Оберон-технологии: что это такое?

И. Е. Ермаков

Сегодня для многих программистов языки и системы «Оберон» уже не являются terra incognita. Много написано публикаций на русском языке об истории и современности этого семейства языков программирования, являющихся развитием Паскаля и Модулы в рамках многолетнего эксперимента швейцарской школы программирования. В обсуждениях на форумах все чаще используется термин «Оберон-технологии» - и все чаще выдвигается требование четко сформулировать, что это такое. Действительно, чтобы говорить о некотором явлении, а тем более критически его осмысливать, надо ясно представлять себе его структуру и суть. Попробуем разложить по полочкам основные компонентны Оберон-технологий. Здесь можно выделить несколько уровней:

I. Концептуальный — общая идеология, подходы и стиль мышления.

II. Технический и методологический — конкретные механизмы и методологии, которые состаляют арсенал программиста.

III. Практический и конкретно-языковый — уровень «важных мелочей», из которых складывается повседневная работа и которые в совокупности играют очень большую роль.

IV. Исторический — практический опыт, накопленный в рамках направления; традиции, так сказать.

I. Концептуальный уровень.

1. Упрощение. Об этом много говорилось — здесь работают таки принципы, как «не множь сущности без необходимости» (бритва Оккама), «чем проще система, тем она надежнее» и т.д. Важен стиль мышления, направленный на поиск оптимального решения, достаточно, но не избыточно общего. Мотивом к такому поиску является понимание того, что решаемые задачи сами по себе достаточно сложны, чтобы еще искать для себя сложностей в изучении темных закоулков инструментария. Программист ориентирован на решение задач, а не сам по себе процесс программирования.

2. Базисность. Стремление на каждом этапе выделить некоторый компактный набор основных принципов (возможностей), на основе которых можно дальше строить систему. При этом в базисе не должно быть избыточных элементов, которые дублируются или пересекаются друг с другом. Когда решение ИТ-задач ведется на основе небольшого набора примитивов, повышается надежность систем, облегчается понимание их устройства, легко обеспечивать контроль, сопровождение и расширение.

3. Открытость. Программист стремится строить всегда расширяемые системы, которые могут быть дополнены или интегрированы в дальнейшем. Используемый инструментарий позволяет не заботиться о технических аспектах такой расширяемости, в частности, не требуется разрабатывать собственных протоколов интеграции и взаимодействия компонентов — все это обеспечивается автоматически. Также инструментарий дает гарантии безопасного использования компонент, разработанных различными производителями. Эти аспекты подробно описаны в книгах К. Пфистера «Компонентное ПО» (`http://blackbox.metasystems.ru/`, раздел Статьи) и Szyperski C. Component Software. Beyond Object-Oriented Programming, Addison Wesley Longman, 1998.

4. Жесткий контроль корректности кода на всех этапах и во всех формах — на уровне компилятора, механизмами времени выполнения, явное специфицирование предусловий в коде. При таком подходе совершить ошибку становится гораздо труднее, а выявление происходит очень быстро, даже при отсутствии специализированного этапа тестирования. Системы обладают предсказуемым поведением, отклонение от которого быстро выявляется и полностью локализуется. Большинство Оберон-систем создавалось малыми командами, и возможности организовать специализированный этап контроля качества обычно не было. Тем не менее, соотношение «трудозатраты — надежность» несравнимо выше среднего по индустрии.

II. Технический и методологический уровень.

Здесь в первую очередь будем говорить о первом поколении Оберонов (Оберон-1, Оберон-2, Компонентный Паскаль). Второе поколение (Active Oberon, Zonnon) является пока экспериментальным и включает в себя много дополнительных составляющих, о которых нужно говорить особо.

1. Обероны построены на гармоничном балансе структурного программирования и ООП. Архитектурной единицей программной системы является модуль. Это единица проектирования, разработки, инкапсуляции, компиляции, распространения, развертывания, загрузки и выполнения. ООП в Оберонах не несет на себе архитектурной нагрузки, оно затрагивает исключительно абстракции данных. Классическая Оберон-система состоит из модулей — «черных ящиков», связанных между собой процедурными шинами, по которым передаются объектно-ориентированные данные.

2. Прямая и полная поддержка компонентного подхода, а конкретно — следующих его составляющих:

   (a) динамическая загрузка и компоновка модулей;

   (b) ООП (однако рекомендуется не использовать межмодульное наследование реализации, из-за проблемы «хрупкого базового класса», fragile base class. Ее суть в том, что в случае наследования реализации разработчик базового класса уже не может менять реализацию своего класса, не опасаясь нарушить работу чужих расширений);

   (c) автоматическое управление памятью («сборка мусора»);

(d) динамическая работа с типами и метапрограммирование.

В таком сочетании для эффективного компилируемого языка эти качества впервые появились именно в Обероне. Например, сборка мусора, как и динамические среды выполнения, задолго до Оберона активно использовались в функциональных и объектных интерпретируемых языках (LISP, Smalltalk). «Первой ласточкой» среди компилируемых императивных языков стал Оберон в 1989 году.

3. «Базисная» реализация ООП. ООП зародилось как самостоятельная методология, альтернативная императивному и функциональному программированию. Классический объектный язык Smalltalk базируется на единственной концепции — концепции объекта, выполнение программы — это посылка сообщений объектам. (Smalltalk — интересный язык, идеология которого во многом созвучна идеологии Оберонов. На простом языке и минимальном числе базовых концепций строятся очень гибкие и надежные системы, в том числе оригинальные пользовательские среды и операционные системы). Однако сама идея ООП — рассматривать предметную область как совокупность взаимодействующих объектов и отражать эти объекты в абстракциях языка — могла быть и была применена в отрыве от объектного языка программирования. Объектная ориентация де-факто присутствовала в структурных модульных языках с мощными системами типов данных (Модула, Ада). Тип данных — это и есть эквивалент класса, а экземпляр типа данных — объекта. Инкапсуляция реализована через концепцию модуля (пакета). Поэтому введение в полноценный структурный модульный язык дополнительных концепций «класс» и «объект» никакого смысла не имеет, но злостно нарушает сформулированный выше принцип базисности, т.к. получаем в языке два равноправных понятия (структура и объект) с идентичной смысловой и прагматической нагрузкой. Единственное, что требовалось ввести — это возможность расширения типов данных, построения из них иерархий на основе отношения «род-вид» — то, что в чистом ООП называется наследованием. Возможность расширения типа и была введена Виртом в Оберон в 1989 году, придав потомку Модулы качественно новые возможности. Точно таким же образом в 1995 году был выработан новый стандарт для Ады, включающий ООП (кстати, здесь уместно будет заметить, что основной язык Минобороны США Ada был разработан во Франции Жаном Ишбиа и является ответвлением семейства Алгола и Паскаля, в полной мере следующим традициям европейской школы программирования. Долгий промежуток времени между Ada-83 и Ada-95 — хороший пример того, как ресурс, заложенный в мощный структурный и модульный язык, позволил успешно обходиться без полной реализации ООП). Массовая американская ИТ-индустрия в популярных языках и средах (C++, Delphi, Visual Basic и множество более мелких) пошла другим путем, предпочтя «множить сущности без необходимости» в исключительно маркетинговых целях, увеличивая спрос на свой инструментарий за счет загромождения его свежими модными понятиями и оборачивания давно известных идей в новую блестящую обертку. Получилась ситуация, когда средний программист просто проскакивает мимо структурно-модульной методологии, а получая «в наследство» соответствующие конструкции языка, продублированные конструкциями ООП, не представляет, как их можно эффективно использовать.

4. Сочетание в Обероне модульности и ООП, в частности, перенос инкапсуляции на

уровень модуля, делает язык удобным для системного программирования с применением ООП. Как известно, системщики недолюбливают C++, предпочитая чистый C (Танненбаум, Торвальдс и другие разработчики ОС критически высказывались о возможности применения C++ для написания ядра ОС). Одна из причин этого в том, что для системного программирования с применением ООП характерно наличие групп мелких классов, которым необходимо тесно взаимодействовать между собой без дополнительных накладных расходов. Это требует либо чрезмерно открытых интерфейсов классов, при которых инкапсуляция практически сводится на нет, либо использования для связанных групп классов каких-либо средств обхода инкапсуляции. В Обероне эта проблема отсутсвует в принципе, благодаря реализации инкапсуляции на основе модулей. «Класс» не является самостоятельной сущностью с независимым поведением, это не элемент архитектуры, а тип данных. Поведение и взаимодействие для некоторой группы типов обеспечивает тот модуль, в котором они определены.

5. Автоматический контроль спецификаций, ивариантов. В Оберон-парадигме работает принцип: везде, где компилятор или среда выполнения (без весомого ущерба для быстродействия) может проконтролировать соответствие кода и поведения спецификациям, это должно быть сделано. Идеология Оберона предполагает программирование без использования пошаговой отладки. В большинстве сред пошагового отладчика нет как такового. Вместо этого обычно применяется дамп-отладчик, позволяющий при нарушении защиты или какого-либо предусловия удобно просмотреть состояние стека процедур и модулей. Система поддержки информации о типах и метаинформация модулей позволяет легко это сделать.

6. Стремление использовать единый язык для проектирования, спецификации и реализации — единого подхода здесь пока нет, но тенденции в Оберонах именно таковы. В частности, Компонентный Паскаль появился как промышленная версия Оберона-2, дополненная некоторыми средствами, позволяющими строже выражать в коде проектные решения. Другой пример — в синтаксис экспериментального языка Zonnon были включены конструкции, позволяющие специфицировать протокол взаимодействия объектов в расширенной нотации Бэкуса-Наура. В индустрии распространен подход, при котором для проектирования и реализации используются разные нотации. Однако преимущество такого подхода спорно и является скорее следствием слабой пригодности распространенных языков программирования для наглядного выражения проектных решений. В частности, графическое представление, принятое в нотации UML, является чрезмерно громоздким и неудобным по сравнению с удачной символической записью (показательно, что инженеры-электронщики отказались от графического представления схем, перейдя на символьные языки, подобные языкам программирования).

7. Использование в качестве основы для пользовательских интерфейсов и графики концепции составных документов. Документ в широком смысле — это структурированные данные и способ, которым они отображаются для пользователя. Документ состоит из иерархии графических объектов, в которой ключевую роль играют объекты-контейнеры, предназначенные для того, чтобы размещать в себе любые другие объекты. Документная модель позволяет объединять в рамках одного документа и строить дерево из объектов, ничего не знающих друг о друге,

обеспечивать их совместное отображение и обработку команд пользователя. Эффективно управлять таким деревом объектов через их ОО-интерфейсы затруднительно, т.к. для этого среде потребовалось бы уметь работать с бесконечным множеством таких интерфейсов. На помощь приходит парадигма процедурной шины. Каждый визуальный объект подключен к единой шине сообщений. По шине передаются сообщения — расширяемые записи. Поскольку Оберон поддерживает строгую динамическую проверку типов, такой механизм прост и надежен. Каждый объект решает самостоятельно, обработать ли сообщение или передать вниз по дереву объектов. Если некоторому объекту потребуется поддержка дополнительных возможностей, не потребуется менять его ОО-интерфейс (что привело бы к перекомпиляции всех клиентских модулей), достаточно будет ввести новый тип сообщений и научить объект их обрабатывать. Такой механизм применяется отнюдь не только для управления составными документами, однако здесь его преимущества видны яснее всего. В Оберонах (в частности, в BlackBox) также нашел себе место подход Модель-Отображение-Диспетчер (Model-View-Controller, MVC, впервые предложен в Smalltalk), при котором реализация визуальных объектов разбивается на три части: модель — реализует хранение и работу с данными, отображение — реализует визуальный показ данных, контроллер — реализует интерактивное поведение объекта.

8. Широкое использование метапрограммирования. Метапрограммирование позволяет работать с модулями, типами данных и другими сущностями языка программирования, которые были неизвестны на этапе компиляции. В сочетании с динамической модульностью это одно из оснований, на которых строятся расширяемые системы. Во время выполнения приложение может проанализировать структуру любого нового модуля, структуру каждого типа, создать экземпляры типа и т.д. То, что в полноценном виде появилось в других компилируемых языках, фактически только в .NET, существовало в Оберонах уже с начала 90-х (про то, что в динамических языках это существовало еще с 70-х, уже было сказано). Метапрограммирование позволяет, например, организовать удобную работу с внешней СУБД без использования языка препроцессора (например, Ebbedded SQL). Библиотеки позволяют скрыть всю работу с БД за типами данных языка. Точно так же метапрограммирование оказывается полезным для реализации параллельных вычислений или обработки исключений на уровне библиотек, а не языка.

9. Технология промежуточного представления М. Франца. В 1993 году в диссертации Михаэля Франца был предложен «метод представления программ в виде абстрактном и независящим от возможной целевой архитектуры, который позволяет получить представление файлов в два раза более компактное, нежели машинный код для CISC-процессора. Этот метод лег в основу реализованной автором системы, в которой процесс кодогенерации откладывается до этапа загрузки. В этот момент загрузчик с динамической кодогенерацией и порождает машинный (native) код». Эта технология Just-In-Time-компиляции впервые была реализована для ОС Оберон. Только потом эта идея использовалась в JIT-реализациях Java и .NET. Однако важным преимуществом представления Франца перед принятым в Java байт-кодом является сохранение в нем значительной части высокоуровневой семантической информации, что позволяет генерировать оптимальный машинный

код. (Байт-код Java ориентирован на интерпретацию на стековой машине и эффективно переведен в машинный код для регистровых CISC-процессоров быть не может. В этом причина того, что JIT-реализации Java сильно уступают по быстродействию классическим компиляторам). Казалось бы, работа Франца, как и многие другие, никак не привязана к Оберонам и могла бы быть проделана на любом другом языке. Но здесь дело не в языке, а в самом стиле мышления, характерном для швейцарской школы программирования. Собственно говоря, основной ее принцип можно сформулировать словами Франца из «благодарностей» в его диссертации, в которых он говорит о Н. Вирте: «Я искренне преклоняюсь перед его редким даром мгновенно выделять существенную сторону явлений и перед тем мужеством, с которым он отказывается от всего лишнего и наносного, даже когда другие считают отброшенные им вещи просто незаменимыми».

Суммируя вышесказанное, можно подчеркнуть два момента, могущие в первую очередь заинтересовать промышленного программиста.

Во-первых, Оберон-системы предлагают динамическую объектную модель (т.е. механизм загрузки и компоновки двоичных модулей) для компонентного программирования. В этом смысле Обероны являются аналогом (а исторически — первообразом) таких компонентных моделей, как CORBA, COM, .NET (более сложных, но и более громоздких). Однако Оберон-системы — это моноязыковые объектные модели (хотя есть и двуязыковые, ОС JBed — Компонентный Паскаль + Java). В общем случае языковонезависимые компонентные модели имеют больше возможностей, однако в очень многих случаях они оказываются излишне громоздкими. Если есть необходимость поддержки компонентов в рамках одной программной системы (для легкого обновления, сборки различных версий, установки расширений, «плагинов» и т.п.), то моноязыковая Оберон-модель будет великолепным выбором. К тому же стоит заметить, что среда BlackBox является одной из лучших сред разработки COM-приложений и заслужила в этом качестве медаль международного конкурса CeBiT.

Во-вторых, среды выполнения Оберонов в плане гибкости и динамичности предоставляют те же возможности, какие характерны для интепретируемых систем, таких как, например, Smalltalk, Python и т.п. Это позволяет, кроме всего прочего, использовать Оберон не только как язык для разработки некоторой системы, но и как язык пользовательского программирования и конфигурирования, вместо разработки дополнительного интерпретируемого скриптового языка, как это обычно сейчас делается (например, 1С).

III. Практический и конкретно-языковый уровень.

1. Синтаксис Оберона появился в результате очищения и доведения до совершенства сначала Паскаля, потом Модулы. Лексика паскалевской линейки языков ориентирована на легкое восприятие и чтение исходников. Важнейшие принципы — для каждой цели в языке должна иметься по возможности одна конструкция, а конструкции разного назначения должны выглядеть по-разному. В Обероне нет никаких метасинтаксических средств, которые позволяли бы модифицировать и расширять языковые конструкции, и их введение противоречило бы идеологии языка. Мощность Оберона достигается не за счет гибкости на уровне языка, а

напротив — за счет его жесткости, базисности. Попробуем объяснить это на образном примере. Если язык программирования - это материал для изготовления программ, то большинство ЯП можно отнести к одной из двух групп: язык - «пластилин» и язык — «набор кубиков». В первом случае язык предоставляет метасинтаксические средства, позволяющие «вылепить» из него требуемый для решаемой задачи подъязык, на котором и будет записана программа. В наиболее чистом виде к «пластилину» можно отнести LISP и Forth. В той или иной степени таковыми являются также многие функциональные языки. Для их синтаксиса характерна простота и однородность, опора на одну ключевую конструкцию (в LISP - списки, в Forth - инверсная польская запись). Такие языки дают программисту большую свободу в творении собственных «вселенных», позволяют кратко и красиво записывать решение многих задач. Однако есть и обратная сторона - отсутсвие жесткости, «бесхребетность». Каждый фрагмент программы оказывается тесно связан с контекстом, с окружением, в котором «вылеплены» необходимые определения. Если же язык к тому же не является однородным, а обладает множеством «подъязыков» уже на уровне базового синтаксиса, как C++, то исходные коды могут становиться крайне сложны для понимания и изменения, любая опечатка может вести к цепи труднообнаруживаемых ошибок. Та свобода, которая полезна для экспериментов, оказывается препятствием к построению надежных систем с длительным жизненным циклом. Оберон (как и Delphi, Ada и все паскалевское семейство) построен на иной идеологии — язык как «набор кубиков», который необходим и достаточен для сборки программных систем. Каждый кубик является неделимой единицей, его нельзя разобрать, изменить форму и т.п. Из кубиков собираются крупные блоки — модули. Модуль становится более крупным кубиком, но таким же монолитным, жестким. Модуль в исходном коде — это не включаемый текстовый файл, и не размытое по нескольким исходникам пространство имен. Двоичный модуль — это не DLL и не сборка .NET. Модуль является самостоятельной обособленной единицей, проходящей через все стадии жизненного цикла ПО. На основе жестких элементов можно собирать сколько угодно большие и сложные системы. Кроме того, код может быть не только легко собран, но и разобран на отдельные блоки, которые используются повторно. В заключение надо сказать, что выбор между «пластилином» и «кубиками» зависит как от специфики решаемой задачи, так и от стиля мышления разработчика. Первый подход характерен для математиков, второй — для инженеров. Оберон поощряет инженерный стиль мышления, реализуя принципы, общие для конструирования любых технических систем, а не только программных.

2. Однозначная понимаемость кода в отрыве от контекста. Этот аспект непосредственно связан с предыдущим. В синтаксисе Оберона сделано все, чтобы обеспечить легкое чтение кода. В частности, любые нелокальные связи записываются явно, и каждый блок кода можно читать без необходимости держать в памяти все его окружение. Исходный код представляет собой легко читаемую принципиальную схему программной системы. Синтаксически это обеспечивается следующими составляющими:

   (a) Требование полной квалификации имен. Импорт модуля не открывает новое пространство имен. Любое обращение к сущности другого модуля сопровождается явным указанием либо его имени, либо назначенного псевдонима.

(b) Указание объекта для связанных процедур явным параметром, как в их заголовке, так и в коде. Внутри связанных процедур не открывается дополнительное пространство имен для полей объекта.

(c) Простая иерархическая структура пространств имен: глобальное текущего модуля, локальные процедур модуля, произвольная глубина вложенных процедур. Нигде в коде не могут пересекаться два пространства имен, если одно из них не является вложенным в другое. Никакие операторы внутри блоков кода не влияют на пространство имен.

3. Реализация полиморфизма. На крайне компактном Обероне полиморфизм, тем не менее, может быть обеспечен несколькими способами.

   (a) Оператор `WITH rec: T1 DO ... | rec: T2 DO ... | ... ELSE END`, реализующий различное поведение (и одновременно приведение типа) в зависимости от динамического типа переменной. Механизм тегов типов в Оберонах реализован так, что определение динамического типа переменной возможно всего за одно сравнение. Оператор `WITH` чаще всего применяется к параметрам процедур, делая эти процедуры полиморфными.

   (b) Поля записей процедурного типа. В Обероне-1 связывание процедур с типами делалось именно таким способом. Такой подход потенциально мощнее обычных методов, поскольку позволяет не просто каждому подтипу, а каждому экземпляру типа иметь особое поведение. Инициализация процедурных полей записи называется инсталляцией связанных процедур. Для динамических записей ее может выполнять процедура-фабрика, для статических — специальная процедура инициализации. Однако ООП на основе процедурных полей сопряжено с неудобной записью при вызове: `object.Do(object,....)`. Если на месте object стоит некоторое выражение, то это становится ощутимым. Заметим, что подход, аналогичный процедурным полям, получил в C# название делегатов.

   (c) Начиная с Оберона-2 в языке имеются связанные с типом процедуры — виртуальные методы, которые позволяют работать с ООП и полиморфизмом методов в стиле, привычном для C++ и Delphi-программистов стиле.

   (d) В Компонентном Паскале введен специальный псевдотип `ANYREC`, который позволяет передавать параметрами (полями) экземпляры любых типов, с последующим применением `WITH`. Но особенное преимущество ANYREC дает в сочетании с применением метапрограммирования, что позволяет создавать процедуры, которые работают с данными абсолютно любых типов, вне зависимости от того, были эти типы известны на этапе компиляции или нет. Это позволяет легко строить универсальные библиотеки. Например, при работе с СУБД в BlackBox библиотека Sql выполняет чтение кортежа БД в любую переданную запись, если ее поля имеют подходящие типы.

4. Родовые интерфейсы сообщений (generic message buses). Про них уже упоминалось в п. II.7. «Родовые интерфейсы сообщений — еще одна интересная конструкция Oberon'a. При работе с такими интерфейсами проектировщикам программного обеспечения необязательно раз и навсегда закреплять за данным классом определенный набор сообщений. Родовые интерфейсы сообщений позволяют объектам

обрабатывать «неизвестные» сообщения или, в крайнем случае, принимать такие сообщения и передавать их потомкам дальше по цепочке. А это важно для таких, например, сообщений, которые передаются в пространство отображения. Концепция типа допускает к тому же дальнейшее расширение протокола сообщений». (Ю. Гуткнехт «Oberon In Education»).

5. В Компонентном Паскале были введены специальные модификаторы, позволяющие компилятору автоматически контролировать правильность построения иерархий типов. Таким образом, контроль спецификаций выполняется не только «в малом», но и в некоторых аспектах архитектуры. Разработчик модуля явно специфицирует, какие типы и связанные процедуры являются расширяемыми, а какие — нет; может ли экземпляр типа создаваться вне модуля; процедуры могут быть экспортированы «только для реализации», то есть, определяются в клиентском модуле, но вызываться могут лишь из модуля-владельца. Процедуры, которые вводятся для типа впервые, помечаются как `NEW`, что исключает возможность случайного перекрытия имен при добавлении новой процедуры для базового типа. Супервызовы (вызовы к перегруженной процедуре базового типа) использовать не принято, они могут быть в дальшейшем исключены из языка. Вместо этого используется политика определения в базовом типе пустой процедуры `SomeProc2` (с экспортом «только для реализации») и ее вызова из основной процедуры `SomeProc`. В расширенных типах `SomeProc2` определяется с желаемым дополнительным поведением, а если требуется - то и `SomeProc3` для дальшейшего расширения. При этом расширенная процедура не может «забыть» вызвать базовую процедуру, а модули-клиенты не могут вызывать отдельно процедуры `SomeProc2`, `SomeProc3` ..., поскольку они экспортированы «только для реализации».

6. Отделение интерфейса от реализации. В Оберонах нет конструкторов, поэтому всегда используются фабрики типа. Если тип требует какой-либо инициализации, то его экземпляр может быть порожден либо непосредственно своим модулем, либо какой-либо переменной-фабрикой. В ООП в общепринятом стиле основное назначение конструктора — доступ к закрытым полям еще несуществующего экземпляра типа. Как уже упоминалось, эта проблема, как и многие другие, решена в Обероне за счет переноса инкапсуляции на уровень модуля. Общепринятым подходом является полное разделение интерфейса и реализации, когда клиент работает только с абстрактными типами, экземпляры которых создаются фабрикой модуля. Такой подход часто применяется и в других языках, однако в полной мере его мощь проявляется в совокупности с динамической модульностью. Статическая зависимость со стороны интерфейсных модулей к модулям реализации может отсутствовать в принципе, модули реализации подгружаются динамически и инсталлируют в интерфейсные модули свои фабрики. Реализация может быть заменена «на лету» прямо во время выполнения. Это позволяет также клиентским модулям быть переносимыми в машинном коде между различными операционными системами (если они работают на одной аппаратной платформе). Достаточно установить модули с платформенно-зависимой реализацией для конкретной платформы.

7. Среди конкретных приемов можно упомянуть также паттерн Носитель-Курьер-Проектор (Carrier-Rider-Mapper, CRM), которая служит для организации работы

с какими-либо структурированными данными. Например, носителем может выступать модель из некоторой триады MVC. Однако интерфейс носителя не содержит процедур для непосредственной работы с данными, носитель — это «вещь в себе». Для работы с данными у носителя запрашивается курьер — это тот канал, через который производится чтение/запись данных. Курьеры бывают двух видов — считыватели и записыватели. (Наиболее близкая к курьеру концепция — поток ввода-вывода. Однако есть отличие — курьер не является самостоятельной сущностью, он привязан к носителю, реализован в одном с ним модуле, что и дает возможность процедурам курьера работать с носителем в обход инкапсуляции, через внутренние закрытые интерфейсы). С одним и тем же носителем может одновременно работать несколько курьеров, каждый из которых находится на своей позиции относительно данных. И, наконец, третья сущность — проектор — служит для создания дополнительного интерфейса ввода/вывода более высокого уровня. Проекторы не имеет прямого доступа к носителю, они могут быть реализованы независимо в отдельных модулях. Проектор переводит запросы высокого уровня в более простые запросы к курьеру.

IV. Исторический уровень.

История Оберонов очень подробно освещалась в русскоязычных публикациях (многие из них можно посмотреть на `http://oberon2005.ru/`). Найти ссылки на русские ресурсы и англоязычные первоисточники можно в соответсвующем разделе «ссылки» сайта `http://blackbox.metasystems.ru/`. По англоязычным ссылкам легко найти информацию об операционных системах, реализованных в рамках Оберон-направления (классический Оберон: OS Oberon, Oberon System 3, ETH Oberon, Oberon V4; на основе активных объектов в Active Oberon: OS BlueBottle; 3 промышленные ОС жесткого реального времени: на Оберон-2 — XOberon, XO/2, на Компонентном Паскале — JBed). Все эти системы разрабатывались небольшими коллективами от 2 до 10 человек. В целом опыт швейцарской школы программирования можно охарактеризовать как опыт разработки крупных программных систем небольшими командами. Этот опыт может быть изучен и использован, даже в отрыве от Оберона.

Однако естественное и эффективное использование описанных подходов достигается именно на базе языка Оберон. Его применение, как было показано выше, оказывается эффективным на всех стадиях разработки, позволяя максимально сократить переход от проектирования к кодированию, а затем — к очень легкому этапу выявления и исправления ошибок. Дальнейший эффект будет сказываться в процессе сопровождения и развития компонентной системы, обеспечивая ей долгую жизнь. Если же по каким-либо причинам разработка должна вестись на другом языке, Оберон-среды, подобные BlackBox, могут оказать большую помощь в апробации новых идей и быстром создании макета системы.