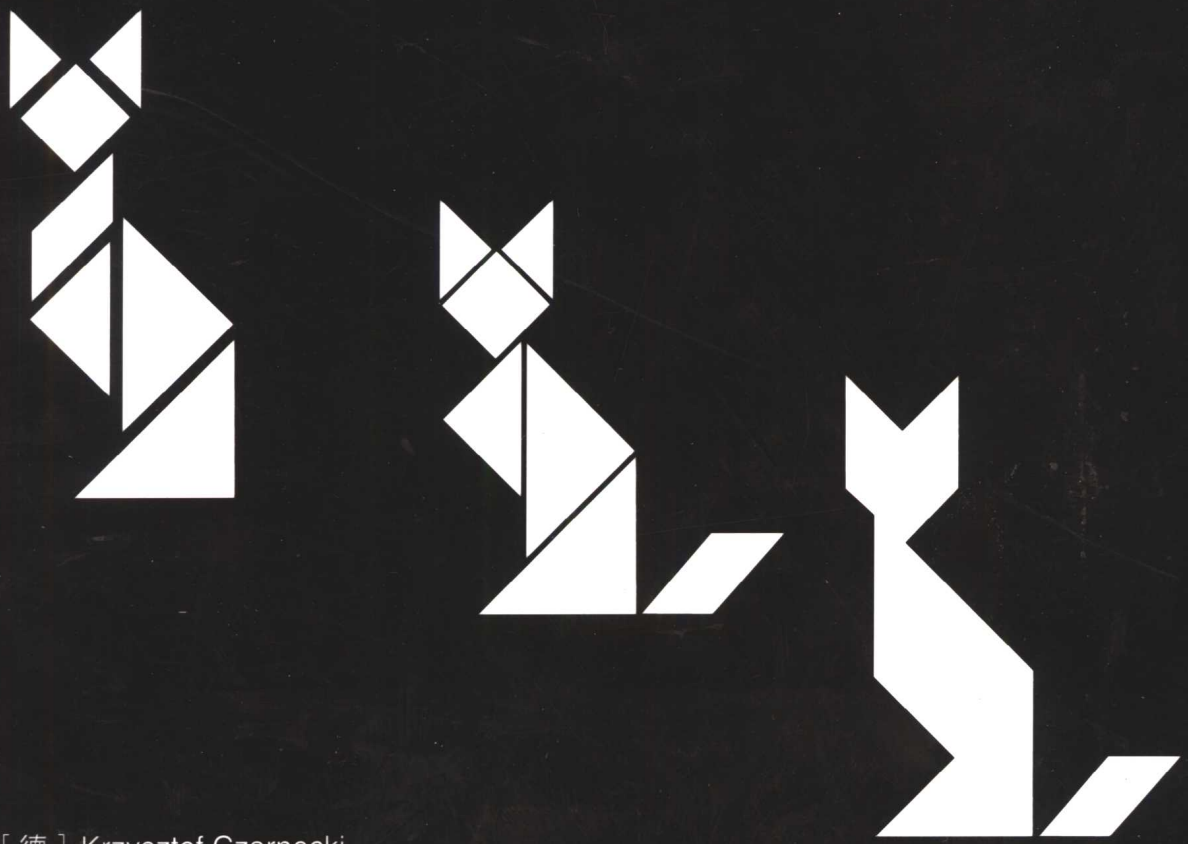
开 发 大 师 系 列

Addison Wesley

Generative Programming
Methods, Tools, and Applications

产生式编程
方法、工具与应用



[德] Krzysztof Czarnecki 著
Ulrich W. Eisenecker
梁海华 译

中国电力出版社
www.infopower.com.cn

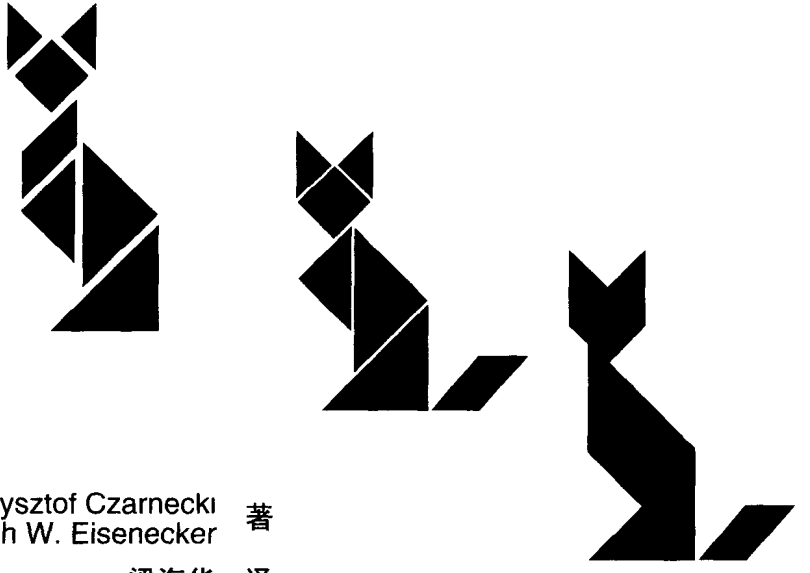
Generative Programming:
Methods, Tools, and Applications

产生式编程：方法、工具与应用

开 发 大 师 系 列

Generative Programming
Methods, Tools, and Applications

产生式编程
方法、工具与应用



[德] Krzysztof Czarnecki 著
Ulrich W. Eisenecker
梁海华 译

中国电力出版社
www.infopower.com.cn

Generative Programming: Methods, Tools, and Applications (ISBN 0-201-30977-7)
Krzysztof Czarnecki, Ulrich W. Eisenecker

图书在版编目（CIP）数据
产生式编程——方法、工具与应用 /（德）扎莱基，（德）爱森克编著；梁海华译．—北京：
中国电力出版社，2004
（开发大师系列）
ISBN 7-5083-1826-9
Ⅰ.产... Ⅱ.①扎...②爱...③梁... Ⅲ.软件开发 Ⅳ.TP311.52
中国版本图书馆 CIP 数据核字（2003）第 110217 号

丛 书 名：开发大师系列
书　　名：产生式编程——方法、工具与应用
编　　著：（德）Krzysztof Czarnecki, Ulrich W. Eisenecker
翻　　译：梁海华
责任编辑：姚贵胜
出版发行：中国电力出版社
　　地址：北京市三里河路6号　　邮政编码：100044
　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：北京丰源印刷厂
开　　本：787×1092　1/16　　印　　张：36　　字　　数：923 千字
书　　号：ISBN 7-5083-1826-9
版　　次：2004 年 2 月北京第 1 版　　2004 年 2 月第 1 次印刷
定　　价：59.80 元

前　言

在很多新的程序设计书籍出现的时候，都会使人们产生这样的疑问：这本书在历史发展中会占据（或者将要占据）什么样的位置呢？就是说，一本书甚至可以促使问题本身来讲述某些激动人心的东西——有关范型的改变，或者是有关新思路或者打破传统方式的一些东西。而现在，这对于本书来说也是十分中肯的。

过去十年里，面向对象方面的著作由于覆盖了很多明显不相关的思路，可以说是已经到了汗牛充栋的地步，这些思路包括从面向主题的程序设计到组件，林林总总。它们有一个共同点，就是总有一些有关它们的东西肯定不是面向对象的，虽然它们都在面向对象的光环里面花费了很多时间。但是在过去的一年或者两年里，这些零散的片段中有许多都发现了共同的基础，并且围绕一个共同的主题联合起来：元对象协议（metaobject protocol）、自省（reflection）、意图（intentionality）、一种见解深刻的对组件的解释、跨结构的特征剪裁，以及对简单模块性的经典模型的突破。对我来说，1999年度最精彩的场面之一是在德国的Erfurt举办的第一次国际产生式与基于组件的软件工程讨论会（GCSE'99），会议上，众多研究人员把许多这种想法联系起来，交流对于它们的重要意义的感想。但是，这只是这些思路汇集起来的一个论坛；我们在论坛上看到了与许多会议讨论小组和某些新的文献中相同的东西。

在形成历史的某些事件中评估历史总是很困难的，但是比起在事件发生后，远离第一手资料来解释它，可能不用负更多的责任，也并不危险。在这种思路之下，看看周围，可以认识到，现在，我们可能正处于计算机科学的一个关键的转折点，特别是程序设计和设计技术领域。业界已经挣扎着努力超出面向对象范型的局限。模式是一个值得重视的尝试，虽然它们非常有利于把注意力集中到程序设计中经验的价值和人的因素上，但是，计算机科学如果忽视亚力山大的观点或者系统思考，就不可能取得什么进步。软件很少能在Kuhnian的意义上取得一个真正的范型转变。可能我们是一个保守团体，而且我们对小说的迷恋使我们甚至不能进行最简单的学习；许多本国的组件运动的原则受到面向对象设计的早期原则的影响，这些是经验建议应该在几年的应用后丢掉的东西。

但是现在有了一些改变的萌芽的印记。可能业界不会很容易地进行革命，但是，它可以容忍向一个在现有状态之上构建的新技术迈进。一直以来，有一种强烈的、连续不断的运动，想要发现超出对象和概念与特征的范畴之外的编程和设计表达方式。这就是意图编程的基本含义，例如，很大程度上，也是类似领域工程的基本含义。这种思想的脉络在整个业界、在很多论坛上都很有市场。我们在通俗（非亚力山大）模式运动中看到了它；我们在面向方面的程序设计中看到了它；我们在泛型编程和一些例如多范型设计中也看到了它。由于一些会议例如OOPSLA的关注点越来越不集中于那些被称为对象的东西的基础，而是越来越集中于概念性扩展，所以我们开始对各个部分思考得越来越少，而对系统和特征思考得越来越多。并且由于OOPSLA影响力减弱了，所以一些类似GCSE的会议正在越来越多，越来越普遍。今天，就是一个基于广度的沿此方向的运动，并且本书发现自己正好处于这个转变的中心。我相信，读者既不会低估这个转变的意义，也不会低估本书在沟通和

影响这个转变中的地位。我们正处于演变的开端，本书是这个新流派的早期标准化工作。

这本书可能反映了计算机科学成熟的一个程度，就是说，第一次，尝试利用其自身的能力，证明一种整合观点，而不是通过与先驱区别开来颂扬一种技术。所以，这本书覆盖了很多领域，并且应用了很多原则：组件、对象、方面、自省、意图程序设计和泛型程序设计（用于以技术为焦点的地方）和领域工程（用于系统的考虑）。对于那些第一次遇到它们的人来说，本书是一个很好的对于正在涌现和建立的技术的精彩介绍。有人会误以为本书是一本已有实践的纲要，当然是在领域工程的范围内，但是，在更远的范围内，对于超出对象以外的那些东西，本书也是适用的。

但是这本书远远不止是技术的一个集合。这本书围绕统一原则提出了强力的主题，这些原则将把零散的东西捆绑在一起，还提出了非常值得注意的领域工程和元程序设计。作者用领域工程的概观好好地招待读者，并引导读者鉴赏它的必要性。领域工程可能超出了任何一种单独的思路，提供了一个通用的思路或者主题，读者可以使用这些来“解释”产生式编程。但是，更重要的是，作者把其他程序设计的学派作为基础材料，来描绘以一种广泛的、恰到好处的视角，称为产生式编程。结果肯定超过了各个部分的总和；大多数这些部分都曾经存在了很久，但是关于怎样把它们整合起来的想法刚刚涌现出来，并得到了广泛的揭示。

要理解下面这一点是很重要的，本书既不是一些新的分支，也不是仅仅对一些值得注意的小范围的技术进行一个展示。它是一种更加巨大的欢愉的预兆，预示着将打开通向新纪元的一扇门。它包括一些正在成熟的一些技术，例如自省、元程序设计和方面，这些可能会增强甚至取代对象，在未来的几年里来成为主流程序设计实践的基础。

而且即使这种历史性的分析是错的，也不能减低这本书的价值。这些思路非常成熟而且是不朽的。虽然这本书使用了大量的以前的C++代码，但是这些思路会促进使用任何派别的语言社团思考，并且对于大多数编程意识流派都是适用的。

本书只是一个起点而已。本书终将引起无数的追随者：他们将会发展出能够帮助设计人员和程序员更好地掌握和应用产生式编程的方法和过程，还会发展出将这项工作推向下一代研究的形式化方法和统一模型。对于那些愿意探究对象的限制的研究人员和实践人员来说，这本书指出了障碍，并挑战他们来更进一步把这种想法推向更远。我相信没有什么东西比起你迎接挑战更能够让作者高兴的了。当手中拿到第一本成品时，一本书的出版给作者带来的最初的喜悦，几乎可以算作惊喜若狂。但是更大的喜悦，甚至可以替代开始的喜悦，将会是在很久以后，在逐渐出现的实践中，看到这本书留下了它的印记。很少有书能给作者的投入带来如此回报。我非常希望，也非常相信本书就是这少数几本书中的一本，那是因为我相信这些思路的威力，将会深深吸引作为读者的你，并且促使你采用这些新的令人印象深刻的编程和设计构造。我希望在从本书的学习中，你能够获得极大的快乐，就像我过去一样，并且恳请你通过你自己的程序中的技巧，来帮助推动这些思路成为准则，从而找到你作为这场运动的先驱在历史上的位置。

James Coplien

Naperville，Illinois

致　谢

首先，我们要感谢我们的家庭，因为有了他们的爱、鼓励和耐心的支持，这本书才得以诞生。

在我们进行本书相关的研究工作中，我们很高兴指导了两名非常优秀的学生：Tobias Neubert 和 Johannes Knaupp，对他们我们表示特别的感谢。Tobias 实现了在 14 章展示的通用 C++矩阵库的一个更加全面的版本。Johannes 提出了对模板元程序设计技术的几项改进，扩展了 C++矩阵库，并且开发了一个可用于 LU 矩阵因子分解算法族的配置产生器。他也评阅了本书的部分内容。

同样要特别地感谢 Charles Simonyi、Greg Shaw 和微软研究院的整个 IP（Intentional Programming，意图编程）团队成员，他们为我们提供了 IP 系统，并且在 1998 年 1 月 Krzysztof Czarnecki 的研究访问期间，耐心地为他讲解 IP 的方式。

我们同样要感谢 Gregor Kiczales、Crista Lopes、John Lamping 和施乐 PARC 的 AOP（Aspect-Oriented Programming，面向方面编程）团队的其他成员，他们为我们讲解了 AOP 的思想，为我们提供了 AspectJ 的早期版本，并分享了很多启发性的想法。

在有关泛型编程的 Dagstuhl 研究会期间，就是 1998 年 3 月，我们与 Robert Glück、David Vandevoorde 和 Todd Veldhuizen 一起，成立了一个研究活动库（Active Libraries）的非正式工作组。我们在 Dagstuhl 进行的很多讨论以及以后通过电子邮件的交流，对这项工作的后期产生了很有意义的影响。我们非常感谢所有的小组成员。

在几次会议和专题讨论会期间，我们遇到了很多有趣的人，他们来自软件重用、面向对象和其他的社区，在讨论或者评阅这本书的部分内容时，他们给了我们难以估价的反馈，并与我们一起分享他们的灵感。我们感谢 Mehmet Aksit、Ira Baxter、Don Batory、Ted Biggerstaff、Kai Böllert、Ulrich Breymann、Greg Butler、Simon Dobson、William Harrison、Mike van Hilst、Sholom Cohen 和研究领域工程和对象技术的 WISR'97 工作小组的参与者、Jim Coplien、Serge Demeyer、Stan Jarzabek、Ullrich Köthe、Shriram Krishnamurthi、Karl Lieberherr、Andrew Lumsdaine、Satoshi Matsuoka、Mira Mezini、Pavol Navrat、Oscar Nierstrasz、Harold Ossher、Elke Pulvermüller、David Quinlan、Dirk Riehle、Lutz Röder、Jeremy Siek、Mark Simos、Yannis Smaragdakis、Douglas Smith、Andreas Speck、Christoph Steindl、Patrick Steyart、Allan Stokes、Peri Tarr、Hans Wegener、Bruce Weide、Eric Van Wyk 和研究产生式和基于组件的软件工程小组（www.prakinf.tu-ilmenau.de/~czarn/generate）的许多成员。我们也要感谢许多其他的研究人员，包括 John Favaro、Martin Griss 和 Jim Neighbors，他们的工作对于拓展我们的思路起了很重要的作用。我们知道这份名单不能囊括所有帮助过我们的人们，希望忘记提及的人们可以原谅我们。

我们还要感谢 Addison-Wesley 的整个团队，特别是 Marina Lang、Heather Peterson、John Fullet 和 Anne Marie Walker，本书的成功推出正是得益于他们的帮助和支持。

这项成果有一部分是 Krzysztof Czarnecki 在攻读博士学位期间开发的，这项研究由 Daimler-Benz AG 和德国教育、科学、研究和技术总署（BMBF）通过 OSVA 项目提供资金。Krzysztof 非常感谢他在 Ilmenau 技术大学的博士导师 Dietrich Reschke 和 Reinhold Schönefeld，他在 Ulm 的 DaimlerChrysler 研究和技术中心的同事，特别是他的经理 Wolfgang Hanika 和 Roland Trauter，他在 OSVA 项目组的同事 Bogdan Franczyk、Michael Jungmann、Wolfgang Köpf 和 Witold Wendrowski 和最后但不可忽略的 Udo Gleich。

目　录

第一部分　分析和设计方法与技术

第二部分 实现技术

第三部分 应用例子

第四部分 附 录

第 1 章

本书所讨论的主要内容

1.1 从手工作坊到自动装配线

人们对于软件工程的期望可谓高矣。这些期望涉及到掌握高度的复杂性，取得高度的生产效率和质量，以及便于有效地维护和改进。不幸的是，当前的软件工程并不能满足这些高标准的期望。它落后了制造业一个世纪还多。它更像是手工作坊的“一种一个”的解决方案，而不像是一个成熟的工程。

产生式编程就是使用组件并以一种自动化的方式（其他工业已经使用这种方式成年累月地生产机械、电子和其他产品）来制作软件产品的。在软件中，向自动化制作软件的方向转变需要经历两个步骤。首先，我们需要把焦点从开发一个系统转移到开发一个系统族上来——这就允许我们开发出正确的实现组件；第二，我们需要使实现组件的装配成为自动化的，这就需要使用产生器（generator）。

让我们使用一个隐喻来解释这个思路[CE99a]：假设你正在购买一辆小汽车，却不打算要一辆现成的小汽车，你搞到了自己组装一辆小汽车所需的所有部件。实际上，可能有些部件彼此之间不是很匹配，所以你必须做一些修正以使它们配合好（也就说改造它们以使它们适用）。这就是当前在基于组件的软件工程中的实践。Brad Cox 把这种情况比作工业革命前的临界点：在经历了 25 年不成功的努力以后（例如 Eli Whitney 的探索工作），直到 1826 年，John Hall 最后成功地使用可交换部件制造出了步枪（参见[Cox90,Wil97]）。然后，又经过了几十年的时间，这种使用可交换部件大规模生产的突破性思路才扩展到其他的部门。

即使你使用一个为适用而设计的基本组件库（例如 C++标准模板库、STL，参见 6.10 节），你仍然要手工装配它们，还有很多的细节需要考虑。换句话说，即使你不需要进行修正，你还是必须自己装配汽车——这就是计算机技术的“拼装游戏原则”。

的确，你更愿意只说出自己想说的话，通过使用抽象的术语来描述它，来定购一辆现成的小汽车。例如，“给我一辆 Mercedes-Benz S 级，包括所有的附件”或者“一辆为竞赛而专门定制的 C 级，具有高性能的 V8 发动机、4 轮排气盘式制动器，还有一个滚柱罩”。这就是 Generative Programming（产生式编程）想要为应用程序员解决的问题：程序员使用抽象术语强调他们需要什么，产生器产生希望的系统或者组件。

如果你设计实现组件，使之适应于通用产品线结构，同时对配置知识（configuration knowledge）建模，强调如何把抽象的需求转变成特定的组件群，并且使用产生器实现配置知识，那么这个魔术

就可以运作了。这些步骤分别与汽车制造工业中所发生的事情相对应：Ransom Olds 在 1901 年引入了可交换部件的原则，这是装配线引入的前提，它后来由 Henry Ford 在 1913 年进一步精化和推广，并且最后在 20 世纪 80 年代使用工业机器人实现了自动化生产。

注　意

一些人认为装配线的主要目的是生产大批一模一样的产品，这在软件中与拷贝 CD-ROM 相对应。这离真相可是十万八千里。例如，德国 Sindlfingen 的 Mercedes-Benz 装配线生产了成千上万的 C-、E-和 S-级的变种（单单 E-级就有大约 8000 座舱变种和 10000 座位变种）。几乎没有两辆一模一样的汽车在同一天在同一条装配线上“落地”。这就意味着正确种类的发动机或者其他组件必须可以在正确的地点和时间能够找到（以减少存储消耗）。当不同的人在汽车代理商那里订购不同的汽车时，整个过程就开始了。满足顾客的订单需要大量的后勤和组织工作，涉及到很多的配置知识（事实上，他们基于配置规则使用产品配置器）。由此可见，在汽车工业与利用标准组件库构造解决软件方案之间的类比并不是没有可比性的。

工业革命

工业革命，就是从修正性的手工作坊生产转变到利用可交换部件自动化地大量生产，这个转变花费了大约 200 年。该转变过程有 3 个重要里程碑：可交换部件的引入、装配线的引入和自动化装配线的引入（参见图 1-1）。

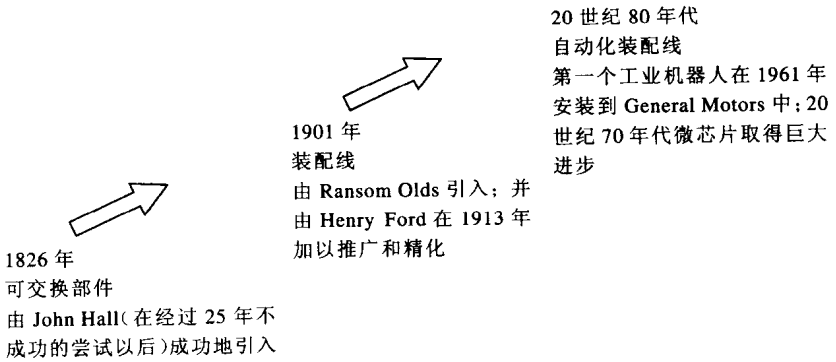


图 1-1　工业革命的重要里程碑

可交换部件的引入

Allan Willey 如下描述在制造业中可交换部件的引入（解释于[Wil97]）：虽然我们今天认为可交换部件的概念司空见惯，但是在 19 世纪开始的时候，没有任何制造出的商品是利用可交换部件生产的。故事开始于 1785 年，当时任驻法国大使的 Thomas Jefferson 参观了 Honore Blanc——一个法国机械师的商店，Honore Blanc 正在制造步枪，使用的是手工打造的部件，精度能够使它们互相更换。根据直觉，Jefferson 立即开始认识到这个思路的价值。后来，在 1798 年担任美国副总统期间，Jefferson 在推动一个与 Eli Whitney 的政府合同中扮演了一个重要角色，这个合同要求在 1800 年 9

月 30 日以前，使用可交换部件的概念生产 4000 支步枪。Whitney 的计划有 3 个原则：（1）水力机械的使用；（2）生产统一的或者可交换部件；（3）由不具备技巧但是“可靠，冷静”的人来运行。Whitney 没有成功地实现这一计划。他在 1809 年交付了最后一套步枪，比预期进度落后了 9 年。没有一支步枪包含了可交换部件，并且政府花费的成本比开始的合同上的数目大了很多倍。但是，他的使用可重用部件生产小型武器的想法启发了其他的几个竞争者。特别值得一提的是 Roswell Lee 和 James Stubblefield，他们分别管理两家归政府所有的军工厂，这两家工厂分别位于 Springfield 和 Harper 的渡口。他们两个保持着长期的思想交流、互相访问和性能评估等等。他们两个也都与外部的承包人一起工作，以促进统一性。最后的成功是由 John Hall 取得的，他在位于 Harper 的渡口的工厂工作。Hall 在 1822 年开始了一个完全独立的操作，经过 4 年以后，第一次做出制造机器，然后生产出了来福枪。所以，使用真正的可交换部件生产出的第一件商品是来福枪，时间是 1826 年，在 Eli Whitney 去世一年以后。但是，仍然又花了 20 多年，这个概念才扩展到整个小型武器工业。在 1834 年，Simeon North 开始生产可以与 Hall 生产的部件交换的部件。最后，在 1845 年，有几个国家军工厂和私人承包者利用可交换部件生产出来了 1841 型步枪和来福枪。

在汽车工业中引入装配线

像今天存在的这种形式的汽车，出现于 19 世纪 80 年代（参见[Mur99]）。第一个轻量级、高速汽油发动机由 Gottlieb Daimler 成功地制造出来，而 Karl Benz 在 1885 年，将汽油发动机应用于一种 3 轮框架，创造了第一辆真正的汽车。但是，汽车的生产过程仍然是缓慢和繁杂的，而且，一般认为汽车既不容易得到，对大多数人来说也不必要。这种情况在 1901 年发生了变化，当时 Ransom Olds 采用了一种装配线方法，大量生产汽车——与一个世纪以前，Eli Whitney 首创的生产步枪的方法思路相同。1908 年，当 Henry Ford 开始使用装配线生产著名的 T 型车时，这种方法取得了巨大的成功。这个方法使汽车可以让大多数人负担得起：T 型车的价格自从它在 1908 年引入以来，很少发生变化——它一直与美国的平均年收入保持一致[1]。在美国，汽车的数量在 1910 年～1920 年之间，从 458000 辆增长到了 800 万辆，并在 1929 年达到了 2700 万辆，这时，General Motors 夺走了 Ford 在市场中的领先地位[Pak96]。为了满足日益增长的对所有各型汽车的需求，Ford 在 1913 年，通过向生产汽车车体和发动机的生产线中引入输送带，极大地增加了生产力。这个方法使产量增长了 3 倍还多[Enc98]。虽然 Ford 不是首创也不是第一个采用装配线原则的人，但他还是被公认为装配线原则被普遍接受和随后在美国工业中大大扩展的主要贡献者。

装配线的自动化

Richard Jarrett 如下描述在制造业中引入机器人的迫切要求（解释于[Jar90]）：第一个工业机器人由一个称为 Unimation 的公司在 1961 年安装于 New jersey 的 General Motors 工厂。第一个机器人相当不灵活，效率也很低。它们生产出大量的剪报，但是没有收入，而且 Unination 直到 1975 年以前，一直不能显示盈利。工业机器人的繁荣发生在 20 世纪 80 年代。这主要归功于微芯片的引入，它成了机器人的“大脑”。另一些因素是美国和日本之间的竞争、工资上涨和工人的成本增加。在

1 根据[Pak96]，1912 年 T 型车的价格是 600 美元，在当时美国的平均年收入是 592 美元。在 1916 年，价格是 360 美元，平均年收入是 708 美元。在 1924 年，价格是 290 美元，平均年收入是 1303 美元。在 1996 年，新型 Ford Taurus 的起价是 21000 美元，平均年收入是 26000 美元。

1984年末，大约在全世界有10万个机器人。今天，对自动化的挑战是如何使机器人在小规模到中等规模的生产中也有利可图，在这个领域，机器人仍然要与传统的生产技术竞争一段相当长的时间。

1.2 产生式编程

❑ 产生式编程的定义

产生式编程（Generative Programming，GP）是一种软件工程范型（paradigm），基础是对系统族建模。就是说，给定一个特定的需求说明书（specification），就可以根据要求制作出一个高度定制、优化的中间产品或者最终产品。这需要使用基本的、可重用实现组件通过配制知识的方式实现。

❑ 产生式领域模型的基本组成

产生式编程目标集中于系统族，而不是一种一个的系统（one-of-a-kind system）。不是从头构造一个单独的系统族成员，而是基于一个通用的产生式领域模型（generative domain model）（参见第5章）。就是说系统族模型具有下面3个基本组成部分：指定系统族成员的方法；可以组装出每一个成员的实现组件；在成员说明书和一个已有成员之间的配置知识（configuration knowledge）映射关系。在订购一辆汽车时，你会面临相似的设置：有一个订购汽车的系统，有从中可以组装成汽车的组件，还有怎样组装与订单对应的汽车的配置知识。用来指定家族成员的术语被称为问题空间（problem space），而使用可能的配置的实现组件组成了解空间（solution space）（参见图1-2）。

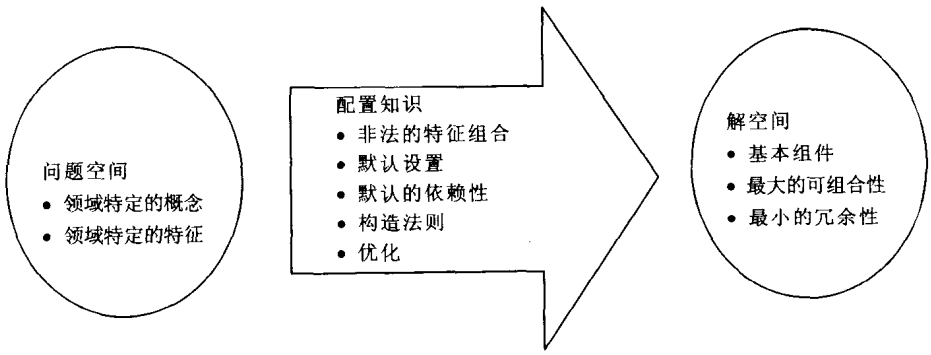


图1-2 产生式领域模型的组成

❑ 实现配置知识

配置知识指定了：

✧ 非法的系统特征组合（例如，你不能指定带有遮阳顶的篷式汽车）。

✧ 默认设置（例如，如果你没有指定一个遮阳顶，就假设没有）。

✧ 默认的依赖性（例如，一辆使用直流发动机的电动汽车不需要换档转换器）。

✧ 优化（例如，调整发动机以达到最优性能）。

✧ 构造知识指定组件的哪一个配置满足特征的哪一个配置。

❑ 捕捉制作知识（production knowledge）

在汽车制作工业中，有相当大一部分必要的配置知识嵌入到了自动装配线里。类似地，在产生

式编程中，我们需要用程序的形式捕捉配置知识。

当前软件工程的问题是，通常我们最后得到了一个具体的软件系统，但是并不知道我们是怎样得到它的。大多数设计知识（design knowledge）丢失了，这就使软件难以维护和演化，并且进行维护和演化的成本很高。在产生式编程中，我们尽可能多地以程序形式捕捉制作知识。制作知识不仅包括配置知识，还包括测量工具、测试策略和计划、错误诊断、调试支持以及程序可视化等等。这些不同的方面是特定领域的，并且被打包成可重用的库，我们称之为动态库（active library）[CEG+98]。

- *动态库和可扩展编程环境*

动态库封装了通用的和领域特定的可重用抽象，以及在程序员使用它们的时候，提供支持的代码。它们使用用于程序可视化、调试、错误诊断和报告、优化、代码生成、规定版本等等的抽象特定的代码扩展了编程环境。基于动态库扩展编程环境的一个例子如在第 11 章描述的意图编程系统（intentional programming system）。意图编程的一个重要思想就是用语言抽象的动态库代替固定编程语言。固定编程语言（例如 C++或者 Java）强迫我们使用一组某种固定的语言抽象，而动态库却允许我们使用一组为手上的问题而优化配置的抽象。它们使我们可以提供真正的多范型和领域特定的编程支持。

- *为重用而开发以及使用重用的开发*

框架和组件现在被认为是获取软件重用的最有效的技术。不幸的是，没有一种面向对象（OO）的分析和设计方法支持它们的开发。另一方面，产生式编程包含了两种重要的完整开发循环：一种是设计和实现一个产生式领域模型（为重用而开发）；另一种是使用产生式模型制作具体的系统（使用重用的开发）。两个过程都与开发一种一个的系统过程不同，例如统一过程（参见第 3 章）。为重用而开发的范围是系统族，而不是一个单独的系统。使用重用的开发必须被仔细设计，从而以一种系统化的方法利用可重用资源。

- *确定一个系统族的范围*

为重用而进行的开发循环最重要的一个属性就是它的目标集中于系统族（参见第 2 章）。它的第 1 个步骤包括确定感兴趣的家族范围，就是说，决定应该包括哪个特征，不应该包括哪个特征。这就要求分析项目风险承担者（project stakeholder）和他们的目标、当前和潜在的市场、技术预测等等。家族或者领域范围界定是非常重要的，以避免专案产生。专案产生（ad hoc generalization）指的是特征和差异点被丢掉，而不必要的特征和差异点却被引入，从而引起了额外的开发和维护成本。

- *特征建模*

开发循环的下一步是确定家族成员的共同特征和差异特征，以及差异特征之间的依赖。使用特征模型（feature model）记录分析的结果（参见第 4 章）。特征建模比起其他的建模方法[例如统一建模语言（Unified Modeling Language，UML）]有几个优点。首先，特征模型以明确的方式表示差异点（variation point）和它们之间的依赖性。这种表示提供了取得一个系统族的实现组件类别的基础，并提供了指定家族成员和配置知识的方法。第二，特征建模区分在一个家族成员内部和家族成员之间的差异。凭借这种方法，我们避免了导致“胖”应用程序的“胖”组件或者框架。这个问题对于当前的框架和组件技术是很常见的，其中用以实现应用程序内部差异性的机制（例

如，动态多态）也用以实现应用程序之间的差异。最后，特征模型提供了一种与实现无关的表示差异性的方法，这就使你可以在分析模型之外做出关于差异性机制的决策。这与当前的 OO 方法是不同的，例如 UML；当你画 UML 类图的时候，你必须决定是否使用继承、聚集、类的参数化或者其他的某些实现机制以表示一个差异点。

- 系统族模型的设计和实现

基于特征模型，我们为系统族设计了一个产生式领域模型，该模型包括指定家族成员的方式、一个通用架构（包括实现组件的类别）以及配置知识，这种配置知识把一个成员的规范映射成如何装配实现它的实现组件。最后，我们使用合适的基于组件的技术和产生式技术实现模型。

- 实现组件的概念
- 泛型编程

如前面所强调的那样，家族成员由基本实现组件装配而成。这些组件被设计出来以减少代码重复，并且可以以很多种方式组合，从而增加重用性。这些原则也是泛型编程（generic programming）的核心（参见第 6 章和第 7 章），一个出色的例子就是标准模板库（STL）（参见 6.10 节）。产生式编程超越了泛型编程，它不仅仅捕捉一个系统族的基本构件块，而且还要捕捉配置知识，这种配置知识允许我们根据高层次的规范自动地产生家族成员。例如，你可以指定你需要一种矩阵类型，来保存下三角形矩阵，这种类型为速度和进行绑定检查做了优化。用一个产生器为一组矩阵实现组件实现配置知识，这个产生器将自动装配出希望的矩阵类型，并且你将不需要担心是用什么容器、适配器和算法组装出矩阵类型。换句话说，泛型编程代表了组织一个产生式领域模型解空间的一种重要方法（参见图 1-2），但是这只是产生式领域模型的 1/3。

组 件

现在，在软件工业中，软件组件的想法非常普遍。我们把软件组件简单地定义成构件块，从中可以组装成不同的软件系统。就像前面讨论的那样，我们希望将它们设计成插件兼容式（plug-compatible），并且可以以尽可能多的方式组合。我们需要使代码的冗余最少，而重用性达到最大。这些以及其他的属性决定了组件的质量，但是，一般而言，却不是组件成为一个组件的必要或充分条件。

一个组件总是一个定义良好的制作过程的一部分。例如，一块砖是构建房屋过程中的一个组件，却不是汽车的组件。经常引用的标准，例如，二进制形式、互操作性、语言无关性等等，总是与制作过程相关的。例如，如果你需要 C++中的容器，那么 STL 组件，它是源码级的，就很适于使用。如果你需要构造图形用户界面（GUI），那么需要可视化组件（例如 JavaBean）。如果你需要与语言无关的、分布式组件，则可以使用 CORBA（Common Object Request Broker Architecture，通用对象请求代理架构）技术。

试图为软件组件提出一个正规定义，不仅没用，而且有害。我们有对象的正规定义。一个正规定义意味着我们能够提供单独一组必要和充分的属性来定义一个对象：对象具有标识符、状态和行为。[Boo84]

组件的概念有一个完全不同的性质。它是自然概念的一个例子，而不是一个人工的、构造的概念。根据概念的理论（参见附录 A），我们能够轻易地使用正规定义——“对象”来构造概念，并且许多数学概念都是构造的、正规定义的概念。但是，大多数自然概念没有一个正规定义，就是说，