МИНИСТЕРСТВО ОБР МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Автоматизированных Систем Управления

РЕФЕРАТ

по дисциплине Программная инженерия

на тему Реинжиниринг и рефакторинг ПО

Выполнил студент группы

АВТ-812 Березин Дмитрий

Принял

Астапчук Виктор Андреевич

Новосибирск 2021 г.

Оглавление

[ВВЕДЕНИЕ 3](#_Toc87961541)

[1. Реинжиниринг ПО 5](#_Toc87961542)

[2. Рефакторинг ПО 10](#_Toc87961543)

[ЗАКЛЮЧЕНИЕ 14](#_Toc87961544)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 15](#_Toc87961545)

ВВЕДЕНИЕ

Разнообразие программного обеспечения куда больше, чем технических решений. Так как они решают самые различные задачи, начиная от связи с оборудованием (драйвера) и заканчивая автоматизацией бухгалтерского учета или 3-х мерными играми.

Однако даже при таком большом разнообразии программных решений может оказаться, что нет полностью удовлетворяющего программного решения.

Для решения данной проблемы предприятие, как правило, находит программиста, который пытается реализовать данную программу. Проходит время, программа внедряется на предприятии и с ней начинает работать большое количество персонала. Привыкнув к программе, сотрудники уже не представляют себя без столь удобного инструмента, как программа. Проходит еще время, программист покидает место работы и далее поддерживать проект не может. В результате, предприятие сталкивается с большой проблемой: есть программа, с которой привык работать персонал и подобной на рынке не найти, но нет ее дальнейшего совершенствования и поддержки. Данная программа начинает резко устаревать. Сначала оказывается, что в ней нет возможностей, которые стали необходимы после увольнения программиста, а затем она не может эффективно работать с современным оборудованием или начинает медленнее работать из-за большого количества введенной информации.

Спустя некоторое время оказывается, что в данной программе больше нельзя работать, так как она крайне медленно работает, некоторые функции начинают выдавать некорректный результат или вовсе перестают работать.

Столкнувшись с данной проблемой, предприятие начинает искать нового программиста или компанию, которая способна привести данную программу в рабочее состояние. Однако предприятие сталкивается с новой проблемой. Спустя долгое время становится все сложнее найти разработчика, способного дорабатывать и модифицировать устаревшее ПО. Предприятию очень повезет, если оно сразу найдет грамотного и ответственного программиста, способного освежить код программы. Однако, грамотные программисты дешево не стоят и постоянно хотят перспектив. Поэтому, если вы не быстро развивающееся программное предприятие, да еще не так уж и много платите, то придет момент, что данный программист уйдет тоже. И опять начинается все снова: 2-3 безответственных программиста, потом один профессиональный и ответственный, который 2-3 месяца будет вникать в курс дела и через 2-3 года уйдет.

Подобные подходы к разработке ПО крайне неэффективны, потому что требуют большого количества времени и могут приводить к большому количеству ошибок и недочетов. Поэтому предприятия, которые работают долго и успешно на рынке, в результате приходят к выводу, что для дальнейшего совершенствования программ необходимо нанять компанию-разработчика. Компания – разработчик обладает возможностями регулярного обновления и поддержки ПО. Кроме этого они могут оперативно вести добавление новых функций и исправление ошибок в уже существующих. В данном реферате описаны основные подходы для наиболее эффективной поддержки и модификации ПО.

1. Реинжиниринг ПО

**Реинжиниринг программного обеспечения** — процесс создания новой функциональности или устранения ошибок, путём революционного изменения, но используя уже имеющееся в эксплуатации программное обеспечение. Выражаясь менее формально, реинжиниринг является изменением системы программного обеспечения после проведения обратного инжиниринга

Реинжиниринг можно разделить на несколько этапов:

**Начальная фаза.** Начать процесс реинжиниринга следует с определения того, что есть по существующей системе (исходные тесты, БД, описания и т. д.). При этом фиксируется текущее состояние наследуемой системы (все изменения, вносимые в нее после этого момента, при выполнении реинжиниринга не учитываются). Если есть возможность выполняется развертывание наследуемой программной среды у разработчика.

**Определение системных архитектур.** Работы по описанию архитектур начинаются фактически на начальном этапе, когда определяется состав оборудования и стандартное программное обеспечение, необходимые для инсталляции и запуска существующей зафиксированной системы. Тем самым фактически определяются архитектуры БД, оборудования и стандартного ПО, телекоммуникаций. Все архитектуры представляются в нотации UML и при необходимости дополняются текстовыми описаниями. Поостренные архитектурные модели в процессе реинжиниринга будут уточняться и дополняться.

**Автоматический реинжиниринг.** Автоматический реинжиниринг осуществляется с помощью инструментальных средств визуального моделирования. Его выполнение позволяет построить модели, которые могут быть приняты как исходные. Автоматическому реинжинирингу подвергается как бизнес логика (если есть исходные коды на объектно-ориентированном или объектно-базированном языке), так и БД.

Автоматический реинжиниринг бизнес логики может быть выполнен только в случае, когда имеются (полностью или частично) исходные тексты программ. В результате автоматического реинжиниринга кодов создаются диаграммы классов и диаграммы компонент UML.

Реинжиниринг БД выполняется с помощью инструментальных средств проектирования БД. Результатом является реляционная модель данных, которая может графически отображаться этим средством. Полученная реляционная модель может по усмотрению разработчиков переведена в диаграмму классов UML путем использования применяемого инструментального средства разработки БД или программных мостов со средствами визуального объектно-ориентированного моделирования.

**Редактирование диаграмм моделей.** Модели, полученные автоматически, весьма сложно читать и анализировать, поскольку элементы модели размещаются без учета наглядности диаграмм. Поэтому построенные модели должны быть отредактированы. В процессе редактирования не следует выполнять содержательные преобразования (удалять или добавлять элементы модели). Главной целью редактирования на этом этапе является достижение наглядности диаграмм. Для этого используется перемещение элементов диаграмм. В процессе редактирования могут вноситься комментарии к элементам модели. Например, можно прокомментировать назначение отдельных классов. Комментарии заносятся в поля спецификаций элементов моделей.

Если диаграмма содержит слишком много элементов, то анализировать ее сложно. Попробуйте проанализировать диаграмму, содержащую более 100 классов! Поэтому целесообразно разбить такую диаграмму на несколько отдельных диаграмм, оставляя в каждой примерно по 7 – 10 элементов.

Метод повышения наглядности диаграмм хорошо известен. Это иерархическая реструктуризация. Средством ее осуществления в UML являются пакеты. Сложные программные среды обычно включают несколько подсистем, имеющих разное целевое назначение и функциональность. Поэтому на верхнем уровне иерархии можно показать пакеты – подсистемы. Каждый из таких пакетов должен получить имя, отражающее суть соответствующей подсистемы. На этом уровне можно также показать пакеты классов, являющиеся общими для всей системы и используемые подсистемами. На начальной стадии реструктуризации логической модели можно ввести пакет верхнего уровня, куда помещаются классы, которые трудно отнести к какому-либо другому пакету. В любой программной среде есть пользовательский интерфейс, связь с БД, управление, обработка, классы данных. Такого типа пакеты можно ввести в каждой подсистеме на следующем уровне иерархии.

**Построение функциональной модели.** Модель функционирования описывается с помощью диаграмм вариантов использования и детализирующих их диаграмм последовательностей и деятельностей. Источником для ее построения является работающая наследуемая система и проводимые с ней эксперименты. На этом этапе особенно эффективно привлечение к работам по реинжинирингу эксперта организации заказчика. С его помощью можно быстрее и точнее определить состав акторов наследуемой системы и основные варианты использования. Эксперт заказчика может словесно описать, кто использует систему и что она должна делать для пользователей каждого типа. Он может также информировать, с какими другими системами взаимодействует наследуемая программная среда, какие работы осуществляются периодически. Все эти сведения будут способствовать более точному пониманию функциональности системы разработчиками.

**Определение акторов.** Для нахождения акторов следует искать ответы на следующие вопросы:

* Кто является непосредственными пользователями системы? Необходимо при ответе на данный вопрос указывать роли, а не конкретных людей, исполняющих эти роли.
* С каким внешним оборудованием или программами осуществляет взаимодействие система?
* Выполняет ли система работы, активизируемые наступлением конкретного времени или истечением определенных интервалов времени (при положительном ответе одним из акторов является таймер)?

Ответы на поставленные вопросы можно получить либо путем опроса экспертов заказчика, либо из документации на систему, либо (если таковых не имеется) путем запуска системы и анализа экранных форм или меню.

Акторам следует присвоить имена, отражающие их роли в работе с системой.

**Определение вариантов использования.** Определение вариантов использования выполняется на основе анализа экранов работающей системы. Сначала определяются пакеты вариантов использования. Для этого следует найти все первичные экраны и для каждого из них в модели на головной диаграмме вариантов использования завести отдельный пакет. Таким образом, на этой диаграмме будет пакет акторов и несколько пакетов вариантов использования.

Далее выполняется детализация построенных пакетов вариантов использования на основе анализа экранных форм. Рекомендуемые правила:

* если экран содержит меню, то это пакет вариантов использования. При этом каждая строка меню – это либо подпакет, либо отдельный вариант использования;
* если основной экран – форма, то это отдельный вариант использования.

**Определение взаимодействий акторов и вариантов использования.** Поскольку очень важно показать, как акторы соотносятся с вариантами использования, после нахождения вариантов использования определяется, какие акторы взаимодействуют с системой в этом варианте. В модель включаются ассоциации. Они имеют направления, соответствующие направлениям передачи информации между акторами и вариантами использования.

**Распределение по пакетам.** Если число акторов или вариантов использования слишком велико, то для упрощения поддержки модели вариантов использования целесообразно разделить их по пакетам. Это также упрощает понимание модели и распределение ответственности путем назначения разработчиков, ответственных за пакеты вариантов использования. Можно использовать следующие критерии упаковки вариантов использования в один пакет:

* если они взаимодействуют с одним актором;
* имеют друг с другом отношения включения или расширения.

Могут быть и другие способы обеспечения наглядности модели, важно лишь иметь четкую стратегию разбиения на пакеты.

**Построение навигации экранов.** Одновременно с выделением вариантов использования строится навигация экранов наследуемой системы в виде диаграммы классов UML. Каждый экран показывается в модели как отдельный класс, в котором полям соответствуют атрибуты, функциональным кнопкам – операции, а кнопкам меню – одноименные отношения.

**Детализация функциональности.** Детализация функциональности представляет собой построение сценариев реализации вариантов использования, представленных в модели вариантов использования. Она выполняется с помощью диаграмм последовательностей и диаграмм деятельностей UML. Выбор вида диаграмм в каждом конкретном случае зависит от того, что преобладает в данном варианте использования – логика выполнения или передачи данных. В первом случае предпочтительно применять диаграммы деятельностей, где легко показывать ветвления и параллельную обработку, во втором – диаграммы последовательностей.

Детализация требуется в особенности для тех вариантов использования, в которых важна последовательность действий, учитывается состояние системы, имеется разветвленная логика выполнения. Целесообразно детализировать выполнение варианта использования, определяющих основную функциональность системы. Если заказчик высказывает пожелания относительно того, что из наследуемой системы должно быть сохранено в новой программной среде, то именно эти варианты использования должны быть детализированы.

Детализация осуществляется на основе анализа исходных кодов. По текстам программ выявляются ветвления, выражения, циклы. Это позволяет восстановить алгоритмы, представив их в виде диаграмм деятельностей или диаграмм состояний. Другой путь – это проведение экспериментов с работающей наследуемой системой. Варьирование входных данных и анализ реакции системы на эти данные делает возможным обнаружение ветвлений и ограничений. Можно также выдвигать и проверять гипотезы об алгоритмах вычислений.

Модели, построенные в результате реинжиниринга, являются основой для определения требований к проектируемой программной среде, а также для построения логической и функциональной моделей новой системы.

2. Рефакторинг ПО

Рефакторинг кода – это процесс внесения изменений в исходный код программы посредством небольших преобразований, смысл которых эквивалентен прежним фрагментам, но результат выглядит более стройным, органичным, логичным и «прозрачным».

Этап рефакторинга нельзя считать однократным и его уместно проводить после добавления новой функциональности, поскольку такие действия легко могут привести к необходимости провести ряд преобразований, связанных с манипуляцией классами и их элементами.

Кроме этого, отдельного внимания, должны быть удостоены и те части кода, которые давно не редактировались (не были затронуты в процессе исправления ошибок или расширения функциональности), поскольку вряд ли они настолько невосприимчивы к вносимым изменениям, хотя и сохраняют корректное поведение.

Проведение рефакторинга важно, поскольку программу еще нужно будет сопровождать и совершенствовать. Искать и исправлять ошибки также куда быстрее и эффективнее в стройном и прозрачном коде. Рефакторинг не поможет исправить нерабочий код, но может помочь диагностировать причины таких неприятностей.

Кроме этого, сложные программные компоненты часто разрабатываются совместно несколькими людьми и нужно понимать, что любой опытный разработчик захочет изучить логику работы программы, анализируя непосредственно исходный код. Вот для этих целей нужно, чтобы код был «читабельным». Таким образом, можно сделать вывод, что рефакторинг кода может помочь ускорить процесс совместной разработки программного продукта.

Основополагающие принципы рефакторинга:

1. Применять рефакторинг при добавлении новой функции. Необходимо стараться лучше понимать код, который надо модифицировать. Также исправлять дизайн, не способствующий легкому добавлению новой функции.
2. Применять рефакторинг, если требуется исправить ошибку. При исправлении ошибок польза рефакторинга во многом заключается в том, что код становится более понятным. При проведении рефакторинга часто оказывается, что такая активная работа с кодом помогает найти в нем ошибки.
3. Применять рефакторинг при разборе кода. Благодаря разбору кода знания становятся достоянием всей команды разработчиков. При этом более опытные разработчики передают свои знания менее опытным. Разборы помогают большему числу людей разобраться с большим числом аспектов крупной программной системы. Они также очень важны для написания понятного кода.

Методы рефакторинга являются техническими способами проведения, предназначенными для достижения определенных целей рефакторинга. В перепроектировании важна концепция малых шагов, требующая применения серий микрорефакторингов, а составным элементом микрорефакторинга можно считать применение метода на практике.

Наиболее часто используемые методы рефакторинга:

1. **Изменение сигнатуры метода** (Change Method Signature). Суть изменения сигнатуры метода заключается в добавлении, изменении или удалении параметра метода. Изменив сигнатуру метода, необходимо скорректировать обращения к нему в коде всех клиентов. Это изменение может затронуть внешний интерфейс программы, кроме того, не всегда разработчику, изменяющему интерфейс, доступны все клиенты этого интерфейса, поэтому может потребоваться та или иная форма регистрации изменений интерфейса для последующей передачи их вместе с новой версией программы.
2. **Инкапсуляция поля** (Encapsulate field). В случае, если у класса имеется открытое поле, необходимо сделать его закрытым и обеспечить методы доступа. После «Инкапсуляции поля» часто применяется «Перемещение метода».
3. **Выделение метода** (Extract Method). Выделение метода заключается в выделении из длинного и/или требующего комментариев кода отдельных фрагментов и преобразовании их в отдельные методы, с подстановкой подходящих вызовов в местах использования. В этом случае действует правило: если фрагмент кода требует комментария о том, что он делает, то он должен быть выделен в отдельный метод. Также правило: один метод не должен занимать более чем один экран (25-50 строк, в зависимости от условий редактирования), в противном случае некоторые его фрагменты имеют самостоятельную ценность и подлежат выделению. Из анализа связей выделяемого фрагмента с окружающим контекстом делается вывод о перечне параметров нового метода и его локальных переменных.
4. **Перемещение метода** (Move Method). Перемещение метода применяется по отношению к методу, который чаще обращается к другому классу, чем к тому, в котором сам располагается.
5. **Замена условного оператора полиморфизмом** (Replace Conditional with Polymorphism). Условный оператор с несколькими ветвями заменяется вызовом полиморфного метода некоторого базового класса, имеющего подклассы для каждой ветви исходного оператора. Выбор ветви осуществляется неявно, в зависимости от того, экземпляру какого из подклассов оказался адресован вызов.

Код, обладающий признаками проблем в системе, может являться первым сигналом о необходимости проведения рефакторинга, индикатором общего состояния кода. Но, следует отметить, что код, содержащий возможные признаки проблем, не всегда является таковым, и в каждом конкретном случае необходим анализ как самого кода, так и его контекста для вынесения окончательного решения о необходимости рефакторинга. Примерами такого «ложно-проблемного» кода может являться высокооптимизированный код, когда в приоритете над читаемостью, расширяемостью и простотой находится быстродействие и потребление ресурсов разрабатываемым ПО, а также некоторые вырожденные случаи.

Не существует четких правил поиска «ложно-проблемного» кода, т.к., как уже отмечалось выше, само это понятие применимо в зависимости от контекста. Ниже приведен список основных признаков, на которые в первую очередь следует обращать внимание и которые могут сигнализировать о проблемах в коде:

* Дублирование кода – одинаковые структуры кода в нескольких местах.
* Длинный метод – чем длиннее метод, тем сложнее понять его назначение. Как правило, если у метода хорошее название, то не требуется изучать код для понимания чем занимается конкретный метод.
* Большой класс – класс выполняет слишком много функций, что может приводить к дублированию кода. Отличительный признак таких классов – большое количество атрибутов.
* Длинный список параметров – в таких списках крайне сложно разбираться, и они тяжелы в использовании. Кроме того, они могут приводить к противоречивым ситуациям.
* Расходящиеся модификации – проблема, возникающая при невозможности найти нужное место для модификации системы. Возникает вследствие плохой структурированности кода.
* Стрельба дробью – при любом изменении в коде возникает необходимость вносить много мелких изменений в большое число элементов кода.
* Временное поле – поля, которые нужны только при определенных обстоятельствах. Это может мешать чтению кода.
* Отказ от наследства – если наследник использует лишь малую часть унаследованных методов, то это может быть признаком неправильной иерархии.
* Комментарии – излишнее стремление писать комментарии может сигнализировать о том, что код становится нечитаемым и его структуру следует переработать.

ЗАКЛЮЧЕНИЕ

Модификация программного обеспечения – важная часть разработки. Любой квалифицированный разработчик должен уметь правильно и быстро находить все недостатки кода, которые могут привести к ошибочным результатам. Реинжиниринг и рефакторинг – два процесса, взаимодополняющих друг друга. Без рефакторинга исходного кода реинжиниринг может производиться в разы дольше, так как код может оказаться нечитаемым либо содержать много участков с повторяющимся кодом.

Знание принципов рефакторинга позволяет отличить участки кода, которые можно назвать «некачественными» или даже ошибочными. Рефакторинг дает возможность сделать код более управляемым с точки зрения добавления новых функций и поддержки уже существующих, сократить время на правку необходимых функций и читаемость написанного кода.

Реинжиниринг содержит в себе принципы, позволяющие безболезненно добавлять новые функции к уже существующему коду. Знание этих принципов позволит разработчику максимально эффективно и быстро оценить требования к нововведениям, а также реализовать их в максимально сжатые сроки. Связано это с тем, что принципы реинжиниринга предполагают следование жестким правилам при разработке новых функций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Рефакторинг кода // Википедия URL: https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3 (дата обращения: 15.11.2021).
2. Реинжиниринг программного обеспечения // Википедия URL: https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B8%D0%BD%D0%B6%D0%B8%D0%BD%D0%B8%D1%80%D0%B8%D0%BD%D0%B3\_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE\_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F (дата обращения: 15.11.2021).
3. Фаулер М. Рефакторинг: улучшение существующего кода. - СПб: Символ-Плюс, 2003. - 432 с.
4. Кериевски Дж. Рефакторинг с использованием шаблонов. - М.: ООО "И.Д. Вильямс", 2006. - 400 с.