**文章题目：软件组件中的数据封装**

摘要：数据封装是面向对象编程中的常见属性。数据封装不仅有助于在现实世界中对事物进行建模，而且它也通过创建同一个类的多个实例来促进重用 ，每个实例都有自己的标识和私有数据。

对于CBSE（Component-based software engineering，基于组件的软件工程）来说，这种重用显然是关键的需求之一。

但是它必须和组件结合一起实现，这是CBSE的核心。

在本文中，我们通过扩展我们之前定义的组件模型，展示了数据封装如何与组件结合。

**1.介绍**

在面向对象编程中数据封装是对象的常见属性，它不仅可以用于对现实世界中的事物进行建模，而且还可以通过创建同一个类的多个实例来促进重用，每个实例都有自己的标识和私有数据。

对于CBSE，这种重用显示也是关键需求之一，因为组件被认为是多个组件实例的可重用模板。

但是，因为CBSE的核心是组合，所以问题是如何设计组合机制或者运算符，使组合的每个级别都可以进行数据封装，也就是说，如何确保组合创建的每个组合组件都封装自己的数据。

在本文中，我们认为数据封装和组合的结合在当前的组件模型中是不可能的；然后表明它可以用过扩展我们之前定义的组件模型来实现。

当前的组件模型可以在很大程度上分为两类：（1）组件是对象的模型，比如在面向对象编程中；（2）组件是架构单元的模型，比如在软件架构中。

这两类的示例EJB（Enterprise JavaBeans）和ADL（architecture description languages）。

在组件是对象的模型中，组件通过方法调用进行组装。然而这不是（代数）组合，因为通过调用对象2的方法与对象2组装的对象将导致两个对象，而不是一个（复合）对象。

因此，即使分别在对象1和对象2中进行数据封装，也没有可以正确组成对象1和对象2的组合机制，更不用说保留数据封装了。

在组件是架构单元的组件模式中，端口连接提供组合机制，并且可以提供组合。但是，数据封装并不总是可能被定义。事实上，数据在架构单元中的作用非常不明显。

这些单元既可以表示计算和数据，也可以只表示数据，并且通常数据封装不被视为组合的一部分。在架构单元具有数据端口的地方，可以认为这些端口代表数据封装。

然而，即使在这种情况下，也无法确定数据封装在每个组件级别上是否可行。

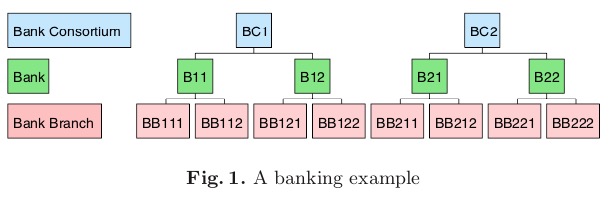
在本文中，我们描述了一种组合方法，它允许在组合的每个级别进行数据封装。我们的方法基于一个组件模型，其中组合运算符是第一级的组件（first-class citizens），并且他们还使每个组件实例（特别是复合组件实例）能够封装其自己的数据。

**2.具有数据封装的组合**

组件用于组合，因此它们应该是组合的，即如果C1和C2是组件，则C1和C2的组合成的C3也必须是组件。此外，组合物应定义为组合运算符（composition operator），将组件组成新的（复合）组件。换句话说，在组件模型中，组件和组合运算符应该是一级概念（first-class citizens）。任何组件模型都应具有此属性。

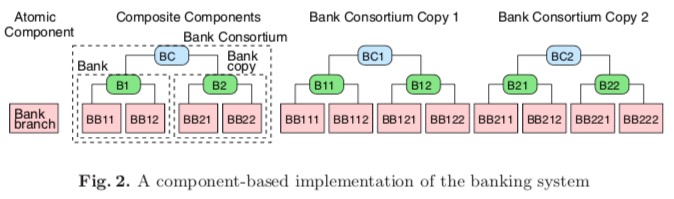
一个好的组件模型也应该允许组件封装数据，但要真正实用，它应该在每个组合级别都这样做。我们在[6]中提出了一个组件模型，在本文中，我们描述了如何扩展该模型并使用它来实现这种数据封装。

为了便于说明，我们将假设一个简单的银行示例。



示例1.考虑一个银行系统具有两个银行联盟BC1和BC2，分别由银行集合{B11，B12}和{B21，B22}组成。紧接着，每个银行由一组分支组成，例如 银行B21有分支{BB211，BB212}，依此类推。

在我们的组件模型中，我们将通过使用组合运算符组合组件来构建系统（参见图2）。



在我们的模型中有两种组件：原子组件和复合组件。复合组件是由原子（和其他复合）组件构建的，但所有组件都只是模板，除了常量之外没有数据，但是它们会提供服务代码。由于组件是模板，因此制作组件副本是有意义的。对于银行系统，针对特定的银行联合体，银行分支可以是原子组件，其具有用于诸如取款，存款和支票余额的常规操作的代码。这为所有银行分行提供了一个模板，因此我们可以构建许多银行分行（所有BB）作为该组件的副本。

此外，在我们的模型中，可以从不同的组件副本创建实例。例如，银行分行BB111和BB211，它们是BB11的不同副本（其又是银行分支原子组件的副本），并且属于不同的联盟，可以用它们自己的地址，分类代码和客户账户进行实例化。

类似地，可以通过组成银行组件来构建银行联盟组件。在图2中，使用合适的合成运算符，由银行组件副本组成的银行组件产生银行联盟组件。这个新组件可以进一步用自身的副本组成，以构建银行系统。

值得注意的是，在银行示例中，只需要一个原子组件（银行分支）和一个组合运算符就饿能构建整个银行系统。我们实现中的每个组合都会生成一个正确定义的组合。显然，我们的模型提供了适当的组合机制。问题是我们是否也可以对其封装数据。

**3.我们的组建模型**

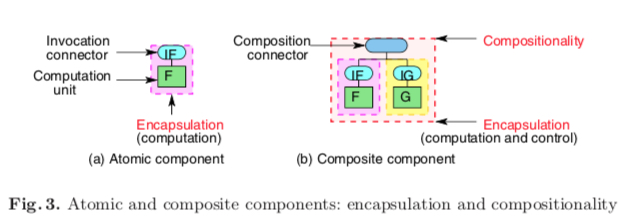
在我们讨论如何扩展组建模型以启用数据封装之前，在本节中我们将简要概述我们在[6]中提出的模型。

在我们的模型中，我们有两种基本实体：（i）计算单元，和（ii）连接器。计算单元U封装计算，提供了一组方法（或服务）。封装意味着U的方法不会调用其他计算单元中的方法，同时当被调用时，它们的所有计算都发生在U中。因此，U可以被认为是一个不调用其他类中的方法的类。

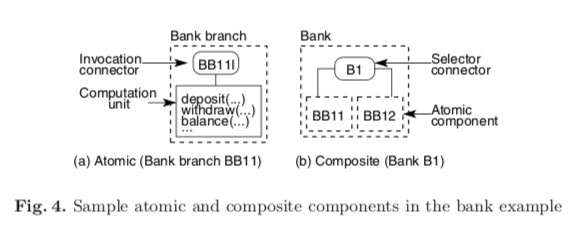
连接器有两种：（i）调用，和（ii）组合。调用连接器连接到计算单元U，以便提供对U的方法的访问。

组合连接器封装控件。它用于定义和协调一组组件（原子或复合）的控制。例如，组成组件C1,…,Cn的顺序控制程序连接器可以按同样的顺序调用C1,…,Cn的方法，另一个例子是选择器连接器，它选择（根据某些指定的条件）它组成的一个组件，并调用它的方法。

组件根据计算单元和连接器定义。有两种组成成分：（i）原子,和（ii）复合（见图3）。原子组件具有调用连接器的计算单元组成，该调用连接器提供组件的接口。复合组件是一组由组合连接器组成的组件（原子或复合组件）组成的。合成连接器提供复合的接口。



举例说明，在示例一的银行系统（见图2）中，原子组件银行分支BB11，可以如图4（a）所示定义，具有调用连接器IBB11，以及具有该分支连接器的计算单元，有存款，取款，还贷等方法。复合组件银行B1，如图4（b）所示定义，作为使用选择器连接器的原子部件BB11和BB12的组成（为方便起见，也可以）。图2中的银行联合（BC）组成部分也可以使用选择器连接器（来自银行）组成，因为联盟必须选择具有客户账户所属分支的银行。

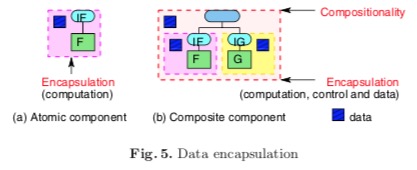


在我们的模型中，调用和组合连接器形成层次结构[8]。这意味着组合以分层方式完成。此外，每种组合物都保留了封装。这种组合性是我们组件模型的显著特征。一个原子组件封装了计算（图3（a）），即计算过程被他的计算单元封装。复合组件封装了计算和控制（图3（b）），它封装的计算是封装在其子组件中的；它封装的控件是由其组件连接器封装的。在复合中，子组件中的封装被保留。实际上，连接器的分层特性意味着组合组件与其子组件自相似；这个属性为分层组合提供了基础。

在下一节中，我们将展示如何扩展我们的模型以包含数据封装。

**4．数据封装**

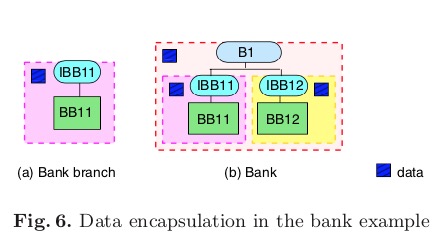
我们的数据封装方法如图5所示。基本上，我们希望扩展我们的模型（图3）以允许每个原子或复合组件在设计时为其自己的数据定义占位符，这些占位符由图5中的方块图案表示。



因此，在我们当前的模型中，复合封装了计算和控制（见图3（b））,在扩展模型中，复合组件另封装了数据（图5（b））。

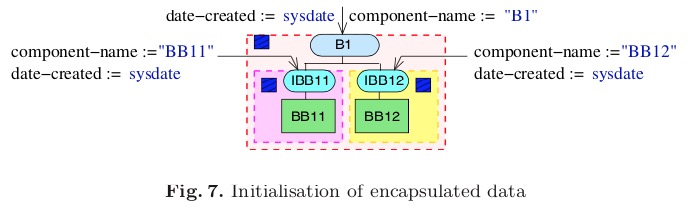
我们的扩展以组件的构造函数为中心。我们希望能够在设计时制作组件的副本，以便它们都具有相同类型的数据占位符。组件的副本也将具有与原始组件相同的构造函数。在运行时，我们希望能够使用组件的构造函数创建组件或副本的实例，并且我们希望能够使用实际数据初始化其数据占位符。这样不同副本（原子或复合）的实例可以封装它们各自的私有数据。

针对银行的例子，正如图6所示，分支组件BB11封装其客户的数据。使用合适的组合连接器，可以从分支组件BB11和BB12构造组件，其中两个组件实际上是银行分支组件的副本。如图6（b）所示，银行（复合）组件封装其自己的数据，该数据与其子组件的数据分开。后者在每个分支子组件中的封装被保留在组合组件中。



实例化组件/副本的过程通常需要初始化封装数据。这些数据可以是组件设计阶段定义的常量；或是在实例化时创建的数据。因此，组件必须具有能够执行数据初始化的构造函数。在我们的模型中，为了这个目的，我们在组件构造函数中使用数据构造函数。数据构造器可能需要在数据初始化过程中从外部资源读取数据。因此，连接器必须具有数据I / O语义才能执行其任务。在我们的模型中，连接器要能够执行数据I / O操作。

对封装在银行示例中的数据进行初始化如图7所示。必须使用作为常量字符串的分支名称初始化银行分支组件，因此必须读取并保留有关拥有分支组件实例的进程的数据。后者可能包括日期，进程帐户和网络信息等。



为了简单起见，我们假设只记录其实例化日期。分支组件的调用连接器在组件构造运行时期间访问这些数据值。由两个分支组成的银行组件也具有其自己的独立构造器，用来执行其数据初始化操作。这些操作包括：设置银行名称以及在实例化日期和其他系统数据上进行记录。就数据而言，银行组件及其子组件独立设置其初始数据，每个组件使用自己的构造函数。在图7中，银行组件读取B1银行名称（B1）和日期（sysdate），其子组件在创建实例时执行类似的数据初始化操作。B1的每个连接器执行的初始化由箭头上的数据名（arrows bearing data names）表示。

数据封装对复制重用而言是一个有价值的概念。封装了数据（除了计算和控制）并具有自己的数据构造函数的组件是一个适合复制的单元。组件在设计阶段将其本地数据指定为占位符，复制组件会创建一个封装自己数据（规范）的新组件。复制的数据构造器执行每个副本的数据初始化。因此，数据封装可以复制我们的组件。

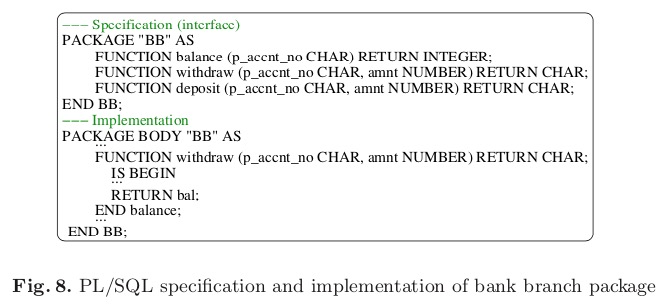
此外，在基于组件的系统中，通常希望创建一个组件的多个实例或者创建一个组件的不同副本的实例。在银行系统中，所有分支都是分支组件的副本，实例化所有这些分支必须是可能的。实际上，如果每个分支没有将其数据与数据构造函数一起封装，则不可能从不同的副本中获得不同的实例。在我们的模型中，甚至可以制作同一组件的许多副本，因为实例维护自己的数据。

**5.实现**

在本节中，我们将展示如何使用银行示例在扩展组件模型中实现数据封装，并展示如何从两个连接器和一个计算单元构建银行系统。首先，我们基于Oracle数据库10g 10.1.0.4.0企业版的使用概述了扩展组件模型的实现。数据库语言选择了自然语言，因为我们关注的是这里的数据。

**5.1.连接器和组件**

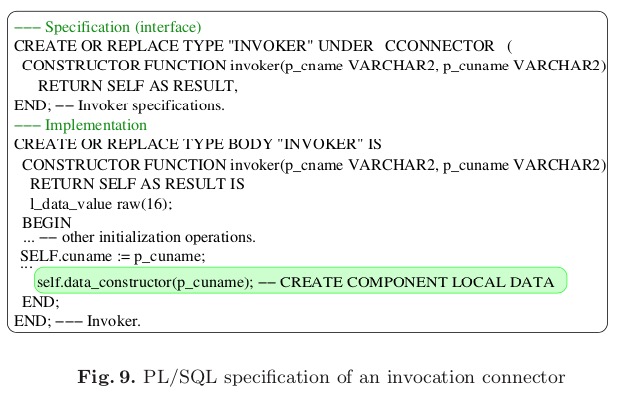
我们已经将扩展组件模型实现为一个存储库（repository），它在设计时和运行时都储存计算单元以及连接器和组件（原子和组合）的模板。虽然存储库依赖于Oracle在计算单元上维护的元数据，但是连接器和组件模板作为记录存储在数据库表中，例如， CONNECTORS，COMPONENTS，ENC-DATA，ENC-DATA-INST等。存储库提供诸如创建和复制组件，搜索，浏览和组件实例化等服务。组件，连接器和计算单元用PL / SQL编程语言编写。PL / SQL是一种4GL编程语言，Oracle使用它来指定程序的接口，因此它显然是我们的选择。但是，PL / SQL缺乏对反射的支持，而这对于实现我们的组件模型是必要的。因此，我们对存储库的实现必须弥补这一不足。



在我们的实现中，计算单元是Oracle包（package），连接器是Oracle对象类型。

包是一种数据库结构，它对逻辑上相关的PL / SQL类型，变量和子程序（函数和过程）进行分组。它可以实现我们组件模型中的一个计算单元，前提是它的子程序不会调用其他包中的子程序。包有两个部分：规范（specification）和主体。规范是包的接口，它声明（publishes）了类型，变量，常量，异常，游标（记录集）和子程序；主体实现了游标和子程序。图8显示了一个PL / SQL包的示例。它是银行示例中银行分行的包的大纲。

对象类型是一种用户定义的复合数据类型，表示数据结构和操作数据的子程序。它就像面向对象编程语言中的对象。与包类似，对象类型由两部分指定，即规范（接口）和主体（实现）。图9是一个对象类型的示例，它是调用连接器类型的概述（Invoker）（见下文）。



我们使用对象类型来分层实现连接器。层次结构的根是超类型CConnector，有三个子类型Invoker，Selector和Sequencer，分别用于调用连接器，选择器连接器和定序器连接器。举例说明，在图9中，Invoker类型继承自CConnector（由关键字UNDER表示）。

超类型CConnector提供了数据最重要的过程：数据构造函数的实现。在这个过程中构造函数根据组件中的数据规范创建和初始化数据实例。CConnector的所有子类型的构造函数调用超类型的数据构造函数来创建和初始化其组件的数据实例。在子类型中，构造函数首先创建并初始化其内部数据。然后，它调用数据构造函数，该过程从存储库中读取其组件数据规范; 创建和初始化所需的数据实例。在图9中可以看出，针对Invoker的构造函数， 高亮显示对数据构造函数的调用。

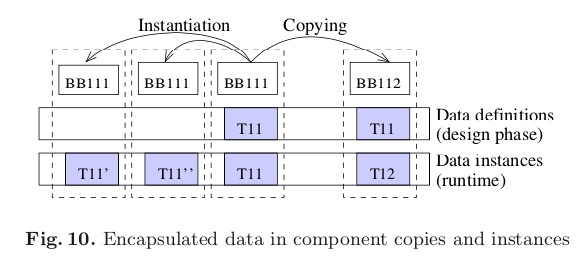
原子组件由用于计算单元的包和用于调用连接器的对象类型构成，调用连接器保留了对计算单元名称的引用。例如在图9中，为变量cuname分配了用户提供的计算单元名称p\_cuname。通过执行生成规范的过程并将其作为记录存储在相关的存储库数据库表中，该组件在存储库中被创建。这些记录包含有关组件，其调用连接器，接口，服务及其返回类型和参数等数据。此时组件还处于设计阶段。复制组件等同于检索所有组件规范（记录）并将其存储在新组件名称下，新名称应该是唯一的用户或系统提供的字符串，并且计算单元必须存在于存储库中才能成功完成此操作。

复合组件由现有组件构成。构成的过程采用三个参数：新组件的字符串名称，包含子组件名称的列表和连接器类型。可以组成的组件数量取决于组合连接器的语义（semantics）。此过程以与构造原子组件的过程类似的方式生成并存储构造组件的规范。在我们的存储库中，复合组件始终由其他组件的副本构成。也就是说，如果F和G是存储库中的组件，那么在F和G的复合H中，子组件必须是F和G的副本，每个都由存储库重命名。因此，存储库是以任何合成操作都保持组合过程中涉及的原始组件不变的方式实现的。

**5.2.数据封装**

在构建组件并将其存储在存储库后，可以使用一组操作来支持用于组件封装的数据规范。组件的数据规范包括有关组件的封装数据元素的属性的信息，例如它们的名称，类型，状态（持久或临时），初始值，初始操作等。存储库在运行时使用规范来创建和管理数据实例。

在设计阶段，存储库的操作在名为ENC\_DATA的存储库数据库表中生成，更新和存储组件的数据规范。将数据元素封装在组件中的操作采用参数（正整数值）来指定数据实例化的顺序。数据元素的名称在拥有它们的组件的范围内必须是唯一的。数据元素条目的数据属性（Data properties of a data element entry）使用用于定义任意数据元素的通用过程或者兼容特定类型的过程来设置，设置的属性有整数和字符串数据元素等。复制存储库中的组件也会复制其数据规范， 因此，对于可重复使用的组件，数据初始化必须延迟到运行时阶段，除非数据不是特定于应用程序的。



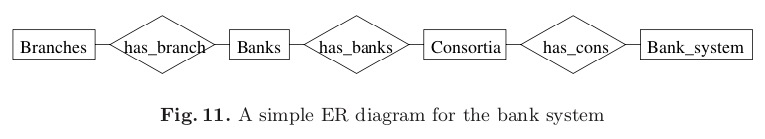
在运行时，组件构造函数用于创建组件的实例，此实例在我们的运行时系统中通过CID（组件ID）唯一标识。在构造过程中，将调用数据构造函数过程。它检索存储在ENC\_DATA表中的组件数据规范，创建并初始化数据实例，并将这些实例存储在全局临时数据空间中。数据元素的初始化可以通过简单分配常数或计算值来实现，计算值指定为脚本（匿名PL / SQL块），并在运行时的组件构造期间由系统执行。存储在数据空间中的数据实例可通过引用提供给它们的组件，存储库通过其CID识别它们。全局数据空间被实现为一个Oracle全局临时表。当组件的父进程终止时，与特定组件相关的此类表中的条目将自动进行垃圾回收。组件的进一步实例化为每个组件实例创建新的，相异的和独立的数据实例。组件的不同实例在运行时维护自己的数据实例。图10展示了在银行示例中的组件副本和实例的数据封装。分支机构BB111拥有其数据定义T11。副本BB112具有其自己的数据定义的副本T11。BB111的两个实例拥有两个不同的数据实例：T11’和T11’’，这两个实例以相同的数据开始，但随着时间的推移，它们的数据会变得不同。

组件构造器必须能够执行数据初始化所需的数据I / O操作等，这些读写操作被从各种数据源（包括全局临时数据空间）用于输入输出数据的数据连接器实现。存储库自动为每个封装的数据元素，方法参数和返回值创建数据连接器。对于标准数据源（如关系数据库），可以使用一整套数据连接器并随时可以使用，而对于非标准或未知的数据源，数据连接器作为存根（stubs）被创建，必须在运行系统之前手动更换。可以将用于非标准数据源的数据连接器添加到存储库，并在构建新系统时重复使用。目前的组件模型很少支持关系数据源，只有.NET支持额外的XML数据源。

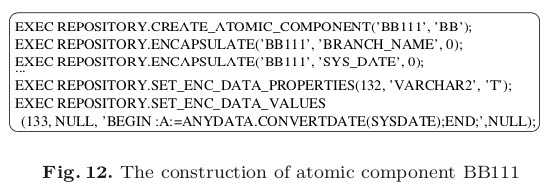
**5.3银行系统**

现在，我们可以详细地基于示例1（图1）中的银行示例来设计银行系统的实现。特别是我们演示了数据封装以及我们的组件模型如何实现其组件的复制和多个实例化。我们还展示了模型中的组合方案在每个组合级别保留并传递数据封装。

考虑由2个联盟BC1和BC2组成的银行系统，如图1所示，具有简化的实体关系（ER）图，如图11所示。

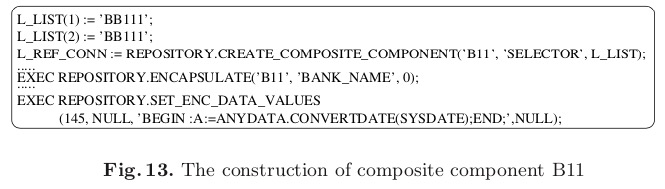


为简单起见，我们选择在每个组件中仅封装三个数据元素，即组件名称，实例化日期和数据详细信息。银行分行单独封装其名称（BRANCH-NAME），日期和客户数据（CUSTOMERS）。银行组件将数据封装在其分支机构（BRANCH）以及银行名称（BANK-NAME）和日期。一个银行联盟持有其成员银行（BANK）的本地数据，联盟的名称（CONST-NAME）和日期。银行系统保存其拥有的所有银行联盟（CONST），系统名称（BNET）和日期的本地数据。日期（在每个实例中命名为SYS-DATE）是存储组件实例化日期的占位符，它包含了初始化脚本（PL / SQL匿名块），可在运行时由系统执行以返回实例化日期。在图11中，关系has-cons，has-banks和has-branch分别是封装在组件银行系统，银行联盟和银行中的数据。

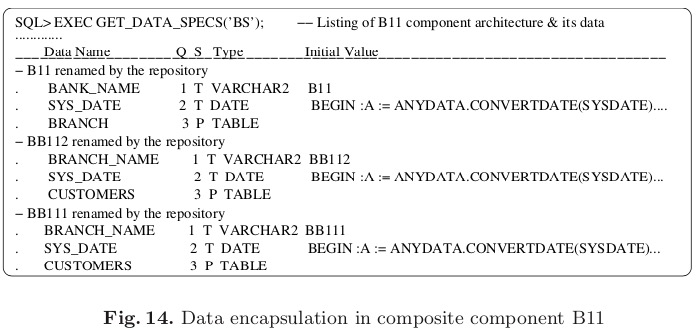


为了构建系统，我们从一个计算单元BB（一个Oracle包），一个调用连接器和一个选择器组合连接器开始。这三个要素足以构建上述整个银行系统。我们的第一个组件是银行分行的原子组件（BB111），它由BB和一个调用连接器构成。在图12中，第一行命令创建组件，第三行命令定义其封装数据元素之一的SYS\_DATE，此变量以唯一ID（整数）存储在存储库中，该ID在运行时用于初始化数据元素。最后一行命令为SYS\_DATE分配一个脚本（'BEGIN ... END;'），该脚本在执行时返回系统日期。每个分支组件都需要此脚本，因此在设计阶段已将其赋值给SYS\_DATE。BB111的其余封装数据在最终系统构建之前不会被初始化。

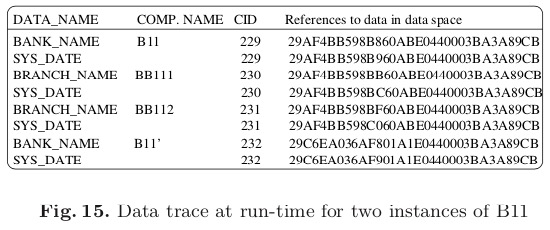
我们假设所有银行分行的业务逻辑是相同的。因此，银行组件可以由BB111的副本构成。在图13中，我们使用选择连接器构造银行组件B11以组合BB111的两个副本。生成的复合组件封装了自己的数据。它的SYS\_DATE也使用与BB111相同的脚本进行初始化。封装在B11中的数据独立于其子组件的数据。图14显示了B11规范的列表，其中封装在其子组件中的数据不受组合物的影响。组合过程将子组件的封装数据和传播的数据（propagated data）封装保留到下一级。



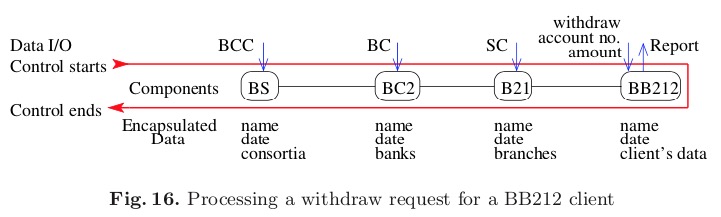
类似地，可以以相同的方式创建银行联合组件BC1，还定义了它的封装数据。最后，我们从BC1的两个副本和其本地日期的定义创建银行系统组件（BS）。通过此步骤，系统创建完成，可以继续进行数据初始化。数据初始化基于存储库提供的有关组件封装数据的知识，执行与初始化SYS\_DATE类似的许多步骤以使BS准备好运行。



从上面概述的设计阶段过程可以清楚地看出，我们的模型中的组合方案如何支持数据封装。 组合保留数据封装并传播它。 此外，通过复制重用已经在BS的创建中得到证明：所有银行分支都是BB111的副本，银行是B11的副本，而银行联盟则是BC1的副本。



在运行时，在实例化BS之前，必须首先用数据初始化BS及其子组件。在此之后，创建BS组件会为其所有子组件创建实例。每个组件构造函数都创建其独立的数据实例，并将它们存储在由CID标识的数据空间中。从运行时系统中为B11和B11'的两个实例提取的数据跟踪显示了每个组件的不同数据实例（图15）。



系统现在可以接收客户请求，例如取款，存款，余额等，如图16所示。通过ATM获取客户帐户信息，顶级连接器（top-level connector）读取客户的联盟代码（BCC）以决定哪个银行联盟直接控制；联盟的顶级连接器（BC2）读取银行代码（BC）以选择银行；然后，银行读取客户端排序代码（SC）以确定客户的银行分行；银行分行读取所请求的服务，帐号和金额，处理客户的请求并返回报告。在此过程中，每个组件的顶级连接器执行必要的I / O操作，以协调到右侧分支的控制流。 图16展示了传送到分支BB212的客户端请求（取款）。

**6.讨论和相关工作**

略

**7.结论**

略