NATO 软件复用标准导论

An Introduction to NATO Software Reuse Standards

郭立峰 郭 耀 常继传
(北京大学计算机科学技术系 北京 100871)

Abstract Practice has demonstrated that software reuse can improve the productivity and quality effectively, which consists of two life cycles, the development of reusable software component and development with reusable software component. During both life cycles, the adoption of a proper standard to identify and develop reusable component will advance the practice of software reuse greatly. Therefor, NATO established a whole series of guidelines for software reuse to help the project management department of NATO host nations and contractors reuse software effectively. The standards consists of Standard for the Development of Reusable Software Components, Standard for Management of a Reusable Software Component Library and Standard for Software Reuse Procedures, which respectively provide guidance and help for software reuse in each phase of software life cycle.

Keywords NATO, Software reuse, Development of reusable component, Management of component library, Reuse procedure

软件复用包括开发可复用软件构件和基于可复用构件的软件开发两个生存周期。在这两个生存周期中,采用一个适当的标准以识别和开发可复用软件将大大促进软件复用的实践。为此,NATO(北大西洋公约组织)制定了一整套软件复用的指导性标准,以帮助 NATO 及其参与国和承包商的项目管理部门进行有效的软件复用。这套标准包括《可复用软件构件开发指南》、《可复用软件构件库管理指南》和《软件复用过程指南》三个文档,分别从软件生存周期的各个阶段对软件复用进行指导,以便最大限度地减少复用代价和增加复用收益。

为了吸收和利用国外软件复用的最新研究成果,我们对国际上相关研究和实践工作进行了深入分析,以期能拓宽思路,取长补短,使我们的工作能和国际主流接轨。本文便是对 NATO 软件复用标准的总结和介绍,希望能为国内相关工作的进展和相关的软件从业人员起到一个借鉴和参考的作用。下面先介绍一些基本概念,然后详细介绍 NATO 的上述三个标准。

一、基本概念

本节介绍复用的基本术语和概念,并解释制定这套标准的实际目的,为读者理解软件复用的好处和挑战提供了一个参考性的框架。

1.1 有关的术语定义

标准中使用的关键术语简要定义如下:

复用(Reuse):在一个新的语境中(同一系统的其它地方或另一个系统中)使用已有的软件构件。

可复用性(Reusability):软件构件的可以被复用的程度或范围。遵循一个适当的设计和编码标准将增强构件的可复用性。

可复用软件构件(RSC):可以被复用的软件实体;它可以是设计、代码或软件开发过程的其它产品。RSCs 有时称为"软件资产"。

复用者(Reuser):复用 RSC 的个人或组织。

可移植性(Portability):原来在一台计算机和操作系统上开发的软件构件可以在另一台计算机和(或)操作系统上使用的程度。若构件可移植性好,则

它的复用潜力就大。

领域(Domain)：指一类相关的软件应用。领域有时可划分为"垂直的"和"水平的"。"垂直的"领域包括某个应用范围内的所有开发层次(如 MIS 领域)，而"水平的"领域则指一种特定类型的、不限于某类应用的软件过程(如对堆栈的操作)。一般来说，构件在同一个领域中复用的潜力较大。

领域分析(Domain analysis)：对一个选定的领域进行分析以标识出其中通用的结构和功能，目的是提高复用的潜力。

构件库(Library)：可复用软件构件的集合，包括向用户提供构件时所需的过程和功能。

检索系统(Retrieval system)：支持可复用软件构件分类和检索的自动化工具。

软件生存周期(Software life cycle)：软件系统在开发和配置时所经历的一系列阶段。尽管不同项目在具体阶段上会有差别，但通常都会包括下列阶段：需求分析、设计、编码、测试和维护。

1.2 复用的优越性

软件复用可以提高软件生产率并减少开发代价，还可以提高软件系统的质量。具体来说，可以归纳为下列五个方面：

1)**提高生产率**。这是软件复用最明显的好处，从而减少开发代价。生产率的提高不仅体现在代码开发阶段，在分析、设计及测试阶段同样可以利用复用来节省开销。用可复用的构件构造系统还可以提高系统的性能和可靠性，因为可复用构件经过了高度优化，并且在实践中经受过检验。

2)**减少维护代价**。这是软件复用另一个重要的优越性。由于使用经过检验的构件，减少了可能的错误，同时软件中需要维护的部分也减少了。例如，要对多个具有公共图形用户界面的系统进行维护时，对界面的修改只需要一次，而不是在每个系统中分别进行修改。

3)**提高互操作性**。软件复用一个更为专业化的好处在于提高了系统间的互操作性。通过使用接口的同一个实现，系统将更为有效地实现与其它系统之间的互操作。例如，若多个通讯系统都采用同一个软件包来实现 X.25 协议，那么它们之间的交互将更为方便。

4)**支持快速原型**。复用的另一个好处在于对快速原型的支持，即可以快速构造出系统可操作的模型，以获得用户对系统功能的反馈。利用可复用构件库可以快速有效地构造出应用程序的原型。

5)**减少培训开销**。复用的最后一个好处在于减少培训开销，即雇员在熟悉新任务时所需的非正式的开销。如同硬件工程师使用相同的集成电路块设计不同类型的系统，软件工程师也将使用一个可复用构件库，其中的构件都是他们所熟悉和精通的。

1.3 复用的维

软件复用有多个维，可从不同角度对复用进行划分。NATO 复用标准用以下方式对复用进行了分类：

组装式复用与生成式复用。也可以称为产品复用与过程复用。组装式方法利用库中的底层构件自底向上地开发系统，关键是要解决构件的分类和检索技术，以及开发出一个自动化系统以支持组装过程。生成式方法是特定于应用领域的，它采用标准的领域构架(architecture)模型(即类属的构架)和一致的构件接口，目的是根据一个适当的参数规约生成新的系统。(目前在商业软件中使用的第四代生成语言[4GLs]可以被认为是生成式复用的一个例子。)这种方法在成熟的领域中非常有效，但在开发初始模型时需要很大的努力。

小规模复用与大规模复用。复用的另一个维是可复用构件的规模。小规模复用(例如使用数学函数库)现在已得到了广泛的应用，但个别的复用节省的代价并不大，必须要普遍地复用才能获得收益。大规模复用针对的是整个子系统(例如飞机导航或消息处理子系统)，这时个别的复用即可获得很大的收益，因为复用了成千上万行代码。但是对一个特定的大构件来说，复用的机会是非常有限的。

原样复用与带修改的复用。构件既可以原样复用，也可能需要修改。一般来说，可复用构件应设计得比较有弹性，例如可以设计成参数化的构件，但为了满足复用者的需求，适当的修改也是必要的。易修改性(软件构件易于修改的能力)是可复用软件中尤为重要的性质。

通用性与性能。在构件的通用性与性能之间常常存在折衷。设计得通用和灵活的构件常常包括为了支持通用性而增加的额外开销。

1.4 实现复用的困难

软件复用需要改变传统的软件开发方法，也得转变一些传统观念。为了达到完全的收益，就得克服

下面一系列挑战：

识别复用机会。识别复用机会是一个主要的技术问题。软件工程师可能知道类似的软件已经编写过，但如何找到它是一个问题。复用库有助于解决这个问题。当找到一个构件时，可能很难判断它是否真的满足自己的需求，要进行修改也是很困难的。不少看上去可复用的软件实际上是不可复用的，因为它可能有不适当的接口、隐含的依赖、不可改变的功能限制等，或者仅仅是由于它难于理解而使得复用者宁愿重新开发。NATO 的软件复用标准有助于避免这些问题。

投资。制作可复用软件通常比开发一次性的系统需要更多的投资，这些投资包括使得软件更具有弹性、保证它的质量、提供所需的额外文档等。每个组织必须决定如何支持这项投资。

"无创造性"的顾虑("Not Invented Here")。开发者常常不愿复用别人的软件。软件工程师喜欢创造性的劳动，复用软件时就会感到创造性的消失。有效的管理、鼓励以及培训等措施有助于让工程师们将创造性的视角转向更大的"构造块"—可复用软件构件。

评价和度量。对软件开发行为进行评价和度量总是很困难的，怎样才算是一个"好"软件？评价的标准很多。已经有一些比较有效的管理性方法，但在复用环境中这些传统的度量方法还需要修改，在这方面几乎没有什么经验。

合同、法律和所有权问题。软件复用还受到大量关于合同、法律和所有权问题的影响。现在这种订合同的方式使得承包商不愿复用已有的软件或提供软件给别人复用，在对软件构件质量的责任和保证上也会引发法律纠纷，而且还应确定由谁来负责维护软件构件。

二、可复用构件开发标准

NATO 的可复用软件构件(以下简称构件)开发标准为创建具有最大复用潜力的软件提供指南。该标准面向 NATO 的项目管理者和项目承包商，针对软件生存周期中的需求分析、设计、详细设计与实现、质量保证与测试以及文档等方面分别论述，帮助用户组织一个可复用软件构件的开发过程。标准的大部分内容是独立于编程语言的，在实现时采用特定语言的编码标准作为该标准的补充。

分析、设计和测试阶段的成果本身都有被复用的潜力，应被当作构件，尤其在新系统中复用前期的软件工程成果往往带来对相应后期产品的复用。可复用构件应该以方便复用的方式来表示，易于识别，易于独立提取，与系统特定的和易变的成分隔离；在组织内部采用一致的机器可读的记号表示分析和设计构件，以便进行自动的信息提取和转换。从分析到设计到编码的转换应该遵循上一阶段的复用考虑，保持相邻阶段构件之间的良好映射和可跟踪性质。以下分几个方面介绍。

2.1 需求分析和领域分析

这一阶段为软件复用打下基础，此时对复用给予的关注将在相当大的程度上影响所开发软件的可复用性。主要应该在以下四个方面作出努力。

(1)**建立一个鼓励复用现有软件的机制** 需求规约必须认识到复用的必要性并鼓励软件复用。好的需求规约应该只规定所需的功能和性能指标，允许开发者决定操作上和实现上的细节。一个极端明细的规约是很难与任何现有软件相匹配的，不必要的系统需求将限制软件复用。将软件复用作为需求之一，在规约中规定所有必需的和期望的复用活动。

· 要逐条检查每一需求的必要性，确信其中不包括进一步的设计选择。

· 改变以往将需求规约在招标之前定死不变的做法，向潜在的承包商提供需求规约的草稿，让开发者参与进来，标识出可以修改并促进复用的地方，由客户和承包商共同完成项目的规约文档。

· 复用需求必须解释什么是复用和如何评估复用。为开发者指定一个复用目标，比如复用的代码、函数的数量等，在合同中为复用制定奖励措施以鼓励复用。

(2)**将可复用软件的开发列入需求** 如果希望所开发的系统是可复用的，就应该在需求中显式声明出来，并且应该使这种项目需求在客观上是可测试的。必须定义什么是可复用性和如何评估构件的可复用性，应该规定对构件所期望的复用范围(在项目内部或者跨项目的复用，在不同的 OS 上复用等等)，规定与某一复用标准的符合程度，规定构件所必须具备的文档，要求对软件的可复用性进行测试，以及要求开发者对构件进行维护等等。

(3)**领域分析的作用** 领域分析活动不同于通常对特定系统进行的需求分析，它是对特定应用领

域中已有的系统、预期的需求变化和技术演化进行分析，目的是标识出整个领域中通用的构架和相同的功能与接口。领域分析的结果将影响到系统需求的取舍，由此构造出的系统由于更适应变化的需求，日后被复用的可能性也更大。

·要评估领域分析的必要性和可行性。在适当的时机进行领域分析或者采用现有的分析成果。在利用现有领域分析成果时应该评价其适用性，即使整个构架未必可用，仍有可能复用某些标准的构件和接口。

·即使没有时间或能力进行完整的领域分析，仍然可以快速标识出本领域中可复用的子系统和接口（比如标准设备、通信协议、用户接口、应用算法等），为今后的开发储备"零部件"。

·在应用领域分析成果时，应该及时提供反馈，补充或改进其成分，促进领域模型随实际领域的演化而演化。

2.2 概要设计原则

可复用性在很大程度上受到软件设计方法和设计决策的影响。利用软件工程原则与方法进行软件的概要设计主要涉及以下几个方面。

(1)从需求分析向设计转换 设计阶段应该支持前面建立的可复用需求，并为实现这些需求建立良好的框架。高层设计活动将不同需求划分给软件构架中的不同构件。这种分配必须保持需求的完整一致并便于向实现转换。这一活动的重要性在于标识并定义了需要建造的构件及其接口，等到详细设计阶段再为特定的构件设计算法和实现构件所应具有的通用性及易修改性。

·将需求构件映射到一个或几个设计构件上，一个设计构件不要同时对应于可复用需求和特定于系统的需求，在设计阶段要保持二者的分离。

·从复用出发标识出设计中的并未在需求规约中显式声明的可复用构件。

·提供清晰的需求跟踪性，保证需求/设计/代码/测试之间的良好映射，这是在复用库中反映这种对应关系的前提。

·参数化标识了构件所能操作的数据的范围。将通用构件的可变特征作为参数，复用者可以根据需要确定参数以定制构件。检查各个构件是否能够参数化以使其更为通用。

(2)利用模型、构架和接口设计得到可复用的系统框架 模型用于声明软件需求，将需求与现有功能匹配，完成从功能需求到实现决定的映射。结构良好而一致的模型是开发者与复用者之间的共同语言。模型应当对应于真实世界中的概念，为提高可复用性，应将模型进一步通用化。例如：用"填写表单"作为数据查询程序的用户界面的模型是易于理解的。再将表单中域的数目、位置和数据类型定义为可变参数，即得到更为通用的模型，由此创建出可复用性较高的构件。

用分层的构架来分离注意的焦点，由此来隔离出可复用的子系统。分离的各层可以被独立替换以便升级、移植和集成。一个特定层的下层为其上层定义了一个虚机器，相邻层次之间依靠特殊的预定义接口进行交互，不可跨层访问。比如：可以将数据库管理系统(DBMS)自上至下划分为数据操纵层、数据存取层、文件I/O层和物理设备层四层。在分层时应该适时地采用或符合现有的标准构架，如ISO—OSI参考模型。

要为所有构件划分并细化接口。接口应该清晰、简洁、干净。构件的接口实际上决定了复用该构件的系统所必须遵守的约束条件，所有这些约束都应记录在接口的文档中。应该根据"必要且充分"的原则进行接口设计，既要求提供充分的通用性和灵活性，又不能过分复杂而难以复用，这要求在简单性和通用性之间作出权衡。同时应该尽可能采用现有的标准接口（如采用SQL语言作为RDBMS的访问接口）。标准接口规约本身就是可复用的实体，因此应使之与设计规约中的其他成分相分离，并以方便复用的方式表示出来。

(3)在设计时考虑构件被修改的可能性 构件有被修改后复用的可能。这可能是由于构件的运行平台、OS、窗口系统或通信协议的改变，或者系统功能的扩充和协作构件的升级等原因；当构件的生存周期较长时，往往有许多人维护、修改和改进过它，技术的演化也将导致对现有构件的修改。在设计构件时预见到可能的修改并采取相应的措施将大大提高其可复用性。具体措施有：(1)在考虑未来的修改时，应该权衡当前的开发成本与预期的复用收益。通常从一开始便加强易修改性会节省总开销。(2)应该标识出可被修改的地方并进行隔离，采用分层的构架和封装原则将修改局部化，甚至可以在文档中给未来的修改者以建议。

(4)采用适当的软件设计方法 采用适当的设计方法能够有效地组织软件系统并提高个体构件的可复用性。应该将构件封装为自包含的抽象实体，尽可能减少构件的外部依赖，在这方面 OO 方法能够更好地支持复用。传统的结构化功能分解方法强调构造低耦合、高内聚的模块，保持模块之间清晰的接口，但采用 OO 方法时仍然应该注意贯彻抽象和封装的原则，否则会减少复用的潜力。采用 OO 设计方法的具体措施包括：(1)将对象及其操作封装为一体，形成逻辑上完整的实体。将封装体的实现细节与用户的视角隔离，实现信息隐蔽。利用继承机制显式表达构件的共性。(2)考虑采用一种已经被接受的 OO 方法学，指导从需求分析到系统维护的全过程。(3)在设计上避免那些不能映射到编程语言的成分(如多继承)。

(5)选择 CASE 工具 CASE 工具帮助贯彻设计原则，实现一致的软件工程方法学。市场上有许多可用的 CASE 工具，各自支持软件生存周期的一个或几个阶段和某种方法学。困难在于如何从中选择一组工具，使之集成合作以全面支持某种方法学。最好是先考虑从需求分析到系统维护的所有活动，选择适合整个项目的方法学，然后再选择能够被一致地集成进来的工具。使用不适当的 CASE 工具比没有工具支持的效果更差。合适的编译器将帮助提高运行系统的效率、减少复用对性能的影响，因此同样应该加以考虑。

2.3 详细设计和实现

详细设计与实现活动必须支持此前作出的设计决定，并考虑提高每个构件的可复用性。实现可复用代码的技术与书写“好的”代码在标准上类似，模块性、易维护性和可移植性对代码的质量至关重要。

(1)从设计转换到代码 从设计到代码的转换必须遵循此前定义的系统结构并建造在其上，必须意识到代码质量对复用至关重要。构件必须格外健壮、结构清晰并易于维护。要采用适当的映射规则指导由设计元素到编程语言的映射，任何与原有设计的偏离都必须由配置管理过程接管和批准，以保证需求和设计规约得到及时更新。具体措施包括：(1)利用 CASE 工具辅助生成代码框架、函数原型和流程图，维护代码与需求、设计间的映射。(2)在选择编程语言时考虑它对代码质量和可复用性的影响，比如是否显式支持封装与抽象原则，是否支持参数化、模块化和注释，是否进行严格的类型检查等等。

(2)程序结构 对程序结构的选择决定了个体构件的可复用性。应将需要一起复用的实体组织到一起，尽量减少它们与外部的依赖关系。模块化、信息隐蔽、分而治之等传统的软件工程原则在此是适用的。

·每个构件都应当实现一个完整的对象，为复用者提供完备的创建和操作对象的方法(包括创建和初始化、终结、对象转换、状态转换、状态查询和输入/输出操作)。

·控制构件的实现与底层平台之间的依赖性；如果平台依赖是不可避免的，则尽量将它独立出来，与程序的其他部分隔离。在构件的文档中应显式声明对运行环境的依赖，说明如何发现和替换相关代码以适用于不同的运行环境。

(3)接口 构件的接口定义了它与外界的关系，建立了它被复用的框架。从某种意义上说，构件完全由其接口来定义。应该为潜在的可复用构件建立文档良好的接口规约，在代码前言和文档中列出它的接口清单，为每个具体接口提供简单的文本描述、类型规约、参数的取值范围和对越界参数的处理方法等信息。

·在接口的规约中可以按照以下分类描述接口的类型：

①构件由复用者调用的子程序调用

②构件由复用者的子程序调用

③构件是一个由复用者的任务激活的任务

④构件是一个激活复用者任务的任务

⑤构件与复用者的子程序共享内存

⑥构件是一个与复用者任务共享内存的任务

⑦构件通过一个共享文件与复用者通信，其中一方为读者、另一方为写者

⑧构件通过一个共享文件与复用者通信，双方都可以读写该文件

⑨构件通过消息传递机制或“邮箱”机制与复用者通信

·考虑用算法来描述入参数与出参数间的关系，从而精确地刻画构件的功能。

·构件只提供必要的通用性。在功能充分的前提下，少而简单的接口易于学习和理解，更容易用在新的环境中，也将更有效地支持复用。

·在设计构件的接口时应贯彻低耦合与高内聚

的原则，利用质量保证过程来显式地检查构件接口的充分性和必要性。

·向复用者提供处理边界情况的"钩子函数"(hooks)，允许复用者指定在发生越界时执行的动作。

(4)**参数化和例外处理** 参数化除了用于实现构件的功能，也有助于提高构件的可复用性。例外处理则可以确保构件足够健壮。

·提供在构件执行之前确定参数取值的方法，在源代码中明确标识出环境变量和系统参数。可以利用类属常量、编译时变量或源代码预处理器进行特定系统的配置，以利用编译优化能力，提高构件的执行效率。利用类属布尔常数或预处理器等机制，移去不必要的安全性检查。

·在对象定义中为每个元素提供缺省初值或者在创建对象时请求初值，以保证所有对象都被恰当地初始化。

·在参数无效或其它异常情况下提供一种将控制权返回给调用程序的机制。如果编程语言支持例外处理，就利用 Exception 向调用者报错，否则就利用过程参数来指定复用者的出错处理例程。如果没有类似机制，就利用返回参数指明出现的问题及其原因。

·检查构件正确操作所依赖的所有假设，利用例外处理实现"安全的"构件。如果引发例外处理的代价太高，则考虑使用一种让用户自己来改正错误的替代机制。

2.4 质量保证和测试

质量保证 QA 和测试活动有助于保证开发活动遵循所采用的标准，保证所开发的构件确实是可复用的。以下分别考虑评价活动、度量和测试过程。

(1)**评价活动** 与复用标准相符是 QA 审查的一个方面，跟踪性审查则保证不会在相邻开发阶段的转换过程中损失可复用性。QA 具有持续的职能，评价活动不能等到编码结束后才开始。在软件生存周期的每个阶段，QA 都执行适当的评价以确保该阶段的活动符合复用标准和用户需求。每次设计复审、每一特定的产品和阶段活动都应伴随有 QA 审查过程。对复用而言，需求分析和顶层设计阶段的审查尤为重要，因为它们是可复用构件开发的基础。具体措施包括：(1)不仅要审查产品，更要审查开发过程。比如是否使用 CASE 工具，是否起到预期的作用；构件的改动是否处于配置管理过程的控制之下，测试是否按计划进行了等。(2)利用审查清单(Checklist)和自动工具，能够覆盖所有关键问题，帮助记录审查结果和准备问题报告。(3)执行需求/设计/代码/测试之间的跟踪性 QA 审查。(4)评估本项目采用的复用标准的有效性，并进行改进。

(2)**度量** 度量主要起三个作用：度量构件的可复用性，评估复用项目并估测复用的预期收益，为复用库提供所需的信息。复用度量的收益者主要不是初始项目开发组，而是实施复用的整个组织。首先应该定义一组度量指标及其相关关系，然后再收集所需的度量信息。复用度量指标可能包括：为软件增加可复用性带来的附加成本，构件的复用次数，各种构件(分析、设计、代码、测试构件)的构成比例，构件被修改后复用的次数，复用对于项目进度的影响等。这些度量有助于使软件复用的成本合理化，为估计下一个项目的进度和成本提供经验。另外要收集复用库需要的度量信息，有条件的用户可以使用度量工具。

(3)**测试过程** 测试活动既要保证构件的质量和健壮性，又要保证满足整个项目的复用需求。应该将每个构件作为独立的产品进行比单元测试更加严格的测试。对构件的测试也应该作为构件的一部分来管理，不要在测试中加入构件本身没有的平台相关性和系统相关性，在修改构件时应该说明如何修改相应的测试用例，测试结果应该表明构件与需求相符。

·对构件的所有接口进行测试，测试数据应该覆盖参数取值的所有组合。

·测试每一条显式声明的可复用性需求。比如：对"可以将本构件移植到 ABC 平台上"这一需求的测试必须经过实际移植，这种测试的代价很高，但却是必须的。有时可以放宽类似的测试要求，改为"本构件应该遵循复用标准中所有关于可移植性的指导原则"。

·简单地说"某测试针对某构件"是不够的，测试应该直接针对构件的不同需求。这样就允许在复用者只使用构件的一部分功能时裁剪构件的测试集合。

2.5 文档

构件的文档构成其复用价值的一个关键部分，它除了有通常文档的作用外，还为复用者提供了明

确的指导。

(1)**应用传统的文档标准** 构件的文档(也是构件的一部分)会比大多数常规文档用得更多,它应该符合已被复用者团体所接受的文档标准。在开发构件的文档时,应该考虑复用者的特殊需求,同时要确保文档与代码的一致性。文档还必须能帮助复用者理解和使用构件。构件的文档本身也可作为文档构件,应该采用一致的组织结构和文档格式,具有清晰完整的风格和机器可读的形式。

每个构件的文档都应该是完备的和自包含的,能够分隔成与可复用代码构件一样的单元。这就是说,使用户易于得到只与他关心的构件有关的文档,不同构件的文档应该相互独立,避免过多的外部依赖和相互参照。

(2)**为复用库准备文档和复用者手册** 提交给复用库的构件需要附加特殊的文档帮助对构件进行分类、标记和提取,这些文档与项目开发所需的文档是分开的,通常包括构件功能的摘要描述和构件对环境的依赖(依赖某个 DBMS、某种数据格式、某个版本的编译器等),必要时提供构件的分类信息。这些文档的内容和格式由不同的复用库确定。另外,还要提供构件的评价信息(尽可能包括所有的质量度量和可用性度量),标明构件的显著问题以及对它的改进建议。在构件的摘要与代码前言中应该声明复用者受到的商业上和法律上的限制。如果构件在物理上不存放于库中,则说明获取该构件的方法。

复用者手册为希望评价、修改和使用构件的用户提供了额外的支持,下述 8 条为 NATO 推荐的手册大纲:

1° 介绍

· 本手册的目的 · 构件概述

2° 功能

· 操作 · 应用范围

3° 接口

· 构件的规约(标识所有外部可见的操作) · 向外的引用及其参数 · 接口分类(参见第 2.3 节(3)接口部分)

4° 效率

· 假定 · 对资源的需求 · 例外处理(构件对不正确输入的反应) · 测试结果(包括所有效率度量信息) · 已知的局限

5° 安装

· 如何初始化本构件(确定类属参数) · 接口(接口清单和用法) · 复用方案 · 修改方案 · 诊断过程(发生问题时做些什么) · 使用样例

6° 构件获取和技术支持

· 构件来源(如果构件不在复用库中) · 构件的所有权(法律上的或合同上的限制) · 构件的维护(从何处获得技术支持,如何联系)

7° 参考文献(包括所有可用的文档)

8° 附录(如果必要)

三、可复用构件库管理标准

构件库是一个包括人员、工具和过程的组织,主要目的是提供软件生存周期产品的复用机制以满足特定的软件代价-效益和生产率的目标,并作为开发可复用软件构件和基于可复用构件开发这两个生存周期的联系中介。这里将对 NATO 的可复用构件库管理标准作一简介。

3.1 构件的入库

一旦标明了对可复用软件构件的需求,就需要去获得满足这些需求的实际构件,使其遵循 NATO 的软件复用标准,并加入到构件库中,以等待软件工程师来检索,这就是构件的整个入库过程。

(1)**对构件的评价和分级** 构件库应对每个推荐入库的可复用构件进行定性和定量的评价,主要目的是:

· 澄清对构件的需求,保证能正确理解和实现构件提供者的愿望。

· 适当拓宽对构件的需求,以便能满足多个用户的需要。

· 对需求定量化,并预测该构件将带来的收益。

· 评估构件库在获得和对该构件入库时需付出的代价。

· 对推荐的构件按优先级分级,以便能有效地分配稀缺的资源。

通常构件提供者对构件的应用都有自己的想法,但构件库应保证该构件能应用到所有可能的目标程序和环境,如有必要,应拓宽对构件的需求。同时构件库应比较复用该构件所需的代价和所获得的收益,以此作为对构件分级的主要标准。一个理想的构件还包括下列特性:(1)满足用户已提出的特定需求。(2)广泛的适用性。(3)高度的可见性,即构件在一个广为人知的项目中使用过。(4)兼容性。(5)复杂

性低，即构件易于理解和组装。(6)解决了一些开发或测试中的比较乏味的问题。(7)具有吸引力，即构件采用了通用的或新的技术，良好设计的用户界面等。构件库应与构件提供者一起复审构件的评价结果，并将评价结果与构件保存在一起。

(2)**构件的获取** 构件库中的构件是来自软件生存周期各个阶段的可复用产品，为使用户能快速准确地检索到所需构件，并能正确地理解、安装和使用，构件在入库时至少应提供以下部分：

· 为复用者提供的有关构件特性、安装、验证及操作的完整指令(即复用者手册)。

· 构件的摘要信息。

· 构件的分类信息。

· 实际要复用的部分(源代码或文档)。

· 构件的测试计划、目标、脚本及预期的结果。

构件库中不需要存放每个构件的实际拷贝。当然拥有构件的一份拷贝会便于提取和管理，但在下列情况下只能保存构件的索引：(1)构件是由某个外部组织维护和发布的。(2)构件带有使用限制或许可协议。(3)构件是可执行的程序。由于可执行程序在传输时可靠性难以保证，因此只将源代码和文档存入构件库。

3.2 构件库的分类模式

刻面分类方法将关键词置于特定的语境中，从而避免了关键词的杂乱无章，而且它通过从不同视角(刻面)来观察要分类的项，从而导致了更加精确和准确的分类。多数专为构件的检索而设计的工具都采用这种方法。NATO 标准推荐在构件库中采用刻面分类模式，对构件的分类使用一组{刻面，刻面术语}对，也称为描述符，它们按下面的规则组合起来：

1. **刻面(Facet)**是一个单词或短语的固定集合，用于描述构件的某个方面或视角。

2. **刻面术语(Facet term)**是来自构件库特定刻面术语列表中的单词或短语。在新库的初始阶段，刻面术语个数迅速增长，此时构件库小组将逐渐熟悉用户首选的术语。但是刻面术语不久后必须保持稳定，此后只是偶尔加入新的术语。

3. 对每个特定构件的分类可以使用任意数量的刻面—术语对，即每个刻面可以出现任意多次(包括零次)。但是出于性能的考虑，对于"对象"或"功能"刻面应至少出现一个术语。

刻面的数量应比较少(一般是五到十个，最多可达到十五个)。每个刻面应该认真选择，对于用户来说是清晰和无二义的，但不要求不同刻面之间相互独立，也不必应用到每个构件上。

一个典型的刻面集合可以包括：

对象(object)——构件实现或操作的软件工程抽象，如 stack，window 或 sensor 等。

功能(function)——构件完成的过程或动作，如 sort，assign 或 delete。

算法(algorithm)——与某个功能或对象相关联的特殊方法名。例如对于描述符{function，sort}来说，bubble 即是一种算法。

构件类型(RSC type)——构件所处的特定的软件开发阶段，如 code，design 或 requirement。

语言(language)——构造构件所使用的方法或语言，如 Ada，C++，postcript，English 或 French。

环境(environment)——构件专用的任何硬件、软件或协议，如 UNIX，MS-DOS，postcript 或 sql。

通过对构件每个适用的刻面赋予适当的刻面术语即可完成对构件的分类，这样用户可以根据刻面术语来检索构件。下表是对一个排序例程的构件分类：

刻面	刻面术语
对象	address
对象	mail code
功能	sort
算法	binary sort
构件类型	code
语言	Ada

表中刻面"对象"出现了两次，没有出现刻面"环境"(该例程不依赖于任何特定的环境)。

对一个更通用的排序例程的分类如下所示：

刻面	刻面术语
功能	sort
算法	binary sort
构件类型	code
语言	Ada

此时"对象"刻面消失了，因为要排序的对象现在是任意类型的类属参数。

用户不必熟悉所有的分类术语，许多自动化的查询系统允许在查询时使用同义词。同义词(synonyms)是一组具有相同意义的刻面术语，共同表示一个概念。例如 delete，remove，erase 和 pop 都是

同义词。在一组同义词中选一个术语作为代表(representative)术语,用于实际的构件分类。其余的同义词存放于一个称为同义词词典(thesaurus)的列表中,以代表术语作为索引。对构件进行分类和查询时,这些同义词与代表术语是完全等价的。

在NATO标准中给出了对构件分类的基本步骤(即如何为构件选择术语):

1)检查构件所有可获得的文档(包括源代码),并写一个构件摘要的草案。此时应该对构件有了一个清晰的理解。

2)观看每个刻面的定义,考虑它是如何应用到构件的。根据每个刻面写下自己心中的术语,此时不用考虑刻面中已有的代表术语。

3)将你想好的术语与刻面中的术语进行协调。尽量用刻面的表示术语或它的同义词来分类构件,该术语必须是意义最清晰的,而且是无二义的。如果没有合适的术语,就需要向构件库建议加入新术语。

4)列出构件在每个刻面中的分类术语,如果是新术语,就用某种显著的方式标注出来。

5)如有必要,分类完毕后应修正构件的摘要。

6)若系统产生了一个关于该构件的"查询失败"日志,此时应根据用户的查找需求复审对构件的分类术语,可能它需要更新了。

构件库的分类模式应设计得易于增加和修改,这样才能增强它的描述能力。刻面分类模式分为刻面和术语两部分,它们都可以修改,但修改的环境是不同的。

刻面集合提供了分类模式的基本结构,因此刻面应该谨慎选择,以便能满足各种各样的分类需要。对刻面集合的修改需要严格的比较和分析,而且本身就是一项艰苦的工作,因为构件库中所有构件可能都需要重新分类。而对刻面术语的增加可以在对新构件分类时按需要随时进行,在增加新术语时也有如下原则:(1)确保新术语不是现有术语的同义词。若新术语与刻面中现有术语同义,则应作为同义词加入。(2)在同一刻面中的术语名称必须唯一,甚至不能与刻面中其它术语的同义词同名,但不同刻面中的术语及同义词可以同名。(3)新术语应与刻面中原有术语保持风格一致,如都用小写字母,或都使用单数名词。(4)加入新术语时,应同时加入该术语明显的同义词。另外,应定期删除系统中从未使用过的术语。

3.3 构件的配置管理

从一个RSC进入构件库中并可以被项目工程师获得的时刻起,它就被作为一个产品来管理了。构件库的增量式改进(incremental improvement)的方法和对可复用产品广为散布(dispersion)的特点都需要有效的配置管理。对RSC的配置管理,在很大程度上同所有其它项目产品的配置管理一样,是经过培训的专门的配置管理小组的责任。

构件库可以提供这样一种服务,即作为复用者和配置管理小组之间的联络员,同时也对维护进行协调。当构件库小组没有按常规对RSC进行维护时,RSC的客户(复用该构件的工程师)就有责任对维护问题作出快速而有效的回应。一般来说,应该由RSC的制作者或用户来进行维护。他们都应该有兴趣修正RSC的问题或增加新功能。由谁来完成这件事并不重要,构件库主要关心解决问题时不要重复劳动。

构件库小组应该记录下复用者对构件每次提取的情况,以此作为对构件进行配置管理和维护的依据,同时这对构件库性能的改进也很有帮助。大部分关于提取的信息可以由构件库的查询和检索工具自动收集,包括RSC的使用、问题报告、失败的查询和其它构件配置管理所需的数据,并以此来改进构件库的查询机制。构件库小组还应追踪以各种方式从构件库中提取过构件的复用者,并坚持用构件库工具追踪对RSC的复用,这样复用者就可以毫不费力地进行使用注册,同时也由此得到了复用者的合作。一般来说,每个复用者在查询和检索系统中都有一个帐户,每次提取时,复用者只需要提供一个"反馈日期"的协议,表明自己将何时反馈回该RSC的使用意见以及复用的经验。

NATO标准推荐将以下四个度量作为对构件质量和可复用性的评价标准:

· **检查次数(Number of inspection)**。复用者查找该构件的次数。

· **复用次数(Number of reuses)**。复用者实际复用该构件的次数。

· **复杂性(Complexity)**。一般是基于McCabe提出的软件复杂性评价方法。

· **问题报告次数(Number of problem reports)**。已知构件的显著缺陷或错误。

3.4 构件库工具的需求

构件库的工具分为两类，即由构件库工作人员使用的管理工具和由普通的用户使用的查询及检索工具。这里分别介绍NATO标准对这些工具的需求。

(1)**构件库管理工具的需求** 构件库管理工具将直接影响到库的使用效率，如果没有足够的工具支持，对构件的维护是不可想象的。

支持对构件的分类。工具必须用一种易于使用的方法将一组分类术语与构件相关联，并能支持基于这种分类的快速查找。刻面分类模式即是一种有效的分类方法。

支持对分类机制的维护。构件库的分类模式不能是静态的，随着各种技术的发展，刻面分类术语将不断动态地增删。

生成事务和状态报告。构件库必须能生成日志记录，并基于日志产生报告。报告至少应包括：(1)构件的使用情况。包括谁提取过，提取时间，用于什么项目，必要时还应包括用户的联络地址。这样便于发布问题报告、修改声明、需求评价及性能度量。(2)分类模式。按刻面列出描述过构件的术语。(3)用户信息。包括用户的名称、电话号码、邮件地址和正在从事的项目。(4)构件的历史。包括构件的修改日志(这将映射到问题报告中)、反馈日志(包括提取过但实际没有使用的构件)、与其它构件间的关系(尤其是继承和依赖关系)。(5)问题报告。对每个构件记录它较为突出的问题。

支持对构件和分类词汇的配置管理。构件库系统必须避免对库中构件未经授权的修改(注意用户可以对提取出的构件进行修改)。构件库工具必须记录构件的版本变化，以及用户针对构件的问题报告或为了增强功能而对构件进行的修改。同样，对用于分类构件的术语的修改也要严格控制。

支持对问题报告的追踪。构件库应作为解决构件问题的协调者。由于构件间存在各种关系，构件又是采用增量式改进的策略，构件库工具必须要考虑到一个问题的解决对其它相关构件的影响(尤其是具有继承关系的构件)。

支持对用户和项目的追踪。为了发布构件的新版本和问题报告，获取使用反馈，构件库必须要知道复用过构件的项目和用户的联络地址。

(2)**构件库用户工具的需求** 构件库的用户工具指一个为复用者提供的有效的构件查询和检索系统，这是一项很关键的技术，因为它要解决的是阻碍复用的一个最大故障，即减少查询的时间和代价，以及识别和获取可复用产品的困难。以下特性是用户工具必不可少的：

简易的规约。复用者不用学习一种新语言或熟悉一些复杂的界面即可描述出对构件的需求。

反复的精化。系统应该允许并协助复用者根据每次查询的结果逐步修正对构件的需求。

详细的信息。当查询到的候选构件数量已缩小到一个适当的范围后，复用者应有权访问构件的摘要、质量评价、软件度量及其它必要的信息，这样可以找出与需求最为匹配的构件。

四、软件复用过程标准

NATO的软件复用过程标准为希望使用复用库中的可复用软件构件进行复用实践的软件项目提供指南。复用库提供了复用的基础，但构件库本身并不能实现复用，这时还需要一个特定的过程来指导软件项目有效地使用构件库。这个过程与前面可复用构件的开发过程非常相似，正好是一个生产-消费的关系。

4.1 需求分析和领域分析

(1)**建立复用需求并尽早识别复用机会** 对软件系统的需求规约将直接影响到构件库中构件的复用机会，因此在需求分析阶段就应识别出复用的机会并建立对复用的需求。具体措施包括：

1)在提交具体需求之前检查构件库中是否有可以应用的构件。库中可用的构件及其结构和功能的知识将影响系统的需求规约。

2)避免对需求的过分细化。在需求分析中说明了过多的实现细节将阻碍对已有构件的复用，例如，要是需求分析中声明"从一个数组中读取……"，如果存在一个可以用较短时间或较少资源实现相同功能的队列或链表，可能就无法复用该构件了。

3)用与构件库分类模式一致的术语来说明系统需求。需求分析中使用的对象和功能名称应尽量与构件库中的分类术语保持一致，这样增大了找到构件来复用的可能性。

4)选择支持复用的开发方法与开发工具。大多数设计方法都支持复用，但对复用的支持程度不同。例如结构化方法支持函数级的复用，这种复用比较简单，运用也较为普遍，但主要用在详细设计、编码

和单元测试阶段。而面向对象方法扩展了复用在软件生存周期的适用范围，使得在需求分析阶段就可以开始复用。

5)用构件类的形式标识出每个具体的复用需求，但不要指定具体复用哪个构件，除非该构件的质量和可靠性是明确可知的。

(2)**领域分析的产品** 应该在需求分析阶段尽量复用已有领域分析的结果，因此有必要再介绍一下领域分析的过程和结果。在NATO标准中，领域分析过程主要可分为以下四个步骤：

1)知识获取：收集和分析特定问题领域的信息，并以领域内部和外部对象的形式来表示。

2)领域定义：对在知识获取阶段标识的对象进行分析，以决定该领域的具体边界。

3)建模：对属于该领域内的对象建模，以进一步理解它们在领域中的角色。

4)模型演化：对模型进一步地分析和精化，以开发出领域的分类模型，用于证明模型的结构和语义。

领域分析过程通常会产生下列输出：

· 功能层次：功能需求的一个层次性模型。

· 实体-关系模型：在领域中定义的对象间关系和接口的模型。

· 类属软件构架：将E-R模型中描述的对象表示为软件构件。

· 分类法：提供一个分类方案以定义领域中的对象。

· 标准需求：在该问题域中任何系统都必须实现的一套一般性的需求。

· 领域特定语言：用于描述和分类领域特定构件的词汇。

· 设计和开发准则：基于组成该领域的构架和构件的一个一般性的开发框架。

(3)**复用构架和子系统** 对大规模构件的复用，无论是复用构架还是复用构架的代码实现，都可以给自己的项目带来巨大的收益。而不认真对待，也会有很大风险。NATO标准给出了一些必须注意的建议：

(1)检查现有的可复用构架和子系统与所开发系统的接口和设计方法间的兼容性。

(2)评价复用现有的构架和子系统会产生的影响。在带修改的复用与重新开发之间作一个代价-效益的折衷分析，并根据识别出的构件评价复用对项目/进度的影响。

(3)将构架构件与需求分析的其它结果进行集成。当构件与开发方法相符合时(例如结构化的构件对应结构化的开发方法，面向对象的构件对应面向对象的开发方法)，集成是很简单的，只需将对象或模块放到系统中的适当位置。而当构件与开发方法不符时(如结构化构件与面向对象的开发方法)，集成是很困难的。NATO对此有如下建议：1. 将结构化构件封装成一个具有可见接口的抽象数据类型对象，以便集成到面向对象系统中。2. 将一个面向对象的构件在一个结构化模块层次的适当位置进行重复。这种集成的关键是构件的粒度，一般大粒度的构件(如子系统)更易于集成。

4.2 设计阶段

软件设计是发生复用的关键阶段，复用设计构件对实现代码复用是必要的，甚至在不复用代码时也是很有价值的。

(1)**在设计时要考虑完整地复用已有的系统和子系统的设计** 当现有系统或子系统的设计满足一定数量的需求时，应考虑将整个设计作为一个构件来复用，例如现有产品中的电子邮件系统、数据库查询或出错处理子系统等。复用的构件粒度越大，节省的时间越多。

在开始设计前检查可用的构件，并标识出可以复用的设计。注意要包括那些从已被复用的需求构件派生出的设计构件，因为这样的构件复用潜力最大。

评价每个构件的复用状态。基于构件的设计表示(如使用专门的设计语言或图形表示)和对需求的匹配程度列出构件的复用状态，如下表所示：

构件类型	设计表示	对需求的匹配情况
设计	表示匹配	完全匹配
设计	表示匹配	不完全匹配
设计	表示不同	完全匹配
设计	表示不同	不完全匹配
代码	无设计表示	完全匹配
代码	无设计表示	不完全匹配

根据复用的状态，可以决定构件是否必须经过修改才能复用。一般来说，对构件的修改越少越好，应只对那些不完全满足需求的构件进行修改。

只在绝对必要时才修改设计构件。例如，Ada中可对构件增加抽象和封装层次，以加入新的功能或

隐藏不需要的功能。这种通过重新包装来进行的修改是比较适宜的。

将设计构件作为"黑盒"复用。"黑盒"构件指使用构件提供的服务的用户不需要了解或只需要了解很少一点构件的内部工作,这意味着构件的所有服务都是通过标准的、可见的接口方法来获得的,不要试图通过跳过构件的外部接口来访问或修改构件内部的状态数据。

(2)将代码构件再工程为设计并复用该设计 这也是一种比较有效的复用方法,但在再工程时必须规定和遵循一套标准化的方法。例如,要从代码构件生成一个详细设计时,应首先从代码中抽取出PDL(程序设计语言,Program Design Language),将简单语句翻译为英语文字表示的语句,并保持程序的单元结构不变,复杂的语句结构也保持不变,将复杂语句中的条件部分翻译为英文表示。如果可能,应尽量使用自动化工具的支持。

设计对象必须与实际的代码对象保持一致,这包括:(1)与代码对象保持接口一致。(2)整个术语与代码对象保持一致。再工程后应将设计提交给构件库,以便能复用到其它工程中。该设计构件必须向构件库解释它的来源和开发历史,并维护好设计件与代码件之间的联系。

(3)将设计构件与整个系统的设计进行集成 在将设计构件集成时,要选择适合于整个项目的设计方法和目标的集成途径。集成的关键在于要保证现有构件的接口能支持系统的需求,并且要确保这些接口按照它们声明的设计和功能来使用和集成,这可能需要对原有构件进行修改或封装。

在设计集成时可以考虑使用CASE工具的支持,好处在于:(1)CASE工具常常对设计进行分析,收集度量信息,并自动检查一致性。(2)CASE工具可以生成一致的图形和文本化的文档,并且不需要用户过多的参与即可生成代码框架(code skeleton)。

当然,CASE工具也有一些缺陷,这主要体现在CASE产品是自成体系的,这种封闭的系统不会认可其它的设计方法和别的工具生成的设计产品,这使集成变得非常困难,甚至不可能实现。同时,CASE工具可能需要大量的用户培训和冗长的学习曲线。

4.3 实现阶段

实现阶段将继续贯彻在需求和设计阶段开始的复用,并可以识别出对底层构件的新的复用机会。这主要涉及到将代码构件集成到开发的系统中,同时构件的修改问题也是很重要的。

(1)代码构件的集成 将代码构件集成到程序中时,应该用一致的注释将复用的代码部分明确标识出来,这样可以避免对它们进行不必要的维护和测试。

为了简便安全地复用代码,应建立一种编码标准,这包括:

- 选取和加强命名规则,使得标识符名称不会相互冲突。某些实现语言尽管已淡化了这个问题,但缺少一个标准的命名规则会大大降低代码的可读性。
- 应该以一种标准而一致的方式使用任务和例外处理机制,这将简化和加速实现及维护过程。
- 避免使用全局数据项,尤其是共享的全局数据。全局数据的通讯可通过外部可见的接口来实现。

(2)对构件的修改 修改一个现有的代码件比重新写代码要好,但必须要小心地控制它的修改以避免不必要的开销和风险。修改时,尽量不要改动构件本身的代码,而是把它封装成一个新对象,在其中加入新的接口或屏蔽掉不需要的接口。修改构件后,可以提高代码的效率,并使之与项目的其它部分保持一致。但对构件的修改可能会引入新的错误,而且需要重新测试,对构件将来发布的版本也不能方便地升级。

对构件的所有修改情况都应该用文档记录下来,并且通知构件库自己已修改了该构件(包括修改的原因)。对来自构件库的构件照例要进行测试,包括单元测试和集成测试,不论对构件是否作了修改。

后记:NATO软件复用的三个标准分别从构件开发、构件库管理和利用构件开发新系统这三个方面阐述了一套软件复用的指导性原则,这三个方面相辅相承,互为补充,充分体现了以构件为中心的软件复用开发方式。但这个标准还只是操作性和经验性的,缺乏一个形式化理论模型的支持,而且对构件复用也没有自动化的工具支持。

参考文献

1 NATO Standard for the Development of Reusable Software Component

2 NATO Standard for Management of a Reusable Software Component Library

3 NATO Standard for Software Reuse Procedures