Лабораторная работа №9

«Инспекция кода модулей проекта»

Цель работы получить практические навыки разработки модулей программной системы и интеграции этих модулей.

Теоретические сведения Термин «интеграция» относится к такой операции в процессе разработки ПО, при которой вы объединяете отдельные программные компоненты в единую систему. В небольших проектах интеграция может занять одно утро и заключаться в объединении горстки классов. В больших — могут потребоваться недели или месяцы, чтобы связать воедино весь набор программ. Независимо от размера задач в них применяются одни и те же принципы.

Тема интеграции тесно переплетается с вопросом последовательности конструирования. Порядок, в котором вы создаете классы или компоненты, влияет на порядок их интеграции: вы не можете интегрировать то, что еще не было создано. Последовательности интеграции и конструирования имеют большое значение.

Поскольку интеграция выполняется после того, как разработчик завершил модульное тестирование, и одновременно с системным тестированием, ее иногда считают операцией, относящейся к тестированию. Однако она достаточно сложна, и поэтому ее следует рассматривать как независимый вид деятельности.

Аккуратная интеграция обеспечивает:

- упрощенную диагностику дефектов;
- меньшее число ошибок;
- меньшее количество «лесов»;
- раннее создание первой работающей версии продукта;
- уменьшение общего времени разработки;
- лучшие отношения с заказчиком;
- улучшение морального климата;
- увеличение шансов завершения проекта;
- более надежные оценки графика проекта;
- более аккуратные отчеты о состоянии;
- лучшее качество кода;
- меньшее количество документации.

Интеграция программ выполняется посредством ***поэтапного*** или ***инкрементного*** подхода.

Поэтапная интеграция состоит из этапов, перечисленных ниже:

1. «Модульная разработка»: проектирование, кодирование, тестирование и отладка каждого класса.
2. «Системная интеграция»: объединение классов в одну огромную систему.
3. «Системная дезинтеграция»: тестирование и отладка всей системы.

Проблема поэтапной интеграции в том, что, когда классы в системе впервые соединяются вместе, неизбежно возникают новые проблемы и их причины могут быть в чем угодно. Поскольку у вас масса классов, которые никогда раньше не работали вместе, виновником может быть плохо протестированный класс, ошибка в интерфейсе между двумя классами или ошибка, вызванная взаимодействием двух классов. Все классы находятся под подозрением.

Неопределенность местонахождения любой из проблем сочетается с тем фактом, что все эти проблемы вдруг проявляют себя одновременно. Это заставляет вас иметь дело не только с проблемами, вызванными взаимодействием классов, но и другими ошибками, которые трудно диагностировать, так как они взаимодействуют.

Поэтому поэтапную интеграцию называют еще «интеграцией большого взрыва»

Поэтапную интеграцию нельзя начинать до начала последних стадий проекта, когда будут разработаны и протестированы все классы. Когда классы, наконец, будут объединены и проявится большое число ошибок, программисты тут же ударятся в паническую отладку вместо методического определения и исправления ошибок.

Для небольших программ — нет, а для крошечных — поэтапная интеграция может быть наилучшим подходом. Если программа состоит из двух-трех классов, поэтапная интеграция может сэкономить ваше время, если вам повезет. Но в большинстве случаев инкрементный подход будет лучше.

При **инкрементной интеграции** вы пишете и тестируете маленькие участки программы, а затем комбинируете эти кусочки друг с другом по одному. При таком подходе — по одному элементу за раз — вы выполняете перечисленные далее действия:

1. Разрабатываете небольшую, функциональную часть системы. Это может быть наименьшая функциональная часть, самая сложная часть, основная часть или их комбинация. Тщательно тестируете и отлаживаете ее. Она послужит скелетом, на котором будут наращиваться мускулы, нервы и кожа, составляющие остальные части системы.
2. Проектируете, кодируете, тестируете и отлаживаете класс.
3. Прикрепляете новый класс к скелету. Тестируете и отлаживаете соединение скелета и нового класса. Убеждаетесь, что эта комбинация работает, прежде чем переходить к добавлению нового класса. Если дело сделано, повторяете процесс, начиная с п. 2.

Инкрементный подход имеет массу преимуществ перед традиционным поэтапным подходом независимо от того, какую инкрементную стратегию вы используете:

Ошибки можно легко обнаружить Когда во время инкрементной интеграции возникает новая проблема, то очевидно, что к этому причастен новый класс. Либо его интерфейс с остальной частью программы неправилен, либо его взаимодействие с ранее интегрированными классами приводит к ошибке. В любом случае вы точно знаете, где искать проблему.

В таком проекте система раньше становится работоспособной Когда код интегрирован и способен выполняться, даже если система еще не пригодна к использованию, это выглядит так, будто это скоро произойдет. При инкрементной интеграции программисты раньше видят результаты своей работы, поэтому их моральное состояние лучше, чем в том случае, когда они подозревают, что их проект может никогда не сделать первый вдох.

Вы получаете улучшенный мониторинг состояния При частой интеграции реализованная и нереализованная функциональность видна с первого взгляда. Менеджеры будут иметь лучшее представление о состоянии проекта, видя, что 50% системы уже работает, а не слыша, что кодирование «завершено на 99%».

Вы улучшите отношения с заказчиком Если частая интеграция влияет на моральное состояние разработчиков, то она также оказывает влияние и на моральное состояние заказчика. Клиенты любят видеть признаки прогресса, а инкрементная интеграция предоставляет им такую возможность достаточно часто.

Системные модули тестируются гораздо полнее Интеграция начинается на ранних стадиях проекта. Вы интегрируете каждый класс по мере его готовности, а не ожидая одного внушительного мероприятия по интеграции в конце разработки. Программист тестирует классы в обоих случаях, но в качестве элемента общей системы они используются гораздо чаще при инкрементной, чем при поэтапной интеграции.

Вы можете создать систему за более короткое время Если интеграция тщательно спланирована, вы можете проектировать одну часть системы в то время, когда другая часть уже кодируется. Это не уменьшает общее число человеко-часов, требуемых для полного проектирования и кодирования, но позволяет выполнять часть работ параллельно, что является преимуществом в тех случаях, когда время имеет критическое значение.

При поэтапной интеграции вам не нужно планировать порядок создания компонентов проекта. Все компоненты интегрируются одновременно, поэтому вы можете разрабатывать их в любом порядке — главное, чтобы они все были готовы к часу X.

При инкрементной интеграции вы должны планировать более аккуратно. Большинство систем требует интеграции некоторых компонентов перед интеграцией других. Так что планирование интеграции влияет на планирование конструирования — порядок, в котором конструируются компоненты, должен обеспечивать порядок, в котором они будут интегрироваться.

Нисходящая интеграция

При нисходящей интеграции класс на вершине иерархии пишется и интегрируется первым. Вершина иерархии — это главное окно, управляющий цикл приложения, объект, содержащий метод main() в программе на Java, функция WinMain() в программировании для Microsoft Windows или аналогичные. Для работы этого верхнего класса пишутся заглушки. Затем, по мере интеграции классов сверху вниз, классы заглушек заменяются реальными.

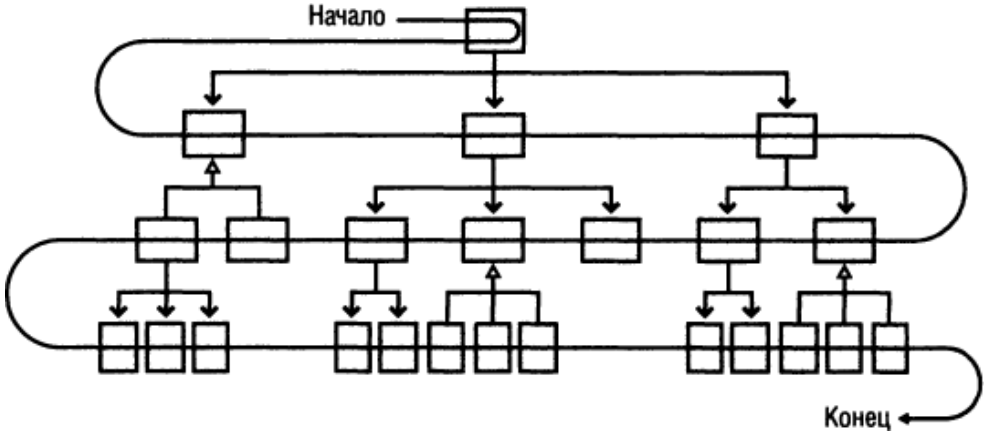
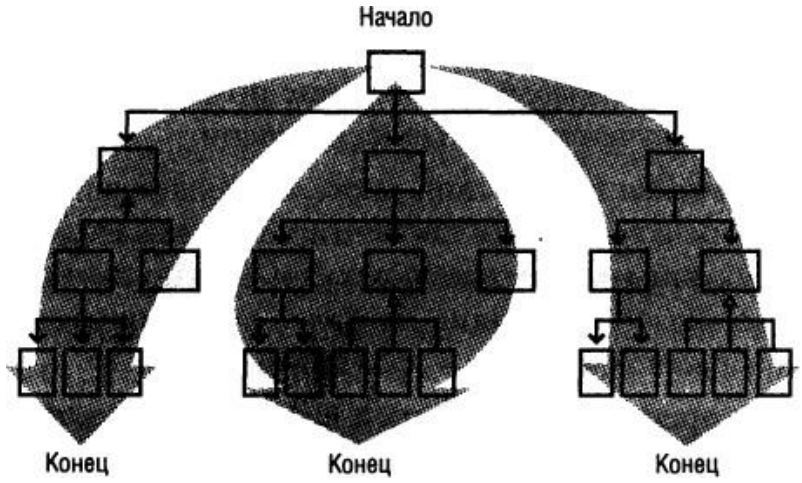


Рис.3 Нисходящая интеграция

При нисходящей интеграции вы создаете те классы, которые находятся на вершине иерархии, первыми, а те, что внизу, — последними.

Хорошей альтернативой нисходящей интеграции в чистом виде может стать подход с вертикальным секционированием.

Рис. 4 Вертикальное секционирование



При этом систему реализуют сверху вниз по частям, возможно, по очереди выделяя функциональные области и переходя от одной к другой.

Восходящая интеграция

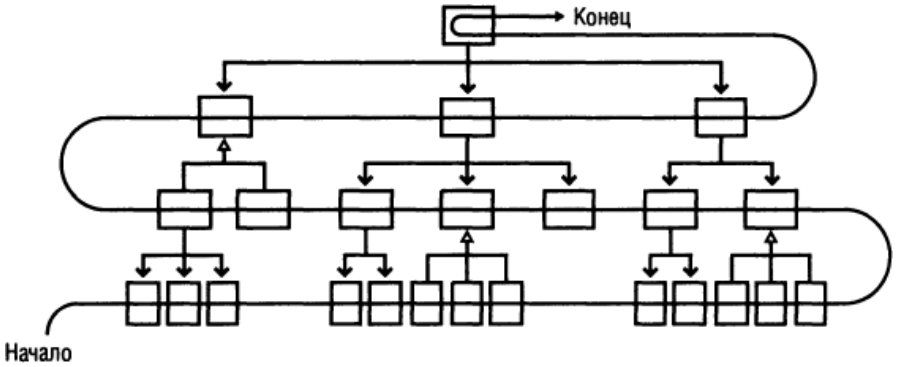


Рис.5 Восходящая интеграция

При восходящей интеграции вы пишете и интегрируете сначала классы, находящиеся в низу иерархии. Добавление низкоуровневых классов по одному, а не всех одновременно — вот что делает восходящую интеграцию инкрементной стратегией. Сначала вы пишете тестовые драйверы для выполнения низкоуровневых классов, а затем добавляете эти классы к тестовым драйверам, пристраивая их по мере готовности. Добавляя класс более высокого уровня, вы заменяете классы драйверов реальными.

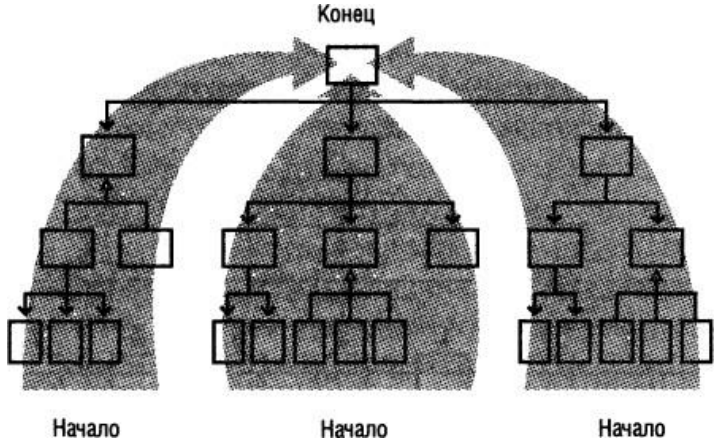


Рис. 6 Гибридный подход при восходящей интеграции

Как и нисходящую, восходящую интеграцию в чистом виде используют редко — вместо нее можно применять гибридный подход, реализующий секционную интеграцию.

Сэндвич-интеграция

Проблемы с нисходящей и восходящей интеграциями в чистом виде привели к тому, что некоторые эксперты стали рекомендовать сэндвич-подход.

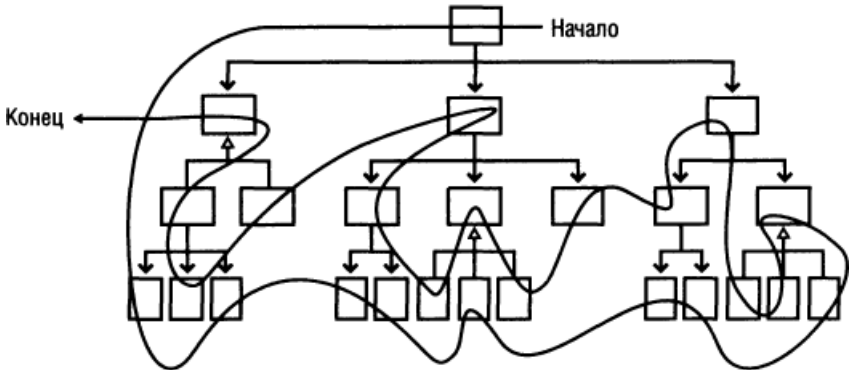


Рис. 7 Сэндвич-интеграция

Сначала вы объединяете высокоуровневые классы бизнес-объектов на вершине иерархии. Затем добавляете классы, взаимодействующие с аппаратной частью, и широко используемые вспомогательные классы в низу иерархии.

Напоследок вы оставляете классы среднего уровня.

Риск-ориентированная интеграция

Риск-ориентированную интеграцию, которую также называют «интеграцией, начиная с самых сложных частей» (hard part first integration), похожа на сэндвич-интеграцию тем, что пытается избежать проблем, присущих нисходящей или восходящей интеграциям в чистом виде. Кроме того, в ней также есть тенденция к объединению классов верхнего и нижнего уровней в первую очередь, оставляя классы среднего уровня напоследок. Однако суть в другом.

При риск-ориентированной интеграции вы определяете степень риска, связанную с каждым классом. Вы решаете, какие части системы будут самыми трудными, и реализуете их первыми.

Функционально-ориентированная интеграция

Еще один поход — интеграция одной функции в каждый момент времени. Под «функцией» понимается не нечто расплывчатое, а какое-нибудь поддающееся определению свойство системы, в которой выполняется интеграция.

Когда интегрируемая функция превышает по размерам отдельный класс, то «единица приращения» инкрементной интеграции становится больше отдельного класса. Это немного снижает преимущество инкрементного подхода в том плане, что уменьшает вашу уверенность об источнике новых ошибок. Однако если вы тщательно тестировали классы, реализующие эту функцию, перед интеграцией, то это лишь небольшой недостаток. Вы можете использовать стратегии инкрементной интеграции рекурсивно, сформировав сначала из небольших кусков отдельные свойства, а затем инкрементно объединив их в систему.

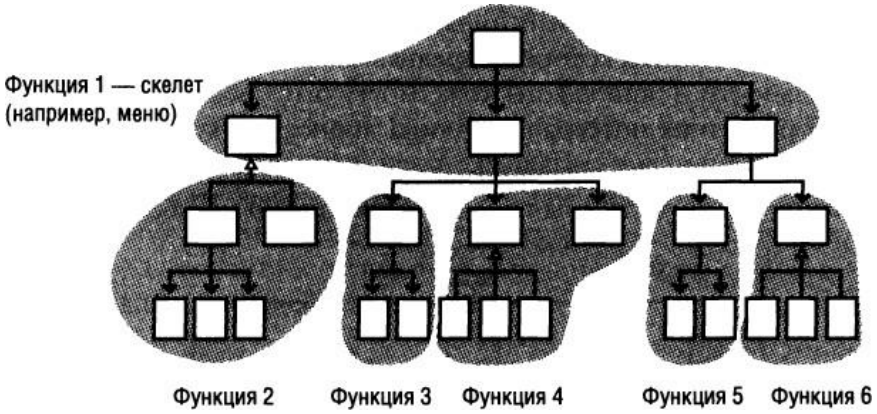


Рис. 8 Функционально-ориентированная интеграция

Обычно процесс начинается с формирования скелета, поскольку он способен поддерживать остальную функциональность. В интерактивной системе такой изначальной опцией может стать система интерактивного меню. Вы можете прикреплять остальную функциональность к той опции, которую интегрировали первой.

Т-образная интеграция

Последний подход, который часто упоминается в связи с проблемами нисходящей и восходящей методик, называется «Т-образной интеграцией». При таком подходе выбирается некоторый вертикальный слой, который разрабатывается и интегрируется раньше других. Этот слой должен проходить сквозь всю систему от начала до конца и позволять выявлять основные проблемы в допущениях, сделанных при проектировании системы. Реализовав этот вертикальный участок (и устранив все связанные с этим проблемы), можно разрабатывать основную канву системы (например, системное меню для настольного приложения). Этот подход часто комбинируют с риск-ориентированной и функционально-ориентированной интеграциями.

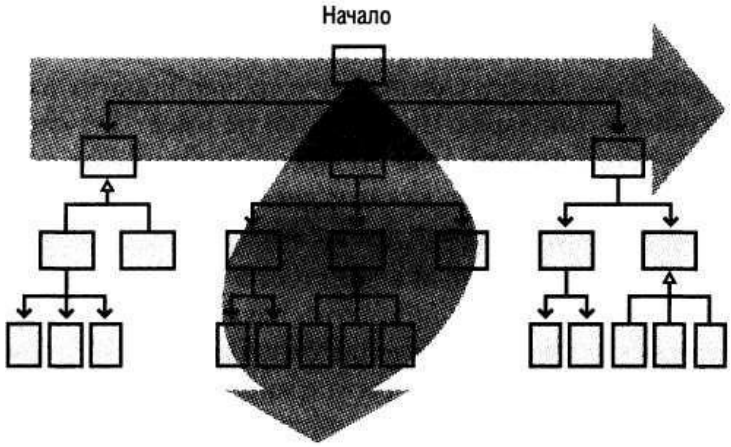


Рис. 9 Т-образная интеграция

Задание.

1. Оформить внешнюю спецификацию.
2. Составить в виде блок-схемы алгоритм решения задачи.
3. Спроектировать и разработать модули программы для решения задачи на любом алгоритмическом языке программирования.
4. Выполнить отладку и тестирование модулей программы.
5. Выполнить инкрементную интеграцию модулей с использованием одного из подходов.
6. Выполнить системное тестирование программы.
7. Оформить отчет по лабораторной работе.

Отчет по лабораторной работе должен включать:

1. Внешнюю спецификацию.
2. Алгоритм решения задачи.
3. Текст программы на языке программирования.
4. Набор тестов для отладки модулей программы.

5. Описание процесса интеграции модулей.

Задача. Задан двумерный массив размерности n×m. Отсортировать элементы строк массива по возрастанию значений, а затем отсортировать строки массива по возрастанию среднего арифметического элементов строк.

Реализовать сортировку разными способами и сравнить эффективность этих способов для разных исходных данных.

Контрольные вопросы.

1. Значение фазы интеграции программных модулей.
2. Подходы к интегрированию программных модулей.

Эффективность и оптимизация программ.