Oberon с гаджетами — простая компонентная инфраструктура

Юрг Гуткнехт, ETH Цюрих; Михаэль Франц, UC Ирвайн *

1999

Аннотация

Мы представляем простую компонентную инфраструктуру, которая, „в двух словах“, обладает многими основными чертами компонентно-ориентированных программных сред, с частичным акцентом на однородность и единообразие концепций. Некоторые темы фокусируются на полностью иерархическом представлении составных объектов, представлении перманентных объектов, интерактивных и наглядных инструментов для объектной композиции и автономных и мобильных объектов. Следует обратить внимание на (а) новый вид интерфейсов родовых объектов в сочетании с протоколом сообщений, которые четко следуют принципам родительского контроля, (б) универсальную концепцию индексированных объектных библиотек и (в) альтернативный подход (в сравнении с виртуальной машиной Java) в реализации переносимого кода при помощи динамической компиляции. Наша инфраструктура основана и встроена в Oberon, язык и систему, которая работает как на „голом“ Intel-совместимом компьютере так и поверх многих коммерческих операционных систем. Из многих проектов, реализованных при помощи нашей инфраструктуры, будут кратко рассмотрены три приложения.

Ключевые слова: объектная композиция, объектно-ориентированные системы, перманентные объекты, переносимый код, компиляция на лету, мобильные объекты, Oberon, гаджеты.

Keywords: Object Composition, Object-Oriented Systems, End-User Objects, Persistent Objects, Portable Code, Just-in-time Compilation, Mobile Objects, Gadgets.

1 От объектно-ориентированных языков к компонентной культуре

Возможно, что наиболее ключевая концепция в современной аппаратной промышленности — повсеместная компонентная культура. Устройства обычно состоят из функциональных компонентов всевозможных видов и размеров, которые разрабатываются и выпускаются специализированными командами, возможно во всем мире.

*Перевод Андреева М.В. „Oberon with Gadgets - A Simple Component Framework“

Интересно, что подобной культуры не существует в программной промышленности, вероятно из-за отсутствия (а) хорошо продуманной и общепринятой нотации интерфейсов и (б) соответствующего „рынка“.

В [1] Брад Кокс (Brad Cox) рассматривается более передовая компонентная культура как первостепенная для будущих этапов разработки и использования программ. С учетом такой перспективы мы разработали экспериментальную компонентно-ориентированную инфраструктуру, которая, „в двух словах“, обладает многими основными чертами компонентной ориентированности. Наша инфраструктура тесно взаимодействует с языком Oberon и системой так, как описано в [2], [3], [4], но выходит за рамки обычного объектно-ориентированного уровня в трех отношениях: (а) перманентное представление отдельных объектов в их текущем состоянии вне рабочей среды, (б) конструирование и композиция отдельных объектов и (в) мобильность объектов.

В то время, как лежащий в основе объектно-ориентированный язык предоставляет одновременно композиционную инфраструктуру для классов объектов (позволяя вывод специализированных классов) и фабрику для создания экземпляров родовых объектов, наша компонентно-ориентированная инфраструктура в добавок поддерживает конструирование, эксплуатацию, организацию и повторное использование отдельных, сборных компонентов. Следующий пример может проиллюстрировать нашу мысль.

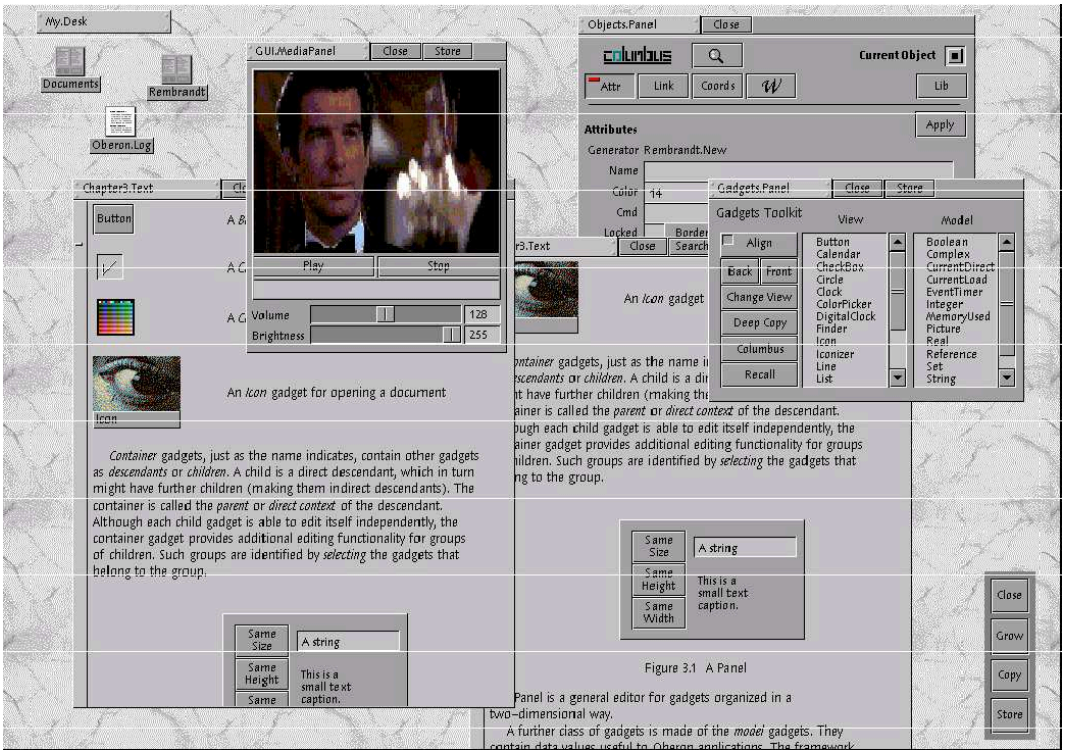

Рис. 1: Пример экрана системы Oberon с мультимедийной панелью, отображением текстового документа, инструментарием Gadgets и Columbus Inspector на рабочем столе.

Рисунок 1 показывает снимок экрана системы Oberon. Мультимедийная панель в верхней левой четверти представляет собой перманентную композицию несколь-

ких компонент: главная панель, субпанель фильма, две надписи, два бегунка и два тексовых поля. Субпанель фильма сама является композицией панели масштабирования (которая автоматически масштабирует свои компоненты), видеоэкрана, двух кнопок и текстового поля. Важно в этой связи еще раз обратить внимание на концептуальное различие между *родовыми* объектами, такими как пустая панель, надпись, бегунок, текстовое поле и т.п., которые могут быть получены напрямую из библиотеки системных классов через создание экземпляров, и *сборными* объектами, такими как субпанель фильма и сама мультимедийная панель.

Этот пример показывает двумерную структуру „пространства конструирования программ“ в компонентной культуре, где осями являются *разработка родовых компонент* и *создание и композиция объектов* соответственно. С методической точки зрения эти две оси ортогональны. Разработка родовых компонент ориентирована на классы. Это особенно ценно при программировании внутри заданной *инфраструктуры* классов, например для получения подклассов из существующих классов. Наоборот, объектная композиция ориентирована на экземпляры классов. Например, родовая панель масштабирования определена через подкласс *ScalingPanel*, который получен из класса *Panel* добавлением функциональности автомасштабирования, в то время, как специальная субпанель фильма является индивидуальной композицией родовых компонетов панели масштабирования, видеоэкрана, двух кнопок и текстового поля.

2 „Легковесная“ компонентная инфраструктура

На данный момент доступны различные программные компонентные системы. В числе наиболее известных COM/OLE/ACTIVE-X компании Microsoft и JavaBeans компании Sun. Кроме того, технологии подобные OpenDoc и CORBA от OMG используют родственные идеи, такие как составные документы и стандартизированное клиент-серверное взаимодействие соответственно.

В последующих разделах мы представим альтернативную „легковесную“ компонентную инфраструктуру. Мы распределим нашу презентацию по четырем темам, в которых мы рассмотрим (а) основные элементы компонентного программного обеспечения, (б) построение концептуальной основы и „связующего слоя“ („connecting glue“) в нашей инфраструктуре и (в) оригинальные решения, используемые в нашей системе. Эти темы следующие: (1.) протоколы сообщений в составных объектах, (2.) объектные базы данных, например перманентное представление коллекций объектов, (3.) инструменты для сборки объектов и (4.) автономность и мобильность.

3 Протоколы сообщений в составных объектах

Концепция *составных объектов* имеет фундаментальное значение в любой компонентной архитектуре. Поддерживая точную терминологию, мы ограничим себя составными объектами *контейнерного типа*, которые частично известны как элементы графического пользовательского интерфейса. Рассмотрим снова рисунок 1, в нем мы видим целую иерархию вложенных контейнеров и оконечных *атомарных* объектов: рабочий стол → мультимедийная панель → субпанель фильма → кнопка. Рисунок 2 показывает интерпретацию данной иерархии в терминах структуры данных. Мы

подчеркиваем, что наш строгий иерархический подход вознаграждается получением весьма единообразной объектной модели, в которой крупные объекты, такие как рабочий стол, средние объекты, такие как панели, и небольшие объекты, такие как кнопки, полностью унифицированны.

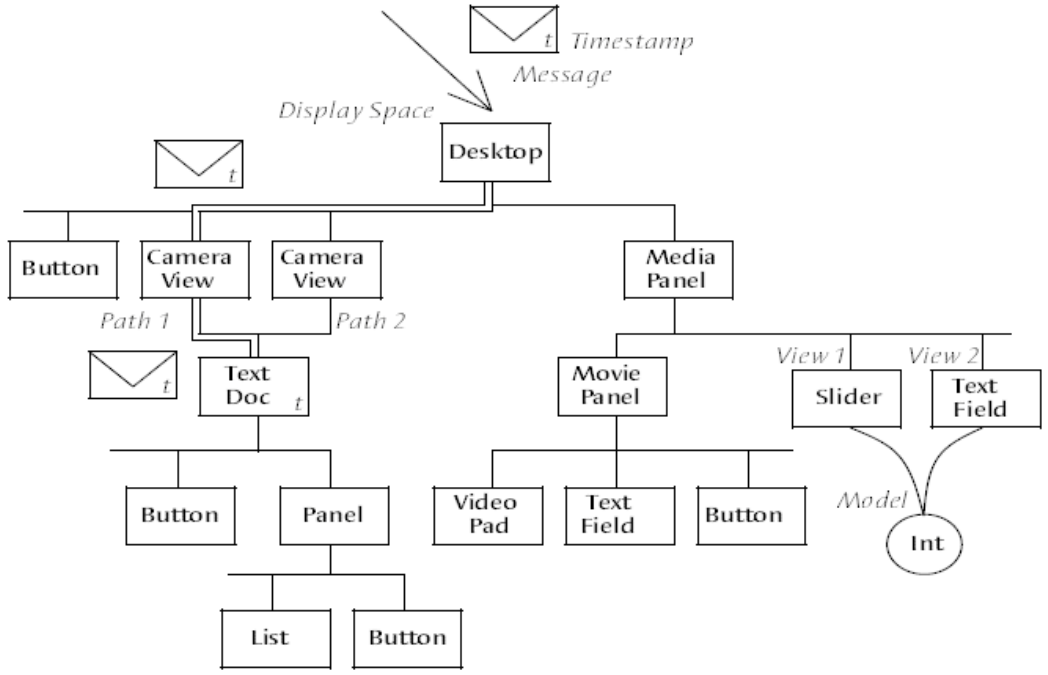

Рис. 2: Структура данных для пространства экрана из примера.

Принцип родительского контроля

Роль *контейнеров* характеризуется в нашей инфраструктуре единственным постулатом *родительского контроля*, который одновременно дает контейнерам разрешение и возлагает полную ответственность за управление их содержимым. Данный постулат имеет далеко идущие последствия. В первую очередь, он, по существу, исключает любой трафик сообщений к содержимым объектам в обход их контейнера. Другими словами, родительский контроль обязательно влечет готовность контейнеров быть посредниками в передаче сообщений и запросов.

Термин *трафик сообщений* требует пояснения. Как и в любой объектно-ориентированной среде, сообщения используются в нашей инфраструктуре для указания запросов и ответа на запросы. Однако объекты со статичным интерфейсом, как они обычно представлены в объектно-ориентированных языках, не совместимы с принципом родительского контроля, по меньшей мере в комбинации с родовыми контейнерами, которые могут содержать потенциально неизвестные (будущие) типы и, соответственно, получать потенциально неизвестные сообщения.

Исходя из этих соображений, мы используем новый вид *родового объектного интерфейса*, который полагается на встроенный интерпретатор, способный интерпретировать и пересылать произвольные приходящие сообщения соответствующим образом.

Технически, если *Message* — это базовый тип сообщений, принимаемых экземплярами некоторого данного типа объектов, а *MessageA* и *MessageB* — подтипы *Message*, тогда диспетчер будет иметь следующую структуру:

```
PROCEDURE Dispatch (me: Object; VAR m: Message);
BEGIN
  (* общие для всех сообщений действия *)
  IF m IS MessageA THEN (* обработка сообщений подтипа MessageA *)
    ELSEIF m IS MessageB THEN (* обработка сообщений подтипа MessageB *)
    ELSE (* обработка прочих сообщений *)
  END
END Dispatch;
```

Заметим в частности, что диспетчер (а) способен выполнять некоторые общие (подготовительные) действия, (б) использует безопасную *проверку типа IS* языка Oberon для распознавания сообщений, (в) вызывает обработчик нераспознанных сообщений и (г) он расширяем благодаря подтипам сообщений без необходимости изменения его интерфейса.

Ради единообразия, мы также используем родовые интерфейсы для атомарных (т.е. неконтейнерных) объектов. Вместе с этим, посылка сообщений в пространстве экрана выражается в иерархическом обходе структуры, управляемом диспетчерами вышеуказанного вида. Простейший случай — это *целе-ориентированная* посылка, когда целевой объект указан в самом сообщении. Типичные примеры целе-ориентированных сообщений — это запросы на получение, установку и перечисление атрибутов („свойств“) некоторого объекта. Однако, существуют интересные варианты посылки сообщений. Например, универсальное уведомление без указания цели, обычно *широковещательное* сообщение в пространстве экрана или в одном из его подпространств. Другие варианты затрагивают *инкрементальную обработку* (инкрементальный вклад в обработку сообщения различными диспетчерами) и *контекстно-зависимое поведение* (поведение зависит от пути, по которому пришло сообщение). Применение будет рассмотрено в последующих разделах.

Камеры (Camera Views)

Схема MVC [12] — это фундаментальный шаблон проектирования и неотъемлемый аспект каждой объектной архитектуры, которая обеспечивает концептуально чистое разделение в отношениях *моделирования*, *отображения* и *контролирования* объектов. В нашем случае используется весьма общая интерпретация MVC. В простом случае один или более *визуальных объектов* (с изображением в пространстве экрана) обслуживаются как отображение некоторого *абстрактного объекта*. Примеры данного рода: (а) чекбокс отображение для объекта *Boolean*, (б) бегунок или текстовое поле (или оба одновременно) для объекта *Integer*, (в) цветовая палитра для вектора цвета (красный, зеленый, синий).

Более сложный вариант возникает в случае отображения отображений, названный впоследствии *камерой (camera views)*. Камеры полезны для множества целей. Они обеспечивают концептуальную основу для множества отображений одного и того же визуального документа на одном или нескольких экранах. Например, рисунок 1

содержит двойное отображение одного текстового документа со встроенными визуальными объектами. Интересный вариант камеры — *функциональные отображения*, которые дополняются некоторой специфичной функциональностью. Например, в дополнение к обычным *пользовательским отображениям* могут быть предложены *отображения для разработчика (developer views)* для поддержки интерактивного редактирования и конструирования визуальных объектов на месте.

Камеры реализованы как специальный вид визуальных объектов, который способен отображать визуальные модели. Как приятное следствие, общие части структуры данных представляющих пространство экрана автоматически доступны камере и ненужное дублирование избегается, как это показано на рисунке 2. Ясно, что это добавляет одновременно и сложность и гибкость в процедуру обработки сообщений в пространстве экрана. Сообщения теперь могут поступать к объектам по различным путям и возникает необходимость ставить *временные метки (time-stamp)* для определения многократных доставок. С другой стороны, теперь возможно использовать преимущества *контекстно-чувствительной* обработки, например, для реализации вышеупомянутых отображений для разработчика.

Последующие упрощенные примеры обработки сообщений в общем и в пространстве экрана в частности могут поспособствовать лучшему пониманию концепций, обсуждаемых в этом разделе, и их комбинаций.

Уведомления об изменениях

Извещения об изменениях посылаются моделью или контроллером (в случае разумной группировки (smart bundling)) для уведомления потенциальных отображений о изменении состояния. Они всегда направлены в пространство экрана в виде явного *широковещательного запроса*. Затронутые отображения обычно восстанавливают согласованность с их моделью после запроса о ее текущем состоянии. Широковещательные сообщения проще и обобщеннее чем альтернативные методы, такие как списки обратных вызовов, но менее эффективны. Обычно, и это показывает практика, снижение эффективности незначительное. Можно легко произвести оптимизацию, например, добавляя „знания“ в критические контейнеры. Мы делаем вывод, что интерфейсы родовых сообщений абсолютно необходимы для применения широковещательного метода в строго типизированной инфраструктуре.

Запросы отображения

Данный тип сообщений используется для запроса визуального *целевого объекта* в пространстве экрана показать себя. Например, данный запрос может быть сделан реорганизованным контейнером к каждому из его содержимых объектов или получателем уведомления об изменениях для настройки своего отображения. Запросы отображения также адресованы всему пространству экрана в целом. Они требуют инкрементальной обработки во время обхода иерархии контейнера в двух отношениях: накопление относительных координат и вычисление трафарета перекрытия (overlap mask). Если используются камеры, то возможны различные пути к целевому объекту, таким образом он должен быть подготовлен к множественным доставкам сообщения. Все доставки обрабатываются одинаковым способом, возможно с различными абсолютными координатами и трафаретами перекрытия.

Запросы копирования

Копирование или *клонирование* — это элементарная операция над объектами. Тем не менее, в случае составных объектов, довольно сложная. Очевидно, что обобщенная операция копирования объектов эквивалентна алгоритму копирования любой произвольной и действительно разнородной структуры данных. Более того, существуют различные варианты копирования. Например, *детальная копия (deep copy)* составного объекта состоит из копии как контейнера, так и копий его содержимого. В то время как *поверхностная копия (shallow copy)* обычно включает только новые отображения исходного содержимого.

Наша реализация операции копирования снова базируется на широковещании сообщений. В этом случае множественные доставки сообщения должны быть обработаны с большой осторожностью.

```
IF сообщение пришло первый раз THEN
  создать копию контейнера;
  IF запрошена детальная копия THEN
    передать сообщение содержимому;
    связать копии содержимого с копией контейнера;
  ELSE (* запрошена поверхностная копия *)
    связать содержимое с копией контейнера;
  END
END;
RETURN копия контейнера
```

Заметим, что получатели на самом деле обладают некоторой свободой в обработке запросов копирования. Например, „тяжеловесный“ объект, получив сообщение о детальном копировании, может решить просто вернуть некоторое новое отображение себя или даже самого себя (что ведет к *копированию по ссылке (copy by reference)*) вместо настоящей копии.

4 Объектные библиотеки как многосторонняя и универсальная концепция

Объектная персистентность обычно выражается в возможности отдельных объектов быть сохраненными на некотором внешнем устройстве (обычно диске) вместе с их текущим состоянием. Неотъемлемая часть любой подобной возможности — два преобразующих метода, называемых *выгрузка (externalizer)* и *загрузка (internalizer)* соответственно. Выгрузка используются для преобразования объектов из их внутреннего представления в инвариантную, линейную форму, загрузка используются для обратного преобразования. Проблема аналогична в своей сути проблеме копирования, которая уже обсуждалась. Тем не менее, один дополнительный аспект следует уточнить: инвариантное представление указателей.

Наш подход в представлении инвариантных указателей основан на образовании индексированных множеств объектов, названных *объектными библиотеками (object libraries)*. Идея заключается в реализации объектной линеаризации через (рекурсивную) регистрацию компонентов в некоторой объектной библиотеке с заменой

указателей на ссылочные индексы. Вместе с этим, выгрузка и загрузка становятся „распределенными“ дву-проходными процессами, которые снова используют широковещательные сообщения внутри нужного объекта:

Алгоритм выгрузки

```
Externalize (объект X) =
  { Create(библиотека L); Register(X, L); Externalize(L) }

Register (объект X, библиотека L) = {
  WITH X DO
*   FOR ALL компонентов x из X DO Register(x, L) END
  END;
  IF X незарегистрирован THEN
    назначить индекс и зарегистрировать X в L
  END
}

Externalize (библиотека L) = {
  WITH L DO
    FOR индекс i := 0 TO max DO
      WITH объект X[i] DO выгрузить X[i];
*       заменить указатели на индексы
        и выгрузить дескриптор X[i]
      END
    END
  END
}
```

Ясно, что ацикличность связей содержимого является необходимым условием для данного алгоритма. Далее заметим, что операторы, помеченные заком „*“ должны быть реализованны как объектные методы, т.к. они зависят от типа объекта.

Алгоритм загрузки

```
Internalize (библиотека L) = {
  WITH L DO
    FOR индекс i := 0 TO max DO
      загрузить X[i];  создать дескриптор X[i]
    END
    FOR индекс i := 0 TO max DO
*     загрузить дескриптор X[i]
      и заменить индексы на указатели
    END
  END
}
```

Заметим, что загрузка библиотеки — потенциально рекурсивный процесс, т.к. индексы в загружаемых объектных дескрипторах могут ссылаться на другие библиотеки. И, снова, операторы, помеченные знаком „*“ должны быть реализованы как объектные методы.

Объектные библиотеки неожиданно оказались многосторонней и универсальной концепцией. Диапазон их применения намного шире, чем кто-либо мог предположить. Помимо поддержки загрузки и выгрузки отдельных объектов они выполняют роль простых *объектных баз данных*, которые служат для организации любого локального или распределенного пространства объектов. Некоторые типичные олицетворения этого: (а) Коллекции логически связанных повторно используемых *компонентов*, (б) Коллекции *общедоступных* объектов, разделяемых множеством документов и (в) множество *приватных* объектов в некотором документе.

Вливание объектов в текст (Objects Flowing in Text)

Другое универсальное применение концепции объектных библиотек — *обобщенные тексты (generalized texts)*. Простая новая интерпретация обычных (многошрифтовых) текстов как последовательностей ссылок на шаблоны символов (где номер ссылки является ASCII-кодом) начинает путь к далеко идущему обобщению. Позволяя ссылки на различные объектные библиотеки вместо только шрифтовых, мы немедленно получаем тексты со встроенными объектами произвольного вида, включая изображения, связи, элементы форматирования, целые функциональные панели и другие сущности подобные „аплетам“ (applets) *Java*. Дополнительная гибкость обеспечивается возможностью встраивать как приватные объекты (собранные в так называемой *приватной библиотеке (private library)* текста), так и общедоступные объекты (принадлежащие некоторой общедоступной библиотеке).

Такая высокая интеграция текста с объектами весма полезна в основном в области документирования. Благодаря этому функциональные сущности любой сложности и гранулярности, которые могут быть разработаны где угодно, можно просто скопировать и совместить с их текстовой документацией. Показательный пример, глава из электронной книги об Oberon, показан на рисунке 1.

5 Инструменты композиции объектов

В принципе, существует два различных метода для конструирования и композиции объектов: интерактивный и описательный. *Интерактивные методы* основаны на непосредственном редактировании в противоположность *описательным методам*, которые обычно полагаются на некоторый формальный язык и соответствующий интерпретатор. В большинстве случаев эти два метода взаимозаменяемы. Несмотря на это, интерактивный метод больше подходит для конструирования визуальных GUI-объектов, а описательный метод предпочтителен для конструирования упорядоченных структур (regular layouts), необходим для програмно созданных объектов (таких как „список свойств“ („property-sheets“) и т.п.) и для невизуальных (модельных) объектов. Следующая таблица резюмирует вышесказанное:

Вид объекта	Подходящий метод конструирования
Визуальный GUI	интерактивный
Упорядоченная структура	описательный
Программно созданный	описательный
Невизуальная модель	описательный

Независимо от метода конструирования, компоненты могут быть полученны альтернативно из (а) генераторов атомарных объектов, (б) генераторов контейнерных объектов и (в) библиотек готовых объектов.

Интерактивное конструирование

Наша инфраструктура поддерживает интерактивное конструирование на различных уровнях. На *инструментальном уровне (tools level)*, инструментарий *Гаджетов (Gadgets)*[5, 6, 7] и *Columbus inspector*, показанный на рисунке 1, предлагают функциональность для

- создания новых экземпляров существующего типа
- вызова готовых экземпляров из объектной библиотеки
- выравнивания компонентов в контейнерах
- установления связей модели с отображением
- инспектирования состояния, атрибутов и свойств объектов
- привязки команд Oberon к объектам GUI

На *уровне отображения* ранее упомянутые отображения для разработчика позволяют редактировать визуальные объекты. На уровне объектов поддержка редактирования на месте обеспечивается встроенными *локальными редакторами (local editors)*. Сообщения о событиях мыши помечаются парой абсолютных координат указателя мыши и подвергаются координатно-ориентированной диспетчеризации в пространстве экрана. Т.к. события мыши должны быть обработаны по разному в контекстах пользователя и разработчика, большинство обработчиков событий мыши выигрывают от использования контекстно-зависимой обработки сообщений.

Описательное конструирование

Описательное конструирование требует наличия формального описательного языка в качестве основного компонента. Т.к. спецификация компоновки функциональна, мы выбрали функциональный язык с LISP-подобным синтаксисом.

Мы в основном различаем два метода обработки функционального описания объектов: (а) *компиляция* и (б) непосредственная *интерпретация*. Отдельные описания компилируются в объектные библиотеки, в то время как описания, которые встроены в контекст некоторого документа (например HTML-страницы), обычно интерпретируются и транслируются прямо во встраиваемый объект. Рисунок 3 представляет наглядно метод компиляции. Отметим, что родовые объекты извлекаются

компилятором из библиотеки классов клонированием, в то время как готовые объекты импортируются из любой библиотеки объектов либо клонированием либо по ссылке.

Следующего прокоментированного примера функционального описания мультимедийной панели из рисунка 1 будет достаточно, чтобы получить представление об описательном конструировании в нашей инфраструктуре.

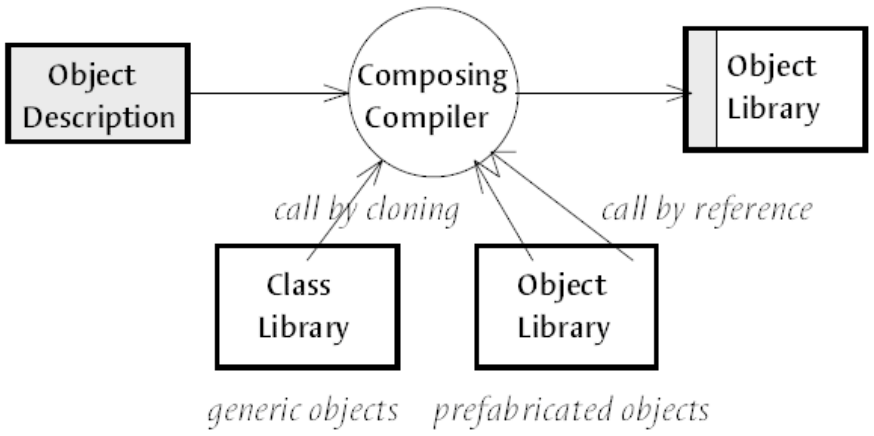

Рис. 3: Сборка объектов через компиляцию функционального описания.

```
(LIB GUI
  (FRAME MediaPanel (OBJ Panels.NewPanel)
    (Volume (OBJ BasicGadgets.NewInteger (Value 100)))
    (Brightness (OBJ BasicGadgets.NewInteger (Value 200)))
    (GRID 2:50 1:* 1:25% 1:50% 1:25%)
    (PAD 2 @ 2)
    (FRAME (POS 1 @ 1) (OBJ TextFields.NewCaption))
      (Value "Brightness"))
    (FRAME (POS 1 @ 2) (OBJ BasicGadgets.NewSlider)
      (Max 255)
      (Model Brightness)
      (Cmd "Movie.SetBright #Value Movie"))
    (FRAME (POS 1 @ 3) (OBJ TextFields.NewTextField)
      (Model Brightness)
      (Cmd "Movie.SetBright #Value Movie"))
    (FRAME (POS 2 @ 1) (OBJ TextFields.NewCaption))
      (Value "Volume"))
    (FRAME (POS 2 @ 2) (OBJ BasicGadgets.NewSlider)
      (Max 255)
      (Model Volume)
      (Cmd "Movie.SetVol #Value Movie"))
    (FRAME (POS 2 @ 3) (OBJ TextFields.NewTextField)
      (Model Volume)
      (Cmd "Movie.SetVol #Value Movie"))
    (FRAME (POS 3 @ 1:3) (OBJ MoviePanel.Movie)
      (SIZE 296 @ 246))))
```

Коментарии

1. (Составные, визуальные) объекты в основном определяются как вложенные иерархии *фреймов*, где каждый фрейм может опционально использовать выражение OBJ для определения объекта-контейнера (carrier-object).

2. В результате компиляции вешеуказанного описания получается файл объектной библиотеки, названной *GUI*, содержащей экземпляр конструируемого объекта, названного *MediaPanel*.

3. Фреймы могут опционально определять локальные объекты, которые обычно используются как модели. Например, здесь определены два подобных модельных объекта, один для регулировки громкости, другой для регулировки яркости.

4. Внутри структурного компонента OBJ первый идентификатор определяет либо генерирующую процедуру *Module.Procedure* (представляя класс требуемого объекта), либо готовый объект *Library.Object*, который *импортируется* по ссылке или копируется из библиотеки объектов.

5. Визуальные объекты обычно определяют *сетку* вида *строки* @ *столбцы*. В нашем случае сетка состоит из трех строк и трех столбцов соответственно. Высота первых двух строк равна 50, в то время как высота третьей строки не задана и определяется содержимым. Ширина столбцов указана в процентах, а общая ширина снова не указана. Первые две строки снизу представляют элементы регулировки яркости и громкости соответственно. Каждая строка состоит из заголовка, бегунка и текстового поля, причем бегунок и текстовое поле сдвоены локальной моделью. Третья строка охватывает три столбца и отображает готовый объект, названный *MoviePanel*, который импортируется из библиотеки, названной *Movies*.

Мы должны отметить, что рассмотренный инструмент функционального описания объекта — это лишь часть более общей экспериментальной системы, названной *Powerdoc*, которая позволяет описывать конструкцию произвольных обобщенных документов, которые могут включать любые виды пассивных или активных объектов („applets“) и потоки данных.

6 От перманентных объектов к мобильным объектам

В предыдущей дискуссии мы раскрыли нашу компонентную архитектуру до состояния, включающего механизм для внешнего, линейного представления обычных объектов. До сих пор мы использовали эту возможность только для сохранения объектов на внешних запоминающих устройствах. Тем не менее существует второе потенциальное применение: *мобильность (mobility)*. В данном случае линейное представление, очевидно, должно сопровождаться (а) *автономностью (self-containedness)* и (б) *переносимостью (portability)*. Эта идея рассматривается в данном и последующем разделах.

Необходимое требование для *автономных объектов (self-contained objects)* — это их полнота в смысле использованных ресурсов. Благодаря нашей весьма унифицированной архитектуре, только два вида ресурсов существует: *библиотеки объектов (object libraries)* и *программные модули (program modules)* (элементы библиотеки классов).

К сожалению, невозможно определить все ресурсы, которые использует некоторый общий объект. Дело в том, что ресурсы часто указываются явно в неприметной строке. Например, имена команд *Module.Procedure* обычно спрятаны в строковых атрибутах кнопок, а имена объектов *Library.Object* могут встречаться в произвольном прокручиваемом списке.

Наше решение проблемы определения ресурсов заключается в двух новых ингредиентах: (а) *сообщения запроса ресурсов (resource query message)* используется для сбора информации о ресурсах и (б) *объект управления ресурсами (resource management object)* действует как упаковщик автономных объектов. Если X — произвольный автономный объект и M — объект управления ресурсами, то упакованная композиция MX выгружается следующим способом:

```
Выгрузка автономного объекта MX = {
  Послать сообщение запроса ресурсов Q объекту X;
  Выгрузить M;
  Выгрузить X
}
```

В случае контейнеров, снова используются преимущества стратегии широковещания для обработки сообщения запроса ресурсов:

```
Обработка сообщения запроса ресурсов Q = {
  Послать Q содержимым объектам;
  Отчитаться о своих ресурсах менеджеру ресурсов
}
```

В комбинации с *мобильностью*, за кажущейся простотой данного алгоритма скрыто несколько значительных проблем. Среди них (а) область видимости имен ресурсов и (б) защита целевой системы от вредоносного или ошибочного кода. Теперь мы кратко затронем проблему (а), и пока отложим краткое высказывание по проблеме (б) до следующего раздела.

Мобильные объекты разрабатываются, в общем, без какого бы то ни было глобального согласования. Как следствие, они определяют свою собственную область видимости ресурсов, которая, в случае миграции, нуждается в отображении в отдельное пространство целевой системы. Тем не менее, разумно выделить некоторый общий набор *ресурсов ядра (kernel resources)*, который должен быть одинаково доступен на любой целевой системе. Очевидно, что нет необходимости переносить ресурсы ядра с каждым индивидуальным объектом, но есть необходимость в проверке их совместимости, возможно при помощи *fingerprints* [13] .

Мы делаем вывод, что наш подход к мобильным объектам общий в том смысле, что, в принципе, любой перманентный объект может быть сделан мобильным. Поэтому спектр возможностей мобильных объектов покрывает впечатляющий диапазон: от простых кнопок и чекбоксов до контрольных панелей и документов, и, наконец, рабочих столов, представляющих целую систему Oberon.

7 Эффективный подход к переносимому коду

Финальный аспект мобильных объектов который заслуживает рассмотрения — это *кроссплатформенная переносимость* реализующего их кода. Т.к. ожидается, что мобильные объекты надолго переживут создавшую их среду и все существующие на данный момент аппаратные архитектуры, то выбор *формата распространения (distribution format)* программ должен руководствоваться не доминирующими на данный момент семействами процессоров, а более фундаментальными суждениями. В начале может показаться прекрасным тактическое решение на данный момент распространять мобильные объекты в форме бинарного кода i80386, который может исполняться на большинстве используемых компьютерах (и интерпретироваться на большинстве остальных), но это может оказаться плохим выбором в долговременной перспективе. Формат распространения, ориентированный на будущее, должен удовлетворять трем основным требованиям: он должен быть (а) подходящим для *быстрой трансляции* в собственный код современных и будущих микропроцессоров, (б) не препятствовать расширенной *оптимизации кода*, требующейся для современных суперскалярных процессоров и, предвидя важность низкоскоростных безпроводных сетей в недалеком будущем, он должен быть (в) *очень компактным*.

Наша концепция переносимого кода заключается в формате распространения, названном *Slim Binaries* [14], который удовлетворяет всем перечисленным требованиям. В противоположность решениям подобным *p-code* и *Java byte-code* [15], которые опираются на идею виртуальной машины, формат slim binary — это адаптивно-сжатое представление *синтаксических деревьев*. В кодировании slim-binary каждый символ описывает поддерево абстрактного синтаксического дерева в терминах поддеревьев, предшествующих ему. Прще говоря, процесс кодирования заключаетя в выгрузке поддеревьев и одновременно в постоянном расширении „словаря“ , который используется для кодирования последующих частей программы.

Данный формат имеет очевидный недостаток — он не может быть декодирован простой поточечной интерпретацией. Семантика любого отдельного символа в потоке данных slim-binary раскрывается лишь после того, как будут обработаны все предшествующие ему символы. Произвольный доступ к отдельным инструкциям, как это обычно требуется для интерпретации, не возможен.

Тем не менее, отказываясь от поточечной интерпретации (чья ценность так или иначе ограничена в следствии низкой эффективности), мы получаем несколько важных преимуществ. Первое их них в том, что формат наших программ исключительно компактен. Например, он более чем в два раза плотнее Java байт-кода и его производительность значительно лучше чем у стандартных алгоритмов сжатия, таких как *LZW*, примененных либо к исходному, либо к объектному коду (для любой архитектуры). Это преимущество нельзя переоценить. Факт, опыты показывают, что генерация кода „на лету“ дается даром благодаря высокой компактности выбранного представления программ, так как дополнительные вычислительные расходы компенсируются полностью снижением расходов на операции ввода/вывода [16].

Второе, древовидная структура нашего формата обладает значительными преимуществами в случае, если архитектура целевой машины требует расширенную *оптимизацию*. Множество современных технологий оптимизации кода использует структурную информацию, которая доступна на уровне синтаксических деревьев, но ее трудно выделить из байт-кода.

Третье, в отличии от других представлений, кодировка slim-binary сохраняет информацию о *типе (type)* и *области действия (scope)*. Соответственно это дает возможность определять вредоносный или некорректный код, который потенциально может нарушить целостность базовой системы. Например, легко отловить любую попытку доступа к скрытым переменным общедоступных объектов, которая может быть разрешена нестандартным компилятором. В случае байт-кода такой анализ более сложен.

В четвертых, мы можем использовать нашу технологию оптимизации slim-binary кода за границами модуля. Такая глобальная оптимизация, как, например, встраивание кода (inlining) и межпроцедурное распределение регистров, весьма эффективна в компонентной среде, собранной из большого числа относительно небольших и независимых частей кода.

Когда кусок slim-binary кода первый раз загружается в систему, он транслируется в „родной“ код за один проход, платя эффективностью кода за скорость компиляции. После создания кусок кода немедленно становится объектом для (а) исполнения и (б) оптимизации. После выполнения шага оптимизации, предыдущее поколение кода просто заменяется на новое. Очевидно, совместно с анализом времени исполнения, эта процедура может повторяться многократно и производить более совершенные поколения кода.

Применения

Наша инфраструктура используется в многочисленных проектах. Три серьезных приложения — среда *Computer Aided Control System Design* (CACSD) [17], родовая *архитектура управления роботами (robot controller architecture)* [18] и служба реального времени потокового аудио/видео (real-time audio-/video-stream service) в локальной коммутируемой сети. В среде CACSD заслуживают внимание гаджеты matrix и plot, которые связаны (за сценой) с движком Matlab, и мощные узловые объекты, которые представляют действия системы контроля в дереве действий. В проекте управления роботами объектные библиотеки выгодно используются для перманентного, но независимого представления *объектов ввода/вывода*, например, сенсоров, актуаторов и т.п. Другая интересная возможность — *удаленные отображения (remote views)*, это , например, отображения, которые в системе разработки (подключенной к роботу через интернет), показывают состояние модели робота. В заключение, в проекте потокового сервера (stream-server) новый вид визуальных объектов, показывающих удаленно получаемый контент (видеопотоки, не пересылаемые в оперативную память клиентов), беспрепятственно интегрирован в систему гаджетов (Gadgets).

Заключение

Используя оригинальную систему и язык Oberon, мы разработали компонентную инфраструктуру которая, строго говоря, делится на три взаимодействующие подсистемы:

- инфраструтура компонентов пользовательского интерфейса, которые могут быть настроены интерактивно или описательно

- инфраструктура загрузки и выгрузки объектных структур
- инфраструктура для упаковки объектных структур в замкнутую структуру

Подсистемы объеденены двумя универсальными и унифицированными возможностями: (а) „программная шина“ („software bus“) в форме родового протокола сообщений, подчиняющегося принципам родительского контроля, и (б) объектная база данных в форме иерархии индексированных объектных библиотек.

В противоположность *COM* и *Corba*, акцент в нашей системе сделан на единообразие, а не на платформонезависимость. Фактически, компоненты Oberon не жизнеспособны ни в какой среде, кроме среды Oberon, с важным исключением в лице HTML среды со встраиваемым модулем Oberon. С другой стороны, компоненты Oberon заполняют целый диапазон от простых буквеных глифов до полноценных документов и рабочих столов.

Наша архитектура характеризуется четким разделением композиции объектов от разработки компонентов. В то время, как родовые компоненты (атомарные и контейнеры) представлены классами Oberon и запрограммированы на Oberon (особенно реализация протокола сообщений родительского контроля), составные объекты созданы либо интерактивно при помощи встроенных редакторов, либо описательно в LISP-подобной нотации. В отличии, например, от мастеров Visual Basic и Developer Studio фирмы Microsoft, Borland Delphi и Sun JavaBeans, инструменты компоновщика Oberon не отображают составные объекты в классы и конструкторы, а напрямую создают структуры данных в форме ОАГ (DAG), которые можно выгрузить в идентифицируемые элементы некоторой объектной библиотеки. Любой составной объект может быть вызван и использован где угодно либо при помощи ссылки, либо при помощи клонирования.

Система работает на линейке платформ Intel PC. Так же доступны версии для систем Windows и Macintosh. На данный момент в систему включены все упомянутые возможности, множество приложений и продвинутый навигатор, который спроектирован единообразно, работает как с составными документами Oberon, так и с HTML гипертекстом. Мобильные объекты и динамическая рекомпиляция на данный момент находятся в стадии разработки. Планы в конечном счете нацелены на парадигму „Oberon in gadget“.

Несколько солидных проектов реализовано при помощи интенсивного использования нашей компонентной инфраструктуры. Они продемонстрировали не только ее мощь, но и достаточный потенциал для дальнейшего обобщения.

Список литературы

[1] Brad J. Cox, „Object Oriented Programming, An Evolutionary Approach“, Addison Wesley, 1986.

[2] N. Wirth, „The Programming Language Oberon“, Software – Practice and Experience, 18(7), 671-690.

[3] N. Wirth and J. Gutknecht, „The Oberon System“, Software – Practice and Experience, 19(9), September 1988.

[4] N. Wirth and J. Gutknecht, „Project Oberon“,
Addison Wesley, 1992.

[5] J. Gutknecht, „Oberon System 3: Vision of a Future Software Technology“,
Software-Concepts and Tools, 15:26-33; 1994.

[6] Johannes L. Marais; „Towards End-User Objects: The Gadgets User Interface System“,
Advances in Modular

[7] A. Fischer and H. Marais, „The Oberon Companion“, A Guide to Using and Programming Oberon System 3, vdf Hochschulverlag AG an der ETH Zürich, 1998.

[8] K. Brockschmidt, „Inside OLE“; Microsoft Press; 1993.

[9] J. Feghhi, Web Developer's Guide to Java Beans, Coriolis Group Books, 1997.

[10] The OpenDoc Design Team, „OpenDoc Technical Summary“, Apple Computer, Inc.; October 1993.

[11] R. Ben-Natan, „CORBA: A Guide to Common Object Request Broker Architecture“, Mc Graw-Hill; 1995.

[12] G.E. Krasner and S.T. Pope, „A Cookbook for using the Model-View- Controller user interface paradigm in Smalltalk-80“, Journal of Object-Oriented Programming, 1(3):26-49; August 1988.

[13] R. Crelier, „Extending Module Interfaces without Invalidating Clients“, Structured Programming, 16:1, 49-62; 1996.

[14] M. Franz and T. Kistler, „Slim Binaries“, Communications of the ACM, 40:12, 87-94; December 1997.

[15] T. Lindholm, F. Yellin, B. Joy, and K. Walrath, „The Java Virtual Machine“, Specification; Addison-Wesley; 1996.

[16] M. Franz, „Code-Generation On-the-Fly: A Key to Portable Software“, Doctoral Dissertation No. 10497, ETH Zürich, simultaneously published by Verlag der Fachvereine, Zürich, ISBN 3-7281-2115-0; 1994.

[17] X. Qiu, W. Schaufelberger, J. Wang, Y. Sun, „Applying O3CACSD to Control System Design and Rapid Prototyping“, The Seventeenth American Control Conference (ACC'98), Philadelphia, USA, June 24-26, 1998.

[18] R. Roshardt, „Modular robot controller aids exible manufacturing“, Robotics Newsletter, International Federation of Robotics, No. 16, Dec. 1994.

Благодарности

Мы благодарим Никлауса Вирта, без его инициативы и самотверженности оригинальный язык и система Oberon не были бы созданы. Среди множества работ мы особенно выделяем работу Hannes Marais, чья инфраструктура и пакет разработчика Gadgets указали нам направление исследований. Мы так же весьма благодарны работам Emil Zeller, Ralph Sommerer, Patrick Saladin, Joerg Derungs и Thomas Kistler в области составных и распределенных документов, описании композиции объектов и конструирования компиляторов соответственно. Большое спасибо Pieter Muller, который реализовал „родное“ ядро Oberon для PC. И последнее, но не менее важное, мы благодарны за множество конструктивных замечаний анонимных рецензентов.