УДК 004.41

ОБЕСПЕЧЕНИЕ КАЧЕСТВА ПРОГРАММНЫХ СРЕДСТВ В УСЛОВИЯХ ИЗМЕНЯЮЩИХСЯ ТРЕБОВАНИЙ

В.В. Бахтизин, С.Н. Неборский

Белорусский государственный университет информатики и радиоэлектроники,
Минск, Беларусь, 220000, ул. П. Бровки, 6.
Тел.: +375 017 293 2388.
E-mail: bww@bsuir.by, sergei.neborski@gmail.com

Рассматривается проблема обеспечения качества программных средств в условиях неясных и изменяющихся требований. Предлагается новый гибкий метод разработки программных средств – метод разработки, ведомой требованиями. В основе предлагаемого метода лежат такие принципы, как создание программной архитектуры на основе требований, управление реализацией изменяющихся требований, ориентация на быструю модификацию программных средств. Предлагаемый метод отводит требованиям основополагающую роль в процессе разработки программных средств. Он предполагает декомпозицию программного средства на компоненты, каждый из которых реализует отдельное требование или группу логически связанных требований, содержит метаинформацию о реализуемых требованиях, информацию о связях с другими компонентами, а также механизмы взаимодействия с другими компонентами.

This paper outlines the problem of assuring software quality when requirements to software are changing, and these requirements are uncertain. A new agile method is presented – requirement-driven development. The method is based on the following principles: defining solution architecture from requirements, managing the implementation of changing requirements, and targeting on the ability to easily and fast modify software. The method defines that software requirements are always the key, and they are always on top of the things. The presented method splits the software to a set of components each of which implements a separate requirement (or a group of requirements), contains some metadata on the requirements being implemented, contains the information on links to another components, as well as the mechanisms to interoperate with another components.

Обеспечение качества программных средств

Проблема обеспечения качества программных средств (ПС) всегда являлась и продолжает являться важной и актуальной. Несмотря на то, что постоянно появляются новые методы и подходы к разработке ПС, совершенствуются существующие методы, развиваются ПС, повышающие эффективность разработки. Создаются и развиваются международные и национальные стандарты в области ПС, систем и связанных с ними процессов [1]. Это не удивительно, несмотря на рост инструментов обеспечения качества, растут и сами требования к качеству ПС.

Современный рынок компаний, предлагающие услуги разработки ПС, не беден – и выигрывает тот, кто способен создавать качественные ПС за меньшее время. Но очевидно, не все проекты программной инженерии успешны. Существует множество причин краха проектов, основными из которых являются [2]:

- нечеткие и нереалистичные цели проекта;
- неверная оценка ресурсов, необходимых для выполнения проекта;
- плохо определенные, неясные требования.

Требования – единственный инструмент, позволяющий заказчику поставить задачи перед разработчиком. Но в самом начале разработки, когда есть проблема, которую нужно решить, или цель, которой необходимо добиться, вряд ли возможно четко определить все требования к ПС. Что касается самого процесса разработки, то практика показывает, что от некоторых требований приходится отказываться, вместо них возникают новые, происходит модификация существующих требований. Таким образом, изменение требований – это неизбежный процесс. И необходим некий способ обеспечения высокого качества ПС в условиях изменяющихся требований.

В данной работе предложен новый гибкий (agile) метод разработки ПС – метод разработки, ведомой требованиями. Суть его в том, что он позволяет управлять изменением программной реализации при изменении требований, за счет чего решается проблема поддержания целостности (согласованности) требования и его реализации. Именно это и позволяет обеспечивать высокое качество ПС в условиях изменяющихся требований. В основе разработки, ведомой требованиями, лежат такие принципы: создание программной архитектуры на основе требований; управление реализацией изменяющихся требований; ориентация на быструю модификацию ПС.

Разработка, ведомая требованиями, предполагает, что главную роль в процессе создания ПС играют требования. Именно на основе требований создается программная архитектура. Причем программная архитектура определяется как отображение требований в компоненты программной реализации. Это создает определенные предпосылки для того, чтобы реализация требований стала управляемой. Управление реализацией изменяющихся требований означает, что любое изменение требования автоматически обнаруживает те компоненты, которые должны быть пересмотрены вследствие произведенного изменения. Это

достигается за счет использования разработчиком определенных приемов, например, таких, как управляемая сборка ПС.

Для того, чтобы разрабатываемое ПС было быстро модифицируемым, предлагаемый метод предполагает ведение разработки на основе шаблона проектирования: “модель – представление – контроллер” (Model – View – Controller, MVC). Шаблон MVC должен быть реализован в рамках каждого компонента, а также должна существовать возможность интеграции моделей и контроллеров разных компонентов.

Определив цели и принципы предлагаемого метода, следует рассмотреть существующие подходы к разработке ПС. Причем при их рассмотрении будут показаны отличия от предлагаемого метода. После этого будет приведено детальное описание метода разработки, ведомой требованиями.

Подходы и методы разработки программных средств

Если говорить о подходах к разработке ПС, то на сегодняшний день речь не идет об использовании той или иной модели жизненного цикла ПС. Речь идет об организации всего процесса создания ПС. Так, существуют различные методологии, детально определяющие, что как и кем должно быть сделано для того, чтобы разрабатываемое ПС было именно, таким, каким его хочет видеть заказчик, и все работы были выполнены в срок. В качестве примеров таких методологий можно привести унифицированный процесс Rational (Rational Unified Process, RUP) и каркас решений Microsoft (Microsoft Solutions Framework, MSF). Данные модели – фундаментальны, относятся больше к управлению всем проектом. Они не концентрируются на стадии разработки. При этом, подходы к разработке могут быть использованы разные. Существуют гибкие методы (agile methods), которые концентрируются как раз на разработке. Гибкие методы имеют такие характерные особенности [3]:

наивысший приоритет отводится удовлетворению заказчика;
устойчивость к изменениям требований;
частая и непрерывная доставка работающего ПС;
совместная работа специалистов предметной области и разработчиков;
сборка ПС для удовлетворения потребностей заинтересованных лиц;
прямая связь с заказчиком;
простота – реализация лишь того, что действительно нужно сделать;
лучшие требования и архитектура – результат работы самоорганизующихся команд.

Наиболее широко применяемые гибкие методы – такие, как FDD (Feature-Driven Development – разработка ведомая функциями), XP (eXtreme Programming – экстремальное программирование), TDD (Test-Driven Development – разработка на основе тестов), ASD (Adaptive Software Development – адаптивная разработка ПС), DSDM (Dynamic Systems Development Method – динамичный метод разработки систем).

Метод FDD был разработан Джеффом Де Люка в 1997 году. FDD – итеративный и инкрементный метод, в основе которого лежат такие принципы: моделирование предметной области; разработка ПС на основе его функциональных особенностей; личное (персонифицированное) владение кодом; управление конфигурацией; регулярные сборки ПС; видимость прогресса и результатов [4].

FDD предполагает наличие пяти четко определенных процессов [5]:

разработка общей модели;
построение списка функций;
планирование на основе функций;
проектирование на основе функций;
сборка на основе функций.

Метод FDD опирается на функции как на некоторые формализованные удобные для реализации описания требований. Устойчивость по отношению к изменяющимся требованиям заключается в том, что на новом инкременте разработки ПС некоторая функция может быть модифицирована. При этом, после построения списка функций не допускается их модификация на текущем инкременте – требования считаются зафиксированными. В отличие от FDD, предлагаемый в данной работе метод допускает изменение требований в любой момент времени и позволяет эффективно управлять данными изменениями.

Метод XP был разработан Кентом Беком, Вордом Цуннингамом и Роном Джеффрисоном. XP ориентирован на малые команды разработчиков, которые реализуют неясные и изменяющиеся требования. По сути, он является противопоставлением всеобъемлющим фундаментальным методам, которые зачастую несут целый спектр ненужных и избыточных активностей для малых команд. XP фокусируется на доставке ПС в установленный срок, и это ПС удовлетворяет требованиям пользователя. XP сосредоточен на разработке, и не рассматривает прочие работы, связанные с реализацией ПС. В основе данного метода лежат следующие принципы [3]:

- коммуникация – обеспечение постоянной связи разработчика и пользователя;
- простота – ориентация на простоту в реализации ПС, усложнение реализации лишь по мере увеличения и усложнения требований;
- обратная связь – предоставление возможности пользователям давать свои отзывы о работе ПС.

Несмотря на то, что XP ориентирован на изменяющиеся требования, процесс реализации данных изменений является неуправляемым. Так, концентрируясь на разработке, в рамках XP довольно трудно

отследить изменения, касающиеся взаимосвязанных требований. Отсутствие же моделей, определяющих архитектуру разрабатываемого ПС, ставит под вопрос легкость модификации и дальнейшего развития ПС при переходе к очередной итерации. Можно рассмотреть интеграцию XP и метода гибкого моделирования (agile modeling), однако и здесь остаются нерешенными проблемы, связанные с управлением изменяющихся требований. Кроме того, даже такая интеграция не гарантирует легкость модификации ПС.

Разработка, ведомая тестами, – метод, изначально развивавшийся в рамках XP, и оформившийся позже в отдельный подход к реализации ПС. Общая схема работы данного метода такова: требования, представленные в виде вариантов использования, декомпозируются в наборы поведений, которые нужны для реализации требования. Отдельное такое поведение реализуется отдельным программным модулем – классом, методом класса, набором методов класса. Для каждого такого поведения разрабатывается тест модуля, данный тест будет проверять корректность требуемого поведения. После создания теста модуля необходимо проверить корректность самого теста. Для этого достаточно просто исполнить данный тест: так как необходимое поведение еще не реализовано, тест будет неудачным и это станет подтверждением корректности теста. Только после этого разрабатывается код модуля и тест запускается повторно [6].

Метод TDD имеет ряд недостатков. Причем эти недостатки относятся как к трудности реализации самого метода (например, одна из известных проблем связана с модульным тестированием графического интерфейса пользователя), так и к его концепции. Так, можно говорить о корректности работы ПС лишь с той долей вероятности, с которой корректны разработанные тесты. Кроме того, разработка хороших и всеобъемлющих тестов модулей требует временных затрат и модификация требований только увеличивает данные затраты. В отличие от TDD метод, предлагаемый в данной работе, позволяет минимизировать временные издержки, связанные с изменением требований.

Заявленное преимущество TDD – автоматическое получение архитектуры ПС на основе тестов [7]. Однако такая архитектура представляет собой снимок текущей функциональности ПС, и не содержит гибкого механизма добавления существующей новой функциональности или модификации. Разработка, ведомая требованиями, которая предлагается в данной работе, позволяет определить программную архитектуру, которая решает вышеописанные проблемы.

Создание программной архитектуры на основе требований

Очевидно, изменение требований отражается на архитектуре ПС. Для эффективного управления изменяющимися требованиями существуют системы управления требованиями (СУТ). В качестве примеров СУТ можно привести Telelogic DOORS, IBM RequisitePro, Borland CaliberRM. Для создания архитектуры ПС используются различные CASE-средства, большинство из которых основано на использовании стандартов MDA (Model Driven Architecture – архитектура, ведомая моделями). Гибкие методы разработки ПС выступают с критикой существующих подходов к моделированию, заявляя, что модели должны быть использованы лишь тогда, когда в них действительно есть необходимость. Более того, развитие ПС не преследует цели развития моделей, если разработчик понимает, что должно быть сделано и как развивается проект, модель не эволюционирует и приобретает вид какого-то рудимента [3].

В рамках предлагаемого в данной работе метода программная архитектура строится автоматически на основе требований к ПС. Разработка, ведомая требованиями, не предполагает создания детализированной архитектуры с различными видами диаграмм, как это сделано в MDA или подобных подходах к созданию архитектуры. Предлагаемый метод концентрируется на том, что на основе требований автоматически выделяются базовые компоненты, которые связываются согласно требованиям, за счет чего получаемая архитектура становится управляемой. Обоснованием предлагаемого данным методом автоматического создания архитектуры является то, что использование моделирования для выявления требований и использование СУТ для управления ими позволяет определить требования в понятном для разработчика и готовом для реализации виде.

Идея создания программной архитектуры на основе требований заключается в том, что каждое отдельное требование или группа логически связанных требований реализуется в виде отдельного компонента, т.е., можно задать следующее отображение *F*:

$F : R \to C$, где R – множество требований,
C – множество компонентов;
причем $\forall c \in C \exists r \in R \mid F(r) = c$.

Следует отметить, что функция $F : R \to C$ является сюръекцией, так как для каждого $c \in C$ существует некоторое $r \in R$ такое, что $F(r) = c$. При этом функция $F : R \to C$ не является инъективной, ведь нельзя утверждать, что из $F(r) = F(r')$ следует $r = r'$.

Компонент ПС $c \in C$ реализует некоторое требование $r \in R$ (или набор логически связанных требований) и хранит определенную метаинформацию о требованиях. Данные компоненты и формируют программную архитектуру. Далее, в процессе разработки ПС, полученные компоненты заполняются кодами классов, интерфейсов, и т.п., которые непосредственно реализуют требование. Логическое представление компонента ПС показано на рис. 1.

Рис. 1. Компонент программного средства

Управление реализацией изменяющихся требований

Получаемая программная архитектура содержит механизм контроля изменения требований. Данный механизм реализуется посредством задания фактов зависимости требований. Компоненты, которые реализуют зависимые требования, также являются зависимыми, их сборка происходит в строго определенном порядке, установленном на основе данных зависимостей [8]. Рис. 2 поясняет описываемый подход к созданию управляемой программной архитектуры. Именно, благодаря такому подходу программная архитектура является управляемой, т.е. такой, которая позволяет автоматически выявлять в исходном коде модули (классы, интерфейсы, методы), на которые влияет изменение требований.

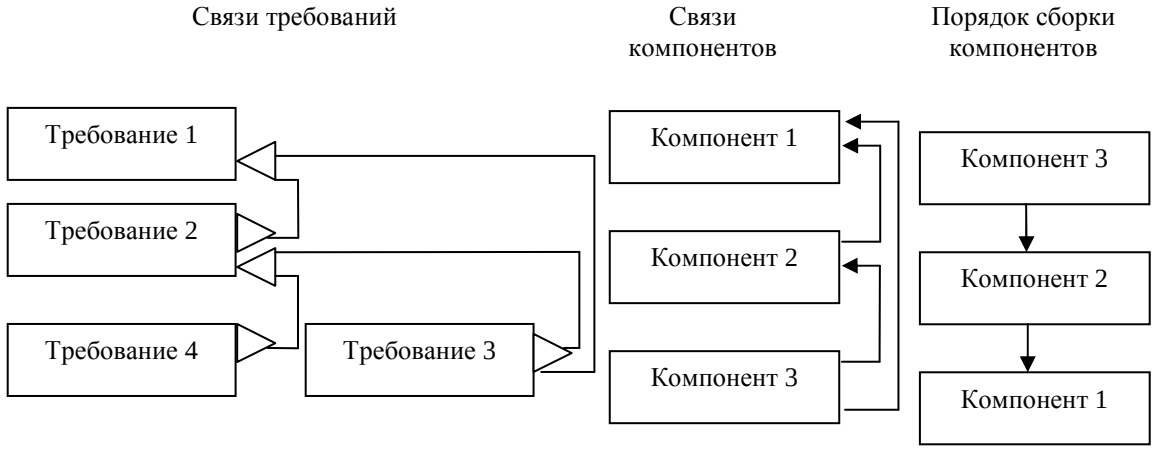


Рис. 2. Связи требований и компонентов ПС

На рис. 2 показано требование 1, зависящее от требований 2 и 3, а также требование 2, зависящее от требований 3, 4. Обозначая множество требований как *R* мощности *n* отношение зависимости *Q* на $R \times R$ можно представить в виде булевой матрицы $B=[B_{ij}]$ размерностью $n \times n$:

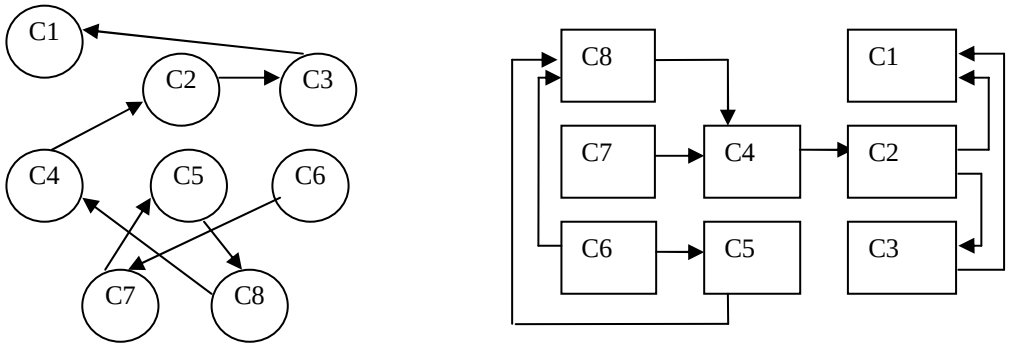
$$B_{ij} = \begin{cases} 1, \text{если} (r_i, r_j) \in Q; \\ 0, \text{если} (r_i, r_j) \notin Q. \end{cases}$$

В соответствии с данным получаются и компоненты ПС. Подразумевая, что требования 3 и 4 связаны логически, их реализация может быть помещена в один компонент – компонент 3.

Управляемая программная архитектура позволяет гарантировать целостность исходного кода ПС при изменении требований, причем поддержка целостности осуществляется автоматически. Целостность означает следующее: код реализации некоторого требования всегда соответствует данному требованию, т. е. всегда требование и его реализация согласованы. Если же данная согласованность нарушается, то сборка всего ПС становится невозможной.

Для реализации автоматической поддержки целостности предлагается задавать порядок сборки компонентов *C*, причем этот порядок должен устанавливаться на основе зависимостей требований *Q*. Реализовать такой механизм можно построив полугамильтонов граф *H*: *H(C,E),* где *C* – множество вершин графа (компоненты ПС); *E* – множество ориентированных ребер, таких, что $(c_i, c_j) \in E, i \neq j;\ i, j = \overline{1,n}$, n – количество компонентов *C*.

Полугамильтонов граф – это граф, в котором существует гамильтонова цепь, т.е. цепь, проходящая через все вершины по одному разу [9]. Вершинами полугамильтонова графа *G*(*C*) будут являться компоненты. Тогда порядок, в котором они появляются в пути, будет соответствовать порядку построения ПС. На рис. 3 показан пример полугамильтонова графа для некоторого ПС.



Порядок сборки компонентов Зависимости компонентов

Рис. 3. Пример полугамильтонова графа для некоторого ПС

Для сигнализации о несогласованности кода предлагается использовать ошибки или предупреждения на этапе сборки проекта, т.е. ошибки и предупреждения, выдаваемые средой разработки. Так, необходимо к каждому компоненту добавлять некоторую метаинформацию, которая будет отражать факт того, согласован ли код данного компонента с тем требованием, которое он реализует. При сборке отдельного компонента должен происходить анализ данной информации. В том случае, если код компонента не соответствует текущему состоянию требования, сборка данного компонента и ПС в целом прекращаются. Обращаясь к рис. 2, если не удалось построить компонент 2, то вместе с тем не будет построен и компонент 1, т.е. зависимости требований действительно учтены. На рис. 4 показан пример возможной реализации механизма управляемой сборки отдельного компонента, который основан на контроле версий требований.

Стрелками на этом рисунке обозначены операции работы с данными. Например, такими операциями могут являться запросы к базе данных требований или вызовы функций API интегрированной среды разработки для извлечения метаинформации. Логически, данный рис. показывает, что изменение требования и его реализация происходят параллельно. Факт изменения требования будет обнаружен разработчиком при выполнении сборки ПС.

Можно оптимизировать вышеприведенную схему управляемой сборки с тем, чтобы исключить запросы к базе данных требований каждый раз при сборке ПС. Для этого можно включить признак, определяющий факт согласованности требования и его реализации, в метаинформацию компонента. Данную метаинформацию необходимо устанавливать каждый раз при выполнении процедуры регенерации программной архитектуры. Регенерация предполагает изменение кодов существующих компонентов, если отображаемые ими требования были изменены, а также добавление или удаление новых компонентов согласно текущему состоянию требований. Регенерация – это автоматический процесс, который должен происходить непрерывно на протяжении разработки ПС. На рис. 5 показаны отдельные фазы итерационного процесса разработки ПС в условиях изменяющихся требований к нему.

Рис. 4. Пример реализации механизма управляемой сборки компонента

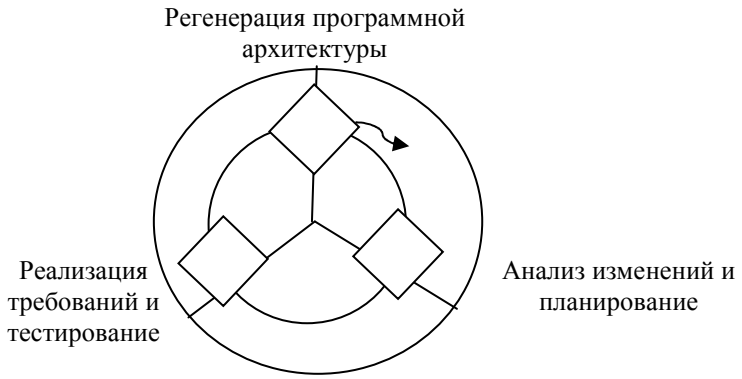


Рис. 5. Итеративность разработки, ведомой требованиями

Ориентация на быструю модификацию ПС

Несмотря на то, что создание архитектуры на основе требований и управляемая сборка ПС позволяют улучшить качество разрабатываемого ПС и всего процесса реализации ПС, о действительной эффективности предлагаемого метода можно говорить лишь доказав легкость модификации ПС. Так, разработка, ведомая требованиями, предполагает, что требования изменяются постоянно и следовательно, необходим простой и универсальный механизм модификации ПС при модификации требований к нему.

Одним из путей реализации механизма легкого изменения ПС является использование шаблона проектирования MVC. Впервые шаблон MVC появился в реализации Smalltalk-80, развитие концепции MVC связано с именами Голдберга, Робсона, Краснера. Ключевым моментом здесь является разделение представления, поведения и данных. Представление является графическим отображением данных, с которыми работает ПС. Модель – непосредственно те данные. Управляет этими данными, т.е. определяет поведение ПС, контроллер [10].

Говоря о подходах к модификации ПС, можно упомянуть инфраструктуру для реконфигурирования приложений, как Lira [11]. Данная инфраструктура предлагает осуществлять модификацию ПС средствами агентов, обменивающихся сообщениями. Агенты представляют собой программные элементы, ассоциированные с компонентами ПС. Lira описывает набор интерфейсов, которые и служат основой для обмена информацией между агентами. В отличие от Lira, позволяющей управлять изменением компонентов ПС посредством внешних по отношению к этому ПС агентов, предлагаемый в данной работе метод разработки, ведомой требованиями, основывает концепцию модификации ПС на шаблоне MVC.

Шаблон MVC должен быть реализован в рамках каждого компонента. Очевидно, в силу специфики требований некоторые элементы MVC могут отсутствовать (например, если требования, реализуемые в некотором компоненте, описывают лишь представление данных и механизм их обработки, модель в этом компоненте может отсутствовать). За счет MVC, реализованного в отдельном компоненте, достигается легкость модификации данного компонента при изменении требований, реализуемый в нем, тех требований, от которых он зависит. Последнее означает, что благодаря стандартному механизму хранения, отображения и обработки данных гарантируется легкость модификации кода некоторого компонента при изменении другого компонента, от которого он зависит. Формально описать предлагаемую реализацию шаблона MVC можно так:

$$S = M \cup V \cup K = \{s | s \in M \vee s \in \mathrm{V} \vee s \in K\},$$

где S – объединение множеств элементов MVC; *M* – множество моделей; *V* – множество представлений; *K* – множество контроллеров.

Множество *S* как объединение множеств *M, V, K* удобно показать в виде диаграммы Эйлера (очевидно, сами множества *M, V, K* не пересекаются; рис. 6):

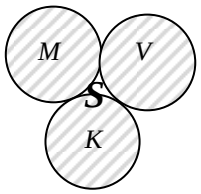


Рис. 6. Объединение множеств MVC

$G : K \to S$, где *G* – отображение множества компонентов *C* в объединение множеств элементов MVC *S*,

причем $\forall c \in C \exists$ как минимум одно $\mathrm{s} \in S \,|\, G(s) = c$

и $\forall s \in S \exists$ одно и только одно $c \in C \,|\, G(s) = c$.

Однако еще большую способность к модификации приобретает ПС в том случае, если реализован механизм интеграции элементов MVC в различных компонентах. Так, определив и стандартизировав механизм взаимодействия моделей, представлений и контроллеров различных компонентов, компоненты становятся не только повторно используемыми, но и динамически подключаемыми. Главным достоинством динамически подключаемых компонентов то, что они позволяют обеспечить быструю и простую модификацию ПС после поставки пользователю. На рис. 7 показана схема реализации механизма интеграции элементов MVC на уровне компонентов ПС.

Пунктирными линиями на рис. 7 обозначены косвенные связи, сплошными линиями – прямые. Разница между данными типами связей состоит в том, что прямые связи устанавливаются на этапе сборки ПС (процессы компиляции, линкования), а косвенные – устанавливаются во время работы данного ПС (процессы инициализации, конфигурирования). Также на рис. 7 подписан порядок выполнения операций. Следует обратить внимание на то, что прямые связи между компонентами отсутствуют, взаимодействие между ними осуществляется лишь под управлением диспетчера компонентов и на основе стандартных интерфейсов. Именно так достигается динамическое подключение и динамическое связывание компонентов.

Интеграция элементов MVC образует часть инфраструктуры компонентов, на которых, как можно видеть, основан весь метод разработки, ведомой требованиями. К другим элементам данной инфраструктуры могут быть отнесены различные сервисы, например, сервис обработки ошибок или сервис доступа к данным. Разработка, ведомая требованиями, предполагает наличие некоторого базового набора повторно используемых программных блоков, которые используются компонентами.

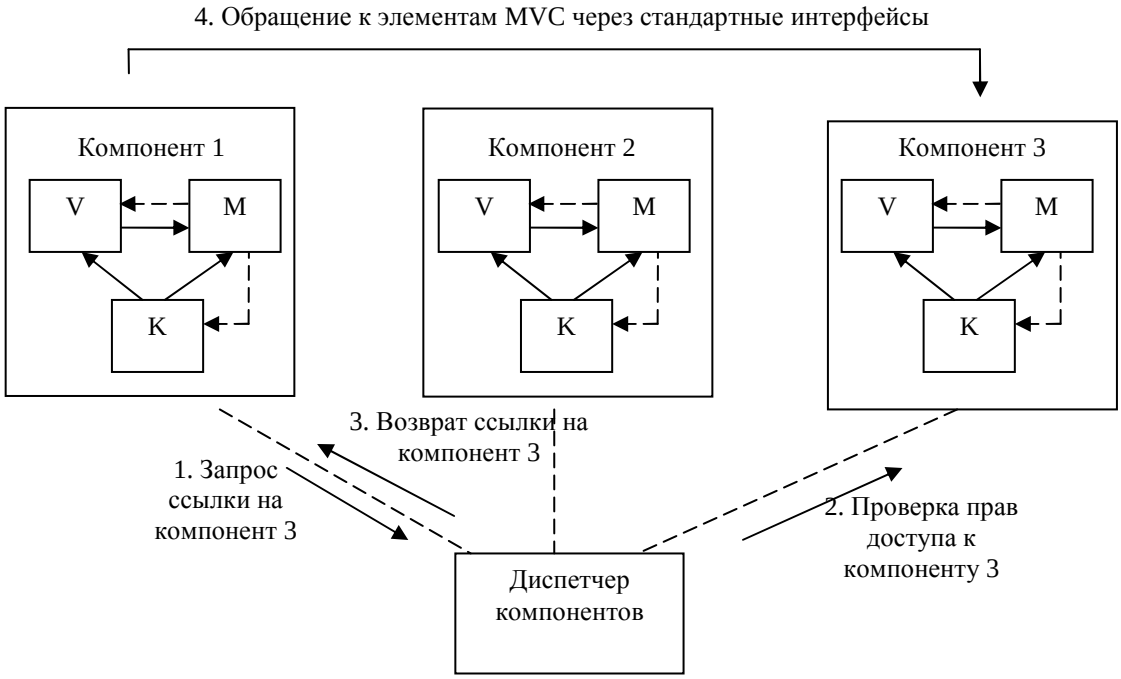


Рис. 7. Реализация механизма интеграции элементов MVC на уровне компонентов ПС

Выводы

Таким образом, разработка, ведомая требованиями, – это новый гибкий метод разработки ПС, который ставит во главу угла требования. Данный метод основан на декомпозиции ПС на компоненты, каждый из которых реализует отдельное требование или группу логически связанных требований, содержит метаинформацию о реализуемых требованиях, информацию о связях с другими компонентами, а также механизмы взаимодействия с другими компонентами. Преимуществами предлагаемого метода являются:

- автоматическое гарантирование согласованности требования и кода его реализации;
- возможность модификации требований на любом этапе разработки ПС;
- легкость модификации ПС при изменении требований к нему;
- сокращение времени разработки за счет автоматического получения программной архитектуры и поддержания ее целостности;
- общение разработчика и заказчика на одном языке – языке требований.

Все это и позволяет утверждать, что данный метод позволит обеспечить высокое качество ПС при разработке его в условиях неясных и изменяющихся требований.

1. *Бахтизин В.В., Глухова Л.А.* Стандартизация и сертификация программного обеспечения: Учеб. пособие. – Мн.: БГУИР, 2006. – 200 с.
2. *Charette R.N.* Why Software Fails // IEEE Spectrum, sep 2005.
3. *Hunt J.* Agile Software Construction. Springer-Verlag London Limited, 2005. – 254 c.
4. *Palmer S.R., Felsing J.M.* A Practical Guide to Feature-Driven Development. Prentice Hall, 2002. – 304 c.
5. *Coad P., Lefebvre E., De Luca J.* Java Modeling in Color With UML: Enterprise Components and Process. Prentice Hall International, 1999. – 221 p.
6. *Gold R., Hammell T., Snyder T.* Test Driven Development: A J2EE Example. Apress, 2004 – 296 p.
7. *Newkirk J., Vorontsov A.* Test-Driven Development in Microsoft .NET. Microsoft Press, 2004. – 304 p.
8. *Бахтизин В.В., Неборский С.Н.* Создание управляемой программной архитектуры // Программные продукты и системы. – 2006. – № 3 (75). – С. 2 – 5.
9. *Оре О.* Теория графов. – 2-е изд. Главная редакция физико-математической литературы. – М.: Наука, 1980.
10. *Krasner G., Pope S.* A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System // J. of Object Oriented Programming. – 1998. – Vol. 1, N 3. – P. 26–49.
11. *Castaldi M., Carzaniga A., Inverardi P., Wolf A.* A Lightweight Infrastructure for Reconfiguring Applications // B. Westfechtel, A. van der Hoek (Eds.): SCM 2001/2003, LNCS 2649, Springer-Verlag Berlin Heidelberg, 2003. – P. 231–244.