关注点分离在计算思维和软件工程中的方法论意义

何明昕
(暨南大学计算机科学系 广州 510632)

摘 要 关注点分离可追溯到柏拉图对整体与部分关系的思考。作为基本的系统化计算思维原则,关注点分离体现在问题求解、算法设计、软件设计、软件架构描述、软件开发过程等诸多方面。简要归纳了软件和计算的本质特点;重点分析关注点分离作为重要的方法论原则在软件工程中的主要作用和体现形式;介绍了近期有关关注点高级分离的研究,包括关注点的多维分离和面向方面软件开发;最后阐述了关注点分离原则与具体问题具体分析策略相结合的实践意义。

关键词 关注点分离,计算思维,软件工程,方法论
中图法分类号 TP301 **文献标识码** A

Separation-of-concerns as a Methodological Principle in Computational Thinking and Software Engineering

HE Ming-xin
(Department of Computer Science, Jinan University, Guangzhou 510632, China)

Abstract The idea of separation-of-concerns (SoC) can be traced back to Plato's thinking on the relationship between the universe and its elements. As a fundamental systematic principle in computational thinking, it is incarnated in many aspects such as problem solving, algorithm design, software design, software architectures documenting and development processes. The essential characteristics of software and computing were summarized. The significance and embodiments of SoC as a methodological principle in software engineering were analyzed. The recent researches on advanced SoC, including multidimensional SoC and aspect-oriented software development (AOSD) were introduced. The pragmatic significance of combining separation-of-concerns principle with concrete-problem-with-concrete-analysis strategy were stressed to solve a complex problem.

Keywords Separation of concerns (SoC), Computational thinking, Software engineering, Methodology

1 引言

关注点分离(Separation of Concerns, SoC)是日常生活和生产中广泛使用的解决复杂问题的一种系统思维方法。大体思路是,先将复杂问题做合理的分解,再分别仔细研究问题的不同侧面(关注点),最后综合各方面的结果,合成整体的解决方案。

在概念上分割整体以使实体个体化的观点可以追溯到柏拉图[1]。柏拉图把探究自然比作在关节处切割自然,窍门在于要找到关节,不要像生疏的屠夫那样把关节切得粉碎。庄子在《庖丁解牛》寓言中也阐释了类似的真知灼见。

作为最重要的计算思维[2]原则之一,关注点分离是计算科学和软件工程在长期实践中确立的一项方法论原则[3]。此原则在业界更多的时候以"分而治之"(Divide-and-Conquer)的面目出现,即将整体看成为部分的组合体并对各部分分别加以处理[4]。模块化(Modularity)是其中最有代表性的具体设计原则之一[2,5,6]。

关注点分离原则不仅体现在问题求解、算法设计、软件设计等设计方法中,同时也体现在软件开发过程、软件项目管理以及软件开发方法学等诸多方面[5]。在某种意义上,正是对软件开发不同关注点的分离视角和关注重点的差别,导致了软件开发技术和开发方法的演变和发展[7]。

因此,从方法论角度,研究关注点分离在计算科学和软件工程中的作用,对相关教学和实践都有重要的现实意义。

本文简要归纳了软件和计算的本质特性;重点讨论了关注点分离作为方法论原则在软件工程中的主要作用和体现形式;最后介绍了近期关于关注点高级分离的研究,包括多维关注点分离[8]和面向方面软件开发(AOSD, Aspect-Oriented Software Development)[9]。这些创新的思想涉及专门编程技术和整体开发方法,被认为是继结构化开发范型(Paradigm)及面向对象开发范型之后最重要的新的软件开发范型。

2 软件和计算的本质特性

软件(系统)作为软件工程的产品,和其他工程活动的产品一样,是一种人工制品(artifact, artificial)[10]。作为人工系统的软件,有它特定的结构(Structures),能表现特定的内部

到稿日期:2008-10-30 本文得到广东省自然科学基金(编号 5006061)资助。
何明昕(1963-),男,副教授,博士研究生,主要从事软件工程、并行分布式网络计算、电子商务方面的研究,E-mail:mx. he@yeah. net。本文为作者在2008年全国"计算思维与计算机导论"专题学术研讨会上所作的大会报告。

及外部行为(Behaviors),实现特定的功能(Functions),以满足用户的特定需求,同时具备必要的质量特性。

从技术应用的角度,可以将软件分为信息系统、实时系统、分布式系统和嵌入式系统[5]。从设计方法论的角度,可以将软件分为变换式系统(Transformational System)和反应式软件(Reactive System)[11]。变换式系统是无状态的符号处理机器,将输入数据集合变换成输出数据集合;而反应式系统针对不同的外部事件做出相应的响应,通常具有交互性、实时性和分布计算等特性。

软件系统具有其他工程制品没有的独特性质,主要体现在4个方面:

1) 软件是柔性的(malleable)。可以很容易地改变软件(代码或配置)自身,而要获得满足需求的软件的设计过程却异常复杂[5]。

2) 软件作为离散的数字产品,被 Dijkstra 称为激进的新生事物[4],与物质世界的连续产品具有本质的区别,一个比特(数位)之差就可能导致异想不到的甚至是灾难性的后果。

3) 软件是智力密集性产物,是分析、设计和实现活动紧密结合的创造物;而在其他工程领域,设计与制造活动通常界限分明,通常制造/施工环节占据了最终产品的大部分成本。在软件分析(问题描述)和设计(解决方案、实现)之间存在不可避免的交织[12]。任一层次的描述总是另一较高层次的实现;任一层次的实现总是另一较低层次的描述;对人来说最低层次的程序实现,对计算机而言便是最高层次的描述[13]。

4) 软件产品具有一系列质量属性,包括互相关联的外部属性及内部属性,如正确性、可靠性、稳健性、性能、可验证性、可维护性(可修复性、可演化性)、可用性、可移植性、可理解性、互操作性等等。由于软件产品本身的不可见性,对软件开发还提出了生产率、时间性、过程可见性等质量要求[4]。与其他工程领域比较,这些诸多的软件质量属性大都具有特殊性质,更进一步增加了软件工程的复杂性。

马希文在文献[14]译序中阐明了计算机能解问题的3个必要前提:1) 问题必须能够形式化,即必须以严格的手段建立问题的模型;2) 问题必须是可计算的,即找到求解问题的算法并能用程序实现;3) 问题必须有适当的复杂度,即程序可在有限计算空间和时间内得出结果。这3个条件揭示了计算(computing)的本质特性,给出了软件系统可能的边界,也有助于人们对软件开发活动本质的认识。

软件工程面对的关键问题是有效处理复杂性。一类是问题本身的计算复杂性,这在科学方法上是无法超越的,只能通过启发式方法寻求可接受的近似解。另一类复杂性来自对业务的理解、软件开发方法和技术、项目组织、人员等综合因素。

Ghezzi 在文献[5]中总结了在软件开发和管理过程中具有普遍性的7项原则:1) 严格与形式化(Rigor and Formality);2) 关注点分离(Separation of Concerns);3) 模块化(Modularity);4) 抽象(Abstraction);5) 预期变化(Anticipation of Change);6) 泛化(Generality);7) 递增性(Incrementality)。

这些原则通过具体方法和技术体现在软件开发实践中。特定的方法和技术的组合构成软件开发方法论,最终在支持这些方法和技术的软件开发工具/环境中得到具体应用。

3 软件工程中的关注点分离原则

软件工程的主要目标是改善软件质量,降低软件产品的成本,便于维护和演化。业界不断追寻开发技术和方法,以降低软件复杂度,提高可理解性和重用性,同时促进演化。控制复杂性和提高可理解性需要分解机制将软件分割为有意义可管理的片段。也需要组合机制将这些片段组合成有用的整体。

在软件产品开发过程中,涉及众多具体决策,从软件产品的不同特性、软件开发过程和组织方式、不同涉众的利益关注点,到技术、工具、架构、设计方案等多方面的决策问题。这些不同的问题侧面既有各自独立的需求和约束,可以分别从其关注点的角度予以考虑;同时,不同的侧面可能互相交织,甚至互相冲突,必须从更高的角度以统一的观点来综合考虑。

通过将问题领域和实现领域的关注点分离,以及各类软件开发技术和技能的分离,实现了软件开发工作责任的分离和人员的分工,为大规模软件开发提供了合作基础。典型的是,Microsoft 对每一个软件产品采用产品团队、开发团队和测试团队的分工模式。

通过对软件开发时间关注点的分离,发展了不同的软件开发过程,从早期的瀑布模型、经原型法及螺旋模型,到以迭代递增为主的统一过程及各类敏捷过程。

通过系统组件的分离,使得软件系统可以处理更大更复杂的现实问题。

通过对软件功能特性与软件质量特性的分离与综合,为实现软件系统多方面的质量要求提供了可能。

以下讨论关注点分离的一般模式,并从软件设计方法和软件架构视图两个角度探讨关注点分离原则的具体运用。

3.1 关注点分离的一般模式

对一个典型的复杂问题 P,通过关注点分离解决问题的基本思路可一般地描述为以下3个步骤[13]:

1) 先将待解问题 P 分解为不同的关注点 $P_1, P_2, \cdots, P_n$,即:$P \rightarrow D(P_1, P_2, \cdots, P_n)$,$D$ 表示问题分解策略,即关注点分离方法;

2) 对 $P_1, P_2, \cdots, P_n$ 分别考虑,求得各自的解 $S_1, S_2, \cdots, S_n$;

3) 通过对 $S_1, S_2, \cdots, S_n$ 的合成,求得问题 P 的解 S,即:$C(S_1, S_2, \cdots, S_n) \rightarrow S$,$C$ 表示解的合成方法。

解的合成方法与关注点分离策略密切相关,对具体问题做具体的分析,充分掌握问题相关的具体知识,把握住其关键特征,在关节处实施分割,往往事半功倍。

对复杂问题可能需要多层次多维度的分解。自顶向下逐步分解求精的软件设计方法以及螺旋递增的软件开发过程都是例证。

3.2 分离视角与软件设计方法的演变

在软件设计方法和软件开发工具的演变发展过程中,关注点的分离视角和关注重点扮演了十分重要的角色。体现分而治之思路的模块化设计是最重要的设计原则。

Parnas 于1972年在文献[3]中提出模块划分的基本原则,即通过信息隐藏降低模块之间的耦合,从而降低系统的复杂性。"高内聚、低耦合、信息隐藏"已成为软件设计的十字真经。Wirth 提出著名的公式:Algorithms+Data Structures=

Programs[15]，强调了算法和数据结构在软件程序中的重要性。Jackson 提出通过自顶向下的功能分解设计程序的结构化方法[16]。Warnier 等提出以数据结构的分解为主逻辑地推导程序结构的方法[17]。Dijkstra 和 Gries 等发展了强调程序的正确性证明以形式化推导为主的程序设计方法学[18]，并成为静室软件工程中的“需缺陷程序设计”的基础[19]。

以功能和过程为主要分离关注点的结构化(过程式)程序设计方法对以变换式系统为主的应用十分有效，其方法和思路也是其他程序设计方法的基础。

Mayer 在文献[20]中总结了模块化的 5 个标准和 5 项实现原则。5 个标准是可分解性、组合性、可理解性、连续性和保护性；保证模块满足上述标准的 5 项原则是语言模块单元、很少的接口、小接口、显式的接口和信息隐藏，其中关于接口的 3 项原则可以归纳为简单接口原则。

经历了抽象数据类型(ADT，Abstract Data Type)以及人工智能中的框架(Frame)知识表达之后，逐渐演化出以职责(Responsibilities)和契约(Contracts)为主要分离视角的面向对象的设计风格。对象(类)集成了数据和操作，但功能和过程分散在多个对象的协作之中。

面向对象的方法虽然受到软件厂商和产业界的追捧，但真正掌握面向对象方法、设计出高质量(能够正确有效地运行、代码便于修改扩展、便于与人交流)的软件系统，仍是一项具挑战性的工作。

Horstmann 在文献[21]中归纳了判断类接口质量的 5C 准则：Cohesion (内聚性) 、Completeness (完整性) 、Convenience (便利性) 、Clarity (清晰性) 和 Consistency(一致性)。

各种基于面向对象技术的组件化开发方法也是模块化设计的另一具体表现形式。

考虑到反应式系统是当前应用软件的主要类型，Wieringa 在文献[11]中提出了结合结构化方法、面向对象方法以及信息系统分析与设计思路，提出从功能、实体(数据)、行为、通信 4 个角度把握软件的主要关注点，综合分析和设计软件系统的系统方法。

Jacobson 在文献[9]中提出了以用例为基础的面向方面的软件分析与设计方法，将面向数据和职责的对象(类)与面向用例(功能)的方面(Aspect)分别建模，有效地分离了不同的关注点，同时又提供了简洁优雅的组合机制。本文第 4 节专门介绍面向方面的软件开发新方法。

3.3 软件架构视图分离原则

软件架构在现代软件开发中起到关键性作用。本节分析软件架构视图的分离原则。

软件架构作为近十几年业界研究和应用的热点之一，其定义还有待统一。Perry 等在文献[22]给出的定义与我国学者徐家福的定义比较接近：软件架构 = { 结构元(Elements)，结构型(Forms)，结构理(Rationale) }，即软件架构是由若干具特定结构形按结构理组合起来的结构元的集合。结构元包括处理元、数据元和连接元。

软件架构视图(Architectural Views)是从某一个特定的关注点对软件架构的刻画和描述，以满足不同涉众或不同目的的需要。

Kruchten 于 1995 年在文献[23]中提出了 4+1 的架构视图模型：1) 逻辑视图(Logical View)描述系统的行为需求；2) 过程视图(Process View)强调并发性、分布性、系统完整性、容错等；3) 开发视图(Development View)着重软件模块在软件开发环境中的组织结构；4) 物理视图(Physical View)考虑系统方面的需求，包括系统可用性、稳定性、性能、可扩展性等。作为“+1”的用例(场景)视图则是展示 4 个视图联合工作的综合描述。

Soni 等同时在文献[24]中提出了概念视图(Conceptual View)、模块互联视图(Module Interconnection View)、运行视图(Execution View)和代码视图(Code View)的架构视图。这些视图被称为西门子 4 维视图模型(Siemens Four View Model)。

Clements 在文献[25]中将软件架构视图分为 3 类：1) 模块(Module)类：系统主要的实现单元及关系；2) 构件与联接器(Component-and-Connector)类：记录系统运作的单元及关系；3) 配置(Allocation)类：记录软件及其开发和运行环境的关系。

SOA 作为一种企业综合集成的软件架构蓝图和 IT 战略实施方法，近年得到国际 IT 巨头的重视和业界普遍的关注，被认为是未来企业 IT 建设的重要基础设施[26]。其基本思路就是将业务关注点和技术关注点分离，通过基本服务层、中介代理层、流程管理层和企业综合服务层的逻辑分层以及分步骤的建设实施，实现企业内部各系统各部门以及企业与业务伙伴之间的系统整合，达到更广泛更高效的信息共享和业务合作。

4 关注点的多维分离与面向方面的软件开发新方法

所有现代软件形式体系都通过特定的分解和组合机制，支持某种程度的关注点分离。但通常只提供小规模的受限的分解和组合机制，即只支持一个主要的分离维度。结构化方法以功能分解为主要维度，面向对象方法以实体及责任分解为主要维度。这些单一的分解维度限制了软件工程目标的实现。

将关注点沿单一的主要维度分解具有实在的价值，但常常不够充分。属于其他关注点维度的单元最终会分散在多个模块中，而且彼此混杂在一起。

Tarr 等提出了通过支持多维度分解重叠关注点的并发操作，提出了一种新的软件制品的模型，通过多维分解和组合来克服“主要分解的强制性”[8]。该模型允许并发、多维分解和组合，对现有形式体系进行补充，为开发者提供了额外的模块化的灵活性。

通过引入超切片，封装除主导维度外的其他维度上的关注点。超切片内部的模块包含且只包含所有那些处理给定关注点的单元或与之相关的单元。超切片可以相互覆盖，一个特定的单元可能以不同的形式出现在不同的超切片中，从而实现根据关注点的多维度进行并发分解的支持。

面向方面的编程(AOP，Aspect-oriented Programming)研究及实践[7,8]部分实现了关注点多维分解的模型。早期 AOP 研究及应用大都局限在日志、基于角色的安全、事务划分、持久性、延迟装载、异步调用、同步、缓存等非业务功能类的技术服务的实现。

Jacobson 在文献[9]中提出的基于用例的面向方面软件

开发方法(AOSD, Aspect-Oriented Software Development),利用传统的面向对象技术建立领域模型及对象模型,实现为类(class)模块;通过用例(use case)捕捉业务功能需求,实现为方面(aspect)模块。每一个横切(crosscutting)方面可以跨越多个类,但其组织及变更又限制在单独的方面模块内。这就有效地将结构化方法的主要关注维度(功能需求)和面向对象方法的主要关注维度(对象实体)各自分离,成为各自独立的模块,并通过编织(weave)机制使模块合成为一个整体;在实现复杂功能的同时,使软件系统模块结构更清晰,并具有方便的可扩展性,为递增式演化开发提供了一种切实可行的程序设计方法论。

这些创新思想涉及专门编程技术和整体开发方法,被认为是继结构化开发范型和面向对象的开发范型之后,最重要的软件开发范型。AspectJ, JMangler, Spring, Tapestry, JBoss等工具和框架为Java应用提供了面向方面的实现支持。其他语言支持AOP特性的工具也在研究发展中。可以预见,随着对关注点分离在软件工程中的方法论意义的更深入更全面的认识,面向方面的开发方法将在软件开发实践中得到更广泛的应用。

架构作为近十几年业界研究和应用的热点之一,其定义还有待统一。Perry等在文献[21]给出的定义与我国学者徐家福的定义比较接近:软件架构={结构元(Elements),结构型(Forms),结构理(Rationale)},即软件架构是由若干具特定结构形按结构理组合起来的结构元的集合。结构元包括处理元、数据元和连接元。

结束语 关注点分离作为一种普适的处理复杂问题的系统思维方法和原则,在计算思维和软件工程中有着重要的方法论意义。关注点分离原则给出了处理复杂性的基本思路,在软件开发实践中呈现为具体设计原则和设计方法。本文对软件和计算本质的归纳,对关注点分离原则在软件工程中的具体体现的分析,只是冰山一角。

在实践中,关注点分离原则必须和具体问题具体分析策略相结合,以期获得恰如其分的分离视角以及简明优雅的合成策略。正如书法家刘炳森所说,世间的大学问全在于分寸和火候的把握。对应用领域的问题,须通过深入细致的调查研究,把握问题精微细致的"关节"。在"关节"处解"牛",才能体现融科学与艺术为一体的工程之精妙。

参 考 文 献

[1] 欧阳莹之. 复杂系统理论基础[M]. 田国宝,等译. 上海:上海科技教育出版社,2002:90-91

[2] Wang J M. Computational Thinking. Communications of ACM, 2007,49(3):33-35

[3] Parnas D L. On the criteria to be used in decomposing systems into modules. Comm. ACM,1972,15(12):1053-1058

[4] Dijkstra E W. On the cruelty of really teaching computing science. Comm. ACM,1989,32(12):1398-1414

[5] Ghezzi C, Jazayeri M, Mandrioli D. Fundamentals of Software Engineering (2nd edition). Pearson Education,2003

[6] Baldwin C Y,Clark C Y. Design Rules (vol. 1): The Power of Modularity. Cambridge,Mass: MIT Press,1999

[7] Stanley JR S M, Rouvellou I. 面向方面软件开发的关注点建模// R. E. Filman,et al. ,eds. 莫倩,等译. 面向方面的软件开发[M]. 北京:机械工业出版社,2006:299-315

[8] Tarr P,Ossher H,Sutton JR S M,et al. N度分离:关注点的多维分离// Filman R E,et al. ,eds. 莫倩,等译. 面向方面的软件开发[M]. 北京:机械工业出版社, 2006:21-36

[9] Jacobson I, Ng P W. Aspect-Oriented Software Development with Use Cases. Pearson Education,2005

[10] Simon H A. The Science of the Artificial. Cambridge. Mass: MIT Press,1969

[11] Wieringa R J. Design Methods for Reactive Systems: Yourdon, Statemate, and the UML. San Francisco: Morgan Kaufmann Publishers,2003

[12] Swatout W,Balzer R. On the inevitable intertwining of specification and implementation. CACM,1982,25(7):438-440

[13] 何明昕. 面向问题的系统化程序设计方法及其描述工具[J]. 计算机科学,1995,22(5):54-57

[14] Dreyfus H L. 计算机不能做什么—人工智能的极限[M]. 宁春岩,译. 北京:三联书店,1986

[15] Wirth N. Algorithms + Data Structures = Programs. Prentice Hall,1976

[16] Jackson M A. Principles of Program Design. Academic Press, 1975

[17] Warnier J D. Logical Construction of Programs. NY: Von Nostrand Reinhold,1976

[18] Gries D. The Science of Programming. Springer-Verlag,1981

[19] Stavely A M. 零缺陷程序设计[M]. 夏昕,王尧,译. 北京:机械工业出版社,2003

[20] Mayer B. Objected-oriented Software Construction, Englewood Cliffs NJ: Prentice Hall,1988

[21] Hornstmann C. 面向对象的设计与模式[M]. 张琛恩,译. 北京:电子工业出版社,2004:107-111

[22] Perry D E, Wolf A L. Foundations for the study of software architecture. Software Engineering Notes,1992,17(2):40-52

[23] Keuchten P. The 4+1 view model of architecture. IEEE Software,1995,12(6):42-50

[24] Soni D, Nord R L, Hofmeister C. Software Architecture in industry applications// Proc. of the 17th Inter. Conf. on SE. 1995: 196-207

[25] Clements P, Bachmann F, Bass L, et al. Documenting Software Architectures: Views and Beyond. Boston. MA: Pearson Education,2003

[26] Krafzig D, Banke K, Slama D. Enterprise SOA: Service-Oriented Architecture Best Practice. Pearson Education,2005

(上接第37页)

[22] Kowaliw T, et al. Using Competitive Co - evolution to Evolve Better Pattern Recognisers [J]. International Journal of Computational Intelligence and Applications,2005,5(3):305-320

[23] Angeline P J, Pollack J B. Competitive environments evolve better solutions for complex tasks [A]// Proceedings of the Fifth international Conference on Genetic Algorithms [C]. 1993:264-270

[24] Hugues J, Jordan B P. Coevolutionary Learning : a case study [A] // The Fifteenth International Conference on Machine Learning [C]. 1998:251-259

[25] Claverie J M, et al. Robust nonlinear control design using competitive coevolution// Proceedings of the 2000 Congress on Evolutionary Computation. 2000(1):403-409

[26] Floreano D, Nolfi S. Adaptive Behavior in Competing Co-evolving Species [C] // Fourth European Conference on Artificial Life. 1997:378-387