УДК 004.451

Назаров С.В.,
доктор технических наук, профессор, ЗАО «Московский научно-исследовательский телевизионный институт, Москва, Россия, s_nazarov@mail.ru, mniti.ru

ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ

Аннотация: в статье исследуются три технологии проектирования программных си-стем: структурное проектирование и программирование, объектно-ориентированное проектирование и программирование и компонентное проектирование. Каждая из этих технологий предоставляет определенные возможности в части декомпозиции архитектуры проектируемой программной системы. В данной работе рассматриваются достоинства и недостатки каждой технологии и целесообразность их использования при проектировании архитектуры программной системы и последующей ее реализации.

Ключевые слова: технология проектирования программных систем, программные средства, трансляция, структурное проектирование программных средств, объектно-ориентированное проектирование программных средств, компонентное проектирование программных средств.

S.V. Nazarov,
Doctor of Technical Sciences, Professor, CJSC Moscow Research Television Institute, Moscow, Russia, s_nazarov@mail.ru, mni-ti.ru

TECHNOLOGY OF DESIGN PROGRAM SYSTEMS

Abstract: *The article explores three technologies for designing software systems: structural design and programming, object-oriented design and programming, and component design. Each of these technologies provides certain opportunities in terms of the decomposition of the architecture of the projected software system. This paper discusses the advantages and disadvantages of each technology and the feasibility of using them when designing the architecture of a software system and its subsequent implementation.*

Keywords: *software design technology, software, translation, structural software design, object-oriented software design, component software design.*

Введение

До сих пор продолжаются дискуссии о лучшей технологии и методах проектирования и программирования программных систем. Основных претендентов – три: структурное проектирование и программирование и связанный с ним модульно-интерфейсный подход,объектно-ориентированное проектирование и программирование и компонентное проектирование. Каждая из трех технологий, обсуждаемых в этой статье, предоставляет определенные возможности в части декомпозиции архитектуры проектируемой программной системы. В данной работе рассматриваются достоинства и недостатки каждой технологии и целесообразность их использования при проектировании архитектуры программной системы и последующей ее реализации. Выявляются положительные и отрицательные стороны каждой технологии и устанавливаются возможности сочетания полезных свойств этих технологии при проектировании архитектуры программной системы.

1. Общие принципы проектирования архитектуры ПС

В статье [1] рассмотрены архитектурные представления программных систем (ПС) инфокоммуникационных предприятий. Отмечено, что процесс создания архитектуры программной системы предполагает разработку системы с конкретными свойствами и функциональностью, причем каждое из этих свойств имеет заданный приоритет. Начиная с общей структуры, которая поддерживает всю требуемую функциональность, архитектор проводит декомпозицию функциональности и распределяет ее между элементами (модулями, объектами, компонентами и т.п.) структуры системы.

В целом сущность разработки ПС можно описать как ряд процессов перевода информации из одного представления в другое, начинающихся с постановки задачи и заканчивающихся набором подробных инструкций, управляющих компьютером при решении задачи. Создание ПС в этом случае представляется последовательной совокупностью процессов трансляции, т.е. перевода исходной задачи в различные промежуточные решения, пока, наконец, не будет получен подробный набор команд машинного языка. Такую последовательность можно представить схемой или макромоделью перевода следующего вида [2]:

$$< Z, L_z, P_z > \rightarrow^{PR_1} < TR_s, L_{tr} > \rightarrow^{PR_2} < C_s, L_c > \rightarrow^{PR_3} < A_s, L_A > \rightarrow \ldots^{PR_N} \rightarrow < S, P_s, L_M >.$$

Здесь приняты следующие обозначения:

Z –постановка задачи (возможно техническое задание) на разработку ПС;

P_z – параметры (характеристика) задачи;

L_z – язык описания задачи;

PR_i – процедура трансляции на i-ом этапе перевода, i = 1, 2, …;

N – количество различных этапов разработки системы;

TR_s – требования, предъявляемые к программной системе;

$$TR_s = TR_f \cup TR_{nf};$$

TR_f –функциональные требования, предъявляемые к системе;

TR_{nf} – нефункциональные требования, предъявляемые к системе;

L_{tr} – язык описания требований;

C_s – цели программной системы (программного продукта);

L_c – язык описания целей;

A_s – архитектура программной системы;

L_A – язык описания архитектуры программной системы;

S – код разработанной программной системы;

P_s – параметры кода программной системы;

L_M – машинный язык.

Заметим, что на каждом этапе процесса создания ПС предполагается использование определенного языка описания (постановки) задачи. Степень формализации этого языка растет по мере продвиже-

ния от одного этапа к следующему. Если на начальном этапе постановка задачи на разработку ПС заказчик использует чаще всего естественный разговорный язык, то с каждым следующим этапом язык становится все более формализованным, и наконец, созданный программный продукт на машинном языке исключает его неоднозначную трактовку. Практическая реализация схемы (1) связана с возможными ошибками на каждом ее этапе. Сложность – основная причина ошибок перевода и одна из главных причин ненадежности программных систем. В общем случае сложность объекта является функцией взаимодействия между его составными частями. Сложность архитектуры ПС определяется связями между ее подсистемами (компонентами, модулями и т. п.[1]) системы. Сложность отдельного модуля – функция связи между его командами.

В борьбе со сложностью применимы концепции общей теории систем. Первая – независимость. В соответствии с этой концепцией для минимизации сложности необходимо усилить независимость компонентов системы. По существу, это означает такое разбиение системы, чтобы высокочастотная динамика (взаимодействие) системы была заключена в единых компонентах, а межкомпонентные взаимодействия представляли лишь низкочастотную динамику системы.

Вторая концепция – иерархическая структура и вытекающая из нее методология проектирования, основанная на понятии виртуальной системы. Иерархия позволяет стратифицировать систему по уровням понимания. Каждый уровень представляет собой совокупность структурных отношений между элементами нижних уровней. Концепция уровней позволяет понять систему, скрывая несущественные уровни детализации. Это, по сути, третья концепция декомпозиции – абстракция – отвлечение от несущественных подробностей с целью лучшего понимания стороны изучаемого явления. Декомпозиция наиболее эффективна тогда, когда осуществляется на базе абстракции.

И все же, существует ли наилучший метод проектирования программных систем? По словам Г.Буча [3], на этот вопрос нет однозначного ответа. По сути дела, это был бы ответ на вопрос: «Существует ли лучший способ декомпозиции сложной системы?»Лучшего способа пока никто не заявил, но достаточное количество различных методов декомпозиции и методов проектирования архитектуры программных систем известно и применяется на практике достаточно давно.

2. Структурно-функциональная декомпозиция и методы проектирования

2.1. Модульно-интерфейсный подход

В общем случае модульная программная система представляет собой

[1] *Далее любую часть ПС будем называть модулем или компонентом.*

древовидную структуру, в узлах которой размещаются программные модули, а направленные дуги показывают статическую подчиненность модулей. Если в тексте модуля имеется ссылка на другой модуль, то их на структурной схеме соединяет дуга, которая исходит из первого модуля и входит во второй модуль. При этом модульная структура программной системы, кроме структурной схемы, должна включать в себя совокупность спецификаций модулей, образующих эту систему.

Функции верхнего уровня обеспечиваются главным модулем. Он управляет выполнением нижестоящих функций, которым соответствуют подчиненные модули. При определении набора модулей, реализующих функции конкретного алгоритма, необходимо учитывать следующее:

1) модуль вызывается на выполнение вышестоящим по иерархии модулем и, закончив работу, возвращает ему управление;

2) принятие основных решений в алгоритме выносится на максимально высокий по иерархии уровень;

3) если в разных местах алгоритма используется одна и та же функция, то она оформляется в отдельный модуль, который будет вызываться по мере необходимости.

Структурное проектирование обычно подразумевает использование структур для разделения различных частей системы. Каждая часть при этом может проектироваться отдельно. Один из наиболее традиционных методов проектирования операционных систем в 60-е годы 20 века состоял в том, что проект каждого основного модуля системы выполнялся отдельно. Этот подход называется модульно-интерфейсным.

Анализируя существующие системы, можно заключить, например, что операционная система состоит из системы ввода-вывода, планировщика времени процессора, менеджера памяти, системы управления файлами и т. д. Каждый модуль выделяется и описывается, кроме того, определяется его интерфейс с другими модулями. Модули и их взаимные связи образуют абстракцию системы высокого уровня. После этого этапа для дальнейшего проектирования и реализации модули рассматриваются по отдельности. Такая же процедура может быть повторена в отношении какого-либо одного модуля системы, и последующую работу над модулем можно расчленить на индивидуальные задания по программированию. После того как все эти модули спроектированы и реализованы, они связываются воедино в соответствии с заранее определенными интерфейсами. Соблюдение правильного интерфейса представляет собой серьезную проблему в модульно-интерфейсном подходе.

Модули и их интерфейсы описываются весьма неформально и неточно. Поэтому, хотя отдельные модули и пишутся в соответствии с заданными спецификациями, часто оказывается, что в них заложено неверное представление об операционном окружении. И когда модули связываются вместе, они не взаимодействуют должным образом. После того, как

модули получены в коде компьютера, разрешение возникших конфликтов может оказаться чрезвычайно трудным делом.

В принципе, если интерфейс полностью специфицирован, то в этом отношении не должно возникать трудностей. К сожалению, первоначальный общий проект и планирование редко доводят до такого уровня детализации, когда используется язык программирования. В результате важные решения, которые должны были быть приняты опытными разработчиками в начале работы над проектом, отодвигаются внутрь модулей. В конечном счете, программисты принимают решения, имеющие глобальные последствия, базируясь на очень ограниченной и узкой точке зрения на систему. Решения могут приниматься и чрезвычайно квалифицированными людьми, но не в тот момент, когда нужно.

Правильное определение и выделение модулей представляет собой трудную задачу (заметим, что при объектно-ориентированном проектировании возникает не менее сложная задача определения объектов и выделения классов). Тесно связанные между собой части системы должны входить в один и тот же модуль. Было предложено считать мерой связности двух частей системы число предположений, которое одна часть должна делать относительно другой части. Слишком много или слишком мало информации об окружении, в котором работают соседние модули, может иметь плохие последствия при проектировании модуля.

Модульно-интерфейсный подход – один из методов структурного проектирования. Спецификации модулей и их интерфейсов дают структурную основу для проектирования каждого модуля и системы в целом. Спецификация программного модуля состоит из функциональной спецификации, описывающей семантику функций, выполняемых этим модулем по каждому из его входов, и синтаксической спецификации его входов, позволяющей построить на используемом языке программирования синтаксически правильное обращение к модулю.

Существуют различные методы разработки модульной структуры программной системы, в зависимости от которых определяется порядок разработки структуры системы, программирования и отладки модулей, указанных в этой структуре. Обычно в литературе выделяют два основных метода: метод нисходящей разработки (метод сверху-вниз или top-down) и метод восходящей разработки (метод снизу-вверх или bottom–up).

По мнению В. Турского, известного специалиста в области модульного программирования[4], формулировка аналитический и синтетический подходы к проектированию программ является более правильной, поскольку действительная суть проблемы не столько в выборе направления при проектировании, сколько в определении того, что именно преобладает: факторизация (аналитические шаги) или композиция (синтетические шаги). Ни один из этих подходов в их чистом виде не является жизнеспособной методологией проектирования: чисто аналитическое

проектирование сравнимо с построением "пирамиды с плавающей в воздухе вершиной", а чисто синтетическое – с построением пирамиды" стоящей на голове". Реальная стратегия проектирования почти всегда представляет собой разумное сочетание этих двух подходов.

2.2. Метод восходящей разработки (снизу-вверх)

Этот метод применялся в 60-х годах 20 века при разработке операционных систем, в частности операционной системы THE [5]. Он получил название метода восходящей разработки, или синтетического, или снизу-вверх. Применяя этот метод, разработчик начинает с основного аппаратного оборудования. "Чистая" аппаратура представляет собой для пользователя довольно неудобную машину, чтобы решать на ней задачу. Поэтому разработчик на первом шаге добавляет к компьютеру слой программного обеспечения $ПО_1$. Этот слой программ вместе с нижележащей аппаратурой обеспечивает выполнение некоторого множества команд, определяющих новую виртуальную машинуVM_1(рис. 1). На следующем шаге выделяется другое нужное свойство, добавляется новый слой программного обеспечения и получается очередная виртуальная машина. Процесс продолжается до тех пор, пока не будет получена виртуальная машина с требуемыми пользователю свойствами.

Аналогично проводится процесс конструирования программных систем, который назван архитектурным подходом. Модульная структура системы формируется в процессе программирования модулей, начиная с самого низшего уровня, затем следующего и т. д. При этом модули реализуются в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. Главной целью архитектурного подхода является повышение уровня используемого языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяют типичные функции, каждую из которых можно использовать при решении разных задач в этой области.

Основные преимущества метода проектирования снизу-вверх вытекают из способа структурирования, ограничивающего построение системы. Между различными уровнями можно организовать четкий интерфейс. При этом, поскольку любой процесс может требовать обслуживания только от процесса на более низком уровне, практически исключаются непредвиденные ситуации. Тестовые процедуры, использующиеся в этом методе проектирования, могут быть исчерпывающими, так как каждый уровень можно проверить по отдельности и достаточно полно. Когда некоторый уровень проверен до конца, можно добавлять части следующего уровня, реализующего более сложные функции и продолжать проверку.

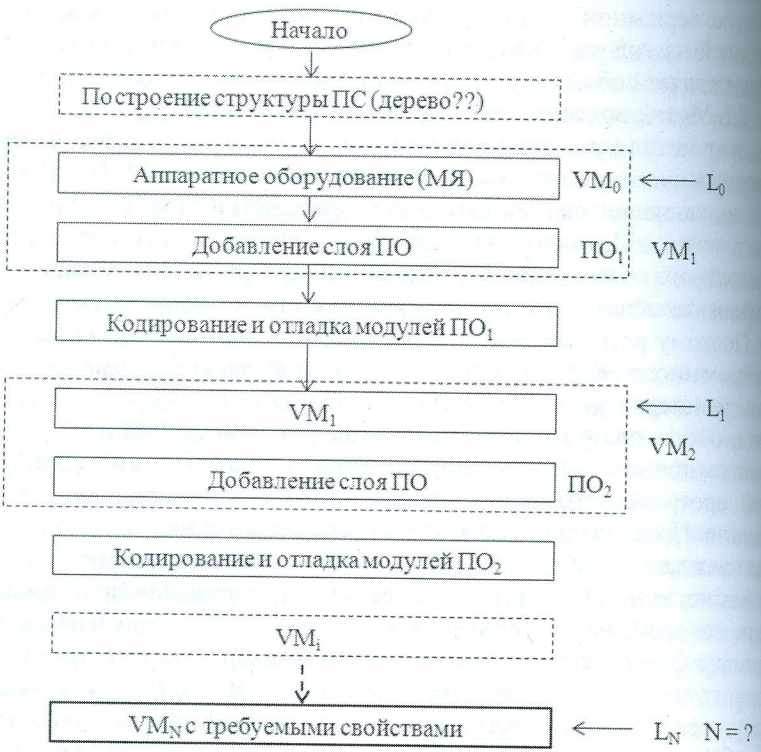

Рис. 1. Последовательность проектирования снизу-вверх

При восходящем программировании разработчик начинает с полного набора базовых средств, обеспечиваемых выбранным языковым уровнем (например, машинным языком МЯ). Все, что можно запрограммировать на этом языковом уровне, можно выразить в терминах таких средств. Для облегчения процесса программирования разработчик создает более высокие слои модулей таким образом, чтобы они облегчали использование доступных средств в форме, позволяющей абстрагироваться от обременительных деталей. Но на каждом следующем уровне (слое) разработчик должен быть уверен, что все средства, выразимые в терминах, установленных на этом слое понятий, доступны и что слой модулей представляет собой полное и логическое описание всей совокупности средств, обеспечиваемых базовым слоем.

Главная трудность в применении метода восходящей разработки заключается в выборе числа уровней и иерархическом упорядочении. Головной модуль проектируется и реализуется в последнюю очередь, это не дает возможности продемонстрировать его работу заказчику и про-

верить его соответствие спецификациям. Разработчики, использующие синтетический подход к проектированию программ, имеют значительные трудности на начальных этапах работы, поскольку они затрагивают самые базовые понятия и связаны с написанием текстов на машинном языке программирования. С другой стороны, нельзя их пропустить и сразу перейти к понятиям "среднего уровня". Описание представляет собой как бы двойную загадку: первое – как догадаться, какие понятия являются полезными для решения конкретной задачи; второе – как получить эти промежуточные понятия из базовых.

На первый взгляд порядок разработки снизу-вверх кажется вполне естественным. Каждый модуль при программировании выражается через уже запрограммированные, непосредственно подчиненные модули, а при тестировании используются уже отлаженные модули. Однако современная технология не рекомендует такой порядок разработки программной системы. Во-первых, для программирования какого-либо модуля можно обойтись без текстов используемых им модулей, достаточно чтобы используемый модуль был специфицирован. Во-вторых, каждая программа в какой-то степени подчиняется некоторым внутренним для нее, но глобальной для ее модулей информацией, что определяет ее концептуальную целостность. Такая информация формируется в процессе разработки ПС. При восходящей разработке для модулей нижних уровней такая информация еще не ясна в полном объеме. поэтому эти модули часто приходится перепрограммировать. В-третьих, при восходящем тестировании для каждого модуля (кроме головного) приходится создавать ведущую программу, которая должна подготовить для тестируемого модуля необходимое состояние информационной среды и произвести требуемое обращение к нему.

Это приводит к большому объему отладочного программирования и не дает гарантии, что тестирование модулей производилось именно в тех условиях, в которых они будут выполняться в рабочей программной системе. Заметим, что последовательные слои программной системы, спроектированной снизу-вверх, не обладают свойством быть решением, но обеспечивают весь спектр возможностей, предоставляемых базовым слоем, хотя и выраженных через более общие понятия. Необходимо еще раз отметить, что слои возникают в синтетическом проекте тогда, когда разработчик получает набор модулей, реализующих понятия, в терминах которых можно выразить любую программу, выразимую в базовых понятиях. Модули, в которых детали, если они не выразимы в данном слое, упрятываются, не меняя при этом смысла программы (выраженного в понятиях, доступных в рассматриваемом слое).

2.3. Метод нисходящей разработки (сверху-вниз)

Альтернативный метод проектирования ПС получил название анали-

тического, или нисходящего проектирования, или сверху-вниз (рис. 2). В этом случае разработчик исходит из желаемых свойств виртуальной машины пользователя и последовательно разрабатывает уточнения в направлении аппаратуры. Реализация этой виртуальной машины полностью не специфицируется. Неопределенные части проектируются в дальнейшем как компоненты (модули) системы. Эта работа продолжается до тех пор, пока система не определена настолько, что ее основные функции реализуются аппаратурой или модулями, непосредственно выполняемыми на аппаратуре.

В результате проектирования получается множество вложенных друг в друга компонентов. На каждом шаге проектирования у разработчика имеется некоторая абстракция функционального описания некоторого компонента (алгоритм), и он должен уточнить эту абстракцию, разбив ее на более мелкие и более подробно проработанные части. На начальном шаге, используя внешний проект (спецификации системы), формируется перечень всех функций (алгоритмов) системы. Затем определяются их подфункции. Далее каждая подфункция может расчленяться до тех пор, пока ее составные части не будут окончательно уточнены. Метод нисходящего проектирования, иногда называемый функциональной декомпозицией, основан на двух стратегиях: пошаговом уточнении (детализации), разработанном Е. Дейкстрой, и анализе сообщений, базирующемся на работах Йордана, Константайнаи Майерса[6]. Эти стратегии отличаются способами определения начальных спецификаций, методами, используемыми при разбиении задачи на части, и правилами записи.

Для проверки проекта в процессе его разработки можно с самого начала применять моделирование. Этот подход побуждает разработчика думать о том, какие функции должен выполнять некоторый компонент, а не как данный компонент будет их реализовать. Компоненты системы можно моделировать более детально, по мере того, как продвигается работа по проекту. Первоначальное представление какого-либо компонента может быть некоторым алгоритмом, который для данного входа вырабатывает с некоторой временной задержкой соответствующий выход, и таким образом моделирует работу данного компонента. По мере продвижения работ по проектированию данный алгоритм замещается последовательным набором обращений к следующему множеству проектируемых компонентов. На каждой стадии разработки проекта можно оценить, чтобы проверить, продолжает ли он соответствовать поставленным целям.

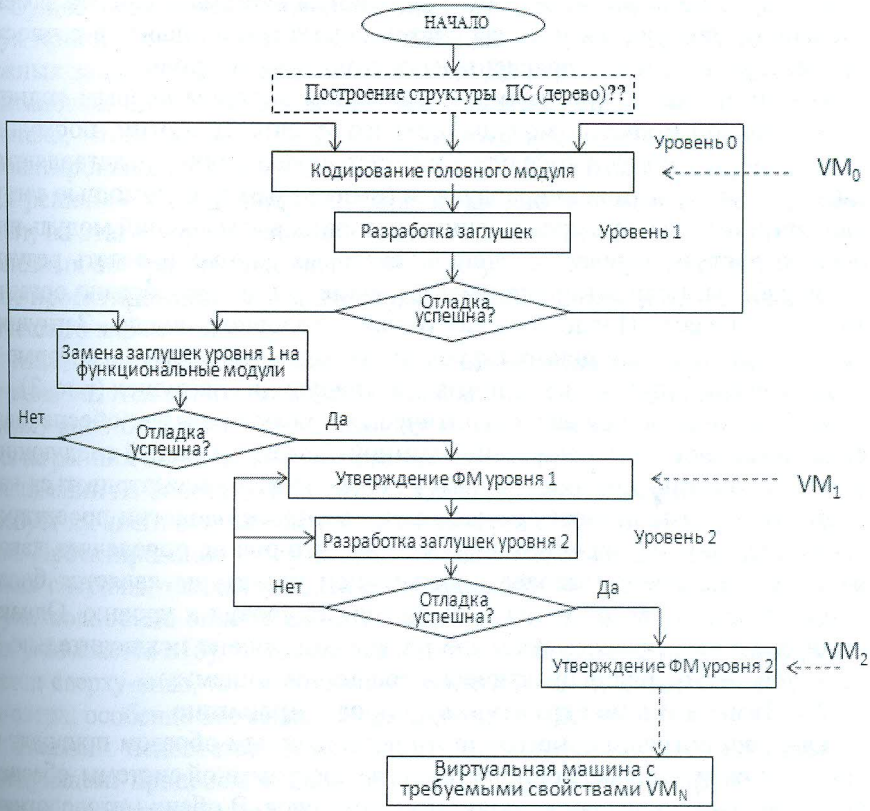

Рис. 2. Последовательность проектирования сверху-вниз

Каждая абстракция системы соответствует некоторой программе моделирования, которая построена из иерархии процедур. Эта программа определяет, как нужно действовать с теми переменными, которые влияют на состояние системы на этом уровне абстракции. Программа, назовем ее P, соответствующая некоторому уровню абстракции, управляется программой более высокого уровня абстракции, которую назовем Q. Программа Q принимает более глобальные решения, основываясь на значениях ее собственных переменных , которые на самом деле являются абстракциями переменных программы P. Изменения в переменных программы Q соответствуют изменениям в переменных программы P. Аналогично некоторые процедуры программы P делают запросы на выполнение работ к программам более низких уровней абстракции.

Когда система окончательно спроектирована, ее можно реализовать путем замены основных алгоритмов на самом низком уровне абстракции и средств, обеспечиваемых программой моделирования теми эле-

ментами, из которых должна быть построена система. Если эта замена сделана правильно, полученная система будет представлять в точности те средства, которые определены на самом высоком уровне.

При реализации программы нисходящим методом первым кодируется головной модуль – модуль верхнего уровня. При этом, поскольку этот вызывает модули соседнего низшего уровня, они представляются заглушками – фиктивными модулями (имитаторами). С помощью заглушек организуется непосредственный возврат в вызывающий модуль или осуществляться передача значений текстовых данных и печать результатов работы. Если заглушки имеют параметры, целесообразно организовать их печать. После того как отлажен головной модуль, Заглушки последовательно заменяются функциональными модулями, которые в свою очередь будут вызывающими для следующих заглушек (рис. 2).

В большинстве случаев аналитическое проектирование обеспечивает естественное расслоение программной системы: каждый следующий уровень понятий в расширяющемся дереве может рассматриваться как слой. Иногда выдвигаются возражения против методологии проектирования сверху-вниз, заключающиеся в том, что она не определяет, какое из двух направлений дерева программной структуры является более предпочтительным: от ветви к ветви или от уровня к уровню. Однако рекомендация выражать решение на каждом уровне исключительно в терминах этого уровня практически решает эту дилемму.

2.4. Замечания по структурному проектированию

Обе рассмотренные методологии естественным образом приводят к явно выраженной расслоенной структуре программной системы, обеспечивая равную доступность модулей одного слоя. В обеих методологиях слой представляет собой полное непротиворечивое множество модулей, даже если понятие полноты в каждом случае имеет различные оттенки. Сферы применения этих двух методов дополняют друг друга. Кратко это можно сказать так: при программировании решения отдельной задачи предпочтительнее аналитическое проектирование, а при создании программных систем для решения класса задач – синтетический метод. Поэтому синтетическое проектирование часто используется при написании операционных систем, а аналитическое – при написании специализированных программных систем.

Возможен другой путь использования методов структурного проектирования. Система может быть спроектирована методом сверху-вниз, а реализована методом снизу-вверх. В этом случае общий проект структуры программной системы выполняется методом сверху-вниз, а модули фактически объединяются в систему по методу снизу-вверх. При таком подходе налицо все преимущества реализации методом снизу-вверх и в тоже время не приносится в жертву возможность общего охвата системы, предоставляемая методом проектирования сверху-вниз.

Существует другая интерпретация взаимной дополняемости этих двух принципов проектирования, объясняющая, почему в большинстве важных задач используется оба подхода. Для трудных и больших задач обстановка, обеспечиваемая базовым уровнем, как правило, слишком удалена, чтобы к ней можно было прийти аналитическими методами проектирования. Поскольку при выборе абстрактных свойств, полезных для решения данной широкой задачи, можно учитывать предыдущий опыт, на этапах приведения системы к уровню этих предположительно полезных абстрактных понятий можно с успехом применять синтетическое проектирование, что значительно снижает объем разработок аналитического характера.

Между сторонниками методов проектирования снизу-вверх и сверху-вниз ведется много дискуссий. Многие полагают, что проектирование сверху-вниз больше соответствует идее, когда следует принимать решения на ранних стадиях проектирования и оставлять решения по детальной реализации на более поздние этапы. Однако проектирование сверху-вниз требует лучшего осмысливания на ранней стадии процесса проектирования. Проектирование снизу-вверх не требует вначале полного осмысливания системы, так как каждый слой программной системы добавляется к уже полностью определенному предыдущему слою. Вследствие этого, как указывается в [6], некоторые разработчики программ пропагандируют метод сверху-вниз, но на практике проектируют системы методом снизу-вверх; особенно это касается операционных систем.

Главный недостаток структурного проектирования заключается в следующем: процессы и данные существуют отдельно друг от друга, причем проектирование ведется от процессов к данным. Таким образом, помимо функциональной декомпозиции, существует также структура данных, находящаяся на втором плане.

3. Объектно-ориентированный поход (ООП)

В ООП основная категория объектной модели – класс – объединяет в себе на элементарном уровне как данные, так и операции, которые над ними выполняются. Именно с этой точки зрения изменения, связанные с переходом от структурного подхода к ООП, являются наиболее заметными. Разделение процессов и данных преодолено, однако остается проблема преодоления сложности системы, которая решается путем использования механизма пакетов. Существует две методики разработки объектно-ориентированных программных систем, которые можно условно называть “потребление” и “производство». Команды разработчиков могут работать совместно, используя либо одну из методик, либо обе. Но если руководитель проекта не уверен и не знает, хватит ли квалификации его сотрудников для того, чтобы использовать чужие объекты, или способны ли они разрабатывать собственные, то это может привести к серьезным проблемам.

Первая методика заключается в том, что команда разработчиков активно использует библиотеки и классы, разработанные другими. При такой методике разработчики понимают, что их опыт и квалификация пока недостаточны для разработки серьезных объектов. Вторая методика применяется тогда, когда команда хорошо осведомлена о процессе разработки шаблонов, рефакторинге и имеет удачный опыт разработки объектно-ориентированных проектов, включая разработку собственных классов. Обе техники приемлемы для использования, но важно знать, какая из них дает большие шансы добиться успеха.

Выявление классов, правильно описывающих предметную область проекта, является самой сложной задачей, которую предстоит решить разработчику. Выявление нужных классов намного сложнее процесса создания диаграмм, используемых в структурном подходе. Если не определены классы, точно описывающие задачу, то в конечном счете есть большая вероятность того, что созданные модели окажутся неработоспособными. Не каждый класс является классом предметной области. Массивы, наборы и классы графического интерфейса не являются классами предметной области. Классы предметной области – это сущности, описывающие задачу предметной области. Каждому классу предметной области соответствует объект предметной области.

Объект – это сущность, которая используется при выполнении некоторой функции или операции (преобразования, обработки, формирования и т.д.). Объекты могут иметь динамическую или статическую природу: динамические объекты используются в одном цикле воспроизводства, например, заказы на продукцию, счета на оплату, платежи; статические объекты используются во многих циклах воспроизводства, например, оборудование, персонал, запасы материалов. На концептуальном уровне построения модели предметной области уточняется состав классов объектов, определяются их атрибуты и взаимосвязи. Таким образом строится обобщенное представление структуры предметной области.

Рекомендации по объектно-ориентированному программированию предлагают выполнять поиск в предметной области существительных, а затем – привязку к ним глаголов. Существительные впоследствии переходят в классы, а глаголы – в методы. Это самый легкий способ, но он дает только 20 – 30 % всех классов, которые реально потребуется выявить. Если затем на этапе анализа обнаружились только классы и методы, выявленные сочетанием существительных и глаголов, то, вероятнее всего, в разработанных моделях будет наблюдаться нехватка классов и потребуется дополнительный углубленный анализ. Тем не менее, выявление существительных и глаголов предметной области является хорошим началом.

В дополнение к тем деталям по предметной области, которые могут сообщить эксперты заказчика, разработчику придется решить, как в системе представить эти данные в доступной форме для заказчика, и

в любом случае придется решать задачу того, как сохранять информацию, вводимую потенциальными пользователями программной системы. Для решения этих вопросов используются интерфейсные классы, классы-контроллеры и классы-сущности. Интерфейсный класс – это класс, предназначенный для связи элементов вне системы с элементами, входящими в состав системы. Поэтому такой класс часто называют граничным. Классы сущности представляют данные. Обычно сущности передают информацию, которая хранится в базе данных. Классы-контроллеры управляют другими классами.

Как правило, пользователи (заказчик) могут дать много полезной информации о классах-сущностях и помочь определить вид графического интерфейса пользователя, основываясь на том, как они выполняют задачи своей повседневной деятельности. Однако этого недостаточно, и разработчику нужно еще выявить классы-контроллеры и интерфейсные классы. Рекомендуется помнить тот факт, что эксперты по предметной области могут сообщить разработчику большое количество информации. Они могут рассказать многое о данных, которые им необходимо хранить, пояснить кое-какие детали о процессах, с помощью которых они получают эти данные, и немного о том, каким образом им хотелось бы вводить эти данные в компьютер.

Вторым важным моментом является то, что эксперты предметной области в рамках своей деятельности могут выполнять большое количество действий, которые будут просто непонятны постороннему наблюдателю. Это означает, что проектировщик, возможно, никогда и не пытался исследовать в целом то, что делает фирма-заказчик и как она это делает, и не пытался найти способ, как оптимизировать ее деятельность. В результате проектировщик может получить много информации от клиентов об их деятельности, которая, с его точки зрения, будет иметь слабое отношение к разрабатываемой системе. Но при этом эксперты по предметной области будут считать, что эта информация очень существенна.

После того, как разработчик выяснит у пользователей все о классах-сущностях, его задачей становится определение классов-контроллеров и интерфейсных классов. При разработке модели системы к сущностным классам добавляется стереотип “entity” и специальный символ класса-сущности, поддерживаемый во многих инструментах разработки, например, Rational SoftwareArchitect. Класс-контроллер является связующим кодом, который управляет другими классами и обеспечивает передачу данных между классами-сущностями и интерфейсными классами. Классы-контроллеры обозначаются при помощи стереотипа “control”, добавляемого к обычному классификатору, и специального символа. Интерфейсные или граничные классы моделируются, как показано на этом же рисунке и имеют стереотип “boundary”.

Заметим, что классы-сущности включают в себя логические сущно-

сти. Обычно их появление является результатом какого-либо смешанного запроса в базу данных. Выявление логических и простых сущностей является относительно простой задачей, так как теория реляционных баз данных легка для понимания. Классам-сущностям, определенным для отдельных таблиц и различных смешанных запросов, включающих обращения к нескольким таблицам, можно присваивать стереотип "table". Вообще процесс выявления классов при работе над проектом системы весьма нетривиален. Существует много рекомендаций по организации такой работы. Одна из них – использование CRC (Classresponsibilityandcollaborator) – карты функциональности и сотрудников класса.

Поскольку данные по сравнению с процессами являются более стабильной и относительно редко изменяющейся частью системы, отсюда следует главное достоинство ООП. Г. Буч сформулировал его следующим образом [3]: объектно-ориентированные системы более открыты и легче поддаются внесению изменений, поскольку их конструкция базируется на устойчивых формах. Это дает возможность системе развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.

Г. Буч отметил также ряд следующих преимуществ ООП:

– объектная декомпозиция дает возможность создавать программные системы меньшего размера путем использования общих механизмов, обеспечивающих необходимую экономию выразительных средств. Использование ООП существенно повышает уровень унификации разработки и пригодность для повторного использования не только ПО, но и проектов, что в конце концов ведет к сборочному созданию ПО;

– объектная декомпозиций уменьшает риск создания сложных систем ПО, так как она предполагает эволюционный путь развития системы на базе относительно небольших подсистем. Процесс интеграции системы растягивается на все время разработки, а не превращается в единовременное событие;

– объектная модель вполне естественна, поскольку в первую очередь ориентирована на человеческое восприятие мира, а не на компьютерную реализацию;

– объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования.

К недостаткам ООП относятся некоторое снижение производительности функционирования ПО (которое, однако, по мере роста производительности компьютеров становится все менее заметным) и высокие начальные затраты. Объектная декомпозиция существенно отличается от функциональной, поэтому переход на новую технологию связан как с преодолением психологических трудностей, так и дополнительными финансовыми затратами. При переходе от структурного

подхода к объектному, как при всякой смене технологии, необходимо вкладывать деньги в приобретение новых инструментальных средств. Здесь следует учесть расходы на обучение методу, инструментальным средствам и языку программирования. Для некоторых организаций эти обстоятельства могут стать серьезными препятствиями.

Следует отметить, что многие специалисты (программисты и ученые) высказывали отрицательные суждения по вопросу перспективности объектно-ориентированной технологии программирования. Поводом к таким высказываниям послужила публикация известной в мире программирования статьи "Почему объектно-ориентированное программирование провалилось?". Статья была написана доктором компьютерных наук Стэнфорда Ричардом Гэбриелом, старшим архитектором по разработке ПО сначала Sun, а потом и IBM, [7]. Он не скрывал своего скептического отношения к парадигме ООП. В 2002 году, по прошествии 2 лет после первоначальной публикации критической статьи, его пригласили выступить на ежегодной конференции OOPSLA (центральная конференция IT-специалистов по объектно-ориентированным языкам и методологиям разработки ПО) и изложить свои критические взгляды.

В качестве оппонента пригласили Г.Стила (GuySteele), разработчика языка Scheme, крупнейшего специалиста-теоретика по ООП, авторитет которого в американской академической среде непререкаем. В качестве "анти-объектника" дополнительно пригласили П.Грэма (PaulGraham), крупнейшего специалиста по Lisp, автора многочисленных книг и стандартизаций Lisp. В стан "объектников" пригласили Джеймса Ноубла (JamesNoble), автора одних из первых книг и работ по теории ООП. По свидетельству участников,конференция запомнилась им по тому уровню обсуждения, которое завязалось в этой публичной "интеллектуальной дуэли" фактически диаметрально разных школ программирования. Но факт остаётся фактом: сторона, представлявшая ООП, во время открытой дискуссии с противниками под смех зала даже запуталась в своих же концепциях.

П.Грэм утверждал, что половина всех концепций ООП являются скорее плохими, чем хорошими, в связи с чем он искренне сочувствует ООП-программистам, тогда как вторая половина от оставшихся концепций и вовсе не имеет никакого отношения к ООП, с которыми их почему-то постоянно ассоциируют. Например, он говорил [7]: "В восьмидесятых годах метод повторного использования каким-то неясным мне образом связали с объектно-ориентированным программированием, и сколь угодно многочисленные имеющиеся доказательства обратного, по-видимому, уже не избавят этот метод от клейма ООП. Хотя иногда объектно-ориентированный код годится для повторного использования, таким его делает вовсе не объектно-ориентированность, а программирование в стиле «снизу-вверх». Возьмём, например, библиотеки: их мож-

но подгружать и повторно использовать сколько угодно, потому что, по сути, они представляют собой отдельный язык. И при этом совсем неважно, написаны ли они в объектно-ориентированном стиле или нет".

Читателям будет интересно познакомиться с полным текстом статьи [7]. Однако только время покажет, кто в итоге окажется правым, а кто искренне заблуждался. Тем не менее, критика основных концепций и соответствующих языков ООП с выяснением, какой язык,по-настоящему объектно-ориентированный не прекращается. Признанный эксперт в объектных методологиях программирования и гибких методологиях, таких как agileiscrum, Дэвид Уэст, автор книг ObjectThinking и DesignThinking [8], утверждает, что идея объектов вообще никогда не была по-настоящему воплощена в каком-либо языке, даже в Smalltalk. По его мнению, понятие класса – не объектно-ориентированно. Классы не имеют никакого отношения к объектам. Они были лишь способом эффективно хранить код, позволяя вносить изменения в одном месте вместо многих сразу. Дэвид Уэст также считает, что наиболее действительно настоящий объектно-ориентированный язык – это Self (проект финансировался компанией Sun Microsystemsи в 1994 году был закрыт).

В нем нет ничего, кроме объектов, имеется только один класс, или один тип сущностей, и эта сущность – объекты. Что касается C++, то это не объектно-ориентированный язык, и не задумывался таким. Его автор Страструп писал, что он хотел сделать программистов на C дисциплинированнее. Так что C++ был задуман для того, чтобы программисты писали более хороший и более дисциплинированный код.

В заключение отметим, объектно-ориентированный подход не дает немедленной отдачи. Эффект от его применения начинает сказываться после разработки двух-трех проектов и накопления повторно используемых компонентов, отражающих типовые проектные решения в данной области. Переход организации на объектно-ориентированную технологию – это смена мировоззрения, а не просто изучение новых CASE-средств и языков программирования.

4. Компонентный подход

Компоненты представляют собой автономный код, который может быть повторно использован за счет его независимого развертывания. Компоненты не обязательно должны быть большими по объему кода, но в основном они больше по объему отдельного класса или группы слабо связанных классов. Компоненты имеют множество предоставляемых и требуемых интерфейсов и используются в больших сложных приложениях с десятками и сотнями классов предметной области. В сложных случаях разработки больших систем, например, при разработке корпоративной информационной системы потребуются диаграммы компонентов [9].

Как отмечается автором книги [10], существует два основных способа выявления компонентов. Первым является метод движения сверху

вниз, а вторым – движение снизу-вверх. По сути это методы проектирования архитектуры программной системы, которые характерны и для структурного проектирования с модульно-интерфейсным подходом. Если смотреть глубже – это не что иное, как метод декомпозиция сложной системы в первом случае методом сверху вниз, а во втором случае – синтез сложной системы на основе метода проектирования снизу-вверх. При этом образуется многослойная архитектура программной системы (см. п. 2.3).

В ряде монографий и статей рекомендуется определять архитектуру системы на основе компонентов методом сверху вниз. В этом случае в первую очередь определяются компоненты (крупные модули системы), а потом интерфейсы этих компонентов. Как только компоненты и их интерфейсы определены, можно разделить задачи реализации системы среди участников процесса разработки, поручив проектирование каждого компонента отдельным группам. С того момента, как каждый согласится с разработкой интерфейсов, проектировщики будут свободны в реализации внутренних частей компонента и могут осуществлять ее любым способом, который они выберут.

По мнению автора работы[10], такой поход может быть эффективным в том случае, если команда проектировщиков применяет хорошо продуманные компоненты с понятными и известными им интерфейсами. Однако правильно выявить все новые компоненты путем проектирования сверху вниз достаточно сложно. Кроме того, это достаточно сложный стиль программирования, связанный с тем, что системы, основанные на компонентах, имеют от трех до пяти вспомогательных интерфейсов и промежуточных классов для каждого класса предметной области. Поэтому системы, основанные на компонентах, могут быть достаточно трудоемкими, дорогими и рискованными.

При разработке компонентной системы методом снизу-вверх в первую очередь определяются классы предметной области, т.е. те, которые служат непосредственно для решения задач предметной области, а не вопросов архитектуры приложения. В результате основные усилия концентрируются, прежде всего, на решении поставленной задачи вместо описания сложной архитектуры. Применяя метод разработки снизу-вверх и выявляя классы предметной области, проектировщики имеют больший потенциал для решения поставленной задачи. Кроме того, всегда можно объединить классы предметной области в компоненты, если сложность проекта растет, некоторые группы классов можно развернуть и повторно использовать, включив их в состав компонентов.

Оба метода проектирования компонентных систем имеют право на использование. Для небольших приложений и приложений среднего размера, вероятно, не потребуются компоненты и в этих случаях можно воспользоваться методом снизу-вверх. Для приложений корпоративного

масштаба потребуется более мощная методология, и в этом случае метод проектировании сверху вниз может оказаться более подходящим.

Способы декомпозиции могут быть различными. При модульно-интерфейсном подходе на верхних уровнях осуществляется структурная декомпозиция, на средних и нижних – функциональная. При объектно-ориентированном подходе декомпозиция связана с правильным подходом к определению классов предметной области. Каждый класс должен согласоваться с принципами хорошего объектно-ориентированного проектирования. К принципам хорошего объектно-ориентированного проектирования, в частности, относятся принцип единственной ответственности (ПЕО – SingleResponsibilityPrinciple (SRP)), принцип открытости-закрытости (ПОЗ – OpenClosedPrinciple (OCP)) и принцип подстановки Лискова (ППЛ – LiskovSubstitutionPrinciple (LSP)) [11].

Принцип единственной ответственности говорит о том, что у такого элемента, как класс, должна быть лишь одна, определяемая им ответственность. Если же класс отвечает и за представление данных и доступ к ним, это характерный пример нарушения принципа ПЕО. Принцип открытости-закрытости заключается в том, что класс должен быть закрыт для модификации, но открыт для расширения. При изменении класса всегда существует риск что-то нарушить. Но если вместо модификации класс расширяется подклассом, такое изменение менее рискованно.

Принцип подстановки Лискова (ППЛ) лучше пояснить следующим примером. Допустим, что существует иерархия наследования для классов Person (Лицо) и Student (Студент). При использовании класса Person должна быть возможность использовать класс Student, поскольку он является подклассом Person. На первый взгляд, это всегда происходит автоматически, хотя и не совсем очевидно в отношении рефлексии – метода, позволяющего проверять программными средствами тип экземпляра объекта, считывать и устанавливать его свойства и поля, а также вызывать его методы, ничего не зная заранее об этом типе. Так, в методе, использующем рефлексию для обращения к классу Person, может и не предполагаться подкласс Student.

Проблема рефлексии имеет отношение к синтаксису. Мартин Р. [12] приводит в большей степени семантический пример класса Square (Квадрат), который относится к классу Rectangle (Прямоугольник). Но когда используется метод задания ширины квадрата SetWidth() для класса Square, это не имеет смысла – по крайней мере, для этого нужно еще вызвать (внутренним образом) метод задания высоты квадрата SetHeight(). Такое поведение отличается оттого, что требуется для класса Rectangle.

Безусловно, все эти принципы оказываются спорными в определенных контекстах. Ими следует руководствоваться лишь как рекомендациями, но не "истиной в последней инстанции". Например, применяя ПОЗ, нетрудно переусердствовать до такой степени, когда становится ясно, что для реа-

лизации метода лучше модифицировать класс, чем расширять его. И ввод метода в класс можно также рассматривать как расширение.

Заключение

Разработка архитектуры любой сложной системы связана с тем или иным подходом к ее декомпозиции. Очевидно, что в конкретном проекте сложной программной системы невозможно обойтись только одним способом декомпозиции. Можно начать декомпозицию каким-либо одним способом, а затем, используя полученные результаты, попытаться рассмотреть систему с другой точки зрения. Рассмотрение различных методов представления архитектуры сложных программных систем позволяет констатировать факт представления архитектуры программной системы в виде многослойной иерархической системы, которую удобно представлять системой вложенных виртуальных машин. Структурный подход по-прежнему сохраняет свою значимость и достаточно широко используется на практике. На примере языка UML хорошо видно, что его авторы заимствовали то рациональное, что можно было взять из структурного подхода: элементы функциональной декомпозиции в диаграммах вариантов использования, диаграммы состояний, диаграммы деятельности и др. Основой взаимосвязи между структурным и объектно-ориентированным подходами является общность ряда категорий и понятий обоих подходов (процесс и вариант использования, сущность и класс и др.). Эта взаимосвязь может проявляться в различных формах.

Объектно-ориентированный подход непросто использовать на начальных этапах проектирования системы в связи с неполнотой и сложностью выделения всех объектов, которые должны быть учтены в проекте и которые должны будут стать классами. Одним из возможных вариантов является использование структурного анализа и функциональной декомпозиции как основы для объектно-ориентированного проектирования. После выполнения структурного анализа можно различными способами приступить к определению классов и объектов. Так, если взять какую-либо отдельную диаграмму потоков данных, то кандидатами в классы могут быть элементы структур данных. Структурный подход к определению архитектуры программной системы на основе компонентной технологии напрямую связан со структурными методами проектирования сверху-вниз и снизу-вверх. Это основные способы выделения компонентов системы.

Критикуемый недостаток структурного проектирования заключается в том, что процессы и данные существуют отдельно друг от друга, причем проектирование ведется от процессов к данным. Таким образом, структура данных находится на втором плане. В этом отношении объектно-ориентированный подход имеет преимущество – возможность интеграции объектной и реляционной технологий, поскольку объектно-ориентированное проектирование имеет определенные точки со-

прикосновения с реляционным проектированием. Классы в объектной модели могут определенным образом соответствовать сущностям (которые кстати рассматриваются и в структурных подходах). Однако, такое соответствие имеет место только на ранней стадии разработки системы. В дальнейшем цели объектно-ориентированного проектирования и разработки реляционной БД расходятся. Преодоление данного разрыва возможно построением отображения между диаграммами классов и реляционной моделью. Концепция многоуровневой виртуальной машины в совокупности с пошаговой детализацией и объектно-ориентированной декомпозицией позволяют построить формальную методику разработки модульной архитектуры программной системы, которую предполагается рассмотреть в продолжение данной статьи.

Список использованной литературы

1. Назаров С.В., Вилкова Н.Н. Архитектурные представления программных систем инфокоммуникационных предприятий. Новосибирск, Изд. «Печатный дом-Новосибирск». Журнал «Инфосфера», № 71. С. 15 – 18.
2. Назаров С.В. Архитектура и проектирование программных систем: монография /С.В. Назаров. – 2-е изд., перераб. и доп. – М.: ИНФРА-М, 2016 – 374 с.
3. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ (Второе изд.). Rational Санта- Клара, Калифорния:перевод с английского под редакцией И. Романовского и Ф. Андреева. Спб. Изд. Невский Диалект. 2001. 359 с.
4. Турский В. Методология программирования: пер. с англ. – М.: Мир, 1981.- 264 с.
5. Цикритзис Д.,Бернстайн Ф. Операционные системы. Пер с англ. М.: Мир, 1977. – 336 с.
6. Зиглер К. Методы проектирования программных систем: Пер. с англ. – М.: Мир, 1985. – 328 с.
7. Савчук И. Почему объектно-ориентированное программирование провалилось? [Электронный ресурс]. Режим доступа: http://blogerator.ru/page/oop_why-objects-have-failed
8. Классы – это не объектно: интервью Егора Бугаенко с Дэвидом Уэстом. [Электронный ресурс]. Режим доступа: https://jug.ru/2016/09/bugayenko-west/
9. Кулямин В. В. Технологии программирования. Компонентный подход. [Электронный ресурс]. Режим доступа: http://lib.mdpu.org.ua/e-book/vstup/L/Jogolev.pdf
10. Киммел П. UML. UML-Основы визуального анализа и проектирования UML- Универсальный язык программирования / Пол Киммел; пер. с англ. Кедрова Е.А. – М.: НТ Пресс, 2008. – 272 с.
11. Нильссон Дж. Применение DDD и шаблонов проектирования. Проблемно-ориентированное проектирование приложений с примерами на C# и .NET. Издательство: Вильямс. 2008. – 560 с.
12. Мартин Р., Ньюкирк Дж., Косс Р. Быстрая разработка программ: принципы, примеры, практика. Вильямс, 2004. – 752 с.