选中部分在应用中可用，并且自然高效。另外一种方法（从理论上讲），将模型的部分和应用保存在一起——然而，访问复杂模型通常会很慢，从而这种方法不可行。

如何将元对象和产生的工件联系在一起，有不同方法。如果工件完全是自动产生的，则你可以直接为它增加 getMetaobject()操作。如果这不可行（例如你希望保持工件独立），你还可以使用集中式的注册表，让它提供查询功能 MetaRegistry.getMetaobjectFor(anArtefact)。 还将会产生诸如 mapping 等实现。

元对象不能仅用于“描述”编程元素，还应“处理”它们。这导致了“产生式元对象协议（GENERATED META OBJECT PROTOCOL）”模式。

☆ ☆ ☆

这种方法最有名的例子是 JavaBeans，其中 BeanInfo 类按上述方式描述 Bean 本身，主要用于 GUI 工具。惟一的不同在于，JavaBeans 一般不是产生的，而是手工编写的。

早期一个名为 LPF 的 OR-Mapping 框架有产生式元对象，它描述产生的关系表结构，用于管理运行时持久化。

早在 1987 年，LANSA 4GL 环境基于“active repository”提供了广泛的 LANSA 的元信息访问。LANSA RUOM 模型驱动产生器广泛使用了 LANSA repository。

另一个例子是 Small Components 项目，它基于 C++，其描述型元对象用于“模拟”反射。注意，直接访问模型是不可以的，因为嵌入式系统的基础设施性能和代码规范是很关键的。

框架和 DSL 相结合(P)

框架提供一套服务。通常这些很难使用，因为编程语言不“知道”这个框架，编译器不能提供任何支持。支持程序使用框架的 DSL 能够解决这个问题。

外部模型标记（P）

为了支持源模型到目标模型（或者自动生成代码）的转换，有时需要提供“支持”信息来明确目标源模型。把这些特别用于目标模型的概念增加到源模型中，使源模型受到“污染”。MDA 打算增加“模型标记”，但当前只有很少的工具能较好的支持。相反，我们推荐遵守模型以外的信息（例如在 XML 文件中）；执行转换时，转换引擎将使用辅助的信息。

生长时运行时桥（P)

在许多情况下，详细的模型特性或者系统配置项不仅会影响生成的内容，还会影响在运行时间的行为。例如在 ALMA 数据模型的模型驱动开发过程中，我们有一些数据的编码是遵循 VOTable XML 标准的。依赖于在“外部模型标记（EXTERNAL MODEL MARKING）”模式中的编码定义，对于生成的 XML 我们指定了编码格式。同样，我们必须依赖于在标记中相同的设施，对在运行期的实际数值进行编码。这个解决方案有不同的编码类，每个类针对每个编码方案，并且用代码生成器自动生成各自对象的实例，成为源代码。使用策略模式可以将自动生成代码和非自动生成代码（需要有一些接口才能使用，依赖于曾用过的特定编码策略）结合。

生成式反射层(P)

正如 [Kiczales et。 al， 1991] 中例子所显示的那样，元对象协议对语言的元对象进行内省、修改和具体化。典型地，这是动态实现的（例如在 CLOS [Koschmann，1990]等语言中）。对于模型驱动的软件开发而言，你至少能提供 read-only-MOP 从而支持类的内省以及操作的动态调用。通用接口支持客户程序访问类：

```
public interface RClass {
// initializer – associates with base-level object
public setObject( Object o );
// retrieve information about the object
public ROperation[] getOperations();
public RAttribute[] getAttributes();
// create new instance
public Object newInstance();
}
public interface ROperation {
// retrieve information about op
public RParameter[] getParams();
public String getReturnType();
// invoke
public Object invoke(Object params)
}
public interface RAttribute {
// retrieve information about op
public String getName();
public String getType();
// set / get
```

```
public Object get();

public void set( Object data );

}
```

既然操作是通用的，workbenche 或其他动态工具就可以使用之，只须反射接口能够处理这种数据即可。

GATEWAY 元类(P)

典型地，使用“技术子域（TECHNICAL SUBDOMAINS）”模式会导致不同的元模型和不同子域具体语法。例如，你可以使用活动图描述应用的工作流，使用基于文本的 XML 等语言描述展现层布局。如果你希望从这些不同规约产生有用的系统，产生器需要一种从一种模型到另一种模型的机制（例如从活动到与之关联的 UI 窗体）。为此，需要做到两点：第一，模型元素必须在两个“技术子域”的元模型都出现。如果生成器忽略了具体语法，GATEWAY 元类可以在不同的技术子域中使用不同的具体语法描述。第二，需要通用的元－元模型（meta meta model）。因为产生器总会使用某种元－元模型（或许是隐含的）。为了在同一个产生器内部模型表现中展现两种元模型，这两种元模型必须使用相同的元－元模型。例如 b+m 产生器要求 Java 类用作为所有元模型的元－元模型。

三层实现（P)

当产生面向对象的代码时，你常常不得不将平台代码、产生的代码、手工实现的代码混合在一起。一个不错的解决办法是：在平台中实现一个抽象基类，产生其抽象子类，实现应用逻辑时采用另一个非抽象子类。例如，这支持“强制前处理/后处理代码（FORCED PRE/POST CODE）”模式。

强制前处理/后处理代码(P)

在很多情况下，需要保证在手工实现的代码之前、之后执行某些产生的代码，必须使开发者不能绕开它们。以 vehicle 类为例，它有一个 drive(driver：Person) 操作，同时司机的年龄必须已过 18 岁。可以采用下列的代码模式（idiom）：

```
public interface IVehicle { // public interface for vehicles

public void drive(driver：  Person);

}
```

```
public abstract class VehicleBase implements IVehivle { // generated
public final void drive(driver： Person) { // final; cannot redefine
if ( !(driver。age() >= 18) ) throw new ConstraintViolated();
driveImpl();
}
protected abstract void driveImpl(driver： Person);
// protected; cannot be called by clients
}
public class VehicleImpl extends VehicleBase { // manually implemented
protected void driveImpl(driver： Person) {
// do whatever it takes…


}
}
```

相信再生(P)

最后，实现的应用采用包含了能够重新生成所有产生/转换部分的 build 过程。一旦有手工步骤或产生代码之后还需改代码，那么迟早要放弃产生器，而在代码上继续工作。

通过代理进行多模型集成（P)

常有好几个子模型共同组成了应用模型，例如你为应用的不同方面使用了不同 DSL，或者应用系统很大被分成了几部分。为此：

- 为每个元模型元素创建代理元类（proxy metaclass）

- 一旦实例化，该代理会自动发现它所代表的对象，否则会抛出异常。
- 通常，代理是其所代表的类型的子类。操作会转发到被代表的元素的操作。
- 这些代理可以自动产生。
- 产生器不知道代理和其代表元素的不同。

使用模型（P）

应该使用模型捕捉的信息，以避免重复，并使手工任务最小化。从而你产生的远不止代码：还有用户指南、帮助本文、测试数据、build 脚步等内容。要通过考虑极限编程场景中的“可持续性”，在自动任务和手工任务的工作量之间把握好平衡。正如“拇指定律”：比较击键次数和为了达到目的用户手势要求，然后选择最省体力的方法。可以考虑购买、构建、开源等方式，重复利用其他人的辛勤工作，不要重复发明轮子。

构建 IDE (P)

模型驱动方法可能导致应用开发者的额外复杂性，因为需要产生很多工件，这些工件之间还相互依赖。在某种程度上，这有悖让领域专家使用基础设施的目的，所以必须使用 MDSD 基础设施来简化工作。项目相关 IDE 或领域相关 IDE 是个久经考验的办法，而 Eclipse 等框架为此提供了良好的基础。

使用编译器(P)

同时出来产生的代码和手工代码时，常常引起二者的一致性问题。为此，应该使用编译器发现不一致性。例如：

- 如果你产生具有抽象方法的抽象类，然后又在子类中重写（overwrite）了抽象方法，编译器就会报错，因为方法签名被改变了（是 overwrite 不是 override）。
- 如果你预计手工实现的类有特定操作，因为某种原因，你可能不能强制这些操作由超类接口提供。你可以产生助手类，在其方法中来调用这些操作。该调用不会在运行时使用，而仅仅被系统的其他部分编译以“产生”编译错误。

考虑购买、构建、开源等方式(P)

不要忽视那些潜在的、现货销售的产品，以及有活力的开源基础设施，它们可能被成千上万的机构在使用。软件开发机构的“不是自己发明的”综合症很有名。考虑购买、构建、开源等方式和“使用模型”模式一样，需要考虑经济因素，即不要比较短期成本，而是考虑长期成本。还应考虑维护负担，是否真的值得开发基础设施和不构成业务核心的应用？然而，通过使用不很精良的现成工具代替它们，执行“使用模式（LEVERAGE THE MODEL）”就不会危及辛苦得到的进入领域相关框架和产生器的领域知识。

请注意区分：

- 战略性软件资产——业务的核心，这些关于业务和流程的资产，会成为对人或机器有用的活跃的知识库。
- 非战略性软件资产——必要的基础设施，一般是技术性积累，两到三年就会陈旧。
- 软件负债——会带来成本负担的遗留资产。[译者注：资产和负债是相对应的财务术语]

争取将你的核心业务转化成战略性资产，随着时间的推移它们不断被重用、被精化，其价值会增加。广泛使用的有名气的开源软件也应该作为战略性资产对待——不过是公共资产。商业的第三方软件变成非战略性软件资产，它们通常不是基于开放标准构建的，也不敢假设它们被无限期地支持。从而在商业软件上的投资是会贬值的。

- 要诚实，不要把所有遗留系统都描绘成战略性资产，识别那些负债，并争取用战略性资产和非战略性资产代替它们。
- 应区分第三方产品和这些产品基于组织信息建立其的数据库。其中的一个子集是战略性软件资产，需要保护。成为负债的遗留系统也同样：应用系统成了负债，但其管理的信息是战略性资产。

如果不持续关照战略性软件资产，它们就会退化成负债，从而被无情地淘汰。

如果适当，使用开源基础设施可以降低依赖开发商风险。当评估开源软件时，应认真考量其授权模式：有的开源软件允许被构建到非开源的商业产品中，有的则不行。

注意：一个战略性和非战略性资产的不错的混合方案仅包括很少的战略性软件资产。非战略性资产永远起着支持业务的作用，在某些方面它们是必须的消费品。

感谢

感谢 Tom Stahl 的 b+m AG，它为一些模式赋予灵感，并提供了知名的应用。感谢 Joe Schwarz and Gianni Raffi，他允许我们采用 ALMA 数据模型的案例。我们也要感谢 Ghica van Emde Boas，他们提供了一些原型模式。

参考文献

[Alexander 1977] Christopher Alexander, 1977, A Pattern Language: Towns, Buildings, Construction, Oxford University Press

[Beck 2000] Kent Beck, 2000, Extreme Programming Explained: Embrace Change, Addison-Wesley

[Bettin 2003] Jorn Bettin, 2003, Best Practices for Component-Based Development and Model-Driven Architecture, http://www.softmetaware.com/best-practices-for-cbd-andmda.pdf.

[Bettin 2003b] Jorn Bettin, 2003, Ideas for a Concrete Visual Syntax for Model-to-Model Transformations, http://www.softmetaware.com/oopsla2003/bettin.pdf and http://www.softmetaware.com/oopsla2003/bettin.ppt.

[Bettin 2002] Jorn Bettin, 2002, Measuring the Potential of Domain-Specific Modeling Techniques, http://www.cis.uab.edu/info/OOPSLADSVL2/Papers/Bettin.pdf.

[Bettin 2001] Jorn Bettin, 2001, A Language to describe software texture in abstract design models and implementation, http://www.isis.vanderbilt.edu/OOPSLA2K1/Papers/Bettin.pdf.

[BH 2003] Jorn Bettin, Jeff Hoare, 2003, Time Conscious Objects: A Domain-Specific Framework and Generator, http://www.softmetaware.com/oopsla2003/pos06-bettin.pdf.

[Bosch 2000] Jan Bosch, 2000, Design & Use of Software Architectures, Adopting and Evolving a Produt-Line Approach, Addison-Wesley

[Cleaveland 2001] Craig Cleaveland, 2001, Program Generators with XML and Java, Prentice Hall

[CN 2002] Paul Clements, Linda Northrop, 2002, Software Product Lines, Practices and Patterns, Addison Wesley

[Cockburn 1998] Alistair Cockburn, 1998, Surviving Object-Oriented Projects, Addison-Wesley

[Codagen] Codagen Architect, http://www.codagen.com

[Eclipse GMT] Generative Model Transformer project, http://www.eclipse.org/gmt/

[Evans 2000] Eric Evans, 2003, Domain-Driven Design, Addison-Wesley

[GDP] b+m AG, Generatice Development Process, http://www.architectureware.de/download/b+m_Generative_Development_Process.pdf

[GenFW] Sourceforge.net, openArchitectureWare, http://architecturware.sourceforge.net

[Gentastic] Gentastic, eGen, http://www.gentastic.com

[GHJV95] Gamma, Helm, Johnson, Vlissdes, Design Patterns, Addison-Wesley 1995

[Kiczales et. al., 1991] Gregor Kiczales, The Art of the Metaobject Protocol, MIT Press, 1991

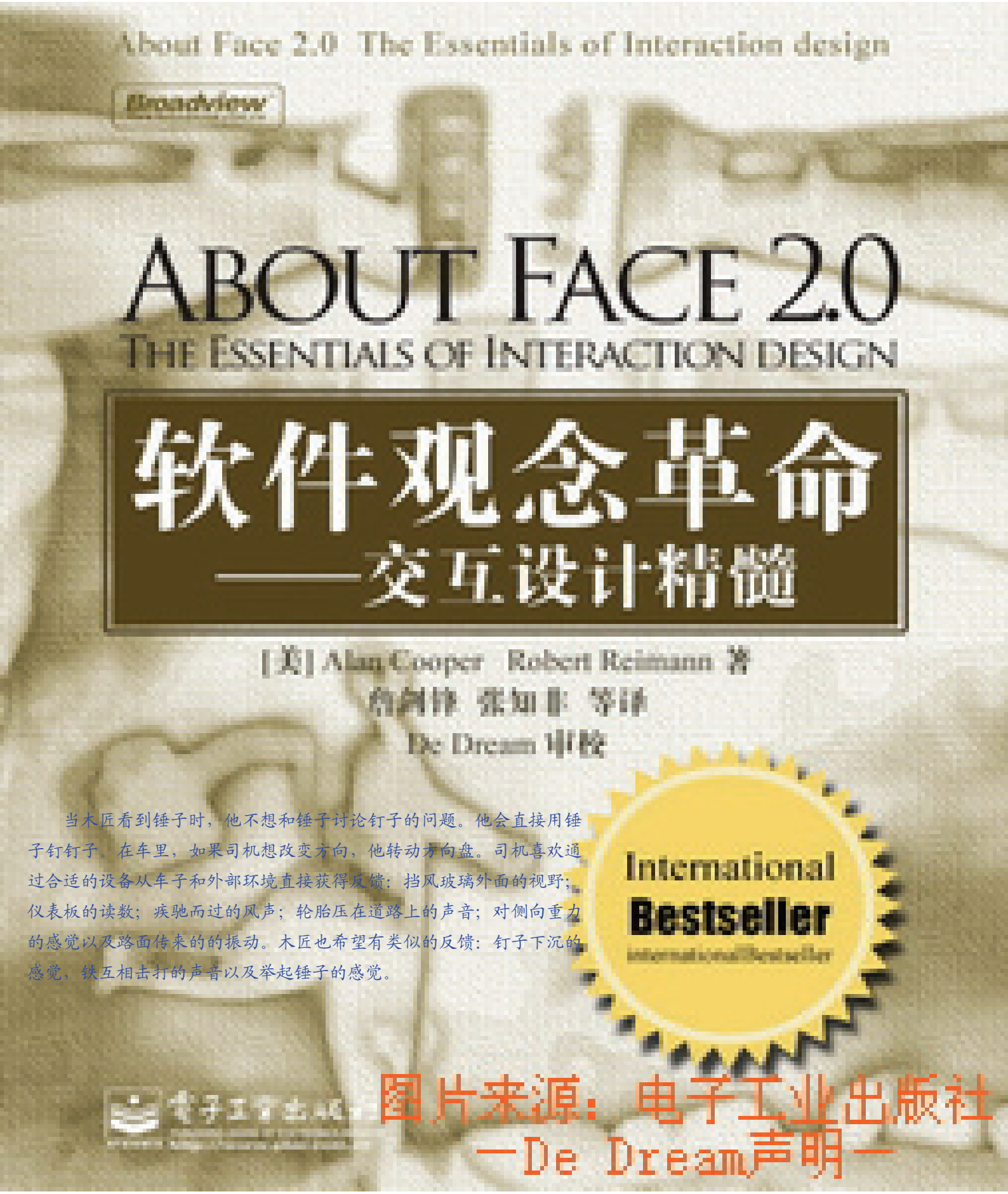
[Koschmann, 1990] Timothy D. Koschmann, The Common Lisp Companion, Wiley, 1990

[LANSA] LANSA Rapid User Object Method, http://www.lansa.com/downloads/support/docs/v10/

lansa060.zip

[OMG MDA] Model-Driven Architecture, http://www.omg.org/mda/

[WL 1999] D. M. Weiss, C.T.R. Lai: Software Product Line Engineering, A Family-Based Software Development Process, Addison-Wesley



UMLChina 辅助教材

国内首个交互设计网站 De Dream'

WILEY

TIMELY. PRACTICAL. RELIABLE.

UMLChina 指定教材

Agile Database Techniques

Effective Strategies for the Agile Software Developer

《敏捷数据》

UMLChina 李巍 译

机械工业出版社即将出版

Scott Ambler

软件工程技术丛书

设计系列

企业应用架构模式

Patterns of Enterprise Application Architecture

(美) Martin Fowler 著

王怀民 周斌 译

UMLChina 审校

UMLChina 训练辅助教材

测试驱动开发全攻略

Brian Sun 著

讨论

TDD 的目标

Clean Code That Works

这句话的含义是，事实上我们只做两件事情：让代码奏效（Work）和让代码洁净（Clean），前者是把事情做对，后者是把事情做好。想想看，其实我们平时所做的所有工作，除去无用的工作和错误的工作以外，真正正确的工作，并且是真正有意义的工作，其实也就只有两大类：增加功能和提升设计，而 TDD 正是在这个原则上产生的。如果您的工作并非我们想象的这样，（这意味着您还存在第三类正确有意义的工作，或者您所要做的根本和我们在说的是两回事），那么这告诉我们您并不需要 TDD，或者不适用 TDD。而如果我们偶然猜对（这对于我来说是偶然，而对于 Kent Beck 和 Martin Fowler 这样的大师来说则是辛勤工作的成果），那么恭喜您，TDD 有可能成为您显著提升工作效率的一件法宝。请不要将信将疑，若即若离，因为任何一项新的技术——只要是从根本上改变人的行为方式的技术——就必然使得相信它的人越来越相信，不信的人越来越不信。这就好比学游泳，唯一能学会游泳的途径就是亲自下去游，除此之外别无他法。这也好比成功学，即使把卡耐基或希尔博士的书倒背如流也不能拥有积极的心态，可当你以积极的心态去成就了一番事业之后，你就再也离不开它了。相信我，TDD 也是这样！想试用 TDD 的人们，请遵循下面的步骤：

编写 TestCase	--> 实现 TestCase -->	重构
（确定范围和目标）	（增加功能）	（提升设计）

[友情提示：敏捷建模中的一个相当重要的实践被称为：Prove it With Code，这种想法和 TDD 不谋而合。]

TDD 的优点

充满吸引力的优点

1. 完工时完工。表明我可以很清楚的看到自己的这段工作已经结束了，而传统的方式很难知道什么时候编码工作结束了。
2. 全面正确的认识代码和利用代码，而传统的方式没有这个机会。
3. 为利用你成果的人提供 Sample，无论它是要利用你的源代码，还是直接重用你提供的组件。
4. 开发小组间降低了交流成本，提高了相互信赖程度。
5. 避免了过渡设计。
6. 系统可以与详尽的测试集一起发布，从而对程序的将来版本的修改和扩展提供方便。
7. TDD 给了我们自信，让我们今天的问题今天解决，明天的问题明天解决，今天不能解决明天的问题，因为明天的问题还没有出现(没有 TestCase)，除非有 TestCase 否则我决不写任何代码；明天也不必担心今天的问题，只要我亮了绿灯。

不显而易见的优点

1. 逃避了设计角色。对于一个敏捷的开发小组，每个人都在做设计。
2. 大部分时间代码处在高质量状态，100%的时间里成果是可见的。
3. 由于可以保证编写测试和编写代码的是相同的程序员，降低了理解代码所花费的成本。
4. 为减少文档和代码之间存在的细微的差别和由这种差别所引入的 Bug 作出杰出贡献。
5. 在预先设计和紧急设计之间建立一种平衡点，为你区分哪些设计该事先做、哪些设计该迭代时做提供了一个可靠的判断依据。

有争议的优点

1. 事实上提高了开发效率。每一个正在使用 TDD 并相信 TDD 的人都会相信这一点，但观望者则不同，不相信 TDD 的人甚至坚决反对这一点，这很正常，世界总是这样。
2. 发现比传统测试方式更多的 Bug。
3. 使 IDE 的调试功能失去意义，或者应该说，避免了令人头痛的调试和节约了调试的时间。
4. 总是处在要么编程要么重构的状态下，不会使人抓狂。（两顶帽子）
5. 单元测试非常有趣。

TDD 的步骤

编写 TestCase -->	实现 TestCase -->	重构
（不可运行）	（可运行）	（重构）

步骤	制品
（1）快速新增一个测试用例	新的 TestCase
（2）编译所有代码，刚刚写的那个测试很可能编译不通过	原始的 TODO List
（3）做尽可能少的改动，让编译通过	Interface
（4）运行所有的测试，发现最新的测试不能编译通过	－(Red Bar)
（5）做尽可能少的改动，让测试通过	Implementation
（6）运行所有的测试，保证每个都能通过	－(Green Bar)
（7）重构代码，以消除重复设计	Clean Code That Works

FAQ

什么时候重构？

如果您在软件公司工作，就意味着您成天都会和想通过重构改善代码质量的想法打交道，不仅您如此，您的大部分同事也都如此。可是，究竟什么时候该重构，什么情况下应该重构呢？我相信您和您的同事可能有很多不同的看法，最常见的答案是"该重构时重构"，"写不下去的时候重构"，和"下一次迭代开始之前重构"，或者干脆就是"最近没时间，就不重构了，下次有时间的时候重构吧"。正如您已经预见到我想说的——这些想法都是对重构的误解。重构不是一种构建软件的工具，不是一种设计软件的模式，也不是一个软件开发过程中的环节，正确理解重构的人应该把重构看成一种书写代码的方式，或习惯，重构时时刻刻有可能发生。在 TDD 中，除去编写测试用例和实现测试用例之外的所有工作都是重构，所以，没有重构任何设计都不能实现。至于什么时候重构嘛，还要分开看，有三句话是我的经验：实现测试用例时重构代码，完成某个特性时重构设计，产品的重构完成后还要记得重构一下测试用例哦。

什么时候设计？

这个问题比前面一个要难回答的多，实话实说，本人在依照 TDD 开发软件的时候也常常被这个问题困扰，总是觉得有些问题应该在写测试用例之前定下来，而有些问题应该在新增一个一个测试用例的过程中自然出现，水到渠成。所以，我的建议是，设计的时机应该由开发者自己把握，不要受到 TDD 方式的限制，但是，不需要事先确定的事一定不能事先确定，免得捆住了自己的手脚。

什么时候增加新的 TestCase？

没事做的时候。通常我们认为，如果你要增加一个新的功能，那么先写一个不能通过的 TestCase；如果你发现了一个 bug，那么先写一个不能通过的 TestCase；如果你现在什么都没有，从 0 开始，请先写一个不能通过的 TestCase。所有的工作都是从一个 TestCase 开始。此外，还要注意的是，一些大师要求我们每次只允许有一个 TestCase 亮红灯，在这个 TestCase 没有 Green 之前不可以写别的 TestCase，这种要求可以适当考虑，但即使有多个 TestCase 亮红灯也不要紧，并未违反 TDD 的主要精神。

TestCase 该怎么写？

测试用例的编写实际上就是两个过程：使用尚不存在的代码和定义这些代码的执行结果。所以一个 TestCase 也就应该包括两个部分——场景和断言。第一次写 TestCase 的人会有很大的不适应的感觉，因为你之前所写的所

有东西都是在解决问题，现在要你提出问题确实不大习惯，不过不用担心，你正在做正确的事情，而这个世界上最难的事情也不在于如何解决问题，而在于 ask the right question！

TDD 能帮助我消除 Bug 吗？

不能！千万不要把"测试"和"除虫"混为一谈！"除虫"是指程序员通过自己的努力来减少 bug 的数量（消除 bug 这样的字眼我们还是不要讲为好^_^），而"测试"是指程序员书写产品以外的一段代码来确保产品能有效工作。虽然 TDD 所编写的测试用例在一定程度上为寻找 bug 提供了依据，但事实上，按照 TDD 的方式进行的软件开发是不可能通过 TDD 再找到 bug 的（想想我们前面说的"完工时完工"），你想啊，当我们的代码完成的时候，所有的测试用例都亮了绿灯，这时隐藏在代码中的 bug 一个都不会露出马脚来。

但是，如果要问"测试"和"除虫"之间有什么联系，我相信还是有很多话可以讲的，比如 TDD 事实上减少了 bug 的数量，把查找 bug 战役的关注点从全线战场提升到代码战场以上。还有，bug 的最可怕之处不在于隐藏之深，而在于满天遍野。如果你发现了一个用户很不容易才能发现的 bug，那么不一定对工作做出了什么杰出贡献，但是如果你发现一段代码中，bug 的密度或离散程度过高，那么恭喜你，你应该抛弃并重写这段代码了。TDD 避免了这种情况，所以将寻找 bug 的工作降低到了一个新的低度。

我该为一个 Feature 编写 TestCase 还是为一个类编写 TestCase？

初学者常问的问题。虽然我们从 TDD 的说明书上看到应该为一个特性编写相应的 TestCase，但为什么著名的 TDD 大师所写的 TestCase 都是和类/方法一一对应的呢？为了解释这个问题，我和我的同事们都做了很多试验，最后我们得到了一个结论，虽然我不知道是否正确，但是如果您没有答案，可以姑且相信我们。

我们的研究结果表明，通常在一个特性的开发开始时，我们针对特性编写测试用例，如果您发现这个特性无法用 TestCase 表达，那么请将这个特性细分，直至您可以为手上的特性写出 TestCase 为止。从这里开始是最安全的，它不会导致任何设计上重大的失误。但是，随着您不断的重构代码，不断的重构 TestCase，不断的依据 TDD 的思想做下去，最后当产品伴随测试用例集一起发布的时候，您就会不经意的发现经过重构以后的测试用例很可能是和产品中的类/方法一一对应的。

什么时候应该将全部测试都运行一遍？

Good Question！大师们要求我们每次重构之后都要完整的运行一遍测试用例。这个要求可以理解，因为重构很可能会改变整个代码的结构或设计，从而导致不可预见的后果，但是如果我正在开发的是一个 ERP 怎么办？运行一遍完整的测试用例可能将花费数个小时，况且现在很多重构都是由工具做到的，这个要求的可行性和前提条

件都有所动摇。所以我认为原则上你可以挑几个你觉得可能受到本次重构影响的 TestCase 去 run，但是如果运行整个测试包只要花费数秒的时间，那么不介意你按大师的要求去做。

什么时候改进一个 TestCase？

增加的测试用例或重构以后的代码导致了原来的 TestCase 的失去了效果，变得无意义，甚至可能导致错误的结果，这时是改进 TestCase 的最好时机。但是有时你会发现，这样做仅仅导致了原来的 TestCase 在设计上是臃肿的，或者是冗余的，这都不要紧，只要它没有失效，你仍然不用去改进它。记住，TestCase 不是你的产品，它不要好看，也不要怎么太科学，甚至没有性能要求，它只要能完成它的使命就可以了——这也证明了我们后面所说的"用 Ctrl-C/Ctrl-V 编写测试用例"的可行性。

但是，美国人的想法其实跟我们还是不太一样，拿托尼巴赞的 MindMap 来说吧，其实画 MindMap 只是为了表现自己的思路，或记忆某些重要的事情，但托尼却建议大家把 MindMap 画成一件艺术品，甚至还有很多艺术家把自己画的抽象派 MindMap 拿出来帮助托尼做宣传。同样，大师们也要求我们把 TestCase 写的跟代码一样质量精良，可我想说的是，现在国内有几个公司能把产品的代码写的精良？？还是一步一步慢慢来吧。

为什么原来通过的测试用例现在不能通过了？

这是一个警报，Red Alert！它可能表达了两层意思——都不是什么好意思——1）你刚刚进行的重构可能失败了，或存在一些错误未被发现，至少重构的结果和原来的代码不等价了。2）你刚刚增加的 TestCase 所表达的意思跟前面已经有的 TestCase 相冲突，也就是说，新增的功能违背了已有的设计，这种情况大部分可能是之前的设计错了。但无论哪错了，无论是那层意思，想找到这个问题的根源都比 TDD 的正常工作要难。

我怎么知道那里该有一个方法还是该有一个类？

这个问题也是常常出现在我的脑海中，无论你是第一次接触 TDD 或者已经成为 TDD 专家，这个问题都会缠绕着你不放。不过问题的答案可以参考前面的"什么时候设计"一节，答案不是唯一的。其实多数时候你不必考虑未来，今天只做今天的事，只要有重构工具，从方法到类和从类到方法都很容易。

我要写一个 TestCase，可是不知道从哪里开始？

从最重要的事开始，what matters most？从脚下开始，从手头上的工作开始，从眼前的事开始。从一个没有 UI 的核心特性开始，从算法开始，或者从最有可能耽误时间的模块开始，从一个最严重的 bug 开始。这是 TDD 主义者和鼠目寸光者的一个共同点，不同点是前者早已成竹在胸。

为什么我的测试总是看起来有点愚蠢？

哦？是吗？来，握个手，我的也是！不必担心这一点，事实上，大师们给的例子也相当愚蠢，比如一个极端的例子是要写一个两个 int 变量相加的方法，大师先断言 2+3=5，再断言 5+5=10，难道这些代码不是很愚蠢吗？其实这只是一个极端的例子，当你初次接触 TDD 时，写这样的代码没什么不好，以后当你熟练时就会发现这样写没必要了，要记住，谦虚是通往 TDD 的必经之路！从经典开发方法转向 TDD 就像从面向过程转向面向对象一样困难，你可能什么都懂，但你写出来的类没有一个纯 OO 的！我的同事还告诉我真正的太极拳，其速度是很快的，不比任何一个快拳要慢，但是初学者（通常是指学习太极拳的前 10 年）太不容易把每个姿势都做对，所以只能慢慢来。

什么场合不适用 TDD？

问的好，确实有很多场合不适合使用 TDD。比如对软件质量要求极高的军事或科研产品——神州六号，人命关天的软件——医疗设备，等等，再比如设计很重要必须提前做好的软件，这些都不适合 TDD，但是不适合 TDD 不代表不能写 TestCase，只是作用不同，地位不同罢了。

Best Practise

微笑面对编译错误

学生时代最害怕的就是编译错误，编译错误可能会被老师视为上课不认真听课的证据，或者同学间相互嘲笑的砝码。甚至离开学校很多年的老程序员依然害怕它就像害怕迟到一样，潜意识里似乎编译错误极有可能和工资挂钩（或者和智商挂钩，反正都不是什么好事）。其实，只要提交到版本管理的代码没有编译错误就可以了，不要担心自己手上的代码的编译错误，通常，编译错误都集中在下面三个方面：

（1）你的代码存在低级错误

（2）由于某些 Interface 的实现尚不存在，所以被测试代码无法编译

（3）由于某些代码尚不存在，所以测试代码无法编译

请注意第二点与第三点完全不同，前者表明设计已存在，而实现不存在导致的编译错误；后者则指仅有 TestCase 而其它什么都没有的情况，设计和实现都不存在，没有 Interface 也没有 Implementation。

另外，编译器还有一个优点，那就是以最敏捷的身手告诉你，你的代码中有那些错误。当然如果你拥有 Eclipse 这样可以及时提示编译错误的 IDE，就不需要这样的功能了。

重视你的计划清单

在非 TDD 的情况下，尤其是传统的瀑布模型的情况下，程序员不会不知道该做什么，事实上，总是有设计或者别的什么制品在引导程序员开发。但是在 TDD 的情况下，这种优势没有了，所以一个计划清单对你来说十分重要，因为你必须自己发现该做什么。不同性格的人对于这一点会有不同的反应，我相信平时做事没什么计划要依靠别人安排的人（所谓将才）可能略有不适应，不过不要紧，Tasks 和 Calendar（又称效率手册）早已成为现代上班族的必备工具了；而平时工作生活就很有计划性的人，比如我:)，就会更喜欢这种自己可以掌控 Plan 的方式了。

废黜每日代码质量检查

如果我没有记错的话，PSP 对于个人代码检查的要求是蛮严格的，而同样是在针对个人的问题上，TDD 却建议你废黜每日代码质量检查，别起疑心，因为你总是在做 TestCase 要求你做的事情，并且总是有办法（自动的）检查代码有没有做到这些事情——红灯停绿灯行，所以每日代码检查的时间可能被节省，对于一个严格的 PSP 实践者来说，这个成本还是很可观的！

此外，对于每日代码质量检查的另一个好处，就是帮助你认识自己的代码，全面的从宏观、微观、各个角度审视自己的成果，现在，当你依照 TDD 做事时，这个优点也不需要了，还记得前面说的 TDD 的第二个优点吗，因为你已经全面的使用了一遍你的代码，这完全可以达到目的。

但是，问题往往也并不那么简单，现在有没有人能告诉我，我如何全面审视我所写的测试用例呢？别忘了，它们也是以代码的形式存在的哦。呵呵，但愿这个问题没有把你吓到，因为我相信到目前为止，它还不是瓶颈问题，况且在编写产品代码的时候你还是会自主的发现很多测试代码上的没考虑到的地方，可以就此修改一下。道理就是如此，世界上没有任何方法能代替你思考的过程，所以也没有任何方法能阻止你犯错误，TDD 仅能让你更容易发现这些错误而已。

如果无法完成一个大的测试，就从最小的开始

如果我无法开始怎么办，教科书上有个很好的例子：我要写一个电影列表的类，我不知道如何下手，如何写测试用例，不要紧，首先想象静态的结果，如果我的电影列表刚刚建立呢，那么它应该是空的，OK，就写这个断言吧，断言一个刚刚初始化的电影列表是空的。这不是愚蠢，这是细节，奥运会五项全能的金牌得主玛丽莲·金是

这样说的："成功人士的共同点在于.....如果目标不够清晰，他们会首先做通往成功道路上的每一个细小步骤....."。

尝试编写自己的 xUnit

Kent Beck 建议大家每当接触一个新的语言或开发平台的时候，就自己写这个语言或平台的 xUnit，其实几乎所有常用的语言和平台都已经有了自己的 xUnit，而且都是大同小异，但是为什么大师给出了这样的建议呢。其实 Kent Beck 的意思是说通过这样的方式你可以很快的了解这个语言或平台的特性，而且 xUnit 确实很简单，只要知道原理很快就能写出来。这对于那些喜欢自己写底层代码的人，或者喜欢控制力的人而言是个好消息。

善于使用 Ctrl-C/Ctrl-V 来编写 TestCase

不必担心 TestCase 会有代码冗余的问题，让它冗余好了。

永远都是功能 First，改进可以稍后进行

上面这个标题还可以改成另外一句话：避免过度设计！

淘汰陈旧的用例

舍不得孩子套不着狼。不要可惜陈旧的用例，因为它们可能从概念上已经是错误的了，或仅仅会得出错误的结果，或者在某次重构之后失去了意义。当然也不一定非要删除它们，从 TestSuite 中除去（JUnit）或加上 Ignored（NUnit）标签也是一个好办法。

用 TestCase 做试验

如果你在开始某个特性或产品的开发之前对某个领域不太熟悉或一无所知，或者对自己在该领域里的能力一无所知，那么你一定会选择做试验，在有单元测试作工具的情况下，建议你用 TestCase 做试验，这看起来就像你在写一个验证功能是否实现的 TestCase 一样，而事实上也一样，只不过你所验证的不是代码本身，而是这些代码所依赖的环境。

TestCase 之间应该尽量独立

保证单独运行一个 TestCase 是有意义的。

不仅测试必须要通过的代码，还要测试必须不能通过的代码

这是一个小技巧，也是不同于设计思路的东西。像越界的值或者乱码，或者类型不符的变量，这些输入都可能会导致某个异常的抛出，或者导致一个标示"illegal parameters"的返回值，这两种情况你都应该测试。当然我们无法枚举所有错误的输入或外部环境，这就像我们无法枚举所有正确的输入和外部环境一样，只要 TestCase 能说明问题就可以了。

编写代码的第一步，是在 TestCase 中用 Ctrl-C

这是一个高级技巧，呃，是的，我是这个意思，我不是说这个技巧难以掌握，而是说这个技巧当且仅当你已经是一个 TDD 高手时，你才能体会到它的魅力。多次使用 TDD 的人都有这样的体会，既然我的 TestCase 已经写的很好了，很能说明问题，为什么我的代码不能从 TestCase 拷贝一些东西来呢。当然，这要求你的 TestCase 已经具有很好的表达能力，比如断言 f(5)=125 的方式显然没有断言 f(5)=5^(5-2)表达更多的内容。

测试用例包应该尽量设计成可以自动运行的

如果产品是需要交付源代码的，那我们应该允许用户对代码进行修改或扩充后在自己的环境下 run 整个测试用例包。既然通常情况下的产品是可以自动运行的，那为什么同样作为交付用户的制品，测试用例包就不是自动运行的呢？即使产品不需要交付源代码，测试用例包也应该设计成可以自动运行的，这为测试部门或下一版本的开发人员提供了极大的便利。

只亮一盏红灯

大师的建议，前面已经提到了，仅仅是建议。

用 TestCase 描述你发现的 bug

如果你在另一个部门的同事使用了你的代码，并且，他发现了一个 bug，你猜他会怎么做？他会立即走到你的工位边上，大声斥责说："你有 bug！"吗？如果他胆敢这样对你，对不起，你一定要冷静下来，不要当面回骂他，相反你可以微微一笑，然后心平气和的对他说："哦，是吗？那么好吧，给我一个 TestCase 证明一下。"现在局势已经倒向你这一边了，如果他还没有准备好回答你这致命的一击，我猜他会感到非常羞愧，并在内心责怪自己太莽撞。事实上，如果他的 TestCase 没有过多的要求你的代码（而是按你们事前的契约），并且亮了红灯，那么就可以确定是你的 bug，反之，对方则无理了。用 TestCase 描述 bug 的另一个好处是，不会因为以后的修改而再次暴露这个 bug，它已经成为你发布每一个版本之前所必须检查的内容了。

关于单元测试

单元测试的目标是

Keep the bar green to keep the code clean

这句话的含义是，事实上我们只做两件事情：让代码奏效（Keep the bar green）和让代码洁净（Keep the code clean），前者是把事情做对，后者是把事情做好，两者既是 TDD 中的两顶帽子，又是 xUnit 架构中的因果关系。

单元测试作为软件测试的一个类别，并非是 xUnit 架构创造的，而是很早就有了。但是 xUnit 架构使得单元测试变得直接、简单、高效和规范，这也是单元测试最近几年飞速发展成为衡量一个开发工具和环境的主要指标之一的原因。正如 Martin Fowler 所说："软件工程有史以来从没有如此众多的人大大收益于如此简单的代码！"而且多数语言和平台的 xUnit 架构都是大同小异，有的仅是语言不同，其中最有代表性的是 JUnit 和 NUnit，后者是前者的创新和扩展。一个单元测试框架 xUnit 应该：1）使每个 TestCase 独立运行；2）使每个 TestCase 可以独立检测和报告错误；3）易于在每次运行之前选择 TestCase。下面是我枚举出的 xUnit 框架的概念，这些概念构成了当前业界单元测试理论和工具的核心：

测试方法/TestMethod

测试的最小单位，直接表示为代码。

测试用例/TestCase

由多个测试方法组成，是一个完整的对象，是很多 TestRunner 执行的最小单位。

测试容器/TestSuite

由多个测试用例构成，意在把相同含义的测试用例手动安排在一起，TestSuite 可以呈树状结构因而便于管理。在实现时，TestSuite 形式上往往也是一个 TestCase 或 TestFixture。

断言/Assertion

断言一般有三类，分别是比较断言（如 assertEquals），条件断言（如 isTrue），和断言工具（如 fail）。

测试设备/TestFixture

为每个测试用例安排一个 SetUp 方法和一个 TearDown 方法，前者用于在执行该测试用例或该用例中的每个测试方法前调用以初始化某些内容，后者在执行该测试用例或该用例中的每个方法之后调用，通常用来消除测试对系统所做的修改。

期望异常/Expected Exception

期望该测试方法抛出某种指定的异常，作为一个"断言"内容，同时也防止因为合情合理的异常而意外的终止了测试过程。

种类/Category

为测试用例分类，实际使用时一般有 TestSuite 就不再使用 Category，有 Category 就不再使用 TestSuite。

忽略/Ignored

设定该测试用例或测试方法被忽略，也就是不执行的意思。有些被抛弃的 TestCase 不愿删除，可以定为 Ignored。

测试执行器/TestRunner

执行测试的工具，表示以何种方式执行测试，别误会，这可不是在代码中规定的，完全是与测试内容无关的行为。比如文本方式，AWT 方式，swing 方式，或者 Eclipse 的一个视图等等。

实例：Fibonacci 数列

下面的 Sample 展示 TDDer 是如何编写一个旨在产生 Fibonacci 数列的方法。

（1）首先写一个 TC，断言 fib(1) = 1;fib(2) = 1;这表示该数列的第一个元素和第二个元素都是 1。

```
public void testFab() {
    assertEquals(1, fib(1));
    assertEquals(1, fib(2));
}
```

（2）上面这段代码不能编译通过，Great！——是的，我是说 Great！当然，如果你正在用的是 Eclipse 那你不需要编译，Eclipse 会告诉你不存在 fib 方法，单击 mark 会问你要不要新建一个 fib 方法，Oh，当然！为了让上面那个 TC 能通过，我们这样写：

```
public int fib( int n ) {
    return 1;
}
```

（3）现在那个 TC 亮了绿灯，wow！应该庆祝一下了。接下来要增加 TC 的难度了，测第三个元素。

```
public void testFab() {
    assertEquals(1, fib(1));
    assertEquals(1, fib(2));
    assertEquals(2, fib(3));
}
```

不过这样写还不太好看，不如这样写：

```
public void testFab() {
    assertEquals(1, fib(1));
    assertEquals(1, fib(2));
    assertEquals(fib(1)+fib(2), fib(3));
}
```

（4）新增加的断言导致了红灯，为了扭转这一局势我们这样修改 fib 方法，其中部分代码是从上面的代码中 Ctrl-C/Ctrl-V 来的：

```
public int fib( int n ) {
    if ( n == 3 ) return fib(1)+fib(2);
    return 1;
}
```

（5）天哪，这真是个贱人写的代码！是啊，不是吗？因为 TC 就是产品的蓝本，产品只要恰好满足 TC 就 ok。所以事情发展到这个地步不是 fib 方法的错，而是 TC 的错，于是 TC 还要进一步要求：

```
public void testFab() {
    assertEquals(1, fib(1));
    assertEquals(1, fib(2));
    assertEquals(fib(1)+fib(2), fib(3));
    assertEquals(fib(2)+fib(3), fib(4));
}
```

（6）上有政策下有对策。

```
public int fib( int n ) {
    if ( n == 3 ) return fib(1)+fib(2);
    if ( n == 4 ) return fib(2)+fib(3);
    return 1;
}
```

（7）好了，不玩了。现在已经不是贱不贱的问题了，现在的问题是代码出现了冗余，所以我们要做的是——重构：

```
public int fib( int n ) {
    if ( n == 1 || n == 2 ） return 1;
    else return fib( n - 1 ) + fib( n - 2 );
}
```

（8）好，现在你已经 fib 方法已经写完了吗？错了，一个危险的错误，你忘了错误的输入了。我们令 0 表示 Fibonacci 中没有这一项。

```
public void testFab() {
    assertEquals(1, fib(1));
    assertEquals(1, fib(2));
    assertEquals(fib(1)+fib(2), fib(3));
    assertEquals(fib(2)+fib(3), fib(4));
    assertEquals(0, fib(0));
    assertEquals(0, fib(-1));
}
```

then change the method fib to make the bar grean：

```
public int fib( int n ) {
    if ( n <= 0 ) return 0;
    if ( n == 1 || n == 2 ） return 1;
    else return fib( n - 1 ) + fib( n - 2 );
}
```

（9）下班前最后一件事情，把 TC 也重构一下：

```
public void testFab() {
    int cases[][] = {
        {0, 0}, {-1, 0},  //the wrong parameters
        {1, 1}, {2, 1}};  //the first 2 elements

    for (int i = 0; i < cases.length; i++)
        assertEquals( cases[i][1], fib(cases[i][0]) );

    //the rest elements
    for (int i = 3; i < 20; i++)
        assertEquals(fib(i-1)+fib(i-2), fib(i));  }
```

（10）打完收工。

关于本文的写作

在本文的写作过程中，作者也用到了 TDD 的思维，事实上作者先构思要写一篇什么样的文章，然后写出这篇文章应该满足的几个要求，包括功能的要求（要写些什么）和性能的要求（可读性如何）和质量的要求（文字的要求），这些要求起初是一个也达不到的（因为正文还一个字没有），在这种情况下作者的文章无法编译通过，为了达到这些要求，作者不停的写啊写啊，终于在花尽了两个月的心血之后完成了当初既定的所有要求（make the bar green），随后作者整理了一下文章的结构（重构），在满意的提交给了 Blog 系统之后，作者穿上了一件绿色的汗衫，趴在地上，学了两声青蛙叫。。。。。。^_^

后记：Martin Fowler 在中国

从本文正式完成到发表的几个小时里，我偶然读到了 Martin Fowler 先生北京访谈录，其间提到了很多对测试驱动开发的看法，摘抄在此：

Martin Fowler：当然（值得花一半的时间来写单元测试）！因为单元测试能够使你更快的完成工作。无数次的实践已经证明这一点。你的时间越是紧张，就越要写单元测试，它看上去慢，但实际上能够帮助你更快、更舒服地达到目的。

Martin Fowler：什么叫重要？什么叫不重要？这是需要逐渐认识的，不是想当然的。我为绝大多数的模块写单元测试，是有点烦人，但是当你意识到这工作的价值时，你会欣然的。

Martin Fowler：对全世界的程序员我都是那么几条建议：.....第二，学习测试驱动开发，这种新的方法会改变你对于软件开发的看法。.....

——《程序员》，2005 年 7 月刊

鸣谢

fhawk、Dennis Chen、般若菩提、Kent Beck、Martin Fowler、c2.com

（本文原载于 Brian Sun @ 爬树的泡泡[http://www.blogjava.net/briansun]）

Agile软件开发丛书

PEARSON Addison Wesley

有效用例模式

Patterns for Effective Use Cases

Foreword by Craig Larman

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

UMLChina 指定教材

清华大学出版社

软件与系统思想家温伯格精粹译丛

现代需求技术的基石

探索需求

设计前的质量

Donald C. Gause
Gerald M. Weinberg / 著
章柏幸 王媛媛 谢攀 / 译

Exploring Requirements: Quality Before Design

UMLChina 训练辅助教材

清华大学出版社

UML 工具发展趋势

潘加宇、王继伟 文

UML 已经诞生了八个年头，到现在成为一种表示法的事实标准，也带来建模工具市场的繁荣。没有标准表示法之前，建模工具市场处于一种小农经济的状态。方法学家（如 Peter Coad）不仅要开发方法，还要为自己的方法制定表示法，还要为这种表示法制作一套建模工具。因为相互之间所用的表示法不能相通，他只能事事亲历亲为。UML 标准的确立，在工具开发商的层面上意义重大。小农经济的方式被打破：制定 UML 标准的角色（OMG）、根据标准开发工具的角色（UML 工具厂商）、使用 UML 工具开发软件的角色（开发人员）被剥离了。如果你觉得 Rational 的建模工具不好，你可以自己开一家公司开发一个，所依赖的标准和 Rational 是一样的——因此以 UML 为基础的建模工具的数量和种类出现了爆炸性的增长。不完全的统计请看：http://www.umlchina.com/Tools/Newindex1.htm。

登堂入室和百花齐放

事情似乎突然以加速度发展。2002 年 10 月，Borland 并购了 BoldSoft，然后把它整合到 Delphi 中（现在以 ECO 的面目出现了）。原本生产代码工具的厂商开始要把 UML 纳入自己的领地。2002 年 10 月底，Borland 收购了 UML 工具厂商中的老二 Togethersoft，决心在自己的产品中全面整合 UML。随后，IBM 在 2002 年 12 月收购了 Rational，把 Rational 产品纳入自己的产品线。微软则以在 Whidbey 中提供一个代号为 Whitehorse 的设计工具和框架来应对。Sun 在 2003 年 1 月宣布把它的 IDE 和 Embarcadero Technologies 开发的 UML 建模工具 Describe for Sun ONE Studio 整合（2004 年 5 月更进一步收购了 Embarcadero 公司）。

最近的一桩收购是 Telelogic 收购 Popkin。通过一系列的并购，很多独立的 UML 工具公司消失了，但战场上又出现了更多的新公司（或组织）。

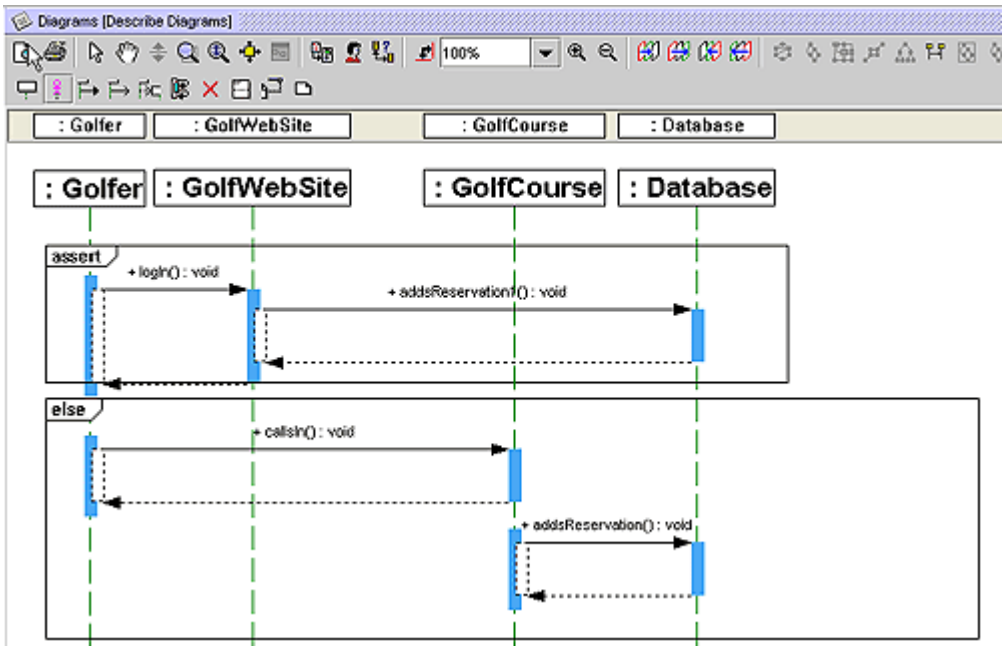


图 Sun ONE Studio 中的 UML2.0 序列图

现在的工具战场上，IBM Rational、Borland、Telelogic、Oracle、Sybase…等名头很响的厂家不用说了。Genttleware、MagicDraw、Ilogix 等也在各显神通。今年 Visual Paradigm 的 SDE 在 Jolt 奖评选中最终胜出，也获得了不少眼球。除了这些名花之外，还有许多工具颇有特点：

跨平台已经不成问题。不用说用Java开发的UML工具能通过Java的特点实现跨平台。如果想要原生的平台支持，也有丰富的选择。想要Linux平台上的UML工具吗？IBM现在对Linux很重视，IBM Rational Software Modeler 和 IBM Rational Software Architect同样也可以运行在Linux上。想要Mac平台上的？Excel Software专门为Mac平台制作的MacA&D（http://www.excelsoftware.com/macosxproducts.html）现在已经支持UML2.0。

嫌 Rational 那些太贵？Enterprise Architect（http://www.sparxsystems.com.au/）以性价比出众著称——功能全面，价格低廉。想要免费的？argoUML（http://argouml.tigris.org）免费而且开源。

有国货吗？有。楚凡科技的 UML/MDA 工具 Trufun Plato & Kant 就是。

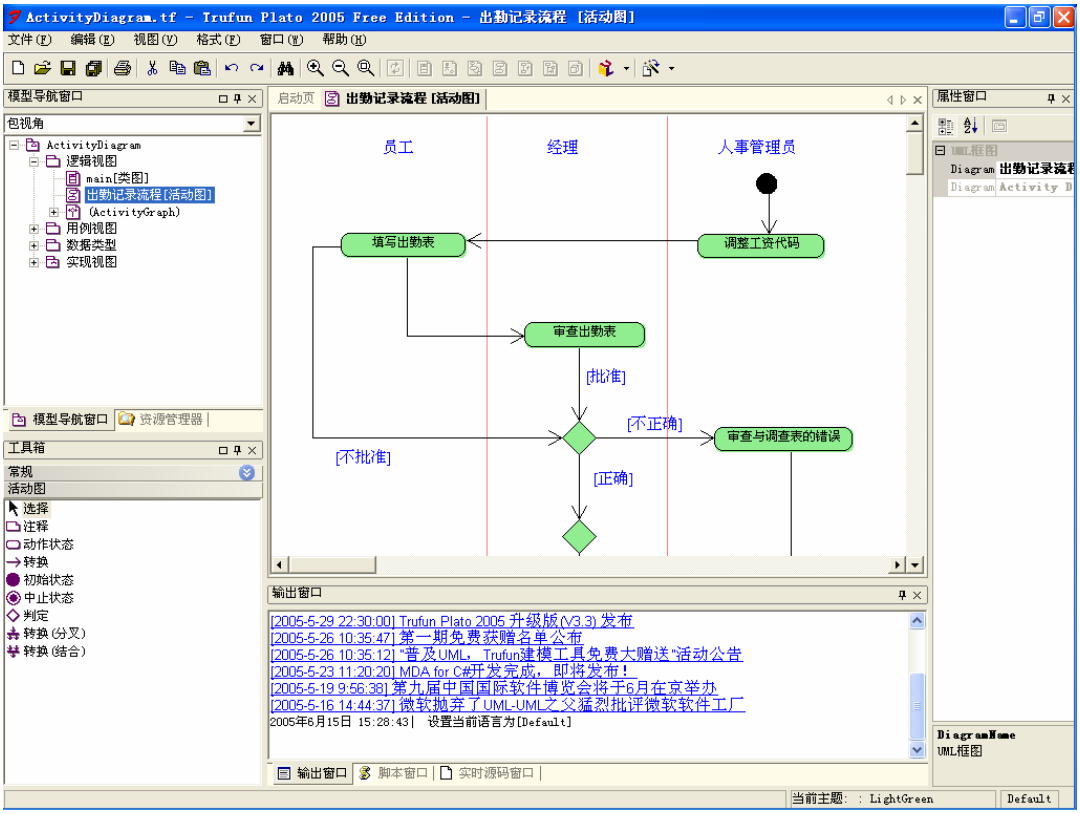


图 国产 UML/MDA 工具 Trufun Plato & Kant

你不喜欢那么正规，觉得白板是最方便的“UML 工具”？也行！使用 Ideogramic UML（http://www.ideogramic.com/products/uml/），你可以随意在白板上建模，这个工具会使用特有的识别算法把你的涂鸦转换成 UML 图，还可以通过 XMI，输入到 Rose 等各种提供 XMI 支持的工具。

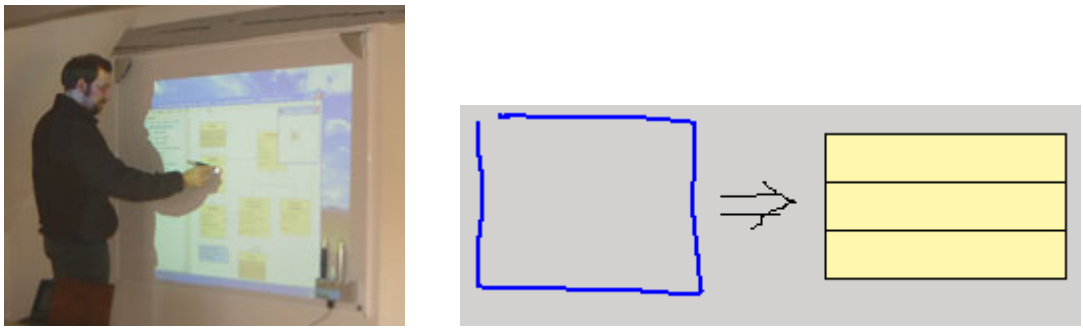


图 Ideogramic UML 允许你用手建模

集成，集成

集成到 IDE

上一代 CASE 工具通常以独立的产品存在。开发人员在 Rose 里建好模型，再通过一个转换器在代码环境生成代码。后来，逐渐演变成为代码环境提供插件（Rational XDE，Together 等），直接在代码环境中创建 UML 模型。

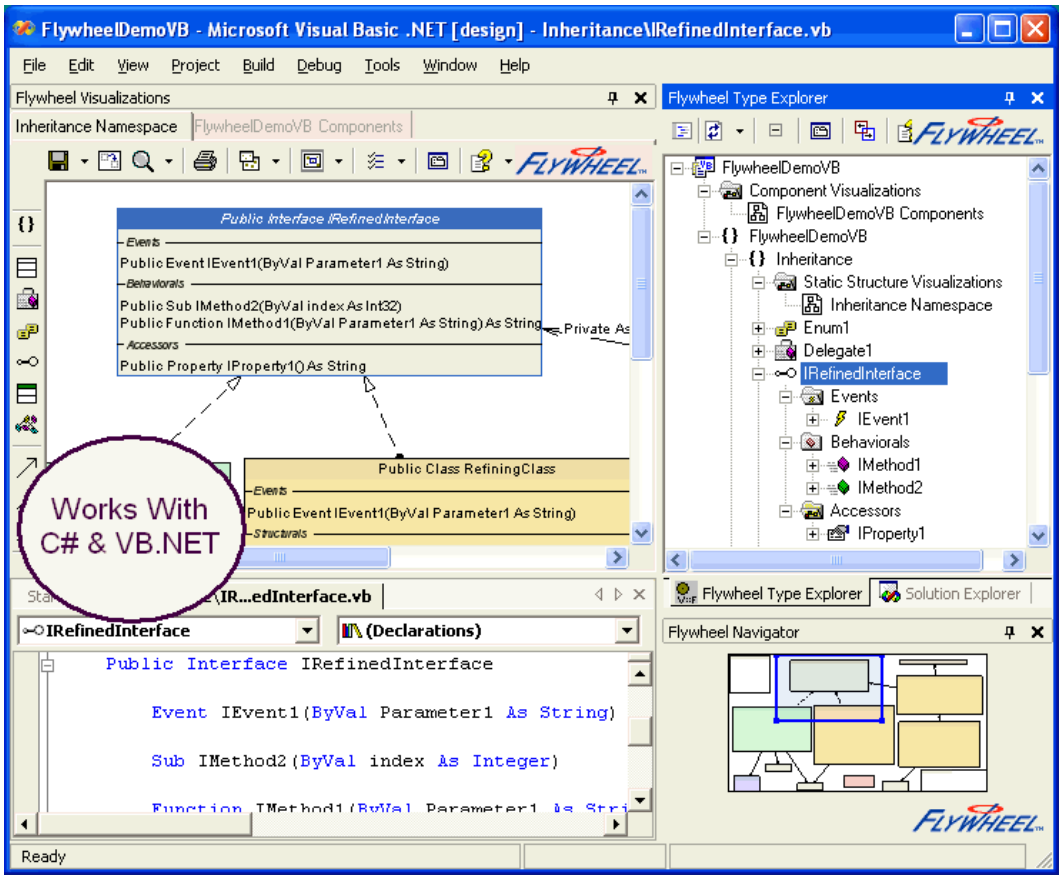


图 插入 VS.NET 中的 FlyWheel 工具

今年 Eclipse3.0 再次获得语言和开发环境类 Jolt 奖，显示出要成为未来所有 IDE 的公共平台的趋势。各 UML 工具厂商都已经或者正在和 Eclipse 集成，IBM Rational 首当其冲，新的产品线都是以 Eclipse3.0 为基础。而 Borland 的 Together for Eclipse 则备受 Eclipse 的 Fans 们称赞。Omondo 公司干脆就直接把它的工具叫做 EclipseUML。

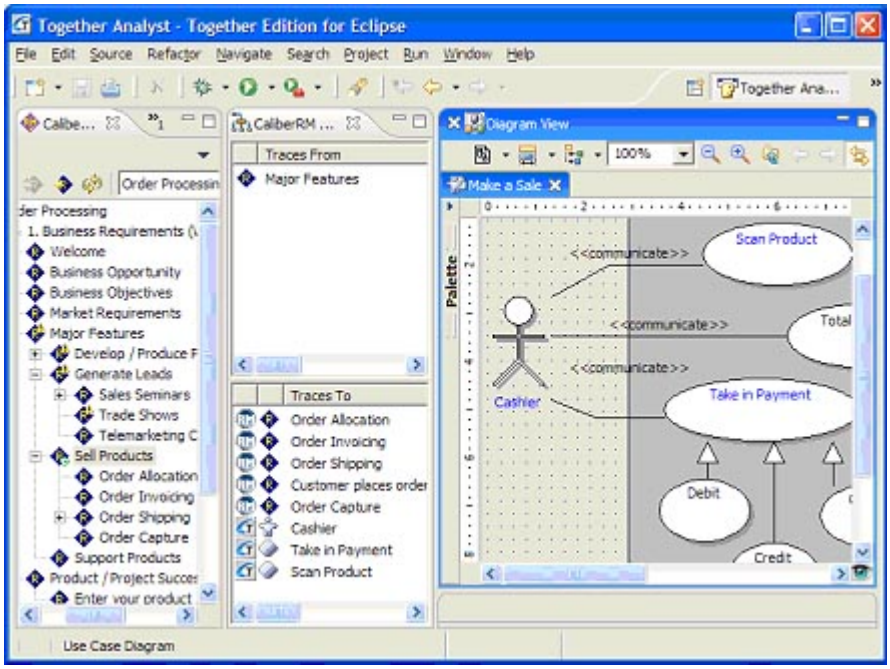


图 和 Eclipse 集成的 Together

和生命周期其他工具集成

不但要和代码环境结合，UML 工具也要和生命周期中的其他工具相结合。Rational 在这方面走得最早，凭借完整的产品线，把 UML 工具 Rose，需求管理工具 Requisite Pro，配置管理工具 Clearcase 等进行了高度集成。

Borland 在 2002 年购买了 TogetherSoft 、Starbase 和 BoldSoft 之后，也拥有了 UML 工具 Together，需求管理工具 CaliberRM 和配置管理工具 StarTeam。

Telelogic 也要将需求管理工具 DOORS、UML 工具 Tau 和配置管理工具 Synergy 整合起来。开发人员不需要从建模工具 Tau 中折腾到 DOORS 里面去，而是可以直接在 Tau 中去看放在 DOORS 数据库中的一个需求快照，同样也可以直接在 DOORS 中去看一个在 Synergy 中进行过的需求变更。

微软不知何时能正式发布的 VSTS，也提供了一系列支持整个开发团队的工具。

和过程集成

Rational 的工具一直和 RUP 集成，这也是 Rational 工具独特卖点之一。由于其他工具厂商往往缺乏过程上的指引，RUP 就被众多开发团队当作教材一样学习，虽然它是一个商业产品。

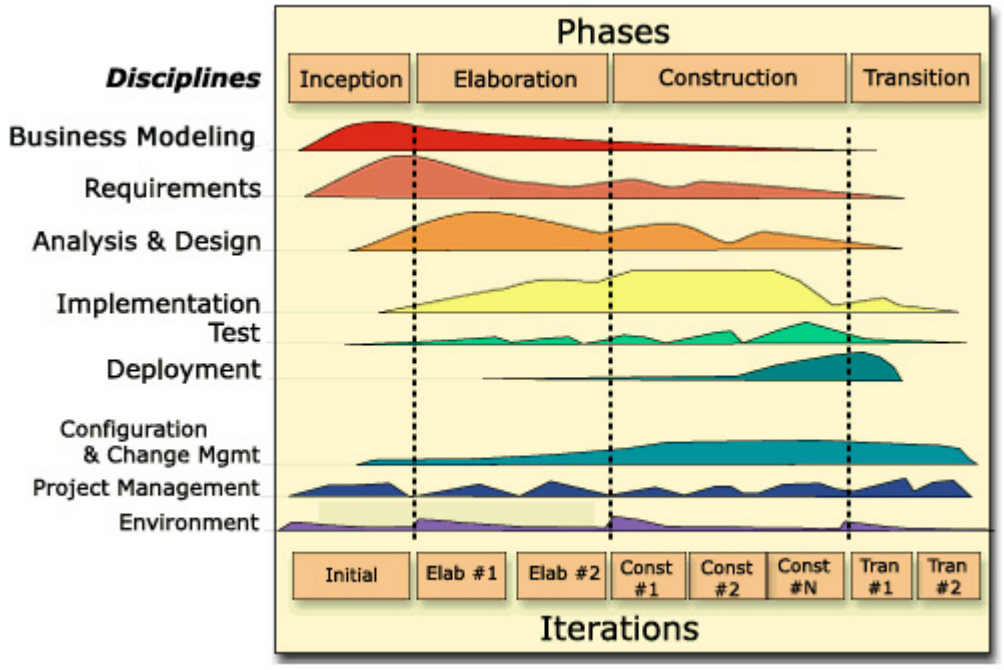


图 Rational Unified Process

2005 年 1 月，Borland 收购了 TeraQuest。通过收购，Borland 拥有了 CMM 的两位作者：Bill Curtis 和 Charlie Webber。Curtis 被任命为 Borland 的首席过程官，负责帮助 Borland 的软件和服务朝 CMM 模型靠拢。

微软在 VSTS 2005 中也提供了两套过程指南：MSF Formal 和 MSF Agile，前者相当于 CMMI3 的标准。

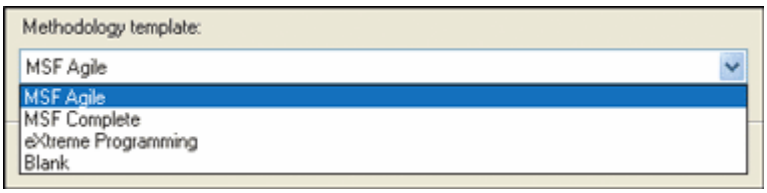


图 在 VSTS 中选择一种过程

和行业集成

OMG 力图通过和各行业标准组织结盟来推广 UML。2004 年 9 月，OMG 和 HL7 结盟，合作建立基于 UML 的医疗领域软件标准。2004 年 12 月 OMG 宣布和 DMTF 结盟，将使用 UML 和 XMI 对 DMTF 的公共信息模型 CIM 进行表示和交换，以简化 CIM 开发者在 UML 建模工具中创建 CIM 模型的工作。

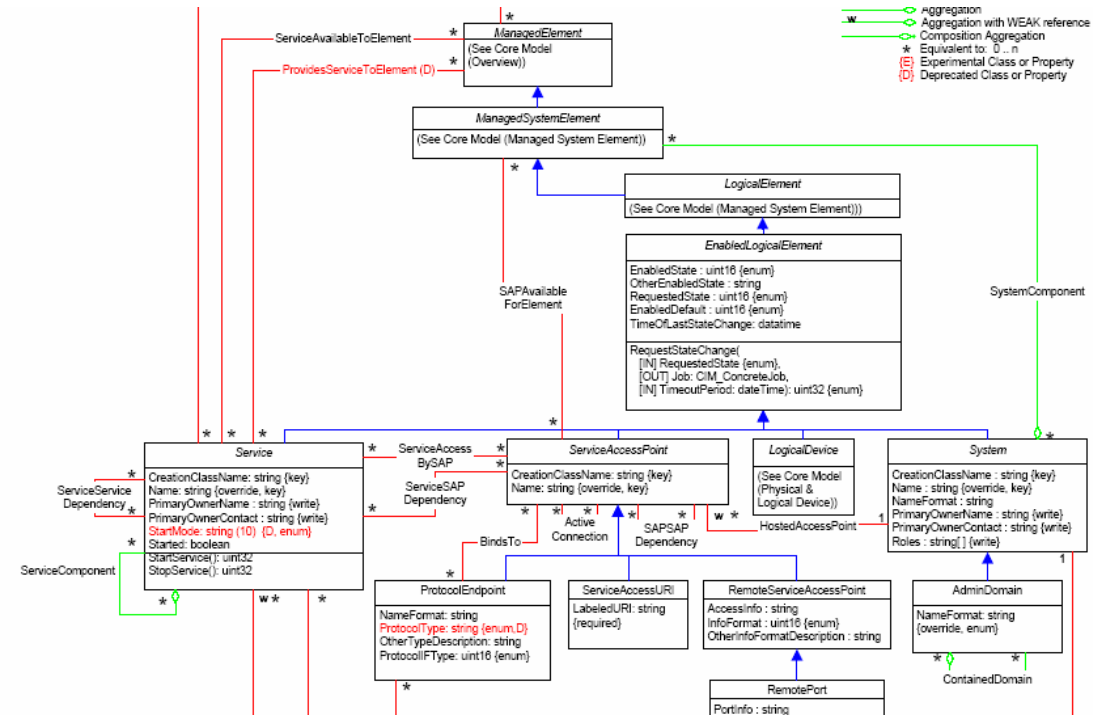


图 用 UML 表示的 CIM 模型

厂商自己也把和行业结合作为工具发展的一种突破点。例如：I-Logix 的 Rhapsody 6.0 就为美国国防部架构框架 DoDAF 提供了即时支持。

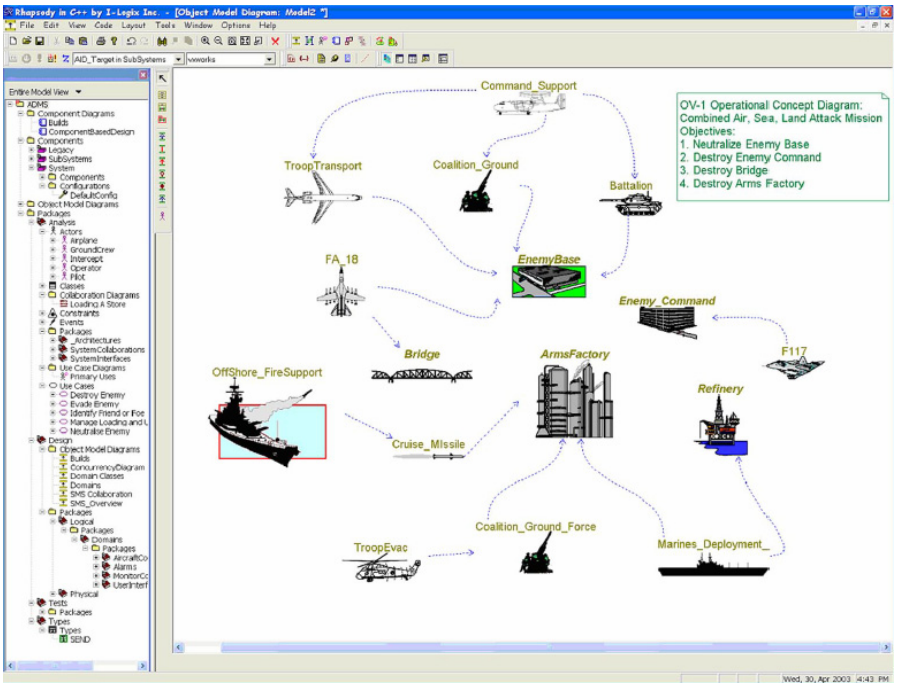


图 I-Logix Rhapsody 对 DoDAF 的支持

向更高层抽象

软件越来越复杂，人脑必须越来越集中于只有人脑才能做的事情，计算机能做的事情尽量交给计算机去做。所以开发语言一直在发展，从直接使用机器语言编程，到使用汇编语言、高级语言、面向对象语言。可直接执行的编程载体离计算机越来越远，离业务现实越来越近。下一步的抽象是什么？目前的编码过程中，有些代码成分并不需要人脑的思考，如果这一部分能用机器取代，软件公司可以省下很多人力物力（毕竟硬件越来越便宜，人力越来越贵），这也是模型驱动开发的吸引力所在。

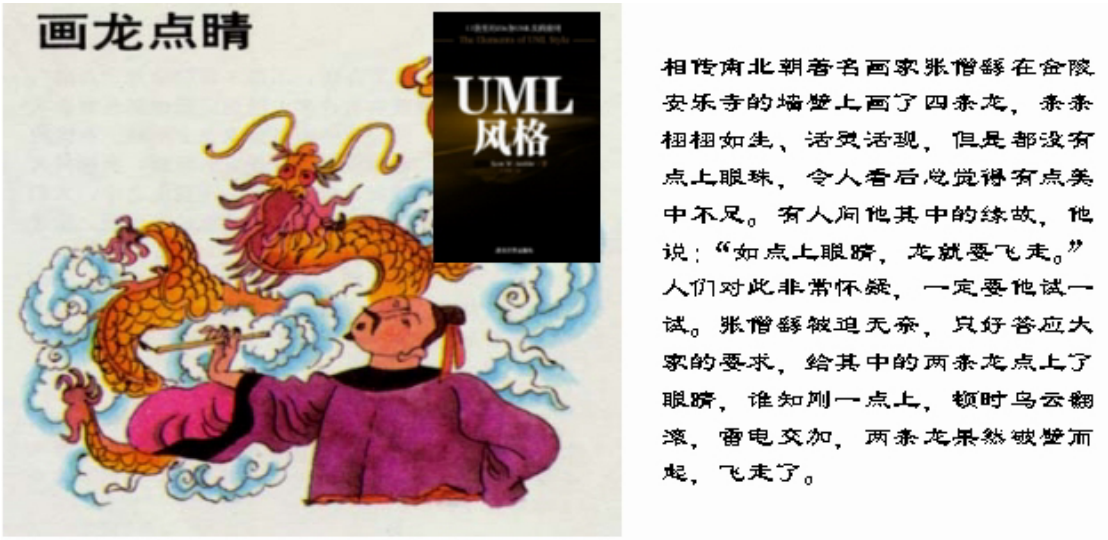
MDA 是 OMG 提出的标准，让模型和代码可以无缝的衔接起来。通过 PIM 到 PSM 映射，语言成为模型的衍生物。或者在代码环境中直接建立 PSM。最后，语言从模型中消失。

有趣的是，很多人不遵循 OMG 的标准，自己探索模型驱动开发的路线。不管如何，向更高级抽象势在必行。但“模型编译器”的成熟尚待时日，所以出现不少针对 MDA 的反对声音，和高级语言取代汇编语言时的反对声音很相似，不过高级语言发展到今天，很少有人再怀疑编译器产生的汇编语言会有什么问题。

（本文首发于《程序员》2005 年第 7 期）

征稿

http://www.umlchina.com/xprogrammer/xprogrammer.htm



它不讲概念，它假设读者已经懂了概念。

它不讲工具，它假设读者已经了解某种工具。

它不讲过程，它假设读者已经了解某种开发过程。

它只是在读者已经了解方法、过程和工具的基础上，提醒读者在绘制 UML 图时需要注意的一些细节。

在这本类似掌上宝小册子中，Ambler 提出了 200 多条准则，帮助读者在画龙的同时，点上龙的眼睛。

UML 相关工具一览 R-S（截止 2005 年 7 月）

整理

以下总结了全世界的各种 UML 相关工具，按工具名称字母排序。

工具（最新版本）	厂商&地址	试用允许	UML 版本	支持代码环境	XMI	平台	备注
RAPID RMA	Tri-Pacific http://www.tripac.com/html/prod-toc.html	有试用版					和 Rose Real-time、i-Logix Rhapsody 结合
Real-time Studio Professional 5.0	ARTiSAN http://www.artisansw.com/products/products.asp	有试用版	2	C++, Java 双向工程, 状态机模拟。Ada83, Ada95, C, SQL-DDL	√	Windows, Solaris（服务器端安装）	支持 UML2.0 和 SysML，可以和 PVCS、VSS、ClearCase、CM Synergy 结合。
reView 3.2.5	Headway http://www.headwaysoftware.com/	有试用版		Java, C++			强大的逆向工程工具，帮助分析软件结构。
Rhapsody 6.0	I-Logix http://www.ilogix.com/products/rhapsody/index.cfm	30 天完整试用	2	IDL, Java, C++, Ada, C, C166, M16C, COM	√	Linux, Windows, Solaris	聚焦于实时嵌入系统开发的模型驱动开发(Model-Driven Development，MDD)工具，能从状态图、活动图生成代码。支持 SysML。支持 DoDAF 框架。
Rational Software Architect	IBM Rational http://www.rational.com/	可以试用	2	Java, C++, VB,Ada, IDL,	√	Windows, Linux, Unix	Rose/XDE 的新版本。基于 UML2.0 的高端建模以及工具

IBM				Delphi, SQL, Oracle			集，还包括应用开发、Web 开发、软件配置管理等工具. 这些工具中都增加了对 Eclipse3.0 的支持，以及减少手工编码的各种新功能
Select Component Factory SELECT BUSINESS SOLUTIONS	Select Business Solutions http://www.selectbs.com/products/products/select_solution_for_mda.htm			IDL, C++, Java, C#, Delphi, Forte, Oracle-DDL, SQL, SQL Server-DDL, VB, Peoplesoft	√	Windows	可以和 ERWin, Caliber-RM 协同工作，Rose 输入输出。 MODEL DRIVEN ARCHITECTURE
Sequence Diagram Editor effexis software	Effexis Software http://www.effexis.com/sde/index.htm	14 天试用					专注于序列图和 call flow 图。
SILVERRUN ModelSphere 2.3 SILVERRUN ModelSphere	magna solutions http://www.silverrun.com/modelsphere2_0.html	有 Demo 版		Java	√	Java	支持业务流程建模、数据建模、UML 建模。
SiSy Developer 2.16 SiSy®	SiSy（德国） http://www.rms-deutschland.de/produkte/produkt_haupt.php			C/C++, Delphi, Pascal, Java, C#, 汇编			
SmartDraw 7 SmartDraw®	SmartDraw http://www.smartdraw.com/	30 天试用	2			Windows	支持包括 UML 在内的各种软件设计图形，支持 UML2.0。
SmartState 3.2 APESOFT SmartState	ApeSoft（印度） http://www.smartstatestudio.com/	有试用版		C++, Java, C, C# , XML.		Windows	状态图工具，100%的代码生成。
System Architect v10	Telelogic Popkin Software	15 天试用		IDL, C/C++, Java,		Windows	能够把数据模型转成类模型，

	http://www.popkin.com/products/system_architect.htm			C#, Delphi, HTML, PowerBuilder, Smalltalk, VB			支持流行的工业架构框架，如 Zachman Framework, DoDAF (C4ISR Framework) 和 TOGAF (The Open Group Architecture Framework). 。值得注意的是增加了两种针对管理层的图形作为原有图形的补充：Enterprise Explorer Diagram 用于帮助了解企业的全貌、Enterprise Direction Diagram 帮助形成企业的目标和战略。

The Unified Modeling Language Reference Manual, Second Edition

JAMES RUMBAUGH
IVAR JACOBSON
GRADY BOOCH

Covers UML 2.0

UMLChina 译
（王海鹏、李嘉兴）

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

CD-ROM included

《UML 参考手册》2.0 版中译本
即将由机械工业出版社华章分社出版

Domain-Driven DESIGN

Tackling Complexity in the Heart of Software

众多问题根源在于
领域模型没有捕获到
业务的深层内涵

Eric Evans

Foreword by Martin Fowler

《领域驱动设计》中译本

即将由清华大学出版社出版，UMLChina 审稿

OBJECT-ORIENTED SOFTWARE CONSTRUCTION

SECOND EDITION

CD-ROM INCLUDED

- *The* Most Comprehensive, Definitive O-O Reference Ever Published
- An O-O *Tour de Force* by a Pioneer in the Field
- CD-ROM Includes Complete Hypertext Version of Book AND Object-Oriented Development Environment

BERTRAND MEYER

《面向对象软件构造》中译本，UMLChina 辅助教材

即将由 机械工业出版社 出版

《人月神话》2005 最新动态

UMLChina 整理

以下均摘自网络，未作改动，包括错别字。

http://sili.mblogger.cn/posts/32624.aspx

没有银弹”初次阅读

今天完成实验后就跑到阅览室找《人月神话》来看，可是到哪里书是找到了但是位置都坐满人。原来现在阅览室可以拿书包进来，所以好多人来这里自习没办法，就站着浏览了一下目录和主要篇目，就记住了“没有银弹”。回来了 pdf，把没有银弹这章看完了，不知道这样从中间看下去会怎么样。但是觉得一些观点和我看过的《软件艺术》是相仿的，都把优秀的设计人员放在重要的位置以及开发人员的交流。不明白为什么叫人月神话原来人月可以理解成人员和月份，也就是画出开发人员和时间的函数曲线。

软件开发的难点

在这一章中，提到的都是一些软件开发的难点：复杂度，可变性，不可见性等

“我认为软件开发中困难的部分是规格化、设计和测试这些概念上的结构，而不是对概念进行表达和对实现逼真程度进行验证。当然，我们还是会犯一些语法错误，但是和绝大多数系统中的概念错误相比，它们是微不足道的。

如果这是事实，那么软件开发总是非常困难的。天生就没有银弹。”

80 年代流行的语言

但是我还是很在意提到的高级语言部分，不知道 80 年代流行什么语言，看到现在一些开发者对语言的崇拜，无语

增量开发

最后是增量开发和重视技术人员，这是和软件工艺相仿的地方，也很赞同这两个观点。只是增量开发,怎么实现？实现的难度多大？不过既然两位著作者都这样说

可见增量开发还是比较好实现的。

http://hj3622003.yculblog.com/

开始沉着气

胡说八道 @ 2005-07-04 10:36

在抚州联通做事，知道了什么叫做事，就是一丝不苟，说得轻巧，四个字，做起来要死人，柳传志说：“细节决定一切”，人月神话里，要求拿出开发软件一般的时间来测试软件，目的是使其能成为产品格兰仕的老总更绝，他操一口不太流利的广普说，“我们既然做制造业，就要做好苦行僧的准备.“他们仿佛在对我说，小子，学着点吧!!!!!!!

http://cryee.yculblog.com/

你们要问是谁啊，那就是我的父母，他们每天都在想我，但我还在这里在不学习狂餐网，我能对得起他们吗，要是对不起的话我能做点什么呢，我在看人月神话的时候有很多感觉，他们简直就不是人，为什么能做出那么复杂的系统来呢，而让我做一点点的小东西就困难得不得了，这是因为我的脑子不如他们吗？可能是一方面的原因，但更大的原因是因为我比较懒，连马克思主义都学不下去了还搞什么计算机啊

http://www.lingchunhui.com/

《人月神话》 [日期：2005-02-28] [来自：C.H.Ling]

上八点多的地铁总是让人恹恹欲睡，

斜对面一张稚气未脱的脸引起了我的注意，

只因他手里捧着一本《人月神话》。

噢，又一个做 IT 的，我想。

如果那时我认真地读完这本书，

现在应该走软件工程这条路了吧。

但这总归只是假设，人生永远是没有对错的单选题。

Believe IT or not.

http://caoxj.blogbus.com/index.rdf

这半年不知道在公司干什么事情，在各个项目中晃来晃去，技术上没学到啥，倒是在项目管理上有些心得。

1，永远不要往延期的项目中添加人手，这个在《人月神话》中说的很清楚，但是在操作上人永远会犯这个错误，公司好几个项目都是这样的，一延期，客户在催了，马上就开始来叫人去帮忙了，结果是越帮越忙。

2，做业务系统，尤其是基于 BS 的项目，3 个熟练的程序员和 30 个生手，我宁愿要 3 个熟练的程序员，这种项目，在最快的时间和客户确定需求，在最快的时间把代码编写完，把测试做完，绝对不能拖，要让客户跟着你跑，千万不能让客户有太多的空余时间。否则客户今天说这里颜色不对，明天说你给我在表里加个字段，你会疯的。

3，sales 永远是 sales，无论是特级还是顶级的，千万不能让 sales 当 PM。

4，无论多小的项目，必须先有需求再设计，千万不能边需求边设计。

http://homepage.mac.com/jslin/iblog/B225643286/C2048188367/E542115222/

所謂的「隔行如隔山」是一點都沒錯的。當不同行業的人聚在一起閒聊的時候，如果是談到了工作的部分，專業術語或是同業間溝通常用的說法一講出來，不同行業別的人要真能聽懂，那還真的是不容易。就拿理髮業來說吧。我從小時候就在我家附近的家庭理髮廳理髮，大概也光顧了二、三十年了，老闆和老闆娘常常在聊天時「勞」來「勞」去的，還真的是聽不懂在講什　。例如說「勞剪」、「勞洽」，這應該是剪髮和洗頭的意思（這是我揣摩出來的）。如果聊到股票的話，「勞賣」應該是賣出，「勞買」應該是買進（這也是我揣摩出來的）。反正在我理髮的過程當中，聆聽老闆和老闆娘一邊理著客人的頭，一邊在那邊說著理髮業的「勞語」，也可以算是我去那邊理髮的一種樂趣。據說這「勞語」是從前上海理髮業發明的，台灣的理髮業也繼續沿用。因為當時上海有蠻多間諜是依

附在理髮廳的，說「勞語」的目的是為了讓別人聽不懂，以免情報外洩。關於這一個典故，我也無從驗證，就姑且聽之吧。

。。。。。。

這種因為用語不同所引起的溝通障礙，在某個層次上面還算影響不大。如果我前面講的理髮業「勞語」的典故是真有其事，這也可以說是用以維護業務機密的手段之一。不過這種溝通障礙在跨領域、異業合作頻仍的今天，確實是讓溝通成本增加的因素之一；但反過來說，卻也是可以賺錢的地方。「人月神話」一書裡就有提到，在軟體開發的整個流程當中，一般人會以為 coding 是佔了成本和時間中最大的一部份，但是事實上，溝通所佔的成本比例往往遠大於一般人所認為的 coding 工作。這一點其實也是不難理解的。因為在一個軟體專案一開始的時候，就是要和客戶洽談，以便了解客戶的需求是什　。需求確定之後，才會知道以後要做的有哪些東西、可不可行等。

一個無法滿足客戶需求的軟體專案，即便是採用了最新、最高級的技術，所開發出來的軟體，其實對客戶來說就是廢物一個。這種「我要城門樓，你造火車頭」的窘境，其實就是溝通不良之下的後果。因此，在職場上，一個善於溝通協調的人，其實往往需多了解對方專業領域一些事物和術語。做好這個「功課」除了可以提高和他人溝通時的親切感外，對於達成溝通所要的目的，讓雙方都可以相互了解處理事務的極限在哪裡，也是相當重要的。

http://211.83.110.77/thantii/archive/2005/05/17/330.aspx

今天看了《人月神话》，看的我脸红。

我和 xionghai 和 syx 做的项目至少犯了七个错误

1、需求不明确

后期 syx 不断变更需求，导致项目目标不能明确，同时难以制定里程碑。

2、同步

我和 xionghai 在两个机器上设计，没有使用 cvs 来保证项目同步。

3、没有技术权威

根据《人》，在遇到问题的时候应该有权威技术人员直接做出决定，而我们是三个人争论最终互相妥协达到一致。

4、正交

初期模块划分的不是很好，我认为还可以提高正交性，有些模块有不应该有的耦合。

5、破窗效应

遇到了腐化代码没有立刻处理，导致了一定程度的积累，已至于已经到了我和 xionghai 不愿意修补的地步。

6、抵抗诱惑

初始项目就引入很多不是必须的功能，可以说是没有抗住诱惑。

7、可重用

代码显然有重复，有个人的问题，也有我们两个合作的原因。

http://blueline.hit.edu.cn/blog/skipper/archive/2005/03/02.aspx

上个礼拜六在 killkill 兄那里蹭了一本《人月神话》于是拿来仔细研究一番。

鉴于自己还没有什么开发经验，而且才刚刚看了 2 章，所以感触不多，且都比较浅显。

感觉最深的是作为一个团队交流、沟通是十分重要的。

于是建议 sunny 的成员都来看一遍，然后大家再谈一下感想，总结出解决的办法。

http://www.msnspaces.com/members/iceivy/feed.rss

做程序员有些时候了，当我不住的向计算机里输入代码的时候，我突然想起大学时代看过的《人月神话》，那时我还不是很懂。现在想来，通篇的论述都应该转为两个字——重用。

代码只有在不断的被重用的情况下才会增值。OOP 的设计意图，应该也是最大限度的利用已有的代码吧。人总是倾向于从复杂的体系中提取共性的东西，从变化的流程中抽象不变的逻辑，然后重用它们。这就是程序的沉淀，而且往往是做的时候能想多远，做出来的东西就能用多久。

健壮

程序需要应付大多数可能发生的情况，不论它们是被希望的还是被诅咒的，设计者会考虑每一种可能的情况——只要它会发生——并尽可能的妥善处理。永远不要假定输入是某种形式的东西，并只针对这一种情况编写代码。

弹性

动态的代码，不需要现在就划地为牢，或轻易的设定限制。永远不要在代码里为用户作决定，与其以有限的情报去猜测，不如把球踢给他们为好。

控制粒度

只有想不到，没有做不到。这句话用来形容编程真是再合适不过了。因为程序本身就是逻辑，只要逻辑上行得通，就可以用代码表述出来。

写的有些乱了，我突然发现自己在试图给自己制定框架，从而约束我的行为。这是不好的倾向。核心的意思应该是强调代码必须能够被重用，或者能经历尽量少的改动而被重用。而在设计编码的时候，就应该想到这些。

http://blog.hjenglish.com/laoshenmo/archive/2005/03/03.html

2005 年 3 月 3 日

闲闲忙忙

这两天本来还算是清闲的，每天的工作量不是很大，看着项目逐渐走向尾声，心里也很平静，常常是工作一阵之后就看些自己的东西。上午发了一份报告，本来是例行公事的，没想到几个部门的领导对这个项目突然提高了重视，一下子我也忙起来了。下午又和项目经理进行了交涉，结果是明天前需要用 xform 把那些头疼的数据模型写出来。真的是头大了，不过没有办法，谁让根据人月神话，这个时候多加人也是无益的呢。看来今晚又要加班了。不忙则已，一忙惊人啊！

http://loveotheloo.blog.edu.cn/user1/3191/archives/2005/140711.shtml

不得不说一说学校的课程与老师。

我是学工科的，计算机科学与技术专业。坦白讲，学校的授课计划还是经过论证的，那一系列的课程名称，大部分都是比较重要的。可是饮料的标签，与饮料本身的味道，又有什么关系呢？都是清茶，不同的茶馆泡制出来的，味道是天壤之别。现在的大学，的确失去了泡茶的好传统。

后来经历的老师多了，就悄悄的把大学的老师分为三类。

。。。。。。

其三就是误人子弟的。这样的老师，比不思进取的学生还可恨。而且千姿百态，更有多种表现。或课堂枯燥，言语渐无，或嬉皮笑脸，不着边际，或治人为乐，无上光荣……我们广为流传的一句，是在上软件工程的课时，老师侃出来的一句话：我发现了一个秘密，就是使用 VB 时，你使用一次 Tab 键能让光标退四个格，使用两次就退八个。我的软件工程啊，这如何与《人月神话》比较呢？但那句话，成了我们最好笑的笑话，想起那位老师我们就想起了这句话。或许有些罪过，可是天下中所有的轻视，最厉害的是学生对老师的轻视，钱钟书先生说的，这种轻视总是最为深刻。

就第三类老师，大学也占了四分之一。

http://www.cjsdn.com/post/view?bid=16&id=27159&tpg=11&ppg=1&sty=1&age=0

强烈反对“人月神话”的书名，

应该改成“人*月神话” (*该使用中间的‘.’号来表示)

----人家英文'The Mythical Man-Month"

Man-Month ----本意应该是算人力任务资源时，人*月(*该使用中间的‘.’号来表示)，一个人做了一个月的活的意思。而整本书也正是为了，表述在软件开发中没有能够代替人*月，每人辛辛苦苦的苦干的“银弹”。

http://jarod.turbozv.com/index.php?job=art&articleid=a_20050128_103534

忧郁了很久以后，我决定重构 kitty 的结构，否则，我们会在上面花更多的时间。

我的计划是把结构调整得简陋一些，便于更为合适得移植到在轻巧得设备上。现在得结构已经复杂到不能维护得地步了；

关于 kitty 第一个版本开发的教训：

1。没有快速的建立起可以迭代开发的基准版本；

2。对 c 函数的单元测试做得不好；

3。对 jit 得设计没有一个完整得概念；『参考人月神话』

4。人员的安排问题

kitty2 的开发需要注意如下几个要点：

1。快速的建立起可以迭代开发的基准版本；

2。设计精简实用的结构。

3。测试要自动化。

http://merlin.tianyablog.com/blogger/view_blog.asp?blogID=33631&CategoryID=0&idWriter=0&Key=0&NextPostID=220873&PageNo=3

2004-4-18 星期日(Sunday) 晴

방방看到《人月神话》第二章的 menu，让我想起了上次 formal hall 的经历。

방방剑桥的 formal hall 有着很长的历史，是一种在高雅气氛下的社交活动。

방방每个学院都有 formal hall，时间多为通常在周二和周四各有一次。

방방

방방 formal hall 实际上就是正式聚餐。还记得 Harry Potter 里面魔法学校的 feast 吧？

http://hollando.blogchina.com/hollando/1386528.html

有幸跟 Qi 博打交道了一段时间，觉得他确实是有一套！本人获益匪浅，总结 Qi 博项目管理的几个特点：

一是非常重视计划，项目里程碑排的满满的，基本上两个里程碑（他称之为牛肉串，与《最寒冷的冬天是旧金山的夏天》中关璐的提法有点相似）之间不会超过两周时间。每周周五下午开一次两个小时的研发例会，每人都要填一份表，向组内成员介绍自己每一天都干了什么，目标完成率是多少？用在研发上的时间占的比例是多少？介绍完一周干了什么之后是计划部分，项目组的成员先根据大家的目标完成率进行讨论，适当地调整项目计划。如果有成员本周完成不了分工的，采取相应的措施。或调整分工，或改变需求抛弃不重要的功能（他也看过人月神话，探索需求等书）确保项目能完成。每人根据讨论的结果填写表中的计划部分。然后再向大家说一遍，其中项目组什么东西是并行的，什么东西先做什么东西后做他必须要你说清楚。

这种做法的结果是你如果真的是按照你计划来做的话，发现你 8 个小时的工作时间已经非常足够了。当然研发例会会消耗掉 2 个小时，但计划的时候已经把那时间算上了。计划好了你就会有事半工倍的效果。另外，由于每人做了什么都要向别人说一遍，所以避免了 IT 企业能者多劳而又吃力不讨好的弊病。

http://www.windcat.idv.tw/archives/000196.html

2005-03- 8, 06:06 PM

無心工作..

沒辦法想事情...更別說是看程式了...

掏空的感覺...就是這樣讓人厭惡...

翻著人月神話..雖然只是為了讓手晃一晃...

有意...亦或是無心...看到一段話...

造成災害的原因通常不是突如其來的龍捲風，而是源自白蟻日積越累的啃蝕

愛戀的感覺不也是這樣嬤.....

慢慢的..越來越多..越來越...

想到我的時候....妳會開心嬤.....

http://www.sevenkiss.tianyablog.com/blogger/view_blog.asp?BlogID=71928&CategoryID=68619&idWriter=0&Key=0

其实《人月神话》真的也很不错，只不过我当时看的影印版，除了没看懂的应该就是理解有偏差的，无法作个评定。不过只要是我看不大懂的应该就是好书。

http://loyaldog.blogchina.com/1197621.html

我是这样领导一个学生项目的(2)- -

第二次会议在一周之后举行，也就是昨天（还记得吗，我是在小组成立的第八天记录下的这篇文章）。一周以来，我明确了一个思想：项目经理必须和首席架构师分开，哪怕是四五个人的小型项目。这得感谢《人月神话》以及其他的软件项目管理的书籍（他们大多来自于机械工业出版社的软件开发技术丛书系列）。在以前的两个项目中，我都将项目管理和系统开发以及首席编程和测试融为一身，以至于曾经一个组员反问道：你都做完了还要我们干吗啊？的确，明确的分工是必要的，用人不疑，疑人不用。一个成功的项目经理应当懂得如何将工作交给适当的人去完成，而不是独自去完成所有的事。于是，我决定由另一个人来担任首席架构师的角色而不是我自己，这在我看来是一个破天荒的决定（尽管有些可笑），很多人（特别是学生和一些积极性不高的人）都会认为技术最好的人理应当成为项目经理，这也许也是我成为项目经理的原因，然而事实上要记住的是，项目经理是管理人的，而不是管理技术的，他所要做的仅仅是辅助首席架构师完成协调的工作，技术的事还是交给别人去完成吧。

http://pander.yculblog.com/post.667871.html

脚印 @ 2005-05-12 10:58

我们发专业英语的课本，也叫人吐血啊！一个汉字找不到不说，里面的软件工程思想更是表达的那个叫曲折！：（ 不过名字还是好听的，叫《人月神话》。刚开始看的时候感觉很浪漫，后来知道人月就是一个人一个月的工作量

本来该半个小时读四篇的,却一篇读了半个小时,还都是错的,好受不了哦.

读完都觉的很内疚呢.

http://yehou.blogchina.com/906888.html

《如何借鉴商业常识进行人的管理》（摘）--

《程序员 2004.12》P.64

在明白了基本原则后，实际操作部分是很简单的。现在就让我们看看如何进行具体操作。我比较喜欢的一招，就是推荐程序员阅读软件工程和项目管理方面的书籍（例如《人月神话》什么的），同时我还要开玩笑地告诉程序员，我推荐项目管理书籍的目的就是希望程序员明天就能取代我的位置（顺便说一下，我从不向我管理的程序员推荐《给加西亚的信》,《没有任何借口》之类的所谓"员工必读书籍"）。我相信这样做可以完全证明我是真诚地关心每一个程序员的。其他关心程序员的办法还有一些。例如在一个项目组内，尽可能地使每个程序员都有机会尝试不同的工作（配置管理，程序设计，需求分析等等），以使得程序员掌握程序开发的完整流程。又例如承诺不加班并且兑现承诺等等。

http://mental.mentsu.com/mental/atom.xml

你可以接受技术不如你的人的管理吗？

事实上对于在软件公司里的底层开发人员来说，面临着两种权威：一个是技术权威，一个是行政权威。二者是分而治之还是合二为一，便是在国外也没有统一的做法。在《人月神话》[5]第七章中，Brooks 认为：对于小团队来说最好合二为一，而大团队则最好分而治之。而微软的产品团队[7]则由产品管理、项目管理、软件开发、软件测试、后勤管理、用户培训六个环节组成一个环形结构，保持相互的良好沟通。这六个团队本身是平等的，每个团队有自己的负责人，六个负责人组成一个管理团队，同样要保持相互的沟通。如 Brooks 所说[5]：交流及交流的结果--组织，是成功的关键。

可以注意到上面说的微软团队有一个很关键的地方：平等。其实 Brooks 认为大团队应该将行政管理与技术管理分而治之时，虽然他倾向于由产品负责人主导工作，但是他也反复强调了产品负责人必须保障技术负责人的技术权威，而且他在其中举例说明应该由产品负责人主导时，表达的意思也是：让技术负责人专注于技术管理方面，由产品负责人去承担其它方面。

这就意味着在这样的组织中，并不存在绝对的上下级关系，人治是行不通的，只有通过完善的法治才有可能。而如果放在中国的文化背景下，很可能演变成：产品负责人对组织有绝对的管理权；或是微软式的组织中，产品管理和项目管理团队被提升到高于另外四个团队的情况。而一旦这种平等状态被打破，则良好的交流环境就不复存在（上下级间的交流是不对称的），先进管理理念反而可能造成不良的后果，一如失败的王安石变法。

自从印度软件业的成功之路在国内被大肆宣传开来之后，国内充斥着大量王安石式不切实际的想法。虽然约二十年前 Brooks 就说过：没有银弹[5]。

这一经典论断导致了两个极端的结果：一部分人坚持有银弹，另一部分人则认为没有任何弹。因为 Brooks 只预言了十年的时间，现在已经快二十年了，没有银弹已经成为共识，但还是如 Brooks 所说：乐观主义是这个行业的职业病[5]。认为没有任何弹的人还是少数，大多数人坚信有铜弹，至少是铁弹。不能得到一个数量级的提升，能有三五倍甚至只是不到一倍的提升也是值得的。

应该说这种想法是有道理的，而且我们也看到这二三十年来软件业的长足进步，证明了铜弹铁弹的存在。但问题在于，这些“弹”不是包治百病的大力丸，谁吃都可以的。这个“某弹”在别人那里能得到一倍的提升，那只是对于别人而言，对你呢？可能是提升三倍，也可能下降三分之二。

我们需要的是对症下药因情况而异。

在《大历史观》[9]一文中，我说到中国目前存在着大量的小软件公司，是因为国内目前的软件项目市场不规范造成的，也就是说有一个环境在那里放着。而这个环境是我们所无力改变的，我们所能做的，就是去适应它。

对于那些靠一两个项目起家的小软件公司来说，一个重要任务就是如何摆脱这种靠天吃饭（通过关系拿项目）的困境。特别当看到像印度这样的国家--我虽然没有去过印度，但是认识的人里去那取过经的为数很是不少，据说印度国内的经济状况比我们要差得多--其软件业的发展程度却让我们望尘莫及。于是国内的人们开始试图通过取经学习，以达到大跃进的目标。

然而如我上面说，没有看到双方情况的差异下引进管理理念、开发技术以及一切貌似“某弹”的东东，结果却发现它们都只是“苦杏仁苷”[6]。剥一句《手机》里费老的名言：

盲目，人家情况不同的。

http://rosat.blogdriver.com/rosat/517565.html

为啥做这行都这样呢？- -

http://community.csdn.net/Expert/topic/3388/3388903.xml?temp=.2011835

也许这样作,至少可以给自己的初恋一个全尸?

放爱一条生路,留给真正值得自己去爱的人吧!

为什么爱情不是想像中的那样?

程序代码再复杂,我总可以从容的面对,付出总会有真实的回报,

同样是感情,为什么,对人,就这么复杂?

漂泊的心灵,依旧继续漂泊...

爱已经要求得这么简单,却也不能编译出可以运行的版本?

看来我还是应该好好研究一下<<人月神话>>,加强一下项目管理的经验...

对于爱情这个失败的项目,又该怎么去重构...:-(

唉，彼此，so so。

爱情没有重构，就像生活没有 Debug

http://www.javamodel.com/?q=blog

再论托普得“倒掉”

提交人：smallduzi 时间：Mon, 07/12/2004 - 02:51

而这家北京公司在南京共 96 人在开发南京地税的项目，做了半年竟然没有一点进展，老总在南京那门怎么 96 个人怎么没有进展呢？哈哈，看来他要看看《人月神话》了。听说后来又接了其他地税的项目（厉害）。在中国只要软件差不多，主要还是看你有没有过硬的关系，这也就会产生中国式的程序员。

http://shaokun305.blogdriver.com/shaokun305/724847.html

在回答这个问题时，我们可以通过回顾 OOP 的历史学到很多东西，特别是 80 年代的那段历史。第一个需要处理的是避免过度神话某项技术。任何看过《人月神话》的人都要谨慎的对一个新技术做出过高的判断。我们必须小心不让我们的狂热使得我们听起来正在宣称 AOP 讲解决所有的软件问题，是 Brooks 说的软件产业的银弹。AOP 是一个封装代码的技术——比如一些好的开发者，他们理解要解决的问题，他们也许可以用更多的人月的工作时间去完成他们的工作，AOP 将帮助他们开发更好的，更干净的，更容易重用的代码而且更快更简单。它实际上也许比那做得还要多一些，因为 AOP 会帮助他们更容易使用 OOP 技术，一个可以帮助我们在软件流程中很好划分代码的现代主流封装技术。AOP 是开发者工具箱中重要的新工具，解决 OOP 开发中类层次的软件结构上的困难问题。当然我们对此必须要很小心，因为这很容易妨碍我们对 OOP 方法正确应用，把一些可以用 OOP 技术很好解决的问题，用 AOP 勉强处理而导致非常复杂的流程，OOP+AOP 也许会产生四不像技术。从这种角度上说，AOP 也许仅仅是一种新的工具，但是我们要知道，在历史上很多时候，衡量一个技术的成熟，其实很大程度上是根据其使用的依据的。在分子生物科学，核物理，使用的工具往往决定了发展的速度。

http://www.ihower.idv.tw/index.php?cat=9

一天不到就看完啦... :p

作者用一個神奇的故事來講專案管理的一些秘訣...

其中提到了人月神話啦~壓力對專案的影響~估計的重要等還不錯的概念...

不過感覺可讀性沒有想像中的好看... ^^||

http://www.uooule.com/rss/feed.php?channel=133&y=2005&m=06&d=03

2005-6-3

一、一个适合的团队规模、风险级别和应用领域的开发进程。

多大的团队规模才是合适的？《人月神话》已经清晰的告诉我们，团队的规模不是成线性曲线的。曾听说过一个组长带领三个核心组员，一个核心组员带领三个组员的搭配，这样的话就是 1+3+9=14。

风险级别无疑是和回报挂钩，风险高意味着回报相对高，反之也亦然。但是我们往往会忽视一个问题，我们的团队能承受多大的风险？我们的团队风险抵御能力有多大？

。。。。。。

二、高度集成的并且支持项目要求的交流类型的工具。

Software Engineering 讨论最多的问题就是交流的问题，团队怎么交流成了一个热点，当然也是一个难点。很多团队把几乎 60%的时间用于交流，剩余的 40%的时间用于干实事。（王云先生语）由此可见，沟通一直是团队最头痛，又是没有办法去回避的事情，很不幸的再告诉您一个事情——交流没有银弹。当然，所用的工具也就没有没有了必杀计。

http://www.hotlong.com/forum/thread43130-0.html

在经历了 Alitalk 那段还算不错的日子，自己也确实该接受<人月神话>里面所谓的第二项失败的项目了，虽然并没有想象居然会做到如此难堪的地步．由此，我也更加确信，自己在

系统设计/项目流程/为人处世等等方面的缺陷也是到该强化补习的时候了．

http://koman.blogchina.com/1535583.html

闲人的出现我认为是好事情,应该组织他们学习一些新的东西,做一些内部资料的整理,搞搞技术交流讲座．

在现有的项目组中突然间出现大量人员,项目负责人都有点忙坏了,任务要重新分配,交流要重新进行,原有的项目进度被打乱,因为有人认为项目进度理所当然的应该加快了．

虽然都看过<人月神话>,那又怎样了?

也许有人理解了项目进度不是简单的人月数除以人头,可是文章中提到的主要问题和次要问题,有几个人能理解呢?

创造出结果的是人,确切的说是积极的主观能动性,从工作中获取成就感．

加了人,我们的合同既不会早签也不会晚签,钱不会早来也不会晚来．