УДК 004.4’2

Ю.А. Водолазский, Ю.О. Калиберда, А.И. Михалёв

ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ОГРАНИЧЕННЫХ КОНТЕКСТОВ И КОНТЕКСТНЫХ КАРТ

Аннотация. В работе исследуются отдельные стороны ограниченных контекстов и алгоритмы построения контекстных карт для внедрения ключевых решений при разработке программного продукта. Предлагается методика, позволяющая контролировать сложность архитектуры приложения на основе ограниченных контекстов и контекстных карт.
Ключевые слова: архитектура программного обеспечения, предметно-ориентированное проектирование, моделирование предметной области, шаблоны проектирования, методология проектирования, контекстные карты, синергия.

Введение. Существуют множество подходов для объектно-ориентированного моделирования программного обеспечения. Однако достаточно часто архитектура приложения, получающаяся в результате использования этих подходов, плохо поддаётся масштабированию при увеличении размера и сложности приложения. Контекстные карты, как техника общего назначения, являющаяся частью дизайн-инструментария из парадигмы предметно-ориентированного проектирования (DDD - Domain Driven Design) [1], помогает разработчикам справляться со сложностями создания программных продуктов. В отличие от других хорошо известных DDD шаблонов, создание контекстных карт применимо к любому сценарию разработки и представляет собою высокоуровневую абстракцию, которая помогает разработчикам принимать стратегически важные решения.

Кроме того, контекстная карта является превосходным инструментом, позволяющим визуализировать синергию ограниченных контекстов [2], составляющих архитектуру приложения.

Проектирование с учётом контекста. Предметно-ориентированное проектирование уделяет большое внимание поддер-

 ISSN 1562-9945

жанию концептуальной целостности модели разрабатываемого приложения. Это достигается за счет комбинации нескольких факторов:

– гибкий процесс, основанный на постоянной обратной связи от пользователей и экспертов в области разрабатываемого продукта;

– доступность экспертов предметной области и продуктивное сотрудничество с ними;

– единая разделяемая версия модели приложения (в приложении и тестовом коде), чётко сформулированная в терминах *всеобъемлющего языка* [3];

– открытая и прозрачная среда, которая способствует обучению и исследованию.

Всё это имеет решающее значение для создания благоприятной среды, в которой высококачественный дизайн привносит положительный эффект. При этом типичные элементы DDD, такие как сущности, значения и агрегаты создаются и добавляются к сложной доменной модели приложения по мере их выделения из предметной области. Как показано на рисунке 1, основное назначение всеобъемлющего языка в DDD - это обеспечение целостности модели. Использование одного набора терминов, с очень точным и однозначно определяемым смыслом, порождаемого в процессе обсуждений с экспертами предметной области и внедряемого на уровне кода, гарантирует, что все члены команды разделяют одинаковое понимание задачи и способы её реализации в программном продукте.

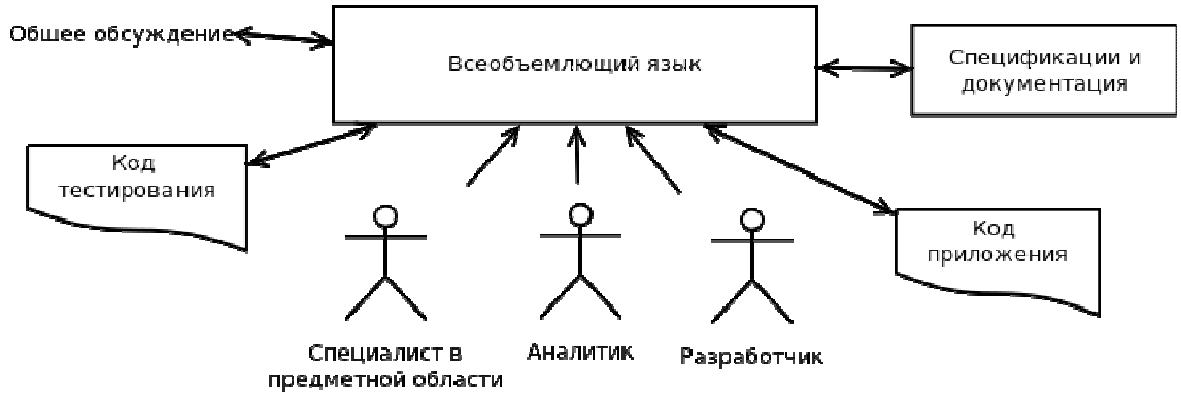

Рисунок 1 - Всеобъемлющий язык, как базисный язык описания модели предметной области

Введение в ограниченные контексты. В предметно ориентированном проектировании термин контекст формулируется как: "Набор

условий, при которых определяется значение какого-либо слова или выражения" [4].

Для каждой модели или семейства связанных моделей целесообразно выделять отдельные ограниченные контексты, в которых они существуют. Рассмотрим наиболее распространённые случаи, при которых введение ограниченных контекстов целесообразно и покажем, что это увеличивает масштабируемость проекта.

Случай 1: Один термин, разные значения

Наиболее распространённым является пример, в котором двусмысленность проявляется на уровне терминологии. Некоторые слова могут иметь разные значения в зависимости от контекста, в котором они используются.

Предположим, что мы разрабатываем приложение по управлению личными финансами с web-интерфейсом. Это приложение может использоваться для управления банковскими счетами, акциями, сбережениями, для контроля за бюджетом и расходами и т.д.

В нашем приложении термин *account* может означать разные концепции. В контексте банковских операций этот термин означает виртуальное "хранилище денег". В этом случае стоит ожидать, что соответствующий класс в коде программы должен иметь такие атрибуты, как баланс, номер счёта, и т.д. Но, в контексте web-приложения, термин *account* имеет совершенно другой смысл, связанный с аутентификацией и пользовательскими данными. Соответствующая модель, показанная на рисунке 2, будет полностью отличаться от предыдущей.

Рисунок 2 - Пример неоднозначности: термин учётная запись (*account*) может иметь разные значения в зависимости от контекста, в котором он используется

Данная проблема обычно решается путем разделения пространства имен или добавления префикса к имени класса (который может быть частью имени пакета). К сожалению, работа на уровне имён

 ISSN 1562-9945

классов - это не всегда лучшее решение: в области банковских услуг, термин *Banking Account* может обозначать другую концепцию. Однако самая большая ошибка в попытке использовать другой термин будет заключаться в том, что мы хотим изменить всеобъемлющий язык (словарь, описывающий систему), для разрешения архитектурных трудностей, которых изначально нет в предметной области. Поэтому необходимо сохранить одинаковые названия для моделей, но поместить их в разные контексты. Изобразим это на контекстной карте.

В случае, когда двусмысленность становится проблемой, необходимо иметь инструмент, с помощью которого команда разработчиков могла бы работать с двумя различными контекстами в пределах одного приложения. Неоднозначность всеобъемлющего языка - это наибольшая проблема, от которой необходимо избавляться. Лучший способ сделать это заключается в изображении структуры предметной области в терминах ограниченных контекстов на контекстной карте. Рисунок 3 показывает простую контекстную карту.

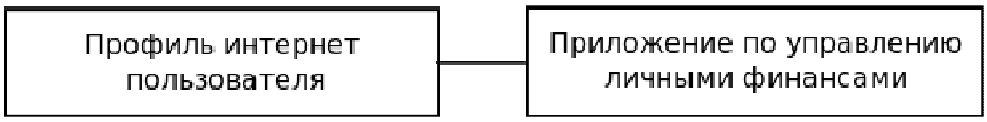

Рисунок 3 - Контекстная карта с двумя контекстами предметной области

В DDD [1], контекстная карта описывается как основной инструмент, использующийся для того, чтобы сделать границы контекстов явными. Нужно попытаться нарисовать границы контекстов на доске, желательно заполнив их соответствующими именами классов или терминами предметной области. К сожалению, такая диаграмма является не вполне корректной с точки зрения стандартов UML. В то же время, это есть рабочий инструмент, позволяющий внести определённость в нечёткую ситуацию.

Случай 2: Проблема: одна концепция, различное использование

Более запутанное разделение может возникнуть, когда концепция, лежащая в основе, одинаковая, но используется по-разному, что в конечном итоге приводит к разным моделям. Моделью банковской учётной записи может служить класс *Banking Account,* как показано на рисунке 4.

Как известно некоторые приложения по управлению личными финансами позволяют также управлять платежами, обычно сохраняя список получателей платежей (*Payee Registry*). При таком сценарии, получатель может быть связан с одним или более банковскими счётами, но в данном случае мы ничего не знаем, ни о "внутренностях" банковского счёта получателя, ни о том, можем ли мы совершать какие-либо операции с этим счетам. Ставится вопрос: “Стоит ли моделировать банковский счёт получателя, используя уже существующий класс *Banking Account*, разработанный выше?”

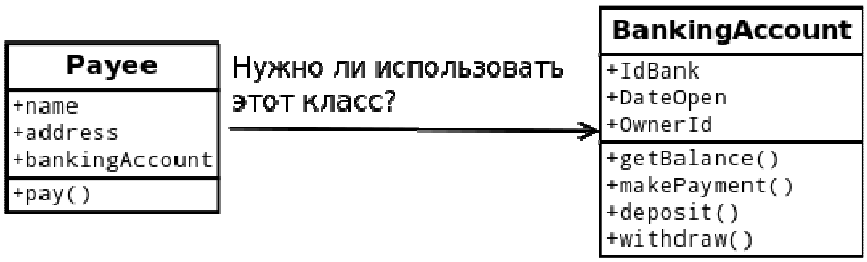

Рисунок 4 - Классы *Payee* и *Bank Account*

Итак, имеем, в конце концов, что это, одна и та же концепция. В реальном мире банковский счёт и счёт получателя могут быть в одном и том же банке. Однако, эта концепция ещё не совсем правильная: нельзя иметь возможность совершать какие-либо действия на банковском счёте получателя, или отслеживать изменение баланса этого счёта (хотя можно совершать эти операции на личном банковском счёте). Даже хуже: делая так, вероятно допускается концептуальная ошибка в архитектуре нашего приложения.

Опять возникает проблема с двумя различными контекстами в одном приложении. Поэтому правильным решением будет создать новый класс и поместить его в отдельный контекст. Класс *BankingAccount* может по-прежнему предоставлять возможность выполнять (или отслеживать) определённые операции, такие как взнос или обналичивание средств, в то время как отдельный класс *Payee Account*, содержащий некоторые общие данные с классом *Banking Account* (например, *account Number*), но в целом имеющий более простую модель данных и совершенно другое поведение (например, у нас нет доступа к балансу получателя платежа). Рисунок 5 иллюстрирует использование банковского счёта двумя различными способами в

 ISSN 1562-9945

приложении, несмотря на общее значение и единую концепцию, лежащую в основе этого понятия.

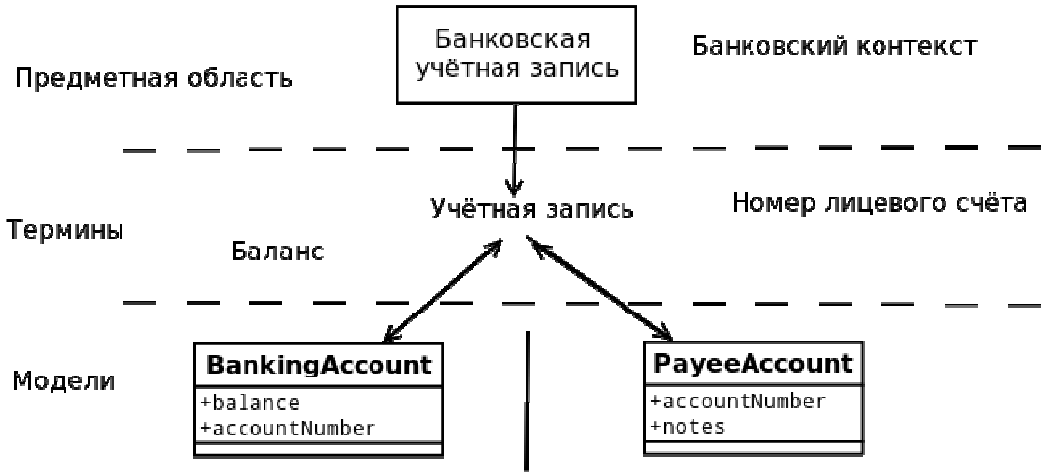

Рисунок 5 - Классы для банковского счёта и счёта получателя

Контекстная карта, иллюстрирующая вышеприведённый пример, может выглядеть, как показано на рисунке 6. В данном случае проведено разбиение контекста приложения по управлению личными финансами на два других: банковские операции и отслеживание расходов.

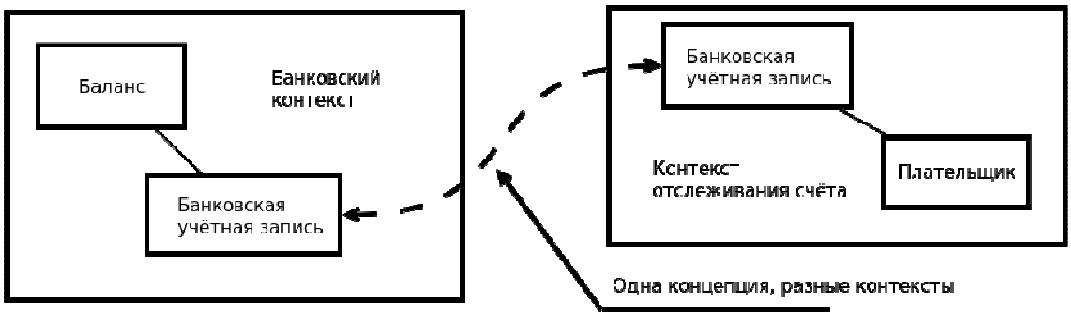

Рисунок 6 - Упрощённая контекстная карта: контуры вокруг частей модели предметной области показывают области, где концептуальная целостность сохраняется

Как результат, - концепция банковского счёта используется разными способами в разных частях приложения, подразумевая архитектуру, состоящую из двух моделей. Однако, эти две модели возможно достаточно близко пересекаются. Кроме сохранения концептуальной целостности модели в границах контекста, контекстная карта помогает сосредоточиться на том, что происходит между различными контекстами.

Случай 3: Взаимодействие с внешними системами

Вернёмся к рассмотрению приложения для управления личными финансами. Множество реализаций таких приложений позволяют совершать тот или иной обмен данных между интернет-сервисами финансовых учреждений. В некоторых случаях, банки предоставляют доступ реального времени к банковским сервисам, в других случаях они просто позволяют клиенту скачать выписки в стандартных форматах. Рисунок 7 показывает взаимодействие рассматриваемой системы с банковским сервисом.

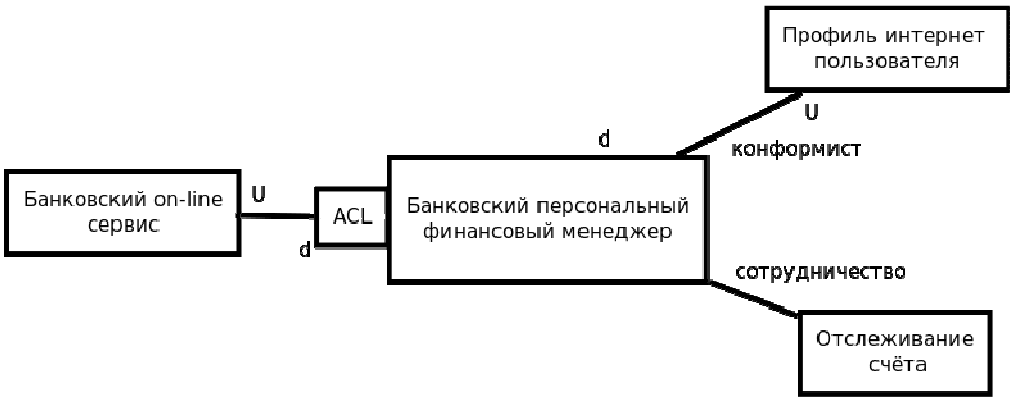

Рисунок 7 - Предохранительный уровень (ACL) на границе приложения по управлению финансами, предотвращающий прямое воздействие банковского сервиса внутрь ограниченного контекста. Показано направление взаимосвязи между контекстами (верхний(u)-нижний(d))

Даже если эти две модели разработаны, чтобы представлять одни и те же данные (по крайней мере, на данном уровне детализации), они будут развиваться по-разному с течением времени и будут служить разным целям. Поэтому разделение ограниченных контекстов необходимо. Случай 1 так же попадает в эту категорию, если профиль пользователя моделируется с использованием сторонней библиотеки.

В случае, когда приложение охватывает несколько контекстов, необходимо обеспечивать их синергию. При этом наиболее важный аспект разработки, о котором стоит позаботиться, - это обеспечение направления связей между контекстами. Известно, что DDD использует термины верхний (upstream) и нижний (downstream) контексты. Верхний контекст влияет на нижнего коллегу, тогда как обратное - неверно. Это влияет как на организацию кода (библиотеки, использующие другие библиотеки), так и на менее технические аспекты, та-

кие как порядок работы или последовательность исправления ошибок, или добавления изменений. В нашем примере существует внешняя система, которая очевидно не будет меняться по нашему запросу, в то время как приложение по управлению финансами может измениться быстро, в случае, если банковская система изменит свой API по каким-либо причинам. Рисунок 7 иллюстрирует также связи между двумя контекстами предметной области.

Возможно принятие запроса на изменение способа взаимодействия с внешней системой по внешнему запросу, но, скорее всего, потребуется некая защита от изменений, приходящих от вышестоящего контекста, так же как необходимо будет сохранить концептуальную целостность банковского контекста. DDD описывает несколько организационных шаблонов, которые помогают описывать и/или управлять способами взаимодействий (синергий) между контекстами. При этом, наиболее приемлемый - это шаблон предохранительный уровень (*Anti-Corruption Layer, ACL*) [5], который также показан на рисунке 7. Данный слой выполняет преобразования на уровне кода между контекстами или, что даже лучше, на внешней границе банковского контекста.

В свою очередь, вызывать другой контекст свойственно не только лишь внешним системам. Классическим примером изолированного контекста является устаревший модуль или компонент, который часто содержит модель, плохо поддающуюся изменениям.

В свете наличия других связей на контекстной карте поднимается вопрос: “Существует ли возможность определить их в терминах связующих шаблонов DDD?” Как только мы предполагаем, что разработка происходит в пределах одной команды, шаблоны становятся не так важны. Однако, если банковский контекст и контекст по управлению за расходами поддерживаются различными командами, то взаимодействие между этими командами должно быть хорошо налажено, т.к. они обе стремятся к одной и той же цели. В данном случае стоит использовать шаблон сотрудничество (*Partnership*). В этом случае, если реализация пользовательского профиля выполнена с использованием стороннего модуля, то вероятно будет использоваться этот модуль "как есть", подразумевая, что наш контекст является нижним, по отношению к контексту стороннего модуля. Для описа-

ния такого взаимодействия больше всего подходит шаблон конформист (*Conformist*) [5] (шаблоны также показаны на рисунке 7).

Случай 4: Проблема при расширении организации

На данный момент рассмотрен простой сценарий с одной командой разработчиков. Это позволило игнорировать стоимость взаимодействия, предполагая (возможно оптимистически), что каждый разработчик заботится о том, что происходит с моделью. В свою очередь, более сложный сценарий может включать воздействие следующих факторов:

– сложность предметной области, требующей наличия многих экспертов из этой области;

– сложность организации труда;

– растянутый во времени проект;

– очень большой проект (по количеству людей и времени);

– множество внешних составляющих или кода, доставшегося в наследство;

–большой размер команды или команда, состоящая из разработчиков разного уровня;

– распределённая или удалённая команда;

– человеческий фактор.

Каждый из этих факторов влияет на взаимодействие внутри команды и организационные процессы в целом, что в конечном итоге сказывается на разрабатываемом продукте.

Предположим, что наше приложение по управлению финансами выросло. Другая команда (команда Б) начала работать вместе с нами (командой А) над новым торговым модулем того же приложения. Команда Б может находится в другой комнате, стране или даже компании, которая пишет код для в сфере торговли. В примере ниже команда А работает над кодом совместно с командой Б, даже, если они стремятся работать над различными участками кода. Со временем команда Б может написать некий класс (рисунок 8), который будет реализовывать функциональность необходимую команде Б, которая уже реализована командой А.

 ISSN 1562-9945

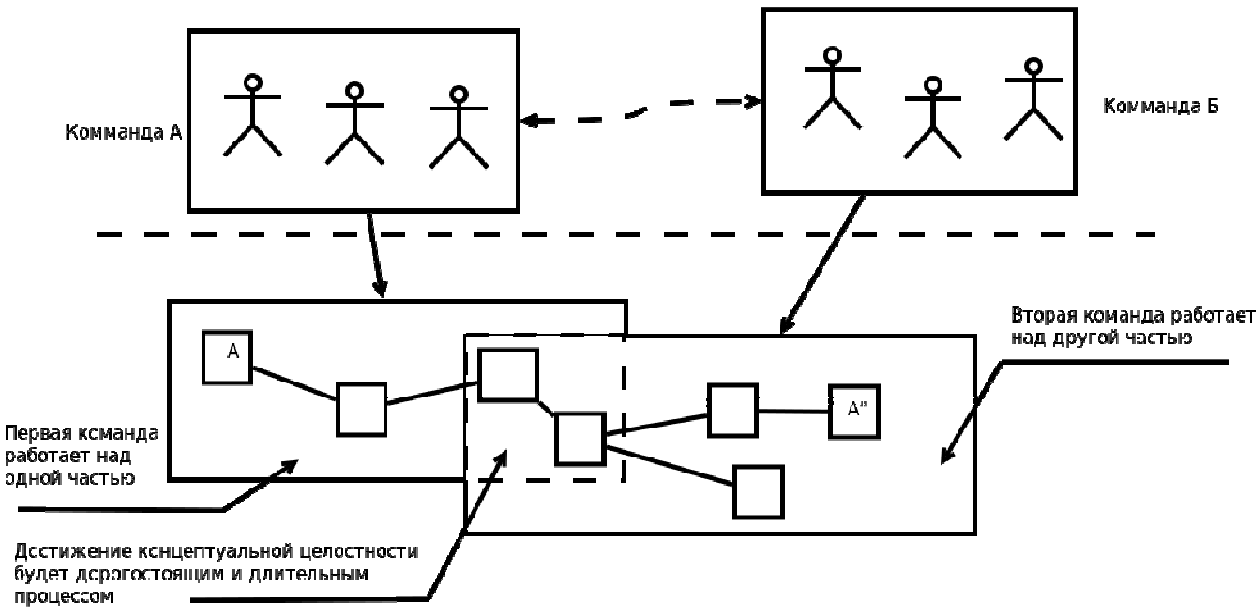

Рисунок 8 - Схема сценария, где разные команды работают над одним кодом, могут быть различные точки зрения на отдельные части модели. Физическое удаление команд будет влиять на качество обмена информацией между ними

Так появляется дублирование кода - корень зла. Для небольшого, чётко определённого, ограниченного контекста это утверждение справедливо. Однако по некоторым причинам это свойственно некоторым не тривиальным проектам Это является знаком, что возможно существует не совсем хорошо изолированный контекст. Иногда более эффективно иметь два изолированных контекста для структурирования модели предметной области, чем постоянно поддерживать целостный и непротиворечивый взгляд на модель среди участников двух команд.

Рисунок 9 - Модель торгового контекста, требующая дальнейшего исследования

Восклицательный знак на схеме означает, что там что-то не так: эти два контекста частично перекрываются и их отношения не ясны. Вероятно, - это одна из первых областей, в которой нужно искать решение: пытаться создать согласованные и устойчивые отношения в контексте, такие как, клиент-поставщик (*Customer-Supplier*), непрерывная интеграция (*Continitous Integration*) или разделяемое ядро (*Shared Kernel*) [6]. В то же время, это является дальнейшей задачей, поскольку контекстная карта является инструментом для решения текущих задач, и для рассмотренного примера с торговым модулем вопрос разделения остается открытым, поэтому восклицательный знак присутствует на схеме.

Шаблоны стратегий в DDD

Существует небольшие нюансы использования шаблонов проектирования в предметно-ориентированном проектировании [6]. Используемый повсеместно принцип - доказанное решение для повторяющейся проблемы - предоставляет мало вариантов, из которых можно выбрать. Чаще всего, организационная структура навязывает модели и наша единственная надежда распознать их прежде чем ситуация станет безвыходной. Иногда есть шанс выбрать лучший вариант, или исправить существующую ситуацию, но мы должны знать, что изменения на организационном уровне могут потребовать больше времени, чем предусмотрено на выполнение проекта, или они могу быть просто за пределами наших возможностей.

В свою очередь, если есть сомнения с чего начать, нужно начинать с команды разработчиков. Команда - это, де-факто, наибольший организационный модуль, который может успешно создавать знания о системе. Контексты, будучи единожды распознаны командой, могут ею же и прорабатываться, до создания наиболее предпочтительного архитектурного решения.

С другой стороны, каждый шаблон имеет разную стоимость внедрения. Даже если два шаблона решают одинаковую проблему, например, связывание контекстов, они не могут быть легко заменены друг другом. Например, шаблон ACL (предохранительный уровень) сильно влияет на код приложения (вносит дополнительный слой) и почти не влияет на организацию работы команды, в то время как, шаблон *Partnership* (сотрудничество) или *Customer-Supplier* (заказчик-поставщик) вероятно потребует меньше изменений на уровне кода.

 ISSN 1562-9945

Однако он не будет работать без эффективного канала взаимодействия и хорошо поставленных процессов разработки. Попытка использовать шаблон *Partnership* вне хорошо организованной среды взаимодействия - это тупиковый путь.

Выводы

Проанализированы типичные ситуации, встречающиеся в процессе разработки программного обеспечения, при которых возникает неоднозначность в создаваемых моделях предметной области:

– неоднозначность терминологии;

– различное использование одной и той же концепции;

– интеграция с внешней системой или неподдерживаемым более модулем;

– расширение команды разработчиков.

Для каждого случая предложен способ разрешения неоднозначности с использованием ограниченных контекстов, контекстных карт и применением подходящего шаблона проектирования. Анализ носит качественный характер, учитывая особенность исследуемой предметной области. Результатом анализа является методика использования ограниченных контекстов и контекстных карт на этапе проектирования программного обеспечения.

Несмотря на естественное желание единообразия, создаваемые модели не могут иметь очень широкую сферу применения. Ограниченные контексты предоставляют хорошо определённую и безопасную среду, позволяющую моделям совершенствовать сложность, не жертвуя концептуальной целостностью.

Выявлено что, побочным эффектом применения в больших проектах является то, что контекстная карта, созданная на начальном процессе разработки, показывает неявные границы, существующие в пределах организации, и является ярким естественным снимком уровня, к которому будет стремиться проект. Хорошая контекстная карта даёт представление о факторах, действующих против команды разработчиков, которая может узнать будет ли предполагаемая форма организации труда содействовать или препятствовать этому проекту, даже до его фактического начала.

Таким образом, как выявилось, разбиение на контексты чрезвычайно полезно для быстрого осознания ключевых деталей предметной области проекта и является хорошим инструментом для приня-

тия стратегических решений (что собственно и является назначением любой карты). В результате того, что контекстная карта является инструментом визуализации синергии компонентов разрабатываемой системы, она предоставляет всеобъемлющий взгляд на систему, который диаграммой UML не отражается. Как следствие, контекстная карта позволяет сконцентрироваться на принятии действительно жизнеспособных решений, избавляя от неэффективных сценариев.

ЛИТЕРАТУРА

1. Eric Evans. Domain-driven Design: Tackling Complexity in the Heart of Software. - Addison-Wesley Professional, 2004.
2. Лоскутов А.Ю., Михайлов А.С. Введение в синергетику: Учебное руководство. - М.:Наука. Гл. ред. физ.-мат. лит., 1990.
3. Vaughn Vernon. Implementing Domain-Driven Design. - Addison-Wesley Professional, 2013.
4. Floyd Marinescu, Abel Avram. Domain-Driven Design Quickly. - LULU Press, 2007.
5. Jimmy Nilsson. Applying domain-driven design and patterns: with examples in C# and .NET. - Addison-Wesley, 2006.
6. Marting Fowler. Patterns of Enterprise Application Architecture. - Addison-Wesley Professional, 2002.
7. Scott Millett. Professional Domain Driven Design Patterns. - Wiley, 2014.

 ISSN 1562-9945