УДК 681.3

В.Н. Грищенко

ФОРМАЛЬНЫЕ МОДЕЛИ КОМПОНЕНТНОГО ПРОГРАММИРОВАНИЯ

Предложен подход к построению формальных моделей компонентного программирования как основы создания компонентной теории. Рассмотрены модели компонентов, компонентных сред, определены внешняя и внутренняя компонентные алгебры. Проанализированы различные аспекты предложенных моделей и их связь с практикой программирования.

Введение

В своем развитии, как отдельное направление в теории и практике программирования, компонентное программирование основывалось на различных концептуальных взглядах, методических основах и подходах, формальных моделях и методах. Такая ситуация достаточно часто встречается в различных отраслях науки и свидетельствует о том, что соответствующая область знаний находится в процессе формирования и постепенного повышения уровня формализации. И только в последние годы концептуальные основы были достаточно четко сформулированы и сформировался необходимый базис для построения теории компонентного программирования. Приведем основные концепции.

1. Компонентное программирование — это по своей сути композиционное программирование, где в качестве базовых элементов композиции выступают программные компоненты, обладающие определенными свойствами и характеристиками. Компоненты являются базовыми объектами, для которых определяются методы агрегирования в более сложные структуры и правила взаимодействия при построении интегрированных сред и систем [1].

2. При построении формальных основ компонентного программирования одинаково важную роль играют как практические аспекты его применения, так и результаты соответствующих теоретических исследований [2]. Игнорирование этой концепции часто приводило к разработке теорий и формальных моделей, которые либо не могли учитывать всех его особенностей, либо не обеспечивали достаточно гибкий базис дальнейшего развития теории, либо просто не были востребованы теоретиками и практиками программирования.

3. Компонентное программирование — это самостоятельный стиль программирования, дополняющий и использующий результаты других дисциплин и подходов программирования: модульного и сборочного программирования [3, 4], объектно-ориентированного подхода [5], повторного использования знаний и программных объектов [6,7] и т.д. Оно имеет свою концептуальную базу, теоретическое обоснование моделей и методов, соответствующие методологии и инструментальные средства.

4. Компонентное программирование — один из подходов к созданию программных систем, и поэтому оно также характеризуется основными концептуальными понятиями программной инженерии [7], присущими другим программным подходам. В частности, для него имеется свой жизненный цикл с особенностями проектирования на отдельных фазах и этапах [8], методология тестирования отдельных компонентов и компонентных конфигураций, методы обеспечения качества и повышения надежности и др.

Эти концепции обеспечивают цельный взгляд на суть, особенности и роль компонентного программирования в современной теории и практике программирования. Поэтому любая теория должна в той или иной мере учитывать приведенные выше аспекты

и основы. В настоящей статье предлагается и описывается такой подход к разработке теоретического базиса компонентного программирования с учетом его современного состояния, перспектив развития, актуальных задач теории и практики программирования.

Определение базовых понятий и терминов

Основываясь на приведенных выше концепциях, введем определение компонентного программирования.

Под **компонентным программированием** понимается метод создания программных систем, основанный на применении концепций композиции на всех этапах жизненного цикла, где базовым элементом композиции есть специальная программная единица — компонент.

Из этого определения следует, что суть данного метода определяется свойствами и характеристиками компонентов, а также операций с ними, конечная цель применения которых состоит в построении компонентной программы. Систематизируя и упорядочивая существующие теоретические и практические результаты [9], можно ввести следующее определение программного компонента.

Программный компонент или просто **компонент** — это независимый от языка программирования, самостоятельно реализованный программный объект, который обеспечивает выполнение определенного множества программных сервисов и представлен как взаимозаменяемый контейнер, доступ к которому возможен только с помощью интерфейсов, определяющих его функциональные возможности и порядок обращения к его операциям.

В компонентном программировании компонент является неделимой и инкапсулированной сущностью, удовлетворяющей определенным функциональным требованиям, а также требованиям архитектуры, структуры и организации взаимодействия в компонентной программе. Поэтому совокупность требований к конкретному компоненту на самом деле определяет классы компонентов, то есть все множество компонентов разбивается на совокупность классов эквивалентности относительно требований для каждой создаваемой компонентной программы. Любые представители этих классов обеспечат построение данной программы при условии, что для этого над ними выполняются допустимые операции. Исходя из этих предпосылок, введем определение компонентной программы.

Компонентная программа — совокупность компонентов (представителей из соответствующих классов эквивалентности), необходимых для обеспечения функциональных и других требований, которая построена и функционирует в соответствии с определенными правилами создания компонентных конфигураций и компонентного взаимодействия. Эти правила составляют основу компонентной модели.

Компонентная модель — это определенная совокупность архитектурных, структурных и поведенческих требований к компонентам, а также правила взаимодействия и построения компонентных конфигураций. Данное определение более сильное, чем соответствующее понятие, встречающееся в практике программирования. Например, для CORBA [10] только сравнительно недавно была специфицирована компонентная модель, хотя история ее практического применения насчитывает около 10 лет. Одной из первых среди формализованных и специфицированных моделей была Enterprise JavaBeans (EJB) [11], которая хорошо реализована для различных платформ. Для технологии COM [12] если и можно говорить о какой-то специфицированной модели, то только в контексте двоичных реализаций компонентов для одной платформы WINDOWS.

Компонентная модель является основой построения компонентных конфигураций. Введем соответствующее определение. **Компонентная конфигурация** — это архитектурно упорядоченная относительно определенной

компонентной модели совокупность компонентов, построенная и взаимодействующая согласно требованиям и правилам этой модели. Необходимо отметить следующие два следствия из этого определения.

1. Компонентная конфигурация определяется в рамках требований к конкретной компонентной программе. Не каждый набор компонентов будет определять конфигурацию. Необходимо (согласно определению), чтобы между ними существовали взаимодействия, а для этого они должны иметь согласованные интерфейсы, что выполняется не для каждой пары компонентов. В свою очередь совокупность требуемых интерфейсов должна определять функциональные свойства компонентной программы. Этим и обусловливается относительность компонентной конфигурации.

2. Количество компонентов в компонентной конфигурации может быть произвольным, но, согласно предыдущему следствию, определяется относительно требований для конкретной компонентной программы. Минимальное значение равно единице, а максимальное — не больше числа классов эквивалентности для всего множества компонентов (как следует из дефиниции компонентной программы).

Приведенные выше определения являются базовыми для формализации компонентного программирования. Рассмотрение вопросов конкретных архитектур моделей, аспектов практической

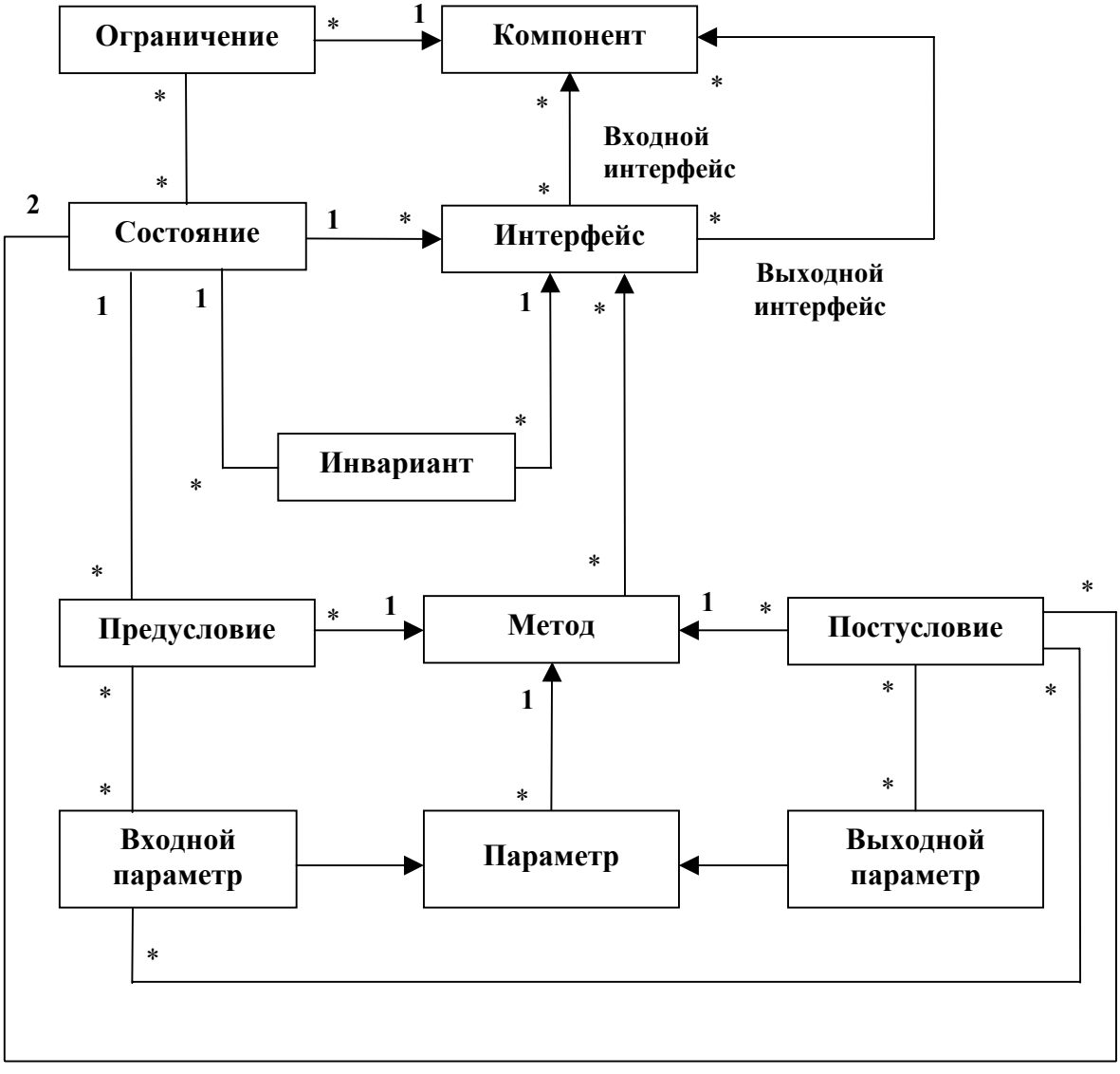

Рис. 1. Модель семантической спецификации компонента

реализации требует дополнительных определений, которые будут вводиться по ходу статьи в соответствии с контекстом изложения. Относительно определения понятий, связанных с самим компонентом (интерфейсы, реализации, методы и т.д.), то они достаточно хорошо изложены в существующей литературе, например в [9, 13]. Взаимосвязь большинства этих понятий иллюстрирует рис. 1, где представлена модель семантической спецификации компонента.

Обобщенная архитектура компонентной среды

Современная архитектура компонентной среды является одним из

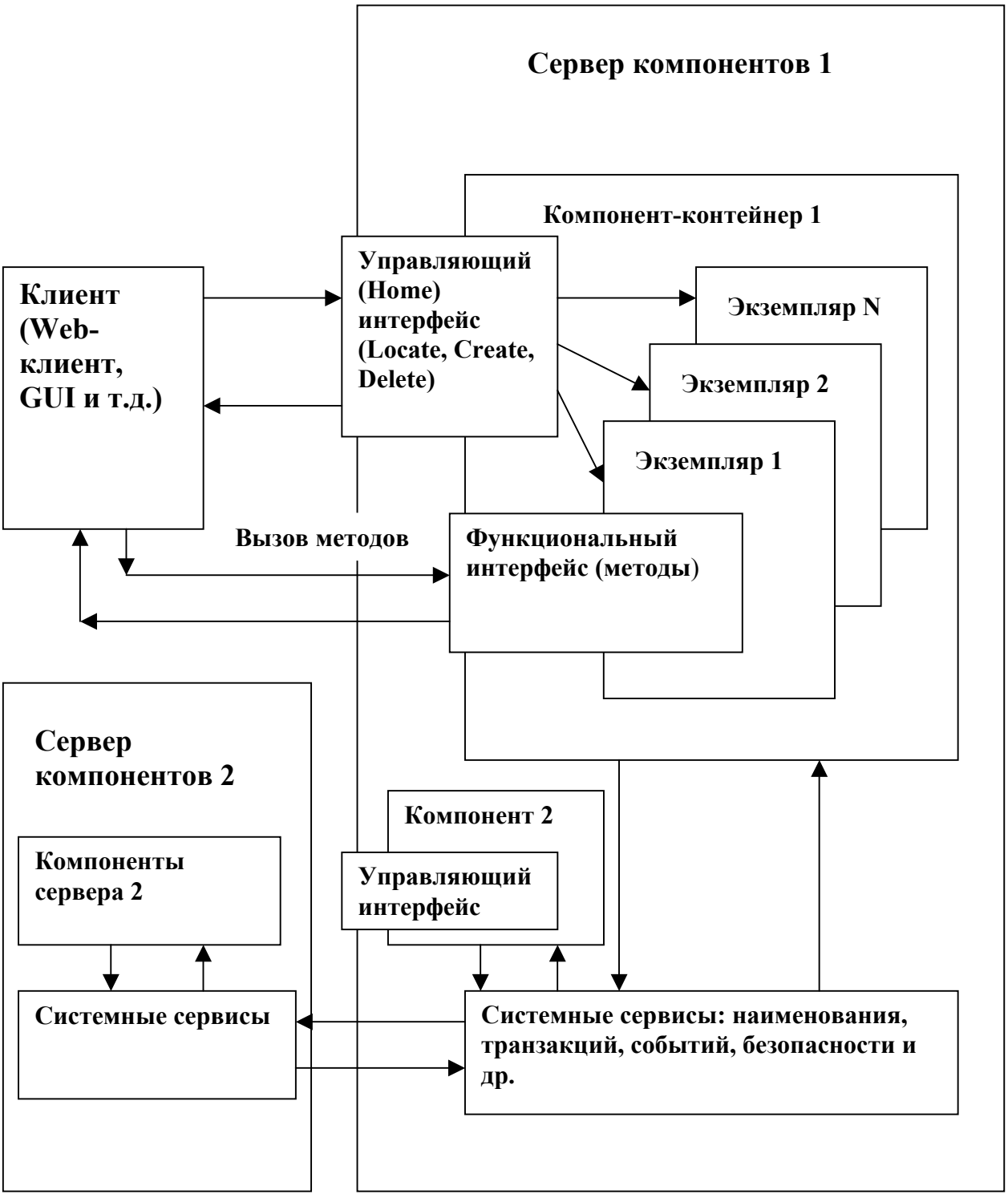

Рис. 2. Обобщенное представление компонентной архитектуры

основных источников идей и подходов для построения теории компонентного программирования. Она была разработана как расширение классической модели "клиент–сервер" с учетом специфики построения и функционирования программных компонентов, а также результатов практических реализаций и их апробирования. По отдельным вопросам и концепциям архитектуры, построения и функционирования компонентной среды имеется достаточно обширная литература, например [14–17].

В данной статье рассматривается представление обобщенной архитектуры как некоторой абстракции существующих моделей, сред, распределенных систем и их реализаций (рис. 2). Основой компонентной среды служит

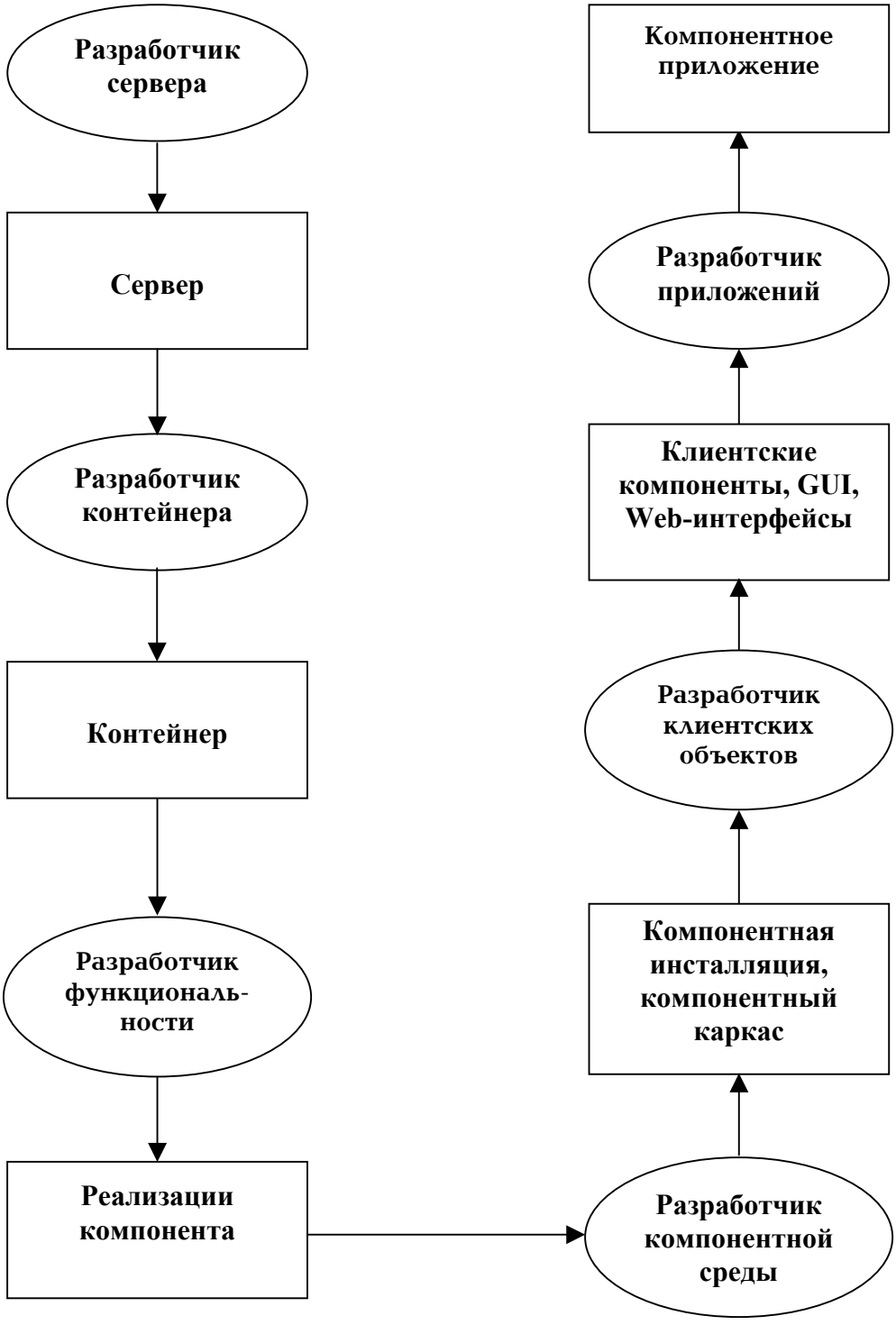

Рис. 3. Роли и объекты в компонентном программировании

множество серверов компонентов (часто их называют серверы приложений — application servers). Внутри сервера разворачиваются (устанавливаются) компоненты, представленные как контейнеры. Для каждого сервера может существовать произвольное количество контейнеров.

Контейнер представляет собой оболочку, внутри которой реализуется функциональность компонента. Взаимосвязь и взаимодействие контейнера с сервером строго регламентированы и осуществляются через стандартизованные интерфейсы. Контейнер управляет порождаемыми им экземплярами компонента, которые являются реализациями соответствующей функциональности. В общем случае внутри него может существовать произвольное число экземпляров-реализаций, каждая из которых имеет уникальный идентификатор.

С каждым контейнером связаны два типа интерфейсов для взаимодействия с другими компонентами и интерфейс системных сервисов, необходимых для функционирования самого контейнера и реализации специальных функций, например поддержка распределенных транзакций, в которых участвуют несколько компонентов. Первый тип интерфейса (в некоторых архитектурах называется Home интерфейс) обеспечивает управление экземплярами компонента с обязательными реализациями методов поиска, создания и удаления отдельных экземпляров.

Ко второму типу относятся интерфейсы, обеспечивающие доступ к реализации функциональности компонента. Фактически с каждым экземпляром связан свой функциональный интерфейс.

Экземпляры внутри контейнера могут взаимодействовать друг с другом, связываться с помощью системных сервисов с экземплярами, расположенными в других компонентах. Сами компоненты могут размещаться как внутри одного сервера, так и в разных серверах для различных платформ. Такое взаимодействие обеспечивает уникальная идентификация компонентов и экземпляров, а также регламентированные методы взаимодействия посредством интерфейсов и системных функций.

Роли и жизненные циклы в компонентном программировании

Анализируя архитектуру компонентной среды, можно отметить, что она состоит из различных типов программных объектов. В их число входят:

— серверы компонентов;

— контейнеры компонентов;

— реализации функциональности, представленные как экземпляры внутри контейнеров-компонентов;

— компонентные среды, определяемые компьютерными платформами, реализациями компонентных моделей, объектами, обеспечивающими установку и конфигурирование отдельных компонентов;

— клиентские компоненты, обеспечивающие интерфейсы конечного пользователя, реализуемые в виде различных типов клиентов (Web-клиенты, полноценные реализации графического интерфейса и т.д.);

— законченное компонентное приложение, представленное как компонентная программа.

Каждый тип объектов может реализоваться отдельно, так как для него существуют свои спецификации и требования, а также правила взаимодействия с другими объектами компонентного программирования. Все типы объектов образуют цепочку, определяющую порядок реализации компонентного приложения. Каждый тип объектов может реализоваться отдельным разработчиком и в соответствии с этим определяется его роль в процессе создания компонентной программы. На рис. 3 представлена такая цепочка объектов компонентного программирования с условным обозначением соответствующей роли.

Согласно разделению объектов компонентного программирования на типы и учитывая определенное место

каждого из них в процессе создания компонентного приложения, следует сделать вывод, что жизненный цикл компонентной программы значительно сложнее, чем жизненные циклы при других подходах программирования. Фактически речь идет о нескольких отдельных жизненных циклах для каждого типа объектов. Для обобщения этих данных все циклы с их краткими характеристиками сведены в таблицу. Более полное и подробное рассмотрение этих жизненных циклов выходит за рамки настоящей статьи.

Таблица. **Модели жизненного цикла в компонентном программировании**

Роль	**Объект компонентного программирования**	**Базовые концепции и спецификации**	**Модель жизненного цикла объекта**
Разработчик сервера	Сервер компонентов	Спецификация сервера. Спецификация компонентной модели. Спецификация системных сервисов	Модель ЖЦ сервера
Разработчик контейнера	Контейнер	Спецификация контейнера. Спецификация компонентной модели. Спецификация системных сервисов	Модель ЖЦ контейнера
Разработчик функциональности компонента	Реализации компонента	Спецификации прикладных интерфейсов. Спецификация компонентной модели	Модель ЖЦ реализации компонента
Разработчик компонентной среды, специалист по инсталляции компонента	Инсталляция и параметризация компонента в компонентной среде	Спецификация размещения и настройки компонента. Спецификация компонентной модели. Спецификация системных сервисов	Модель развертывания компонента. Модель построения каркаса
Разработчик клиентских объектов	Клиентские компоненты, GUI-объекты, Web-интерфейсы	Спецификации клиентских интерфейсов. Спецификация компонентной модели	Модель ЖЦ клиента
Разработчик приложения	Компонентное приложение	Спецификация приложения. Спецификация размещения и настройки компонентов	Модель ЖЦ компонентного приложения

Подход к определению формальной модели компонентного программирования

Как было отмечено выше, подход к построению формальных моделей компонентного программирования обязан учитывать множество аспектов, в частности, базироваться на современных концепциях компонентного программирования, обеспечивать простой переход от теории к практике с отображением и представлением сущест-

вующих архитектур и моделей (CORBA, COM, EJB), реализовать основу для создания формальных методов построения компонентных программ и систем. Такие задачи одновременно не могут быть разрешены в рамках одной определенной модели. Поэтому необходимо рассматривать семейство формальных моделей и других абстрактных структур, объединенных единой понятийной, терминологической и математической базой. Такой подход предлагается в настоящей статье, где в качестве основных абстрактных структур рассматриваются:

– модель компонента;

– модель компонентной среды;

– внешняя компонентная алгебра.

Выбор таких структур обусловлен следующими причинами. Во-первых, перечисленные модели относятся к базовым моделям компонентного программирования. Это означает, что любая компонентная теория должна включать такие или аналогичные им модели. Во-вторых, они достаточно типовые и емкие, чтобы наметить дальнейшие пути теоретических исследований. В-третьих, эти модели наглядно определяют связи теоретических и практических аспектов компонентного программирования, а также показывают взаимосвязи с другими подходами. Рассмотренные замечания о требовании практической применимости моделей отражены в их отдельных элементах и их связях.

Определение модели компонента

Модель компонента в общем случае представляется следующим образом:

$$Comp = (CName, CInt, CFact, CImp, CServ),$$

где CName — уникальное имя компонента;

$CInt = \{CInt^i\}$ — множество интерфейсов, связанных с компонентом;

CFact — интерфейс управления экземплярами компонента;

$CImp = \{CImp^j\}$ — множество реализаций компонента;

$CServ = \{CServ^r\}$ — интерфейс, определяющий множество системных сервисов, необходимых для поддержки функционирования компонента.

Имя компонента должно быть уникальным в любом пространстве имен для компонентной среды. Практически это обеспечивается специальными алгоритмами построения пространств имен в виде иерархического дерева, где каждая вершина однозначно определяется маршрутом (последовательностью вершин) от корня дерева.

Множество $CInt = CIntI \cup CIntO$ состоит из интерфейсов двух типов. К первому (множество CIntI) относятся интерфейсы, которые реализуются в среде данного компонента, т.е. имеют соответствующие реализации методов. Ко второму (множество CIntO) относятся интерфейсы, реализуемые в других компонентах, но функциональность которых требуется для выполнения методов данного компонента.

Каждый интерфейс компонента представлен следующим образом:

$$CInt^i = (IntName^i, IntFunc^i, IntSpec^i),$$

где $IntName^i$ — имя интерфейса;

$IntFunc^i$ — функциональность, реализуемая данным интерфейсом (совокупность методов);

$IntSpec^i$ — спецификация интерфейса (описания типов, констант, других элементов данных, сигнатур методов и т.д.).

$Provide(CInt^i)$ обозначается реализация методов, определяемых интерфейсом $CInt^i$ и поставляемая некоторой компонентной реализацией.

Интерфейс CFact определяет необходимые методы для управления экземплярами компонента. К ним относятся:

– поиск и определение нахождения требуемого экземпляра компонента Locate;

– создание экземпляра компонента Create;

– удаление экземпляра компонента Remove.

Таким образом, CFact = {Locate, Create, Remove}.

Каждая реализация компонента описывается следующим образом:

$$CImp^j = (\ ImpName^j\ ,\ ImpFunc^j\ ,\ ImpSpec^j\ ),$$

где $ImpName^j$ — идентификатор или имя реализации компонента;

$ImpFunc^j$ — функциональность, выполняемая данной реализацией (совокупность реализаций методов);

$ImpSpec^j$ — спецификация реализации (описание условий выполнения, описание параметров настройки реализации и т.д.).

Необходимым требованием правильного представления компонента есть условие

$$(\forall CInt^i \in CIntI)\ (\exists CImp^j \in \in CImp)\ Provide(CInt^i) \subseteq CImp^j.$$

Наличие знака включения в данной формуле означает, что выбранная реализация может обеспечить поддержку не только требуемого интерфейса, но и других. Практические технологии и языки программирования (CORBA, Java, C++ и др.) содержат для этого необходимые средства. Отметим также, что для каждого из таких интерфейсов может существовать несколько реализаций, различающихся особенностями функционирования (например, операционной средой, средствами хранения данных и т.д.).

Определение связи компонентной модели с ОО-моделями

Связь с объектно-ориентированными моделями прослеживается на основе следующего построения.

В процессе своего функционирования компонент посредством метода Create из интерфейса CFact порождает экземпляры

$$CFact.Create:\ Comp \rightarrow \{CIns_k^{ij}\},$$

$$CIns_k^{ij} = (IIns_k^{ij},\ IntFunc^i,\ ImpFunc^j),$$

где $CIns_k^{ij}$ — экземпляр k компонента, предоставляющий свою функциональность посредством интерфейса $IntFunc^i$ и обеспечивающий реализацию этого интерфейса посредством реализации $ImpFunc^j$;

$IIns_k^{ij}$ — уникальный идентификатор экземпляра компонента.

Пусть имеется некоторая объектная система, представленная в виде диаграммы классов

$$OSyst = (OClass,\ G),$$

где $OClass = \{OClass^i\}$ — множество классов;

G — объектный граф, отражающий связи и отношения между классами и экземплярами.

Каждый класс представим в виде

$$OClass^i = (ClassName^i,\ Method^i,\ Field^i\},$$

где $ClassName^i$ — имя класса;

$Method^i = \{Method_j^i\}$ — множество методов;

$Field^i = \{Field_n^i\}$ — множество переменных, определяющих состояние экземпляров класса.

Пусть $PField^i \subset Field^i$ — множество внешних переменных (public), которые доступны извне. Каждому $PField_n^i \in \in PField^i$ поставим в соответствие методы $get<PField_n^i>$ и $set<PField_n^i>$ для присвоения и выборки значений соответствующей переменной, т.е. эти переменные становятся атрибутами в терминах современных компонентных моделей. Соответственно в других классах вместо непосредственного обращения к таким переменным будут использоваться указанные методы. Введем новое множество методов

$$IMethod^i = Method^i \cup \{get<PField_n^i>\} \cup \cup \{set<PField_n^i>\},$$

которому сопоставим интерфейс $IFunc^i$, состоящий из прототипов методов, входящих в $IMethod^i$. Параллельно с OSyst рассмотрим систему

$$ISyst = (IFunc,\ IG),$$

где $IFunc = \{IFunc^i\}$ — множество интерфейсов;

IG — интерфейсный граф, идентичный графу G.

Класс $OClass^i$ порождает свои экземпляры (объекты)

$$Obj_k^i = \{ObjName_k^i, Method^i, Field^i\},$$

которым в системе ISyst будут соответствовать интерфейсные элементы

$$IObj_k^i = \{IName_k^i, IFunc^i\}.$$

Для каждого такого элемента не определена реализация соответствующего интерфейса. Сопоставив некоторому интерфейсу реализацию $ImpFunc^j$ (которая обеспечивает выполнение методов интерфейса), формируем элемент

$$IObj_k^{ij} = \{IName_k^i, IFunc^i, ImpFunc^j\},$$

который по своей сути эквивалентен экземпляру компонента

$$CIns_k^{ij} = (IIns_k^{ij}, IntFunc^i, ImpFunc^j).$$

Основные различия определяются следующими факторами. Во-первых, выбор подходящей реализации может происходить на этапе развертывания, а не на более ранних этапах, как это требуется для объектно-ориентированного подхода. Во-вторых, экземпляр объектного класса порождается на основе его описания и не может содержать элементов больше, чем есть в самом классе или в его суперклассах. В противоположность этому, реализация компонента может поддерживать несколько не связанных между собой интерфейсов. В остальном обе системы (объектная и интерфейсная) эквивалентны.

Таким образом, для построения компонентной программы могут применяться объектно-ориентированный подход и соответствующие инструментальные средства. Для этого с использованием ОО-методов проектирования параллельно строятся объектная система и соответствующая ей интерфейсная система без конкретизации реализации этих интерфейсов. Результат такого ОО-проектирования представляется как совокупность интерфейсов, для которой рассматривается задача покрытия интерфейсов соответствующими компонентными реализациями. На данном этапе жизненного цикла нет необходимости учитывать реализацию функциональности создаваемой программы. Эти реализации предоставляются компонентами на этапах интеграции и развертывания компонентного приложения.

Определение основных типов отношений между компонентами

Пусть выражение вида

$$Comp_n = (CName_n, CInt_n, CFact_n, CImp_n, CServ)$$

определяет конкретные компоненты. Рассмотрим основные типы компонентных отношений.

Отношение наследования. *Одиночное наследование*. Компонент $Comp_1$ наследует компонент $Comp_2$, если $\forall CInt_1^i \in \in Comp_1.CIntI$ $\exists Cint_2^k \in Comp_2.CintI$ такие, что интерфейс $CInt_1^i$ наследует интерфейс $CInt_2^k$ (в смысле ОО-наследования интерфейсов).

Множественное наследование. Компонент $Comp_1$ наследует компоненты $Comp_2$, $Comp_3$, ..., $Comp_N$, если $\forall CInt_1^i \in \in Comp_1.CIntI$ $(\exists n \in \{2, \ldots, N\})$ & $(\exists CInt_n^k \in \in Comp_n.CintI)$ такие, что интерфейс $CInt_1^i$ наследует интерфейс $CInt_n^k$.

Отношение наследования обладает свойствами рефлексивности, антисимметричности и транзитивности.

Отношение экземпляризации. Экземпляр $CIns_k^{ij} = (IIns_k^{ij}, IntFunc^i, ImpFunc^j)$ есть экземпляр компонента Comp, если $\exists CIntI^m \in CInt$ и $\exists CImp^k \in CImp$ такие, что

$$CIntI^m.IntFunc^m = IntFunc^i,$$
$$CImp^k.ImpFunc^k = ImpFunc^j,$$
$$Provide(CIntI^m) \subseteq CImp^k.$$

Отношение контракта. Взаимодействие между двумя компонентами описывается на основе модели контрактов. Если для функционирования одного компонента требуется функциональность, реализуемая в другом, то первый заключает контракт со вторым на предоставление требуемых услуг. Связь обеспечивается через соответствующие интерфейсы.

Пусть $Comp_1$ и $Comp_2$ — два компонента и $CIntO_1^i \in CInt_1$ — интерфейс, определяющий требуемую внешнюю

функциональность. Будем говорить, что между $Comp_1$ и $Comp_2$ возможно заключение контракта, если для $CIntO_1^i$

$$(\exists CIntI_2^m \in CInt_2) \ \& \ (\exists IMap_{12}^{im}: CIntO_1^i.IntFunc_1^i \rightarrow CIntI_2^m.IntFunc_2^m) \ \& \ \& \ IMap_{12}^{im}(CIntO_1^i.IntFunc_1^i) \subseteq \subseteq CIntI_2^m.IntFunc_2^m.$$

Отображение $IMap_{12}^{im}$ реализует соответствие между методами, описанными в обоих интерфейсах, на уровне их сигнатур и типов данных для передаваемых параметров. Для интерфейса $CIntI_2^m$ на основе общих свойств компонента всегда будет существовать реализация в $Comp_2$. Этим определяется контракт между двумя компонентами, который записывается в виде

$$Cont_{12}^{im} = (CIntO_1^i, CIntI_2^m, IMap_{12}^{im}).$$

В общем случае компонент может заключать контракт сам с собой, если оба интерфейса, входящие в определение контракта, принадлежат ему.

Отношение связывания. Отношение связывания определяется для двух экземпляров компонентов. Пусть $Comp_1$ и $Comp_2$ — два компонента и для них определен контракт $Cont_{12}^{im}$. Для компонента $Comp_1$ порожден экземпляр $CIns_{1k}^{ij}$, а для компонента $Comp_2$ — $CIns_{2p}^{mq}$. Будем говорить, что в этом случае для экземпляров $CIns_{1k}^{ij}$ и $CIns_{2p}^{mq}$ существует отношение связывания в соответствии с контрактом $Cont_{12}^{im}$, а сама связь будет обозначаться следующим образом:

$$Bind\ (IIns_{1k}^{ij}, IIns_{2p}^{mq}) = = (IIns_{1k}^{ij}, IIns_{2p}^{mq}, Cont_{12}^{im}).$$

В общем случае отношение связывания может существовать для двух экземпляров одного и того же компонента.

Определение модели компонентной среды

Определим компонентную среду как

$$CE = (NameSpace, IntRep, ImpRep, CServ, CServImp),$$

где $NameSpace = \{CName^m\}$ — пространство имен, представляющее множество имен компонентов среды;

$IntRep = \{IntRep^i\}$ — репозитарий интерфейсов, содержащий интерфейсы компонентов среды;

$ImpRep = \{ImpRep^j\}$ — репозитарий реализаций, содержащий реализации для компонентов среды;

$CServ = \{CServ^r\}$ — интерфейс, определяющий множество системных сервисов, необходимых для поддержки функционирования компонента;

$CServImp = \{CServImp^r\}$ — множество реализаций для системных сервисов.

Из определения этой модели следует, что она согласуется с обобщенной архитектурой компонентной среды, представленной на рис. 2.

Элемент репозитария интерфейсов определяется как пара

$$IntRep^i = (CInt^i, CName^m),$$

где $CInt^i$ — интерфейс для определенного компонента;

$CName^m$ — имя компонента, который реализует этот интерфейс.

Элемент репозитария реализаций определяется как пара

$$ImpRep = (CImp^j, CName^m),$$

где $CImp^j$ — реализация для определенного компонента;

$CName^m$ — имя компонента, который содержит эту реализацию.

Введем понятие нейтральной (нулевой) компонентной среды, которая будет называться каркасом (framework):

$$FW = (\varnothing, \varnothing, \varnothing, CServ, CServImp),$$

где пространство имен, репозитарии интерфейсов и реализаций являются пустыми множествами.

Элементы модели CServ и CServImp определяют конкретный тип компонентной среды. Совместимость компонентных сред различных типов определяется наличием отображения между соответствующими интерфейсами системных сервисов.

Пусть FW_1= (∅, ∅, ∅, $CServ_1$, $CServImp_1$), FW_2 = (∅, ∅, ∅, $CServ_2$, $CServImp_2$) — два каркаса разных типов. Отметим, что в соответствии с определением для каждого типа компонентной среды существует только один каркас. Если существует отображение SMap: $CServ_1$ -> $CServ_2$ такое, что $SMap(CServ_1) \subseteq CServ_2$, то каркас FW_1 совместим с каркасом FW_2. Для совместимости каркаса FW_2 с каркасом FW_1 необходимо существование обратного отображения с аналогичными свойствами. Ниже покажем (в соответствии с операциями внешней компонентной алгебры), что каждая компонентная среда определенного типа включает соответствующий каркас. Тогда совместимость каркасов будет определять и совместимость компонентных сред рассматриваемых типов.

Практически совместимость каркасов и компонентных сред означает следующее. Пусть для первой среды в состав $CServ_1$ входит некоторый сервис, например сервис наименования. Тогда и в состав $CServ_2$ тоже должен входить аналогичный сервис. И, кроме того, между этими сервисами должна существовать такая связь, что сервис первой среды должен понимать наименование объектов из второй и наоборот. Аналогичная ситуация должна быть и для остальных системных сервисов. Так как любая компонентная среда определенного типа использует одни и те же сервисы, то такие связи будут определять суть совместимости.

В частности, взаимодействие компонентов, расположенных в разных серверах компонентов, поддерживается сервисами этих серверов (этот факт отмечался при рассмотрении обобщенной архитектуры). Если совместимость между серверными сервисами существует, то такие компоненты могут входить в состав общего компонентного приложения.

Основы внешней компонентной алгебры

Алгебра называется внешней потому, что она определяет операции над компонентами и компонентными средами как над цельными объектами. Обозначим CSet = $\{Comp_n\}$ множество компонентов, а CESet = $\{CE_n\}$ множество компонентных сред. Все компоненты соответствуют условиям компонентной модели, а среды — условиям модели компонентной среды.

Определим операцию инсталляции (развертки) компонента в компонентной среде

$$CE_2 = Comp \oplus CE_1$$

со следующей семантикой:

$$CE_2.NameSpace = \{Comp.CName\} \cup \cup CE_1.NameSpace;$$

$$CE_2.IntRep = \{Comp.(CInt^i, CName)\} \cup \cup CE_1.IntRep;$$

$$CE_2.ImpRep = \{Comp.(CImp^j, CName)\} \cup \cup CE_1.ImpRep.$$

Определим операцию объединения компонентных сред

$$CE_3 = CE_1 \cup CE_2$$

с аналогичной семантикой:

$$CE_3.NameSpace = CE_1.NameSpace \cup \cup CE_2.NameSpace;$$

$$CE_3.IntRep = CE_1.IntRep \cup CE_2.IntRep;$$

$$CE_3.ImpRep = CE_1.ImpRep \cup CE_2.ImpRep.$$

Операция ⊕ имеет более высокий приоритет, чем операция ∪. Этот факт легко объясним, так как прежде, чем работать с компонентными средами, необходимо инсталлировать их компоненты. Отметим очевидные свойства операций:

$$\forall CE\ CE \cup FW = CE;$$

$$\forall CE_1, \forall CE_2\ CE_1 \cup CE_2 = CE_2 \cup CE_1;$$

$$\forall CE_1, \forall CE_2, \forall CE_3\ (CE_1 \cup CE_2) \cup CE_3 = = CE_1 \cup (CE_2 \cup CE_3);$$

$$\forall Comp, \forall CE_1, \forall CE_2\ (Comp \oplus CE_1) \cup \cup CE_2 = (Comp \oplus CE_2) \cup CE_1;$$

$$\forall Comp_1, \forall Comp_2, \forall CE\ Comp_1 \oplus (Comp_2 \oplus \oplus CE) = Comp_2 \oplus (Comp_1 \oplus CE).$$

Определим операцию удаления компонента из компонентной среды

$$CE_2 = CE_1 \setminus Comp$$

со следующей семантикой:

$$\exists\, CName^m \in NameSpace \;\&\; (CName^m = Comp.CName) \Rightarrow CE_2.NameSpace = CE_1.NameSpace \setminus \{Comp.CName\} \;\&\; CE_2.IntRep = CE_1.IntRep \setminus \{(\forall I \;\&\; IntRep^i.CName = Comp.CName)\, IntRep^i\} \;\&\; CE_2.ImpRep = CE_1.ImpRep \setminus \{(\forall j \;\&\; IntRep^j.CName = Comp.CName)\, IntRep^j\}.$$

Справедливо следующее равенство (доказывается на основе соответствующих операций над множествами, входящими в определении компонента и среды):

$$(Comp \oplus CE) \setminus Comp = CE.$$

При другом порядке скобок равенство не всегда выполняется. Это означает, что операция $\oplus$ имеем более высокий приоритет, чем операция \.

Операцию замещения компонента $Comp_1$ компонентом $Comp_2$ можно выразить через операции $\oplus$ и \:

$$CE_2 = Comp_2 \oplus (CE_1 \setminus Comp_1).$$

Пусть Ω обозначает множество операций:

$$\Omega = \{\oplus, \setminus, \cup\},$$

расположенных в порядке уменьшения приоритетов. Тогда

$$\Psi = \{CSet, CESet, \Omega\}$$

определяет внешнюю компонентную алгебру, включающую множества компонентов, компонентных сред и операции над их элементами.

Другие модели компонентного программирования

Среди других моделей, которые здесь подробно не рассматриваются, отметим такие.

Внутренняя компонентная алгебра. Как было отмечено, во внешней компонентной алгебре компоненты представляют цельные объекты. Но согласно модели компонента он имеет собственную структуру. Поэтому целесообразно рассмотреть операции над отдельными элементами такой структуры, которые (по аналогии с соответствующими операциями для других методов программирования, например объектно-ориентированного [18]) называются операциями рефакторинга. Суть этих операций — изменения в именах, множествах интерфейсов, множествах реализаций, соотношениях и связях между этими множествами. Особый интерес представляет множество таких операций рефакторинга, которые сохраняют целостность понятия компонента, т.е. условия определения и существования компонента в результате рефакторинга не меняются. Примером таких операций могут служить:

— добавление новой реализации для существующего интерфейса;

— добавление нового интерфейса и новой реализации для него (этот пример характерен для программирования в модели COM);

— объединение существующих интерфейсов, и, если необходимо, объединение и их реализаций.

Множество элементов модели компонента и множество указанных операций рефакторинга будут определять внутреннюю компонентную алгебру.

Модель системных сервисов компонентной среды. Выше была отмечена важность построения такой модели (в частности, для определения совместимости компонентных сред). Эта модель детализирует CServ с указанием сервисов, которые необходимы для поддержки функционирования компонентов и компонентных сред в рамках задач компонентного программирования, что обеспечивает формальное определение дополнительных статических и динамических свойств компонентных сред.

Обобщенная компонентная алгебра. Эта алгебра на единой формальной платформе объединяет внешнюю и внутреннюю компонентные алгебры, а также модель системных сервисов.

Помимо непосредственного объединения возможностей указанных алгебр, она позволит формализовать дополнительные задачи:

— для выбранного множества компонентов определить последовательность операций рефакторинга, которые обеспечат построение компонентного приложения (или доказать, что для выбранного множества эта задача неразрешима);

— указать необходимые условия, чтобы выбранная компонентная конфигурация обладала свойствами транзакционности (или доказать отсутствие такого свойства);

— обеспечить реализацию комбинированных операций, например переименования компонентов и/или их интерфейсов (которые по сути являются рефакторингами), с образованием нового компонента, дополняющего существующую компонентную среду.

Формулировка основной задачи компонентного программирования

Как отмечалось в начале статьи, основная задача компонентного программирования состоит в построении компонентных программ и систем на основе компонентных моделей, методов и средств. Переопределим и сформулируем эту задачу в терминах, понятиях и моделях, рассмотренных выше. Выполним последовательно следующие шаги.

Определение базовой компонентной среды. Пусть CSet — множество имеющихся компонентов и $\Psi = \{ CSet, CESet, \Omega\}$ — внешняя компонентная алгебра. Обозначим $CE^c \in CESet$ компонентную среду, в которой развернуты все компоненты из CSet:

$$CE^c = CE_1 \oplus CE_2 \oplus \ldots \oplus FW.$$

В дальнейшем компонентные среды типа CE^c будем называть базовыми для соответствующего множества CSet.

Определение расширения базовой компонентной среды. Рассмотрим множество компонент CSet. В результате применения операций рефакторинга из внутренней компонентной алгебры и операций из обобщенной алгебры строим множество CSet*, которое объединяет CSet и все допустимые результаты применения этих операций над всеми элементами CSet. Будем говорить, что CSet* является замыканием CSet относительно множеств операций из указанных алгебр.

Для множества $CSet^*$ строим новую базовую компонентную среду CE^{*c}, которая является расширением предыдущей. Подобный процесс расширения может быть неоднократно повторен с получением новых компонентов и базовых компонентных сред.

Компонентное представление программы. Под компонентным представлением программы понимается такое абстрактное определение конструируемой программы, в котором его элементами являются представления, которым можно поставить в соответствие реальные компоненты, обеспечивающие реализацию функциональности задачи и выполнение других нефункциональных требований. Для одной и той же программы существует множество ее компонентных представлений в зависимости от концепций проектирования и используемого компонентного подхода. Рассмотрим обобщенный уровень такой модели.

Основная задача проектирования компонентной программы состоит в представлении ее функциональности и других требований как совокупности контрактов между отдельными объектами — программными компонентами, из которых она будет состоять. Пусть $\{A_i\}$ — множество контрактов для программы, определяющих ее функциональность. Каждому A_i можно поставить в соответствие интерфейс I_i, который описывает контракт как клиент—серверное взаимодействие с соответствующими методами и структурами данных. Согласно этому каждый интерфейс можно сопоставить с парой In_i и Out_i, которые будем называть соответственно реализующим представлением для I_i и определяющим пред-

ставлением для I_i. Здесь используются эти термины вместо понятий входной и выходной интерфейс потому, что реально они не являются такими интерфейсами (для них могут не существовать соответствующие интерфейсы в существующих компонентах). В результате перепроектирования они могут быть переопределены и изменены с целью адаптации их к реальным интерфейсам. Out_i определяет условия и цель контракта со стороны клиента, а In_i задает аспект реализации контракта со стороны сервера.

После того как все In_i и Out_i определены, их можно группировать в различных сочетаниях. Рассмотрим произвольную совокупность $M_v = \{In_i\} \cup \cup \{Out_j\}$. Назовем каждое M_v шаблоном компонента. Фактически каждый шаблон содержит некоторое множество определяющих и реализующих представлений интерфейсов. По этим представлениям в дальнейшем осуществляется сопоставление шаблонов и реальных интерфейсов существующих компонентов.

Если результаты сопоставления для всех M_v успешны, то в итоге подобных операций группирования получается множество $CP = \{M_v\}$, которое и будет называться компонентным представлением программы. Для одной и той же программы существует множество представлений, которые определяются множествами контрактов (или интерфейсов) и способами построения шаблонов.

Определение основной задачи компонентного программирования. Эта задача формулируется таким образом. Дано множество компонентов CSet и указанные компонентные алгебры. Цель — построить компонентную программу с заданными функциональными и нефункциональными требованиями на искомом множестве компонентов.

Задача состоит в том, чтобы представить результаты проектирования программы в виде компонентного представления $CP = \{M_v\}$ таким образом, чтобы для любого M_v существовал компонент из CSet или он мог быть получен из этого множества в результате конечного числа допустимых операций из соответствующих компонентных алгебр.

На практике такие условия могут выполняться не всегда (например, разрабатывается программа с принципиально новой функциональностью). В этом случае реализуются дополнительные компоненты и расширяется множество CSet.

Выводы

В настоящей статье описан подход к построению теории компонентного программирования. Последовательно рассмотрены концепции построения такой теории, определения основных терминов, существующие теоретические и практические результаты, влияющие на построение теории, а также описаны основные модели компонентного программирования. На базе рассмотренных моделей сформулирована основная задача компонентного программирования.

Отличительной особенностью предлагаемого подхода является его целостность. Все концепции, аспекты, терминология, математические методы увязаны в единую схему со строгим определением причинно-следственных связей. Изложение показывает, что, несмотря на сложность и неоднозначность поставленной задачи, подход к построению единой теории возможен, а построенная теория может получить практическое применение.

Дополнительной особенностью подхода является определение новых направлений исследования. Это относится как к построению новых моделей и методов, так и к практическим аспектам применения полученных теоретических результатов.

Построение теории компонентного программирования создаст необходимый базис для формирования практических методик и методологий создания компонентных программ, разработки новых инструментальных средств и улучшения качества разрабатываемых продуктов.

1. *Грищенко В.Н., Лаврищева Е.М.* Методы и средства компонентного программирования // Кибернетика и системный анализ. — 2003. — № 1. — С. 39—55.
2. *Грищенко В.Н., Лаврищева Е.М.* Компонентно-орентированное программирование. Состояние, направления и перспективы развития // Проблемы программирования. — 2002. — N 1—2. — С.80—90.
3. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. — Киев: Наук. думка, 1991. — 213 с.
4. *Лаврищева Е.М.* Сборочное программирование. Некоторые итоги и перспективы // Проблемы программирования. — 1999. — № 2. — С. 20—31.
5. *Буч Г.* Объектно-ориентированный анализ и проектирование. — М.: Бином, 1998. — 560 с.
6. *Rada R., Moore J.* Standardizing Reuse // Communications of the ACM. — 1997. — **40**, N 3. — P. 19—23.
7. *Бабенко Л.П., Лаврищева К.М.* Основи програмної інженерії. — К.: Знання, 2001. — 269 с.
8. *Грищенко В.Н.* Особливості компонентно-орієнтованої розробки програмного забезпечення // Проблемы программирования. — 2001. — N 3—4. — С. 75—92.
9. *Грищенко В.Н.* Систематизований підхід до визначення програмних компонентів // Там же. — С. 23—30.
10. *Сигел Дж.* CORBA 3. — М.: Малип, 2002. — 412 с.
11. *Roman E., Ambler S., Jewell T.* Mastering Enterprise JavaBeans. — New York: Wiley Comp. Publ., 2002. — 670 с.
12. *Lowy J.* COM and .NET Component Services. — O'Reilly, 2001. — 384 p.
13. *Crnkovic I., Hnich B., Jonsson T., Kiziltan Z.* Specification, Implementation and Deployment of COMPONENTS // Comm. ACM. — 2002. — Oct. — P. 35—40.
14. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft/COM и Java/RMI. — М.: Мир, 2002. — 510 с.
15. *Цимбал А.А, Аншина М.Л.* Технологии создания распределенных систем. Для профессионалов. — СПб.: Питер, 2003. — 576 с.
16. *Андон Ф.И., Лаврищева Е.М.* Методы инженерии распределенных компьютерных систем. — К.: Наук. думка, 1997. — 228 с.
17. *Грищенко В.М.* Парадигма перетворення даних, що передаються в мережі // Проблемы программирования. — 2001. — N 1—2. — С. 84—94.
18. *Фаулер М.* Рефакторинг: улучшение соответствующего кода. — СПб.: Символ-Плюс, 2003. — 432 с.

Получено 26.05.03

Об авторе

Грищенко Владимир Николаевич
канд. физ.-мат. наук, докторант

Место работы автора:
Институт программных систем НАН Украины,
просп. Академика Глушкова, 40,
Киев-187, 03680,
Украина

Тел. (044) 266 3470