科学通报 2014 年 第 59 卷 第 1 期: 21 ~ 42
评 述 www.scichina.com csb.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS

面向特征的软件复用技术——发展与现状

张伟[①②], 梅宏[①②*]

① 高可信软件技术教育部重点实验室(北京大学), 北京 100871;
② 北京大学信息科学技术学院软件研究所, 北京 100871
* 联系人, E-mail: meih@pku.edu.cn

2013-03-25 收稿, 2013-07-10 接受
国家重点基础研究发展计划(2009CB320701)和国家自然科学基金(61121063, 61272162)资助

摘要 传统产业的工业化生产主要关注如何实现物理产品的大规模复制性生产. 实现软件产业的工业化生产, 达到其发展所需的质量和生产效率, 软件复用是一条现实可行的途径. 软件复用的主要关注点是如何实现软件产品的大规模定制性生产. 面向特征的软件复用提供了一种实现软件大规模定制性生产的技术途径, 对软件复用的研究和实践产生了重要影响. 本文介绍并分析了面向特征的软件复用涉及的基本概念和核心思想, 总结了近年来关于面向特征的软件复用研究的一些关键技术和重要成果. 具体而言: 从内涵和外延两个方面总结了现有研究中关于特征的定义, 分析了特征在软件复用中承担的基本责任; 阐述了特征模型的基本结构及其命题逻辑语义, 介绍了对特征模型基本结构和语义的 3 种扩展; 总结了在特征模型与软件实现制品之间建立追踪关系的两种基本方式, 分别针对用况模型、软件体系结构模型以及软件实现代码这 3 种软件实现制品, 对在特征模型与这 3 者之间建立追踪关系的重要研究成果进行了介绍和评述; 指出了值得进一步关注的问题及未来研究方向. 通过介绍和评述, 期望能够进一步凝聚本领域的研究者对相关问题及研究现状的理解, 进而在面向特征的软件复用的研究上取得新的进展.

关键词
特征
特征模型
追踪关系
领域工程
软件复用
软件工程

软件工程的主要研究问题是如何不断提高软件的质量和开发效率, 提升软件产业的工业化生产水平. 与传统产业相比, 软件产业的工业化生产存在其独有的特点: 传统产业实现工业化生产的主要途径是对同一物理产品的大规模复制性生产; 而对于软件这样一种数字化产品, 对其进行大规模复制不存在任何困难, 其工业化生产的主要技术困难在于如何高效和高质量地构造出满足用户/客户多样性需求的软件. 软件需求的多样性体现在时间和空间两个维度: 在空间维度上, 不同的客户/用户对同一类型的软件产品会存在不同的个性化需求; 在时间维度上, 同一客户/用户对同一软件产品的需求会随着时间的推移而发生演化. 不可否认, 在传统产业中, 客户/用户对物理产品的需求也会存在时间和空间维度上的多样性, 但其多样性的复杂性程度远不及软件产品. 一方面, 物理产品的构成部件在加工成型后的重塑和重组需要较高的成本, 而软件产品的数字化表现形式使得人们认为对软件产品的修改似乎不存在成本; 因此, 人们更愿意对软件产品提出个性化需求. 另一方面, 物理产品一般运行在相对固定的物理环境中, 且物理产品与其环境的交互关系也相对固定, 而软件产品的操作环境除了包括相对固定的物理环境外, 还包括不断变化发展的数字化环境(如在这个数字化环境中会不断出现新的软件实体、新的通讯协议、新的软件运行平台等), 且软件产品往往与其所在的数字化环境存在复杂的交互关系; 因此, 软件产品需求多样性(不仅涉及软件的基本功能, 还包括软件对其数字化环境的兼容性、互操作性、及适应

引用格式: 张伟, 梅宏. 面向特征的软件复用技术——发展与现状. 科学通报, 2014, 59: 21–42
Zhang W, Mei H. Feature-oriented software reuse technology—state of the art (in Chinese). Chin Sci Bull (Chin Ver), 2014, 59: 21–42, doi: 10.1360/972013-341

性等方面)的复杂程度远大于物理产品的需求多样性. 随着软件应用范围的不断扩展、应用程度的不断加深以及应用形态的不断创新, 软件产品的需求多样性也在不断的加剧. 同时, 软件固有的复杂性、一致性、易改变性、不可见性等特点[1]以及软件产品规模的不断增长, 导致了软件生产困难持续增加: 在软件开发方法和技术不断发展的现实情况下, 软件的质量和开发效率仍然无法满足软件产业发展的客观需要.

要持续提升软件的工业化生产水平, 达到软件产业发展所需要的质量和生产效率, 软件复用是一条现实可行的途径. 软件复用作为避免重复劳动、提高软件质量和生产效率的解决方案, 其出发点是软件产品的开发不再采用"从零开始"的模式, 而是以已有的工作为基础, 充分利用过去软件开发中积累的知识和经验(如源代码、设计方案、测试用例以及需求规约等), 将开发的重点集中于软件应用的特有成分[2]. 通过软件复用实现软件的工业化生产, 其背后存在两点基本假设[3]: (1) 不同的软件产品之间存在共性成分; (2) 不同软件产品之间的共性成分具有复用价值(即, 对其进行复用具有可获利性). 针对这两点基本假设, 研究者提出了"程序家族" (program family)[4]、"领域"(domain)[5]或"软件产品线" (software product line)[6]的概念, 用来指代一个软件产品集合, 其元素之间存在具有复用价值的共性成分. 通过软件复用实现软件的工业化生产, 其主要技术困难是如何实现对一类相似软件产品的大规模定制性生产: 给定用户/客户关于软件产品的需求, 通过对该产品所在领域的可复用成分的定制性复用(即, 在领域的所有可复用成分中, 定位到适用于当前软件产品的可复用成分, 并对其进行复用), 高效率和高质量地生产出满足需求的软件产品.

实现面向领域的软件产品的大规模定制性生产, 涉及一个重要的问题: 如何对领域中的可复用成分进行有效的管理. 一般而言, 在软件开发的不同阶段, 通常会产生不同的实现制品(如需求阶段产生需求模型、设计阶段产生软件体系结构模型、实现阶段产生软件源代码等). 一项领域可复用成分, 通过软件开发活动, 其实现元素最终会体现在一组由不同开发阶段产生的不同实现制品中. 同时, 一个实现制品, 可能会涉及多项领域可复用成分, 即多项领域可复用成分对应的实现元素在同一个实现制品中相互交织. 对一项领域可复用成分的复用可以分为两个层次进行: 首先是在概念层次上意识到一项可复用成分的存在并对其进行复用, 然后是在实现层次上对该项可复用成分对应的实现元素进行复用. 概念层复用是实现层复用的前提: 没有在概念层次上意识到一项可复用成分的存在, 则很难进一步对其实现元素进行复用. 实现层复用则是对概念层复用的深化: 对实现元素的复用, 进一步放大了概念复用的效果. 基于上述分析, 对领域可复用成分的有效管理, 从技术上可以进一步落实为两个方面. 一方面, 如何保证可复用成分的易定制性, 即能够容易地将领域可复用成分的任一有效子集从其中定位并剥离. 可复用成分的易定制性覆盖两个层次: 在概念层次上, 如何对领域可复用成分进行建模, 并保证建模结果的易定制性; 在实现层次上, 如何保证领域可复用成分涉及的实现制品具有易定制性. 另一方面, 如何保证可复用成分的易追踪性, 即: 能够方便且准确地追踪到一项领域可复用成分对应的、分散在多个实现制品中的实现元素. 图 1 对可复用成分有效管理的两个方面进行了进一步的明确. 由于领域本身所具有的内聚性, 领域中的不同可复用成分之间并不相互独立, 而存在复杂的依赖关系; 因此, 在概念层次上, 需要对可复用成分及其之间的依赖关系进行有效地建模, 从而保证可复用成分的易定制性. 而在实现层次上, 在对实现制品进行定制时, 需要考虑两个方面的因素: (1) 定制结果必须满足该制品对应的元模型所具有的各种约束; (2) 定制结果中实现元素之间的关系必须满足在概念层中定义的可复用成分之间固有的依赖关系. 同时, 在建立概念层元素和实现层元素之间的追踪关系时, 需要采用一种能够有效包容或屏蔽不同类型实现制品(具有不同的元模型)差异性的追踪关系建模机制.

面向特征的软件复用提供了一种面向领域的软

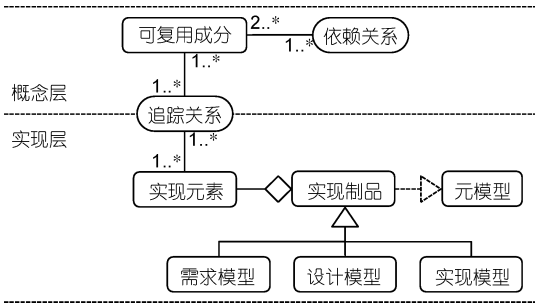


图 1 可复用成分的概念、实现及其关系

件产品大规模定制性生产的技术途径. 其基本思想是将软件的特征作为管理领域可复用成分的基本单元, 通过对领域中软件产品的特征、特征之间的关系、特征与实现制品之间的追踪关系进行系统地建模, 实现对领域可复用成分的有效管理. 在此基础上, 用户/客户根据其具体的业务环境, 通过对一个领域的特征集合进行定制, 产生一个能够有效满足其业务需求的特征集合, 从而明确其对软件产品的需求(概念层复用); 然后, 通过特征与其实现制品之间的追踪关系, 定位到相应的实现制品, 从而对其进行复用(实现层复用).

本文主要从技术的角度对面向特征的软件复用中领域可复用成分管理相关的重要研究进展和成果进行综述. 对相关研究工作的组织按照"如何保证可复用成分的易定制性"和"如何保证可复用成分的易追踪性"这两个研究问题来进行. 在面向特征的软件复用中, 易定制性问题主要针对概念层可复用成分的易定制性, 其解决方案体现在特征和特征模型这两个概念中; 在此基础上, 易追踪性问题则具体化为如何在特征模型和软件实现制品之间建立追踪关系; 而对于实现层可复用成分的易定制性问题, 则通过上述两个问题的解决方案的组合来间接解决, 即保证概念层可复用成分的易定制性, 并通过追踪关系实现易定制性从概念层向实现层可复用成分的传播. 面向特征的软件复用, 其基本思想自 1990 年被提出[7]以来, 已经过二十几年的探索、研究和实践. 从研究的角度来看, 其发展过程大致可以分为两个阶段: 2005 年以前, 其研究主要关注于特征的定义以及特征模型的结构和语义, 取得了一些目前看来已成为共识的研究成果; 2005 年以后, 其研究一方面关注如何对传统特征模型的结构和语义进行扩展, 另一方面则主要关注特征模型与软件实现制品之间的追踪关系.

1 基础知识

1.1 特征

在一般意义上, 特征是一个事物所展现出的具有区分作用的特点. 根据其区分作用的不同, 特征可以划分为两类: 一类称为领域特征, 刻画的是一个特定类别/领域内的所有成员都具有的共性, 其区分作为体现为将一个事物所属的类别/领域与其他类别/领域区分开; 另一类称为个体特征, 刻画的是特定类别/领域内的部分成员具有的个性特征, 其区分作用体现为将同一类别/领域内的不同成员区分开. 一个特征是领域特征或是个体特征, 依赖于类别/领域的边界[10]: 当类别/领域的边界发生变化时, 一个领域特征可能会变为个体特征, 反之亦然. 基于特征的概念, 对一个事物的刻画可以通过一组特征来实现. 基于特征的两类区分作用, 对一个事物的刻画可以分为两个阶段: 首先, 通过一组领域特征刻画出该事物所属的类别/领域; 然后, 通过一组个体特征将该事物与同一类别/领域内的其他事物区分开.

在面向特征的软件复用中, 对特征这个概念的理解和使用, 与其一般意义是相符的: 使用一组特征对一个领域的所有软件产品的共性和个性特点进行建模, 并通过对该领域对应的特征集合的定制定位到领域中特定的一个或一组软件产品. 从发展历史来看, 在软件领域中, 1982 年即有学者[11]指出"使用客户可感知的特征对软件需求进行组织是一种非常自然的手段". 同时, 特征的概念在电信系统中得到了广泛的使用[12], 被用来指代电信服务提供商对用户提供的一系列标准服务(如呼叫转移、呼叫等待等). 1990 年, 美国卡耐基·梅隆大学软件工程研究所的学者提出了面向特征的领域分析方法[7](feature-oriented domain analysis, FODA), 使用特征模型(特征及其之间的依赖关系)来组织领域中具有复用价值的软件需求. 在随后的关于软件开发[13~15]和软件复用[6,16~19]的研究中, 面向特征的思想得到了广泛的认同和使用.

特征这个概念在软件领域中的定义, 体现出两类不同的视角. 在软件复用的研究中, 研究者大多是从外延的角度对特征进行定义, 认为特征是一种软件特点. 例如, 面向特征的领域分析方法 FODA[7]认为特征是"软件系统具有的一种显著的或具有区分作用的、且用户可见等特点"; 另一种软件复用方法 FeatuRSEB[17]则认为特征是"软件产品线中的产品具有的特点, 对于用户和客户对产品线中不同产品的描述和区分具有重要作用". 在一般性的软件工程的研究中, 研究者则更多从内涵的角度对特征进行定义, 认为特征是一组软件需求. 例如, 在一项系统性探讨特征在软件工程中的作用的研究中[13], 特征被定义为"一个功能性或非功能性需求的集合"; 在一项关于软件再工程的研究中[15], 特征被认为是"一组单个需求, 描述了一个与软件生命周期中特定视角

相关的功能单元"; 需求工程领域的学者 Wiegers[20]认为特征是"一组逻辑相关的功能性需求构成的集合, 为用户提供了一种满足特定业务需求的能力". 另外, IEEE 软件工程术语词典[21]将特征定义为"一个被软件需求文档明确或隐含说明的软件特点", 这个定义既反映了特征的外延(一个软件特点), 又在一定程度上体现了特征的内涵(这个软件特点是被软件需求所说明的). 将上述两种视角融合在一起, 即可得到目前的软件研究领域对特征这个概念的共识性理解: 就内涵而言, 特征是由一组相对紧密关联的单个需求构成的集合; 就外延而言, 特征是一种具有用户或客户价值的软件特点[22]. 由特征的内涵可知, 特征本质上是对一组需求的分割和概念化, 即, 把一个完整需求集合分割为一组相对独立的需求子集, 并为每个需求子集进行命名以对其进行概念化并方便对其引用. 特征的外延则提供了评判一个需求子集是否可被认为是一个特征的标准, 即, 这个需求子集是否描述了一种具有充分的用户或客户价值的软件特点.

在面向特征的软件复用中, 特征提供了一种对领域可复用成分进行建模和管理的基本单元. 一般而言, 一个软件产品与两个空间相关. 一个空间称为问题空间, 包含软件产品需要解决的问题, 以及为了解决特定问题的软件需要提供的能力(即对软件的需求). 问题空间在本质上是面向软件的用户和客户的, 虽然问题空间是软件开发者进行软件开发的依据, 但软件开发者更关心如何将问题空间的元素变换到具体的实现方案和细节. 问题空间的成分在软件开发活动中会被软件需求规约文档这种制品所体现. 另一个空间则称为实现空间, 包含了满足特 定需求的软件的具体实现方案和细节. 实现空间在本质上是面向软件开发者的, 软件的用户和客户并不关心软件的具体实现方案和细节. 实现空间的成分在软件开发活动中会通过软件设计、编码、测试等活动被软件体系结构模型、源代码、测试用例等制品所体现. 一个领域的可复用成分涉及到这两个空间中相关的软件制品, 以及不同软件制品之间的追踪关系. 为了实现对可复用成分的有效管理, 需要保证可复用成分的易定制性和易追踪性. 其中涉及的一个基本问题是对可复用成分进行建模和管理的基本单元是什么. 如果这种单元的粒度过大, 则无法保证能够准确地对可复用成分进行定制性复用; 如果这种单元的粒度过小, 虽然能保证可复用成分的可定制性, 但却会导致定制和追踪过程的繁琐性. 面向特征的软件复用认为, 对领域可复用成分进行建模和管理的基本单元不能以实现空间为出发点, 而应该以问题空间为出发点, 即站在用户/客户的角度上, 确定领域可复用成分的基本单元是什么; 实现空间中制品是对问题空间中制品的一种实现方式, 其本身具有不稳定性(技术的演化会导致实现方案及其制品发生相应的改变); 以实现空间中的特定制品为出发对可复用成分进行建模和管理, 只看到了可复用成分的表象, 而没有抓住可复用成分的本质. 进一步而言, 面向特征的软件复用认为, 从用户/客户所能感知的软件对外表现的具有价值的特点, 是对可复用成分进行建模和管理的一种合适的基本单元.

特征提供了一种对软件需求进行分割的方式. 在软件开发中, 一种得到广泛应用的需求分割方式是用况(use case)[23,24]. 但由于用况所指代的需求集合粒度的僵化性, 它并不适合作为建模和管理领域可复用成分的基本单元[17]. 用况是对软件用户使用软件的一项功能时所进行的交互过程的描述. 如果使用用况作为分割可复用成分的基本单元, 那么, 任何一项可复用成分都必须体现为一种使用某项软件功能时的完整交互过程. 显然, 一项可复用成分不一定体现为一个完整的用况, 而仅可能和用况的一个片段相关. 进一步而言, 即使将用况分解为一组更细粒度的构成成分(例如, 用况中包含的活动、以及活动之间的时序关系), 这些更细粒度的构成成分也不适合作为建模和管理领域可复用成分的基本单元. 主要原因在于两点: (1) 对于用况的构成成分之一——"活动", 其粒度虽然比用况小, 但仍然存在粒度的僵化性问题(一项可复用成分可能仅体现为某个活动的某些成分, 或体现为一组活动中的若干片段的组合); (2) 由于用况具有的操作型语义, 使其包含了太多的交互细节, 使得建模产生的可复用成分缺乏必要的抽象性和易理解性. 例如, 图 2 展示了分别采用用况和特征两种方式对用户登录这样一个软件功能进行建模的结果. 图 2(a)的用况建模结果展示了用户登录系统的完整交互流程: 一方面, 可以把整个交互流程看作一项可复用成分; 另一方面, 这个交互流程还包含了更细粒度的若干项可复用成分(比如, 涉及到一个活动的片段的"账户号码"、"账户密码"、"验证码"、以及"当用户输入账户密码时, 以星号代替客户输入"(见图中被下划线标识的文字); 涉及到多个活动

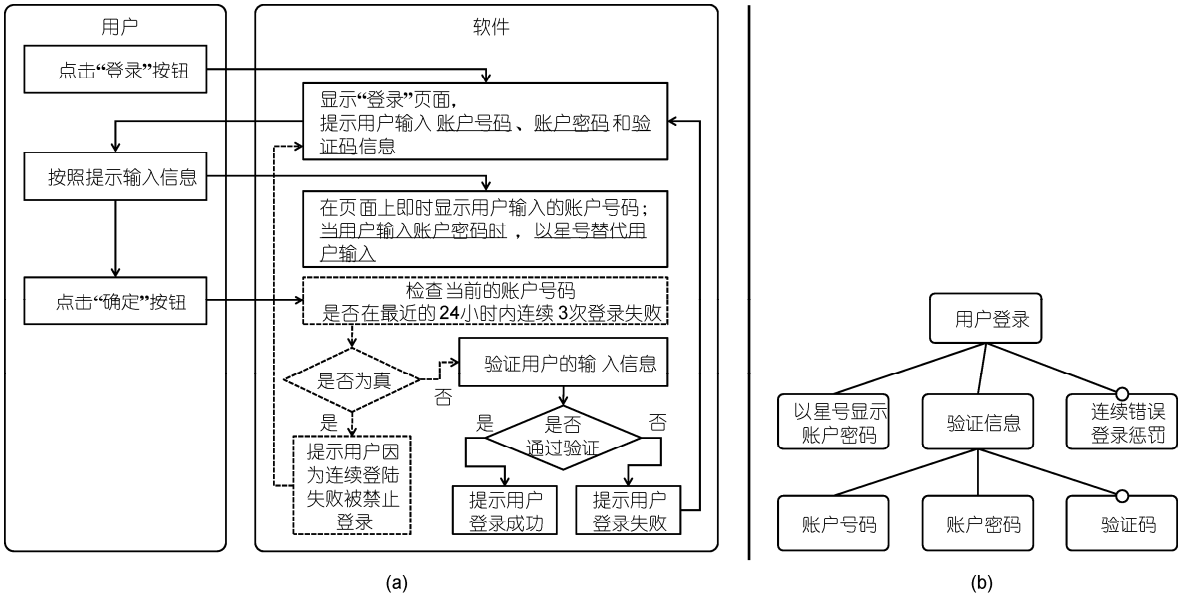


图 2　面向用况与面向特征的建模方式对比示例

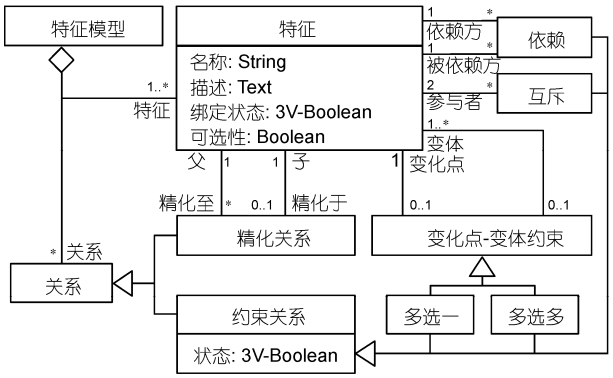
(a) 用户登录的用况; (b) 用户登录的特征模型

及其时序关系的“连续错误登录惩罚”(见图中被虚线标识的成分). 图 2(b)展示了用特征建模相同的可复用成分的结果: 每项可复用成分被建模为一个特征; 通过特征之间的关系将一组特征组织在一起. 对比这两种建模方式, 可以看到面向特征的方式在简洁性、灵活性、易理解性及易扩展性等方面都要优于面向用况的方式.

1.2　特征模型: 基本结构和语义

特征提供了分割需求的一种方式, 但特征之间并非完全相互独立, 而是存在多种类型的关系. 在特征概念的基础上, 为了对特定领域中的软件特征及其之间的关系进行有效的建模和管理, 面向特征的软件复用进一步提出了特征模型的概念. 特征模型在 FODA 方法[7]中被首次提出后, 众多研究者对特征模型的结构和语义进行系统化的研究, 并对特征模型进行了丰富多样的扩展. 本小节主要介绍关于特征模型基本结构和语义已形成共识的一些研究成果; 下一节将进一步介绍对这些共识性研究成果的若干扩展.

图 3 给出了一个特征模型元模型: 这个元模型体现了在当前研究中关于特征模型的基本共识; 大多数对于特征模型的扩展都是在这种共识的基础上进行的. 在高层结构上, 一个特征模型由一组特征以及特征之间的关系构成; 同时, 一个合理的特征模型还必须满足一定的约束条件. 具体而言, 特征模型中的一个特征具有 4 个基本属性: 名称、描述、绑定状态、可选性. 其中, 名称是对特征的概念化; 描述是对特征指代需求的详细说明; 绑定状态刻画了在特征模型的一个定制结果中, 特征可能具有的 3 种状态: 被绑定、被删除、待确定; 可选性描述了当一个特征的父特征(如果存在)处于被绑定状态时, 该特征是否一定需要处于被绑定状态(是, 表示该特征是一个必选特征; 否, 表示该特征是一个可选特征). 特征之间具有两种基本类型的关系: 精化关系、约束关系. 精化关系是特征之间的一种二元关系. 通过精化关系, 不同粒度和抽象层次的特征形成树形结构(即, 一个特征作为父特征, 可以精化为零个、一个或多个子特征; 作为子特征, 则最多具有一个父特征). 约束关系描述了特征的绑定状态之间具有的约束关系. 一个约束关系具有两个基本属性: 公式、状态. 其中, 公式刻画一组特征的绑定状态之间约束关系对应的命题逻辑; 状态记录了约束关系可能具有的 3 种状态: 被满足、被违反、待确定. 特征模型中常见的约束关系包括涉及两个特征的依赖关系和互斥关系, 以及涉及父子特征的多选一和多选多约束关系. 基于上述建模元素, 一个合理的特征模型及其定制结果还应满足 9 种约束条件: 其形式化定义见图 3 下半部分



OCL 约束关系:

context fm: 特征模型
① inv: fm.特征→select(精化于→empty())→size = 1
context f: 特征
② inv: f.精化于→empty()
implies (not f.可选性 and f.绑定状态)
context r: 精化关系
③ inv: r.子.绑定状态 implies r. 父.绑定状态
④ inv: not r.子.可选性implies
(r.父.绑定状态 implies r.子.绑定状态)
context c: 依赖
⑤ inv: c.状态 = (c.依赖方.绑定状态
implies c.被依赖方.绑定状态)
context c: 互斥
⑥ inv: c.状态 = (c.参与者→exists(绑定状态<>true))
context c: 变化点-变体约束
⑦ inv: c.变体 = c.变化点.精化至.子→asSet()
context c: 多选多
⑧ inv: c.状态 = (c.变化点.绑定状态
implies c.变体→exists (绑定状态))
context c: 多选一
⑨ inv: c.状态 = (c.变化点.绑定状态
implies c.变体→select(绑定状态)→size = 1

图 3 特征模型元模型

给出的 9 条 OCL 约束[25]; 其解释见表 1 所述.

当一个针对特定领域的特征模型被建立之后, 即可以通过定制的方式从其中导出所有可能的具有一致性和完整性的特征集合. 一致性是指这个集合中的特征之间不存在冲突; 完整性是指如果一个特征依赖于其他特征, 则这些被依赖的特征也一定存在于这个特征集合中. 保证定制结果的完整性和一致性的充要条件是确保定制结果符合元模型中②③④⑤⑥⑧⑨这 7 种约束条件. 图 4 给出了一个特征模型的图形化示例: (a)是一个面向手机服务领域的特征模型, (b)则展示了这个特征模型一种可能的定制结果.

对于图 3 定义的特征模型, 其语义可被形式化地表示为一组命题逻辑公式[26~28]. 表 2 给出了将特征模型变换为等价的一组命题逻辑公式的 9 条规则.

观察这 9 条规则, 可以看到变换产生的命题逻辑公式或者是一个子句(即, 一组命题变量的析取)或者是若干子句的合取; 这两种类型的命题逻辑公式都是符合合取范式(CNF)的逻辑公式. 因此, 经过变换后的特征模型会表现为一个符合合取范式的命题逻辑公式. 不失一般性, 设这个公式具有 n 个子句 $C_i(i=1,2,\cdots n)$, 则该公式可以表示为 $CST=C_1\wedge C_2\cdots\wedge C_i\cdots\wedge C_n$. 基于这种形式, 一个合理的特征模型或一个合理的特征模型定制结果的充要条件可以统一且简洁地表示为“命题逻辑公式 CST 是可满足的”.

表 1 特征模型元模型中 9 条 OCL 约束的解释

编号	解释
①	一个特征模型中有且只能有一个根特征. 一般而言, 根特征的名称与当前领域的名称是相同的(在有些特征模型中, 允许存在多个根特征; 在这种情况下, 为这多个根特征建立一个新的父特征, 即能满足这种约束)
②	根特征必须是必选特征, 且其始终处于被绑定状态. 这种约束要求特征模型的任一定制结果中至少包含一个根特征; 如果根特征是可选的, 且在定制中被删除, 则会产生一个不包含任何特征的没有意义的定制结果
③	在一个精化关系中, 如果子特征处于被绑定状态, 则父特征也必须处于被绑定状态
④	在一个精化关系中, 如果子特征是必选特征, 且如果父特征处于被绑定状态, 则子特征也必须处于被绑定状态
⑤	在一个依赖约束关系中, 如果依赖方处于被绑定状态, 则被依赖方也必须处于被绑定状态
⑥	一个互斥约束关系中的两个特征最多只能有一个处于被绑定状态
⑦	在一个变化点-变体约束关系中, 任何一个变体特征的父特征必须是变化点特征. 在这个意义上, 变化点-变体约束关系可以看作是关于一个特征与其子特征之间的一种约束关系. 其中, 父特征具有变化点的角色; 子特征具有变体的角色
⑧	在一个多选多约束关系中, 如果变化点特征处于被绑定状态, 则至少有一个变体特征必须处于被绑定状态
⑨	在一个多选一约束关系中, 如果变化点特征处于被绑定状态, 则有且只能有一个变体特征处于被绑定状态

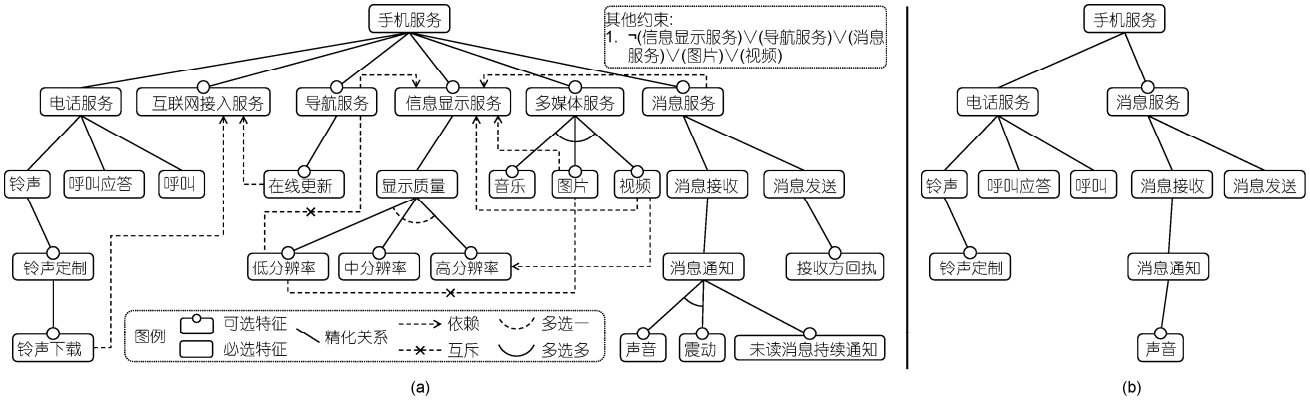


图 4　特征模型示例

(a) 特征模型; (b) 特征模型的一种定制结果

表 2　特征模型到命题逻辑公式的变换规则

规则名称	OCL 约束依据	模型元素	命题逻辑公式
根特征	②	<<Root>> *r*	r
可选特征精化	③	*a* *b*	$\neg b \vee a$
必选特征精化	③ ④	*a* *b*	$(\neg b \vee a) \wedge (\neg a \vee b)$
依赖关系	⑤	*a* ⇢ *b*	$\neg a \vee b$
互斥关系	⑥	*a* ⇢✳⇠ *b*	$\neg a \vee \neg b$
多选多结构	⑧ ③	*p* c_1 c_2 … c_n	$(\neg p \vee c_1 \vee c_2 \cdots \vee c_n) \wedge$
多选一结构	⑨ ③	*p* c_1 c_2 … c_n	$(\neg p \vee c_1 \vee c_2 \cdots \vee c_n) \wedge$ $\wedge$
被绑定的特征	无	*f* 绑定状态=真	f
被删除的特征	无	*f* 绑定状态=假	$\neg f$

标记	含义
f	一个名称为 *f* 的特征: 可以是一个可选特征，或必选特征
命题逻辑公式中的特征名称	该特征的绑定状态

2 重要研究进展

2.1 对特征模型基本结构和语义的几种扩展

(ⅰ) 特征之间的关系. 在传统特征模型中, 特征之间存在两种类型的关系: 精化关系、约束关系. 引入这两种关系的主要目的是对特征进行合理的组织并保证特征模型定制结果的一致性和完整性. 但是, 这两类关系对于如何基于特征模型进行其他可复用软件制品的开发并没有直接的帮助[29]. 针对这个问题, 研究者对特征之间的关系类型进行了扩展.

对特征之间关系的扩展可以从两个方面进行. 一方面, 对现有的特征关系类型进行进一步的分类, 使其能够体现一些与软件实现相关的信息. 例如, FORM 方法[30]将特征之间的精化关系细分为 3 种类型: 分解(decomposition)、特殊化(specialization)、实现(implementation). 这种分解借鉴了面向对象的方法中对象之间的关系类型: 如果把特征作为一种对象, 那么, 对象之间的各种关系类型都可以作用到特征上. 基于这种观点, 在 Zhang 等人[31]的研究工作中, 进一步将对象之间的属性化关系和实例化关系引入到特征模型中. 这种基于对象关系的扩展, 使得特征模型表现了更多的语义信息, 并在特征模型与面向对象软件开发方法中的对象模型之间建立了较为直接的类比关系: 软件开发人员可以将对象模型及其相关的软件开发技术类比应用到基于特征模型的可复用软件实现制品的开发中.

另一方面, 可以根据软件实现的需要, 引入全新的特征关系类型. 例如, 在 Lee 和 Kang[29]的工作中, 引入了一组全新的特征之间的依赖关系类型, 包括: 使用(usage)、修改(modification)、互斥激活(concurrent-activation)、并发激活(sequential-activation)、顺序激活(sequential-activation). 其中, "使用依赖"用于建模一个特征的实现对另一个特征实现的依赖; "修改依赖"用于建模一个特征对另一个特征的行为的改变; 激活类型的依赖则用于建模不同特征在激活时应该满足的限制. 在此基础上, 该项工作还进一步分析了这些依赖关系对设计领域中可复用软件构件的帮助. Zhang 等人[31]认为, 传统特征模型中的精化和约束表现的是特征之间的静态关系, 而软件在开发完成并投入运行后则主要体现出动态的特性. 因此, 如果能在特征模型中显式建模特征之间在运行时刻展示出的动态关系, 则会对软件设计、编码等阶段可复用软件制品的开发带来更为直接的帮助. 基于这种理解, Zhang 等人[31]为特征模型增加了一组特征之间的动态交互关系, 包括通知、资源配置、元层配置和流. 对于两个特征 *A* 和 *B*, "*A* 通知 *B*"表示 *A* 向 *B* 发送了一条信息, 告知其特定事件的发生或特定条件的满足; "*A* 资源配置 *B*"表示 *A* 对 *B* 所依赖的一些资源进行了修改, 从而间接改变 *B* 的行为; "*A* 元层配置 *B*"表示 *A* 对 *B* 的绑定状态进行了改变; "*A* 流 *B*"表示 *A* 和 *B* 之间的顺序执行关系(等价于 Lee 和 Kang[29]工作中顺序激活关系). 进一步, Zhang 等人[31]发现, 特征之间的静态关系和动态交互关系之间并不是相互独立的, 两者可以被看作是从两个不同的角度对特征之间的关系进行的建模; 基于这种观察, 只需要从传统特征模型中的精化和约束这两种静态关系出发, 就能够发现特征之间在运行时刻存在的动态交互关系. 在 Peng 等人[33]的研究中, 则采用本体建模技术, 定义了刻面(facet)、使用/决定(use/decide)、配置依赖(config-depend)等 3 种二元关系, 使得特征模型更够表达更丰富的语义信息.

从提高概念层可复用成分易定制性的角度来看, 上述研究中关于特征关系的扩展, 在本质上并没有增加特征模型的易定制性: 这些扩展没有引入新的表现特征之间约束关系的建模元素. 这些扩展的主要目的是为了缩小概念层可复用成分与其实现之间的差距, 即通过对与实现相关的特征关系的建模, 使得尽可能平滑地从特征模型过渡到对其实现制品的设计与编码活动中. 这种关注点反映了在目前的研究和实践中, 还缺少一种成熟的特征模型驱动的软件开发方法. 考虑面向特征的软件复用的最终目标, 本文认为, 与现有的软件开发方法相比, 这种特征模型驱动的软件开发方法必须充分考虑如何在软件实现制品与特征模型之间建立准确的追踪关系, 以方便对实现制品的有效定制以及对可复用成分实现元素的有效复用.

(ⅱ) 基于基数的特征模型. 保证特征模型具有易定制性的主要手段是显式建模特征之间存在的约束关系. 研究者在实践中发现, 有些类型的约束关系在传统的特征模型中不能得到方便/有效的表达[34]. 为此, 出现了各种关于增强特征模型约束表达能力的研究工作. 其中, 一种得到较多关注的研究是基于基数(cardinality)的特征模型[34~36].

基于基数的特征模型引入了特征基数(feature

cardinality)和组基数(group cardinality)两个概念, 对传统特征模型中的约束关系进行了两个方面的增强和扩展. 特征基数为一个特征关联了一个整数集合, 并用一个或多个整数区间来表示这个整数集合. 例如: 使用区间[5..*]表示了由所有大于等于 5 整数构成的集合; 使用区间[3..3]表示了仅有一个整数 3 的集合; 或将上述两个区间顺序列出[5..*][3..3], 表示上述两个整数集合的并集. 给定一个特征, 其对应的特征基数的语义是指: 在一个合理的特征模型定制结果中, 在这个特征的父特征的所有子特征中, 这个特征的出现次数必须属于其特征基数所表示的整数集合. 因此, 特征基数是对传统特征模型中必选特征和可选特征所表达的约束关系的一种扩展或增强: 对于必选特征, 其特征基数为[1..1], 表示在其父特征的所有子特征中, 该特征必须且只能出现 1 次; 对于可选特征, 其特征基数为[0..1], 表示在其父特征的所有子特征中, 该特征或者不出现, 或者仅出现 1 次; 除了[1..1]和[0..1]这两种特征基数之外, 传统特征模型无法表达其他类型的特征基数. 从另一个角度看, 特征基数在传统特征模型中增加了克隆的语义, 即在定制过程中, 一个特征及其所有子孙特征通过精化关系形成的树形结构可以被多次克隆. 但在 Czarnecki 等人[34,36]关于特征基数的研究中, 没有考虑特征克隆对传统特征模型中命题逻辑型约束关系的影响. 例如, 对于特征 A 和 B, 假设其存在约束关系"A 依赖 B", 当特征 B 被克隆为 $B_1,B_2,\cdots B_n$ 等 n 个特征后, 在特征 A 和克隆特征 B_i (i=1,2,$\cdots n$)之间是否应存在约束关系, 如果是, 应该存在何种形式的约束关系? 针对该问题, 在 Zhang 等人[22]的研究工作中, 提出了两种模式对克隆特征与其他特征之间的约束关系进行建模. 第一个模式称为生成模式, 定义了一个特征的一组克隆体与该特征之间应该保持的若干种可能的约束关系(即, 生成了一组新的约束关系); 第二个模式称为适应模式, 定义了如何对一个特征已经参与的约束关系进行适应性调整, 形成该特征的任一克隆体与其他特征之间的约束关系. 组基数[35]为一个特征的若干个子特征关联了一个整数类型的区间. 给定一个特征的若干子特征, 其对应的组基数的物理含义是指: 在一个合理的特征模型的定制结果中, 如果这个特征被绑定, 那么这组子特征中被绑定特征的数量必须属于组基数所表示的整数集合. 因此, 组基数是对传统特征模型中多选多和多选一类型约束关系的一种扩展或增强: 对于一个多选多约束关系, 当其中的父特征被绑定后, 其对应的一组子特征中必须有 1 个或多个特征被绑定(对应的组基数为[1..*]); 对于一个多选一约束关系, 当其中的父特征被绑定后, 其对应的一组子特征中必须且只能有一个特征被绑定(对应的组基数为[1..1]); 除了[1..*]和[1..1]这两种组基数之外, 传统特征模型无法表达其他类型的组基数.

特征基数和组基数两个概念的引入, 增强了特征模型表达特征之间约束关系的表达能力. 这两个概念的引入在某种程度上借鉴了面向对象技术中类图的"基数"建模元素: 类图中的基数用于表达参与两个类之间的关联关系的对象(类的实例)在数量上的约束关系. 但这两个概念, 特别是特征基数的引入, 也为特征模型语义的清晰性带了一定的负面影响. 其中, 最值得关注的一个问题是我们如何理解"特征被克隆"的语义. 基于把特征作为一种封装需求的基本单元的定位, 该问题可被进一步具体化为"需求被克隆"的语义是什么: 是用于表达一条需求有多个实例, 或是表达一条需求有多种变体, 或说明包含大于等于 2 的基数的特征已经不是传统意义上的特征、或是其他. 目前的研究还缺乏上述问题的明确答案.

(ⅲ) 具有属性的特征模型. 在关于增强特征模型约束表达能力的研究工作中, 除了基于基数的特征模型之外, 具有属性的特征模型是另一种得到较多关注的关于增强特征模型约束表达能力的研究工作[34,37~39].

具有属性的特征模型为传统特征模型中的特征增加了属性的概念. 特征的属性是特征的一个可量化的特点, 每一个特征属性都具有一组可能的取值, 称为该属性的值域. 特征属性的概念最早出现在 Czarnecki 等人[34]关于嵌入式软件特征建模的实例研究中. 随后, Benavides 等人[37,38]提出可以将具有属性的特征模型转化为约束可满足问题(constraint satisfaction problem, 缩写为 CSP), 并对其性质进行自动化分析. 但这些研究工作并没有对特征属性对特征模型中约束关系的影响进行系统性的说明. 针对这个问题, 在 Karatas 等人[39]的研究中, 将具有属性的特征模型的约束关系分为 3 类: 特征-特征之间的约束、特征-属性之间的约束、以及属性-属性之间的约束关系. 其中, 特征-特征之间的约束覆盖了传统特征模型中的约束关系, 即关于特征的绑定状态之间

的各种约束. 对于后两种新引入的约束关系, Karatas 等人通过引入特征属性值之间的比较运算, 将传统特征模型等价的命题逻辑公式中的基本命题扩展为特征属性值之间的关系运算. 例如, 用“内存.容量≥512 M”表示关于内存属性取值的一种关系运算; 用“*F.a*==*X.b*+*Y.c*−60”表示关于特征 *F*, *X*, *Y* 的 3 个属性 *a*, *b*, *c* 之间的一种关系运算. 这种关于特征属性取值的关系运算与关于特征绑定状态的基本命题组合在一起, 则能表达更为丰富的约束关系. 例如, “图形加速器→(内存.容量≥512 M)”表示了这样一种约束: 当图形加速器特征处于绑定状态时, 内存的容量必须大于 512 M.

在某种意义上, 传统特征模型中的特征也具有属性(如名称、描述、可选性、绑定状态等), 但这些属性是在元模型中被预先定义的, 不具有领域特定性, 且无法针对单个特征对其增加新的属性. 具有属性的特征模型为传统特征模型增加了自定义属性的建模能力, 并将传统特征模型的命题逻辑语义进一步扩展为CSP问题. 一方面, 这种扩展增强了特征模型的约束表达能力: 传统特征模型采用“枚举+约束”的方式表达可复用成分的可能组合, 具有属性的特征模型则将枚举扩展为区间(即, 通过区间表达一组元素, 而不必将区间中包含的元素用枚举的方式显式列出); 另一方面, 这种扩展也增加了对特征模型的各种性质进行分析的困难程度: 原来使用 SAT 问题求解软件即能进行的分析工作, 现在则必须使用 CSP 求解软件才有可能进行.

2.2 特征模型与软件实现制品的追踪关系

特征模型提供了一种对特定领域问题空间中的可复用成分进行建模和管理的手段. 通过对特征模型的定制, 可以实现对特征模型所指代的软件需求的有效复用. 但仅仅对特征模型的定制, 并不能产生最终的软件产品. 为了实现软件产品的大规模定制性生产, 需要在特征模型和软件开发不同阶段产生的实现制品之间建立准确的追踪关系, 进而在这种追踪关系的支持下, 从任何一个合理的特征模型定制结果追踪到相应的实现制品, 并通过对这些实现制品的复用产生最终的软件产品.

在抽象意义上, 软件开发的不同阶段产生的实现制品均可被认为是一种模型(例如, 需求阶段产生的需求模型、设计阶段产生的软件体系结构模型、实现阶段产生的代码模型、测试阶段产生的测试用例模型等). 因此, 如何在特征模型和软件实现制品之间建立追踪关系, 可以被抽象为如何在特征模型(源模型)和其他不同类型的模型(目标模型)之间建立追踪关系.

(ⅰ) 特征模型和其他模型的追踪关系: 基于标注的方式. 在软件开发中产生的大多数模型都可以被分解为一组按照某种规则关联在一起的建模元素. 在通过标注的方式建立特征模型和目标模型之间的追踪关系中, 一个合法的特征模型定制结果对应的目标模型的构造被转化对一个全局目标模型的缩减(通过删除其中的某些建模元素). 其关键问题在于如何根据一个定制结果确定全局目标模型中的哪些元素需要被删除. 其解决方案如下[40]: (1) 为全局目标模型中的每一个建模元素标注一个逻辑运算表达式, 称为存在条件; (2) 给定一个合理的特征模型定制结果 *P*、全局目标模型中的一个建模元素 *e* 及其存在条件 *C*, 如果 *P* 使得 *e* 的存在条件 *C* 为真, 则表示从 *P* 能追踪到 *e*(即在 *P* 对应的软件产品中一定存在元素 *e*). 在不同类型的特征模型中, 存在条件可以具有不同的形式. 例如, 在传统的具有命题逻辑语义的特征模型中, 存在条件可以是关于特征绑定状态的一个命题逻辑公式. 最简单的一种存在条件可以是仅包含一个特征的绑定状态的命题逻辑公式. 例如, 给定特征模型中的特征 *f*、以及全局目标模型中的建模元素 *e*, 若 *e* 的存在条件为 *f*, 则表示从特征 *f* 能追踪到元素 *e*(即在任何一个存在 *f* 的特征模型定制结果对应的软件产品中, 一定存在元素 *e*). 在基于基数或具有属性的特征模型中, 则可以针对基数和属性的特点, 定义出具有更丰富语义的存在条件. 根据产生方式的不同, 存在条件可以分为两类: 显式存在条件、隐式存在条件. 显式存在条件是指由建模人员显式指定的存在条件. 隐式存在条件则不需要由建模人员指定, 而是基于目标模型本身具有的合法性约束即能自动产生的存在条件. 例如, 设目标模型是一个类图, 其中存在两个类 *x* 和 *y*、及其之间的一个二元关系 *r*, 给定一个特征模型的定制结果 *P*, 若 *P* 使得 *x* 或 *y* 的存在条件为假. 那么二元关系 *r* 的存在条件显然也应该为假. 在这种情况下, 元素 *r* 的存在条件(即, *x* 和 *y* 的存在条件的合取), 不需要由建模人员显式指定, 而可以在工具的支持下自动生成. 基于对全局目标模型的标注结果, 给定一个合法的特征模型定制

结果, 可以通过两个步骤自动产生其对应的目标模型: (1) 根据特征模型定制结果计算出全局目标模型中每个元素的存在条件的真值; (2) 在全局目标模型中删除所有存在条件为假的元素.

通过标注的方式建立特征模型和目标模型之间的追踪关系, 需要建模人员首先建立一个面向整个特征模型的全局目标模型, 然后再对其中的建模元素进行标注, 最后通过对全局目标模型的缩减产生一个面向特定定制结果的目标模型. 这种方式的一个优点是建模人员可以对全局目标模型包含的建模元素及其之间的关系进行整体性的考虑, 从而在一定程度上能够确保目标模型的质量和正确性. 但同时, 这种方式也带来一个严重的问题: 一个领域的特征模型通常对一组软件产品的需求进行了建模, 因此, 在一个全局目标模型中同时对一组软件产品的实现制品进行建模, 会在很大程度上增加该模型的复杂性, 对其正确性与易理解性带来负面影响, 进而增加了对全局目标模型进行标注的困难. 另外, 基于标注的方式将"全局目标模型的构造"与"特征模型和全局目标模型之间追踪关系的建立"这两个活动孤立看待; 但实际上, 由于全局目标模型仅仅是特征模型的一种实现制品, 在全局目标模型的构造过程中, 就已经隐式地建立了其与特征模型之间的追踪关系; 因此, 一种更自然的方式是在构造全局目标模型的过程中, 增量地将其与特征模型之间的追踪关系显式化.

(ⅱ) 特征模型和其他模型的追踪关系: 基于增量建模的方式. 建立特征模型和目标模型之间追踪关系的另外一种方式称为增量建模(delta modeling)[41,42]. 增量建模将一个合法的特征模型定制结果对应的目标模型的构造转化对一个初始目标模型进行的一系列增量修改. 在增量建模中, 特征模型、目标模型以及两者之间的追踪关系被建模为 5 个部分: 特征模型、共性目标模型、一组增量修改声明、增量修改声明的偏序关系、以及增量修改声明的应用条件. 其中, 特征模型定义了一组合法的特征模型定制结果, 每一个定制结果代表了领域中一个有效软件产品包含的特征集合; 共性目标模型定义了特征模型的共性特征集合(即, 在每个特征模型定制结果中都存在的一组特征)对应的目标模型; 增量修改声明定义了对目标模型的一组修改操作, 体现了对目标模型的增量建模; 增量修改声明的偏序关系定义了两个修改声明应用于目标模型时的顺序关系(对于两个修改声明, 若不存在偏序关系, 则可以任意顺序作用于目标模型; 若存在偏序关系, 则必须按照偏序关系将其应用于目标模型); 增量修改声明的应用条件定义了当前的修改声明在何种特征模型定制结果中有效(相当于基于标注的方式中目标模型元素的存在条件). 基于这样一种建模方式, 给定一个合法的特征模型定制结果, 可以通过下述过程自动产生其对应的目标模型: (1) 根据特征模型定制结果计算每条增量修改声明的应用条件的真值; (2) 对于所有应用条件为真的增量修改声明, 形成一个符合偏序关系的序列; (3) 按照顺序依次将该序列中的增量修改声明应用于目标模型(目标模型的初始值为共性目标模型).

在基于标注的方式中, 给定一个合法的特征模型定制结果, 其对应的目标模型是通过对一个全局的目标模型进行缩减而产生的; 而在基于增量建模的方式中, 一个定制结果对应的目标模型是通过对一个共性目标模型进行一系列的增量扩展而产生的. 与基于标注的方式不同, 基于增量建模的方式将"全局目标模型的构造"与"特征模型和全局目标模型之间追踪关系的建立"统一为一个活动: 构造全局目标模型的过程也是建立其与特征模型的追踪关系的过程; 全局目标模型构造完成后, 其与特征模型的追踪关系也自然形成. 因此, 与基于标注的方式相比, 增量建模的方式更为合理, 效率更高. 同时, 在增量建模的方式中, 建模人员并不需要显式维护一个全局目标模型: 全局目标模型被分解为一个最小规模的共性目标模型和一组相对独立的增量修改声明. 因此, 与基于标注的方式相比, 基于增量建模的方式具有更好的可扩展性. 当特征模型中增加了一个新的特征后, 原有的共性目标模型和增量修改声明无需改变, 而只需要在现有建模结果的基础上增加一组新的修改声明即可. 但另一方面, 基于增量建模的方式面临的一个主要问题是如何确保建模结果的正确性. 由于目标模型中的元素被分散在共性目标模型和一组增量修改声明中, 建模人员缺乏一个全局的视角对建模结果的正确性进行检查和确认. 为了应对这个问题, 需要进一步研究如下问题: (1) 如何确保一组增量修改声明的完整性; (2) 如何检测并消除特征之间可能存在的特征交互问题; (3) 如何确定增量修改声明之间的偏序关系; (4) 给定一个合法的特征模型定制结果、对应的一组增量修改声明、以及由

这组修改声明形成的符合其偏序关系的所有可能的序列, 如何确保应用不同序列产生的目标模型具有唯一性.

下面我们选取在需求阶段、设计阶段、实现阶段产生的用况模型、软件体系结构模型、实现代码, 分别阐述在特征模型与这 3 种目标模型之间建立追踪关系的重要研究进展.

(ⅲ) 特征模型与用况模型之间的追踪关系. Griss 等人[17]提出的 FeatuRSEB 方法将一种以用况为中心的软件复用方法[43]与特征建模方法相结合. 其中, 用况模型被定位为一种面向用户的模型, 刻画了"What of a Domain"; 特征模型被定位为一种面向复用者的模型, 刻画了"Which of a Domain". 对于用况模型和特征模型两者之间的追踪关系, FeatuRSEB 方法认为, 可以先构造一个面向领域的用况模型, 然后将用况及其之间的关系(包括使用关系和扩展关系)按照一定的规则映射为特征及其之间的精化关系. 但该方法提供的映射规则比较粗糙: 一个用况对应一个特征; 两个用况之间的一个使用/扩展关系则对应两个特征之间的一个精化关系(类似的映射规则也出现在 Braganca 等人[44]的工作中). 在本质上, 这种映射规则是通过标注的方式在特征模型和用况模型之间追踪关系的一个简单版本. 其中, 每一个用况的存在条件只与一个特征相关. 另外, 这种规则的一个缺点是其无法在特征和用况中包含的元素之间建立追踪关系.

这种缺点在 Eriksson 等人[18]提出的 PLUSS 方法中得到了改进. 与 FeatuRSEB 方法类似, PLUSS 方法是一种以特征、用况、用况实现为核心制品的领域建模方法. 其中, 特征模型主要被用来管理领域中的用况模型包含的可复用成分. 与 FeatuRSEB 方法不同, PLUSS 方法对用况的建模没有停留在黑盒的层次上, 而是采用了一种基于表格的结构化方式对用况包含的人机交互序列进行建模; 同时, 还采用了参数化的方式对人机交互包含的子元素进行显式化. 基于这种用况建模方式, 该方法可以将用况中的人机交互以及人机交互中包含的参数与特征之间建立追踪关系. 目前该方法仅支持将一个建模元素关联到一个特征上, 但显然, 该方法支持更为复杂的元素存在条件并不是问题. 该方法另一个可以改进的地方在于其对用况的结构化建模机制: 可以对人机交互这种元素进一步分解, 进而将其中外部活动者的动作和软件的动作这两种元素显式化(在理论上, 使用参数化机制也可实现这种目的; 但滥用参数化机制会给领域用况模型增加不必要的复杂性).

Yu 等人[45]的研究则抛弃了基于标注的方式, 而采用基于增量建模的方式在特征模型和用况模型之间建立追踪关系. 其中, 采用流程图对用况进行结构化建模; 在此基础上, 提炼了一组对流程图进行增量修改的语句, 并根据增量建模理论[41,42]定义了一种面向特征模型和用况模型的追踪关系描述语言(支持任何命题逻辑公式形式的增量修改声明应用条件); 同时, 基于特征之间的约束关系, 提出了一组规则用于判断追踪关系建模结果的合法性. 经过必要的扩展, 这种追踪关系描述语言可以支持在特征模型与具有流程图语义的模型之间建立追踪关系.

(ⅳ) 特征模型与软件体系结构模型之间的追踪关系. Hendrickson 等人[46]提出了一种通过变化集(change set)和变化集之间的关系建模面向特定领域的软件体系结构的方法. 其中, 一个变化集包含一组对于目标软件体系结构模型的元素进行添加、删除或修改的操作声明; 变化集之间的关系定义了一组合法的变化集组合(类似于特征模型通过特征之间的约束关系定义了一组合法的特征组合). 具体而言, 该方法把变化集之间的关系划分为 3 种类型: 结构依赖、兼容性、组合. 对于两个变化集而言, 若一个变化集的操作声明中引入的元素依赖于另一个变化集的操作声明中引入的元素, 则前者对后者存在结构依赖. 兼容性刻画了一组变化集在语义上能否同时应用于一个目标体系结构模型. 组合则支持通过对已有的变化集进行组合形成更高层次的概念(例如, 若干变化集可以通过组合形成一个子系统, 甚至形成一个完整的产品). 同时, 该方法还提供了若干种基本类型的关系模式及其变体对这 3 种关系进行规约. 基于对变化集及其之间关系的建模结果, 给定一个合法的变化集组合, 即可以通过将变化集包含的操作声明作用至一个目标模型, 从而产生该变化集组合对应的软件体系结构模型. 但为了应对结构依赖可能形成环的情况, 在如何将一组变化集作用于目标模型的问题上, 该方法并没有采用顺序作用的方式, 而是根据其采用的体系结构元模型的特点, 通过如下的方式将一个变化集组合作用于目标模型: 首先, 按照"先构件、再接口、最后关联"顺序将变化集中的所有添加操作作用于目标模型; 然后, 按照

“先关联、再接口、最后构件”的顺序将变化集中的所有删除操作作用于目标模型. 通过这样的方式, 即使一组结构依赖形成了环, 其涉及的一组变化集也能够可控且一致地作用于目标模型. Hendrickson 等人[46]认为, 变化集从一个更高的层次对软件体系结构进行了刻画, 但他们并没有严格地将这种更高的层次限定为特征层. 从方法论的角度来看, 这种模糊性能够尽可能减少该方法对外部因素的依赖. 但如果将这种更高的层次限定为特征层, 那么, 该方法能够得到进一步的精简, 并能够提供一种通过增量建模在特征模型与体系结构模型之间建立追踪关系的技术手段: 如果把变化集抽象为特征(即, 把一个特征对于软件体系结构的影响建模为一个变化集), 那么, 变化集之间的关系则等价于特征之间的约束关系, 因此, 对变化集之间的关系进行建模的责任则可以完全从该方法中剥离, 而交由特征模型来承担; 在此基础上, 该方法只需要关注如何将一个合法的特征模型定制结果对应的一组变化集(增量)进行融合从而形成相应的软件体系结构模型. 从这个角度观察, 该方法提供了一种能够应对特征之间循环依赖的增量融合机制; 但该机制能否适用于其他种类的目标模型, 还有待确认.

Zhang 等人[31,47]提出了一种特征驱动的软件体系结构构造方法. 一般而言, 特征和构件(软件体系结构模型的基本构成成分之一; 另一种基本成分为构件之间的交互)之间存在多对多的复杂关系(即一个特征的实现通常会散布在多个构件中; 而一个构件中包含的成分则可能与多个特征相关). 基于这种观察, Zhang 等人认为, 为了能够在特征模型与软件体系结构模型之间建立准确的追踪关系, 仅仅停留在这种多对多关系的表象上是不够的, 而必须进一步明确导致这种多对多关系的原因是什么. 在这种思想的指导下, 该方法提出了责任、特征操作化、责任分配等3个概念来解释这种多对多关系的成因. 在内涵上, 责任是一组紧密关联的程序规约; 在外延上, 责任是对程序员进行任务分派的基本单元. 特征操作化是指将一个特征的实现分解为一组责任及其之间的交互. 责任分配, 则是将特征操作化产生的责任分别分配由特定的构件来实现. 基于这 3 个概念, 特征和构件之间多对多关系的成因可以理解为: 由一个特征操作化产生的两个责任被分配到两个不同的构件上. 这种理解具有两个优点: 从静态的视角来看, 这种理解可以将特征与构件之间的多对多复杂关系分解为两组一对多的简单关系(即, 一个特征与多个责任相关; 一个构件与多个责任相关); 从动态的视角来看, 这种理解提供了一种基于特征模型构造软件体系结构的方法(即, 通过特征操作化和责任分配形成构件; 若两个存在交互关系的责任被分配至两个构件, 则这种责任交互关系也形成了构件之间的交互关系; 构件及其之间的交互关系则自然形成了软件体系结构模型). 相比 Hendrickson 等人[46]的软件体系结构模型, 该方法中的软件体系结构模型处于一个更高的层次: 该方法不关注软件体系结构中的构件对外提供的接口以及接口之间的连接关系(语法层次), 而更关注构件本身承担的责任以及构件之间的交互中包含的成分(语义层次). 基于特征操作化和责任分配的建模结果, 给定一个合法的特征模型定制结果, 可以通过将其中每个特征操作化产生的责任分配到相应的构件上, 从而形成相应的软件体系结构模型. 在这种意义上, 这种方法提供了一种通过增量建模在特征模型与体系结构模型之间建立追踪关系的技术手段. 其中的增量体现为一个特征的所有责任对应的一组分配声明. 该方法的一个主要缺点在于其没有考虑非功能需求对体系结构的影响. 通常, 一组功能需求可以被几种具有不同风格的软件体系结构所满足, 但这些不同风格的体系结构往往会体现出不同的非功能属性取值; 在特征操作化或责任分配的过程中, 必须充分考虑非功能需求的影响, 才有可能产生一个具有所需非功能属性取值的软件体系结构.

(V) 特征模型与实现代码之间的追踪关系. 面向特征的编程(feature-oriented programming, FOP)旨在通过增量建模的方式在特征与其实现代码之间建立清晰的追踪关系, 并通过对特征实现代码的组装形成软件产品的源代码. FOP 的概念最早由 Prehofer[14]提出. 在其方法中, 一组特征与其实现代码(用面向对象的方法实现)通过如下方式建立追踪关系: (1) 为这组特征指定一个全序关系(定义了对特征进行组装的顺序); (2) 为每一个特征关联一个代码片段(即特征的实现代码); (3) 对每一个特征, 分别考虑其与全序关系中先于其组装的每一个特征是否存在特征交互问题(即, 当一个特征存在时, 另一个特征的行为必须要做出相应的调整). 若存在, 则声明一个用于解决该特征交互问题的代码片段; (4) 给定一组特征,

按照全序关系依次将每个特征的实现代码添加到目标代码中(目标代码的初值为空); 当一个特征的实现代码被添加至目标代码之后, 如果该特征与已经存在的特征之间存在交互问题, 则分别将解决这些交互问题的代码集成到目标代码中(需要对目标代码中已有的一个或一组方法进行重写); 最终产生的目标代码即是这组特征对应的实现代码.

在 FOP 的研究中, 另一个受到广泛关注的进展是由 Batory 等人提出的 GenVoca 方法[48]及其扩展 AHEAD 方法[49]. 与 Prehofer[14]的方法类似, GenVoca 方法主要关注在面向对象方法中, 如何在一组特征与其实现代码之间建立追踪关系. 该方法认为, 在面向对象程序中, 一个特征的实现代码被分散在多个类中, 而已有的面向对象编程语言对特征实现代码还缺乏有效的封装机制; 在软件中新增一个特征, 可以认为是对原有代码的一种精化, 因此, 可以利用逐步精化的方式在一组特征与其实现代码之间建立准确的追踪关系. 在此基础上, GenVoca 方法通过为一组特征指定一个全序关系来确定程序精化的顺序, 并使用基于 mixin 的继承机制[50]实现对已有面向对象程序的精化(即增量式修改). 与 Prehofer 的方法不同的是, GenVoca 方法没有显式区分特征的实现代码与面向特征交互的实现代码, 而将两者统称为对程序的精化. 在 GenVoca 方法的基础上, AHEAD 方法则进一步将其扩展至在特征与任何非代码型的实现制品之间建立基于逐步精化的追踪关系. 该方法认为, 一个软件产品的实现制品中, 不仅仅包含代码制品, 也包含很多非代码型的制品; 软件实现制品中的一个模块可以被建模为一个由各种制品通过包含关系形成的层次结构(例如, 一个 J2EE EAR 文件中包含一组 JAR 文件、部署描述文件、以及 HTML 文件; 一个 JAR 文件又包含了一组 Package; 一个 Package 又进一步包含一组类文件等); 逐步精化机制需要具有良好的可扩展性来实现对不同类型的实现制品以及由其形成的层次结构的精化. 基于上述理解和观点, AHEAD 方法采用面向对象的方式对任何类型的实现制品的结构进行建模, 并采用基于 mixin 的继承机制对实现制品进行精化. 因此, 当面对一种新的实现制品时, 为了支持对该制品的精化, 建模人员仅需要进行两项活动: (1) 定义制品实例之间的继承关系; (2) 通过基于 mixin 的继承机制, 实现面向该制品实例的精化操作.

在 Apel 等人[51]的工作中, 提出了特征结构树(feature structure tree)和叠加(superimposition)两个概念, 用于支持对特征的实现制品进行建模和组装. 其中, 一个特征的实现制品被建模为一个特征结构树, 即由一组建模元素及其之间的包含关系形成的树形结构; 一个特征集合对应的实现制品则通过对该集合中的所有特征对应的特征结构树进行叠加而形成. 所谓叠加, 是指在对两个特征结构树的共性结构进行分析的基础上, 对这两个特征结构树进行融合, 在融合的过程中, 两者重叠的部分只保留一份; 两者存在差异的部分则根据预先定义的规则对其进行融合. 例如, 设存在两个特征结构树 $T1$, $T2$, 其中分别存在一个包含方法 $M1$ 的类 C 和一个包含方法 $M2$ 的类 C, 那么, 在 $T1$ 和 $T2$ 的叠加结果中, 将会形成一个包含两个方法 $M1$ 和 $M2$ 的类 C (此处, 方法之间的顺序关系并不重要; 显然, 对于顺序敏感的两个建模元素的融合, 则需要预先定义一种顺序敏感的融合策略); 如果 $M1$ 和 $M2$ 是两个具有相同原型的方法, 则需要按照 $T1$ 和 $T2$ 的叠加顺序决定是方法 $M1$ 覆盖方法 $M2$ 或是相反; 不妨假设方法 $M2$ 覆盖方法 $M1$, 则除了简单的覆盖规则之外, 还可以允许在方法 $M2$ 的实现中声明对下层同原型方法 $M1$ 的调用, 从而实现对方法 $M1$ 的增量式扩展.

Prehofer[14]提出的 FOP 方法和 Batory 等人[48]提出的 GenVoca 方法可以被认为是两种通过增量建模在特征与其实现代码之间建立追踪关系的具体技术手段; 而 AHEAD 方法[49]和 Apel 等人[51]的工作则在此基础上, 将代码型的实现制品扩展至非代码型的实现制品, 从而支持在特征与代码型和非代码型的实现制品之间建立追踪关系. 但是, 与理想的增量建模理论[41,42]及其面临的主要问题相比, 上述方法还存在如下几点不足. (1) 理想的增量建模理论将特征模型与其目标模型之间的追踪关系建模为一个开放系统(建模人员只需要维护一个小规模的共性目标模型和一组相对独立的增量修改声明), 而上述方法并没有完全地体现这种思想: 在这些方法中, 需要维护一个全局性的特征之间的全序关系. 因此, 当特征模型中增加了一个特征之后, 建模人员需要显式地修改已经建立的全序关系, 并对已经存在的增量式声明进行分析和修改. 虽然全序关系能够避免由于偏序关系导致的目标模型的不唯一性问题, 但这仅仅是一种治标不治本的解决方案: 对于两个不存在偏

序关系且不存在冲突的特征而言, 增量建模机制本身需要能够保证两者按照任何顺序应用到目标模型都不会破坏目标模型的唯一性. (2) 在某种程度上, 上述方法关注的是如何在一组特征与目标模型之间建立追踪关系, 而不是在特征模型(不仅包含一组特征, 还包含这组特征之间的依赖关系)与目标模型之间建立追踪关系: 这些方法目前均却少对特征模型中其他建模元素(比如, 特征之间的约束关系)进行有效利用的机制(比如, 如果特征 A 依赖于特征 B, 则显然可知两者之间存在偏序关系, 即特征 B 对应的增量修改声明要先于特征 A 的增量修改声明作用至目标模型). (3) 在上述方法中, 除了 Prehofer 提出的 FOP 方法考虑了两个特征之间的特征交互问题(区分了面向特征的增量修改声明和面向特征交互的增量修改声明), 其他方法均没有显式考虑特征交互问题. 目前的增量建模理论[41,42]中的建模元素"增量修改声明的应用条件", 为特征交互问题的显式化提供了一种可能的技术手段: 若应用条件仅涉及一个特征, 则可认为是面向特征的增量修改声明; 若应用条件涉及多个特征, 则可认为是一种面向特征交互的增量修改声明. (4) 对于如何保障建模结果的正确性这样一个核心问题, 上述方法均却少系统的理论支持.

3 相关工作

软件复用的概念和思想自 1968 年被提出[52]以来, 已经过了 40 余年的研究和发展, 有很对学者对软件复用或软件复用中的特定问题进行了系统性的综述. 在本节, 我们选取其中若干具有影响性的综述性研究, 对其核心思想及其与本文的关系进行说明、分析和对比.

1992 年, 美国卡耐基·梅隆大学的学者 Krueger[53] 发表的一篇关于软件复用的综述论文认为, 软件复用的一个基本要素是抽象(即, 如何在尽可能高的抽象层次上对软件可复用成分进行建模); 任何一项软件复用技术或方法, 如果对其关注的可复用成分缺乏足够抽象的建模, 则很难实现有效的复用. 在这种思想的指导下, 这篇论文对当时软件复用研究工作的组织采用了一种两维度的分类框架: 制品维度、概念维度. 在制品维度上, Krueger 将软件开发中的可复用制品划分为 8 类. 按照对可复用成分抽象程度逐渐增高的顺序, 这 8 类制品依次为: 高级编程语言、设计与代码、软件模式、代码构件、软件体系结构、规约语言、变换系统、应用生成器. 对于每一种可复用制品, 在概念维度上则分解为抽象、选择、特殊化、集成 4 个方面: 抽象关注的是特定类型的可复用制品包含的对可复用成分的抽象机制; 选择关注的是如何从一个可复用制品库中定位到符合特定条件的软件制品; 特殊化关注的是如何针对当前的复用需求, 对特定的软件制品进行定制; 集成关注的是如何将一组独立存在的软件制品集成为一个软件产品. 在这 4 个方面中, 抽象是最基础的, 贯穿在其他 3 个方面中, 并在很大程度上决定了其他 3 个方面的实施机制和效率. 需要指出的是, 这篇论文中的软件复用指的是一般意义上的软件复用, 而面向特征的软件复用关注的是面向领域的复用, 即如何对特定领域内的软件可复用成分进行复用. 后者研究目标的缩小, 使得可以在一种更为具体的层次上对一般软件复用涉及的各种问题进行研究和落实. 对比 Krueger 论文中的 8 种可复用软件制品, 本文将面向特征的软件复用中涉及的可复用软件制品划分为 2 类: 概念制品和实现制品. 概念制品即特征模型, 其主要责任是在概念层次上对特定领域的软件可复用成分进行建模. 实现制品即能够实现特征模型中的可复用成分的各类软件制品. 本文不再区分具体类型的实现制品, 其原因如下: 在软件领域的研究中, 已经有大量的工作关注如何提高某一类型的实现制品的抽象程度与可复用性, 但这些研究工作往往是从一般意义上看待一种实现制品的可复用性, 其关注的是一种实现制品中包含的对可复用成分的抽象机制是什么; 从面向领域的复用来看, 这些研究固然能够增加特定类型的实现制品的易理解性和易复用性, 但由于缺乏对可复用成分的显式建模, 这些研究无法在可复用成分和实现制品之间建立显式的追踪关系, 因此也无法取得面向领域的软件复用所期望的复用效率. 但应该看到, 在 Krueger 论文的 8 种可复用软件制品制品中, 规约语言、变换系统和应用生成器这 3 种制品已经具有了面向领域的特点, 但由于处于软件复用的早期发展阶段, 面向领域的思想还未受到广泛关注, Krueger 的论文没有对这个方向的研究进行深入地综述. 基于这种观察, 本文将各类软件实现制品统一抽象为一组元素以及元素之间的关系, 并进一步重点关注如何在实现制品中的元素与特征模型中领域可复用成分之间建立准确的追踪关系. 对比 Krueger 的论文中涉及的抽象、选择、特殊化和集成

这 4 个方面, 本文不再关注某一类实现制品包含的抽象机制是什么, 而关注如何采用特征模型对特定领域的实现制品包含的可复用成分进行与实现无关的抽象建模; 同时, 由于面向领域的特点, 选择、特殊化和集成这 3 个方面在本文中综合体现为追踪关系, 即, 特征模型和软件实现制品之间的追踪关系.

1995 年, 加拿大学者 Mili 等人[54]发表的一篇关于软件复用的论文将软件复用从另外两个维度进行了划分: 在一个维度上, 软件复用包含“为复用而进行的开发”和“基于复用的开发”两个阶段; 在另一个维度上, 软件复用的对象包含“制品”和“过程”两种类型, 进而区分出“制品型复用”和“生成式复用”两种复用方法. 基于这种划分, 这篇论文的综述内容, 在“为复用而进行的开发”上, 主要关注面向对象的制品型软件复用技术和基于应用生成器的生成式软件复用技术两个方面; 在“基于复用的开发”上, 主要关注制品型复用中可复用制品的查询、评估和适应等方面. 除此之外, 这篇论文还对软件复用对组织结构、开发模型、经济性等方面的影响进行了分析. 与 Krueger 的观点类似, Mili 等人也认为对领域知识的复用是软件复用的长期研究目标, 但在当时, 关于这方面的研究还没有得到充分的开展. 面向特征的软件复用在本质上即是一种领域知识驱动的软件复用, 领域知识体现为特征模型, 其中包含了一组在特定领域中具有用户/客户价值的软件特征. 另外, 在本文中, 我们没有显示区分“为复用而进行的开发”和“基于复用的开发”两种复用阶段, 以及“制品型复用”和“生成式复用”两种复用方法, 其原因如下: 我们认为, 在面向特征的软件复用中, 这两种阶段/活动的关系已经得到了更加紧密的增强. 具体而言, 在面向特征的软件复用中, 为复用而进行的开发工作(体现为对特征模型、实现制品以及两者追踪关系的建模与实现)完成后, 基于复用的开发则基本上可以一种自动的方式进行实施, 即一旦用户的需求落实为特征模型的某个定制结果后(这项工作无法自动实施, 用户必须参与其中), 基于追踪关系, 即可自动的定位到需求相应的实现制品, 并自动的对实现制品进行定制和集成; 在这种方式下, 为复用而进行的开发对基于复用的开发的主要技术活动实现了自动化的支持, 因此, 对这两者的区分仅存在概念上的必要性, 而无实际上的必要性. 同样, 生成式复用中对开发过程的复用(体现为不同类型制品之间的追踪与变换), 在特征模型和实现制品之间的追踪关系的支持下, 也可以得到自动化的实施; 在这种方式下, “制品型复用”和“生成式复用”得到了有机的融合.

1999 年, 杨芙清等人[2]发表的一篇关于软件复用和软件构件技术的论文, 将软件复用归结为 3 个基本问题: 存在可复用的对象、所复用的对象具有有用性、复用者知道如何复用可被复用的对象. 在此基础上, 这篇论文进一步将实现软件复用的关键因素总结为 8 个方面, 其中软件构件技术关注可复用制品的基本组织单元; 软件体系结构关注可复用制品的组装; 领域工程关注特定领域的可复用软件制品的生产; 软件再工程关注遗产系统中可复用成分的识别与重构; 开放系统技术关注可复用制品接口的标准性以及接口之间的互操作性; 软件过程关注适应软件复用的软件开发过程; 工具和环境关注对软件复用技术、方法和过程的自动化支持; 非技术因素关注软件复用涉及的组织结构、管理、心理、知识产权、经济性等方面. 以这 8 个关键因素为指导, 这篇论文对当时国际上代表性的软件复用研究和实践进行了介绍, 并进一步重点介绍了当时国内软件复用研究的重要进展——基于构件-构架模式的青鸟软件生产线系统. 与 Krueger[53]和 Mili 等人[54]的观点类似, 认为面向特定领域的软件复用更容易成功. 在本质上, 这篇论文所倡导的是借鉴传统产业零部件生产和组装的制品型软件复用方法. 面向特征的软件复用一方面继承了面向领域的软件复用的基本思想, 同时, 又进一步融合了“制品型复用”和“生成式复用”这两种不同的复用方法. 例如, 这篇论文所指出的软件复用的 3 个基本问题之一: 复用者知道如何复用可被复用的对象, 在理想的面向特征的软件复用中已经不复存在: 复用者只需提供其所需的特征集合, 支撑工具即能够通过特征模型与实现制品之间的追踪关系定位到实现该特征集合的所有软件实现元素, 并对这些实现元素进行自动化的组装进而形成一个独立的软件产品. 目前的研究进展和成果已经能够展示这种理想的软件复用的技术可行性.

2006 年, 在 Frake 和 Kang[55]关于软件复用的现状和未来的简要性评述中, 明确指出: 领域工程是软件复用的核心思想. 其中, 领域工程指的是对特定领域的软件可复用成分进行系统化地识别、分析、建模、实现及复用的一系列活动. 这篇论文从业务和资金、度量和实验、构件技术、代表性领域工程方法、程序

语言、软件体系结构、生成式方法、复用性与安全性等方面对面向领域的软件复用的研究进展进行了简要的评述, 并指出仍然需要进一步关注的研究问题, 包括软件复用技术与方法的规模可扩展性、软件制品更好的表现机制、软件复用对安全性和可靠性的影响、对领域需求变化性的预测、软件复用的可持续性等方面. 这篇论文在一定程度上说明当时软件复用研究的关注点已经从传统的一般意义的复用转向面向领域的复用. 同时, 这篇论文没有再强调对“为复用而进行的开发”和“基于复用的开发”以及对“制品型复用”和“生成式复用”的区别. 进一步, 作者指出生成式复用方法与领域工程方法(即面向领域的软件复用)具有紧密的关系, 并认为如果能在领域分析的输出与应用生成器的输出之间建立清晰的映射关系, 那么, 领域工程的实施将会具有非常好的可重复性. 我们非常认同这种观点: 在本文中, 对面向特征的软件复用的主要关注点之一就是如何在特征模型(领域分析的输出)和实现制品之间建立清晰的追踪关系; 追踪关系的显式化提供了一种统一的视角来看待“制品型复用”和“生成式复用”.

特征模型本质上是一种面向领域的需求模型. 因此, 如何在特征模型和实现制品之间建立追踪关系, 在更基础的层次即体现为如何在软件需求与软件实现制品之间建立追踪关系. 追踪关系在软件工程的研究中也受到了广泛的关注[56], 被认为会对软件变更影响分析、软件的验证与确认、软件制品的理解与复用等方面具有重要的价值; 但由于追踪关系的建立和维护的成本较高, 在一般软件开发中, 其更多地出现在需求变更影响分析中[57], 其主要关注点是如何利用追踪关系来确定一条需求的变化会对哪些实现制品的哪些部分产生影响. 特征在追踪关系中的潜在价值在文献[13]中得到系统的分析. 这篇论文提出了特征工程(feature engineering)的概念, 其核心思想是: 特征应该被作为贯穿软件生命周期、跨越问题空间和解空间的一阶实体(即, 使用特征作为在所有软件制品之间建立追踪关系的中介元素). 在此基础上, 这篇论文对特征对软件开发各个阶段可能的影响进行了讨论. 特征工程和面向特征的软件复用在核心思想方面具有很大的共性, 两者的差别主要体现在视角的不同: 特征工程站在面向单个软件产品开发的视角, 而面向特征的软件复用则站在面向特定领域的软件复用的视角. 同时, 由于追踪关系建立和维护的高成本性, 面向特征的思想在软件复用的环境中更容易取得成功: 追踪关系建立和维护的成本可以被分摊到通过复用产生的一组软件产品中.

面向特征的思想在软件开发中的应用在文献[58]中得到了进一步的关注. 这篇论文提出了面向特征的软件开发的概念, 将其定位为一种用于构造、定制和合成软件产品的软件开发范型; 并认为面向特征的软件开发可以支持领域工程的实施. 在此基础上, 这篇论文从领域分析、领域设计和规约、领域实现、产品定制和生成、基本理论等 5 个开发阶段或方面对相关研究工作进行了介绍. 这篇论文虽然没有明确指出面向特征的软件开发是一种面向特定领域的软件复用方法, 但在其对相关工作的分类中, 完全继承了领域工程中的核心活动. 与这篇论文的分类原则相比, 本文的特点主要体现在两个方面: (1) 本文没有关注面向特征的软件复用中具体的过程性和指南性的信息(如采用什么样的原则和步骤去建立特征模型、实现制品或两者之间的追踪关系), 而只关注过程产生的制品的结构和原理(如特征模型的结构、实现制品的结构、追踪关系的结构和原理); (2) 本文没有关注实现制品的具体类型, 而将其统一抽象为包含一组元素和关系的模型, 并将特征模型与实现制品之间的追踪关系作为一个显式和重要的关注点. 在这种思想指导下, 本文对相关工作进行了进一步的分类、汇总、凝炼和分析. 我们期望, 通过这种不同的分类原则, 能够从一种新的视角来定位和理解面向特征的软件复用这样一个研究方向及其核心思想、方法和技术.

2006 年 Schobbens 等人[8]则主要针对特征模型的研究进展进行了综述. 这篇论文中的特征模型主要是指由精化关系(不包含精化关系的具体类型)、多选多/多选一约束关系、以及二元约束关系 3 种成分构成的特征模型; 由于对后面两种成分的语义已经形成共识, 因此, 这篇论文在本质上主要关注在不同的研究工作中由特征及其精化关系所形成的结构具有的差异性: 在一些研究工作中, 这种结构是树或森林(一个特征最多只有一个父特征); 在另外一些研究中, 则是有向无环图(一个特征可以有多个父特征). 为了包容并建模这些差异性, Schobbens 等人在这篇论文及后继工作中[59]提出了一种抽象的特征模型语法和语义, 其核心在于如何看待两个具有精化关系的特征以及可选性, 例如, 假设在一个精化关系中子特征

是可选的, 与现有的研究或将可选性关联在子特征上或将其关联在精化关系上不同, 在这个抽象语法中, 可选性是独立存在的. 本文认为这种区分过于细化, 且采用树形结构与约束关系组合的方式已经能够表现出具有有向无环图结构的特征模型, 因此, 为了概念上的简洁性, 本文没有关注非树形结构的特征模型. 如果读者有兴趣了解这方面的内容, 可以进一步阅读 Schobbens 等人[8,59]的工作. 2010 年 Benavides 等人[9]则进一步对特征模型自动分析领域的研究进展进行了综述. 特征模型自动分析是指在计算机软件的帮助下自动地从特征模型中抽取出特定的信息(如特征模型中存在的不一致性或缺陷, 特征模型包含的合法的定制结果等静态信息, 以及特征模型定制过程中的完整性、一致性检查等动态信息). 由于这篇论文对特征模型自动分析进行了非常系统性地调研, 本文内容没有涉及这方面的研究工作.

4 总结与展望

面向特征的软件复用在本质上是一种面向领域的软件复用. 面向领域的软件复用通常包括两个重要活动: 领域工程和应用工程. 领域工程也称为复用而进行的开发(development for reuse), 其主要任务是系统地识别、建模和生产面向特定领域的可复用软件资产; 应用工程也称基于复用而进行的开发(development with reuse), 其主要任务是通过复用领域工程活动产生的可复用软件资产开发出特定领域的具体软件应用. 在面向特征的软件复用中, 特征模型的建立、实现制品的生产以及特征模型与实现制品之间追踪关系的维护等活动属于领域工程的范畴; 特征模型的定制、基于特征模型的定制结果定位到相应的实现制品等则属于应用工程的范畴. 软件产品线[6]也是一种面向领域的软件复用技术, 其在传统面向领域的软件复用的基础上, 将可复用资产的管理也作为一种与领域工程和应用工程并列的软件复用活动, 并进一步关注如何在这 3 种软件复用活动之间建立更为紧密的集成和持续演化. 软件产品线一般采用特征来刻画领域的可复用成分并定位领域中具体的软件应用; 这一点与面向特征的软件复用的基本思想是一致的. 面向特征的软件复用中的特征建模活动在本质上也是一种需求工程活动, 但与一般软件开发中的需求工程活动仅关注面向单个软件应用的需求不同, 其关注的是领域中一组软件应用的需求, 涉及到如何对不同应用需求的空间变化性进行有效地建模和管理, 因此, 其复杂性在很大程度上要远高于一般软件开发中的需求工程活动. 面向特征的软件复用对制品之间追踪关系的关注在一般软件开发中也存在, 但这两种关注具有不同的目的: 一般软件开发对追踪关系的关注主要是为了解决需求的时间变化性问题(即, 一个软件系统的需求会随着问题领域的变化而发生变更或演化, 进而要求对相关的实现制品进行适应性的修改); 而面向特征的软件复用对追踪关系的关注主要是为了解决需求的空间变化性问题(即, 当确定了对领域中一个软件应用的需求之后, 需要在领域可复用软件资产库中进一步定位到需求所对应的实现制品, 从而产生可实际运行的软件应用).

面向特征的软件复用通过采用特征模型作为组织和管理特定领域的可复用成分的核心模型, 保证可复用成分的易定制性; 通过在特征模型和软件实现制品之间建立追踪关系, 保证可复用成分的易追踪性. 因此, 面向特征的软件复用若要取得成功, 必须解决两个关键问题: 如何构造面向特定领域的特征模型, 以及如何在特征模型和软件实现制品之间建立清晰的追踪关系. 针对这两个问题, 研究者进行了丰富多样的探索并取得了重要进展; 本文从技术的角度对面向特征的软件复用中上述两个问题的重要研究成果和进展进行了综述.

与大规模定制性生产的最终目标相比较, 当前的研究还存在两点不足: (1) 大多数的方法和技术都隐含地假设, 即在特定领域内存在一个或若干个关于本领域的全知者(领域专家), 他们能够有效地捕获并建模领域中的可复用成分. 在信息化进程不断加速, 软件应用领域层出不穷、不断演化与相互融合的现状中, 这一假设过于乐观, 且在多数情况中并不成立. (2) 大多数的方法和技术对建模结果的演化缺乏有效的支持. 这将导致特定领域的特征模型、实现制品及其之间的追踪关系在建立之后, 由于得不到持续地演化, 而无法准确反映领域的发展现状, 从而逐渐丧失其存在价值. 这两个问题的解决存在本质上的困难: 单一软件产品的开发本身就是一个知识、人力和协同密集的活动, 存在本质上的复杂性和困难性[1]; 相比较单一软件产品的开发, 面向特征的软件复用则是一种面向一组软件产品的开发活动, 其所具有的复杂性和困难性进一步增加, 所需的知识、人

力以及协同的密集程度也进一步提升.

我们认为, 对上述两个问题的解决需要同时考虑技术和人两方面的因素. 在技术因素上, 需要继续对现有的面向特征的软件复用方法和技术进行完善和发展: 在特征模型结构和语义的研究中, 要避免对特征模型进行盲目地扩展, 而应该明确围绕"提升对领域可复用成分的有效组织和管理"这样一个核心目标, 探索更具直接性和易用性的建模机制(其中, 特征模型与领域特定语言[60]的结合, 是一个具有重要价值的研究问题); 在特征模型与实现制品之间追踪关系的研究中, 基于增量建模的追踪关系构造方法, 一方面需要进一步增强对其基础理论和基本性质研究, 以明确其理论表达能力, 另一方面需要将其落实到具体的语言和支撑环境中, 以增强其在开发活动中的易用性. 在人的因素上, 要破除现有方法和技术的实施对少数专家的依赖. 由于软件的逐渐泛在化, 一个软件产品的利益相关者群体的规模和多样性也在持续地增强, 少数专家已经很难应对面向领域的软件复用的实施任务. 一种可能的解决方案是采用群体协同的方式对特征模型及其与实现制品之间的追踪关系进行建模和演化. 理论上, 在一定的协同机制的支持下, 只要参与人员在群体层次上具有足够的领域知识(即, 所有参与人员个体知识的总和对领域知识具有足够的覆盖度), 且能够持续地(有意识或无意识地)参与到建模活动中, 就能使得建模结果在保证较高质量的同时不断演化; 现实中, Internet 和 Web 技术的不断发展为大规模的人类群体协同提供了基本的技术支持[61], 而各种基于 Web 的群体智能应用[62]的出现则在一定程度上展示了这种解决方案的可行性. 在这种解决方案中, 如何将面向特征的软件复用中的关键任务和活动构造成为开放、增量和并行化的系统, 将是一个关键技术问题.

参考文献

1 Brooks F P. The Mythical Man-Month: Essays on Software Engineering. 2nd ed. Boston: Addison-Wesley, 1995

2 杨芙清, 梅宏, 李克勤. 软件复用与软件构件技术. 电子学报, 1999, 27: 68–75

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

4 Parnas D L. On the design and development of program families. IEEE T Software Eng, 1976, SE-2: 1–9

5 Neighbors J M. Software construction using components. Doctoral Dissertation. Irvine: University of California,1980

6 Clements P, Northrop L. Software Product Lines: Practices and Patterns. Boston: Addison-Wesley, 2001

7 Kang K C, Cohen S G, Hess J A, et al. Feature-oriented domain analysis feasibility study. Technical Report. Carnegie Mellon University, 1990

8 Schobbens P Y, Heymans P, Trigaux J C. Feature diagrams: A survey and a formal semantics. In: Glinz M, Lutz R, eds. Proceedings of 14th International Requirements Engineering Conference, 2006 September 11–15, Minnesota. Los Alamitos, CA: IEEE Computer Society, 2006. 136–145

9 Benavides D, Segura S, Ruiz-Cortés A. Automated analysis of feature models 20 years later: A literature review. Inform Syst, 2010, 35: 615–636

10 Coplien J, Hoffman D, Weiss D. Commonality and variability in software engineering. IEEE Softw, 1998, 15: 37–45

11 Davis A M. The design of a family of application-oriented requirements languages. IEEE Comput, 1982, 15: 21–28

12 Keck D O, Kuehn P J. The feature and service interaction problem in telecommunications systems: A survey. IEEE T Softw Eng, 1998, 24: 779–796

13 Turner C R, Fuggetta A, Lavazza L, et al. A conceptual basis for feature engineering. J Syst Softw, 1999, 49: 3–15

14 Prehofer C. Feature-oriented programming: A fresh look at objects. In: Aksit M, Matsuoka S, eds. Proceedings of 11th European Conference on Object-Oriented Programming, 1997 June 9–13, Jyväskylä, Finland. Berlin Heidelberg: Springer, 1997. 419–443

15 Mehta A, Heineman G T. Evolving legacy system features into fine-grained components. In: Tracz W, Young M, Magee J, eds. Proceedings of 22rd International Conference on Software Engineering, 2002 May 19–25, Orlando, Florida. New York: ACM, 2002. 417–427

16 Chastek G, Donohoe P, Kang K C, et al. Product line analysis: A practical introduction. Technical Report. Carnegie Mellon University, 2001

17 Griss M, Favaro J, d'Alessandro M. Integrating feature modeling with the RSEB. In: Devanbu P, Poulin J, eds. Proceedings of 5th International Conference on Software Reuse, 1998 June 2–5, Victoria, British Columbia, Canada. Los Alamitos, CA: Computer Society, 1998. 76–85

18 Eriksson M, Borstler J, Borg K. The PLUSS approach-domain modeling with features, use cases and use case realizations. In: Obbink J H, Pohl K, eds. Proceedings of 9th International Software Product Line Conference, 2005 September 26–29, Rennes, France. Berlin Heidelberg: Springer, 2004. 33–44

19 Mei H, Zhang W, Gu F. A feature oriented approach to modeling and reusing requirements of software product lines. In: Helms R, Bae D H, Voas J, eds. Proceedings of 27th International Computer Software and Applications Conference, 2003 November 3–6, Dallas. Los Alamitos, CA: IEEE Computer Society, 2003. 250–256
20 Wiegers K E. Software Requirements. Washington: Microsoft Press, 1999.
21 IEEE Standards Board. IEEE Standard Glossary of Software Engineering Terminology. 1990
22 Zhang W, Yan H, Zhao H, et al. A BDD-based approach to verifying clone-enabled feature models' constraints and customization. In: Mei H, ed. Proceedings of 10th International Conference on Software Reuse, 2008 May 25–29, Beijing. Berlin Heidelberg: Springer, 2008. 186–199
23 Jacobson I. Object Oriented Software Engineering: A Use Case Driven Approach. Boston: Addison-Wesley, 1992
24 Leffingwell D, Widrig D. Managing Software Requirements: A Use Case Approach. Boston: Addison-Wesley, 2003
25 Object Management Group. Object Constraint Language (Version 2.2)
26 Mannion M. Using first-order logic for product line model validation. In: Chastek G J, ed. Proceedings of 2nd International Software Product Line Conference, 2002 August 19–22, San Diego. Berlin Heidelberg: Springer, 2002. 176–187
27 Zhang W, Zhao H, Mei H. A propositional logic-based method for verification of feature models. In: Davies J, Schulte W, Barnett M, eds. Proceedings of 6th International Conference on Formal Engineering Methods, 2004 November 8–12, Seattle. Berlin Heidelberg: Springer, 115–130
28 Batory D. Feature models, grammars, and propositional formulas. In: Obbink J H, Pohl K, eds. Proceedings of 9th International Software Product Line Conference, 2005 September 26–29, Rennes, France. Berlin Heidelberg: Springer, 2004. 7–20
29 Lee K, Kang K C. Feature dependency analysis for product line component design. In: Bosch J, Krueger C C, eds. Proceedings of 8th International Conference on Software Reuse, 2004 July 5–9, Madrid, Spain. Berlin Heidelberg: Springer, 2004. 69–85
30 Kang K C, Kim S, Lee J, et al. FORM: A feature-oriented reuse method with domain-specific reference architectures. Ann Softw Eng, 1998, 5: 143–168
31 Zhang W, Mei H, Zhao H. Feature-driven requirement dependency analysis and high-level software design. Requir Eng, 2006, 11: 205–220
32 Lee K, Kang K C. Feature dependency analysis for product line component design. In: Bosch J, Krueger C C, eds. Proceedings of 8th International Conference on Software Reuse, 2004 July 5–9, Madrid, Spain. Berlin Heidelberg: Springer, 2004. 69–85
33 Peng X, Zhao W, Xue Y, et al. Ontology-based feature modeling and application-oriented tailoring. In: Morisio M, ed. Proceedings of 9th International Conference on Software Reuse, 2006 June 12–15, Turin, Italy. Berlin Heidelberg: Springer, 2006. 87–100
34 Czarnecki K, Bednasch T, Unger P, et al. Generative programming for embedded software: An industrial experience report. In: Batory D S, Consel C, Taha W, eds. Proceedings of ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, 2002 October 6–8, Pittsburgh, PA. Berlin Heidelberg: Springer, 2002. 156–172
35 Riebisch M, Bollert K, Streitferdt D, et al. Extending feature diagrams with UML multiplicities. In: Krämer B J, Ehrig H, Ertas A, eds. Proceedings of 6th International Conference on Integrated Design & Process Technology, 2002 June 23–28, Pasadena C A. Dallas, TX: Society for Design and Process Science, 2002. 1–7
36 Czarnecki K, Helsen S, Eisenecker U. Formalizing cardinality-based feature models and their specialization. Softw Process Improv Pract, 2005, 10: 7–29
37 Benavides D, Trinidad P, Ruiz-Cortes A. Automated reasoning on feature models. In: Pastor O, Cunha J F, eds. Proceedings of 17th International Conference on Advanced Information Systems Engineering, 2005 June 13–17, Porto, Portugal. Berlin Heidelberg: Springer, 2005. 491–503
38 Benavides D, Trinidad P, Ruiz-Cortes A. Using constraint programming to reason on feature models. In: Chu W C, Juzgado N J, Wong W E, eds. Proceedings of 17th International Conference on Software Engineering and Knowledge Engineering, 2005 July 14–16, Taipei, Taiwan. Skokie, IL: Knowledge Systems Institute, 2005. 677–682
39 Karatas A S, Oguztuzun H, Dogru A. Mapping extended feature models to constraint logic programming over finite domains. In: Bosch J, Lee J, eds. Proceedings of 14th International Software Product Lines Conference, 2010 September 13–17, Jeju Island, South Korea. Berlin Heidelberg: Springer, 2010. 286–299
40 Czarnecki K, Antkiewicz M. Mapping features to models: A template approach based on superimposed variants. In: Glück R, Lowry M R, eds. Proceedings of 4th International Conference on Generative Programming and Component Engineering, 2005 September 29–October 1, Tallinn, Estonia. Berlin Heidelberg: Springer, 2005. 422–437
41 Clarke D, Helvensteijn M, Schaefer I. Abstract delta modeling. In: Visser E, Järvi J, eds. Proceedings of 9th International Conference on Generative Programming and Component Engineering, 2010 October 10–13, Eindhoven, Netherlands. New York: ACM, 2010. 13–22

42 Schaefer I, Bettini L, Bono V, et al. Delta-oriented programming of software product lines. In: Bosch J, Lee J, eds. Proceedings of 14th International Software Product Lines Conference, 2010 September 13–17, Jeju Island, South Korea. Berlin Heidelberg: Springer, 2010. 77–91

43 Jacobson I, Griss M, Jonsson P. Software Reuse: Architecture, Process and Organization for Business Success. Boston: Addison-Wesley, 1997

44 Braganca A, Machado R J. Automating mappings between use case diagrams and feature models for software product lines. In: Kang K C ed. Proceedings of 11th International Software Product Line Conference, 2007 September 10–14, Kyoto, Japan. Los Alamitos, CA: IEEE Computer Society, 2007. 3–12

45 Yu W, Zhang W, Zhao H, et al. A traceability description language between feature model and use case. Technical Report. Peking University, 2012

46 Hendrickson S A, van der Hoek A. Modeling product line architectures through change sets and relationships. In: Knight J, ed. Proceedings of 29th International Conference on Software Engineering, 2007 May 20–26, Minneapolis. Los Alamitos, CA: IEEE Computer Society, 2007. 189–198

47 梅宏, 黄罡, 赵海燕, 等. 一种以软件体系结构为中心的网构软件开发方法. 中国科学 F 辑: 信息科学, 2006, 36: 1100–1126

48 Batory D, O'Malley S. The design and implementation of hierarchical software systems with reusable components. ACM T Softw Eng Meth, 1992, 1: 355–398

49 Batory D, Sarvela J, Rauschmayer A. Scaling step-wise refinement. IEEE T Soft Eng, 2004, 30: 355–371

50 Smaragdakis Y, Batory D. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. ACM T Softw Eng Meth, 2002, 11: 215–255

51 Apel S, Kastner C, Lengauer C. FEATUREHOUSE: Language-independent, automated software composition. In: Fickas S, Atlee J, Inverardi P, eds. Proceedings of 31st International Conference on Software Engineering, 2009 May 16–24, Vancouver, Canada. Los Alamitos, CA: IEEE Computer Society, 2009. 221–231

52 McIlroy M D. Mass produced software components. In: Naur P, Randell B, eds. Report of NATO Conference on Software Engineering, 1968 October 7–11, Garmisch, Germany. Brussecs Belgium: NATO Science Committee, 1969. 138–155

53 Krueger C W. Software reuse. ACM Comput Surv, 1992, 24: 131–183

54 Mili H, Mili F, Mili A. Reusing software: Issues and research directions. IEEE T Softw Eng, 1995, 21: 528–562

55 Frakes W B, Kang K C. Software reuse research: Status and future. IEEE T Softw Eng, 2006, 31: 529–536

56 Spanoudakis G, Zisman A. Software traceability: A roadmap. In: Chang S K, ed. Handbook of Software Engineering and Knowledge Engineering, Vol. III: Recent Advancements: World Scientific Publishing, 2005. 395–428

57 Bohner S A. Impact analysis in the software change process: A year 2000 perspective. In: Schneidewind N, Bohner S A, Cimitile A, eds. Proceedings of International Conference on Software Maintenance, 1996 November 4–8, Monterey. Los Alamitos, CA: IEEE Computer Society, 1996. 42–51

58 Apel S, Kastner C. An overview of feature-oriented software development. J Object Tech, 2009, 8: 49–84

59 Schobbens P Y, Heymans P, Trigaux J C, et al. Generic semantics of feature diagrams. Comput Netw, 2007, 51: 456–479

60 Mernik M, Heering J, Sloane A M. When and how to develop domain-specific languages. ACM Comput Surv, 2005, 37: 316–344

61 梅宏, 刘譞哲. 互联网时代的软件技术: 现状与趋势. 科学通报, 2010, 55: 1214–1220

62 Malone T W, Laubacher R, Dellarocas C. The collective intelligence genome. MIT Sloan Manage Rev, 2010, 51: 21–31

Feature-oriented software reuse technology—State of the art

ZHANG Wei[1,2] & MEI Hong[1,2]

[1] *Key Laboratory of High Confidence Software Technology (Peking University), Ministry of Education of China, Beijing 100871, China;*
[2] *Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China*

Most traditional industries focus on the reproduction-based mass production of physical products, which is not suit for the software industry. To achieve efficient and effective production in the software industry, software reuse is a realistic approach, which focuses on the customization-based mass production of software products. Feature-oriented software reuse provides a technical approach to enforcing the customization-based mass production of software products, and has shown important influence on the research and practice of software reuse. This paper clarifies basic concepts and key ideas in feature-oriented software reuse, and presents important achievements in recent years in this field. In particular, this paper: (1) inspects the existing definitions of features from the two viewpoints of intension and extension, and analyzes the responsibilities of features in software reuse; (2) explains the basic structure of feature models and its semantics based on the propositional logic, and presents three kinds of extension to the structure and semantics of feature models; (3) describes two general ways of modeling the traceability between feature models and implementation artifacts, and surveys research work on traceability between feature models and three kinds of implementation artifacts (i.e. use case, software architecture, and source code), respectively. We hope that this paper could facilitate the understanding of essential problems and state-of-the-art and further improve the research progress in the field of feature-oriented software reuse.

feature, feature model, traceability, domain engineering, software reuse, software engineering

doi: 10.1360/972013-341