软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(7):2562−2580 [doi: 10.13328/j.cnki.jos.006278]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向领域驱动设计的逆向建模支持方法*

钟陈星[1,2], 李文君[1,2], 任贵杰[1,2], 荣国平[1,2]

[1](南京大学 软件学院, 江苏 南京 210093)

[2](计算机软件新技术国家重点实验室(南京大学), 江苏 南京 210023)

通信作者: 荣国平, E-mail: ronggp@nju.edu.cn

摘 要: 领域驱动设计作为一种应对领域复杂性的软件开发方法, 近年来得到了广泛应用. 作为其中的关键步骤, 领域建模仍然面临着领域模型与程序设计之间松散的逻辑关系带来的种种问题. 为了应对此问题, 基于模型驱动的逆向工程, 提出了一种面向领域驱动设计的代码到模型的转换方法, 以更好地支持领域建模, 并实现了自动化工具原型. 该方法能够实时抽象出程序设计对应的领域模型, 一方面有助于通过对比程序设计与领域模型的分歧来避免设计偏离模型, 另一方面减少了知识消化过程中对代码实现细节的依赖, 从而促进了程序设计对建模过程的反馈. 案例研究部分将该方法应用于实际软件项目, 结果充分表明了其有效性.

关键词: 领域驱动设计; 代码到模型转换; 模型驱动的逆向工程; 领域模型; 程序设计

中图法分类号: TP311

中文引用格式: 钟陈星, 李文君, 任贵杰, 荣国平. 面向领域驱动设计的逆向建模支持方法. 软件学报, 2022, 33(7): 2562−2580. http://www.jos.org.cn/1000-9825/6278.htm

英文引用格式: Zhong CX, Li WJ, Ren GJ, Rong GP. Reverse Modeling Support Method for Domain-driven Design. Ruan Jian Xue Bao/Journal of Software, 2022, 33(7): 2562−2580 (in Chinese). http://www.jos.org.cn/1000-9825/6278.htm

Reverse Modeling Support Method for Domain-driven Design

ZHONG Chen-Xing[1,2], LI Wen-Jun[1,2], REN Gui-Jie[1,2], RONG Guo-Ping[1,2]

[1](Software Institute, Nanjing University, Nanjing 210093, China)

[2](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: As a software development method to tackling the domain complexity, domain-driven design has been widely applied in recent years. However, as a key activity in domain-driven design, domain modeling is still facing the problems caused by the loose relationship between domain model and programming. To address this issue, this study proposes a code to model transformation method following the model-driven reverse engineering methodology. The method can abstract model designing from code in real-time, thus facilitates domain modeling in two ways. On the one hand, it enables comparing domain model and programs to avoid the deviation of programming from modeling. On the other hand, it reduces the dependence on the code details during the knowledge crunching process, thus promotes the feedback on modeling. A case study is conducted in a real scenario and it proves the validity of the proposed method.

Key words: domain-driven design; code to model transformation; model-driven reverse engineering; domain model; programming

进入 21 世纪以来, 软件对于现代世界的运作扮演着不可或缺的角色. 伴随着软件系统需求的日益复杂化和互联网技术的飞速发展, 软件开发的复杂性急剧增长. 越来越多的软件开发人员认识到, 软件开发复杂性的核心在于软件所运行着的领域本身. 领域驱动设计(domain-driven design)[1] 作为一种应对系统领域复杂性

* 基金项目: 国家自然科学基金(62072227, 61802173); 国家重点研发计划(2019YFE0105500); 江苏省政府间双边创新项目(BZ2020017); 南京大学计算机软件新技术国家重点实验室创新项目(ZZKT2019B01)

本文由“面向持续软件工程的微服务架构技术”专题特约编辑张贺教授、王忠杰教授、陈连平研究员和彭鑫教授推荐.

收稿时间: 2020-09-13; 修改时间: 2020-10-26; 采用时间: 2020-12-15

的软件开发方法, 正是在这样的大环境中得到广泛应用.

领域驱动设计提出围绕领域模型(domain model)进行软件设计和开发. 领域模型是由开发人员与领域专家协作构建出的一个反映深层次领域知识的模型, 该模型对软件所在领域的业务逻辑进行了有组织且有选择的抽象. 领域驱动设计还要求绑定领域模型和代码实现, 从而使建模过程获得的深层次领域知识能够转化为软件开发的最终产品. 领域模型和代码实现的绑定, 还能有效促进软件的维护和后续开发, 使得开发人员能够基于对模型的理解来解释和重构代码. 此外, 在多个领域模型共存的大型项目中, 不同模型的代码组合在一起使软件变得不可靠和难以理解, 领域驱动设计利用限界上下文(bounded context)明确地设置模型边界以保持模型的清晰和完整[2]. 领域驱动设计的这些特点与微服务架构(microservices architecture)[3]相契合, 这使其常被用于拆分微服务应用, 成为微服务应用成功的关键因素[3].

然而, 领域建模是领域驱动设计在实际应用中面临的一项严峻挑战[4]. 为了达到领域模型和程序设计之间的一致, 领域驱动设计追求满足两方面需求的单一模型, 即领域模型的创建应考虑程序设计的实用性, 程序设计也应忠实地表达领域模型. 领域建模过程包含创建领域模型、实现领域模型代码和基于领域模型及代码展开知识消化这 3 个步骤, 并通过迭代这些步骤来不断地精炼模型: 首先, 创建领域模型过程使用了一系列构造块(building block), 每个构造块都以模式语言(pattern language)形式描述了特定领域问题的建模和程序设计思路; 接着, 领域驱动设计还强调应用面向对象编程范式实现领域模型代码, 从而使程序设计能够直接创建对应的模型概念; 最后, 由于领域模型并不总是实用、有价值的, 领域驱动设计还通过迭代的知识消化(knowledge crunching)达到精炼模型的目的, 比如基于代码实验的模型可行性测试.

问题在于, 领域驱动设计本身并未就领域建模过程提供具体指导和相应的工具支持[4], 领域建模很大程度上仍依赖于组织实践领域驱动设计的方式、领域专家和开发人员的个人能力等. 这使其经常面临两种困境: 一方面, 开发人员对领域模型存在理解偏差, 领域模型的一些意图常常在其代码实现过程中丢失[1], 比如领域对象之间的双向关联等; 另一方面, 代码中的大量实现细节使得知识消化过程缓慢且困难, 编码阶段发现的关键知识难以及时地反馈给模型, 比如对某部分工作效率极低的模型规则的改进等. 这些问题的根因在于领域模型和程序设计仅存在松散的逻辑联系, 而适当的方法及工具支持有利于将两者紧密结合起来, 以支持领域建模过程.

为了在领域模型和程序设计之间构建更加紧密的联系, 本文基于模型驱动的逆向工程(model-driven reverse engineering)[5]提出一种代码到模型的自动化转换方法, 为领域建模过程提供理论及工具支持. 所提出的方法及其工具支持可实时抽象出程序设计对应的领域模型, 一方面能够通过对比转换得到的领域模型与初始领域模型, 以快速发现二者的分歧从而避免程序设计偏离领域模型; 另一方面, 能够围绕抽象程序设计得到的领域模型展开知识消化, 减少其对代码实现细节的依赖, 从而加速知识消化过程.

本文第 1 节介绍背景及相关工作. 第 2 节概述代码到模型转换方法的整体框架. 第 3 节给出代码到模型转换过程的各状态所遵循的元模型. 第 4 节讨论代码到模型转换的规则和具体算法. 第 5 节实现自动化工具原型. 第 6 节利用案例研究验证方法的有效性. 第 7 节总结全文, 并提出下一步的工作.

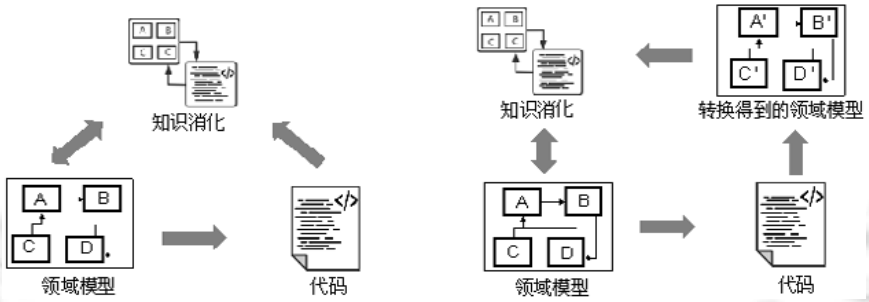
1 背景及相关工作

过去几年里, 一些研究人员致力于解决领域建模过程中领域模型和程序设计之间联系松散的问题. 总体而言, 这些工作都是通过领域模型到代码的转换来为领域建模提供支持, 都可以看作模型驱动工程(model-driven engineering)的具体应用. Le 等人[6]利用基于注解的领域特定语言(annotation-based domain specific language)实现了以软件模块为基本单位的领域模型到代码转换, 每个转换得到的软件模块包括模型、对应的视图及控制器这 3 部分. 该方法首先提出一种领域类规范语言以规范领域模型表达; 然后提出一种基于注解的领域特定语言以规范软件模块的代码表达, 包括两个中间状态代码和一个最终状态代码; 最后给出了算法及支持工具, 以自动化地完成领域模型到软件模块代码的转换. Soares 等人[7]开发了一个领域模型到代码的自动化生成工具 Elihu, 该工具引入多种领域驱动设计模式(domain-driven design pattern)以定义模型语义, 并依

赖裸对象模式框架(naked objects pattern framework)实现代码的自动生成功能. Schneider 等人[8]提出一种领域模型到代码的系统化转换方法, 包括领域对象到代码的转换、添加与数据库相关的代码注解、实现领域业务逻辑为对象方法以及执行测试等步骤. 此外, 与上述工作类似的还有领域驱动设计实现框架 OpenXava[9]和 ApacheIsis[10]. 然而, 这类方法虽然一定程度上促进了领域建模实践, 但仍存在一些亟待解决的问题: 其一, 转换得到的代码通常只实现了简单的业务逻辑, 仍需要开发人员对其手动调整以使其正常运行[8]; 其二, 代码可能被开发人员手动修改, 领域模型也可能发生需求更新, 在领域模型和代码都不断演进时, 如何维护二者的一致性仍是一项难题[11−13]. 更重要的是, 这些工作都忽略了领域建模的知识消化过程, 未能从知识消化角度提供领域建模支持.

鉴于上述方法仍存在诸多不足, 本文探索了一种不同于已有工作的崭新思路, 尝试通过代码到模型的转换来更好地支持领域建模. 该方法可以看作是模型驱动的逆向工程方法的具体应用, 能够从两方面促进领域建模实践.

- 一方面, 可视化地对比转换得到的领域模型与初始领域模型(如图 1(b)所示), 能够更快地发现程序设计与领域模型的分歧, 从而避免程序设计偏移领域模型. 这一过程还能鼓励开发人员在理解领域模型意图的基础上进行编码, 从而更忠实地反映领域模型, 一定程度上促进了开发人员参与领域建模.
- 另一方面, 转换得到的领域模型相比于程序设计具有更高的抽象层级, 这使得领域专家和开发人员的知识消化活动能够仅围绕领域模型展开(如图 1(b)所示), 而不涉及代码的具体细节(比如方法实现细节, 如图 1(a)). 考虑到大部分领域专家对软件开发技术所知有限, 这种转换能在一定程度上减小领域专家参与软件开发的认知复杂度, 从而提高代码到模型的反馈效率.



(a) 领域建模过程 (b) 代码到模型转换如何支持领域建模过程

图 1 领域建模过程和代码到模型转换如何支持领域建模过程

模型驱动的逆向工程方法的特点在于, 其在整个代码到模型转换的过程中, 都使用模型表达系统信息并且使用元模型(metamodel)[5]表达模型规范. 元模型描述了模型的抽象语法, 包括重要的概念及其关系[14]. 其中的重要概念又称为元概念(metaconcept), 比如 UML 元模型定义了类元概念来建模具有某种特征的对象. 在模型驱动的逆向工程中, 不同工作通常选用不同的元模型来表达模型以实现不同的目的[5]. MoDisco 框架[15]依赖知识发现元模型(knowledge discovery metamodel)提供通用、可扩展的代码到模型转换解决方案. Ramón 等人[16]通过提取图形用户接口(graphical user interface)信息恢复遗留系统的高层次布局模型, 得到的最终模型符合具体用户接口(concrete user interface)元模型. Nirumand 等人[17]从安卓应用中提取与通信模型相关的安全漏洞信息, 并将其组织成领域特定模型, 得到的最终模型符合安卓应用安全(Android applications security aspects)元模型. 这些工作通常包括从软件制品中提取初始模型和对初始模型进行一系列模型转换两个阶段, 具体的转换操作根据使用的元模型和最终目的而有所不同. 这些工作选用的元模型可以分为两类: (1) 标准元模型, 比如 MoDisco 框架[15]; (2) 特定领域(domain specific)的元模型, 比如 Ramón 等[16]. 相比于面向通用领域的标准元模型, 面向特定领域的元模型由于结合了该领域的术语和实践规则而更符合该领域的特点, 因此也更易于应用[5]. 尽管已有模型驱动的逆向工程方法支持代码到模型的转换, 但这些方法所使用的元模型要么

面向通用领域, 要么面向其特定领域, 都未结合领域建模的术语, 不特定于领域建模过程.

利用模型驱动的逆向工程支持领域建模, 需要选用特定于领域驱动设计的元模型. 从领域驱动设计问世以来, 元模型一直是该领域的研究热点. Diepenbrock 等人[18]基于本体论(ontology)提出一种元模型来表达不同领域模型间关系, 从而维护不同领域模型在演进过程中的语义一致性, 其所关注的领域建模概念包括领域模型、限界上下文和共享模型(shared model). Rademacher 等人[19−21]基于 UML Profile 机制扩展 UML 元模型, 使模型元素直接表达领域驱动设计的术语, 从而规范领域建模行为, 其所关注的领域建模概念仅参考了 Evans[1]. 此外, 一些模型到代码转换工作同样提供了领域驱动设计的元模型支持, 比如文献[7,8]等. 然而, 这些已有的面向领域驱动设计的元模型一方面仅结合了领域建模的部分术语, 另一方面仅用于创建领域模型而未考虑领域模型的代码实现, 领域驱动设计仍缺少系统的、针对整个领域建模过程的元模型.

为此, 本文面向整个领域建模过程定义了一种相对系统的元模型, 并基于此提出一种代码到模型转换方法(code2model for DDD, C2MD), 同时还实现了自动化工具原型以支持领域建模过程.

2 面向领域驱动设计的代码到模型转换过程

本文提出的 C2MD 代码到模型转换方法基于马蹄模型(horseshoe model)[22], 马蹄模型是一个集成了代码和架构视图的逆向工程框架, 从代码到架构再到代码的转换被表示为“马蹄”形. “马蹄”的左部对应着从软件制品提取信息以恢复架构, 右部对应着从抽象架构到具体代码的传统软件开发活动[15,23]. 本文提出的代码到模型转换方法对应于马蹄模型的左部, 具有 4 个不同的抽象层级(包括代码和 3 种模型状态), 如图 2 所示.

(1) L0 层: 表示代码到模型转换的起点, 对应着领域建模过程中基于领域模型进行代码实现的产物. 代码中除了领域模型的相关概念外, 还可能包含与领域业务规则无关的技术细节, 比如数据库存储等基础设施实现细节. 本文中的代码基于面向对象的编程语言(object-oriented programming language, OOPL)[24]实现.

(2) L1 层: 对应着对代码进行静态分析得到的模型, 系统的每个代码文件都被静态分析并提取相关信息以得到一个抽象语法树, 对应着一个 L1 层模型. 由于 L1 层模型与具体的代码实现技术强相关, 因此被对象管理组织(object management organization)称为平台特定模型(platform specific model, PSM).

(3) L2 层: 对应着集成所有 L1 层的 PSM 模型得到的模型. 每个 L1 层模型都对应着一个 L2 层模型的领域对象, 包含具体的属性与操作细节. 由于此阶段的模型独立于具体的代码实现技术, 因此被对象管理组织称为平台独立模型(platform independent model, PIM).

(4) L3 层: 对应着 L2 层模型的抽象模型, 仅包含领域对象及领域对象间关系, 而不涉及具体的操作实现细节. 由于此阶段的模型独立于具体的计算细节, 因此被对象管理组织称为计算独立模型(computation independent model, CIM).

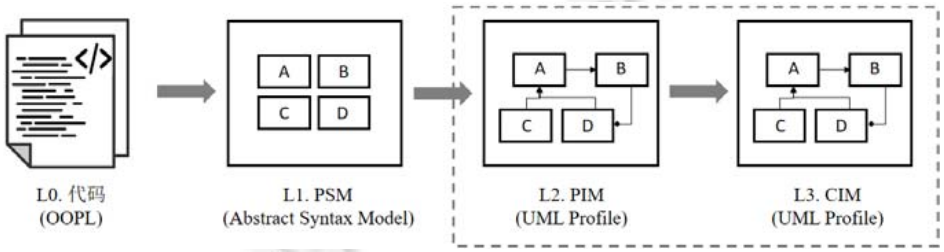


图 2 C2MD 的代码到模型转换过程

值得注意的是, L1 层模型来自对 L0 层代码的静态分析, 因此, L1 层模型的元模型与 L0 层代码元模型(比如 Java 元模型)有着基本一致的元概念(比如 Java 元模型的 class 元概念). 此外, L3 层 CIM 模型的元模型是 L2

层 PIM 模型的子集, 二者的区别仅在于, 前者不包含领域对象的操作实现细节, 更符合传统 UML 规范[25]对领域模型的认识; 而后者则包含领域对象操作细节的设计以支持后续代码实现. 这种区分 CIM 和 PIM 模型的观点与 Evans[1]及 Hippchen 等人[4]一致. 容易看出, PIM 模型和 CIM 模型作为 C2MD 过程的输出, 都可以作为知识消化过程的输入, 辅助领域专家和开发人员提炼深层领域模型.

3 面向领域建模过程的元模型

本文基于领域特定语言(domain specific language, DSL)[26,27]方法定义了面向领域建模过程的元模型, 来表达 C2MD 代码到模型转换过程中不同抽象层级(L0 到 L3)所遵循的模型规范. 基于 DSL 方法定义的元模型包括两种软件制品: (1) 抽象语法元模型(abstract syntax metamodel, ASM),描述了一系列特定于该领域的重要概念、概念间的关系和语法约束; (2) 具体语法元模型(concrete syntax metamodel, CSM)描述了 ASM 在不同语言(比如编程语言 OOPL 和建模语言 UML)中的具体表现形式, 一个 ASM 可以对应多个 CSM. ASM 和 CSM 的这种对应关系符合领域驱动设计的原则, 即领域模型是领域业务逻辑的有组织且有选择的抽象, 可以表现为模型图, 也可以表现为代码或文字[1]. 因此, 代码到模型转换过程的不同抽象层级可以理解为领域模型的不同表现形式.

本文将基于 DSL 定义的面向领域建模过程的元模型命名为领域建模规范语言(domain modeling specific language, DMSL). DMSL 的组成如图 3 所示. 本文首先给出面向领域建模过程的 ASM, 称为概念元模型. 概念元模型描述了领域建模过程的重要概念及其关系, 独立于特定的程序设计和建模技术. 基于概念元模型, 本文定义了代码到模型转换过程的 4 个不同抽象层级所遵循的元模型, 为具体语法元模型 CSM. 本文定义的具体元模型通过特化通用的语言(比如编程语言 OOPL 和建模语言 UML)来表示领域特定概念获得, 因此同时拥有通用性和特定于领域建模两方面好处.

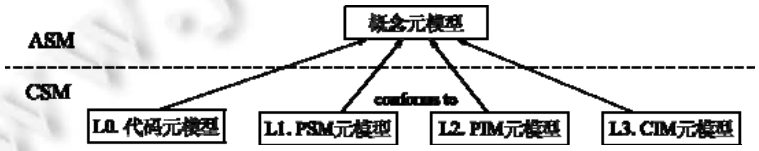


图 3 领域建模规范语言组成

3.1 概念元模型

将领域建模的关键元素及该过程使用的重要概念抽象出来, 得到图 4 所示的 ASM, 即 DMSL 的概念元模型. 其中, 领域建模的关键元素(图 4 黑体部分)描述了领域建模的问题本质. 领域建模遵循面向对象的建模范式. 在领域建模时, 领域模型(domain model)本身是领域业务规则的一种抽象, 每个领域模型对应特定的业务规则. 而每个业务规则可能涉及多个领域对象(domain object), 每个领域对象负责实现业务规则的一部分逻辑. 每个领域对象还具有多个属性(domain attribute)和操作(domain operation).

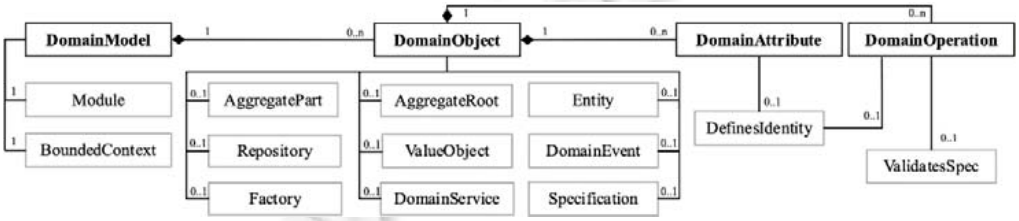


图 4 领域建模规范语言的概念元模型

构造块是领域建模的重要概念, 每个构造块都描述了针对特定领域问题的建模和程序设计思路. 在领域建模时, 通常根据不同领域问题的特征将其建模为特定的构造块, 在创建领域模型和程序设计时, 都使用这

些构造块标记模型/代码元素以表达领域模型语义. 比如, 实体(entity)构造块可用于建模必须与同类领域对象区分开来的领域对象, 一个例子是需要区分的不同软件发布版本. 本文提出的概念元模型正是以构造块为主体, 所关注的构造块列表(图 4 灰体部分)结合了文献[1,19,28]等工作.

下面对这些构造块概念(用加粗斜体表示)展开详细介绍.

- 实体: 由标识定义而非其属性定义的领域对象, ***Entity*** 应具有连续的生命周期. 由于实体有且只有一个标识, 定义 ***DefinesIdentity*** 以表示标识属性和定义标识的方法从而确定其唯一性.
- 值对象(value object): 仅用于描述领域特征而本身没有标识的领域对象, ***ValueObject*** 应具有不可变性.
- 聚合(aggregate): 聚合由一组实体和值对象组成, 聚合内的所有领域对象在对象更改时具有一致性. 每个聚合都有且只有一个实体被定义为聚合根(***AggregateRoot***), 外部对象只能通过聚合根来控制对聚合内部领域对象的访问. 聚合内除聚合根以外的其他领域对象称为聚合部分(***AggregatePart***).
- 服务(service): 与领域概念相关的操作被建模为 ***DomainService***, 且这些操作不应被建模为实体或值对象的职责.
- 领域事件(domain event): 表示领域中所发生事件的领域对象, 将领域内发生的活动建模成一系列离散事件, 被建模的 ***DomainEvent*** 应是领域专家关心的. 由于领域事件有且只有一个标识, 基于 ***DefinesIdentity*** 方法确定事件的唯一标识.
- 资源库(repository): 为需要全局访问的对象类型创建的领域对象, ***Repository*** 负责该类型的所有对象的存储与访问操作.
- 工厂(factory): 用于封装复杂对象创建行为的领域对象, ***Facorty*** 负责该类型的所有对象的创建操作.
- 规格(***Specification***): 描述其他领域对象应具有的约束的领域对象, 用于确定领域对象是否符合某些标准. 规格对象利用其 ***ValidateSpec*** 方法进行约束验证.
- 模块(***Module***): 用于对领域模型进行划分, 使得关系紧密的模型元素属于同一模块, 而模块之间的元素具有低耦合关系.
- 限界上下文: 复杂项目存在多个领域模型, 而 ***BoundedContext*** 用于显式地定义模型边界, 以保持模型的完整性.

图 4 还展示了构造块与领域建模关键元素间的关系. 其中, 实体、值对象等构造块用于建模领域对象, ***DefinesIdentity*** 和 ***ValidateSpec*** 等概念用于建模领域对象的属性和操作, 而模块、限界上下文等构造块提供了领域模型的封装. 注意到, 领域建模的关键元素仅用于组织构造块, 而不用于实际领域建模以避免信息冗余. 比如, 一个领域问题被建模为实体已经表明了其领域对象特征, 而无须再将其显式地标记为“领域对象”.

此外, DMSL 的概念元模型还关注构造块之间的关系. 不同构造块之间的关系被定义为语法约束(constraint)以规范建模行为(见表 1), 从而使基于 DMSL 建模得到的领域模型满足这些构造块间关系约束. 比如, 领域建模原则“实体在其生命周期内具有唯一标识”, 对应着语法约束 C1“Entity 有且只有一个被 DefinesIdentity 标记的标识符”, 任何使用 DMSL 建模得到的有效模型都应遵从该约束. 然而, 针对同一领域建模原则, 不同开发团队可能具有不同的实践方式(比如是否严格遵循该原则), 这使得不同开发团队可能规定了不同的构造块间关系, 比如“实体能否独立于聚合存在” “限界上下文是否允许被嵌套”等就取决于开发团队的实践. 为此, DMSL 仅选用了较通用的构造块约束集, 通过焦点小组(focus group)讨论领域建模原则[1,2]以获得.

接下来展示代码及各阶段模型所遵循的元模型, 即 DMSL 的具体元模型. 每个具体元模型都是概念元模型在相应抽象层级的具体表现形式, 与概念元模型有相同的领域概念及概念间关系. 考虑到 L3 层模型所遵循的元模型是 L2 层的子集, 因此将二者的元模型共同展示.

表 1 定义自文献[1,2]的领域建模规范语言的约束

构造块	约束描述
Entity	C1. Entity 有且只有一个被 DefinesIdentity 标记的标识符
DefinesIdentity	C2. 必须在 Entity 或 DomainEvent 中使用
AggregateRoot	C3. AggregateRoot 必须是 Entity 类型
AggregatePart	C4. AggregatePart 必须是 Entity 或 ValueObject 类型 C5. AggregatePart 必须指定其根, 且根为 AggregateRoot 类型
DomainService	C6. DomainService 只定义操作 C7. 被建模为 DomainService 的领域对象不能被建模为其他类型
DomainEvent	C8. DomainEvent 有且只有一个被 DefinesIdentity 标记的标识符
Repository	C9. Repository 只定义操作 C10. Repository 操作的返回值类型必须是 AggregateRoot 对象 C11. 被建模为 Repository 的领域对象不能被建模为其他类型
Factory	C12. 被建模为 Factory 的领域对象不能被建模为其他类型
Specification	C13. 规格至少有一种 ValdateSpec 方法
ValidateSpec	C14. ValdateSpec 方法必须以一个具体领域对象作为参数 C15. ValdateSpec 方法必须返回布尔值类型 C16. 只能在 Specification 中使用

3.2 L0-代码元模型

领域驱动设计提倡使用面向对象的编程语言(object-oriented programming language, OOPL)进行程序设计. 图 5 描述了 L0 层的代码所需要遵循的编码规范, 它是概念元模型在 OOPL 下的表现形式, 通过将概念元模型映射到 OOPL 元模型获得. 其中, OOPL 元模型如黑体部分所示. 注解(annotation)[29]技术为多种 OOPL (包括 Java 5[30]以上版本和 C#[31])所支持, 它通过为代码元素附加元数据以帮助编译器理解和操作程序结构. 在本文关注的 OOPL 元模型中, 注解可用于修饰 4 种非注解代码元素, 包括类(class)、字段(field)、方法(method)以及命名空间(namespace). 每个注解还包含多个属性(property)以提供更丰富的注解行为. 值得注意的是, 类概念包含类及接口(interface), 后者可以表示为无字段的类. 图 5 展示的 OOPL 元模型是完整 OOPL 元模型的子集, 它省略了描述 OOPL 方法体结构的元概念等内容, 这是由于本文关注领域模型结构, 而不是具体的代码实现.

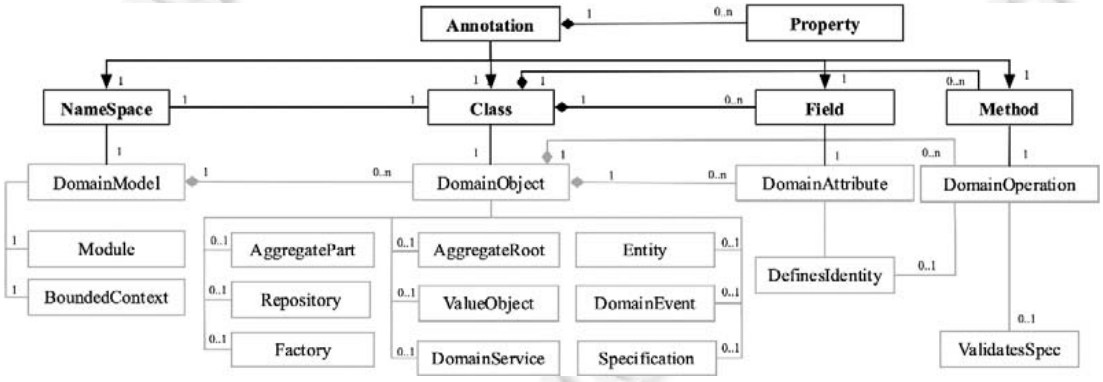


图 5 代码元模型: 简化的 OOPL 元模型及其与概念元模型的对应关系

图 5 还说明了 OOPL 元模型与概念元模型的关系. 这种关系是显而易见的, 领域对象 DomainObject、领域对象的属性 DomainAttribute 及其操作 DomainOperation 在代码层面分别表现为类、字段及方法. 此外, 开发人员通常以命名空间(比如 Java 中的 package)形式组织领域模型, 因此命名空间被映射为领域模型 DomainModel. OOPL 元模型与概念元模型的关系表明, 代码元素(比如类 Class)可用于表达对应的领域模型概念(比如领域对象 DomainObject). 而更具体的代码元素所表达的领域模型语义(即其对应的构造块)则依赖注解表现出来. 概念元模型中的所有构造块(比如实体)都被定义为 OOPL 的注解以修饰对应的代码元素(比如类), 从而使代码元素显式地表达领域模型概念, 即该代码元素被建模为对应的构造块(比如该类被建模为实

体类型的领域对象). 在使用构造块定义的注解修饰代码元素时, 还应遵从概念元模型定义的构造块间关系约束.

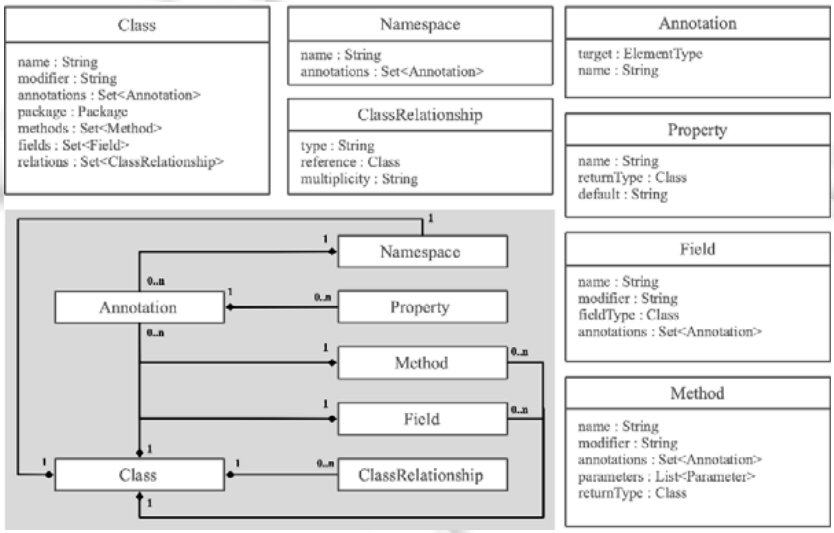
本文所定义的注解仅用于修饰相应代码元素的声明, 以实体为例(如图 6 所示), Release 实体对应的注解@Entity 仅被用于修饰 Release 类的声明, 且并不在 Release 类的 product 字段中对 Product 实体进行重复修饰. 此外, 实体最重要的特征是其生命周期内的唯一标识, 比如图 6 中 Release 实体的 releaseID 属性, 该属性被注解@DefinesIdentity 所修饰.

```
1.  package com.saasovation.agilepm.release;
2.  @AggregateRoot
3.  @Entity
4.  public class Release {
5.  @DefinesIdentity
6.  String releaseID;
7.    BacklogItem[·] backlogItem;
8.    Product product;
9.    RelaseRepository relaseRepository;
10.     public ScheduleBacklogtem schedule(·){
11.   …
12.     }
13. }
```

图 6 被注解修饰的 OOPL 代码(以 Java 为例)

3.3 L1-PSM元模型

PSM 模型来自对代码的静态分析. 每个代码文件都经静态分析被转换成一个 PSM 模型, 遵循图 7 所示的元模型. 其中, 图 7(a)展示了 PSM 模型的元模型; 图 7(b)展示了元模型中所有关键类的详细设计, 包括每个类持有的属性和每个属性的数据结构类型.



(a) 阴影部分, (b) 其他

图 7 PSM 元模型及其关键类的具体设计

由于 PSM 模型和代码有着直接对应关系, PSM 元模型和 L0 层代码元模型具有几乎一致的元概念(如图 7(a)所示), 包括注解、类、字段、方法和命名空间等, 并且注解与其他 4 个非注解元概念的关系与第 3.2 节一致. 类关系(ClassRelationship)关注每个类和其他类之间的关系, 这些关系信息也通过静态分析代码获得, 比如某个类的字段类型是另一个类体现了两个类间的关联关系. 注意到 PSM 元模型与概念元模型的联系和 L0 层保持一致, 包括每种 PSM 的元概念(比如类)与构造块(比如实体)的对应关系, 因此图 7 未赘述二者联系.

在具体设计方面(如图 7(b)所示), 每个 PSM 模型的 Field 对应 L0 层代码中每个类文件的字段信息, 包括字段名称 name、限定符 modifier(比如 public, private 等)、字段类型 fieldType 以及修饰该字段的注解 annotations 等. Method 对应代码中每个类文件的方法信息, 包括方法名称、限定符、修饰该方法的注解、方法参数 parameters 以及返回值 returnType 等. 类关系 ClassRelationship 关注每个类和其他类之间的关系, 具体包括关联和依赖两种关系类型(type). 除此之外, 由于类关系作为关系持有类本身的属性存在, 因此仅存储与该类关系相关的另一类 reference 以及该类关系的多重性特征 multiplicity(比如 0…1, 0…*等). Annotation 对应着修饰非注解代码元素的注解信息, 包括注解修饰的代码元素类型 target(比如类、字段、方法及包)以及注解名 name 等. 其中, 注解名由 L0 层基于构造块定义的注解(见第 3.2 节)转换而来, 与概念元模型的构造块内容保持一致, 包括@Entity, @ValueObject 等.

3.4 L2/3-PIM/CIM元模型

图 8 描述了 PIM 元模型, 即概念元模型在 UML 下的表现形式, 通过将概念元模型映射到 UML 元模型得到. 其中, UML 元模型如黑体部分所示. 选取 UML 作为领域模型基础的原因在于: (1) UML 具有较小的认知复杂度, 更易于领域专家理解[19]; (2) UML 正被广泛地应用于软件开发活动, 具有训练有素的用户基础; (3) 作为一种通用的建模语言, UML 具有更完善的工具支持. 此外, UML 的 Profile 机制还支持自定义构造型(stereotype)来修饰模型元素, 从而使模型元素直接表达特定语义. 在本文关注的 UML 元模型中, 构造型可用于标记 4 种模型元素, 包括包(package)、类(class)、属性(property)和操作(operation)等. 同时, UML 还通过标记值(tag)定义构造型的属性以丰富构造型语义. Relationship 用于描述多个类之间的关系, 本文关注的类间关系包括关联(association)和依赖(dependency)两种.

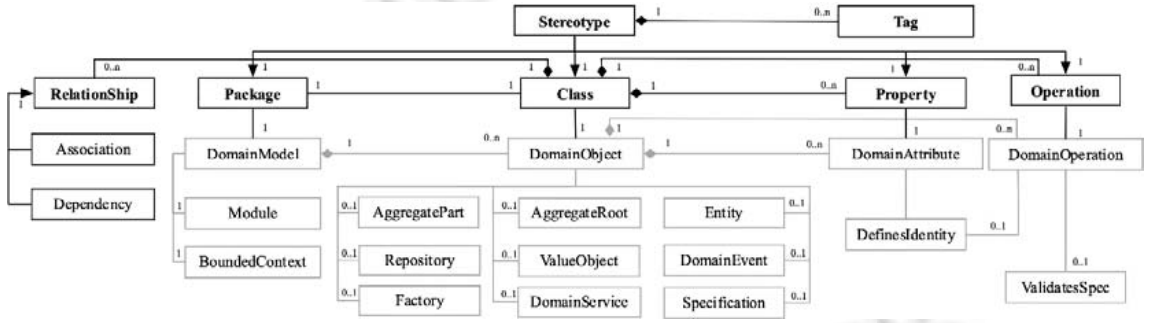


图 8 PIM 元模型: 简化的 UML 元模型及其与概念元模型的对应关系

图 8 还说明了 UML 元模型与概念元模型的映射关系. 在领域建模时, 开发人员通常依赖于结构化的类图(class diagram)来表达领域模型. 本文涉及的类图元概念包括类、属性、操作和类间关系(relationship), 分别表示领域对象 DomainObject、领域对象的属性 DomainAttribute、操作 DomainOperation 及对象间关系. 此外, UML 中的包图(package diagram)可用于对领域对象进行分组以组织领域模型, 因此, 包被映射为领域模型概念 DomainModel. 而更具体的模型元素所表达的领域模型语义(即其对应的构造块)则依赖构造型表现出来. 比如, 定义《Entity》构造型来标记类, 从而显式地说明该类被建模为实体类型的领域对象; 定义《BoundedContext》构造型来标记包, 从而显式地说明该包内所有领域对象属于同一领域模型, 以区分不同领域模型. 在 PIM 模型中, 由构造块定义的构造型间关系也应遵从概念元模型定义的构造块间关系约束.

在转换得到的 PIM 模型中, 用于标记类、属性及操作的构造型被标记于对应模型元素前, 而用于标记包的构造型被标记于包的上方. 图 9 展示了一个 PIM 模型示例(仅包含一个领域对象), 对应于第 3.2 节的示例. CIM 模型仅显示领域对象、领域对象的属性及对象间关系, 而不涉及具体的计算细节. 相比于 PIM 元模型, CIM 元模型缺少操作细节, 即 CIM 模型图对应的第 3 个分栏(compartment)为空.

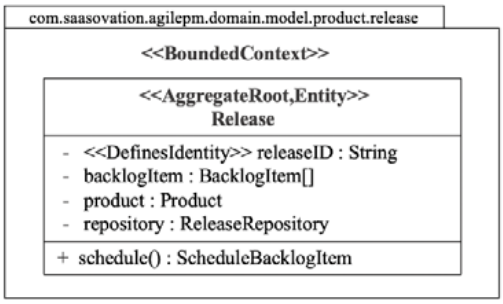


图 9 PIM 模型示例

4 代码到模型的自动化转换

本节首先给出代码到模型转换的映射规则, 而后讨论关键的转换步骤, 包括代码到 PSM 模型(L0 到 L1)转换以及 PSM 到 PIM 模型(L1 到 L2)转换. 由于 CIM(L3)元模型是 PIM(L2)元模型的子集, 从 PIM 到 CIM 模型的转换方法 CIMGEN 较为直接, 因此不对其做深入讨论.

4.1 代码到模型的映射规则

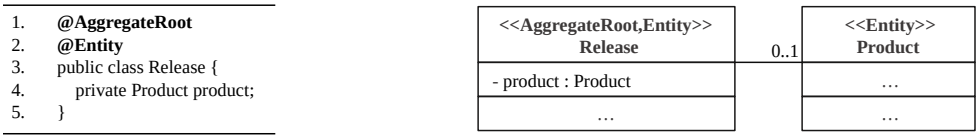
表 2 展示了 C2MD 方法的部分代码到模型转换规则. 这些规则描述了 C2MD 转换过程的输入(OOPL 代码)与输出(基于 UML 的 PIM/CIM 模型)之间的映射, 都使用对应状态下的元概念来表达. 首先, 规则 1–规则 4 描述了 OOPL 的命名空间、类、字段和方法分别被转换为 PIM/CIM 模型的包、类、属性和操作; 其次, 领域模型元素被建模为特定构造块, 在 OOPL 中表现为代码元素被由构造块定义的注解所修饰, 而在 PIM/CIM 模型中表现为模型元素被由构造块定义的构造型所修饰(规则 5); 并且注解的属性 Property 被转换为构造型的属性 Tag(规则 6); 规则 7–规则 9 表示 OOPL 中类字段和方法所体现的两个类(领域对象)的相关关系被转换为 PIM/CIM 模型的类间关系, 包括关联和依赖两种. 比如, 规则 7 描述了当领域对象 *m* 所在类的某属性类型为领域对象 *n* 所在类时, 领域对象 *m* 和 *n* 之间存在关联关系.

表 2 C2MD 方法的代码到模型转换规则

规则	代码元素	模型元素
1	Namespace	Package
2	Class	Class
3	Field	Property
4	Method	Operation
5	Annotation	Stereotype
6	Property	Tag
7	Field, where fieldType is Class with predefined annotations	Attribute, Association
8	Method, where returnType is Class with predefined annotations	Operation, Dependency
9	Method, where one parameter is Class with predefined annotations	Operation, Dependency

接下来, 以规则 7 为例详细说明 C2MD 的代码到模型转换规则. 除了遵循 DMSL 定义的模型规范外, OOPL 代码的具体语法表示以 Java 为例, 遵循 Java 14[30]规范; PIM/CIM 模型的具体语法表示遵循 OMG UML 2.4.1[25]规范. 在领域对象 Release 对应的 Java 文件中(如图 10(a)所示), product 字段属于 Product 类型, 且 Product 本身也被建模为一个领域对象. 因此, 在代码转换为模型时, 除了类 Release 的 product 属性外, 类 Release 和 Product 间还存在一条关联连线, 同时标注了该关联的多重性特征(如图 10(b)所示).

本文提出的 C2MD 代码到模型转换方法面向领域建模过程, 以领域模型的代码实现结果为输入, 转换得到 PIM/CIM 模型以支持知识消化过程. 这意味着 C2MD 方法仅关注标注了构造块注解的 OOPL 类文件(也就是定义领域对象的代码文件), 而忽略其他技术相关代码, 比如实现数据库存储的代码.



(a) 代码 (b) 模型

图 10 代码到模型转换的映射规则 7 示例

4.2 L0-代码到L1-PSM的转换

算法 1 展示了代码到 PSM 模型转换的关键方法 PSMGEN, 该方法以静态分析每个领域对象对应代码文件得到的抽象语法树为输入, 生成该领域对象对应的 PSM 模型. 为便于理解, 算法 1 在关键位置给出了代码注释以解释算法.

算法 1. PSMGEN.

输入: c: 领域对象的 AST 信息.

输出: m: PSM.

1: $n_n \Leftarrow c.namespace$, $n_a \Leftarrow c.annotation$, $n_f \Leftarrow c.field$, $n_m \Leftarrow c.method$, $n_c \Leftarrow c$, $n_r \Leftarrow \{\}$ //初始化①

2: $m_c, m_n, m_a, m_f, m_m, m_r \Leftarrow \{\}$ //初始化 PSM 模型

3: Const k_n, k_c, k_m, k_f //定义常量, 包括 k_n 命名空间注解、k_c 类注解、k_m 方法注解、k_f 字段注解

4: *addNamespace*(n_n) //步骤 1. 从 AST 信息创建 PSM 的 *Namespace*

5: **for** all f in n_a **do**
addAnnotation(f) //步骤 2. 从 AST 信息创建 PSM 的 *Annotation*

6: **for** all f in n_f **do**
addField(f) //步骤 3. 从 AST 信息创建 PSM 的 *Field*

7: **for** all f in n_m **do**
addMethod(f) //步骤 4. 从 AST 信息创建 PSM 的 *Method*

8: *addClassRelationship*(n_c) //步骤 5. 从 AST 信息创建 PSM 的 *ClassRelationship*

9: **Return** m

10: Function *addField*(*Field f*): //创建 PSM 模型的 *Field* 对象

11: *newField=create Field*()

12: *newField.name*⇐*f.name*, *newField.modifier*⇐*f.modifier*, *newField.fieldType*⇐*f.type*

13: *newField.annotations*⇐$\{a|a:f.annotation, a.name \in k_f\}$ //过滤其他框架的注解②

14: m_f.*add*(*newField*) //③

15: Function *addClassRelationship*(*Class c*): //创建 PSM 模型的 *ClassRelationship* 对象

16: **for** all f in *c.field.fieldType* **do**: //扫描领域对象的所有字段以添加类间关联

17: **If** ($\exists f.annotations.name \in k_c$) m_r.*add*(*create ClassAssociation*(*f*)) //②③

18: **for** all f in *c.method.returntype*, *c.method.parameters* **do**: //扫描领域对象的所有方法返回值、参数以添加类间依赖

19: **If** ($\exists f.annotations.name \in k_c$) m_r.*add*(*create ClassDependency*(*f*)) //②③

在算法 1 的初始化部分, 第 1 行对应着基于抽象语法树初始化以获得 PSM 模型的输入数据, 包括对应代码文件的命名空间 n_n、注解 n_a、字段 n_f、方法 n_m、类 n_c 和类关系 n_r. 这是因为静态分析得到的初始抽象语法树包含代码行数、语法符号所在位置等本工作不涉及的信息. 第 2 行用于初始化 PSM 模型, 对应着模型的类 m_c、命名空间 m_n、注解 m_a、字段 m_f、方法 m_m 和类关系 m_r. 第 3 行通过定义常量规定了本工作关注的注解列表(即由构造块定义而来的注解, 见第 3.2 节), 包括 k_n 命名空间注解、k_c 类注解、k_m 方法注解以及 k_f 字段注解

等. 这意味着代码中的其他注解将被移除, 比如数据库注解 JPA(Java persistence API)[32]的@ManyToOne, @ManyToMany 等. 算法 1 的具体实现部分共包括 5 个主要步骤, 分别对应着从抽象语法树 AST 创建 PSM 模型的命名空间 *Namespace*、注解 *Annotation*、字段 *Field*、方法 *Method* 和类关系 *ClassRelationship*(步骤 1 到步骤 5). 以步骤 3 为例, 领域对象的每个字段 *f* 都通过 *addField* 方法(第 10 行–第 14 行)转换为 PSM 模型的一个字段 *Field* 对象, 创建 *Field* 对象的所需字段如第 3.3 节. 此外, 在字段 *f* 的所有注解中, 只有字段注解常量 k_f 定义的注解才会被存入 *Field* 对象(第 13 行). 在创建类关系 *ClassRelationship* 对象时(第 15 行–第 19 行), 领域对象 *c*1 所在类的所有字段和方法都被逐一扫描以检查类间关系, 当仅当 *c*1 的字段类型、方法返回值或参数所属类型的类声明中(对应 *c*2)包含类注解常量 k_c(即 *c*2 也是领域对象), 才会创建 *c*1 和 *c*2 类间关系. 其间存在的类间关系包括关联和依赖两种, 如第 17 行和第 19 行.

算法 1 中标注①–标注③的部分反映了从领域对象的抽象语法树到 PSM 模型的信息过滤. 具体来说, 标注①对应着过滤抽象语法树中本工作不涉及的信息, 比如代码行数等; 标注②对应着过滤代码中其他框架提供的注解信息, 比如数据库注解等; 标注③表示 PSM 模型只关注其元模型定义的模型内容(如第 3.3 节).

4.3 L1-PSM到L2-PIM的转换

算法 2 展示了 PSM 到 PIM 模型转换的关键方法 PIMGEN, 该方法以 L1 层的所有 PSM 模型为输入, 输出一个 PIM 模型. PIMGEN 共包含 3 个关键步骤, 分别对应着基于 PSM 模型生成该领域对象对应的类、基于 PSM 的类关系生成 PIM 的类间关系以及模型初始化操作(步骤 1–步骤 3). 在步骤 1 中, 每个领域对象对应的 PSM 模型都被转换为 PIM 模型的一个类, 包括类名、所在包、属性及操作等. 由于每个领域对象存在多个字段, 因此通过 for 循环遍历所有字段, 以提取每个字段的名称和类型, 从而创建对应的 PIM 模型属性(第 3 行). 在步骤 2 中, 每个 PSM 模型的每个类关系都被转换成 PIM 的一个类间关系, 包括该关系的类型(关联或依赖)、对应的类名称和多重性. 步骤 3 首先基于步骤 1 和步骤 2 得到的类与类间关系生成模型, 而后根据每个类的所在包组织模型.

算法 2. PIMGEN.

输入: *ms*: PSMs.

输出: *p*: PIM.

//步骤 1. 创建类

1. **for** all *m* in *ms* **do** //将 L1 层的所有 PSM 模型生成类
2. $item \Leftarrow item+m_{namespace}+m_{annotations}+m_{class}$ //从 PSM 的类创建 PIM 的类①
3. **for** all *f* in m_{field} **do**
   $item \Leftarrow item+f.annotations+f.modifier+f.name+f.fieldType$ //从 PSM 的字段创建 PIM 的属性②
4. **for** all *f* in m_{method} **do**
   $item \Leftarrow item+f.annotations+f.modifier+f.name+f.parameters$ //从 PSM 的方法创建 PIM 的操作③

//步骤 2. 创建类间关系

5. **for** all *m* in *ms* **do**
   for all *r* in $m_{relations}$ **do** //从 PSM 创建 PIM 的类间关系
   $relation \Leftarrow relation+m_c$.name+*r.reference.type*+*r.reference.name*+*r.reference.multiplicity*

//步骤 3. 生成模型

6. $p \Leftarrow createModel(item, relation)$
7. *decompose*(*p*) //根据类所在包将模型分组④
8. **return** *p*

从 PSM 转换为 PIM 模型, 构造块在领域模型中的存在形式也由注解转换为构造型. 步骤 1 中, PSM 模型的类、字段、方法的注解分别被转化为 PIM 模型的类、属性和操作的构造型, 对应着标注①–标注③. 在步骤 3 中, 命名空间注解被转化为包的构造型, 包内所有类共享一个包构造型(也就是领域模型), 对应着标注④.

5 工具原型

为了提高代码到模型转换的效率，本文实现了 C2MD 原型(https://github.com/thorgits/DddDiagram)，从而为领域专家和开发人员进行领域建模提供自动化工具支持．该工具原型面向 Java 语言，能够将以 Java 实现的领域模型代码转换为 PIM/CIM 模型．C2MD 原型以 IntelliJ Platform (https://www.jetbrains.com)插件的形式存在，支持开发人员通过添加插件以使用．图 11 展示了 C2MD 的图形化用户接口．开发人员在项目结构图展示的包上点击右键，再点击子菜单栏“DDD Graph”的任一项(“PIM”和“CIM”)即可生成对应领域模型．

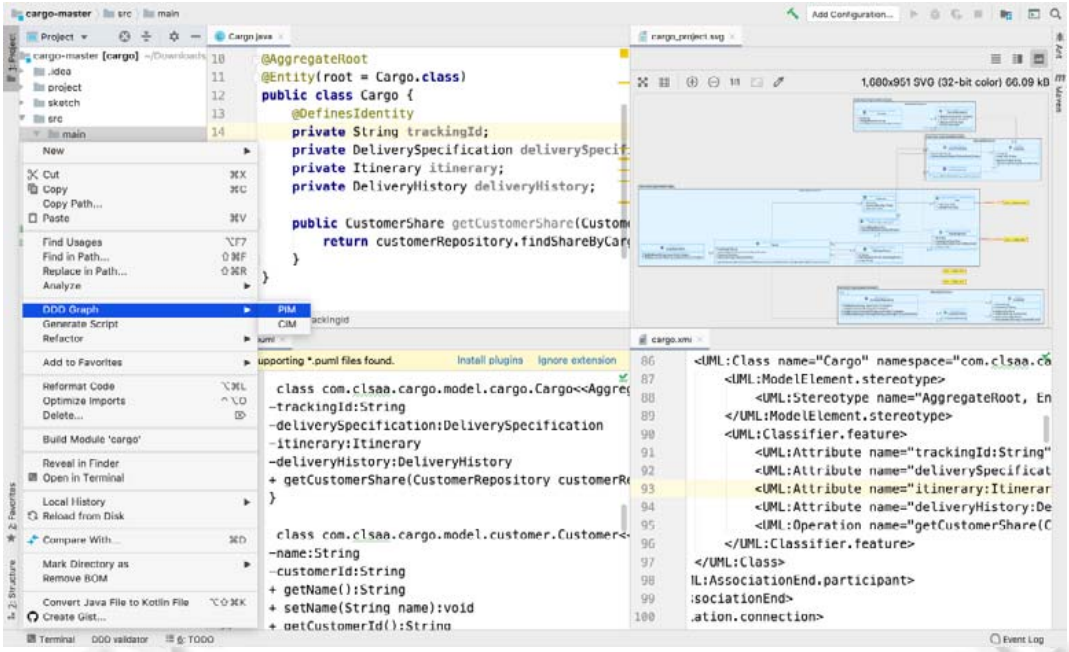


图 11 C2MD 原型界面

C2MD 的实现包括 DMSL 的实例化和代码到模型的转换过程．其中，DMSL 的实例化利用了 Java 提供的注解技术、面向 PSM 定义的数据结构和 UML 提供的 Profile 扩展机制等．代码到模型的转换过程基于 IntelliJ Platform 的程序结构接口(program structure interface)，后者提供了语法分析器 PsiParser 以静态分析 Java 代码得到抽象语法树．此外，C2MD 还集成了 Java 实现的 PSMGEN/PIMGEN/CIMGEN 这 3 部分功能．在当前实现版本中，C2MD 最终生成的 PIM 及 CIM 模型都具有 3 种形式，包括图形表示(svg 格式)以及两种文本表示(puml 和 XML 元数据交换格式(XML metadata interchange, XMI)[33])．图形表示依赖于开源的文本到 UML 转换工具 PlantUML (https://plantuml.com)生成，能够展示可视化的领域模型，而文本表示形式能够支持 C2MD 与其他建模工具集成．图 11 左上的代码编辑器展示了 Release 类的 Java 代码(与第 3.2 节示例一致)，右上、左下和右下都对应着 C2MD 原型输出的 PIM 模型，包括图形表示(svg 格式)和文本表示(puml 以及 XMI 格式)．

6 案例研究及分析

本节通过案例研究，来探究所提出的代码到模型转换方法 C2MD 能否支持真实软件项目的领域建模过程．

6.1 案例研究设计

案例研究(case study)[34,35]是一种经验主义的研究，“旨在探究自然环境中的暂时现象”．这适用于问题边界不确定的软件开发活动，因此被广泛地应用于软件工程研究．本文所使用的案例研究方法是解释性的

(exploratory), 其目的在于验证现有理论, 即探究所提出的领域建模支持工具 C2MD 在真实软件开发中的适用性. 具体来说, 本案例研究旨在探究 C2MD 是否具有以下两个好处: 其一, 通过可视化地对比转换得到的领域模型与初始领域模型, 能够更快地发现二者的分歧, 从而避免不必要的程序设计与领域模型的偏离; 其二, 围绕转换得到的领域模型展开知识消化, 能够减少该过程对代码实现细节的依赖, 从而减小领域专家的认知复杂度、加速知识消化过程. 注意到, 本案例研究所要探究的这两个方面都针对领域建模过程, 并且独立于特定软件项目的领域业务逻辑, 因此, 本案例研究得到的结论同样适用于其他软件项目.

C2MD 方法关注单个领域模型的精炼过程, 因此本案例研究将 C2MD 应用于精炼单个领域模型, 包括创建领域模型、实现领域模型代码、代码到模型转换以及知识消化, 其分析以领域模型为最小单元.

6.2 案例选择

本文选择领域驱动设计的经典项目——货物追踪系统(cargo tracking system, CTS)[1]作为研究案例, 原因在于: (1) CTS 面向物流领域, 具有合适的领域复杂度; (2) CTS 具有合适的问题规模, 便于阐述; (3) 领域驱动设计的多个现有研究[19,36]选择 CTS 作为问题分析场景, 提供了丰富的文档资料.

作为一个货物追踪系统, CTS 最核心的业务过程在于追踪货物(cargo)从位置(location)*A* 运送到 Location *B* 的各个状态. 每件 Cargo 在创建时都被赋予一个追踪号(trackingId)以跟踪该 Cargo 状态. 同时, 每件 Cargo 都对应着一个航线说明(DeliverySpecification), 确定了该 Cargo 的出发 Location 和目的 Location. 每件 Cargo 还对应着一个包含多个航段(leg)的航线(itinerary), 根据现有航程(voyage)为其分配. 其中, Voyage 是指货轮等运输工具的某一调度好的航行路线, 包含一系列运输动作(CarrierMovement), Cargo 正是通过这一系列运输动作在不同地点之间转移. 在 Cargo 的航线确定之后, 处理事件(HandlingEvent)负责对 Cargo 采取不同动作, 比如装货、卸货等. Cargo 执行的所有 HandlingEvent 都将存入该 Cargo 的运送历史(DilveryHistory)中. DeliveryHistory 反映了 Cargo 从创建后经历的各个状态, 这与 DeliverySpecification 恰好相对, 后者描述了目标. 此外, 每个 Cargo 都涉及多个客户(customer), 每个 Customer 承担不同的角色, 比如收货人、付款人等.

6.3 数据收集与分析

由于 C2MD 方法关注领域建模过程, 本案例研究的数据收集与分析自然地与需求收集与分析相吻合. CTS 的需求来源于已发表文献对 CTS 的需求描述, 包括文献[1,19,36]. 考虑到需求数据本身的定性特点, 本文采用文档审查(document review)方式收集并整理 CTS 系统的用例和业务规则, 整理得到 CTS 系统的详细需求见第 6.2 节及在线代码仓(https://github.com/clsaa/cargo).

对 CTS 需求的分析即借助 C2MD 进行领域建模, 包括创建领域模型、实现领域模型代码、代码到模型转换以及知识消化这 4 个步骤. 该过程由 3 个研究人员参与, 包括 1 个领域专家和 2 个开发人员(都具有 2 年以上实际研发经验). 其中, 代码实现领域模型主要由开发人员展开, 代码到模型转换依赖于自动化的 C2MD 工具, 而创建领域模型和知识消化过程涉及所有研究人员. 根据领域驱动设计原则[1,2], 领域建模的这 4 个步骤应迭代进行, 直至得到满意的领域模型. 注意到, 本案例研究所探究的两个方面都在知识消化过程展开, 而该过程是以讨论形式进行的, 因此能够一定程度上缓解单个研究人员的主观性差异对案例研究结果的影响.

6.4 结　果

借助 C2MD 工具进行领域建模得到的中间产物包括领域模型和代码, 前者又包括创建所得的领域模型(初始领域模型)和由代码转换而来的 PIM/CIM 模型. 本节首先讨论针对整个 CTS 的初始领域建模结果(包括 4 个领域模型), 而后以 Cargo 领域模型的一次迭代为例讨论 C2MD 如何支持单个领域模型的建模过程, 包括 4 个具体步骤.

6.4.1 CTS 的初始领域建模结果

图 12 展示了针对 CTS 需求创建的初始领域模型, 共包括 Cargo, Location, Voyage 以及 Customer 这 4 个限界上下文, 每个限界上下文都对应着一个独立且完整的领域模型. Cargo 领域模型负责管理货物, 包括货物的计划路线和实际航线等; Voyage 领域模型负责管理各类运输工具的航行路线; Location 领域模型管理物流运输

中的位置信息; 而 Customer 领域模型管理每个货物的相关客户信息. 4 个领域模型协同合作以实现 CTS 的系统需求.

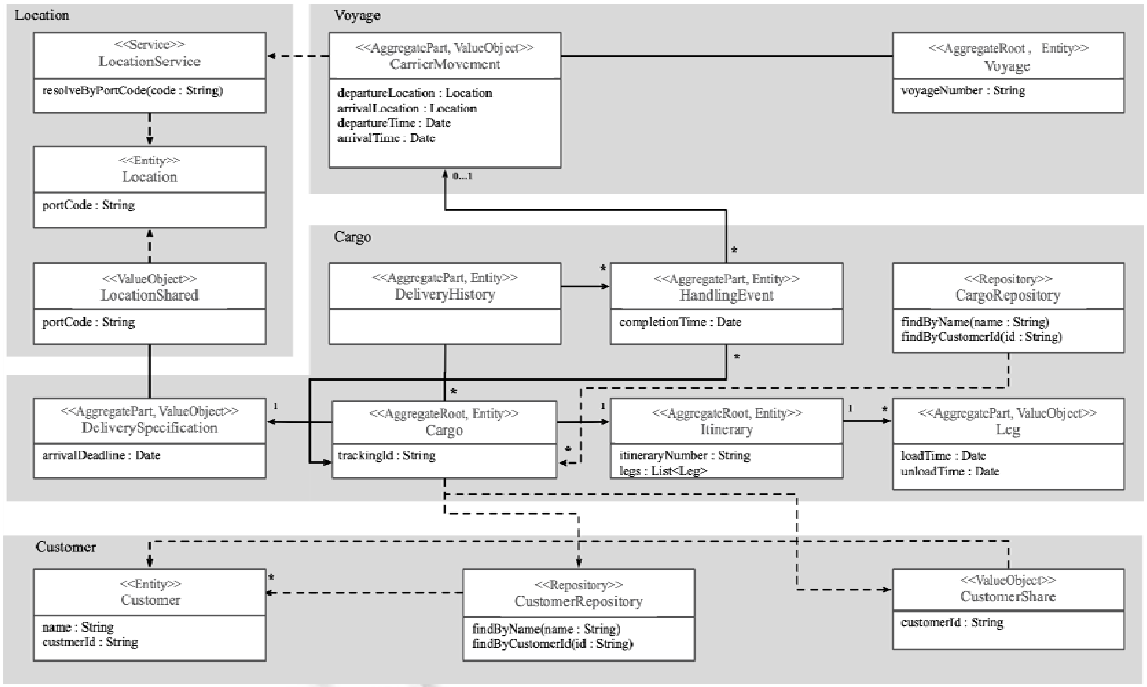


图 12 CTS 的初始领域建模结果

6.4.2 利用 C2MD 进行领域建模: 以 Cargo 领域模型为例

1) 创建领域模型

如图 12 所示, 初始 Cargo 领域模型包含 7 个领域对象. 其中, 实体 Cargo 以 trackingID 为标识, 在其生命周期内具有唯一性. Cargo 的资源库 CargoRepository 封装了所有获取 Cargo 引用所需的逻辑, 其他领域对象都经由 CargoRepository 获取 Cargo 对象. 航线说明(DeliverySpecification)和运送历史(DeliveryHistory)分别管理 Cargo 的计划航线和实际航线信息. 一个 DeliverySpecification 可以被多个 Cargo 共同使用, 因此被建模为值对象; 而 DeliveryHistory 追踪的航线信息较为复杂, 且与 Cargo 具有一对一关系, 因此被建模为实体. 具体航线(itinerary)的确定需要满足航线说明, 并且选择航段(leg)总规模最小的航线. Itinerary 与 Leg 间由于存在整体与部分关系而被建模为聚合. 此外, 每个 Cargo 的运送历史对应着多个处理事件(HandlingEvent).

2) 实现领域模型代码

得到初始领域模型后, 开发人员基于此实现对应的代码, 初始 Cargo 领域模型中每个领域对象都被实现为一个 OOPL 类文件(以 Java 为例). 由于篇幅限制, 图 13 仅给出了 Cargo 领域对象的代码实现, Cargo 领域模型中其他领域对象的代码实现见代码仓(https://github.com/clsaa/cargo). 根据初始 Cargo 领域模型(如图 12 所示), 和 Cargo 存在关系的领域对象包括 DeliverySpecification, DeliveryHistory, Itinerary, HandlingEvent 和 CargoRepository. 其中, Cargo 关联的领域对象包括 DeliverySpecification, Itinerary, DeliveryHistory. 因此, 这 3 个领域对象以 Cargo 类属性的形式出现在 Cargo 类声明中. 此外, Cargo 类被建模为实体与聚合根, 对应地 Cargo 类声明部分添加了注解《Entity》及《AggregateRoot》, 同时, Cargo 标识 trackingId 也被注解《DefinesIdentity》所修饰. 这与图 12 中 Cargo 领域模型相一致.

```
1.  package com.clsaa.cargo.model.cargo;
2.  @AggregateRoot
3.  @Entity
4.  public class Cargo {
5.  @DefinesIdentity
6.      private String trackingId;
7.  private DeliverySpecification deliverySpecification;
8.      private Itinerary itinerary;
9.      private DeliveryHistory deliveryHistory;
10.     public CustomerShare getCustomerShare(CustomerRepository customerRepository, String trackingID) {
11.     }
12. }
```

图 13　Cargo 领域类的原型实现

3) 代码到模型转换

利用 C2MD 工具将 Cargo 领域模型的代码实现转换为 PIM/CIM 模型, PIM 模型如图 14 所示, CIM 模型见代码仓. 与代码相比, PIM 模型中 CargoRepository 仍然封装所有获取 Cargo 引用所需的逻辑. 3 个领域对象 DeliverySpecification, Itinerary, DeliveryHistory 作为 Cargo 类属性出现在 Cargo 类声明中, 对应地, PIM 模型中 Cargo 具有与此 3 个领域对象的关联关系. 同时, Itigerary 与 Leg 的聚合关系仍保持不变.

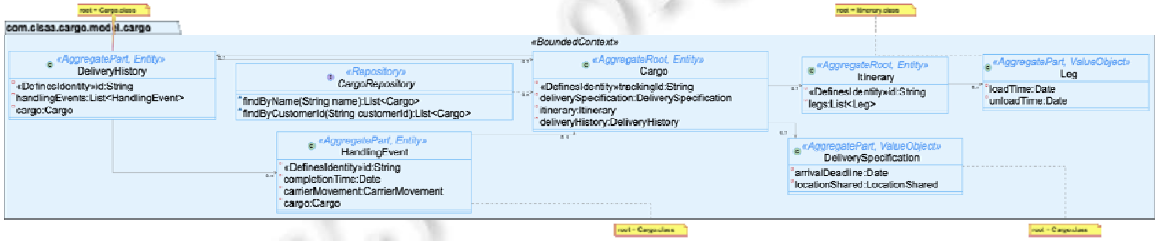


图 14　由 Cargo 领域模型的代码实现转换得到的 PIM 模型

4) 知识消化

利用 C2MD 实现 Cargo 领域模型的代码到模型转换后, 开发人员和领域专家开始围绕初始领域模型和 PIM/CIM 模型展开知识消化. 通过对比图 14 和图 12 的 Cargo 领域模型, 他们发现两个领域模型具有一致的结构, 包括相同的领域对象和领域对象间关系. 这说明了程序设计和初始领域模型的设计呈现一致性. 在围绕初始领域模型和 PIM/CIM 模型进行场景走查时, 开发人员口头表达领域模型在程序设计中的具体表现形式, 包括每个领域对象在程序设计中的行为, 从而确认该领域模型的设计. 比如, 开发人员发现, 将 Cargo 的航线 Itinerary 和航段 Leg 显式地定义出来并建模为聚合, 而不是将 Cargo 运输过程的每段航程都以数据形式存储于数据库表, 能够避免领域逻辑泄露到基础设施层(这里是数据库), 同时还能解耦 Cargo 和具体航段 Leg.

图 14 和图 12 的 Cargo 领域模型具有完全一致的结构, 说明了在本次针对 Cargo 领域模型的迭代中, 开发人员严格地遵循了初始领域模型的设计进行代码实现. 然而, 实际软件开发中转换得到的 PIM 模型和初始领域模型可能并不一致, 这本质上是由代码实现和初始领域模型不一致造成的. 比如, Voyage 领域模型的代码到模型转换结果(如图 15 所示)就与图 12 的 Voyage 模型不一致. 通过可视化地对比转换得到的 PIM 模型和初始领域模型(如图 12 和图 15 所示), 而不是代码实现和初始领域模型, 领域专家和开发人员很快就发现了两种领域模型的分歧点为 CarrierMovement. 当开发人员和领域专家围绕这些领域模型讨论该分歧、进行知识消化时, 开发人员解释其这样进行程序设计的原因: 在实现 Voyage 模型时, 他们发现将 CarrierMovement 实现为实体以及其聚合的根, 能够更方便地跟踪领域中发生的运输事件, 同时便于 Cargo 领域模型对其访问.

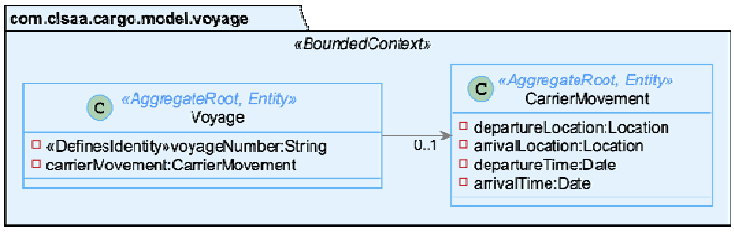


图 15 由 Voyage 领域模型的代码实现转换得到的 PIM 模型

以上 Cargo 和 Voyage 领域模型的知识消化过程说明了 C2MD 代码到模型转换带来的两个好处: 其一, 开发人员和领域专家能够可视化地对比转换得到的 PIM/CIM 模型和初始领域模型(如图 12 和图 14 所示), 而不是代码实现和初始领域模型, 从而更快地发现程序设计与领域建模的分歧, 以避免由于开发人员对初始领域模型的理解偏差而导致的实现偏移设计; 其二, 领域专家和开发人员还可以就可视化的模型展开知识消化, 确认领域模型在程序设计中的适用性(如图 12 和图 14 所示)或者讨论并消化程序设计与领域模型的分歧(如图 12 和图 15 所示). 这能够避免领域专家直接面对代码细节, 减小了知识消化的认知复杂度, 同时促进了程序设计到领域建模的反馈. 利用 C2MD 迭代地创建领域模型、实现领域模型代码、将代码转换为模型以及知识消化, 领域专家和开发人员能够不断地精炼他们对于领域业务规则的认识, 从而提炼更深层次知识的领域模型. 基于以上分析, 不难得出结论, C2MD 能够为领域建模实践提供支持.

7 总结及下一步工作

领域驱动设计围绕领域模型进行软件设计和开发, 从而应对软件开发的领域复杂性问题. 然而, 领域建模本身是应用领域驱动设计时面临的一项重要挑战. 在领域建模过程中, 领域模型和程序设计仅存在松散的逻辑联系, 这带来了程序设计偏离领域模型和知识消化过程缓慢两个问题. 本文基于模型驱动的逆向工程提出了一种面向领域驱动设计的代码到模型转换方法 C2MD, 并实现了自动化工具原型, 从而为领域建模过程提供支持. 为此, 本文首先提出了一种面向领域建模过程的元模型 DMSL, 包括该元模型在代码和模型状态下的不同表现形式. 相比于现有研究, 本文提出的元模型具有更完整的领域建模术语. 案例研究显示, C2MD 工具原型支持开发人员实时生成程序设计对应的领域模型, 一方面能够帮助对比程序设计和领域模型, 从而减小实现与设计的偏移; 另一方面能够促进程序设计到领域建模的反馈, 减少此过程对代码实现细节的依赖.

总的来说, 本文对领域建模问题的贡献如下: (1) 基于领域特定语言方法定义了一种面向领域建模过程的元模型 DMSL, 包括抽象层面的概念元模型和 4 种面向具体编码及建模技术的元模型; (2) 基于 DMSL 和模型驱动的逆向工程方法, 提出了一种面向领域驱动设计的代码到模型转换方法 C2MD; (3) 为了提高代码到模型转换的效率, 本文实现了 C2MD 方法的工具原型以实现自动化的代码到模型转换; (4) 将 C2MD 应用于真实软件项目的领域建模过程, 一定程度上验证了 C2MD 方法在真实软件开发中的适用性.

目前, 该方法仍存在待改进之处及后续工作如下.

1) DMSL 元模型所约束的构造块间关系还不够完整, 这是因为不同开发团队具有不同的领域建模实践, 而不同的领域建模实践方式带来了不同的构造块间关系约束. 后续工作将通过访谈领域建模的最佳实践以确定完整的构造块间关系, 同时通过支持开发人员自定义构造块间关系来提升工具原型的可扩展性.
2) C2MD 方法将 OOPL 代码转换为 UML 类图和包图, 仅考虑了领域对象间关系等结构领域模型内容, 而未考虑领域对象的动态交互行为. 后续工作将支持代码到行为领域模型的转换, 比如 UML 对象交互图, 以应对这一不足.
3) 本文的案例研究仍是实验室环境下进行的, 仅包含 3 个研究人员. 尽管以讨论形式展开的分析能够一定程度上缓解单个研究人员的主观性差异所带来的影响, 但实际软件开发团队可能具有更复杂

的交互行为, C2MD 方法能够多大程度上支持实际软件开发的领域建模活动仍未可知. 后续工作将应用 C2MD 于实际软件开发团队的领域建模以进一步验证其有效性.

References:

[1] Evans E. Domain-driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2004.

[2] Vernon V. Implementing Domain-driven Design. Addison-Wesley, 2013.

[3] Newman S. Building Microservices: Designing Fine-grained Systems. O'Reilly Media, 2015.

[4] Hippchen B, Giessler P, Steinegger R, Schneider M, Abeck S. Designing microservice-based applications by using a domain-driven design approach. Int'l Journal on Advances in Software, 2017, 10(3-4): 432−445.

[5] Raibulet C, Fontana FA, Zanoni M. Model-driven reverse engineering approaches: A systematic literature review. IEEE Access, 2017, 5: 14516−14542.

[6] Le DM, Dang DH, Nguyen VH. Generative software module development for domain-driven design with annotation-based domain specific language. Information Software Technology, 2020, 120: Article No.106239.

[7] Soares SA, Cortés MI. Elihu: A project to model-driven development with naked objects and domain-driven design. In: Proc. of the 20th Int'l Conf. on Enterprise Information Systems (ICEIS 2018). 2018. 272−279.

[8] Schneider M, Hippchen B, Giessler P, Irrgang C, Abeck S. Microservice development based on tool-supported domain modeling. In: Proc. of the 5th Int'l Conf. on Advances and Trends in Software Engineering. 2019.

[9] Paniza J. Learn Openxava by Example. 2011. https://dl.acm.org/doi/book/10.5555/2095958

[10] Haywood D. Apache Isis-developing domain-driven Java apps. Methods & Tools: Practical Knowledge Source for Software Development Professionals, 2013, 21(2): 40−59.

[11] Van Der Straeten R, Mens T, Van Baelen S. Challenges in model-driven software engineering. In: Proc. of the Int'l Conf. on Model Driven Engineering Languages and Systems, Vol.5421. 2008. 35−47. [doi: 10.1007/978-3-642-01648-6_4]

[12] Mantz F, Taentzer G, Lamo Y, Wolter U. Co-evolving meta-models and their instance models: A formal approach based on graph transformation. Science of Computer Programming, 2015, 104: 2−43.

[13] Robles Luna E, *et al*. Challenges for the adoption of model-driven web engineering approaches in industry. In: Proc. of the 13th Int'l Conf. on Web Information Systems and Technologies. 2017. 415−421. [doi: 10.5220/0006382704150421]

[14] Paige RF, Kolovos DS, Polack FAC. A tutorial on metamodelling for grammar researchers. Science of Computer Programming, 2014, 96: 396−416.

[15] Bruneliere H, Cabot J, Dupé G, Madiot F. MoDisco: A model driven reverse engineering framework. Information and Software Technology, 2014, 56(8): 1012−1032. [doi: 10.1016/j.infsof.2014.04.007i]

[16] Ramón ÓS, Cuadrado JS, Molina JG. Model-driven reverse engineering of legacy graphical user interfaces. In: Proc. of the IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2010). 2010. 147−150. [doi: 10.1145/1858996.1859023]

[17] Nirumand A, Zamani B, Ladani B. VAnDroid: A framework for vulnerability analysis of Android applications using a model-driven reverse engineering technique. Software: Practice and Experience, 2019, 49(1): 70−99. [doi: 10.1002/spe.2643]

[18] Diepenbrock A, Rademacher F, Sachweh S. An ontology-based approach for domain-driven design of microservice architectures. In: Proc. of the INFORMATIK 2017. 2017. 1777−1791.

[19] Rademacher F, Sachweh S, Zündorf A. Towards a UML profile for domain-driven design of microservice architectures. In: Proc. of the Int'l Conf. on Software Engineering and Formal Methods. LNCS 10729, 2017. 230−245. [doi: 10.1007/978-3-319-74781-1_17]

[20] Wizenty PN, Rademacher F, Sorgalla J, Sachweh S. Design and implementation of a remote care application based on microservice architecture. In: Proc. of the Federation of Int'l Conf. on Software Technologies: Applications and Foundations. LNCS 11176. 2018. 549−557. [doi: 10.1007/978-3-030-04771-9_41]

[21] Rademacher F, Sorgalla J, Sachweh S. Challenges of domain-driven microservice design: A model-driven perspective. IEEE Software, 2018, 35(3): 36−43.

[22] Kazman R, Woods SG, Carrière SJ. Requirements for integrating software architecture and reengineering models: CORUM II. In: Proc. of the 5th Working Conf. on Reverse Engineering. 1998. 154−163.

[23] Pérez-Castillo R, De Guzmán IGR, Piattini M. Business process archeology using MARBLE. Information and Software Technology, 2011, 53(10): 1023–1044. [doi: 10.1016/j.infsof.2011.05.006]

[24] Stefik M, Daniel GB. Object-oriented programming: themes and variations. AI Magazine, 1985, 6(4): 40–62.

[25] OMG O. OMG unified modeling language (OMG UML). Version 2.4.1, 2011. https://www.omg.org/spec/UML/2.5.1/About-UML/

[26] Ráth I, Ökrös A, Varró D. Synchronization of abstract and concrete syntax in domain-specific modeling languages: By mapping models and live transformations. Software & Systems Modeling, 2010, 9(4): 453–471. [doi: 10.1007/s10270-009-0122-7]

[27] Selic B. A systematic approach to domain-specific language design using UML. In: Proc. of the Object and Component-oriented Real-time Distributed Computing (ISORC 2007). 2007. 2–9. [doi: 10.1109/ISORC.2007.10]

[28] Evans E. Domain-driven Design Reference: Definitions and Pattern Summaries. Dog Ear Publishing, 2014.

[29] Cazzola W, Vacchi E. @Java: Bringing a richer annotation model to Java. Computer Languages, Systems & Structures, 2014, 40(1): 2–18.

[30] Gosling J, *et al*. The Java® language specification. 2020. https://docs.oracle.com/javase/specs/jls/se15/html/index.html

[31] Hejlsberg A, Torgersen M, Wiltamuth S, Golde P. The C# programming language. 2008. https://dl.acm.org/doi/10.5555/1502323

[32] Lai Y, Shi ZZ. An efficient data mining framework on Hadoop using Java persistence API. In: Proc. of the 10th IEEE Int'l Conf. on Computer and Information Technology. 2010. 203–209.

[33] OMG O. XML Metadata interchange (XMI). Version 2.5.1, 2015. https://www.omg.org/spec/XMI/2.5.1/About-XMI/

[34] Yin RK. The case study as a serious research strategy. Knowledge, 1981, 3(1): 97–114. [doi: 10.1177/107554708100300106]

[35] Yin RK, Wrote; Zhou HT, Shi SJ, Trans. Case Study Research: Design and Methods. 5th ed., Chongqing: Chongqing University Press, 2017 (in Chinese).

[36] Zhong CX, Li BB, Zhang H, Zhang C. Evaluating granularity of microservices-oriented system based on bounded context. Ruan Jian Xue Bao/Journal of Software, 2019, 30(10): 3227–3241 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5797.htm [doi: 10.13328/j.cnki.jos.005797]

附中文参考文献:

[35] Yin RK, 著; 周海涛, 史少杰, 译. 案例研究: 设计与方法. 第 5 版, 重庆: 重庆大学出版社, 2017.

[36] 钟陈星, 李杉杉, 张贺, 章程. 限界上下文视角下的微服务粒度评估. 软件学报, 2019, 30(10): 3227–3241. http://www.jos.org.cn/1000-9825/5797.htm [doi: 10.13328/j.cnki.jos.005797]

钟陈星(1995－), 女, 博士生, 主要研究领域为微服务, 领域驱动设计.

任贵杰(1996－), 男, 工程师, 主要研究领域为微服务, 领域驱动设计.

李文君(1995－), 男, 硕士, 主要研究领域为领域驱动设计.

荣国平(1977－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为软件过程实证软件工程.