Глава 3: Разработка программного обеспечения

Из SWEBOK

Содержание

АКРОНИМЫ

API	Интерфейс прикладного программирования
КОТС	Коммерческий готовый
графический интерфейс	Графический пользовательский интерфейс
IDE	Интегрированная среда разработки
МОЙ БОГ	Группа управления объектами
POSIX	Портативная операционная система
TDD	Разработка через тестирование
UML	Единый язык моделирования

ВВЕДЕНИЕ

Термин «конструкция программного обеспечения» относится к подробному созданию работающего программного обеспечения путем сочетания кодирования, проверки, модульного тестирования, интеграционного тестирования и отладки. Область знаний по созданию программного обеспечения (КА) связана со всеми другими КА, но наиболее тесно она связана с проектированием и тестированием программного обеспечения, поскольку процесс создания программного обеспечения включает в себя значительный объем проектирования и тестирования программного обеспечения. Процесс использует выходные данные проектирования и предоставляет входные данные для тестирования («дизайн» и «тестирование» в данном случае относятся к действиям, а не к КА). Границы между проектированием, созданием и тестированием (если таковые имеются) будут различаться в зависимости от процессов жизненного цикла программного обеспечения, которые используются в проекте. Хотя некоторые детальные проекты могут быть выполнены до начала строительства, большая часть проектных работ выполняется во время строительных работ. Таким образом, КА «Конструирование программного обеспечения» тесно связано с КА «Дизайн программного обеспечения». На протяжении всего строительства инженеры-программисты проверяют свою работу как на модульное, так и на интеграционное тестирование. Таким образом, КА конструирования программного обеспечения также тесно связан с КА тестирования программного обеспечения. Разработка программного обеспечения обычно создает наибольшее количество элементов конфигурации, которыми необходимо управлять в программном проекте (исходные файлы, документация, тестовые примеры и т. д.). Таким образом, КА построения программного обеспечения также тесно связан с КА управления конфигурацией программного обеспечения. Хотя качество программного обеспечения важно во всех КА, код является конечным результатом программного проекта, и, таким образом, КА качества программного обеспечения тесно связан с КА конструирования программного обеспечения. Поскольку создание программного обеспечения требует знания алгоритмов и методов кодирования, он тесно связан с Computing Foundations КА, который занимается основами компьютерных наук, поддерживающими проектирование и создание программных продуктов. Это также связано с управлением проектами, поскольку управление строительством может представлять значительные трудности.

РАЗБИВКА ТЕМ ДЛЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

На рис. 3.1 дано графическое представление декомпозиции верхнего уровня структуры программного обеспечения КА.

1 Основы построения программного обеспечения

Основы построения программного обеспечения включают

- минимизация сложности
- ожидание перемен
- построение для проверки
- повторное использование
- стандарты в строительстве.

Первые четыре концепции применимы как к проектированию, так и к строительству. В следующих разделах определяются эти понятия и описывается их применение в строительстве.

1.1 Минимизация сложности

[1]

Большинство людей ограничены в своей способности удерживать сложные структуры и информацию в рабочей памяти, особенно в течение длительных периодов времени. Это оказывается основным фактором, влияющим на то, как люди передают намерения компьютерам, и приводит к одному из самых сильных побуждений в создании программного обеспечения: *минимизации*сложность. Необходимость снижения сложности применима практически ко всем аспектам построения программного обеспечения и особенно важна для тестирования конструкций программного обеспечения. В разработке программного обеспечения снижение сложности достигается за счет создания кода, который является простым и удобочитаемым, а не умным. Это достигается за счет использования стандартов (см. раздел 1.5 «Стандарты в строительстве»), модульного проектирования (см. раздел 3.1 «Конструктивное проектирование») и множества других специальных методов (см. раздел 3.3 «Кодирование»). Он также поддерживается методами обеспечения качества, ориентированными на строительство (см. раздел 3.7 «Качество строительства»).

1.2 Ожидание изменений

[1]

Большая часть программного обеспечения со временем изменится, и ожидание изменений влияет на многие аспекты создания программного обеспечения; изменения в среде, в которой работает программное обеспечение, также по-разному влияют на программное обеспечение. Предвидение изменений помогает инженерам-программистам создавать расширяемое программное обеспечение, а это значит, что они могут улучшать программный продукт, не нарушая базовой структуры. Предвидение изменений поддерживается многими специфическими методами (см. раздел 3.3, Программирование).

1.3 Конструирование для проверки

[1]

Конструирование для проверки означает создание программного обеспечения таким образом, чтобы ошибки могли быть легко обнаружены разработчиками программного обеспечения, пишущими программное обеспечение, а также тестировщиками и пользователями во время независимого тестирования и эксплуатации. Конкретные методы, которые поддерживают построение для проверки, включают, среди прочего, соблюдение стандартов кодирования для поддержки проверки кода и модульного тестирования, организацию кода для поддержки автоматического тестирования и ограничение использования сложных или трудных для понимания языковых структур.

1.4 Повторное использование

[2]

Повторное использование относится к использованию существующих ресурсов для решения различных задач. При создании программного обеспечения типичные активы, которые повторно используются, включают библиотеки, модули, компоненты, исходный код и готовые коммерческие активы (COTS). Повторное использование лучше всего практиковать систематически, в соответствии с четко определенным повторяемым процессом. Систематическое повторное использование может обеспечить значительное повышение производительности, качества и стоимости программного обеспечения. Повторное использование имеет два тесно связанных аспекта: «построение для повторного использования» и «построение с повторным использованием». Первое означает создание повторно

используемых программных активов, а второе означает повторное использование программных активов при построении нового решения. Повторное использование часто выходит за рамки проектов, что означает, что повторно используемые активы могут быть созданы в других проектах или организациях.

1.5 Стандарты в строительстве

[1]

Применение внешних или внутренних стандартов разработки во время строительства помогает достичь целей проекта по эффективности, качеству и стоимости. В частности, выбор допустимых подмножеств языков программирования и стандартов использования является важным средством достижения более высокого уровня безопасности. Стандарты, непосредственно затрагивающие вопросы строительства, включают

- методы коммуникации (например, стандарты форматов и содержания документов)
- языки программирования (например, языковые стандарты для таких языков, как Java и C++) * стандарты кодирования (например, стандарты для соглашений об именах, макета и отступов)
- платформы (например, стандарты интерфейса для вызовов операционной системы)
- инструменты (например, графические стандарты для нотаций, таких как UML (унифицированный язык моделирования)).

Использование внешних стандартов . Конструкция зависит от использования внешних стандартов для языков конструирования, инструментов конструирования, технических интерфейсов и взаимодействия между KA конструирования программного обеспечения и другими KA. Стандарты исходят из многочисленных источников, включая спецификации аппаратных и программных интерфейсов (например, Object Management Group (OMG)) и международных организаций (таких как IEEE или ISO). Использование внутренних стандартов. Стандарты также могут быть созданы на организационной основе на корпоративном уровне или для использования в конкретных проектах. Эти стандарты поддерживают координацию групповой деятельности, сводя к минимуму сложность, предвосхищая изменения и конструируя для проверки.

2 Управление строительством

2.1 Построение в моделях жизненного цикла

[1]

Для разработки программного обеспечения было создано множество моделей; некоторые делают упор на строительство больше, чем другие. Некоторые модели более линейны с точки зрения построения, например, водопадная модель и модель жизненного цикла с поэтапной доставкой. Эти модели рассматривают строительство как действие, которое происходит только после того, как выполнены значительные предварительные работы, включая детальную работу над требованиями, обширную работу по проектированию и детальное планирование. Более линейные подходы, как правило, подчеркивают действия, предшествующие строительству (требования и проектирование), и создают более четкое разделение между действиями. В этих моделях основным акцентом построения может быть кодирование. Другие модели более итеративны, например, эволюционное прототипирование и гибкая разработка. Эти подходы склонны рассматривать конструирование как действие, которое происходит одновременно с другими действиями по разработке программного обеспечения (включая требования, проектирование и планирование) или перекрывает их. Эти подходы, как правило, смешивают действия по проектированию, написанию кода и тестированию, и они часто рассматривают комбинацию действий как конструирование (см. KA «Управление программным обеспечением» и «Процесс разработки программного обеспечения»). Следовательно, то, что считается «строительством», в некоторой степени зависит от используемой модели жизненного цикла. В общем, создание программного обеспечения — это в основном написание кода и отладка, но оно также включает в себя планирование построения, детальное проектирование, модульное тестирование, интеграционное тестирование и другие действия. и они часто рассматривают комбинацию действий как создание (см. KA «Управление программным обеспечением» и «Процесс разработки программного обеспечения»). Следовательно, то, что считается «строительством», в некоторой степени зависит от используемой модели жизненного цикла. В общем, создание программного обеспечения — это в основном написание кода и отладка, но оно также включает в себя планирование построения, детальное проектирование, модульное тестирование, интеграционное тестирование и другие действия. и они часто рассматривают комбинацию действий как создание (см. KA «Управление программным обеспечением» и «Процесс разработки программного

обеспечения»). Следовательно, то, что считается «строительством», в некоторой степени зависит от используемой модели жизненного цикла. В общем, создание программного обеспечения — это в основном написание кода и отладка, но оно также включает в себя планирование построения, детальное проектирование, модульное тестирование, интеграционное тестирование и другие действия.

2.2 Планирование строительства

[1]

Выбор способа строительства является ключевым аспектом строительно-планировочной деятельности. Выбор метода строительства влияет на степень выполнения предварительных условий строительства, порядок их выполнения и степень, в которой они должны быть выполнены до начала строительных работ. Подход к построению влияет на способность проектной группы уменьшать сложность, предвидеть изменения и создавать для проверки. Каждая из этих целей может также решаться на уровне процесса, требований и проектирования, но на них будет влиять выбор метода построения. Планирование строительства также определяет порядок создания и интеграции компонентов, стратегию интеграции (например, поэтапная или пошаговая интеграция), процессы управления качеством программного обеспечения,

2.3 Строительные измерения

[1]

Могут быть измерены многочисленные строительные работы и артефакты, включая разработанный код, измененный код, повторно используемый код, уничтоженный код, сложность кода, статистику проверки кода, скорость исправления и обнаружения ошибок, усилия и планирование. Эти измерения могут быть полезны для целей управления строительством, обеспечения качества во время строительства и улучшения процесса строительства, среди прочего (см. Процесс разработки программного обеспечения (KA или более об измерении).

3 практических соображения

Конструирование — это деятельность, в которой разработчику программного обеспечения приходится иметь дело с иногда хаотичными и меняющимися ограничениями реального мира, и он или она должны делать это точно. Из-за влияния ограничений реального мира строительство в большей степени определяется практическими соображениями, чем некоторые другие KA, а разработка программного обеспечения, пожалуй, больше всего похожа на ремесло в строительной деятельности.

3.1 Строительный проект

[1]

В некоторых проектах значительная часть проектной деятельности отводится строительству, в то время как в других проектирование выделяется на этапе, явно ориентированном на проектирование. Независимо от точного распределения, некоторая детальная работа по проектированию будет выполняться на уровне строительства, и эта работа по проектированию, как правило, диктуется ограничениями, налагаемыми реальной проблемой, решаемой программным обеспечением. Точно так же, как строители, строящие физическую структуру, должны вносить небольшие изменения, чтобы учесть непредвиденные пробелы в планах строителя, разработчики программного обеспечения должны вносить изменения в меньшем или большем масштабе, чтобы конкретизировать детали программного проекта во время строительства. Детали проектной деятельности на уровне конструирования в основном такие же, как описано в KA Software Design, но они применяются в меньшем масштабе алгоритмов,

3.2 Строительные языки

[1]

Строительные языки включают в себя все формы общения, с помощью которых человек может указать выполнимое решение проблемы. Языки построения и их реализации (например, компиляторы) могут влиять на такие атрибуты качества программного обеспечения, как производительность, надежность, переносимость и т. д. Они могут серьезно способствовать уязвимостям безопасности.

Простейшим типом языка построения является язык *конфигурации* , на котором инженеры-программисты выбирают из ограниченного набора предопределенных параметров для создания новых или пользовательских установок программного обеспечения. Текстовые файлы конфигурации, используемые как в операционных системах Windows, так и в Unix, являются примерами этого, а списки выбора в стиле меню некоторых генераторов программ представляют собой еще один пример языка конфигурации. Языки инструментальных средств используются для создания приложений из элементов наборов инструментальных средств (интегрированных наборов многократно используемых частей для конкретных приложений); они более сложны, чем языки конфигурации.

Языки инструментальных средств могут быть явно определены как языки программирования приложений, или приложения могут просто подразумеваться набором интерфейсов инструментальных средств.

Языки сценариев — это обычно используемые виды языков прикладного программирования. В некоторых языках сценариев сценарии называются пакетными файлами или макросами. *Языки программирования* являются наиболее гибким типом языков конструирования. Они также содержат наименьшее количество информации о конкретных областях применения и процессах разработки, поэтому для их эффективного использования требуется наибольшая подготовка и навыки. Выбор языка программирования может иметь большое влияние на вероятность появления уязвимостей во время кодирования — например, некритическое использование C и C++ является сомнительным выбором с точки зрения безопасности. Существует три основных вида обозначений, используемых в языках программирования, а именно:

- лингвистический (например, C/C++, Java)
- формальный (например, Event-B)
- визуальные (например, MatLab).

Лингвистические нотации отличаются, в частности, использованием текстовых строк для представления сложных программных конструкций. Объединение текстовых строк в шаблоны может иметь синтаксис, подобный предложению. При правильном использовании каждая такая строка должна иметь сильную семантическую коннотацию, обеспечивающую немедленное интуитивное понимание того, что произойдет при выполнении программной конструкции.

Формальные обозначения меньше полагаются на интуитивные, повседневные значения слов и текстовых строк и больше на определения, подкрепленные точными, недвусмысленными и формальными (или математическими) определениями. Нотации формальных конструкций и формальные методы лежат в семантической основе большинства форм нотаций системного программирования, где точность, поведение во времени и проверяемость важнее, чем простота отображения на естественном языке. Формальные конструкции также используют точно определенные способы комбинирования символов, что позволяет избежать двусмысленности многих конструкций естественного языка.

Визуальные обозначения гораздо меньше полагаются на текстовые обозначения лингвистической и формальной конструкции и вместо этого полагаются на прямую визуальную интерпретацию и размещение визуальных объектов, которые представляют лежащее в основе программное обеспечение. Визуальное построение, как правило, несколько ограничено сложностью создания «сложных» утверждений с использованием только расположения значков на дисплее. Однако эти значки могут быть мощными инструментами в тех случаях, когда основной задачей программирования является просто создание и «настройка» визуального интерфейса для программы, детальное поведение которой является основным определением.

3.3 Кодирование

[1]

Следующие соображения применимы к деятельности по кодированию конструкции программного обеспечения:

- Методы создания понятного исходного кода, включая соглашения об именах и структуру исходного кода;
- Использование классов, перечисляемых типов, переменных, именованных констант и других подобных сущностей;
- Использование управляющих структур;

- Обработка ошибочных состояний — как ожидаемых, так и исключительных (например, ввод неверных данных);
- Предотвращение нарушений безопасности на уровне кода (например, переполнения буфера или границы индекса массива);
- Использование ресурсов за счет использования механизмов исключения и дисциплины при доступе к последовательно повторно используемым ресурсам (включая потоки и блокировки базы данных);
- Организация исходного кода (в операторы, подпрограммы, классы, пакеты или другие структуры);
- документация по коду;
- Настройка кода,

3.4 Испытания конструкции

[1]

Разработка включает в себя две формы тестирования, которые часто выполняются инженером-программистом, написавшим код:

- Модульное тестирование
- Интеграционное тестирование.

Целью тестирования конструкции является сокращение разрыва между временем, когда в код вносятся ошибки, и временем, когда эти ошибки обнаруживаются, тем самым снижая затраты на их исправление. В некоторых случаях тестовые случаи пишутся после написания кода. В других случаях тестовые случаи могут быть созданы до написания кода. Тестирование построения обычно включает в себя подмножество различных типов тестирования, описанных в KA Тестирования программного обеспечения. Например, тестирование конструкции обычно не включает системное тестирование, альфа-тестирование, бета-тестирование, стресс-тестирование, тестирование конфигурации, тестирование удобства использования или другие более специализированные виды тестирования. На тему тестирования конструкции опубликованы два стандарта: Стандарт IEEE 829-1998, *Стандарт IEEE для документации по тестированию программного обеспечения* .и IEEE Standard 1008-1987, *IEEE Standard for Software Unit Testing* .

(См. разделы 2.1.1. Модульное тестирование и 2.1.2. Интеграционное тестирование в KA по тестированию программного обеспечения для более специализированного справочного материала.)

3.5 Конструкция для повторного использования

[2]

Конструкция для повторного использования создает программное обеспечение, которое потенциально может быть повторно использовано в будущем для текущего проекта или других проектов, имеющих широкую мультисистемную перспективу. Строительство для повторного использования обычно основано на анализе изменчивости и проектировании. Чтобы избежать проблемы клонирования кода, желательно инкапсулировать многократно используемые фрагменты кода в хорошо структурированные библиотеки или компоненты. Задачи, связанные с созданием программного обеспечения для повторного использования во время кодирования и тестирования, следующие:

- Реализация вариативности с помощью таких механизмов, как параметризация, условная компиляция, шаблоны проектирования и т. д.
- Инкапсуляция вариативности, упрощающая настройку и настройку программных ресурсов.
- Тестирование изменчивости, обеспечиваемой многократно используемыми программными активами.
- Описание и публикация повторно используемых программных активов.

3.6 Строительство с повторным использованием

[2]

Создание с повторным использованием означает создание нового программного обеспечения с повторным использованием существующих программных активов. Самый популярный метод повторного использования — это повторное использование кода из библиотек, предоставляемых

языком, платформой, используемыми инструментами или репозиторием организации. Помимо этого, разрабатываемые сегодня приложения широко используют множество библиотек с открытым исходным кодом. Повторно используемое и готовое программное обеспечение часто имеет те же или более высокие требования к качеству, что и вновь разработанное программное обеспечение (например, уровень безопасности). Задачи, связанные с созданием программного обеспечения с повторным использованием при кодировании и тестировании, следующие:

- Выбор многократно используемых единиц, баз данных, тестовых процедур или тестовых данных.
- Оценка кода или проверка возможности повторного использования.
- Интеграция повторно используемых программных активов в текущее программное обеспечение.
- Отчеты об информации о повторном использовании нового кода, тестовых процедур или тестовых данных.

3.7 Качество строительства

[1]

В дополнение к ошибкам, возникающим из-за требований и дизайна, ошибки, возникшие во время строительства, могут привести к серьезным проблемам с качеством, например к уязвимостям в системе безопасности. Сюда входят не только сбои в функциях безопасности, но и другие сбои, которые позволяют обойти эту функциональность, а также другие недостатки или нарушения безопасности. Существует множество методов для обеспечения качества кода по мере его создания. Основные методы, используемые для обеспечения качества строительства, включают

- модульное тестирование и интеграционное тестирование (см. раздел 3.4, Конструктивное тестирование)
- разработка с первым тестом (см. раздел 2.2 в KA по тестированию программного обеспечения)
- использование утверждений и защитного программирования
- отладка
- инспекции
- технические обзоры, в том числе обзоры, ориентированные на безопасность (см. раздел 2.3.2 в KA качества программного обеспечения)
- статический анализ (см. раздел 2.3 KA качества программного обеспечения)

Конкретный выбранный метод или методы зависят от характера создаваемого программного обеспечения, а также от набора навыков инженеров-программистов, выполняющих строительные работы. Программисты должны знать передовой опыт и распространенные уязвимости — например, из общепризнанных списков распространенных уязвимостей. Автоматический статический анализ кода на наличие уязвимостей безопасности доступен для нескольких распространенных языков программирования и может использоваться в проектах, критически важных с точки зрения безопасности.

Деятельность по обеспечению качества в строительстве отличается от других видов деятельности по обеспечению качества своей направленностью. Действия по обеспечению качества построения сосредоточены на коде и артефактах, тесно связанных с кодом, таких как детальный проект, в отличие от других артефактов, менее тесно связанных с кодом, таких как требования, проекты высокого уровня и планы.

3.8 Интеграция

[1]

Ключевым действием при построении является интеграция индивидуально созданных подпрограмм, классов, компонентов и подсистем в единую систему. Кроме того, может потребоваться интеграция конкретной программной системы с другими программными или аппаратными системами. Проблемы, связанные с интеграцией конструкции, включают планирование последовательности, в которой компоненты будут интегрированы, определение необходимого аппаратного обеспечения, создание каркаса для поддержки промежуточных версий программного обеспечения, определение степени тестирования и качественной работы, выполняемой над компонентами до их интеграции, и определение точки в проекте, на которых тестируются промежуточные версии программного обеспечения. Программы могут быть интегрированы посредством либо поэтапного, либо поэтапного подхода. Поэтапная интеграция, также называемая интеграцией «большого взрыва», влечет за собой отсрочку

интеграции составных частей программного обеспечения до тех пор, пока не будут завершены все части, предназначенные для выпуска в версии. Считается, что инкрементальная интеграция предлагает множество преимуществ по сравнению с традиционной поэтапной интеграцией, например, более простое обнаружение ошибок, улучшенный мониторинг прогресса, более ранняя поставка продукта и улучшение отношений с клиентами. При поэтапной интеграции разработчики пишут и тестируют программу небольшими частями, а затем объединяют части по одной. Для обеспечения поэтапной интеграции обычно требуется дополнительная тестовая инфраструктура, такая как заглушки, драйверы и фиктивные объекты. Создавая и интегрируя по одной единице за раз (например, класс или компонент), процесс построения может обеспечить раннюю обратную связь с разработчиками и клиентами. Другие преимущества инкрементной интеграции включают более легкое обнаружение ошибок,

4 Технологии строительства

4.1 Дизайн и использование API

[3]

Интерфейс прикладного программирования (API) — это набор сигнатур, которые экспортируются и доступны пользователям библиотеки или платформы для написания своих приложений. Помимо сигнатур, API всегда должен включать заявления об эффектах и/или поведении программы (т. е. ее семантике). Проектирование API должно быть направлено на то, чтобы сделать API простым для изучения и запоминания, привести к читабельному коду, трудно использовать не по назначению, легко расширять, быть полным и поддерживать обратную совместимость. Поскольку API-интерфейсы обычно дольше своих реализаций для широко используемых библиотек или фреймворков, желательно, чтобы API-интерфейсы были простыми и оставались стабильными, чтобы облегчить разработку и обслуживание клиентских приложений. Использование API включает в себя процессы выбора, обучения, тестирования, интеграции,

4.2 Проблемы объектно-ориентированного времени выполнения

[1]

Объектно-ориентированные языки поддерживают ряд механизмов времени выполнения, включая полиморфизм и отражение. Эти механизмы времени выполнения повышают гибкость и адаптивность объектно-ориентированных программ. Полиморфизм — это способность языка поддерживать общие операции, не зная до момента выполнения, какие конкретные объекты будет включать в себя программа. Поскольку программе заранее не известны точные типы объектов, точное поведение определяется во время выполнения (так называемое динамическое связывание). Рефлексия — это способность программы наблюдать и изменять свою собственную структуру и поведение во время выполнения. Отражение позволяет проверять классы, интерфейсы, поля и методы во время выполнения, не зная их имен во время компиляции.

4.3 Параметризация и обобщения

[4]

Параметризованные типы, также известные как дженерики (Ada, Eiffel) и шаблоны (C++), позволяют определять тип или класс без указания всех других типов, которые он использует. Неуказанные типы предоставляются в качестве параметров в момент использования. Параметризованные типы предоставляют третий способ (в дополнение к наследованию классов и композиции объектов) для составления поведения в объектно-ориентированном программном обеспечении.

4.4 Утверждения, разработка по контракту и защитное программирование

[1]

Утверждение — это исполняемый предикат, помещаемый в программу (обычно подпрограмму или макрос), который позволяет выполнять проверки программы во время выполнения. Утверждения особенно полезны в программах с высокой надежностью. Они позволяют программистам быстрее избавляться от несоответствующих предположений об интерфейсе, ошибок, возникающих при изменении кода, и т. д. Утверждения обычно компилируются в код во время разработки, а затем компилируются из кода, чтобы не снижать производительность. Проектирование по контракту — это подход к разработке, при котором предварительные и постусловия включаются в каждую процедуру. Когда используются предусловия и постусловия, говорят, что каждая подпрограмма или класс образуют

контракт с остальной частью программы. Кроме того, контракт обеспечивает точную спецификацию семантики рутины, и, таким образом, помогает понять его поведение. Считается, что проектирование по контракту улучшает качество разработки программного обеспечения.*Защитное программирование* означает защиту процедуры от нарушения неверными входными данными. Общие способы обработки недопустимых входных данных включают проверку значений всех входных параметров и принятие решения о том, как обрабатывать неверные входные данные. Утверждения часто используются в защитном программировании для проверки входных значений.

4.5 Обработка ошибок, обработка исключений и отказоустойчивость

[1]

Способ обработки ошибок влияет на способность программного обеспечения соответствовать требованиям, связанным с правильностью, надежностью и другими нефункциональными атрибутами. Утверждения иногда используются для проверки ошибок. Также используются другие методы обработки ошибок, такие как возврат нейтрального значения, замена следующей части допустимых данных, регистрация предупреждающего сообщения, возврат кода ошибки или завершение работы программного обеспечения. Исключения используются для обнаружения и обработки ошибок или исключительных событий. Базовая структура исключения заключается в том, что подпрограмма использует throw для создания обнаруженного исключения, а блок обработки исключений перехватывает исключение в блоке try-catch. Блок try-catch может обработать ошибочное условие в подпрограмме или вернуть управление вызывающей подпрограмме. Политики обработки исключений должны быть тщательно разработаны в соответствии с общими принципами, такими как включение в сообщение об исключении всей информации, которая привела к исключению, избегание пустых блоков catch, знание исключений, которые генерирует код библиотеки, возможно, создание централизованного генератора отчетов об исключениях и стандартизация использования программы. исключений. Отказоустойчивость — это набор методов, которые повышают надежность программного обеспечения путем обнаружения ошибок и последующего восстановления после них, если это возможно, или сдерживания их последствий, если восстановление невозможно. Наиболее распространенные стратегии отказоустойчивости включают резервное копирование и повторную попытку, использование вспомогательного кода, использование алгоритмов голосования и замену ошибочного значения фальшивым значением, которое будет иметь благоприятный эффект. избегание пустых блоков catch, знание исключений, которые генерирует код библиотеки, возможно, создание централизованного генератора отчетов об исключениях и стандартизация использования исключений в программе. Отказоустойчивость — это набор методов, которые повышают надежность программного обеспечения путем обнаружения ошибок и последующего восстановления после них, если это возможно, или сдерживания их последствий, если восстановление невозможно. Наиболее распространенные стратегии отказоустойчивости включают резервное копирование и повторную попытку, использование вспомогательного кода, использование алгоритмов голосования и замену ошибочного значения фальшивым значением, которое будет иметь благоприятный эффект. избегание пустых блоков catch, знание исключений, которые генерирует код библиотеки, возможно, создание централизованного генератора отчетов об исключениях и стандартизация использования исключений в программе. Отказоустойчивость — это набор методов, которые повышают надежность программного обеспечения путем обнаружения ошибок и последующего восстановления после них, если это возможно, или сдерживания их последствий, если восстановление невозможно. Наиболее распространенные стратегии отказоустойчивости включают резервное копирование и повторную попытку, использование вспомогательного кода, использование алгоритмов голосования и замену ошибочного значения фальшивым значением, которое будет иметь благоприятный эффект. Отказоустойчивость — это набор методов, которые повышают надежность программного обеспечения путем обнаружения ошибок и последующего восстановления после них, если это возможно, или сдерживания их последствий, если восстановление невозможно. Наиболее распространенные стратегии отказоустойчивости включают резервное копирование и повторную попытку, использование вспомогательного кода, использование алгоритмов голосования и замену ошибочного значения фальшивым значением, которое будет иметь благоприятный эффект. Отказоустойчивость — это набор методов, которые повышают надежность программного обеспечения путем обнаружения ошибок и последующего восстановления после них, если это возможно, или сдерживания их последствий, если восстановление невозможно. Наиболее распространенные стратегии отказоустойчивости включают резервное копирование и повторную попытку, использование вспомогательного кода, использование алгоритмов голосования и замену ошибочного значения фальшивым значением, которое будет иметь благоприятный эффект.

4.6 Исполняемые модели

[5]

[5]

Исполняемые модели абстрагируются от деталей конкретных языков программирования и решений об организации программного обеспечения. В отличие от традиционных программных моделей, спецификация, построенная на исполняемом языке моделирования, таком как xUML (исполняемый UML), может быть развернута в различных программных средах без изменений. Компилятор исполняемой модели (преобразователь) может превратить исполняемую модель в реализацию, используя набор решений о целевой аппаратной и программной среде. Таким образом, создание исполняемых моделей можно рассматривать как способ создания исполняемого программного обеспечения. Исполняемые модели являются одной из основ, поддерживающих инициативу Model-Driven Architecture (MDA) группы управления объектами (OMG). Исполняемая модель — это способ полностью определить независимую от платформы модель (PIM); PIM — это модель решения проблемы, которая не опирается ни на какие технологии реализации. Затем можно создать специфическую для платформы модель (PSM), которая представляет собой модель, содержащую детали реализации, путем объединения PIM и платформы, на которой она основана.

4.7 Методы конструирования на основе состояний и таблиц

[1]

Программирование на основе состояний или программирование на основе автоматов — это технология программирования, использующая конечные автоматы для описания поведения программы. Графы переходов конечного автомата используются на всех этапах разработки программного обеспечения (спецификация, реализация, отладка и документация). Основная идея состоит в том, чтобы строить компьютерные программы так же, как автоматизация технологических процессов. Программирование на основе состояний обычно сочетается с объектно-ориентированным программированием, образуя новый составной подход, называемый объектно-ориентированным программированием на основе состояний. Табличный метод — это схема, в которой для поиска информации используются таблицы, а не логические операторы (такие как if и case). Используемый в соответствующих обстоятельствах код, управляемый таблицами, проще, чем сложная логика, и его легче модифицировать. При использовании табличных методов

4.8 Конфигурация во время выполнения и интернационализация

[1]

Чтобы добиться большей гибкости, программа часто строится так, чтобы поддерживать время позднего связывания своих переменных. Конфигурация во время выполнения — это метод, который связывает значения переменных и параметры программы во время работы программы, обычно путем обновления и чтения файлов конфигурации в режиме «точно в срок». Интернационализация — это техническая деятельность по подготовке программы, обычно интерактивного программного обеспечения, для поддержки нескольких локалей. Соответствующая деятельность, локализация, представляет собой деятельность по изменению программы для поддержки определенного местного языка. Интерактивное программное обеспечение может содержать десятки или сотни подсказок, индикаторов состояния, справочных сообщений, сообщений об ошибках и т. д. Процессы проектирования и построения должны учитывать вопросы, связанные со строками и наборами символов, включая то, какой набор символов следует использовать, какие типы строк используются,

4.9 Обработка ввода на основе грамматики

[1] [6]

Обработка ввода на основе грамматики включает синтаксический анализ или синтаксический анализ потока входных токенов. Он включает в себя создание структуры данных (называемой деревом синтаксического анализа или синтаксическим деревом), представляющей входные данные. Неупорядоченный обход дерева синтаксического анализа обычно дает только что проанализированное выражение. Анализатор проверяет таблицу символов на наличие определяемых программистом переменных, которые заполняют дерево. После построения дерева синтаксического анализа программа использует его в качестве входных данных для вычислительных процессов.

4.10 Примитивы параллелизма

[7]

Примитив синхронизации — это программная абстракция, предоставляемая языком программирования или операционной системой, которая обеспечивает параллелизм и синхронизацию. К хорошо известным примитивам параллелизма относятся семафоры, мониторы и мьютексы. Семафор — это защищенная переменная или абстрактный тип данных, который обеспечивает простую, но полезную абстракцию для управления доступом к общему ресурсу несколькими процессами или потоками в параллельной среде программирования. Монитор — это абстрактный тип данных, представляющий набор определяемых программистом операций, которые выполняются с взаимным исключением. Монитор содержит объявление общих переменных и процедур или функций, которые работают с этими переменными. Конструкция монитора гарантирует, что в каждый момент времени в мониторе активен только один процесс.

4.11 Промежуточное ПО

[3] [6]

Промежуточное ПО — это широкая классификация программного обеспечения, которое предоставляет услуги выше уровня операционной системы, но ниже уровня прикладных программ. ПО промежуточного слоя может предоставлять контейнеры времени выполнения для программных компонентов, чтобы обеспечить передачу сообщений, сохранение и прозрачное расположение в сети. Промежуточное ПО можно рассматривать как связующее звено между компонентами, использующими промежуточное ПО. Современное ПО промежуточного слоя, ориентированное на работу с сообщениями, обычно предоставляет корпоративную служебную шину (ESB), которая поддерживает сервисно-ориентированное взаимодействие и связь между несколькими программными приложениями.

4.12 Методы построения распределенного программного обеспечения

[7]

Распределенная система — это совокупность физически отдельных, возможно, гетерогенных компьютерных систем, объединенных в сеть для предоставления пользователям доступа к различным ресурсам, поддерживаемым системой. Разработка распределенного программного обеспечения отличается от разработки традиционного программного обеспечения такими проблемами, как параллелизм, связь и отказоустойчивость. Распределенное программирование обычно относится к одной из нескольких основных архитектурных категорий: клиент-сервер, 3-уровневая архитектура, n-уровневая архитектура, распределенные объекты, слабая связь или тесная связь (см. раздел 14.3 Computing Foundations KA и раздел 3.2 Программного обеспечения). Дизайн КА).

4.13 Построение гетерогенных систем

[6]

Гетерогенные системы состоят из множества специализированных вычислительных блоков разных типов, таких как цифровые сигнальные процессоры (DSP), микроконтроллеры и периферийные процессоры. Эти вычислительные блоки управляются независимо друг от друга и взаимодействуют друг с другом. Встроенные системы обычно представляют собой гетерогенные системы. При проектировании разнородных систем может потребоваться сочетание нескольких языков спецификаций для проектирования различных частей системы, другими словами, программно-аппаратный код. Ключевые проблемы включают многоязычную проверку, совместное моделирование и взаимодействие. Во время кодирования аппаратного/программного обеспечения разработка программного обеспечения и разработка виртуального оборудования происходят одновременно посредством пошаговой декомпозиции. Аппаратная часть обычно моделируется в программируемых вентильных матрицах (FPGA) или специализированных интегральных схемах ASIC). Программная часть переведена на язык программирования низкого уровня.

4.14 Анализ производительности и настройка

[1]

Эффективность кода, определяемая архитектурой, детальными проектными решениями, а также структурой данных и выбором алгоритма, влияет на скорость и размер выполнения. Анализ производительности — это исследование поведения программы с использованием информации, собранной во время выполнения программы, с целью выявления возможных горячих точек в программе, которые необходимо улучшить. Настройка кода, повышающая производительность на уровне кода, — это практика изменения правильного кода таким образом, чтобы он работал более эффективно.

Настройка кода обычно включает в себя лишь небольшие изменения, затрагивающие один класс, одну подпрограмму или, чаще, несколько строк кода. Доступен богатый набор методов настройки кода, в том числе для настройки логических выражений, циклов, преобразований данных, выражений и подпрограмм.

4.15 Стандарты платформы

[6] [7]

Стандарты платформ позволяют программистам разрабатывать переносимые приложения, которые можно запускать в совместимых средах без изменений. Стандарты платформы обычно включают набор стандартных служб и API, которые должны быть реализованы в совместимых реализациях платформы. Типичными примерами стандартов платформы являются Java 2 Platform Enterprise Edition (J2EE) и стандарт POSIX для операционных систем (Portable Operating System Interface), который представляет собой набор стандартов, реализованных в основном для операционных систем на основе UNIX.

4.16 Тестовое программирование

[1]

Программирование, ориентированное на тестирование (также известное как разработка через тестирование — TDD) — это популярный стиль разработки, при котором тестовые примеры записываются до написания любого кода. Программирование, основанное на тестировании, обычно может обнаруживать дефекты раньше и исправлять их легче, чем традиционные стили программирования. Кроме того, написание тестовых примеров сначала заставляет программистов думать о требованиях и дизайне до написания кода, тем самым быстрее выявляя требования и проблемы дизайна.

5 инструментов разработки программного обеспечения

5.1 Среда разработки

[1]

Среда разработки, или интегрированная среда разработки (IDE), предоставляет программистам комплексные средства для создания программного обеспечения путем интеграции набора инструментов разработки. Выбор среды разработки может повлиять на эффективность и качество создания программного обеспечения. В дополнение к базовым функциям редактирования кода современные IDE часто предлагают другие функции, такие как компиляция и обнаружение ошибок из редактора, интеграция с системой управления исходным кодом, инструменты сборки/тестирования/отладки, сжатые или структурированные представления программ, автоматические преобразования кода и поддержка. для рефакторинга.

5.2 Разработчики графического интерфейса

[1]

Построитель GUI (графического пользовательского интерфейса) — это инструмент разработки программного обеспечения, который позволяет разработчику создавать и поддерживать GUI в режиме WYSIWYG (что видишь, то и получаешь). Построитель графического интерфейса обычно включает в себя визуальный редактор, позволяющий разработчику создавать формы и окна и управлять компоновкой виджетов путем перетаскивания и настройки параметров. Некоторые разработчики GUI могут автоматически генерировать исходный код, соответствующий визуальному дизайну GUI. Поскольку текущие приложения с графическим интерфейсом обычно следуют стилю, управляемому событиями (в котором поток программы определяется событиями и обработкой событий), инструменты построения графического интерфейса обычно предоставляют помощников по генерации кода, которые автоматизируют наиболее повторяющиеся задачи, необходимые для обработки событий. Вспомогательный код связывает виджеты с исходящими и входящими событиями, которые запускают функции, обеспечивающие логику приложения. Некоторые современные IDE предоставляют встроенные конструкторы графического интерфейса или подключаемые модули для построения графического интерфейса. Есть также много автономных конструкторов графического интерфейса.

5.3 Инструменты модульного тестирования

[1] [2]

Модульное тестирование проверяет функционирование программных модулей изолированно от других программных элементов, которые можно тестировать отдельно (например, классы, подпрограммы, компоненты). Модульное тестирование часто автоматизировано. Разработчики могут использовать инструменты и среды модульного тестирования для расширения и создания автоматизированной среды тестирования. С помощью инструментов и сред модульного тестирования разработчик может закодировать критерии в тесте, чтобы проверить правильность модуля в различных наборах данных. каждый отдельный тест реализован как объект, и средство запуска тестов запускает все тесты. Во время выполнения теста эти неудачные тестовые случаи будут автоматически помечены, и о них будет сообщено.

5.4 Инструменты профилирования, анализа производительности и нарезки

[1]

Инструменты анализа производительности обычно используются для поддержки настройки кода. Наиболее распространенными инструментами анализа производительности являются инструменты профилирования. Инструмент профилирования выполнения отслеживает код во время его выполнения и записывает, сколько раз выполняется каждый оператор или сколько времени программа тратит на каждый оператор или путь выполнения. Профилирование кода во время его выполнения дает представление о том, как работает программа, где находятся горячие точки и на чем разработчикам следует сосредоточить усилия по настройке кода. может повлиять на значения указанных переменных в некоторой интересующей точке, что называется критерием среза. Нарезка программы может использоваться для обнаружения источника ошибок, понимания программы и анализа оптимизации.

ДАЛЬНЕЙШИЕ ЧТЕНИЯ

Стандарт IEEE 1517-2010 Standard for Information Technology — System and Software Life Cycle Processes — Reuse Processes , IEEE, 2010 [8].

Этот стандарт определяет процессы, действия и задачи, которые должны применяться на каждом этапе жизненного цикла программного обеспечения, чтобы обеспечить создание программного продукта из повторно используемых активов. Он охватывает концепцию разработки на основе повторного использования и процессы построения для повторного использования и построения с повторным использованием.

Стандарт IEEE 12207-2008 (он же ISO/IEC 12207:2008) Стандарт системной и программной инженерии — процессы жизненного цикла программного обеспечения , IEEE, 2008 [9].

Этот стандарт определяет ряд процессов разработки программного обеспечения, включая процесс создания программного обеспечения, процесс интеграции программного обеспечения и процесс повторного использования программного обеспечения.

ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

[1] С. МакКоннелл, *Code Complete* , 2-е изд., Microsoft Press, 2004.

[2] И. Соммервиль, *Разработка программного обеспечения* , 9-е изд., Addison-Wesley, 2011.

[3] П. Клементс и др., *Документирование архитектуры программного обеспечения: взгляды и не только* , 2-е изд., Pearson Education, 2010.

[4] Э. Гамма и др., *Шаблоны проектирования: элементы многоразового объектно-ориентированного программного обеспечения* , 1-е изд., Addison-Wesley Professional, 1994.

[5] SJ Mellor и MJ Balcer, *Executable UML: A Foundation for Model-Driven Architecture* , 1-е изд., Addison-Wesley, 2002.

[6] Л. Нулл и Дж. Лобур, «Основы компьютерной организации и архитектуры», 2-е изд., издательство Jones and Bartlett Publishers, 2006.

[7] А. Зильбершац, П. Б. Гэлвин и Г. Гане, *Концепции операционных систем* , 8-е изд., Wiley, 2008.

[8] IEEE, «Стандарт IEEE 1517-2010 для информационных технологий — процессы жизненного цикла систем и программного обеспечения — процессы повторного использования», IEEE, 2010.

[9] IEEE, «Стандарт IEEE 12207-2008 (он же ISO/IEC 12207:2008) Стандарт системной и программной инженерии — процессы жизненного цикла программного обеспечения», IEEE, 2008.

Получено с " http://swebokwiki.org/index.php?title=Chapter_3:_Software_Construction&oldid=511 "

- Последнее изменение этой страницы состоялось 24 августа 2015 г., в 23:48.