**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

**Кафедра информационных систем управления**

**Реферат по теме**

**«Особенности интеграционного тестирования для объектно-ориентированного программирования»**

Выполнил студент 13 группы

Жуковский Павел Сергеевич

Минск 2019

Оглавление

[**Введение** 3](#_Toc1161968)

[**1. Интеграционное программирование** 5](#_Toc1161969)

[1.1 Интеграционное тестирование 5](#_Toc1161970)

[1.2 Недостатки нисходящего и восходящего тестирований 7](#_Toc1161971)

[**2. Особенности интеграционного тестирования для объектно-ориентированного программирования** 8](#_Toc1161972)

[**Список используемой литературы** 15](#_Toc1161973)

# **Введение**

Тестирование программного обеспечения – это процесс исследования, испытания [программного продукта](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), имеющий две различные цели:

1. продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям;
2. выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.

Существует несколько уровней тестирования:

1. [Тестирование компонентов](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. Часто тестирование компонентов осуществляется разработчиками программного обеспечения.
2. [Интеграционное тестирование](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — тестируются интерфейсы между компонентами, подсистемами или системами. При наличии резерва времени на данной стадии тестирование ведётся итерационно, с постепенным подключением последующих подсистем.
3. [Системное тестирование](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — тестируется интегрированная система на её соответствие [требованиям](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%BC%D1%83_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8E).
   1. Альфа-тестирование — имитация реальной работы с системой штатными [разработчиками](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82), либо реальная работа с системой потенциальными пользователями/заказчиком. Чаще всего альфа-тестирование проводится на ранней стадии разработки продукта, но в некоторых случаях может применяться для законченного продукта в качестве внутреннего приёмочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы специальным людям, которые занимаются тестированием, для дополнительного исследования в окружении, подобном тому, в котором будет использоваться программа.
   2. [Бета-тестирование](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D1%82%D0%B0-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — в некоторых случаях выполняется распространение предварительной версии (иногда с ограничениями по функциональности или времени работы) для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Часто для свободного и открытого программного обеспечения стадия [альфа-тестирования](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D1%8C%D1%84%D0%B0-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) характеризует функциональное наполнение кода, а [бета-тестирования](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D1%82%D0%B0-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) — стадию исправления ошибок. При этом как правило на каждом этапе разработки промежуточные результаты работы доступны конечным пользователям.

# **1. Интеграционное программирование**

## 1.1 Интеграционное тестирование

Интеграционное тестирование ([*англ.*](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Integration testing, иногда называется Integration and Testing, аббревиатура *I&T*) — одна из фаз [тестирования программного обеспечения](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%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), при которой отдельные программные модули объединяются и тестируются в группе.

Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования.

Основная задача интеграционного тестирования – поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями, а также проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надежности. Тестирование этих проектируемых единиц — объединения, множества или группы модулей — выполняется через их интерфейс, с использованием тестирования «чёрного ящика».

С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки на месте отсутствующих модулей. Основная разница между модульным и интеграционным тестированием состоит в целях, то есть в типах обнаруживаемых дефектов, которые, в свою очередь, определяют стратегию выбора входных данных и методов анализа. В частности, на уровне интеграционного тестирования часто применяются методы, связанные с покрытием интерфейсов, например, вызовов функций или методов, или анализ использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой.

На рис. 1 приведена структура комплекса программ K, состоящего из оттестированных на этапе модульного тестирования модулей M1, M2, M11, M12, M21, M22. Задача, решаемая методом интеграционного тестирования, – тестирование межмодульных связей, реализующихся при исполнении программного обеспечения комплекса K. Интеграционное тестирование использует модель «белого ящика» на модульном уровне. Поскольку человеку, занимающемуся тестированием, текст программы известен с детальностью до вызова всех модулей, входящих в тестируемый комплекс, применение структурных критериев на данном этапе возможно и оправдано.

Интеграционное тестирование применяется на этапе сборки модульно оттестированных модулей в единый комплекс. Известны два метода сборки модулей:

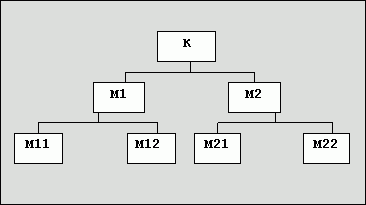


Рис. 1. Пример структуры комплекса программ

1. Монолитный, характеризующийся одновременным объединением всех модулей в тестируемый комплекс;
2. Инкрементальный, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с пошаговым тестированием собираемого комплекса. В инкрементальном методе выделяют две стратегии добавления модулей:
3. «сверху-вниз» и соответствующее ему восходящее тестирование;
4. «снизу-вверх» и соответственно нисходящее тестирование.

Особенности монолитного тестирования заключаются в следующем: для замены неразработанных к моменту тестирования модулей, кроме самого верхнего (К на рис. 1), необходимо дополнительно разрабатывать драйверы и/или заглушки, замещающие отсутствующие на момент сеанса тестирования модули нижних уровней.

Сравнение монолитного и интегрального подхода дает следующее:

1. монолитное тестирование требует больших трудозатрат, связанных с дополнительной разработкой драйверов и заглушек и со сложностью идентификации ошибок, проявляющихся в пространстве собранного кода;
2. пошаговое тестирование связано с меньшей трудоемкостью идентификации ошибок за счет постепенного наращивания объема тестируемого кода и соответственно локализации добавленной области тестируемого кода;
3. монолитное тестирование предоставляет большие возможности распараллеливания работ особенно на начальной фазе тестирования.

Особенности нисходящего тестирования заключаются в следующем: организация среды для исполняемой очередности вызовов оттестированными модулями тестируемых модулей, постоянная разработка и использование заглушек, организация приоритетного тестирования модулей, содержащих операции обмена с окружением, или модулей, критичных для тестируемого алгоритма.

## 1.2 Недостатки нисходящего и восходящего тестирований

Недостатки нисходящего тестирования:

1. проблема разработки достаточно «интеллектуальных» заглушек, т.е. заглушек, способных к использованию при моделировании различных режимов работы комплекса, необходимых для тестирования;
2. сложность организации и разработки среды для реализации исполнения модулей в нужной последовательности;
3. параллельная разработка модулей верхних и нижних уровней приводит к не всегда эффективной реализации модулей из-за подстройки (специализации) еще не тестированных модулей нижних уровней к уже оттестированным модулям верхних уровней.

Особенности восходящего тестирования в организации порядка сборки и перехода к тестированию модулей, соответствующему порядку их реализации.

Недостатки восходящего тестирования:

1. запаздывание проверки концептуальных особенностей тестируемого комплекса;
2. необходимость в разработке и использовании драйверов.

# **2. Особенности интеграционного тестирования для объектно-ориентированного программирования**

Программный проект, написанный в соответствии с объектно-ориентированным подходом, будет иметь Граф Модели Программы (ГМП), существенно отличающийся от ГМП традиционной «процедурной» программы. Сама разработка проекта строится по другому принципу - от определения классов, используемых в программе, построения дерева классов к реализации кода проекта. При правильном использовании классов, точно отражающих прикладную область приложения, этот метод дает более короткие, понятные и легко контролируемые программы.

Объектно-ориентированное программное обеспечение является событийно управляемым. Передача управления внутри программы осуществляется не только путем явного указания последовательности обращений одних функций программы к другим, но и путем генерации сообщений различным объектам, разбора сообщений соответствующим обработчиком и передача их объектам, для которых данные сообщения предназначены. Рассмотренная ГМП в данном случае становится неприменимой. Эта модель, как минимум, требует адаптации к требованиям, вводимым объектно-ориентированным подходом к написанию программного обеспечения. При этом происходит переход от модели, описывающей структуру программы, к модели, описывающей поведение программы, что для тестирования можно классифицировать как положительное свойство данного перехода. Отрицательным аспектом совершаемого перехода для применения рассмотренных ранее моделей является потеря заданных в явном виде связей между модулями программы.

Перед тем как приступить к описанию графовой модели объектно-ориентированной программы, остановимся отдельно на одном существенном аспекте разработки программного обеспечения на языке объектно-ориентированного программирования (ООП), например, C++ или С#. Разработка программного обеспечения высокого качества для MS Windows с применением только вновь созданных классов практически невозможна. Программист должен будет затратить массу времени на решение стандартных задач по созданию пользовательского интерфейса. Чтобы избежать работы над давно решенными вопросами, во всех современных компиляторах предусмотрены специальные библиотеки классов. Такие библиотеки включают в себя практически весь программный интерфейс операционной системы и позволяют задействовать при программировании средства более высокого уровня, чем просто вызовы функций. Базовые конструкции и классы могут быть использованы повторно при разработке нового программного проекта. За счет этого значительно сокращается время разработки приложений. В качестве примера подобной системы можно привести библиотеку Microsoft Foundation Class (MFC) для компилятора MS Visual C++.

Работа по тестированию приложения не должна включать в себя проверку работоспособности элементов библиотек, ставших фактически промышленным стандартом для разработки программного обеспечения, а только проверку кода, написанного непосредственно разработчиком программного проекта. Тестирование объектно-ориентированной программы должно включать те же уровни, что и тестирование процедурной программы - модульное, интеграционное и системное. Внутри класса отдельно взятые методы имеют императивный характер исполнения. Все языки ООП возвращают контроль вызывающему объекту, когда сообщение обработано. Поэтому каждый метод (функция - член класса) должен пройти традиционное модульное тестирование по выбранному критерию C (как правило, С1). В соответствии с введенными выше обозначениями, назовем метод Modi, а сложность тестирования - V(Modi, C). Все результаты, полученные в тестирования модулей, безусловно, подходят для тестирования методов классов. Каждый класс должен быть рассмотрен и как субъект интеграционного тестирования. Интеграция для всех методов класса проводится с использованием инкрементальной стратегии «снизу-вверх». При этом мы можем повторно использовать тесты для классов-родителей тестируемого класса, что следует из принципа наследования - от базовых классов, не имеющих родителей, к самым верхним уровням классов.

Графовая модель класса, как и объектно-ориентированной программы, на интеграционном уровне в качестве узлов использует методы. Дуги данной ГМП (вызовы методов) могут быть образованы двумя способами:

1. прямым вызовом одного метода из кода другого, в случае, если вызываемый метод виден (не закрыт для доступа средствами языка программирования) из класса, содержащего вызывающий метод, присвоим такой конструкции название Р-путь (P-path, Procedure path, процедурный путь);
2. обработкой сообщения, когда явного вызова метода нет, но в результате работы «вызывающего» метода порождается сообщение, которое должно быть обработано «вызываемым» методом.

Для второго случая «вызываемый» метод может породить другое сообщение, что приводит к возникновению цепочки исполнения последовательности методов, связанных сообщениями. Подобная цепочка носит название ММ-путь (MM-path, Method/Message path, путь метод/сообщение). ММ-путь заканчивается, когда достигается метод, который при отработке не вырабатывает новых сообщений (т. е. вырабатывает «сообщение покоя»).

Пример ММ-путей приведен на рис. 2. Данная конструкция отражает событийно управляемую природу объектно-ориентированного программирования и может быть взята в качестве основы для построения графовой модели класса или объектно-ориентированной программы в целом.

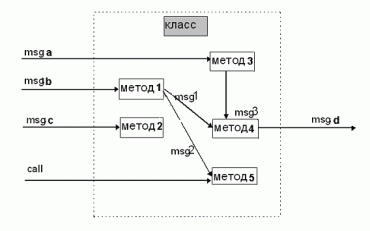


Рис. 2. Пример MM-путей и P-путей в графовой модели класса

Можно выделить четыре ММ-пути (1-4) и один P-путь (5):

1. msg a \to метод 3 \to msg 3 \to метод 4 \to msg d
2. msg b \to метод 1 \to msg 1 \to метод 4 \to msg d
3. msg b \to метод 1 \to msg 2 \to метод 5
4. msg c \to метод 2
5. call \to метод 5

Введем следующие обозначения:

1. Kmsg - число методов класса, обрабатывающих различные сообщения;
2. Kem- число методов класса, которые не закрыты от прямого вызова из других классов программы.

Если рассматривать класс как программу P, то можно выделить следующие отличия от программы, построенной по процедурному принципу:

1. Значение Kext (число точек входа, которые могут быть вызваны извне) определяется как сумма методов-обработчиков сообщений Kmsg (например, в MS Visual C++ обозначаются зарезервированным словом afx\_msg и используются для работы с картой сообщений класса) и тех методов, которые могут быть вызваны из других классов программы Kem. Это определяется самим разработчиком путем разграничения доступа к методам класса (с помощью ключевых слов разграничения доступа public, private, protect) при написании методов, а также назначении дружественных (friend) функций и дружественных классов. Таким образом,

Kext = Kmsg + Kem

1. Принцип соединения узлов в ГМП, отражающий два возможных типа вызовов методов класса (через ММ-пути и Р-пути), что приводит к новому наполнению для множества М требуемых элементов.
2. Методы (модули) непрозрачны для внешних объектов, что влечет за собой неприменимость механизма упрощения графа модуля, используемого для получения графа вызовов в процедурном программировании.

С учетом приведенных замечаний, информационные связи между модулями программного проекта получают новый физический смысл, а формула оценки сложности интеграционного тестирования класса Cls принимает вид:

V(Cls, C) = f (Kmsg, Kem)

В ходе интеграционного тестирования должны быть проверены все возможные внешние вызовы методов класса, как непосредственные обращения, так и вызовы, инициированные получением сообщений

Значение числа ММ-путей зависит от схемы обработки сообщений данным классом, что должно быть определено в спецификации класса.

Данные – члены класса (данные, описанные в самом классе, и унаследованные от классов-родителей видимые извне данные) рассматриваются как "глобальные переменные", они должны быть протестированы отдельно на основе принципов тестирования потоков данных.

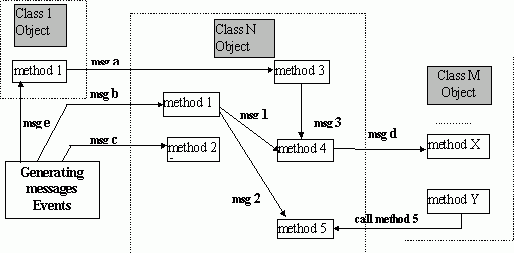


Рис. 3.Пример включения объекта в модель программного проекта, построенного с использованием MM-путей и P-путей

Когда класс программы P протестирован, объект данного класса может быть включен в общий граф G программного проекта, содержащий все ММ-пути и все вызовы методов классов и процедур, возможные в программе рис. 3.

Сложность интеграционного тестирования классов программы P, содержащей n классов, равна:

V(P, C) = ΣV(Clsi, C)

Формальным представлением описанного выше подхода к тестированию программного проекта служит классовая модель программного проекта, состоящая из дерева классов проекта рис.4 и модели каждого класса, входящего в программный проект рис. 5.

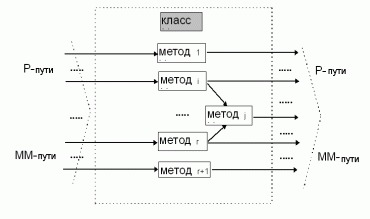


Рис. 5. Модель класса, входящего в программный проект

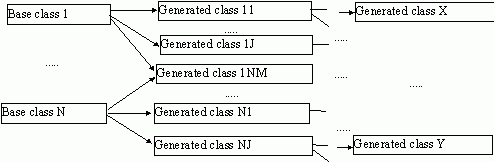


Рис. 4. Дерево классов проекта

Таким образом, определяется классовая модель проекта для тестирования объектно-ориентированной программы. Эта модель поддерживает итерационный инкрементальный процесс разработки программного обеспечения.

Методика проведения тестирования программы, представленной в виде классовой модели программного проекта, включает в себя несколько этапов, соответствующих уровням тестирования [рис. 5](http://www.intuit.ru/studies/courses/48/48/lecture/1434?page=2#image.6.5):

1. на первом уровне проводится тестирование методов каждого класса программы, что соответствует этапу модульного тестирования;
2. на втором уровне тестируются методы класса, которые образуют контекст интеграционного тестирования каждого класса;
3. на третьем уровне протестированный класс включается в общий контекст (дерево классов) программного проекта; становится возможным отслеживать реакцию программы на внешние события.

Второй и третий уровни рассматриваемой модели соответствуют этапу интеграционного тестирования.

Для третьего уровня важным оказывается понятие атомарной системной функции (АСФ). АСФ - это множество, состоящее из внешнего события на входе системы, реакции системы на это событие в виде одного или более ММ-путей и внешнего события на выходе системы. В общем случае внешнее выходное событие может быть нулевым, т. е. неаккуратно написанное программное обеспечение может не обеспечивать внешней реакции на действия пользователя. АСФ, состоящая из входного внешнего события, одного ММ-пути и выходного внешнего события, может быть взята в качестве модели для нити (thread). Тестирование подобной АСФ в рамках классовой модели ГМП реализуется довольно сложно, так как хотя динамическое взаимодействие нитей (потоков) в процессе исполнения естественно фиксируется в log-файлах, запоминающих результаты трассировки исполнения программ, оно же достаточно сложно отображается на классовой ГМП. Причина в том, что классовая модель ориентирована на отображение статических характеристик проекта, а в данном случае требуется отображение поведенческих характеристик. Как правило, тестирование взаимодействия нитей в ходе исполнения программного комплекса выносится на уровень системного тестирования и использует другие более приспособленные для описания поведения модели. Например, описание поведения программного комплекса средствами языков спецификаций MSC, SDL, UML.

Явный учет границ между интеграционным и системным уровнями тестирования дает преимущество при планировании работ на фазе тестирования, а возможность сочетать различные методы и критерии тестирования в ходе работы над программным проектом дает наилучшие результаты.

Объектно-ориентированный подход, ставший в настоящее время неявным стандартом разработки программных комплексов, позволяет широко использовать иерархическую модель программного проекта, приведенная на рис. 6 схема иллюстрирует способ применения.

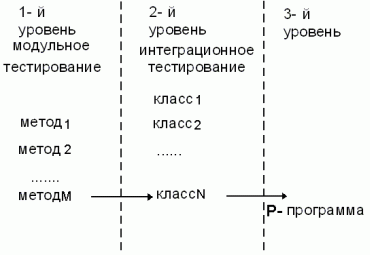


Рис. 6.Уровни тестирования классовой модели программного проекта

Каждый класс рассматривается как объект модульного и интеграционного тестирования. Сначала каждый метод класса тестируется как модуль по выбранному критерию C. Затем класс становится объектом интеграционного тестирования. Далее осуществляется интеграция всех методов всех классов в единую структуру - классовую модель проекта, где в общую ГМП протестированные модули входят в виде узлов (интерфейсов вызова) без учета их внутренней структуры, а их детальные описания образуют контекст всего программного проекта.

Сама технология объектно-ориентированного программирования (одним из определяющих принципов которой является инкапсуляция с возможностью ограничения доступа к данным и методам - членам класса) позволяет применить подобную трактовку вхождения модулей в общую ГМП. При этом тесты для отдельно рассмотренных классов используются снова, входя в общий набор тестов для программы P.

# **Список используемой литературы**

1. Котляров В. П. Основы тестирования программного обеспечения [Электронный ресурс] – Интернет-университет информационных технологий – Режим доступа: http:// www.intuit.ru/
2. Котляров В. П. Основы современного тестирования программного обеспечения, разработанного на C# : учебное пособие / В. П. Котляров, Т. В. Коликова – Санкт-Петербург, 2004 г. – 170 с.
3. Википедия – свободная энциклопедия [Электронный ресурс] – Режим доступа – http:// ru.wikipedia.org