Министерство образования и науки Российской Федерации

МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(государственный университет)

ФАКУЛЬТЕТ УПРАВЛЕНИЯ И ПРИКЛАДНОЙ МАТЕМАТИКИ

КАФЕДРА ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ

Специализация 011600 «Прикладные физика и математика»

“МОДЕЛИРОВАНИЕ СИСТЕМ ИЗ КОМПОНЕНТОВ НА ПРИМЕРЕ РЕАЛИЗАЦИИ ПРИЛОЖЕНИЯ В СРЕДЕ JAVA”

Выпускная квалификационная работа
(магистерская диссертация)

Выполнил:
студент 273а группы _____Губайдулин Айдар Наилевич

Научный руководитель:
д.ф.-м.н., профессор _____Лаврищева Екатерина Михайловна

Москва – 2018

Содержание

ВВЕДЕНИЕ 4
Актуальность работы 4
Цель работы 5
Общие положения теории и методов моделирования систем 5
Глава 1. ПАРАДИГМА ПРОГРАММИРОВАНИЯ ПРИЛОЖЕНИЙ ИЗ ГОТОВЫХ РЕСУРСОВ 8
Базовые понятия компонентной теории 9
Модели компонента, среды и интерфейса 11
Модель компонента 12
Модель интерфейса 13
Модель компонентной среды CE 14
Операции над компонентами в компонентной среде 16
Компонентная алгебра 17
Типы отношений компонентов 19
Глава 2. БАЗОВЫЕ МОДЕЛИ И МЕТОДЫ МОДЕЛИРОВАНИЯ СИСТЕМ 21
Модель MDD (Model Domain Development) 21
Модель MDA (Model Driven Architecture) 22
Модели систем для разных платформ PIM и PSM 24
Вычислительно-независимая модель 25
Преобразование моделей PIM→PSM 26
Моделирование систем в языке UML 28
Моделирование изменяемых систем 29
Конфигурационная сборка ГОР 31
Глава 3. ПОДХОДЫ В РЕАЛИЗАЦИИ КПИ В СОВРЕМЕННЫХ СРЕДАХ 35
Понятие КПИ 35
Внешние связи вариантов 38
Внутренние связи вариантов 40
Глава 4. РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ В СРЕДЕ JAVA 46
Обзор используемых технологий в среде JAVA 46
Enterprise JavaBeans 46

Apache Ant....47
Java Persistence API....48
JavaServer Faces....48
Структура хранения данных....49
Структура приложения....50
Пользовательский интерфейс....51
ВЫВОДЫ....53
ЛИТЕРАТУРА....56

ВВЕДЕНИЕ

Актуальность работы

В настоящее время методы моделирования систем из готовых ресурсов широко обсуждаются и находят применение при разработке программного обеспечения для решения самых разных задач во всех высокотехнологичных областях.

В данной работе проведен анализ современных методов проектирования программ и систем с целью использования новых современных подходов к моделированию программных приложений и систем. Характерной особенностью современных подходов моделирования является построение модели приложения из функций реализуемой предметной области и из готовых программных элементов системного и сервисного типов, содержащихся в большом количестве в общих библиотеках Интернет в виде компонентов повторного использования (КПИ) для разных применений, КПИ отличаются функциональной правильностью и готовностью к работе.

Рассмотрение разных подходов к моделированию новых систем и их анализ показывает, что уровень проектирования систем и приложений выше, чем программирование модулей и КПИ в некотором языке программирования (C+, Ruby, Java и др.).

Среди рассмотренных подходов особое место занимает разработка и стандартизация КПИ для всеобщего применения. При использовании повторных КПИ предлагается набор операций над модулями или компонентами общего назначения. К ним относятся композиция, наследование, полиморфизм, паттерны и другие. Эти операции над КПИ позволяют их изменять и манипулировать ими в разных средах.

Проектирование систем с помощью готовых модулей и КПИ повышает скорость разработки систем программного обеспечния и позволяет быстро включать новые КПИ в имеющуюся структуру приложения.

Вновь разрабатываемые модули проверяются, тестируются и доводятся до готовности в программной среде выбранного языка программирования разрабатываемых компонентов.

Цель работы

Целью данной работы является анализ современных подходов к моделированию систем из готовых ресурсов типа reuses, КПИ, использование теории (моделей и метода) компонентного программирования и реализация на основе этой теории программного обеспечения приложения на языка программирования JAVA.

Общие положения теории и методов моделирования систем

В последние годы в мире сформировался подход на моделирование систем по моделям или каркасам и метод их трансформации к выходному коду для задаваемой платформы, задаваемой моделями PSM (Platform Specific Model, платформенно-зависимая модель**)** и PIM (Platform Independent Model, платформенно-независимая модель**)**. а также модели проектирования систем с использованием КПИ - MDA (Model Driving Architecture) OMG, MDD (Model Driving Development), IBM Rational Rose, MDE (Model Driving Engineering), MSE, SOA (Service Oriented Architecture), SCA (Service - Component Architecture) и др.

Модельный подход стал доминирующим для разработчиков сложных систем и нашел самое широкое применение на теоретическом и практическом уровне. Этот подход к моделированию ежегодно обсуждается на многих конференциях (например,

MODELS) и получил новое развитие в направлении обеспечения изменяемости (вариабельности) систем с помощью новой модели FM (Model Feature) и модели конфигурирования вариантов программных систем в проекте SEI USA ProductLine/Productfamily (2004) и в ряде других проектов. Новые языки и методы моделирования вариабельных программных и информационных систем обсуждаются на международных конференциях SPLE, VAMOS, ICTERI, Reusebility и др., начиная с 2008г.

К новым языкам и средствам моделирования относятся:

1. Язык сценариев сценариев использования (use cases) с помощью диаграммных характеристик UML, FODA, ConIPF, модели архитектуры Koala, xADL и др.;
2. объектно-компонентный метод (ОКМ) определения моделей взаимодействия элементов систем из объектов и компонентов;
3. описания в языках программирования (ЯП) готовых ресурсов и артефактов приложений с использованием вариантных точек в OVM, VSL, ConIPF и др.;
4. описания конфигурационной модели (Kconfig) и Web-инструментов сборки ГОР (Kbuild) и др.
5. Сборка (конфигурация) ресурсов в систему и возможностью компонентов в системе Corba, Prolog и др.;

Основу моделирования систем составляют модели архитектуры и базовые характеристики отдельных компонентов, которые отмечаются в модели MF.

Основная идея метода моделирования систем сформировалась под влиянием UML (Unified Modelling Language, 1994) и объектно-ориентированного программирования (ООП). Моделирование в языке состоит в построении диаграммной модели системы, в вершинах которой находятся отдельные объекты системы, а на дугах – отношения между ними. Элементы модели трансформируются в программный

код на конкретном языке программирования, который потом тестируется и документируется. Основными элементами методов проектирования систем являются готовые программные элементы – assets, artefacts, КПИ, reuses и др.

Далее дается анализ формальных механизмов моделирования систем из КПИ и описание понятия компонента, КПИ и операций над ними в процессе разработки из них новых систем. Приводится описание теории компонентного программированияи новых систем и приложений, разработанной руководителем этой работы проф. Лаврищевой Е.М, обосновывается высокий уровень этой теории и ее применимость к формальному описанию разных видов приложений и систем для работы в среде типа JAVA, .NET и др. Предложена автором конкретизация метода компонентного программирования и операций наследования, сборки и тестирования применительно к отдельному приложению из компонентов в среде JAVA, приводится описание конкретных компонентов типа Java-beans приложения в этом языке программирования и их исполнение в заданной среде.

На основе выполненной разработки приложения делается вывод о целесообразности и эффективности моделирования приложений из компонентов.

Глава 1. ПАРАДИГМА ПРОГРАММИРОВАНИЯ ПРИЛОЖЕНИЙ ИЗ ГОТОВЫХ РЕСУРСОВ

В течение последних двух десятилетий в индустрии разработки программного обеспечеия сформировались теоретические, методические и практические подходы к разработке приложений и программных систем (ПС) из готовых ресурсов (reuses, artifacts, components, ассеты (assets), и др.). Созданы среды разработки ПС из ресурсов (MS.Net, MSF, CORBA, Rational Rose, UML, JAVA и др.). Эти готовые ресурсы накапливаются в виде компонентов повторного использования (КПИ) в разных библиотеках и репозиториях сети Интернет. В системных средах реализованы разные механизмы сборки или конфигурирования готовых ресурсов в ПС.

Теоретическая основа метода объектно-компонентного моделирования (ОКМ) систем и приложений из компонентов и готовых ресурсов КПИ определена в работах [1-8]. Далее приводится описание основных ее положений, необходимое для практического проектирования.

Объектно-компонентная теория (метод ОКМ), основан на алгебро-математическом аппарате представления моделей (объектной и компонентной), интерфейсов взаимодействия этих ресурсов и интеграции их в приложения и системы. Теория позволяет формально выделять компоненты для системы, строить модель предметной области из компонентов, а затем специфицировать каждый ресурс в ЯП, а их отношения между собой в языке интерфейсов IDL, API и др. Технология построения моделей из ресурсов и их изоморфного отображения в структуру системы образует целый набор средств обработки КПИ (сертификация ресурсов, занесение их в репозиторий и выбор) и их сборки в структуру с необходимым преобразованием передаваемых данных между разными компонентами.

Компонентная теория включает модели компонентов, среды и компонентную алгебру для управления готовыми ресурсами системы, преобразования алгоритмов ресурсов в программу реализации ресурса и конфигурированию ресурсов в приложение. Управления развитием системы включает операции реинженерии, рефакторинга и реверсной инженерия для внесения изменений в систему и ее функции [6–10].

Под **готовым ресурсом** понимается разноязычная форма представления функции предметной области. Каждый ресурс – это спецификация алгоритма функции и данных, которые могут относиться к фундаментальным (FDT), общим GDT (ISO/IEC 11404) и нестандартным типам данных (Big Data). Данные используются для обмена разных ресурсов между собой через интерфейс, описанный в IDL.[3]. По современным меркам каждый программный ресурс или КПИ многоразового применения должен иметь сертификат (паспорт, описание функций и типов данных (ТД) передаваемых другим компонентам через интерфейс).

Паспорт – это информационная часть, которая в языке WSDL содержит:

– *название* функции;

– *ID* – идентификатор ресурса;

– *содержание* (функции);

– *параметры* вызова других программ;

– *инструменты* поддержки выполнения программы;

– *необязательные атрибуты* (дата, состояние, версия, право доступа, автор, дата создания, срок пригодности, правила приобретения и т. п.).

Ресурс в стандартной форме упрощает процесс создания из них ПС за счет формального описания параметров и интерфейсов в IDL. По имени ресурса осуществляется поиск и анализ применимости ресурса во вновь создаваемой системе методом сборки.

Моделирование и реализация приложений, ПС из ресурсов предполагает их описание в предметно-ориентированном языке DSL с использованием КПИ и CASE-инструментов (трансляторов с ЯП, тестеров, пруверов, трасформеров, генераторов, сборщиков и т.п.).

Базовые понятия компонентной теории

Готовый компонент является самостоятельным продуктом, который реализует некоторую функцию предметной области и может взаимодействовать с другими компонентами через интерфейсы. Компонент конструируется самостоятельно, как некоторая абстракция, которая содержит в себе артефакт и информацию о нем.

Артефакт – это смысл и описание функции в ЯП, реализация (implementation), интерфейс (interface) и схема развертывания (deployment) компонента.

Реализация – это код, который будет выполняться при обращении к операциям, определенным в интерфейсах компонента. Компонент может иметь несколько реализаций в зависимости от операционной среды, модели данных, СУБД и др.

Интерфейс – это операции обращения к другим компонентам для передачи аргументов и результатов при взаимодействии их между собой. Каждый компонент может иметь множество интерфейсов.

Компонент может отражать некоторое типовое решение и имеет типовую структуру, характеристики и атрибуты которой в интерфейсной части используются для обмена данными при взаимодействии с другими.

Определение 1. *Компонент повторного использования* (КПИ) – это готовый программный компонент (reuses, assets, functions, patterns, services и др.), который оформляется стандартно как готовый ресурс для использования другими пользователями.

КПИ – это неделимый и инкапсулированный объект, удовлетворяющий функциональным требованиям и принципу взаимодействия в разных средах.

С формальной точки зрения $М_{кпи} = (T, I, F, R, S)$,
где T – тип компонента, I – интерфейс компонента, F – функциональность, R – реализация, S – функциональный сервис для взаимосвязи с другими.

Ресурсы, которые имеют общие переменные, функции и поведение, группируются в **классы** или **множества**. Совокупность внешних переменных и методов класса определяют интерфейс между экземпляров класса. Интерфейс состоит из вызовов методов с атрибутами, которые задают внешние переменные экземпляра класса. Для каждой такой переменной делается выборка значения (*get–метод*) и присвоение нового значения (*set–метод*). Существует еще специальный интерфейс – Home–интерфейс в модели EJB языка JAVA.

Класс может быть:

– *объединенным* множеством или прямой суммой нескольких других множеств;
– *пересечением* множеств, являясь общей частью других множеств;
– *агрегированным* множеств, являясь подмножеством декартового произведения

нескольких других множеств.

Элементами или экземплярами класса могут быть:

- компонент, который реализует некоторую функцию;
- КПИ, beans в языке JAVA и др.;
- каркасы, паттерны, контейнеры, artifacts и др.;
- интерфейсы медиаторы, задающие смысл взаимодействия компонентов;
- системные компоненты общего вида – развертывания (Deployment components), рабочие продукты (Work product components), среда выполнения (Execution components) и др.;
- сервисы брокера CORBA (управление передачей параметров в среде клиент-сервер, выполнение компонентов и др.).

На множестве компонентов формируются разные варианты компонентных систем.

Модели компонента, среды и интерфейса

Технология построения ПС из компонентов и КПИ базируется на механизмах поиска и отбора компонентов, аннотированных и размещенных в репозитории [3, 10].

Сущность теории компонентного программирования заключается в создании ПС из готовых компонентов и КПИ в виде системы взаимодействующих КПИ, которые конфигурируются системными средствами современных сред и используются для решения реализованных задач в продукте приложения.

Формализация отдельных аспектов изготовления ПС из готовых КПИ начала проводится для систем из модулей в 1980–годах прошлого столетия и продолжается до настоящего времени, как сборочное программирование систем из разнородных КПИ (объектов, компонентов, сервисов, аспектов и др.). К сборочному программированию отнесены парадигмы (объектная, компонентная, сервисная, аспектная и др.) [3]. Теория сборки получила развитие в ряде фундаментальных проектов (1998–2012), в диссертационных работах и опубликованных монографиях и статьях [1-3, 6-10, 15-23]. Отдельные аспекты компонентной теории прошли апробацию в системе информатизации НАН Украины. В результате ОКМ метод,

объединивший объектный подход моделирования приложений с компонентным подходом, стал практически апробированным формальным методом. Его суть состоит в представлении предметной области объектной моделью ОМ и трансформации функций объектов в программные компоненты, которые интегрируются в ПС с помощью формальных моделей компонентов, компонентной среды, компонентной системы, компонентной алгебры и интерфейсов компонентов, способствующих изоморфному отображению разных ТД компонентов сборки.

Модель компонента

Эта модель определяет типовые решения, касающиеся функциональной сущности компонента, его архитектуры, структуры, свойств и характеристик и имеет вид

$$MComp = (CName, CIn, CFact, CIm, CSer),$$

где $CName$ – уникальное имя компонента;

$CIn = \{CIn^i\}$ – множество интерфейсов, связанных с компонентом;

$CFact$ – управление экземплярами компонента;

$CIm = \{CIm^j\}$ – множество реализаций компонента;

$CSer = \{CSer^r\}$ – множество системных сервисов.

Множество $CIn = CInI \cup CInO$ состоит из входных $CInI$ и выходных $CInO$ интерфейсов. Отличие между ними заключается в том, что для входных интерфейсов компонент имеет реализации, а выходные интерфейсы находятся в других компонентах. $CFact$ определяет методы, которые необходимы для управления экземплярами компонента (поиск, создание, уничтожение и т.п.). К отношениям между объектами отнесены наследование, экземпляризация, контракт, связь и взаимодействие.

Каждый компонент имеет реализацию вида

$$CIm^j = (ImName^j, ImFunc^j, ImSpec^j),$$

где $ImName^j$ – идентификатор имени реализации компонента;

$ImFunc^j$ – функциональность, соответствующая данной реализации (как совокупность реализаций методов);

$ImSpec^j$ – спецификация реализации (описание условий выполнения, параметров настройки и т. д.).

Модель интерфейса

Модель интерфейса компонента имеет вид

$$CIn^i = (InName^i, InFun^i, CIn,^i InSpec^i),$$

где $InName^i$ – имя интерфейса;

$InFun^i$ – функциональность (методы), реализованная данным интерфейсом;

CIn^i – интерфейс управления экземплярами компонента;

$InSpec^i$ – спецификация интерфейса (описания типов, констант, других элементов данных, сигнатур методов и т. д.).

Интерфейс In^i из формулы определяет условия управления экземплярами:

1. поиск и определение необходимого экземпляра компонента – *Locate*;
2. создание экземпляра компонента – *Create*;
3. удаление экземпляра компонента – *Remove*.
4. операции управление компонентами – это $CIn^i = \{Locate, Create, Remove\}$.

Необходимым условием существования компонента – *это целостность*:

$$\forall CIni \in CInI\ \exists CImj \in CIm\ [Provide(CIn) \subseteq CImj],$$

где $Provide(CIni)$ означает функциональность, которая обеспечивает реализацию методов интерфейса $CIni$. Для взаимодействия двух компонентов $C1$ и $C2$ существует следующее необходимое условие: если $CIni1 \in CInO1$, то должен существовать $CInk2 \in CInI2$ такой, что

$$Sign(CIni1)=Sign(CInk2)\&Provide(CIni1) \subseteq CImj2,$$

в котором $Sign(.)$ означает сигнатуру соответствующего интерфейса.

Здесь знак включения означает, что избранная реализация компонента обеспечивает поддержку не только необходимого интерфейса, но и других возможностей. Такие технологии CORBA, JAVA, C++ и др. содержат необходимые интерфейсные средства. Интерфейс может иметь несколько реализаций, которые

различаются особенностями функционирования (например, операционной средой, средствами сохранения данных и т.д.).

Модель компонентной среды CE

Компонентная среда имеет вид:

$CE = (NameSpace, InRep, ImRep, CSer, CSerIm)$,

где $NameSpace = \{CName^m\}$ – множество имен компонентов среды;

$InRep = \{InRep^i\}$ – репозиторий интерфейсов среды;

$ImRep = \{ImRep^j\}$ – репозиторий реализаций;

$CSer = \{CSer^r\}$ – интерфейс множества системных сервисов;

$CSerIm = \{CSerIm^r\}$ – множество реализаций для системных сервисов.

Компонентная среда на уровне модели рассматривается как множество серверов приложений, где разворачиваются компоненты–контейнеры, экземпляры которых обеспечивают реализацию функциональности компонента. Взаимосвязь контейнера с сервером осуществляется через управление стандартизированными интерфейсами *(CFact)*, а связь между компонентами с помощью *CSer*.

Определение 3. Каркас компонентной среды – это совокупность имен компонентов, интерфейсов и реализаций как пустые множества,

$$FW = (\varnothing, \varnothing, \varnothing, CSer, CSerIm).$$

Пусть $FW1 = (\varnothing, \varnothing, \varnothing, CSer1, CSerImp1)$ и $FW2 = (\varnothing, \varnothing, \varnothing, CSer2, CSerIm2)$ – два каркаса, то каркас *FW1* совместим с каркасом *FW2*, если существует отображение $SMap: CSer1 \rightarrow CSer2$ такое, что $SMap(CSer1) \subseteq CSer2$.

Понятие каркаса. Каркас, или паттерн, является расширением понятия компонента, это абстракция, содержащий описание взаимодействия совокупности объектов в общей кооперативной деятельности, для которой определены роли участников и их распределена ответственность. Паттерн представляется повторяющимся частью программного элемента, как схемы или взаимосвязи контекстного описания решения проблемы. Он фиксирует обобщенное решение для общения с заказчиком, а также обеспечивает использование опыта известные проблемы, которые ранее передавались с помощью модели описания знаний. Кроме

того, он позволяет назначать сервисы, которые действуют в различных конкретных обстоятельствах.

С помощью паттерна определяется определенная вариантность взаимодействия компонентов применения таких механизмов, как наследование, композиция и делегирование. Наследование является средством определения участников взаимодействия, то есть участников паттерна, как подкласса абстрактного класса. Реализация каждого класса выполняется независимо от других. Композиция компонентов в паттернах ориентирована на создание новой функциональности участника взаимодействия, независимо от других, в том числе и динамично. Делегирование это механизм передачи запроса через посредника исполнителю для выполнения определенной функции.

Различаются креативные, поведенческие и архитектурные паттерны. Креативные паттерны отражают типовые схемы распределения полномочий по созданию экземпляров компонентов в соответствии вариантности и их генерации. Поведенческие паттерны определяют типовые схемы взаимодействия классов и их экземпляров с достижением вариантности поведения системы. Структурные паттерны определяют типичные композиции классов компонентов и пути достижения вариантности структуры системы.

Применение КП дает значительные преимущества за счет использования КПВ и улучшения организации и управления разработкой СПС из них, при условии, что для компонентов обязательно определенные интерфейсы и разработаны механизмы языкового и программного характера для решения проблем взаимосвязей объектов и компонентов в процессе их композиции в сложные структуры ПС и СПС. Кроме этого, остаются задачи, связанные с управлением процессов разработки ПС с КПВ и различных вариантов ПС и СПС.

Совместимость каркасов и компонентных сред означает, что среду $CSer_1$ должен входить сервис именования, а в $CServ_2$ - аналогичный сервис и между этими

сервисами должна существовать такая связь, при которой сервис первой среды – это именование объектов среды и наоборот. Аналогичная ситуация должна существовать и для других системных сервисов, так как любая компонентная среда определенного типа использует те же сервисы, а их связи определяют совместимость. В частности, взаимодействие компонентов, расположенных в разных серверах, поддерживают сервисы этих серверов. Если совместимость между серверными сервисами существует, то такие компоненты могут входить в состав общего компонентного приложения.

Операции над компонентами в компонентной среде

Далее рассматривается набор операторов, задаваемых математическими операциями $\{\cup, \cap, \oplus, \Diamond, -\}$ для задания и обработки компонентов в компонентной среде.

Операция добавления компонента к компонентой среде обозначается $\oplus$. Добавление компонента к среде выполняется согласно правил $C \oplus CE_1 = CE_2$,

Аксиома1. Операция добавления компонента к компонентной среде коммутативна: $C \oplus CE = CE \oplus C$.

Аксиома 2. Операция добавления компонента к компонентной среде ассоциативна: $C_1 \oplus (CE \oplus C_2) = (C_1 \oplus CE) \oplus C_2$.

Утверждение 1. Каждая непустая компонентная среда CE может быть представлена в виде: $CE = C_1 \oplus C_2 \oplus ,\ldots, \oplus C_n \oplus FW$.

Операция удаления компонента из компонентной среды обозначается знаком $\Diamond$ и подчиняется следующим правилам: $CE_1 \Diamond C = CE_2$,

Теорема 1. Для любого компонента C и компонентной среды CE выполняется равенство $(C_2 \oplus CE) \Diamond C = CE$.

Операция замены компонента задается знаком "–" и имеет вид:

$$CE.NameSpace(C_1) - C_2 = (CE \Diamond C_1) \oplus C_2.$$

Операция объединения (обозначается $\cup$) компонентных сред и выполняется согласно следующих правил:

$$CE_1 \cup CE_2 = CE_3.$$

Теорема 2. Операция объединения компонентных сред ассоциативна:

$$(CE_1 \cup CE_2) \cup CE_3 = CE_1 \cup (CE_2 \cup CE_3). \qquad (9)$$

Доказательство

$$\forall C \in (CE_1 \cup CE_2) \cup CE_3 \Rightarrow C \in CE_1 \cap C \in CE_2 \cap C \in CE_3;$$

$$\forall C \in CE_1 \cup (CE_2 \cup CE_3) \Rightarrow C \in CE_1 \cap C \in CE_2 \cap C \in CE_3.$$

Аксиома 3. Операция объединения компонентных сред коммутативна:

$$CE_1 \cup CE_2 = CE_2 \cup CE_1.$$

Операции управления созданием новых ПС из компонентов в CASE-среде:

1. Link ($C_1 \cup C_2 \Rightarrow C_3$) – объединение компонентов;
2. Conf ($C_1 \cup C_2 \cup C_3$) – конфигурация компонентов;
3. Reing (C_n) – реинженерии и др.;
4. Prove PS (CPI_i) – доказательство правильности ПС из КПИ;
5. Creat (Cj) – образовать класс компонентов и др.

Процесс построения КС включает в себя развертывание компонентов и создание компонентной среды, определение начальных компонентов и формирование множества контрактов в соответствии с функциональными требованиями к компонентной программе.

Компонентная алгебра

Для трансформации моделей к выходному коду разработана компонентная алгебра – внешняя, внутренняя и эволюционная алгебры**:**

$$\sum = \{\varphi_1, \varphi_2, \varphi_3\},$$

где $\varphi_1 = \{CSet, CESet, \Omega_1\}$ – внешняя алгебра, $\varphi_2 = \{CSet, CESet, \Omega_2\}$ – внутренняя алгебра, $\varphi_3 = \{Set, CESet, \Omega_3\}$ – алгебра эволюции компонентов.

Внешняя компонентная алгебра включает операции над компонентами среды: $\varphi_1 = \{ CSet, CESet, \Omega_1 \}$,

где *CSet* – множество компонентов;

CESet – множество компонентных сред; Ω_1– множество операций:

– $CSet_2 = Cset \oplus CESet_1$ инсталляции (развертывания);

– $CESet_3 = CESet_1 \cup CESet_2$ объединения компонентных сред ;

– $CESet_2 = CESet_1 \setminus CSet$ удаления компонента из компонентной среды.

Внутренняя компонентная алгебра имеет вид:

$$\varphi_2 = \{CSet, CESet, \Omega_2\},$$

где *Cset = {OldC, NewC}* – множества старых *OldC* компонентов и множество новых компонентов *NewC,* разработанных или преобразованных из старого к *NewC;*

OldC = (*OldCName,OldIn, CFact, OldIm, CSer*) – интерфейсы реализации старых компонентов в серверной среде;

NewC = (*NewCName, NewIn, CFact, NewIm,CSer*) – интерфейсы реализации новых компонентов в серверной среде;

Ω_2 = {*addIm, addIn, replIn, replIm},*

где *addim* – операции добавления реализации; *addIn* – добавления интерфейса; *replIm* – операция замещения реализации компонента, *replIm* – замещения интерфейса.

Алгебра эволюции имеет вид:

$$\varphi_3 = \{CSet, CESet, \Omega_3\},$$

где $\Omega_3 = \{O_{refac},, O_{Reing}, O_{Rever}\}$, в котором O_{refac} – рефакторинг, O_{Reing} – реинженерия, O_{Rever} – реверсная инженерия компонентов.

Семантика этих операций состоит во внесении изменений (замены, добавления, переименования) компонентов и/или их интерфейсов и добавления новых компонентов.

Модель рефакторинга компонентов: $M_{refac} = \{O_{кefac}, \{CSet = \{NewC^n\}\}$,

где $(CSet, O_{refac})$ – элемент алгебры эволюции;

O_{refac} = {*AddOImp, AddNIm, ReplIm, AddIn*} – операции рефакторинга:

AddOIm – добавить реализацию компонента;

AddNIm – добавить новую реализация;

AddIn – добавить интерфейс во множество C*Set*;

ReplIm – заменить реализацию компонентов в репозитории.

Операция рефакторинга предполагает реализацию компонента в составе нового типа компонента – контейнера. Согласно классификации методов рефакторинга, операция расширения интерфейса добавляет интерфейс для управления экземплярами.

Модель реинженерии компонентов: $M_{Reing} = \{O_{Reing}, \{CSet = \{NewComp^n\}\}$,

где $(CSet, O_{Reing})$ – элемент алгебры эволюции для изменения функциональности;

O_{reing}= {*rewrite, restruc, adop, conver* } – операции реинженерии, означающие:

Rewrite – переписать компонент;

Restruc – реструктуризация структуры компонента;

Adop – адаптация компонента к среде выполнения;

Conver – конвертирование компонента к другому языку.

Модель реверсной инженерии компонентов: $M_{Rever} = \{O_{Reve}, , \{CSet = \{NewC^n\}\}$,

где $(CSet, O_{Rever})$ – элемент компонентной алгебры эволюции;

O_{Rever}= {restruc, designt}– операции реверсной инженерии, включая: операцию:

Design – проектирование компонента или системы;

Restruc. – трансформации структуры системы.

Алгебра реверсной инженерии обозначает полную перестройку ПС из компонентов в некоторой среде, включая изменение ее функций.

Таким образом, операции эволюционной алгебры обеспечивают изменения некоторого компонента и системы. Основная суть этих операций – изменение имен, интерфейсов, реализаций и связей, а также алгоритма преобразования функций.

Типы отношений компонентов

К типам отношений между компонентами относятся

Отношение наследования двух компонентов определяется для входных интерфейсов (как наследование в ООП).

Отношение экземпляризации устанавливается согласно выражения $CIns_k^{ij}$ = $(IIns_k^{ij}, IntFun^i, ImpFun^j)$, задается экземпляр компонента *C*. В нем

$IIns_k^{ij}$ – уникальный идентификатор экземпляра,

$IntFun^i$ – функциональность интерфейса $CInI^i \in CIn$, $ImpFun^j$ – программный элемент, который обеспечивает реализацию $CIm \in CIm$.

Отношение контракта. Контракт между компонентами *C1* и *C2* (рис.5) описывают выражением: $Cont_{12}^{im} = (CIn_1^u, CIn_2^m, Ima_{12}^{im})$, где $CIn_1^u \in CIn_1$ – исходный интерфейс первого компонента, $CIn._2^m \in CIn_2$ – выходной интерфейс второго компонента; $Imap_{12}^{im}$ – отображение соответствия между методами, входящими в состав обоих интерфейсов с учетом сигнатур и типов данных Отношение контракта существует, если компонент *C2* имеет реализацию интерфейса CIn_2^m, выполняющего функциональность $IntFunc_1$.

Отношение связывания. Если *C1* и *C2* связаны контрактом $Cont_{12}^{im}$, то между их экземплярами $CIns_{1k}^{ij} = (IIns_{1k}^{ij}, IntFun^{i}, ImpFunj)$ и $CIns_{2p}^{mq}=(IIns_{2p}^{mq}, IntFun^{m}, ImpFun^{q})$ существует связь $Bind\ (IIns_{1k}^{ij}, IIns_{2p}^{mq}, Cont_{12}^{im})$.

Таким образом, компонентное программирование – это самостоятельный стиль программирования со своей концептуальной базой, теорией, методологией и инструментальными средствами. Его назначение состоит в следующем:

1) формализация отдельных компонентов систем из КПИ, уменьшение времени разработки и улучшения качества системы;

2) стандартное описание интерфейсов компонентов для передачи данных между ними, которые будут выполняться в некоторой общесистемной среде (IBM, JAVA IDE и др.);

3) упрощение процессов моделирования системы за счет формализованных моделей, интерфейсов, конфигурационной сборки компонентов в разные структуры.

Компонентная модель СМ конструируется из компонентов и КПИ, которым сопоставляются реальные функции или методы с соблюдением требованиям к системе.

На общем уровне модель СМ состоит из совокупности компонентов и их интерфейсов или контрактов, которые после конфигурации обеспечивают их взаимодействие между собой в среде функционирования.

Глава 2. БАЗОВЫЕ МОДЕЛИ И МЕТОДЫ МОДЕЛИРОВАНИЯ СИСТЕМ

Моделирование ПС и их семейств выполняется средствами компонентного программирования в части построения отдельных КПИ и их сборки в конфигурационный вариант ПС или СПС на основе спецификаций компонентов в пространстве проблем. Это описание базируется на подходах FOD (Feature-Oriented Development) построения МХ ПрО, MDD (Model Driven Development) и MDA (Model Driven Architecture) и др. Архитектура системы может выполняться с помощью платформо-независимой модели PIM (Platform Independent Model) и платформо-зависимой модели PSM (Platform Specific Models). Концепции двухуровневого моделирования архитектуры MDA (Model Driven Architecture) и отображения PIM↔PSM соответствует концепции отображения пространства проблемы в пространстве решений [1–8] .

Модель MDD (Model Domain Development)

Базируется на унифицированном языке моделирования UML с графической формой представления семантики. В MDD определяется система с четко определенной семантикой, и ее артефакты могут быть преобразованы, трансформированы к коду реализации (аналогично тому, как код JAVA компилируется в байт-код). При MDD определяется модель приложения (Application Model), члены которого имеют общие функции, но зависят от платформы реализации компонентов и данных. Преимущество применения MDD для построения членов семейств заключается в том, что "управление" точкой вариантности для платформы выполняется автоматически путем трансформации моделей PIM – PSM. Члены семейства различаются не только на уровне платформы реализации, но и на уровне функций ПС, требований, а также достижения качества ПС.

Приспособление подхода MDD к потребностям приложения может быть выполнено путем добавления к модели домена вариантных *точек* для управления

альтернативными концепциями вариантной конфигурационной структуры. Это особенно необходимо при их уточнении или изменении некоторых КПИ в ПС. Выбор разных концепций представления модели домена будет предопределять появление разных прикладных моделей ПС, которые, при генерации, автоматически трансформируются к модели MDD.

Модель MDA (Model Driven Architecture)

В основе этой управляемой архитектуре модели лежит идея разделении этапов моделирования и последующей реализации приложения на конкретной платформе. Сначала с помощью специальных средств создается общая и независимая от способов реализации модель приложения, а затем осуществляется ее реализация в некоторой среде разработки. Главная цель этой модели – функциональная совместимость, нереносимость и повторное испольование.

Архитектура MDA возникла на основе наличия ряда стандартов и технологий. Концептуальная основа MDA – спецификации CORBA, ООП, стандарт CIM, языки UML, MOF, MS, IBM.

MDA – это новый виток эволюции технологий программирования в плане описания процесса разработки в целом и с использованием современных средств представления, что позволяет автоматизировать создание приложений. MDA станет промышленным стандартом.

В классической модели MDA действует схема (рис. 1,*а*). С учетом потребностей доменов эта схема модифицируется (рис.1,*б*).

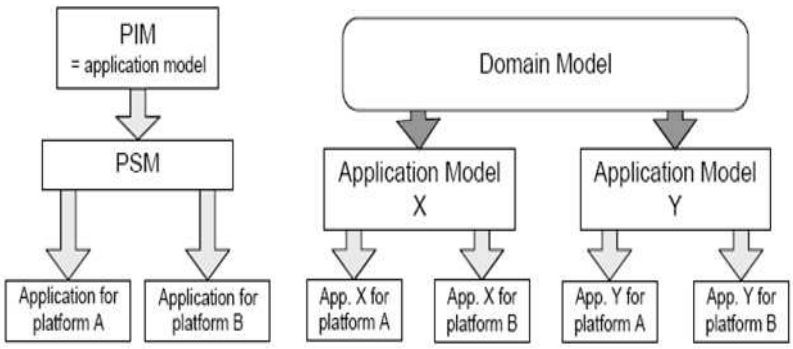
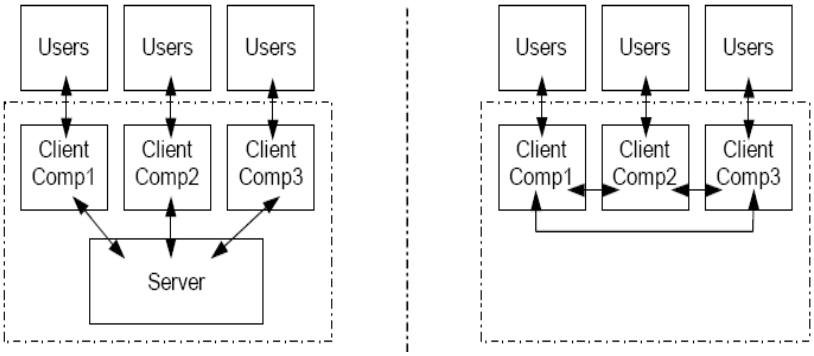


Рис. 1. MDA–два варианта (*а*, *б*) семейств систем

Фактически спецификация ПС соответствует подходу MDD и определяет модель семейства ПС из приложений, члены которого имеют общие функции, но различаются платформами реализации. Как это видно на рис. 2, где показаны два способа реализации информационного взаимодействия прикладных приложений.



а **б**

Рис.2. Два подхода к реализации взаимодействия приложений: *а*– централизованный доступ, *б*– взаимодействие парами

Выбор альтернативных платформ связан с "точкой вариантности" в семействе, которая находится над моделью ПС и она невидимая на уровне конкретной ПС из семейства СПС.

Преимущество применения MDD для построения члена семейств заключается в том, что "управление" точкой вариантности платформ выполняется через трансформацию PIM↔PSM.

Члены семейства различаются не только на уровне платформы реализации, но и на уровне функций ПС, требований к качеству и инфраструктуре, а также активов, которые реализуют альтернативные концепции. Приспособление подхода MDD к потребностям ГП может быть выполнено путем добавления ***мо**дели ПрО*, которая будет определять точки, где могут быть выбраны альтернативные концепции.

Выбор разных концепций представления модели домена (ПрО) предусматривает появление разных прикладных моделей ПС, которые трансформируются автоматически в соответствии с подходом MDD.

Таким образом, преимущества этого подхода следующие: независимость модели от средств разработки; возможность переноса из одной операционной системы в другую и экономия ресурсов приложения одновременно для нескольких платформ; создание частей приложения.

Модели систем для разных платформ PIM и PSM

Проектирование ПС проводится, как правило, с помощью стандартных моделей и моделей ЖЦ (спиральной, водопадной, итерационной и др.). К стандартным моделям проектирования относятся MDA, MDD, PIM, PSM и др. Модели MDA, MDD рассмотрены выше. Здесь дается характеристика моделей PIM и PSM, которые могут адаптироваться к разным платформам компьютеров.

Платформенно-зависимая модель PSM (Platform Specific Model**)** задает состав, структуру, функциональность системы применительно к конкретной платформе. Модель платформы задает технические характеристики, интерфейсы, функции платформы. Она используется при преобразовании модели PIM (Platform Independent Model, **платформенно-независимая модель**) в модель PSM.

В зависимости от уровня детализации платформы, модели могут содержать сведения о различных функциональных частях системы. Уровень бизнес-логики содержит описание основного функционала приложения, обеспечивающего

исполнение его назначения. Уровень данных описывает структуру данных приложения, используемые источники, форматы данных, технологии и механизмы доступа к данным. Для приложений .NET чаще всего применяются возможности ADO.NET. Уровень пользовательского интерфейса содержит состав форм приложения, функциональные элементы управления. Легкость автоматизации этого уровня зависит от того, насколько унифицированы пользовательские операции. Если удается создать типовые шаблоны элементов управления для основных операций, то появляется возможность автоматической генерации форм и их содержания при создании приложений.

Вычислительно-независимая модель

Вычислительно-независимая модель **(**Computation Independent Model, CIM**)**, структурная схема которой приведена на рис. 9, базируется на моделе PIM для ее преобразования к платформенно-зависимой модели с использованием языка моделирования UML.

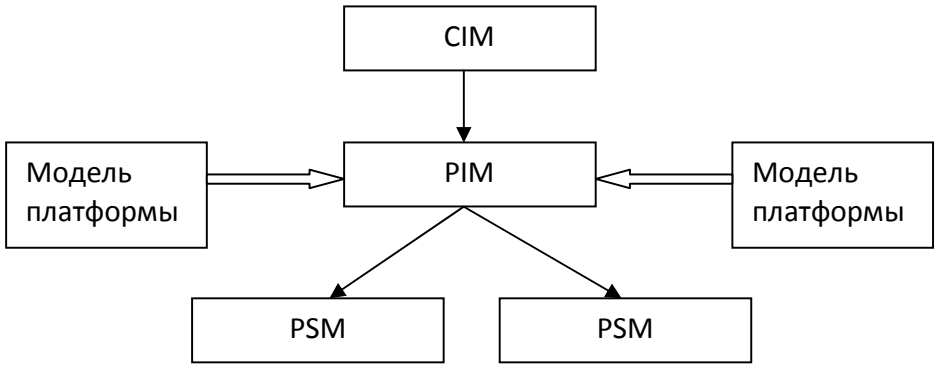


Рис. 3. Структурная схема CIM

Эта модель включает в себя разработку общих требований к системе, создание общего словаря понятий и описание окружения, в котором система будет функционировать. Сущности и понятия, описываемые в модели CIM, должны

тщательно анализироваться и отрабатываться. Модель CIM должна отображать общую концепцию системы, используемую для программирования элементов приложения. Преобразование CIM в платформенно-независимую модель PIM осуществляется средствами языка UML. CIM включает в себя элементы, описывающие бизнес-логику, общую структуру системы, состав и взаимодействие подсистем, распределение функциональности по элементам и требования к пользовательскому интерфейсу. При этом модель PIM включается во все автоматизированные среды разработки приложений, использующих модель MDA.

Переход к платформенно-зависимой модели определяется числом программных платформ, на которых будут функционировать элементы приложения. Отмечаются случаи, когда приложение (или его составные элементы) должны работать на нескольких платформах одновременно. Тогда модель PSM создается путем преобразования модели PIM к модели другой платформы. На этапе создания модели PSM разработка приложения осуществляется с помощью модели MDA, которая преобразуется к коду другой платформы.

Преобразование моделей PIM→PSM

На этом этапе общее описание системы на языке UML включает выполнение следующих действий:

1. разработка схемы преобразования (mapping)
2. маркирование (marking)
3. трансформация (transformation).

Для каждой платформы создается собственная схема преобразования, зависимая от возможностей платформы (формата, языков, средств обработки и др.). Схема преобразования затрагивает как содержание модели (совокупность элементов и их свойств), так и механизмы ее представления (метамодель, используемые типы данных, форматы данных платформы). В схеме преобразования свойствам метамодели и

элементам модели PIM ставятся в соответствие свойства метамодели и элементы платформо-зависимой модели PSM.

При преобразовании PIM→ PSM моделей может использоваться несколько схем преобразования. Для их связывания применяется марка (mark), как самостоятельная структуры данных, принадлежащая не моделям, а схемам преобразования, содержащая информацию о созданных связях (рис. 4). Процесс задания марок называется маркированием. Простейшим случаем является элемент модели PIM, который соединяется маркой с одним элементом модели PSM. В более сложных случаях один элемент модели PIM может иметь несколько марок из разных схем преобразования этой модели. В процессе маркирования используются сведения о платформах, которые содержатся в модели платформы.

Процесс преобразования моделей заключается в переносе маркированных элементов модели и метамодели PIM в модель и метамодель PSM. Процесс преобразования документируется картой переноса элементов модели и метамодели.

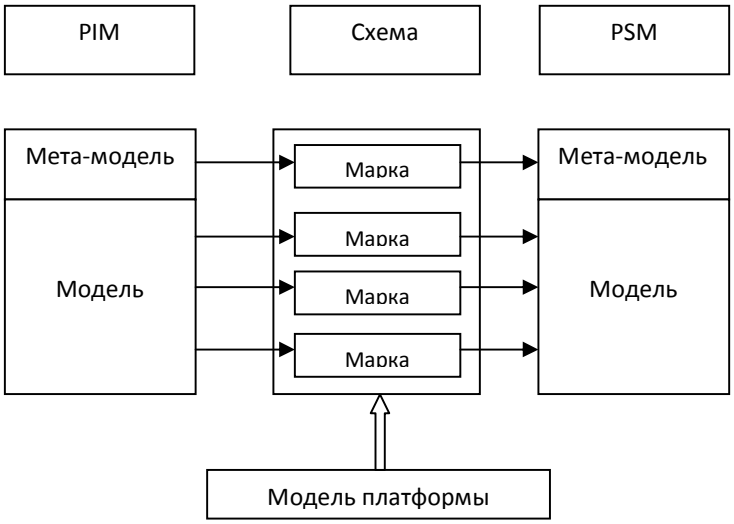


Рис. 4. Схема преобразования метамоделей

Моделирование систем в языке UML

Процесс построения системы в среде UML проводится на этапе анализа ПрО и проектирования. В процессе анализа системы создается концептуальная объектная модель ПрО в виде диаграмм прецедентов. Она уточняет внешнее функциональное поведение системы. На этапе проектирования разрабатываются диаграммы классов и взаимодействия. Они образуют каркас системы. Для определения поведения классов объектов используются диаграммы состояний и деятельности. Размещение объектов фиксируется компонентными диаграммами, а расположение программных модулей в узлах (компьютеров) сети – диаграммами развертывания.

С помощью UML графических диаграмм описываются три основные объектные модели системы:

1. структурная (статическая) для описания структуры сущностей системы, включая классы, интерфейсы, отношения, атрибуты
2. модель поведения (динамическая) описывает поведение объектов системы, включая взаимодействие и изменение состояний отдельных компонент или всей системы
3. модель функционирования описывает операции функционирования системы

Этим моделям соответствуют структурные, поведенческие и аннотационные сущности, а также типы отношений: зависимость, ассоциация, обобщение, реализация. Есть также их вариации, например уточнения, трассировки, включения и расширения, агрегация и композиция.

Для проектирования систем используются следующие виды диаграмм:

- диаграмма вариантов использования
- диаграмма классов
- диаграммы поведения, к которым относятся диаграммы состояний, деятельности, взаимодействия
- диаграммы последовательности и кооперации

- диаграммы реализации, к которым относятся диаграммы компонентов и развертывания.

Моделирование изменяемых систем

В SEI USA (2005) предложен метод SPLE (Software Product Line Engineering) [1] для изготовления ПП – ProductLine/ProductFamily, основанный на модели вариабельности MF и модели конфигурационной сборки готовых ГОР (artifacts, components, reuses, assets, services и др.) по заказу потребителей. Клаус Похл определил понятие вариабельности в SPLE, как модели FM для элементов системы, помечаемых вариантными точками. *Вариабельность* – это свойство продукта (системы) к расширению, изменению, приспособлению или конфигурированию с целью использования в определенном контексте и обеспечения последующей его эволюции. Модель FM формируется в процессе разработки ПП аналитиками и разработчиками и включает общие функциональные и нефункциональные характеристики элементов системы, которые являются готовыми reuses (product configuration) . К главным аспектам обеспечения вариабельности продукта системы отнесены:

– моделирование вариабельности на уровне артефактов с вариантными характеристиками;

– управление (планирование, контроль и регуляция) конфигурацией системы по модели системы и модели вариабельности FM.

Множество видов одной характеристики, каждая из которых относится к *вариантной характеристике*, образует коллекцию, члены которой присоединяются к ядру в точках вариантности.

Точка вариантности – это место в созданной системе, по которой осуществляется выбор элементов вариантов системы. Вариантная характеристика определяет количество точек вариантности системы, присоединяемых к ядру системы.

Основу концепции изменяемости/вариабельности ПП составляют внешние характеристики и свойства функций системы, которые могут изменяться. По модели системы и MF создается конфигурационный файл системы для запуска и выполнения системы. Совокупность систем и подсистем с общим множеством характеристик и требований к ним образуют семейство СПС. Каждый члены семейства проверяются и верифицируются и при обнаружении ошибок в него вносятся измененяемые элементы или удаляются.

Метод сборки КПИ в современных средах

При сборке программных ресурсов и КПИ возникают проблемы сборки большого количества программных ресурсов и взаимодействия между ними. Для сборки большого количества программных ресурсов используются **системы автоматизации сборки**. Для обеспечения взаимодействия – различные **интерфейсы**.

Поэтому методы сборки программных ресурсов можно разделить на несколько больших групп, по способу их организации и решаемым задачам.

Ключевым понятием во всех этих группах – понятие интерфейса, посредством которого обеспечивается взаимодействие.

Основные подходы к реализации метода сборки готовых ресурсов в современных средах.

1. Метод сборки программных ресурсов в рамках одного языка программирования, понимая под ресурсами класс, пакеты, модули, библиотеки, то речь идет о системах автоматизации процесса компиляции и сборки данных ресурсов для конкретной архитектуры и операционной системы. Кроме того, создается API (Application Programming Interface) для обеспечения четко определенных методов взаимодействия программных ресурсов. API – набор объявлений функций, протоколов и инструментов для сборки приложений.

2. Для ресурсов, представленных на разных ЯП, можно использовать ABI (Application Binary Interface) – интерфейс между уже скомпилированными программными модулями, который отличается от API тем, что является интерфейсом на уровне машинного кода. Путем использования единого ABI, объединяются в исполняемый файл или библиотеку модули, написанные на разных языках программирования.
3. ABI является сильно архитектурно зависимым, для объединения программных ресурсов, написанных на разных ЯП, существуют методы объединения таких ресурсов через компьютерные сети. Эти методы являются самыми гибкими и позволяют объединять модули, написанные на языках, компилируемых в машинный код конечных архитектур (C++, Fortran, Go), динамически компилируемых из промежуточного байт-кода в машинный код (Java, .NET), а также написанные на интерпретируемых языках (Perl, PHP). Однако, такой подход менее производителен, хотя и более гибок, чем объединение модулей через ABI. Данный подход реализуется с использованием технологий RPC/RMI, веб сервисов, облачных вычислений, грид-вычислений. Общее понятие для программного обеспечения, обеспечивающего распределенное взаимодействие программных компонентов – middleware.

Конфигурационная сборка ГОР

К операциям конфигурационной сборки отнесены операции выполнения функций управления вариантами с использованием ГОР, которые размещаются в репозитории с учетом требований и предикатов выбора элементов ГОР [23, 24].

К операциям управления конфигурацией ПС в СПС относятся:

- идентификация конфигурации, ее элементов и данных;
- управление процессом изменений ГОР;
- изменение модели вариабельности под новые требования;
- оценка уровня вариабельности ПС и СПС.

Для управления артефактами ПС и СПС и их вариабельностью создается, так называемая *модельная среда* конфигуратора, в которую входят:

- процесс сборки ГОР и артефактов системы;
- схемы формального описания артефактов:
- представление модели FM с отмеченными вариантными точками;
- БД, включающая требования, архитектуру, набор базовых ГОР, тесты;
- Конфигуратор для объединения артефактов в ПС и СПС.

При управлении конфигурацией собираются данные для проведения таких стандартных операций, как *отчетность и аудит конфигурации* на установление запланированной функциональности СПС. Конфигуратор собирает требуемые артефакты и ГОР в структуру СПС. Одним из шагов сборщика является процесс компиляции исходного кода, где файлы превращаются в промежуточный код или в код выполнения. Процесс связи представляет собой замену адресов функций внешними характеристиками библиотек и реальными адресами, используемыми в выполняемой программе. Сборщик готовых элементов – это компонент Конфигуратора, который объединяет функциональные элементы и их интерфейсы по модели MF. Один вариант сборки реализован на фабрике сайта http://programsfactoty.univ.kiev.ua, в AppFab современных систем общего назначения (IBM, .Net, Intel и др.).

Сервисно-компонентная архитектура (SCA)

Модель SCA предназначена для работы с компонентами, которые имеют стандартизированные интерфейсы и взаимодействуют друг с другом по сетевым протоколам.

Архитектура SCA обеспечивает доступ к сервисным компонентам и определяет зависимости между ними через аппарат ссылок. Элементы SCA могут компоноваться и обмениваться данными друг с другом, пересылая сервисные объекты данных, подготовленные в необходимом виде. Этот интерфейс включает определение метода получения и установления свойства данных.

Модель SCM представляет собой обобщение объектно-компонентной модели семейства программных продуктов. В ней каждый член является: системой удаленных reuses, которые обмениваются гетерогенными данными и предоставляют сервисы реализации с множеством общих свойств; композицией сетевых сервисов, которые поддерживают некоторый деловой процесс. Данная модель ориентирована на обеспечение адаптивности ПС к переменным условиям использования запросов к функциям и обрабатываемым ими гетерогенных данных. В интересах этой поддержки для программной реализации ПС используются механизмы сервисных объектов данных и сервисов доступа к ним. Они позволяют размежевать код ПС и код доступа к обрабатываемым данным. В состав модели SCM входят подмодель абстрактных сервисов, интерфейсная подмодель и подмодель объектных ресурсов.

Сервер приложений Java Enterprise Edition — это реализация системы, которая соответствует сфецификациям Java EE и поддерживает следующие сетевые средства:

1. динамическая генерация серверных страниц (Java Server Pages);
2. компоненты повторного использования (Enterprise Java Beans);
3. транзакционность;
4. служба обмена сообщениями (Java Message Queue) и другие сервисные технологии.
5. сетевые службы (веб-сервер, сервлеты)
6. Java Message Service (JMS) — сервис, позволяющий обмениваться сообщениями между сервером и компонентами

Сетевая служба идентифицируется с помощью универсального ресурсного идентификатора URI, ее ресурсы описываются в XML-файле согласно специпикации Web Services Description Language (WSDL), которая разрабатывается World Wide Web Consortium (https://www.w3.org). Доступ к ресурсам осуществляется через протокол SOAP следующим образом:

1. клиент (компонента) формирует на своей стороне запрос (request) на языке XML, который затем передается с помощью сетевых протоколов (HTTP, SMTP) на сервер
2. сервер принимает запрос от клиента, обрабатывает его согласно бизнес-логике и формирует ответ (response), который отсылается клиенту
3. клиент получает ответ сервера, обрабатывает его и продолжает свою работу

Формат сообщений (запросов и ответов) определяется в WSDL-файле с помощью XSD-схем. Поддерживаются как простые типы (строки, целые числа, числа с плавающей точкой, логические типы), так и составные, которые можно формировать из последовательностей и вложений простых типов. Вместо методов классов в сетевые сервисах используются операции, которые определяются входным и выходным сообщениями.

Таким образом, для рассмотренного класса моделей проектирования конкретных приложений необходимо провести декомпозицию функций домена с помощью компонентов, их описать в ЯП и занести в библиотеку проекта. Затем подбираются готовые функциональные или системные компоненты, которые могут находиться в библиотеках общего пользования Интернет, в среде системы IBM или JAVA IDE.
Из этого набора компонентов и КПИ проводится конфигурационная их сборка, тестирование и получение выходного файла для разветрывания в заданной среде функционирования приложения или системы.

Глава 3. ПОДХОДЫ В РЕАЛИЗАЦИИ КПИ В СОВРЕМЕННЫХ СРЕДАХ

Понятие КПИ

Внешние спецификации повторно используемых компонент (КПИ) и информационных ресурсов предоставил консорциум OMG в документе Reusable Asset Specification (RAS). Назначение этих спецификаций – упрощение процедуры объединения информации о КПИ брокером для их накопления, распространения, поиска и использования разными категориями пользователей. Для поддержки механизмов реализации КПИ предлагается модель структуры КПИ с возможностью задания вариантов (рис. 5)

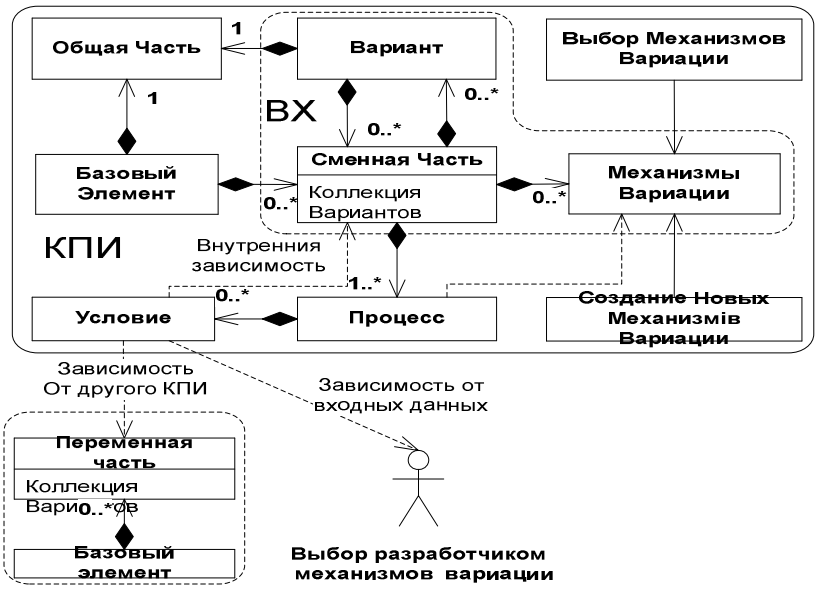


Рис. 5. Модель структуры КПИ

Эта модель представлена диаграммой классов в ООП, но это не означает, что все КПИ системы должны быть написаны в этом подходе. На данной диаграмме базовый элемент является основным классом, который представляет КПИ и взаимодействует с ПС. Другими словами, объектом класса является базовый элемент, который является КПИ. Базовый элемент включает в себя общую часть и необязательную переменную

часть. Если базовый элемент не имеет переменной части, его используют непосредственно путем включении в конкретную ПС.

Переменная часть – это локализация изменений в конкретной точке КПИ. Отсутствие общей части означает, что все КПИ должны быть разработаны для каждой ПС индивидуально, а это противоречит концепции КПИ. Последующие объяснения относительно внутренней структуры КПИ, а также механизмов обеспечения его вариабельности ВХ, будем сопровождать примером, в котором в качестве КПИ будут выступать элементы управления графического интерфейса пользователя (GUI), которые за внешним видом и содержанием выполняемых действий могут адаптироваться к требованиям пользователя.

Следовательно, для элемента управления общей частью может выступать механизм графического вывода на экран и для всех вариантов элементов управления он будет одинаковым. Переменную часть будет представлять набор параметров, которые модифицируют поведение и вид элемента управления. Каждая переменная часть КПИ может содержать несколько механизмов вариации, которые поддерживают создание или выбор вариантов. Механизм вариации может быть интегрирован непосредственно в базовый элемент. Примером такой интеграции является способ изменения параметров элемента управления (размер, цвет и т.п.). Переменная часть задает точку вариации, а механиз мы вариации выступают как средство конкретизации КПИ. Во многих случаях переменная часть включает два механизма: один для создания нового варианта, а другой – для поддержки существующего, который выбирается из репозитория модельной среды. Механизм вариантности включает все необходимое для внесения любых изменений. В случае его отсутствия соответствующие КПИ не будут широко использоваться в ПС, а это будет

свидетельствовать о том, что действия процесса разработки КПИ и соответственно ПС с использованием КПИ должны быть скоординированы.

Вариации выступают как средство конкретизации КПИ. Во многих случаях переменная часть включает два механизма: один для создания нового варианта, а другой – для поддержки существующего, который выбирается из репозитория или библиотеки модельной среды. Механизм вариантности включает все необходимое для внесения любых изменений. В случае его отсутствия соответствующие КПИ не будут широко использоваться в ПС, а это будет свидетельствовать о том, что действия процесса разработки КПИ и соответственно ПС с использованием КПИ должны быть скоординированы.

В примере элемент управления механизма вариантности предоставляет возможность разработчикам ПС изменять его свойства, а разработчикам КПИ – добавлять новые. Результатом применения механизма вариантности к переменной части КПИ является вариант, который содержится в общей части и может быть включен в переменную часть. Обычно, появление нового варианта предопределяется применениям нового механизма вариантности.

Например, добавляя новый механизм связывания (building) элемента управления типа список. Получается новый вариант поведения базового элемента. Элемент управления должен работать во всех ПС, использующих этот механизм. Созданный вариант для одной ПС может быть использован в других ПС путем выбора механизма вариантности.

Для поддержки использования механизмов вариантности при построении ПС в ПС предлагается в описание переменной части включать информацию об использовании условий ее адаптации к потребностям ПС. Процесс применения

использования ПС будет объяснять, как использовать механизмы вариантности при для создании или выборе варианта ПС.

Описание процесса может содержать условия использования. Если условий нет, то он является безусловным и выполняется каждый раз, когда создается новый вариант ПС из базовых компонентов. Условия использования переменной части определяют зависимость от коллекции вариантов, которые влияют на конкретный продукт. Коллекция может быть дополнена вариантами из другой переменной части, или задается разработчиками ПС и, таким образом, будет отображать альтернативу, которая избрана для решения проблемы в конкретной ПС.

Возможны разные виды зависимости переменных частей, а именно: внутренняя зависимость – это зависимость одной переменной части КПИ от его другой переменной части. Внешняя зависимость – это зависимость от переменной части другого базового КПИ.

Внешние связи вариантов

Для организации внешних связей вариантов могут использоваться такие механизмы вариации, как макросы, условная компиляция кода, конфигурирование кода, генерация кода статических библиотек и динамически присоединенных библиотек. Рассмотрим их более детально.

1. **Макрос**. Это программный объект, который при обработке «разворачивается» в последовательность действий или команд. Макросы полностью или частично поддерживаются в Ada, C, C++, D, Erlang, Haskell, Common, Lisp, Nemerle, Ruby и др. Функциональность макроса разделена на две части. Общая часть находится в теле определения макроса, а варьирующиеся часть находится в разных частях

кода и содержит варьирующиеся параметры, как в примере ниже:

```
#include <stdio.h>

#define SUM(x, y) (x + y)

int main(int argc, char *argv[])
{
        int a = 5;
        int b = 10;
        int sum = SUM(a, b);
        printf("%d\n", sum);
}
```

2. **Условная компиляция кода**. Этот механизм вариации обеспечивает возможность контролировать включение или исключение сегмента собранного кода с применением параметров компиляции. Пример:

```
#ifndef TestMode
   #define TestMode
#endif

void OnStart()
{
   #ifdef TestMode
      Print("Test mode");
   #else
      Print("Normal mode");
   #endif
}
```

3. **Конфигурирование.** Исходный код каждого варианта находится в отдельном файле, а для выбора альтернатив используется инструмент управления конфигурацией.
4. **Генерация.** Генератор конвертирует заданную спецификацию, написанную на языке предметной области (DSL, например) или ЯП, который генерируется в выходной код. При этом исходный код, в свою очередь, может быть использован как часть продукта. Например, графический интерфейс пользователя может быть сгенерирован (Silverlight) из графической или текстовой спецификации.
5. **Организация статических библиотек**. Статические библиотеки содержат набор внешних функций, которые можно использовать в приложении после его компиляции. При этом код из библиотеки интегрируется в выполняемый файл.

Вариабельность достигается путем выбора соответствующей библиотеки перед ее присоединением. В Unix-системах, например, библиотеки в основном располагаются в /lib /usr/lib/usr/local/lib, а в Windows-системах – в \System32 и имеют тип “ocx” или “lib”.

6. **Динамически присоединяемые библиотеки**. Данный механизм вариантности основан на использовании DLL-библиотек, которые можно поместить в адресное пространство программы во время ее работы. Управление вариабельностью достигается путем реализации разных вариантов кода в отдельных библиотеках и написания исходного кода, который загружает определенные варианты в приложение во время его работы. Следовательно, знать все возможные варианты во время разработки программы не обязательно. В Unix-системах, например, они называются совместно используемыми объектами (shared objects) и имеют расширение “So”. Mac OS X, которые унаследовали от BSD работу с библиотеками. Они находятся в директориях “bundles” и имеют тип “dylib”.

Внутренние связи вариантов

При описании механизмов, предназначенных для внутренней связи, будем считать, что ПС создается в парадигме ООП, где каждый компонент ПС представлен классом с собственными данными и методами их обработки, а следовательно, все варианты, с которыми пользователь имеет дело, при реализации, представляют собой не что иное, как простые манипуляции классами или другими конструкциями используемого ЯП.

Для реализация вариабельности ПС могут быть выполнены операции композиции, наследования, параметризации, загрузки операторов и методов, динамическая загрузка классов, рефлексия, шаблоны (паттерны), архитектуры и проектирование. Далее приводится краткая характеристика каждой из этих операций.

1. **Композиция.** Механизм вариабельности устанавливается на основе связи между объектами типа «part» и размещении общей части в делегирующем классе, а переменной – в делегированном. На рис.2 представлена UML схема и соответствующий фрагмент кода, который описывает поведение системы в случае изменения расположения панели с элементами управления в окне. Окно не содержит логики относительно того, как именно отображать и группировать разные элементы управления. Окно делегирует эту функцию Панели. Команда панель – Горизонтально = true является общей частью, которая выполняется одинаково в не зависимости от типа панели и элемента управления (ЭУ). Переменная часть находится в панели переписанного ЭУ() и содержит соответствующую логику расположения и отображения элемента для разных ЭУ. На данном примере связь между классами Панель и *ЭУ* является примером агрегации объектов – панель может содержать несколько *ЭУ*; связь между классами Окно и Панель – примеры композиции объектов, где к отношению «part-of» добавляется условие, что панель принадлежит только одному окну и уничтожается вместе с ним. Обобщением обоих отношений (агрегации и композиции) является ассоциация.

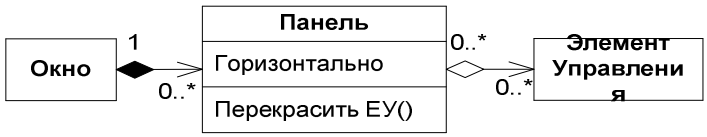


Рис.6. Композиция

2. **Наследование**. Механизм базируется на таком принципе ООП, как наследование через установление отношений «наследник-предок» с возможностью порождать один класс объектов от другого с сохранением всех свойств и методов класса-предка, а, следовательно, и руководить объектами, не имея полной информации о них. Этот механизм разделяет общую функциональность в базовом классе и дополненного в классе наследника. Наследование имеет следующие виды категорий.

- *Множественное наследование* (рис.3). При таком механизме реализации вариабельности новый класс наследуется от нескольких базовых.

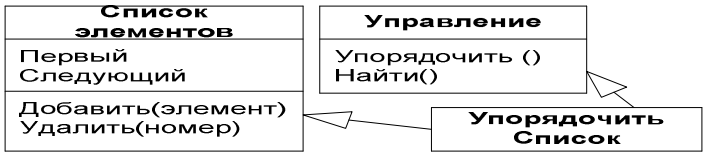


Рис. 7. Множественное наследование.

- *Стандартное наследование* (рис.4). Предусматривает создание иерархической структуры кода, который содержит более общую функциональность на верхнем уровне (в базовом классе ЕУ) для последующего использования ее на нижнем уровне, дополненном варьирующейся функциональностью (классы наследники). Таким образом, классы становятся более специализированными.

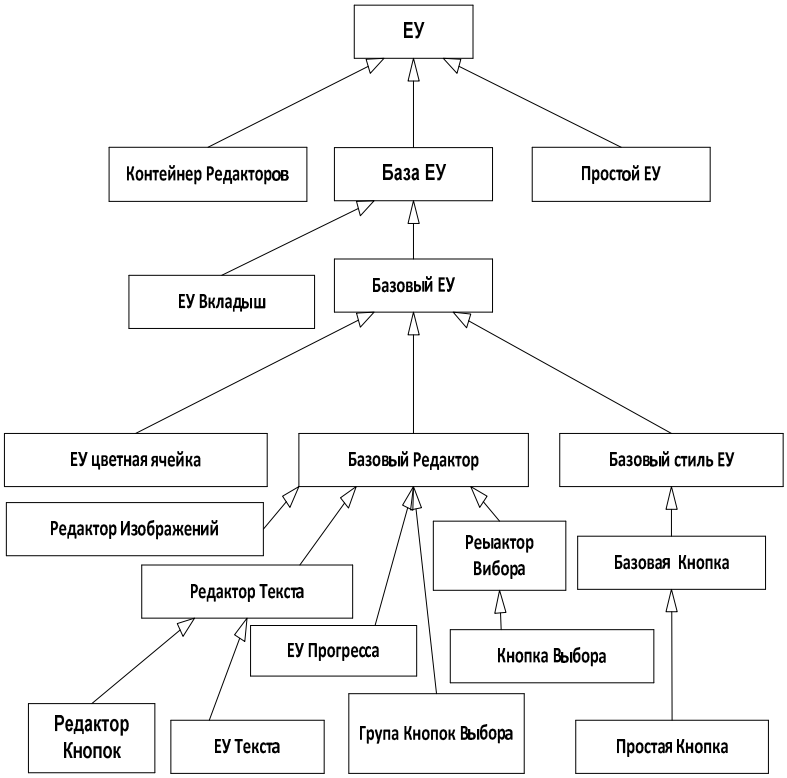


Рис. 8. Пример стандартного наследования классов

- *Наследование с виртуальными методами.* Механизм, подобный стандартному наследованию, но методы базового класса могут быть перегружены классом наследника.

  На данном рисунке класс управления содержит алгоритмы сортировки и поиска и не зависит от элементов, над которыми выполняется операция. Такой тип наследования поддерживается в языках программирования (ЯП): Eiffel, C, Dylan, Python, Perl, Curl, Common Lisp (через CLOS), OCaml.
- *Наследование, основанное на смесях*. Смесь (mix) – элемент ЯП (абстрактный подкласс или модуль), который реализует вариант поведения родительского класса или модуля. При реализации вариабельности он используется для уточнения поведения других классов и не предназначен для порождения самостоятельно используемых объектов ЯП, которые поддерживают смеси – Flavors, Smalltalk, Beta, CLOS, Ruby, D, Python, Scala.
- *Наследование, основанное на объектах* или шаблоне делегирования (delegation pattern). Это механизм вариации, который, в отличие от других, применяется на уровне объектов, а не классов. Это способ, которым объект внешне демонстрирует некоторое поведение, но фактически передает ответственность за выполнение соответствующего действия другому объекту, как в следующем примере, где *Принтер* делегирует объект типа Принтер.

```
class PrinterWorker {//делегированный
    void print() {/*..*/}
}
class PrinterMain {// делегирующий
    PrinterWorker printer = new PrinterWorker();
    void print() { printer.print();}
}
```

3. **Параметризация.** Параметризованным типом может быть класс, структура, интерфейс или метод, в котором есть ссылка на место конкретизации параметров одного или нескольких типов, которыми он оперирует или которые сохраняются

(конструкция вроде placeholders (type parameters)). Этот механизм вариации предусматривает наличие общего кода для работы с разными типами данных. Яркими представителями являются шаблоны и универсальные шаблоны. Они являются синтаксически и функционально подобными, но отличаются внутренним механизмом реализации. Пример использования включает переменные: List<Панель> и List<ЭлементУправления>. Они могут сохранять массивы объектов любых типов, порожденных от типов Панели и ЕУ соответственно. При этом алгоритм работы остается не переменным.

4. **Перегрузка**. Сущность этого механизма заключается в вариации типов, над которыми выполняется оператор, или за которыми вызывается функция. То есть ЯП предоставляет возможность повторного использования имени для операции над другими типами. Выбор варианта осуществляет программная среда ЯП. Применяется к процедурам, функциям или операторам. Языки, которые поддерживают перегрузку: Ada, C++, C#, D, Erlang, F#, Groovy, Java, JavaScript, Haskell, Common Lisp, Nemerle, Scala, VB.NET, Delphi, Ocaml. Язык Java не поддерживает перегрузку операторов, но выполняет внутреннюю перегрузку оператора “+” для сочетания строк. В C#, например, можно перегружать методы, унарные и бинарные операторы. Пример перегрузки методов в языке Java:

```
class Overload {

    // Первая версия
    void ovlDemo() {
        System.out.println("Без параметров");
    }

    // Перезагрузка метода ovlDemo для одного параметра типа int
    // Вторая версия
    void ovlDemo(int a) {
        System.out.println("Один параметр: " + a);
    }
}
```

5. **Динамическая загрузка классов.** Сущность механизма заключается в том, что класс загружается в память лишь тогда, когда он действительно начинает использоваться, как реализовано, например, в виртуальной машине языка Java. Для управления этой функциональностью используется (MainClass.class.getClass Loader ();)
6. **Отображение**. Механизм, основанный на способности программы модифицироваться, имеет собственное поведение. Такие манипуляции достигаются благодаря делению ПС на базовое и цель уровня. Цель уровень предоставляет информацию об избранных системных свойствах. Поскольку базовый уровень, который включает логику программы, ориентирован на цель уровня, на изменение цели уровня и влияет на поведение базового уровня. В СПС общая функциональность расположена на базовом уровне и может меняться во время работы в зависимости от конфигурационной информации, которая используется на уровнях.
7. **Шаблоны** (или Паттерны) архитектуры и проектирования (patterns) могут использоваться в СПС, потому что большинство из них предоставляют готовые решения для управления вариабельностью.

Рассмотренные программные механизмы структур, шаблонов, операций композиции и наследования для проектирования вариантных приложений из компонентов покрывают все задачи практического моделирования приложения из компонентов и их классов для некоторой проблемной области. Эти средства являются наиболее используемыми и в современных языках программирования объектного типа и могут использоваться при формировании новых приложений и их вариантов.

Глава 4. РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ НА ЯЗЫКЕ JAVA

В главах 2 и 3 были рассмотрены современные модели и методы моделирования сложных систем. Наиболее распространенной и перспективной из них разработки на языке Java является сервисно-компонентная модель (SCA). Сервисно-ориентированная модель (СОА) позволяет объединять множество слабо связанных компонент с одну сложную систему, используя КПИ и средства Java EE (сообщения, транзакционность, взаимодействие и др.).

В данной главе описывается применение СОА модели в разработке реального ПО. Автором было создано приложение на языке Java, которое собирается в WAR-файл и запускается на сервере приложений Glassfish. Графический интерфейс приложения представлен окном в интернет-браузере, в котором пользователю доступны функции регистрации, авторизации, создания проектов, пакетов и классов, компиляции и запуска. Для обеспечения этих задач автором реализованы следующие веб-сервисы:

1. DataService – для работы с проектом пользователя (сохранение, сериализацию/десериализацию);
2. GitService – для работы с системой контроля версий Git;
3. RunService – для компиляции и запуска проекта пользователя;
4. TreeService – для отображения дерева проекта (проект, пакеты, классы);
5. UserService – для регистрации и авторизации пользователя.

Эти сервисы описываются в приводимом ниже приложении.

Анализ используемых технологий

Enterprise JavaBeans

Для написания и поддержки серверных компонентов ПО используюются средства Enterprise Java Beans (EJB). Спецификация компонента отражает бизнес-логику. Является частью Java Enterprise Edition (Java EE).

Каждый EJB-компонент является набором Java-классов со строго регламентированными правилами именования методов (верно для EJB 2.0, в EJB 3.0 за счет использования аннотаций выбор имён свободный). Компоненты бывают трёх основных типов:

1. объектные (Entity Bean) — перенесены в спецификацию Java Persistence API
2. сессионные (Session Beans), которые бывают
    a. stateless (без состояния)
    b. stateful (с поддержкой текущего состояния сессии)
    c. singleton (один объект на все приложение)
3. управляемые сообщениями (Message Driven Beans) — их логика является реакцией на события в системе

Apache Ant

Apache Ant — утилита для автоматизации процесса сборки программного продукта. Она является платформонезависимым аналогом утилиты make, где все команды записываются в XML-формате.

В отличие от make, утилита Ant полностью независима от платформы, требуется лишь наличие на применяемой системе установленной рабочей среды Java — JRE. Отказ от использования команд операционной системы и формат XML обеспечивают переносимость сценариев.

Управление процессом сборки происходит посредством XML-сценария, также называемого Build-файлом. В первую очередь этот файл содержит определение проекта, состоящего из отдельных целей (Targets). Цели сравнимы с процедурами в языках программирования и содержат вызовы команд-заданий (Tasks). Каждое задание представляет собой неделимую, атомарную команду, выполняющую некоторое элементарное действие.

Между целями могут быть определены зависимости — каждая цель выполняется только после того, как выполнены все цели, от которых она зависит (если они уже были выполнены ранее, повторного выполнения не производится).

Типичными примерами целей являются clean (удаление промежуточных файлов), compile (компиляция всех классов), deploy (развёртывание приложения на сервере). Конкретный набор целей и их взаимосвязи зависят от специфики проекта.

Ant позволяет определять собственные типы заданий путём создания Java-классов, реализующих определённые интерфейсы.

Java Persistence API

Java Persistence API (JPA) — спецификация Java EE, с помощью которой можно удобно работать с базой данных и представлять хранящуюся в БД информацию в виде Java-объектов.

Ключевым понятием в этой спецификации является Entity. Entity (Сущность) — POJO-класс (Plain Old Java Object), связанный с базой данных с помощью аннотации (@Entity). Сущность должна соответсвовать следующим критериям:

- Должен иметь пустой публичный или защищенный конструктор
- Не может быть вложенным, интерфейсом или перечислением
- Не может быть неизмяемым и не может содержать final-полей
- Должен содержать хотя бы одно поле, помеченное аннотацией @Id

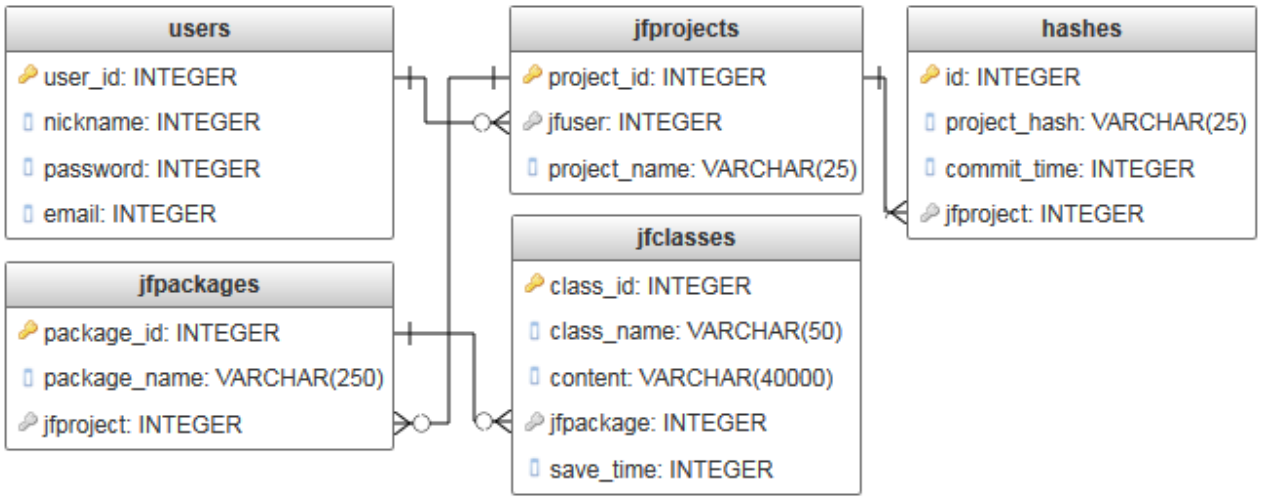
JavaServer Faces

JavaServer Faces (JSF) — это фреймворк для веб-приложений, написанный на Java. Он служит для того, чтобы облегчать разработку пользовательских интерфейсов для Java EE-приложений. В отличие от прочих MVC-фреймворков, которые управляются запросами, подход JSF основывается на использовании компонентов. Состояние компонентов пользовательского интерфейса сохраняется, когда

пользователь запрашивает новую страницу и затем восстанавливается, если запрос повторяется.

Структура хранения данных

В приложении используется СУБД PostgreSQL. На рисунке представлена схема хранения данных.



Используется пять таблиц:

1. В таблице users хранятся сведения о пользователях приложения
   - user_id - уникальный идентификатор пользователя
   - nickname - логин пользователя
   - password – пароль
   - email - электронная почта
2. В таблице jfprojects хранятся сведения о созданных проектах
   - project_id - уникальный идентификатор проекта
   - jfuser - ссылка на пользователя, которому принадлежит этот проект
   - project_name - название проекта
3. В таблице jfpackages хранятся сведения о пакетах (Java packages) проекта
   - package_id - уникальный идентификатор пакета
   - package_name - название проекта

- jfproject - ссылка на проект, которому принадлежит этот пакет

4. В таблице jfclasses хранятся сведения о классах проекта
   - class_id - уникальный идентификатор класса
   - class_name - название класса
   - content - содержимое класса
   - jfpackage - ссылка на пакет, в котором лежит данный класс
   - save_time - время, когда класс был создан
5. В таблице hashes хранятся сведения о хэшах проектов
   - hash_id - уникальный идентификатор хэша
   - project_hash – хэш
   - jfproject - ссылка на проект
   - commit_time - время, когда проект был синхронизирован с GIT.

Структура приложения

В основе приложения лежит компонентная модель, рассмотренная в предыдущих главах.

Приложение состоит из двух компонент:

1. EJB-модуль

   В этом модуле находятся следующие классы:

   a. EJB-бины
   b. классы-утилиты для работы с настройками и пакетами
   c. Entity-классы для каждой из используемых таблиц БД (jfclasses, jfpackages, jfprojects, users, hashes)
   d. Классы для работы с процессами (тип процесса, статус процесса)
   e. Классы для компиляции и запуска проектов пользователя
2. WEB-модуль, который отвечает за весь WEB-функционал

   Этот модуль содержит:

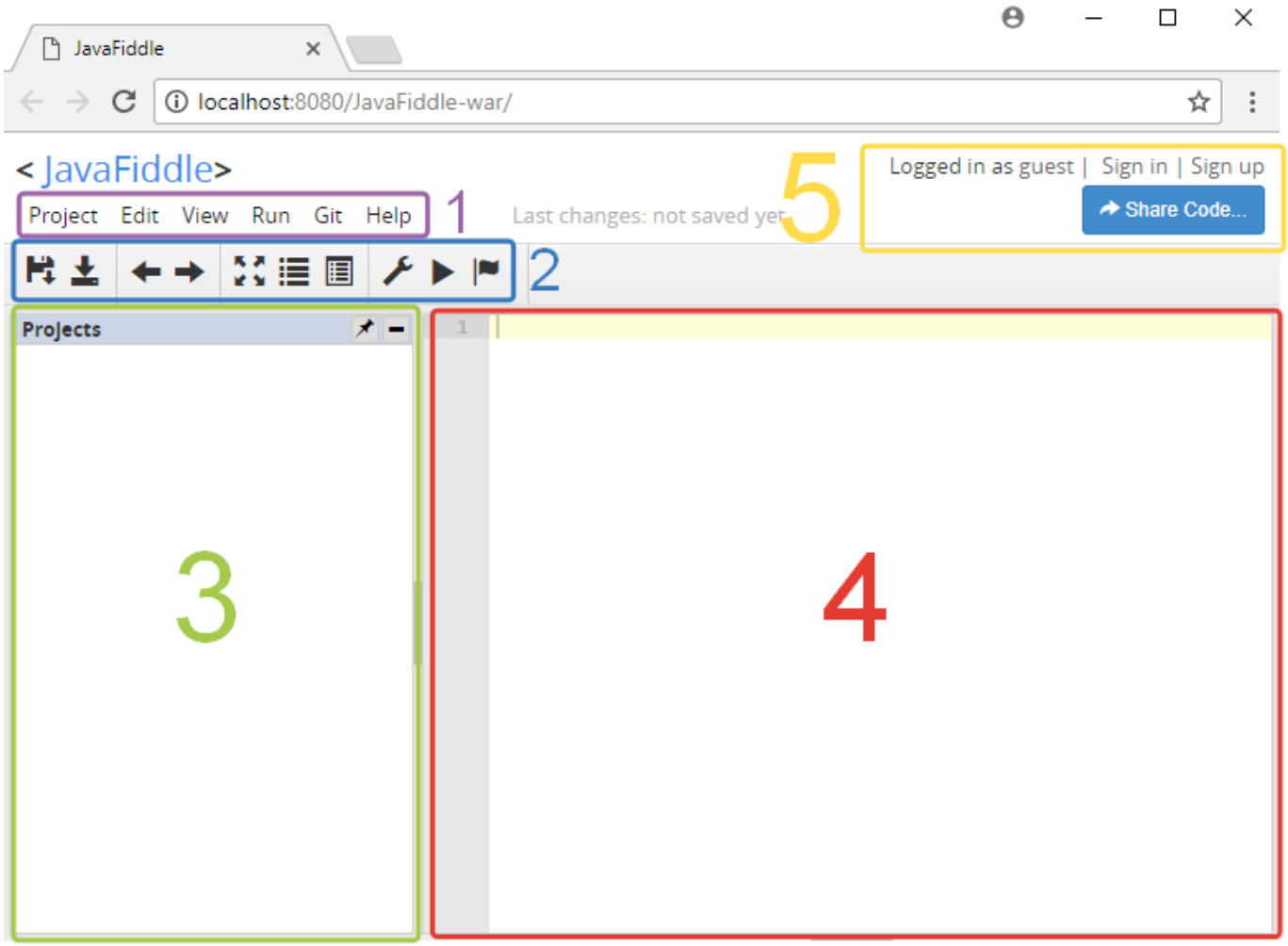
   a. Классы для работы с системой контроля версий Git
   b. Классы для работы с системой сборки Maven
   c. Классы для сохранения (сериализации) пользовательских проектов
   d. Бины для работы с регистрацией и авторизацией пользователя
   e. Веб-сервисы
   f. Классы-утилиты
   g. XHTML-страницы для пользовательского интерфейса
   h. Шрифты, иконки, стили CSS
   i. Скрипты, написанные на языке JavaScript

Кроме этого, каждый проект включает в себя КПИ:

1. библиотеки JAR (opensource)
   a. библиотека Eclipse для работы с Git
   b. библиотека Apache IO
   c. библиотеку Maven
2. Таблицы стилей CSS Bootstrap
3. Opensource-библиотеки JavaScript
   a. Bootstrap
   b. JQuery

Пользовательский интерфейс

Графический интерфейс приложения представляет собой окно в интернет-браузере, которое позволяет пользователю создавать проекты, пакеты и классы, а также производить сборку и запуск проекта.



Можно выделить пять составляющих интефейса (см. нумерацию на рисунке):

1. Строка меню содержит все действия, которые может выполнить пользователь
2. Панель управления содержит основные команды, которые используются при разработке ПО
3. Дерево проектов содержит иерархическую информацию о проекте, пакетах и классах
4. Окно разработки — это основное окно, в котором отображается класс, с которым пользователь работает в данный момент
5. Панель авторизации содержит кнопки регистрации нового пользователя и авторизации уже существующего

Итоги и дальнейшие пути разработки

На основе предложенной сервисно-ориентрованной модели автором было реализовано приложение на языке Java. Выбранная модель обеспечила быстрое проектирование и разработку, приложение получилось вариабельным (изменяемым), а добавление новой функциональности занимает мало времени и требует меньше ресурсов.

Ключевые процессы для сложных приложений – разработка КПИ и внедрение SOA и SCA. Архитектура SOA была разработана и впервые применена компаний IBM. Для банковской сферы в IBM разработан набор компонентов в рамках WebSphere Portal (Process Server, WSphere Service Registry and Repository, WebSphere Enterprise Bus, WebSphere Portlet Factory, WebSphere Application Server), а также соответствующие инструменты разработки новых приложений. Основной особенностью предлагаемой технологии – точки изменчивости, варианты использования и среды выполнения.

В качестве результата автор считает необходимым внедрить предложенные сервисы в среду JAVA, чтобы улучшить работу пользователей, разрабатывающих новые системы и приложения.

ВЫВОДЫ

Проведен анализ методов моделирования систем из готовых ресурсов (модулей, объектов, компонентов, сервисов и др.) с применением современных моделей проектирования: UML (Unified Modeling Language), MDD (Model Driven Development), MDA (Model Driven Architecture), PIM (Platform Independent Model), платформо-зависимой модели PSM (Platform Specific Models) и др.

В разделе 1 работы дано описание готовых ресурсов – компонентов повторного использования (КПИ). Рассмотрена компонентная теория, основанная на моделях компонентов, среды и компонентной алгебры для управления готовыми ресурсами системы, преобразования алгоритмов ресурсов в программную реализацию и конфигурирование готовых к исполнению ресурсов в приложение. Управление развитием системы проводится с помощью стандартных операций реинженерии, рефакторинга и реверсной инженерия.

В главе 2 описывается модельный подход к проектированию систем и приложений. Дано описание и характеристика множества сформировавшихся моделей: MDD, MDA, PIM, PSM, UML и др. Для каждой модели дается описание их сущности и возможности для моделирования, а также установления взаимоотношений между вомпонентами и интерфейсами. Дана оценка моделирования систем с помощью аппарата диаграмм UML, широко используемой во многих организациях. Сделан вывод о громоздкости получаемой структуры и перехода на новую модель UML2.

В разделе 3 дано описание компонентного моделирования систем. Представлена формальная структура модели компонента и способы и операции для формирования с помощь операций классов ООП. Определены внутренние и внешние связи базовых переменных ресурсов и механизмы композиции, наследования и вариаций компонентов в структуре КПИ и их обработка в современных средах. Рассмотрены базовые понятия классов, способов их загрузки, отображения и выполнения.

Определены шаблоны проектирования, взаимосвязи объектов через интерфейсы при построении систем и их вариантов.

В разделе 4 описываются реализованное автором приложение на языке Java на основе рассмотренных в главах 1-3 моделей и готовых ресурсов в виде компонентов. ПО приложения, реализованное на языке Java, состоит из набора компонентов, описываемых ее средствами. Компоненты собирается в WAR файл и запускается на сервере приложений Glassfish. Особое внимание уделено графическому интерфейсу приложения представленных визуально окнами в Интернет-браузере, в котором пользователю доступны функции регистрации, авторизации, создания проектов, пакетов и классов, компиляции и запуска.

В ходе анализа существующих моделей проектирования систем и использования компонентой модели при разработке реальной программной системы удалось показать, что компонентый метод и использование КПИ значительно облегчает и ускоряет проектирование и разработку приложения. При этом приложение становится вариабельным (изменяемым), а добавление новой функциональности занимает мало времени и требует меньше ресурсов.

ЛИТЕРАТУРА

1. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. Основы индустрии программных продуктов. – К.: Наук. думка, 2009. – 371 с.
2. *Лаврищева Е.М.* Software Engineering компьютерных систем. Парадигмы, Технологии, CASE-средства программирования.-Наук. Думка, 2014.-284с.
3. *Лаврищева Е.М.* Парадигмы программирования сборочного типа в программной инженерии.- Межд. Конференция –УкрПРОГ-2014, Сборник трудов.-с.76-92.
4. *Лаврищева Е.М.* Теория объектно-компонентного моделирования программных систем. 48 с. ИСП РАН ISBN 978-5-91474-025-9.
5. Ekaterina Lavrischeva, Andrey Stenyashin, Andrii Kolesnyk, Object-Component Development of Application and Systems. Theory and Practice, Journal of Software Engineering and Applications, 2014, 7, Published Online August 2014 in SciRes http://www.scirp.org/journal/jsea
6. *Лаврищева Е.М.* Компонентная теория и коллекция технологий для разработки индустриальных приложений из готовых ресурсов, Труды Четвертой научно-практической конференции «Актуальные проблемы системной и программной инженерии», АПСПИ-2015, 20-21мая 2015, с. 101-119
7. *Lavrischeva E.M., Petrov I.B.* Ways of Development of Computer Technologies to Perspective Nano, Agenda - Future Technologies Conference (FTC) 2017, 29-30 November 2017| Vancouver, Canada, p. 1-10.
8. *Лаврищева Е.М.* Программная инженерия. Учебно-методическое пособие в 3-х частях. М.: МФТИ.-2016.
9. *Чернецки К., Айзенекер У.* Порождающее программирование. Методы, инструменты, применение.– Издательский дом Питер. – М.– СПб. – Харьков. – Минск.– 2005.– 730 с.
10. *Pohl K., Böckle G., Linden F.J.* Software Product Line Engineering: Foundations, Principles and Techniques. – New York: Springer–Verlag. – 2005. – 437 p.
11. *G. Lenz, C. Wienands.* Practical Software Factories in .NET, 2006.
12. *D. Gasevic, D. Djuric, V. Devedzic.* Model Driven Architecture and Ontology Development. Springer, 2006.
13. *M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang.* Organization domain modelling (ODM) guidebook version 2.0. Technical Report STARS-VC-A025/001/00, Synquiry Technologies, Inc, 1996.
14. *K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson.* Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
15. *J. Coplien, D. Hoffman, and D. Weiss.* Commonality and variability in software engineering. IEEE Software, pages 37-45, November/December 1998.
16. *J.-M. DeBaud and K. Schmid.* A systematic approach to derive the scope of software

product lines. In 21st International Conference on SE, ICSE-99, pages 34-43. ACM, 1999.
17. *R. R. Macala, L. D. Sutckey, and D. C. Gross.* Managing domain-specific product-line development. IEEE Software, 13, May 1996.
18. *P. Hudak.* Building domain-specific embedded languages. ACM Computing Surveys, 28(4es), December 1996.
19. *J. K. Ousterhout.* Scripting: Higher level programming for the 21st century. IEEE Computer, March, 1998.
20. *Holger Knublauch, Ray W. Fergerson, Natalya F. Noy, and Mark A. Musen.* The Protege OWL Plugin: An open development environment for semantic web applications. In 3rd International Semantic Web Conference (ISWC 2004), Hiroshima, Japan, 2004.
21. *Czarnecki K., Antklevicz M.* Multilevel Customization in Application Engineering//Communication of the ACM.-№12.-2006.-pp.61-65
22. *J. Bergey, L. O'Brian, and D. Smith.* Mining existing assets for software product lines. Technical Report Technical Report CMU/SEI-2000-TN-008, SEI, Pittsburgh, PA, 2000.
23. *Trinidad P., Benavides D., Ruiz–Cort´es A.* Improving Decision Making in Software Product Lines Product Plan Management // in: Proc. of the ADIS 2004 Workshop on Decision Support in SE, CEUR Workshop Proceedings. – 2004. – Vol. 120. – P. 51–58.
24. *Sinnema M., Deelstra S.* Classifying Variability Modeling Techniques //Elsevier J. on Inf. and Software. Technology. – 2007. – 49.
25. *Asikainen T., Männistö T., Soininen T.* A Unified Conceptual Foundation for Feature Modelling //in Proc. of the 10th Int. Software Product Line Conf. (SPLC 2006). – 2006. – P. 31 – 40.
26. *Липаев В.В.* Программная инженерия сложных заказных программных продуктов. Учебное пособие.-Мак-Пресс, Москва, 2014.-308с.
27. *Грищенко В.Н., Лаврищева Е.М.* Методы и средства компонентного программирования // Кибернетика и системный анализ.– 2003.– №1.– С. 39–43.
28. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. Основы индустрии программных продуктов. – К.: Наук. думка, 2009. – 372 с.
29. *Лаврищева Е.М., Петрухин В.А.* Методы и средства инженерии программного обеспечения. – М.: МОН РФ, 2007. – 415 с.
30. *Лаврищева Е.М., Слабоспицька О.А., Коваль Г.І., Колесник А.А.* Теоретические аспекты управления вариабельностю в семействах программных систем. – Весник КНУ, серия физ.–мат.наук. – 2011. – №1. – С. 151–158.
31. *Daniel Oberle, Andreas Eberhart, Steffen Staab, and Raphael Volz.* Developing and Managing Software Components in an Ontology-based Application Server
32. *Sanjay Soni, Zhaohui Tang, Jim Yang.* Performance Study of Microsoft Data Mining Algorithms.