

КОМПОНЕНТНАЯ АРХИТЕКТУРА С ОПРЕДЕЛЕНИЕМ ТИПОВ ВО ВРЕМЯ ИСПОЛНЕНИЯ

Е.М. Гринкруг,

*кандидат технических наук, доцент кафедры Управления разработкой программного обеспечения Национального исследовательского университета «Высшая школа экономики»,
e-mail: egrinkrug@hse.ru.*

А.Р. Шакуров,

*аспирант кафедры Управления разработкой программного обеспечения Национального исследовательского университета «Высшая школа экономики»,
e-mail: amir-shak@yandex.ru.
Адрес: г. Москва, ул. Кирпичная, д. 33/5.*

В работе рассматриваются вопросы компонентного подхода к разработке программного обеспечения. На основе анализа основных идей данного подхода, их реализации в существующих на сегодняшний день технологиях, присущих им ограничений и перспективных направлений их развития предлагается новая компонентная архитектура, расширяющая функциональные возможности существующих компонентных моделей. Описываются основные принципы построения данной архитектуры.

Ключевые слова: компонент, компонентная разработка, интерфейс, реализация.

1. Введение

Разработка программного обеспечения — трудоёмкий процесс, и неудивительно, что *повторному использованию кода* [8] придаётся большое значение. Многократное использование реализованной функциональности снижает трудозатраты при разработке больших программных систем, сокращает количество ошибок в коде и упрощает его сопровождение.

Наиболее распространёнными примерами повторного использования кода являются программные библиотеки, паттерны проектирования [5] и каркасы приложений. Объектно-ориентированное и обобщённое программирование [4] также содержат в своей основе данную идею. Однако наиболее многообещающей её формой нам представляется *компонентный подход* к созданию программного обеспечения [9].

Под программным компонентом в общем случае понимают некоторую сущность, содержащую данные и реализующую функциональность, скрытые за чётко определёнными интерфейсами (ср. [10] и [11]). Взаимодействие с подобным «чёрным ящиком» осуществляется исключительно через интерфейс. Более точное определение компонента приведено в разделе 4.

Само понятие «интерфейса» трактуется по-разному в различных технологиях. Это может быть совокупность «свойств» («атрибутов», «членов» и т.п.) [13] или самостоятельная неделимая сущность [2]. Возможны и другие варианты. Ниже мы предложим оптимальный, на наш взгляд, подход и приведём ряд аргументов в его пользу.

В рамках компонентной идеологии соединение компонентов в работоспособную систему полагается сравнительно простой операцией (см., напр., [3]). Но манипулирование структурой компонентной системы осуществляется в разных технологиях различными способами. Наша задача — найти оптимальный вариант организации взаимодействия компонентов, представляющий собой сбалансированное между гибкостью и простотой использования решение.

2. Ограничения существующих компонентных моделей

Несмотря на очевидные преимущества компонентного подхода, его реализации в тех или иных технологиях на сегодняшний день имеют ряд существенных ограничений.

Основная сложность, по-видимому, заключается в том, что сторонний код всегда либо недостаточно универсален, либо содержит избыточную функциональность. Последнее характерно для программных библиотек, т.к. универсальность — критически важное для них свойство. Но избыточность приводит к неоправданному «раздуванию» итоговой системы, усложнению процесса разработки и т.п.

Компонент может быть сделан более специализированным, т.к. он структурно меньше, чем целая библиотека. Но и компромисс между универсальностью и специализированностью здесь более важен: если иногда есть возможность использовать лишь часть библиотеки, то часть компонента использовать не удастся, не нарушив инкапсуляции.

Одним из способов достижения указанного компромисса является разделение работы с компонентом на проектирование и разработку (designtime) и

использование конечным приложением (runtime). Необходимость такого разделения признана и до некоторой степени реализована (см., напр., [13]), однако, на наш взгляд, в этой области возможно достичь большего.

Приведём пример. Пусть, проектируя приложение с графическим интерфейсом, мы должны изменить надпись на компоненте «кнопка». Изменение окончательно: после запуска программы надпись меняться не будет. Реализации данного процесса (запись в соответствующее свойство необходимой строки) в различных технологиях обладают общим недостатком: переменное свойство остаётся изменяемым и после запуска приложения. Т.о., хотя заранее известно, что надпись на кнопке во время исполнения должна быть константой, реализовать это существующими средствами невозможно.

На данную проблему можно взглянуть иначе. Сложность глубокой подстройки компонента под контекст использования (разделение «designtime — runtime» — лишь частный пример) связана с тем, что фактически речь идёт об определении нового типа данных. И т.к. любые манипуляции с компонентом предполагают использование его функциональности и, следовательно, его запуска, определение нового типа должно осуществляться во время исполнения (и без перекомпиляции).

Существуют обходные пути: генерация кода (так поступает большая часть инструментов визуального проектирования GUI (напр., [1])) и вызов компилятора, непосредственная генерация байт-кода и т.п. Однако компилятор доступен далеко не во всякой среде исполнения; данное ограничение, в частности, свойственно встраиваемым системам. По этой причине представляются интересными системы, не прибегающие к помощи компиляторов и т.п. обходных путей и способные создавать из компонентов, настроенных по усмотрению пользователя, новые типы, оставаясь при этом в рамках собственной компонентной модели.

3. Структуризация данных и управление потоком исполнения

Рассмотрим основные аспекты распространённых сегодня языков программирования и компонентных технологий с целью выявить общие идеи, лежащие в их основе, и найти наиболее подходящие реализации каждой из них. Анализ определит основные черты предлагаемой модели, описание которой приведено в следующем разделе.

Детальное рассмотрение конкретных языков и технологий и их отличительных особенностей выходит за рамки данной работы (подобный обзор можно найти в [12]). Мы отталкиваемся от анализа объектно-ориентированной концепции в целом, приводя частные примеры и привлекая технологии, выходящие за рамки объектно-ориентированной парадигмы, когда это необходимо, т.к. требование возможности определения новых типов во время исполнения приводит к необходимости построения компонентной технологии, где время разработки и время исполнения не имеют строгого разграничения.

Всякая среда разработки/исполнения может быть рассмотрена с точки зрения предлагаемых ею принципов и механизмов 1) организации данных и 2) управления потоком исполнения. С точки зрения первого аспекта для объектно-ориентированной программы отметим возможность иерархической организации данных. Это проверенное средство борьбы с нарастающей сложностью программных систем, поэтому предлагаемая компонентная модель предполагает возможность объединения компонентов во взаимодействующие группы, представляющие собой новые компоненты, и т.д. (строго говоря, в группы объединяются *типы компонентов*, после чего создаются сами составные компоненты, см. раздел 4).

В части управления потоком исполнения, как нам кажется, объектно-ориентированный подход не столь удачен. Взаимодействие объектов осуществляется посредством методов, но концепция метода чересчур гибка, что приводит к усложнению структуры программ, использующих её. Метод может возвращать значение, а может не возвращать его. Возвращаемое значение может быть ссылкой на внутреннее поле объекта, но может являться и его защитной копией. Он может иметь произвольное количество параметров произвольных типов, а может не иметь их вовсе. Список можно продолжить. Понятие метода не предоставляет возможности достаточной формализации взаимодействия объектов.

Концепция свойства, присутствующая в некоторых технологиях (C#, JavaBeans), более удачна. Она сочетает в себе как признаки поля объекта (типизированность, строго заданный набор операций: чтение, запись), так и признаки метода (за операциями может скрываться определённая программистом функциональность). Модель взаимодействия компонентов, основанная на свойствах, проста и фор-

мальна в силу ограниченного количества характеристик понятия «свойство» и операций над ним. Однако свойства, как они реализованы, напр., в C#, не могут конкурировать с методами в части предоставляемых разработчику возможностей. Так, если свойство предоставляет лишь доступ для чтения/записи, реализация широко известного механизма обратных вызовов затруднительна. Т.о., мы приходим к необходимости добавления операции *связывания*.

Итак, мы указали на два ключевых принципа — иерархическую организацию компонентов и их взаимодействие, основанное на свойствах, — следование которым, на наш взгляд, необходимо для создания компонентной технологии, удобной в практическом применении. Наряду с требованием возможности определения новых типов компонентов во время исполнения, эти принципы легли в основу предлагаемого решения, к описанию которого мы сейчас переходим.

4. Предлагаемое решение

Охарактеризуем компонентную модель, реализация которой позволит сохранить преимущества существующих технологий и моделей, устранив перечисленные выше ограничения.

К числу основных категорий, которыми оперирует модель, относятся: компонент, тип компонента и редактор типа компонента. Формальные определения приведены ниже, но, забегая вперёд, отметим, что под компонентом понимается сущность, представляющая собой основной «строительный блок» программной системы и инкапсулирующая данные и поведение. Тип компонента — это сущность, описывающая тип данных, хранимых в компонентах данного типа, и содержащая другую метайнформацию. Редактор типа — это средство определения новых типов компонентов на основе существующих.

4.1. Компонент

Идея компонентной разработки программного обеспечения в первую очередь предполагает разграничение «интерфейс — реализация», что позволяет осуществлять разработку и использование компонента независимо.

Под компонентом будем понимать «чёрный ящик», функциональность и данные, скрытые за интерфейсом. Компонент, т.о., характеризуется

интерфейсом и реализацией, а также состоянием, т.е. данными, содержащимися в нём и изменяющимися в течение жизни.

Интерфейс компонента определяется набором его свойств — именованных атрибутов, для которых определён тип значения и права доступа.

Реализация определяет поведение компонента, его функциональность и отличается у *примитивных, составных* и *скомпилированных* компонентов.

Интерфейс

Элементарной ячейкой интерфейсной части компонента является его *свойство*. *Свойством* мы называем имеющую строго определённый тип именованную сущность, представляющую некоторый аспект или атрибут компонента, к которой, возможно, применимы некоторые из трёх операций (см. рис. 1):

- ◆ чтение — получение текущего значения данного свойства;
- ◆ запись — установка текущего значения;
- ◆ связывание — операция подключения «слушателя» события изменения свойства.

Применимость данных операций определяется правами доступа (устанавливаемыми типом компонента, см. раздел 4.2).

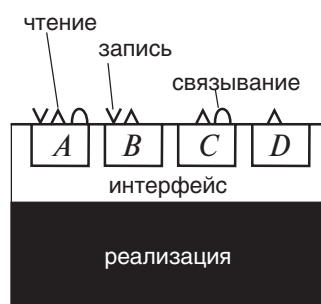


Рис. 1. Интерфейс компонента, состоящий из свойств A, B, C и D с различными режимами доступа.

Значение свойства, если оно доступно для чтения, определяется внутренним состоянием компонента, на которое, в свою очередь, влияют операции записи свойств.

Операция связывания свойства A со свойством B (A и B могут принадлежать различным компонентам) позволяет компоненту-владельцу свойства B получать уведомления о каждом изменении значения свойства A в виде записи нового значения в свойство B.

Реализация

Придерживаясь идеи организации компонентов в иерархические структуры, мы различаем три вида компонентов: примитивные, скомпилированные и составные.

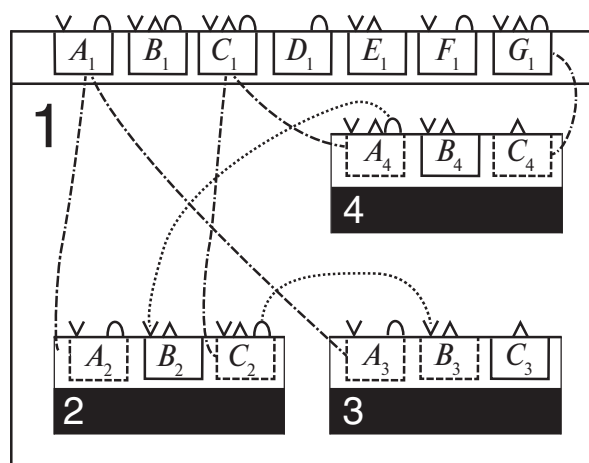


Рис. 2. Реализация: компонент содержит в себе три подкомпонента, связанных друг с другом и со свойствами надкомпонента.

Пунктирные линии показывают установленные событийные связи, штрихпунктирные соединяют между собой разделяемые свойства.

Примитивные компоненты аналогичны примитивным типам в языках программирования. Они инкапсулируют данные наиболее распространённого вида: целые и дробные числа, логические значения, символы, строки символов и ряд других. С точки зрения модели, примитивные компоненты неделимы, т.е. не состоят из других компонентов, и не обладают свойствами.

Примитивные компоненты — типичный пример «объектов-значений», значение которых задаётся при создании и не может быть изменено. Важнейшей характеристикой примитивного компонента является индивидуальность его значения.

Составные компоненты представляют собой (с точки зрения реализации) наборы других компонентов, сочленённых посредством двух механизмов: *событий* и *разделяемых свойств*. В подобных случаях мы будем говорить об объёмлющем компоненте (надкомпоненте) и вложенных компонентах (подкомпонентах).

Событийная связь предполагает, что над свойствами двух подкомпонентов была произведена описанная выше операция связывания некоторых из их свойств.

Связь посредством разделения подразумевает следующее. Компонент, создаваемый «внутри» другого компонента, в процессе создания получает ссылку на свойство своего надкомпонента. Это вынуждает его не создавать новое свойство (что делается при создании компонента «вовне» других компонентов), а использовать существующее свойство обьёмлющего компонента для последующих операций взаимодействия с внешним миром (чтение/запись/связывание). Реализация данного механизма описана в разделе 4.2. Одно свойство надкомпонента может быть разделено несколькими свойствами подкомпонентов, что позволяет достаточно гибко организовывать взаимодействие в системе.

Заметим, что способ «прикрепления» свойства компонента к его реализации (определения, каким образом операции над свойством взаимодействуют с реализацией) в принципе может иметь достаточно сложный характер. Одна крайность — это описание взаимодействия на некотором языке программирования, другая — это тривиальная реализация, когда свойство лишь предоставляет доступ к внутренней переменной компонента. Вышеописанный подход — компромисс, не перегруженный сложностью и не лишённый гибкости.

Подкомпоненты некоторого компонента могут сами являться составными компонентами. Подобная иерархия может иметь рекурсивную вложенность произвольной глубины (ограниченную лишь доступными аппаратными средствами), будучи на нижнем уровне представлена примитивными компонентами.

Скомпилированные компоненты — это компоненты, реализованные средствами, отличными от описанных выше средств реализации составных компонентов. Их поддержка необходима для интеграции со сторонними технологиями (такими как JavaBeans).

Анализ реализации скомпилированного компонента моделью не производится. Этим скомпилированные компоненты схожи с примитивными. Два основных различия: 1) последние не имеют свойств и являются (неизменяемыми) компонентами-значениями; 2) скомпилированные компоненты, как и составные, имеют состояние по умолчанию и могут быть сконструированы вне какого-либо внешнего контекста, тогда как создание примитивных компонентов без указания как минимум их значения лишено смысла.

Перейдём теперь к описанию типов компонентов и механизмов их создания.

4.2. Тип компонента

Тип компонента представляет собой именованную сущность, специфицирующую, каким образом должен создаваться компонент данного типа.¹ Т.о., понятие типа аналогично понятию класса в объектно-ориентированных языках.

Типы содержат в себе описание интерфейса, реализации и их взаимосвязи для компонентов данного типа. «Интерфейсная» часть типа содержит перечень *дескрипторов свойств*, каждый из которых задаёт имя (либо индекс или любой другой «адрес») свойства будущего компонента, его тип, права доступа и, возможно, значение по умолчанию.

«Реализационная» часть типа различна для примитивных, скомпилированных и составных типов (однозначно соответствующих трём рассмотренным категориям компонентов). Для примитивных типов она сводится к хранению текущего значения компонента. Для скомпилированных типов — это указание способа получения реализации создаваемого компонента и связывания его с интерфейсом. Так, для создания компонента, реализация которого основана на технологии JavaBeans, достаточно указания имени соответствующего java-класса — дальнейшие действия осуществляются с помощью механизма отражений Java [6].

Наибольший интерес представляет составной тип. В части реализации он включает в себя список *дескрипторов подкомпонентов*, каждый из которых задаёт тип соответствующего подкомпонента в будущем компоненте и, возможно, его начальное значение.

Эта информация дополняется данными о связях:

- ◆ для каждого свойства каждого подкомпонента указывается, какое свойство надкомпонента оно разделяет (если разделение имеет место);
- ◆ перечисляются событийные связи для тех свойств (подкомпонентов и надкомпонента), для которых это необходимо.

Прежде чем переходить к описанию процесса *инстанцирования* (создания экземпляра) типа, введём понятие *поля* компонента. Поля являются средством реализации механизма разделения свойств. Если компонент создаётся вне другого компонента,

¹ Всякий компонент в момент создания получает и сохраняет ссылку на собственный тип, тем самым однозначно ассоциируя себя с ним.

он создаёт «ячейку», соответствующую этому свойству, с которой взаимодействует его внутренняя реализация. Но если компонент в момент инициализации получает ссылку на некоторое свойство надкомпонента, он должен использовать её везде, где в обычном случае использовал бы упомянутую «ячейку». Т.о., эта «ячейка» должна иметь ссылочный тип, исполняя роль посредника во взаимодействии реализации компонента с его свойствами, а адрес ссылки должен определяться контекстом инициализации компонента. Подобную ячейку-ссылку мы и называем *полем*.²

Опишем теперь процесс конструирования компонента. Он подразумевает следующую последовательность шагов.

1. Создаются и инициализируются (в зависимости от контекста) поля компонента (по одному на каждый дескриптор свойства составного типа или описание свойства скопированного типа).

2. Создаётся реализация компонента. В случае составного типа создаются подкомпоненты (по одному на дескриптор подкомпонента); если свойство A_i некоторого подкомпонента i должно быть разделено со свойством A_0 надкомпонента, в момент инициализации компоненту i передаётся ссылка на A_0 . В случае примитивного типа в компоненте сохраняется его значение, заданное пользователем. В случае скомпилированного типа создание реализационной части компонента зависит от «нижележащей» технологии.

3. Интерфейс «прикрепляется» к реализации. Для составного компонента это означает установку необходимых событийных связей между его свойствами и свойствами подкомпонентов. Для скомпилированного компонента данный шаг, как и предыдущий, зависит от технологии, на основе которой он реализован.

После инстанцирования типа результирующий компонент обладает соответствующими свойствами и реализацией. Все ограничения, отслеживаемые компонентами в процессе функционирования, — контроль типов, доступа и др. — основываются на метаинформации, заложенной в их типах.

Заметим, что, стремясь минимизировать количество независимых категорий, которыми оперирует модель, мы считаем типы компонентами. Т.о., существует тип «тип», экземплярами которого являются все типы (в т.ч. и сам этот тип).

Перечисленное выше «содержимое» типа (перечни дескрипторов свойств, событийных связей, дескрипторов подкомпонентов) — это три свойства типа «тип».

Специфика понятия «тип» накладывает важное ограничение на манипулирование типами как компонентами: тип по своей сути является сущностью неизменяемой, т.к. содержит в себе метаинформацию о других сущностях. Необходимость определения новых типов во время исполнения, о которой шла речь выше, не требует, тем не менее, модификации существующих типов: достаточно иметь возможность создания нового типа, являющегося модифицированной версией прежнего.

4.3. Определение типов во время исполнения

Рассмотрим вопрос определения новых типов во время исполнения. Предлагаемая модель предоставляет возможность следующей логики действий.

1. Пользователь, выбрав тип компонента из библиотеки типов (включающей в себя предопределённые примитивные типы, а также составные типы, определённые ранее), создаёт *редактор типа* — специальную сущность, в которую заключена функциональность определения новых типов на основе существующих. Она принимает на вход существующий тип и позволяет совершать операции, перечисленные ниже.

2. Пользователь имеет возможность настроить полученную структуру.

◆ Он может изменить дескрипторы свойств: имя, значение по умолчанию, права доступа. Важно, что компонент «глубоко» подстраивается под контекст. Так, если пользователь указал, что свойство, прежде доступное для чтения, записи и связывания, будет доступно только для чтения, реализация свойства как переменной должна измениться на реализацию в виде константы (в смысле языка программирования).

◆ Пользователь может редактировать метаданные реализации: добавлять и удалять подкомпоненты, устанавливая и удаляя событийные связи между ними, добавлять и удалять отношения разделения свойств.

3. Когда необходимые модификации произведены, созданный составной тип можно добавить в

²Описанное понятие сходно с концепцией поля в языке VRML [7].

библиотеку типов и в дальнейшем использовать его наравне с другими типами.

Редактор типа, т.о., предоставляет доступ к внутренним данным типа и позволяет модифицировать их *копию*, с тем чтобы иметь возможность создавать видоизменённые варианты типов, экземпляры которых существуют в системе. Реализация редактора типа, благодаря такому подходу, становится достаточно простой; мы не станем останавливаться на ней в данной работе.

В заключение заметим также, что редактор типа позволяет помещать произвольную редактируемую им структуру в составной компонент. Это необходимо для полноты модели, в частности, на начальном

этапе, когда в библиотеке имеются только примитивные типы.

5. Заключение

Мы рассмотрели основные положения новой компонентной архитектуры, расширяющей возможности существующих компонентных моделей. Мы полагаем (и постарались аргументировать эту точку зрения), что принципы, заложенные в ней, обуславливают высокую гибкость и универсальность. Возможность порождения новых типов позволит системам, реализованным на базе данной архитектуры, развиваться значительно более интенсивно, чем это доступно традиционному программному обеспечению. ■

Литература

1. Boudreau T., Glick J., Greene S. NetBeans: The Definitive Guide. — O'Reilly Media, 2002. — 672 с.
2. Bruneton E., Coupaye T., Stefani J.B. The Fractal Component Model specification. Version 2.0-3. — The ObjectWeb Consortium, 2004. — 52 с.
3. Costa Seco J., Silva R., Piriquito M. ComponentJ: A Component-Based Programming Language with Dynamic Reconfiguration. // Computer Science and Information System. — 2008. — Vol. 5, No. 2. — С. 63-86.
4. Dos Reis G., Järvi J. What is Generic Programming? // Library-Centric Software Design. — 2005. — С. 1-11.
5. Gamma E., Helm R., Johnson R. Design Patterns: Elements of Reusable Object-Oriented Software. — Addison-Wesley, 1994. — 416 с.
6. Gosling J., Joy B., Steele G. The Java™ Language Specification. — 3-е изд. — Addison Wesley, 2005. — 688 с.
7. International Standard ISO/IEC 14772-1:1997. The Virtual Reality Modeling Language. URL: <http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/> (дата обращения: 20.05.2010)
8. Krueger C.W. Software reuse. // ACM Comput. Surv. — Vol. 2. — New York: ACM. — 1992. — С. 131-183.
9. McIlroy M.D. Mass produced software components. // Naur P., Randell B., «Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968». — Brussels: Scientific Affairs Division, NATO. — 1969. — С. 138-155.
10. Object Management Group. The Common Object Request Broker: Architecture and Specification. Version 3.1. Part 3 - Components. — OMG document formal/2008-01-08. — 2008. — 192 с.
11. Redmond F.E. DCOM: Microsoft Distributed Component Object Model. — Foster City: IDG Books Worldwide, Inc. — 1997. — 384 с.
12. Stiernerling O. Component-Based Tailorability. — Bonn: Bonn University. — 2000. — 180 с.
13. Sun Microsystems Inc. The JavaBeans™ API specification. Version 1.01-A. — Sun Microsystems Inc. — 1997. — 114 с.