МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Институт сферы обслуживания и предпринимательства (филиал)
федерального государственного бюджетного образовательного
учреждения высшего образования «Донской государственный
технический университет» в г. Шахты Ростовской области
(ИСОиП (филиал) ДГТУ в г. Шахты)

КОЛЛЕДЖ ЭКОНОМИКИ И СЕРВИСА

На правах рукописи

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические указания для специальности
09.02.03 «Программирование в компьютерных системах»

Рассмотрены и рекомендованы для
использования в учебном процессе на
заседании педагогического совета
Протокол № 1от «31» сентября 2018 г

Шахты
ИСОиП (филиал) ДГТУ в г. Шахты
2019

Составитель:
к.ф.н., доцент кафедры «Информатика» О.С. Бурякова

Рецензенты:
гл. бухгалтер ООО «АТП» в г. Шахты Л.И. Долженко преподаватель высш. категории КЭС ИСОиП (филиал) ДГТУ в г. Шахты И.Ю. Бабенко

Технология разработки программного обеспечения : метод. указ. по дисциплине для подгот. обуч. спец. 09.02.03 Программирование в компьютерных системах очной формы обучения / сост. О.С. Бурякова. – Шахты : ИСОиП (филиал) ДГТУ в г. Шахты, 2019. – 37 с.

Технология разработки программного обеспечения содержат общие положения, требования к выполнению практических заданий и рекомендованную литературу. Методические указания позволят обучающимся освоить дисциплину: «Технология разработки программного обеспечения». Предназначено для обучающихся специальности 09.02.03 Программирование в компьютерных системах очной формы обучения.

Режим доступа к электронной копии печатного издания: http://www.libdb.sssu.ru

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Информационные технологии проникли практически во все сферы жизни современного общества. Трудно найти область деятельности, где не было бы потребности в обработке информации. Наступившая цифровая революция привела к генерации огромных объемов данных, обработка которых превышает когнитивные возможности человека. В связи с этим все большую роль будут занимать люди, чья профессия связна с разработкой и кодированием алгоритмов, позволяющих работать с информацией. Важным направлениям в подготовке таких специалистов является дисциплина «Технологии разработки программного обеспечения», в которой раскрываются основные подходы к созданию программного обеспечения.

Учебное пособие состоит из нескольких разделов, в которых представлен обзор основных этапов развития технологий и методов программирования, начиная с неструктурированного подхода и завершая объектно-ориентированным и функциональным. Дан обзор основных стратегий конструирования жизненного цикла программного обеспечения, критерии для оценки качества программ, перечислены факторы, по которым определяют надежность программных средств.

Кроме того, на стадии проверки работы созданного программного продукта важно определить эффективность программы, определить оптимизационные характеристики. Обязательным этапом работы программиста является сопровождение программы, отладка её работы в месте её непосредственного использования. В методическом пособии представлен обзор основных вариантов ошибок, которые могут встретиться в программе.

Учебное пособие предназначено для направления подготовки обучающихся специальности 09.02.03 «Программирование в компьютерных системах» среднего профессионального образования.

1 Структура жизненного цикла программы

Комплексы программ создаются, эксплуатируются и развиваются во времени. Жизненный цикл программных средств включает в себя все этапы развития: от возникновения потребности в программе, определением целевого назначения, до полного прекращения использования этого программного средства, вследствие его морального старения или потере необходимости решения соответствующих задач.

По длительности жизненного цикла, программные средства можно разделить на два класса: с малым и большим временем жизни. Этим классам программ соответствует гибкий (мягкий) подход к их созданию и использованию и жесткий промышленный подход регламентированного проектирования и эксплуатации промышленных изделий.

В научных организациях и образовательных учреждениях преобладают разработки программ 1 класса. А в проектных и промышленных организациях – второго.

Программы с малой длительностью эксплуатации создаются в основном для решения научных и инженерных задач, для получения конкретных результатов вычислений. Такие программы относительно не велики от 1 до 10000 команд. Разрабатываются, как правило, одним специалистом или маленькой группой, не предназначены для тиражирования и передачи для последующего использования в другие коллективы. ЖЦ таких программ состоит из длительного интервала системного анализа и формализации проблемы, значительного этапа проектирования программ и относительно небольшого времени эксплуатации и получения результатов.

Требования, предъявляемые к функциональным и конструктивным характеристикам, как правило, не формализуются, отсутствуют оформленные испытания программ, и показатели их качества контролируются только разработчиками, в соответствии с неформальными представлениями.

Сопровождение и модификация таких программ не нужны и их жизненный цикл завершается после получения результатов вычислений. Основные затраты в жизненном цикле таких программ приходятся на этапы системного анализа и проектирования, которые продолжаются от месяца до одного или двух лет. В результате жизненного цикла редко превышает три года, т.е. основное время идет на анализ и проектирование.

Программы с большой длительностью эксплуатации создаются для регулярной обработки информации и управления в процессе функционирования сложных вычислительных систем. Размеры программных средств могут изменяться в широких пределах (от 1 до бесконечности команд), однако все они обладают свойством познаваемости и возможности модификации в процессе использования различными специалистами. Программы такого класса допускают тиражирование. Они сопровождаются документацией как промышленные изделия и представляют собой отчуждаемый программный продукт.

Проектированием и эксплуатацией программного средства могут заниматься большие коллективы специалистов, для чего необходима формализация требуемых технических характеристик комплекса программ и их компонент. А также формализированные испытания и определение достигнутых показателей качества программных средств.

Жизненный цикл таких программных средств составляет 10-20 лет, из которых 70-90% приходиться на эксплуатацию и сопровождение. Вследствие массового тиражирования и длительного сопровождения совокупные затраты в процессе эксплуатации и сопровождения могут значительно превышать затраты на системный анализ и проектирование.

Жизненный цикл рассматриваемых программ включает в себя следующие основные этапы:

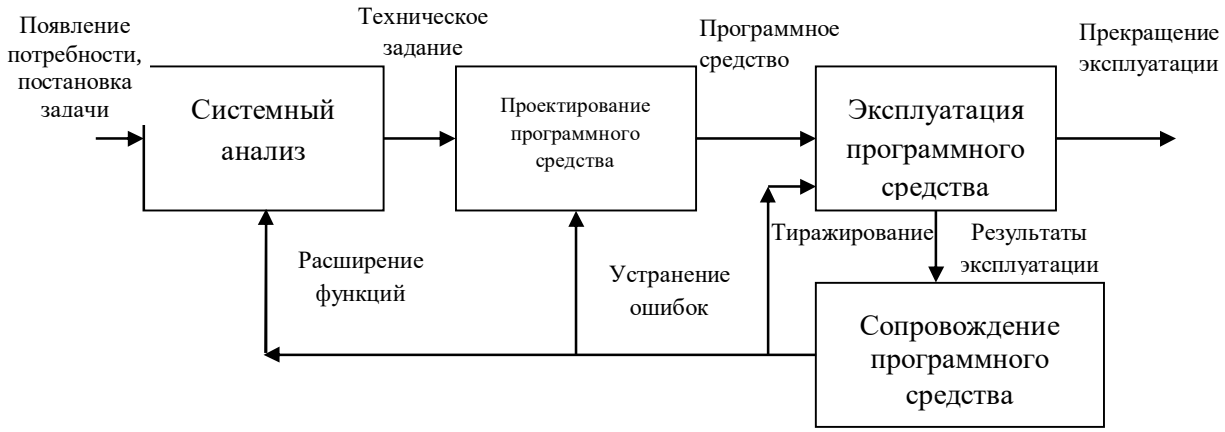

Рисунок 1- Этапы жизненного цикла программных средств

Рассмотрим основные элементы рисунка 1.

Системный анализ в ходе, которого определяются потребности в программном средстве, его назначение, основные функциональные характеристики, оцениваются затраты и возможная эффективность применения такого комплекса программ.

Проектирование программного средства включает в себя разработку структуры комплекса и его компонент, программирование модулей и ряд этапов отладки, а также испытание и внедрение для регулярной эксплуатации, созданной версии комплекса программ.

Эксплуатация программного средства заключается в исполнении, функционировании программы на компьютере, для обработки информации, получения результатов, являющихся целью создания программных средств, а также в обеспечении достоверности и надежности выдаваемых данных.

Сопровождение программного средства состоит в эксплуатационном обслуживании, развитии функциональных возможностей и повышении эксплуатационных характеристик программного средства, тиражировании, переносе комплекса программ на различные типы вычислительных средств.

Среди перечисленных этапов, наиболее специфическим, трудно формализуемым и тесно связанным с функциональным назначением программного средства является этап системного анализа. На этом этапе формируется назначение и основные показатели качества создаваемых программ. Решаемые задачи практически полностью определяются предметной областью системного анализа, поэтому на данном этапе трудно обобщать технологические процессы и критерии качества при создании различных типов программ.

Этапы проектирования и эксплуатации, и сопровождения различаются целями, задачами, методами и средствами.

Следует обратить внимание на особенность взаимодействия этапа сопровождения с этапами эксплуатации и проектирования, в которой сопровождение играет роль необходимой обратной связи от этапа эксплуатации.

Как и любая схема действий, классический жизненный цикл имеет достоинства и недостатки.

Достоинства классического жизненного цикла: дает план и временной график по всем этапам проекта, упорядочивает ход конструирования.

К недостаткам классического жизненного цикла относятся:

- реальные проекты часто требуют отклонения от стандартной последовательности шагов;
- цикл основан на точной формулировке исходных требований к программному средству (реально в начале проекта требования заказчика определены частично);
- результаты проекта доступны заказчику только в конце работы.

Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в приспосабливаемости продукта под операционную систему, форме диалога с пользователем или в эффективности реализуемого алгоритма. В этих случаях целесообразно использовать макетирование.

Основная цель макетирования – снять неопределенности в требованиях заказчика.

Макетирование (прототипирование) – это процесс создания модели требуемого программного продукта.

Модель может принимать одну из трех форм:

1) бумажный макет или макет на основе компьютера (изображает или рисует человеко-машинный диалог);

2) работающий макет (выполняет некоторую часть требуемых функций);

3) существующая программа (характеристики которой затем должны быть улучшены).

Последовательность действий при макетировании:

Макетирование начинается со сбора и уточнения требований к создаваемому программному обеспечению. Разработчик и заказчик встречаются

и определяют все цели программного обеспечения, устанавливают, какие требования известны, а какие предстоит доопределить.

Затем выполняется быстрое проектирование. В нем внимание сосредоточивается на тех характеристиках программного обеспечения, которые должны быть видимы пользователю.

Быстрое проектирование приводит к построению макета.

Макет оценивается заказчиком и используется для уточнения требований к программному обеспечению.

Итерации повторяются до тех пор, пока макет не выявит все требования заказчика и, тем самым, не даст возможность разработчику понять, что должно быть сделано.

Достоинством макетирования является обеспечение определения полных требований к программному обеспечению. Среди недостатков макетирования следует выделить: заказчик может принять макет за продукт; разработчик может принять макет за продукт.

Поясним суть недостатков. Когда заказчик видит работающую версию программы, он перестает сознавать, что детали макета скреплены «жевательной резинкой и проволокой»; он забывает, что в погоне за работающим вариантом оставлены нерешенными вопросы качества и удобства сопровождения программы. Когда заказчику говорят, что продукт должен быть перестроен, он начинает возмущаться и требовать, чтобы макет «в три приема» был превращен в рабочий продукт. Очень часто это отрицательно сказывается на управлении разработкой программы.

С другой стороны, для быстрого получения работающего макета разработчик часто идет на определенные компромиссы. Могут использоваться не самые подходящие язык программирования и операционная система. Для простой демонстрации возможностей может применяться неэффективный алгоритм. Спустя время разработчик забывает о причинах, по которым эти средства не подходят. В результате далеко не идеальный выбранный вариант интегрируется в систему.

2 Стратегии конструирования программного обеспечения

Существуют 3 стратегии конструирования программы:

- однократный проход (водопадная стратегия) – линейная последовательность этапов конструирования;
- инкрементная стратегия. В начале процесса определяются все пользовательские и системные требования, оставшаяся часть конструирования выполняется в виде последовательности версий. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т. д., пока не будет получена полная система;

- эволюционная стратегия. Система также строится в виде последовательности версий, но в начале процесса определены не все требования. Требования уточняются в результате разработки версий.

Инкрементная модель является классическим примером инкрементной стратегии конструирования. Она объединяет элементы последовательной водопадной модели с итерационным макетированием. Каждая линейная последовательность вырабатывает поставляемый инкремент программного обеспечения. Первый инкремент приводит к получению базового продукта, реализующего базовые требования (многие вспомогательные требования остаются нереализованными). План следующего инкремента предусматривает модификацию базового продукта, обеспечивающую дополнительные характеристики и функциональность. По своей природе инкрементный процесс итеративен, но в отличие от макетирования, инкрементная модель обеспечивает на каждом инкременте работающий продукт в соответствии с рисунком 2.

Рисунок 2 – Инкрементная модель конструирования программы

Модель быстрой разработки приложений (Rapid Application Development), представлена на рисунке 3. Она является вторым примером применения инкрементной стратегии конструирования.

Рисунок 3 – Модель быстрой разработки приложений

RAD-модель обеспечивает экстремально короткий цикл разработки. RAD-высокоскоростная адаптация линейной последовательной модели, в которой быстрая разработка достигается за счет использования компонентно-ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, Rad-процесс позволяет группе создать полностью функциональную систему за очень короткое время (60-90 дней). RAD-подход ориентирован на разработку информационных систем и выделяет следующие этапы:

- бизнес-моделирование. Моделируется информационный поток между бизнес-функциями. Ищется ответ на следующие вопросы: Какая информация руководит бизнес-процессом? Какая генерируется информация? Кто генерирует ее? Где информация применяется? Кто ее обрабатывает?

- моделирование данных. Информационный поток, определенный на этапе бизнес-моделирования, отображается в набор объектов данных, которые требуются для поддержки бизнеса. Идентифицируются характеристики (свойства, атрибуты) каждого объекта, определяются отношения между объектами.

- моделирование обработки. Определяются преобразования объектов данных, обеспечивающие реализацию бизнес-функций. Создаются описания обработки для добавления, модификации, удаления или нахождений (исправлений) объектом данных;

- генерация приложения. Предполагает использование методов, ориентированных на языки программирования 4-го поколения. Вместо создания программного обеспечения с помощью языков программирования 3-го поколения, RAD-процесс работает с повторно используемыми программными

компонентами или создает повторно используемые компоненты. Для обеспечения конструирования используются утилиты автоматизации.

- тестирование и объединение. Поскольку применяются повторно используемые компоненты, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть протестированы).

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.

Применения RAD имеет свои недостатки и ограничения.

1. для больших проектов в RAD требуются существенные людские ресурсы (необходимо создать достаточное количество групп);
2. RAD применима только для таких приложений, которые могут декомпозироваться на отдельные модули и в которых производительность не является критической величиной;
3. RAD не применима в условиях высоких технических рисков (т.е. при использовании новой технологии).

Спиральная модель – классический пример применения эволюционной стратегии конструирования.

Спиральная модель базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент – анализ риска, отсутствующий в этих парадигмах в соответствии с рисунком 4.

Рисунок 4 – Спиральная модель жизненного цикла программного обеспечения

Спиральная модель: 1– начальный сбор требований и планирование проекта, 2 – та же работа, на основе рекомендаций заказчика, 3 – анализ риска на основе начальных требований, 4 – анализ риска на основе реакции заказчика, 5 – переход к комплексной системе, 6 – начальный макет системы, 7 – следующий уровень макета, 8 – сконструированная система, 9 –оценивание заказчиком.

Как показано на рисунке, модель определяет четыре действия, представляемые четырьмя квадрантами спирали:

1. планирование – определение целей, вариантов и ограничений.
2. анализ риска – анализ вариантов и распознавание/выбор риска
3. конструирование – разработка продукта следующего уровня
4. оценивание – оценка заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели очевиден при учете измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО.

В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте конструирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации. Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая разработчиков к более общей модели системы. В каждом цикле по спирали требуется конструирование (нижний правый квадрант), которое может быть реализовано классическим жизненным циклом или макетированием. Заметим, что количество действий по разработке (происходящих в правом нижнем квадранте) возрастает по мере продвижения от центра спирали.

Достоинства спиральной модели:

1. наиболее реально (в виде эволюции) отображает разработку программы;
2. позволяет явно учитывать риск на каждом витке эволюции разработки;
3. включает шаг системного подхода в итерационную структуру разработки;
4. использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

1. новизна (отсутствует достаточная статистика эффективности модели);
2. повышенные требования к заказчику;
3. трудности контроля и управления временем разработки.

3 Критерии оценки качества программ

Критерии качества представляют собой измеряемые числовые показатели, в виде некоторой целевой функции, характеризующие степень выполнения объектом своего назначения. В общем случае критерии качества должны отображать обобщенную «полезность» для общества, анализируемого объекта,

эффективность технологии проектирования. Программное средство в первую очередь характеризуется трудоемкостью и длительностью создания, а также достигаемым качеством программ при применении соответствующих технологий.

Анализ критерий качества программного средства жизненного цикла программы является основой для оценки эффективности технологии проектирования. В процессе формирования технического задания на программное средство, выявляются доминирующие показатели, устанавливается относительная важность каждого из них и строится обобщенная функция требуемого качества конкретного программного средства, кроме того устанавливаются допустимые затраты и длительность разработки программных средств, которые должна обеспечить соответствующая технология.

По мере создания комплекса программ, после завершения отладки и испытаний, уточняется достигнутое реальное значение каждого из показателей и обобщенные функции качества всего комплекса. Показатели качества могут уточняться также в процессе эксплуатации.

Разработчики программных средств стремятся выделить и определить единый критерий эффективности программ:

1. *Численно* и в наиболее общем виде *характеризовать* степень выполнения системой своей основной целевой функции.
2. *Позволять* выявить и оценить степень влияния на эффективность системы различных факторов и параметров, и в том числе затрат различного вида на ее реализацию.
3. *Быть* простым и иметь малую дисперсию, т.е. мало зависеть от неконтролируемых, случайных факторов.

Возможность реализации системы, удовлетворяющим некоторым критериям качества, естественно зависит от обеспечения ресурсами и техническими средствами, выполняющими основные функции. Высокая стоимость сложных систем, длительные сроки их проектирования и изготовления особенно остро ставят задачу оценки затрат, при которых та или иная эффективность достигается. Особенно сложно в комплексах программ, содержащих сотни модулей, обеспечить наилучшее использование ресурсов комплекса, с точки зрения основного критерия эффективности, при сохранении ряда частных показателей качества в допустимых пределах.

Многочисленность и сложность путей использования программ требует их высокой устойчивости, как по отношению к ошибкам во входной информации, так и по отношению к внутренним сбоям компьютера, выполняющего программу. Для обеспечения такой устойчивости сложные программы обычно содержат контрольные операции различного типа и имеют специальные модули адаптации и самоорганизации для изменения структуры программ. А в ряде случаев и всей системы управления при перезагрузках, сбоях и частичных отказах.

Команды и данные входящие в программные модули не имеют абсолютной надежности правильного исполнения. Поэтому приходиться применять специальные аппаратные и программные повышения надежности выполнения программ для получения правильных результатов и управляющих воздействий.

Показатели качества могут уточняться также в процессе эксплуатации, в результате чего обеспечивается долгосрочная перспектива объективного измерения и повышения качества программ.

1. *Функциональные критерии* качества отражают основную специфику применения и степень соответствия программного средства их целевому назначению. Для программ управления в них входят показатели точности выходных данных, диапазоны изменения параметров, время реакции, адаптивность к внешним воздействиям и т.д. В системах обработки данных функциональные показатели отражают номенклатуру исходных данных, достоверность результатов, разнообразие функций и другое. Функциональные критерии весьма различны и соответствуют разнообразию целевого назначения, функций и областей применения программного средства. Они являются важнейшими для каждой системы (перечень исходных документов - конкретно).
2. *Конструктивные критерии* качества программного средства более или менее инвариантны к их целевому назначению и основным функциям. К ним относятся сложность программ, надежность функционирования, используемые ресурсы, корректность программ и т.п. Некоторые конструктивные критерии могут быть важны с позиции прямого функционального назначения, определяемых программными средствами (надежность).

Разделение критериев на две группы условно и преследует цель выделения критериев наиболее общих для всех типов программных средств и в наименьшей степени зависящих от их основного функционального назначения.

Особо следует выделить временные показатели жизненного цкла программ:

1. длительность проектирования;
2. продолжительность эксплуатации очередной версии;
3. длительность проведения каждой модификации.

Продолжительность и модификация проведения этих работ в ряде случаев может быть более важным критерием, чем трудоемкость. В некоторых случаях суммарная длительность эффективной эксплуатации является *доминирующим критерием* качества ПС. Для каждой из разработок целесообразно проводить ранжирование (ранги) критериев и факторов на этапах ЖЦ ПС.

Программное обеспечение (software) на данный момент составляет сотни тысяч программ, которые предназначены для обработки самой разнообразной информация с самыми различными целями. В зависимости от того, какие задачи выполняет то или иное программное обеспечение можно разделять все программное обеспечение на несколько групп:

1. Системное программное обеспечение (или системные программы) – предназначено для эксплуатации и технического обслуживания компьютера, управления и организации вычислительного процесса при решении любой конкретной задачи на компьютере и т.д. Системное программное обеспечение обязательная часть программного обеспечения, к нему относятся:

- операционные системы;
- оболочки операционных систем;
- программы-утилиты.

2. Прикладное программное обеспечение (или пакеты прикладных программ) – предназначено для решения определенного класса задач, т.е. это программы, используемые как инструмент при создании документов в повседневной деятельности. Или программы, с помощью которых пользователь решает свои информационные задачи, не прибегая к программированию. К ним относятся:

- текстовые и графические редакторы;
- электронные таблицы;
- системы управления базами данных (СУБД);
- интегрированные среды;
- обучающие и учебные программы;
- прикладные программы;
- игры;
- мультимедиа и т.д.

3. Инструментальное программное обеспечение (или системы программирования) обеспечивают создание всех классов программ: системных, прикладных и новых систем программирования. Или инструментальное программное обеспечение (или системы программирования) обеспечивает создание новых прикладных программ для компьютера. Это комплексы программ и прочих средств, предназначенных каждый для разработки и эксплуатации программ на конкретном языке программирования для конкретного вида ЭВМ. Системы программирования обычно включают некоторую версию языка программирования, транслятор программ и т.д.

4 Надежность программных продуктов. Факторы надежности

Надежность программных продуктов – это уровень, при котором система программ удовлетворяет поставленным требованиям и пригодна для эксплуатации. Сама надежность является составной частью более общего понятия качества. Качественная программа не только надежна, но и компактна, совместима с другими программами, эффективна, удобна в сопровождении и вполне понятна. Надежность программного обеспечения закладывается на всех фазах жизни программы, не только при поиске ошибок.

Свойства надежности программного средства изучаются теорией надежности, которая является системой определенных идей, математических

моделей и методов, направленных на решение проблем оценки, предсказания, и оптимизации различных показателей надежности. В основе теории надежности лежат понятия о двух возможных состояниях системы:

1. работоспособное – такое состояние системы, при которой она способна выполнять заданные функции, с параметрами, установленными технической документацией;
2. неработоспособное.

В процессе функционирования системы возможны переходы от одного состояния к другому. С этими переходами связано событие отказа и восстановления. Существует три вида отказа:

1. устойчивый;
2. самоустраняющийся;
3. перемежающийся.

Для устранения устойчивого отказа требуется проведение специальных мероприятий по восстановлению работоспособности. Самоустраняющийся отказ или сбой характеризуется достаточно быстрым восстановлением работоспособности без внешнего вмешательства. Перемежающийся отказ представляет собой многократно повторяющиеся сбои, для полного устранения которых требуется внешнее вмешательство.

По возможности восстановления работоспособности в процессе эксплуатации объекты делятся на восстанавливаемые и невосстанавливаемые. Невосстанавливаемые объекты не допускают ремонта или замены отказавшей компоненты и не обладают самовосстанавливаемостью во время выполнения своих функций. Такие объекты могут эксплуатироваться либо до 1 отказа. либо до полного выполнения своих функций, либо до момента достижения некоторого предельного состояния, после чего они снимаются с эксплуатации. Длительность возможного последующего ремонта при этом не учитывается.

Восстанавливаемые объекты допускают ремонт и замену отказавших компонент и обладают возможностью самовосстановления. При этом существенную роль играет длительность пребывания в неработоспособном состоянии и проведения восстановительных работ. Рентабельность восстановления и допустимая длительность восстановительных работ зависит от конкретного значения и условия эксплуатации объекта. Для определения факта и степени работоспособности системы или объекта, необходимы средства способные установить соответствие параметров и характеристик данной системы требованиям технической документации. Кроме контроля работоспособности применяется диагностический контроль, основное назначение которого состоит в локализации отказа и установлении его характера и личности. Диагностический контроль может способствовать установлению степени работоспособности системы, однако его основная цель заключается в обеспечении скорейшего восстановления.

Также для оценки качества программного средства пользователем, разработчики должны рассмотреть следующие характеристики:

1) Пригодность для применения:
   - точность – мера, характеризующая приемлемость величины погрешности в выдаваемых программами результатах с точки зрения предполагаемого их использования
   - защищенность - свойство, характеризующее способность ПС противостоять преднамеренным или нечаянным деструктивным (разрушающим) действиям пользователя;
   - способность к взаимодействию;
   - соответствия стандартам и правилам проектирования.
2) Надежность:
   - отсутствие ошибок;
   - устойчивость к ошибкам;
   - перезапускаемость.
3) Применяемость:
   - понятность – свойство, характеризующее степень в которой программное средство позволяет изучающему его лицу понять его назначение, сделанные допущения и ограничения, входные данные и результаты работы его программ, тексты этих программ и состояние их реализации;
   - обучаемость;
   - простота использования.
4) Эффективность:
   - ресурсная экономичность – мера, характеризующая способность программного средсва выполнять возложенные на него функции при определенных ограничениях на ресурсы (память);
   - временная экономичность – мера, характеризующая способность программного средства выполнять возложенные на него функции за определенный отрезок времени.
5) Сопровождаемость:
   - удобство для анализа;
   - изменяемость;
   - стабильность;
   - тестируемость.
6) Переносимость:
   - адаптируемость;
   - структурируемость – свойство, характеризующее программы с точки зрения организации взаимосвязанных их частей в единое целое определенным образом (например, в соответствии с принципами структурного программирования);
   - замещаемость;
   - внедряемость.

При любом виде деятельности людям свойственно непредумышленно ошибаться, в результате чего появляются в процессе создания или применения изделий или систем. В общем случае под ошибками подразумевается дефект,

погрешность или неумышленное искажение объекта или процесса. При строго фиксированных данных программы исполняются по определенным маршрутам и выдают совершенно определенные результаты. Многочисленные варианты использования программ при разнообразных исходных данных представляются наблюдателю, как случайные, поэтому возникновение ошибок считается случайным и имеет разную природу и последствия.

Объектами уязвимости влияющими на надежность программного средства являются:

- динамический вычислительный процесс обработки данных, автоматизирующий подготовку решений и выработку управляющих воздействий;
- информация, накопленная в базе данных, отражающая объекты внешней среды в процессе ее обработки;
- объектный код программы исполняемый вычислительными средствами в процессе функционирования программного средства;
- информация, подаваемая пользователю на исполнительные механизмы, являющаяся результатом обработки исходных данных и информации накопленной в базе данных.

На эти объекты воздействуют различные дестабилизирующие факторы:

1. Внутренние:
   - системные ошибки при постановке целей и задач, создания ПС, при формулировке требований к функциям и характеристикам решения задач, определения условий и параметров внешней среды.
   - алгоритмические ошибки разработки при непосредственной спецификации функций программных средств, при определении структуры и взаимодействии компонент комплекса программ, а также при использовании базы данных.
   - ошибки программирования в текстах программ и описание данных, а также в исходной и результирующей документации на компоненты и программные средства в целом.
   - недостаточную эффективность в используемых методах и средствах оперативной защиты программ и данных от сбоев и отказов, и обеспечении надежности функционирования программных средств в условиях случайных негативных воздействий.
2. Внешние:
   - ошибки оперативного и обслуживающего персонала в процессе эксплуатации программного средства;
   - искажение в каналах телекоммуникации, информации, поступающей от внешних источников и передаваемой потребителям, а также недопустимые для конкретной информационной системы характеристики потоков внешней информации;
   - сбои и отказы в аппаратуре вычислительных средств;

- изменение состава и конфигурации комплекса, взаимодействующей аппаратуры информационной системы, за пределы проведенные при испытании или сертификации и отлаженной работы программы в эксплуатационной документации.

5 Технологические методы и средства разработки качественного программного обеспечения

5.1 Приемы надежного программирования

Надежность программного обеспечения основывается на двух его основных свойствах: это правильность и устойчивость (предполагает, что программное обеспечение не создает аварийных ситуаций определенного вида при отказах оборудования системы). Программа любой сложности и назначения при строго фиксированных исходных данных и абсолютно надежной аппаратуры исполняется по однозначно определенному маршруту и дает на выходе строго определенные результаты, при изменении данных – получается много других маршрутов. Такое количество вариантов исполнения программных продуктов невозможно проверить при отладке и тестировании, что является уже само собой ненадежностью.

Понятие правильной и корректной программы можно рассматривать во временном функционировании программного средства. Восстановление функционирования компьютера (при сбоях работы) позволяют улучшить показатели надежности, для решения этой проблемы необходимо выполнять следующие работы:

- систематический контроль и обнаружение аномалий процесса функционирования или состояния программы и данных;
- диагностировать обнаруженные искажения;
- выбирать методы и средства оперативного восстановления;
- реализовывать оперативное восстановление работоспособности;
- регистрировать каждый произошедший сбой.

Реализация осуществляется за счет введения избыточности в программе, т.е. помимо выполнения основного процесса программные средства должны выполнять программно все вышеуказанные пункты. Существуют такие факторы снижающие надежность, как вызывающие сбои или отказ при исполнении программы (искаженная информация); самоустраняющиеся отказы или сбои в аппаратуре компьютера; не выявленные ошибки в программных средствах, шумы и сбои в каналах связи при передачи сообщений по линиям связи; потери или искажения сообщений в ограниченных буферах компьютера; ошибки в документах, используемых для подготовки данных вводимых в компьютере.

Обеспечение надежности путем введения избыточности.

В процессе проектирования недостаточно формировать правильные программы, выдающие верные результаты, поэтому в компьютере закладывается избыточность для обеспечения надежности функционирования.

Виды избыточности:

- временная – использование некоторой части производительности компьютера для контроля исполнения программы и восстановления вычислительного процесса.
- информационная – дублирование данных, обрабатываемых компьютером.
- программная – контроль и обеспечение достоверности наиболее важных результатов обработки информации (контрольная сумма).

6 Стили программирования

Этап проектирования программ оказывает влияние на стиль программирования, надежность, эффективность, отладку, тестирование и эксплуатационное свойство программ. Таким образом, это важнейшая часть любой программной разработки.

Работая над программой, программист, особенно начинающий, должен хорошо представлять, что программа, которую он разрабатывает, предназначена, с одной стороны, для пользователя, с другой — для самого программиста. Текст программы нужен, прежде всего, самому программисту, а также другим людям, с которыми он совместно работает над проектом. Поэтому для того, чтобы работа была эффективной, программа должна быть легко читаемой, ее структура должна соответствовать структуре и алгоритму решаемой задачи. Как этого добиться? Надо следовать правилам хорошего стиля программирования. Стиль программирования – это набор правил, которым следует программист (осознано или потому, что «так делают другие») в процессе своей работы. Очевидно, что хороший программист должен следовать правилам хорошего стиля:

1. *Стремление к простоте*. Простота проектирования программ – это первый шаг, ведущий к получению легко читаемой программы. Необычное кодирование программы (например, использование скрытых возможностей машины) часто препятствует отладке программы и затрудняет ее использование другими программистами. Структура программы должна раскрывать ее логику, например, если тестовая программа имеет несколько ветвей, было бы удобно расположить части программы соответствующие каждой ветви в определенном порядке, например, по возрастанию значения параметра теста. Этот метод наиболее целесообразен поскольку именно такого решения и ожидает тот, кто читает программу.
2. *Чтение программы*. Программу надо снабжать комментариями и параграфами.
3. *Описание задач*. Идеальная программа – это такая программа, которая дает точный желаемый выход при неясно сформированных требованиях пользователя. Один из путей уточнения описания требования заказчика, это

проведение регулярных просмотров для координации установленных требований и действительных намерений заказчика.

4. *Постановка задачи*. Необходимо четко определить задачу, ее цели, этапы и конечный результат. Далее программист должен переписать спецификации задачи ориентируясь на ЭВМ. Необходимо сжатое, но полное описание программы. Затем программист и заказчик должны тщательно изучить написанные спецификации задачи, чтобы быть уверенным в ее правильном понимании.

5. *Выбор алгоритма*. Важнейшим шагом для получения эффективной и правильной программы является составление алгоритма. При этом предполагается правильный выбор языка и спецификации программы. Таким образом, хороший алгоритм необходимое, но недостаточное условие хорошей программы. Если пользователь формирует задачу в виде четкого алгоритма, то процесс проектирования существенно облегчается. Для эффективности необходимо рассмотреть несколько алгоритмов и из них выбрать наиболее эффективный.

6. *Описание данных.* Другим фактором сравнимым по значению с выбором алгоритма является описание данных. Хорошо продуманное описание данных существенно сокращает программу. Так, полезно, использовать массивы данных в том случае, когда это наиболее очевидный способ их организации. Другой пример такого рода – возможность использовать ссылки и указатели. Например, если нужно проследить отношения между родителями и их потомками на протяжении нескольких поколений, то легче всего это сделать с помощью ссылок и указателей.

7. *Выбор языка программирования*. Часто выбор языка программирования предоставлен данной выигрышной системе, или подготовкой программиста. Существуют серьезные основания для установления языковых стандартов для системы. Если применяют много разных языков для написания программ, то использование последних становится затруднительным.

8. *Универсальность.* Хорошая универсальная программа должна обрабатывать исходные данные (например, число элементов равно 0 ил 1) и печатать сообщения об ошибке. Тогда программа является не только универсальной, но и защищенной от ошибок. Используйте в качестве компиляторов переменные, а не константы. Если же применять константы, то при изменении параметров надо изменять в исходной программе каждый оператор, содержащий прежнюю константу. Укажем некоторые очевидные случаи, когда переменные могут быть использованы вместо констант: 1. размер таблиц, массивов, списков; 2. налоги, скидки, физические константы, проценты; 3. обозначения устройств ввода-вывода. Можно поступить и по-другому, например, хранить все подлежащие изменению параметры в одном массиве. Эти параметры могут быть инициализированы в одном месте программы и достаточно полно прокомментированы.

9. *Библиотеки*. Чтобы повысить эффективность разработки программ, облегчить отладку и тестирование, а, следовательно, и сократить работу по созданию программ используйте библиотеки.

- тип библиотечных подпрограмм представляет собой функции и подпрограммы, имеющиеся в наличии для данного языка программирования. К этой категории относятся все стандартные функции. Для ознакомления с функциями и подпрограммами языка загляните в руководство по соответствующему языку. Встроенные функции, которыми снабжает язык, конечно хорошо и тщательно закодированы и тестированы;
- источник функций и подпрограмм находиться в пределах вашей вычислительной системы, т.е. можно использовать те функции, процедуры или подпрограммы, которые ранее были разработаны;
- источник библиотечных программ представляют книги по программированию. Многие программы содержат программы, которые можно использовать как библиотечные, но должны быть предприняты некоторые усилия, чтобы получить интересующую информацию.

10. *Форматы ввода/вывода*. Форматы входных и выходных данных являются частью этапа проектирования, входные данные должны быть разработаны с учетом максимального удобства для пользователя и минимальные ошибки. Постоянство входных форматов, как правило, также способствуют уменьшению ошибок. Выходные спецификации могут сильно различаться. Иногда даются четкие инструкции, и выходные данные подгоняются под определенный стандарт. Однако часто отсутствуют какие-либо указания, и выходные данные подчас представляют собой страницы, заполненные числами без всякой идентификации. Выходная информация должна идентифицироваться без привлечения других источников. Выходные данные должны содержать: 1. идентификацию выходной информации, 2. описание записи, 3. дату, 4. нумерацию страниц. Кроме того, каждая напечатанная группа элементов должна быть помечена. При табличной форме выдачи должны быть помечены строки и столбцы.

6.1 Структурное программирование

Один из методов улучшения программы является структурное программирование (СП). Оно предназначено для организации проектирования программ и процесса кодирования. Таким образом, чтобы предотвратить большинство логических ошибок и обнаружить те, которые допущены.

Перечислим некоторые достоинства структурного программирования:

1. Структурное программирование позволяет значительно сократить число вариантов построения программы по одной и той же спецификации, что снижает сложность программы.
2. В структурированных программах логически связанные операторы находятся визуально ближе, а слабо связанные дальше, что позволяет обходиться без блок схем и других графических форм изображения алгоритмов.

3. Сильно упрощается процесс тестирования и отладки структурированных программ.
4. Высокую производительность работы за счет того, что действие каждой управляющей структуры хорошо известно и нет необходимости его обдумывать.
5. Ясность и читаемость программ.
6. Высокую эффективность за счет глобальной оптимизации программы.

Структурное программирование сосредотачивается на одном из наиболее подверженных ошибками факторов программирования – логики программы и включает три главные составляющие:

1. *проектирование сверху – вниз* (нисходящее проектирование) – подобно написанию статьи сверху – вниз. Процесс написания статьи имеет иерархическую структуру и начинается с вершины иерархии, т.е. с краткого обзора. Разработку проекта обычно начинают с исследования целей и определения основных задач, ведущих к достижению этих целей. Если проект очень большой, то необходимо провести его разбиение. Вначале необходимо написать то, что хотите сделать на естественном языке. Этот шаг часто многое раскрывает. Нередко можно обнаружить, что не в состоянии записать задачу на естественном языке. В таком случае не надейтесь, что удастся составить программу. Следовательно, важно формулировать задачу правильно на стадиях проектирования, чтобы не исправлять ее позднее на стадиях программирования и отладки. Метод проектирования сверху –вниз предусматривает вначале определения задачи, а затем постепенное уточнение структуры путем внесения более мелких деталей. Проектирование представляет собой последовательность шагов такого уточнения. На каждом шаге необходимо выявить функции, которые нужно воплотить, т.е. данная задача разбивается на ряд, каждому из них будет соответствовать один модуль. Именно такой традиционный и по существу иерархический подход применяется при создании сложных структур в других областях (в технике, в серийном производстве). Далее следует описать данные, указывая их структуру и основные процессы обработки. Это описание должно включать тщательно отработанные примеры, убедительно демонстрирующие функции системы и их наиболее существенные варианты – такие примеры будут полезны позже на стадии тестирования. При описании модуля должны быть описаны его тестовые данные. Тестирование программы неизбежно, поэтому выявление требований к тестированию, заранее на стадиях проектирования, являются хорошей практикой. Логическая проверка фрагментов программы должна уменьшить необходимость тестирования конечной программы. Основное преимущество такого метода работы то, что он обеспечивает создание документации.

2. *модульное программирование*. Чтобы преуспеть в структурном программировании программу следует представить в виде модулей. Модульное программирование – это процесс разделения программы на логические части, называемые модулями, и последовательное программирование каждой части.

Когда большая единая задача делиться на подзадачи, то значительно проще прочесть и понять программу. Если проведено программирование всей задачи сверху – вниз, то она естественно разбивается на подзадачи для возможных модулей. При этом преследуется две цели:

- необходимость добиться того, чтобы программный модуль был правильным и не зависел от контекста, в котором он будет использоваться;
- следует стремиться к тому, чтобы из модулей можно было формировать большие программы без каких-либо предварительных значений о внутренней работе модуля.

Оптимальный размер модуля 60 строк. Следует стремиться к независимости между модулями или программами. Для достижения этого требуется, чтобы модуль не зависел от: 1) источника входных данных; 2) места назначения выходных данных; 3) от предистории. При разбиении программы на функциональные блоки, независимые модули имеют некоторую самостоятельность. Хотя определенная зависимость все-таки существует.

Каждый модуль должен иметь свое назначение, отличающееся от назначения других модулей. Это должен быть замкнутый блок, вход и выход которого четко определены. Стремление к независимости хорошо тем, что менее вероятно, что изменения в одной подпрограмме влияет на оставшуюся часть программы. Воздействие изменения в одном модуле на другую часть программы называется *волновым эффектом*. Этот эффект можно уменьшить сведя к минимуму связь между модулями, т.е. сократить количество путей вдоль которых изменения или ошибки могут проникнуть в другие части. Простой путь уменьшения волнового эффекта – избегать использование глобальных перемен и делать модуль небольшим. Минимизация взаимосвязи между модулями – это модульное сцепление, которое происходит за счет усиления связей между элементами одного модуля (модульная прочность).

Таким образом, тесно связанные элементы надо стремиться поместить в один модуль. Использование модулей приводит к уменьшению сложностей, факторы сложности при этом включают три составляющие: 1)функциональная сложность – обусловлено тем, что один модуль выполняет слишком много функций; 2) распределенная сложность – это сложность идентификации общей функции распределенной между несколькими модулями за счет чего утрачивается возможность уменьшения сложности всей программы при модульном программировании; 3)сложность связи – определяется сложностью взаимодействия модулей, при использовании общих данных.

3. структурное кодирование – это метод написания хорошо структурированных программ, который позволяет получать программы более удобные для тестирования, модификации и использования. Программы произвольных размеров и сложности могут быть написаны на основе ограниченного множества базисных структур. Этот принцип положен в основу проектирования схем, где любая логическая структура может быть создана из элементарных структур:

- структура последовательности – это формализация того, что операторы программы выполняются в порядке их появления в программе, пока что-то не изменит их последовательность;
- структура выбора – это выбор одного из двух действий исходя из выполненного некоторого условия;
- структура повторения – используется для повторного выполнения группы команд до тех пор, пока не выполнится некоторое условие. Получение правильной программы путем замены операторов программы на управляющие логические структуры называется вложением структур.

6.1.1 Структурирование. Методы структурирования программ

В основу структурирования положены следующие простые правила:

1. программа должна составляться мелкими шагами, размер шага определяется количеством решений, применяемых программистом на этом шаге;
2. сложная задача разбивается на простые легко воспринимаемые части, каждая из которых имеет только один вход и один выход;
3. логика программы должна опираться на минимальное число простых базовых управляющих структур.

Фундаментом структурного программирования является теорема о структуризации. Эта теорема устанавливает, что как бы не была сложна задача, блок-схема программы может быть представлена с использованием весьма ограниченного числа элементарных управляющих структур. Эти элементарные структуры могут соединяться между собой, образуя более сложные структуры, при этом они могут представлять собой довольно сложные блок-схемы с одним входом и с одним выходом.

Чтобы обойтись без произвольных передач управления в программе, там, где это возможно, лучше не использовать операторы GOTO, RETURN. Характерной особенностью структурированной программы является не столько отсутствие этих операторов, а сколько наличие жесткой структуры ее организации. Если текст программы будет занимать несколько десятков страниц, то восприятие такой программы будет затрудненно, поэтому рекомендуется вести структурирование текста программы (например, программа состоит из 80 страниц исходного текста, необходимо этот исходный текст заменить на текст, в котором отражаются наиболее важные, в функциональном смысле, фразы в виде сегментов). Если расписать весь программный комплекс крупными сегментами, то описание всего комплекса может занять всего одну страницу, а это наглядно и удобно.

Графическое представление программных алгоритмов имеет широкое распространение благодаря наглядности. Граф-схема представляет собой граф, состоящий из вершин, соединенных направленными стрелками-дугами.

1. вершина – 
2. функциональный оператор – 
3. условный или объединяющий операторы – 

4. останов – конечный оператор
5. дуги, определяющие последовательность выполнения действий

Программа называется простой, если ее граф-схема удовлетворяет следующим условиям: 1) один вход и один выход, 2) через каждую ее вершину проходит хотя бы один путь от входа к выходу. Сложная программа обычно имеет больше, чем один вход или выход.

Если комплекс программ большой и блок-схема программы большая необходимо выделить в ней подграфы, достаточно простые и понятные по структуре.

Структурирование программных компонент. При иерархическом построении комплекса программ важное значение имеют объем, сложность компонент для каждого уровня иерархии. По принципам построения, языку описания, объему и другим характеристикам в структуре комплекса программ можно выделить следующие иерархические уровни:

- операторов и операндов программ, соответствующих компонентам текста программ на языке программирования;
- программных модулей, оформляемых, как законченные компоненты текста программ;
- функциональных групп программ или пакета прикладных программ;
- комплекса программ, оформляемого как завершенное программное средство определенного целевого назначения.

Восходящее и нисходящее проектирование. Многоуровневый иерархический подход позволяет проектировать сложные комплексы программ по принципу «сверху – вниз» с позиции назначения и наилучшего решения основной цели, задачи всей системы.

Иногда основному проектированию «сверху – вниз» сопутствует разработка компонент «снизу – вверх». Разработка начинается с модулей нижнего уровня, далее переходят к разработке модулей следующего уровня и т.д. Достоинством этого принципа является то, что при переходе к разработке модулей более высокого уровня иерархии, модули нижних уровней можно считать готовыми и подключать их к модулям верхнего уровня на стадии отладки.

Общие правила структурного построения программных средств. Если комплекс программ строится на основе модульно иерархической структуры, состоящей из программных и информационных модулей, то на начальных этапах разработки комплекса программ формируется его структура и общие правила взаимодействия компонент. Эти правила состоят в следующем:

1. правило связи программных модулей по управлению. Передача управления вызываемому модулю всегда осуществляется через первый оператор или команду, а выход из вызываемому модуля всегда происходит через его естественное окончание, т.е. через последний оператор или команду. По окончании испытания вызываемого модуля управление передается в вызывающий модуль на оператор, следующий непосредственно за оператором

вызова. Модули нижних уровней или одного уровня иерархии могут вызываться для исполнения только модулями высших уровней, т.е. модули нижних уровней не могут вызывать модули высших уровней, а модули одного уровня – вызывать друг друга;

2. правило связи программных модулей по информации. Информация зон глобальных переменных доступна для использования любыми модулями, входящих в комплекс программ или группу программ, в соответствии с областью действия зоны глобальных переменных, т.е. глобальные переменные могут быть доступны не для всего комплекса программ, а лишь для указанной в описании группы модулей. Локальные переменные доступны лишь в пределах того модуля, в котором они определены. Для взаимодействия вызываемых и вызывающих модулей создаются зоны обменных переменных, информация из которых доступна лишь модулям непосредственно связанных по управлению. Запрещается использование для обмена информацией между модулями регистры и ячейки памяти, используемые как регистры, т.е. после окончания работы вызываемого модуля считается, что регистры не содержат информации. Информация, находящаяся в регистрах вызывающего модуля при вызове должна быть сохранена на период выполнения вызываемого модуля и восстановлена при возврате в вызывающий модуль.

3. типовая структура программного модуля. Под структурой программного модуля понимается совокупность смысловых частей, образующих модуль и используемых для различных целей при его разработке и использовании. Типовая структура модуля в общем случае включает: 1) заголовок; 2) описание переменных; 3) тело модуля; 4) точки входа и выхода.

Элементарные базовые структуры. Любую программу можно синтезировать на основе элементарных базовых конструкций трех типов:

1. простая вычислительная последовательность, заключается в преобразовании или перемещении совокупности исходных переменных. Элементарные конструкции следуют друг за другом, причем конец предыдущего оператора замыкается на начало последующего;

2. альтернатива состоит в проверке некоторого условия;

3. итерация представляет собой структуру, в которой при каждом обращении один из нескольких операторов повторяется более одного раза. Для структурирования программ число итераций должно быть задано до входа в цикл, а не определяться вычислениями внутри цикла. В результате чего исключается возможная неопределенность функционирования программы.

Простота исходных конструкций структурного программирования предотвращает появление сложных информационных связей и запутанных передач управления.

Структурированными считаются программы, которые не имеют циклов с несколькими выходами, не имеют переходов внутрь циклов или условных операторов и не имеют выходов из внутренней части циклов или условных операторов.

6.2 Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) представляет собой способ программирования, который напоминает процесс человеческого мышления. ООП более структурировано, чем другие способы программирования и позволяет создавать модульные программы с представлением данных на определенном уровне абстракции. Основная цель ООП – это повышение эффективности разработки программ.

Весь окружающий мир состоит из объектов, предметов живой и неживой природы, которые представляются как единое целое, а отдельные части объектов образуют сложное взаимодействие друг с другом. При структурном подходе программист обычно разделяет (структурирует) описываемый объект на составные части, стараясь описать свойства отдельных частей, не вдаваясь в подробности взаимодействия между ними.

Базовым в ООП является понятие объекта. Объект имеет определенные свойства. Состояние объекта задается значениями его признаков. Объект «знает», как решать определенные задачи, т.е. располагает методами решения. Программа, написанная с использованием ООП состоит из объектов, которые могут взаимодействовать между собой.

Концепция ООП заключается в том, что каждый объект является экземпляром некоторого класса объектов.

Все объекты с одинаковыми наборами атрибутов принадлежат к одному классу. Однако объединение объектов в классы определяется не набором атрибутов, а семантикой (смыслом). Так, например, объекты конюшня и лощадь могут иметь одинаковые атрибуты: цена и возраст. При этом они могут относиться к одному классу, если рассматриваются просто как товар, либо к разным классам, что более естественно.

Каждый класс имеет свои особенности поведения и характеристик, определяющих этот класс. Один класс отличается от других именем и, обычно, набором поддерживаемых интерфейсов. Интерфейсы, в свою очередь, представляют собою набор сообщений, которые можно посылать объекту.

Объектно-ориентированное программирование является наилучшим инструментом для построения иерархических деревьев или структур данных.

Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщения – это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.

Каждый объект имеет независимую память, которая состоит из других объектов. Каждый объект является представителем класса, который выражает общие свойства объектов. В классе задается поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.

Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

Абстракция данных – это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности его поведения от деталей их реализации. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Объекты представляют собою не полную информацию о реальных сущностях предметной области. Их модели, адекватны решаемой задаче, работать с ними намного удобнее, чем с низкоуровневым описанием всех возможных свойств и реакций объекта.

Инкапсуляция является важнейшим свойством объекта, на котором строится объектно-ориентированное программирование. Инкапсуляция заключается в том, что объект скрывает в себе детали, которые несущественны для использования. В традиционном подходе к программированию с использованием глобальных переменных программист не был застрахован от ошибок, связанных с использованием процедур, не предназначенных для обработки данных, но связанных с этими переменами. «Жесткое» связывание данных и процедур их обработки в одном объекте позволяет избежать многие неприятности. Инкапсуляция и является средством организации доступа к данным только через соответствующие методы.

Инкапсуляция – это принцип, согласно которому любой класс должен рассматриваться как чёрный ящик – пользователь класса должен видеть и использовать только интерфейс (от английского interface – внешнее лицо, т.е. список декларируемых свойств и методов) класса и не вникать в его внутреннюю реализацию. Этот принцип (теоретически) позволяет минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов.

Наследование – это еще одно базовое понятие ООП. Наследование позволяет определить новые объекты, используя свойства прежних, дополняя или изменяя их. Объект – наследник получает все поля и методы «родителя», к которым он может добавить свои собственные поля и методы или заменить («перекрыть») их своими методами.

Наследованием называется возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (иногда его называют суперклассом) и добавляя, при необходимости, новые свойства и методы.

Наследование призвано отобразить такое свойство реального мира, как иерархичность.

Полиморфизм заключается в том, что одно и то же имя может соответствовать различным действиям в зависимости от типа объекта. (Присваивание действию одного имени, которое затем совместно используется вниз и вверх по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, характерным именно для него.)

Полиморфизмом называют явление, при котором классы-потомки могут изменять реализацию метода класса-предка, сохраняя его сигнатуру (таким образом, сохраняя неизменным интерфейс класса-предка). Это позволяет обрабатывать объекты классов-потомков как однотипные объекты, не смотря на то, что реализация методов у них может различаться.

7 Эффективность программ

Главным критерием эффективности программ является распределение ресурсов вычислительных систем. Неравномерность задач по допустимому времени задержки или допустимой вероятности пропуска решений, а также различия параметров вычислительных систем, позволяют изменить качество решения задач выделением соответствующих ресурсов вычислительных систем. Упорядочивание последовательности решения задач и рациональное использование ресурсов вычислительных систем сокращает запаздывание в решении задач, и в некоторой степени приводит к повышению производительности вычислительных систем. Производительность вычислительных систем является одним из важнейших критериев её эффективности и методов распределения ресурсов в частности. Существуют такие понятия, как относительный и абсолютный приоритеты.

При распределении буферной памяти для приема и выдачи сообщений применяются буферные накопители, объем которых ограничивает эффективность использования. Ограничение буферных накопителей зависит от их структурного построения и распределения имеющейся памяти на зоны. На эффективность существенно влияют степень заполнения памяти и передача информации на накопители на обработку.

8 Оптимизация программ. Оптимизирующие компиляторы

Оптимизация программы – это улучшение какой-либо характеристики программы, называемой критерием оптимизации. Оптимизация программ в основном выполняется по двум основным критериям: быстродействие и объему используемых данных.

Производительность приложения определяется самым узким его участком, поэтому в первую очередь нужно определить части программы, на которых будет выполняться оптимизация.

Процесс оптимизации следует начать с профилировки программы. *Профилировкой* называют измерение производительности как всей программы,

так и отдельных ее фрагментов, с целью нахождения «горячих точек» – тех участков программы, на выполнение которых расходуется наибольшее количество времени. При этом важно отметить, что ликвидация не самых «горячих» точек программы, практически не увеличивает ее быстродействия.

Основная цель профилировки – это исследование характера поведения приложения во всех его точках. В зависимости от степени детализации в качестве «точки» рассматривается как отдельная машинная команда, так и целая конструкция высокого языка – функция, цикл, процедура.

Сложная программа состоит из большого числа функций. Нет смысла оптимизировать их все – трудоемкость такого подхода будет выше выгод, полученных от оптимизации программы целиком. Для начала необходимо локализовать участки кода с максимальной вычислительной трудоемкостью. Участки программы, которые в наибольшей степени влияют на ее производительность, в силу наиболее частого выполнения или своей ресурсоемкости называются критическим кодом. В поиске критического кода программы используют профайлеры (профилировщики) – специальные программы, которые измеряют временные затраты на выполнение участков кода программы. Профилировщики представляют возможности для оптимизации программ. К таким программам относятся Intel VTune, AMD Code Analyst, profile.exe и множество других. Наиболее мощным из них на сегодняшний день является пакет от Intel. Эта программа позволяет измерить время обработки каждой команды и вывести полную статистику о состоянии процессора при выполнении каждой команды.

Большинство современных профилировщиков поддерживают следующий набор базовых операций:

- определение общего времени исполнения каждой точки программы;
- определение удельного времени исполнения каждой точки программы;
- определение причины и/или источника конфликтов;
- определение количества вызовов той или иной точки программы;
- определение степени покрытия программы.

Основные правила оптимизации:

1. Прежде чем приступать к оптимизации, необходимо иметь надежно работающий неоптимизированный вариант.

2. Основной прирост оптимизации дает не учет особенностей системы, а алгоритмическую оптимизацию.

3. Обнаружив профилировщиком узкие места необходимо произвести оптимизацию в рамках языка высокого уровня.

Возможны ситуации, где в неудовлетворительной производительности кода виноваты процессор или подсистема памяти, а не компилятор. Лишь после анализа листинга следует приступать к ассемблерной оптимизации.

Оптимизация начинается с выделения профилировщиком критического кода и анализа его неоптимальности. Причем каждое внесенное изменение необходимо проверять профилировщиком. После завершения оптимизации

локального фрагмента программы, необходимо выполнить контрольную профилировку всей программы целиком на предмет обнаружения новых появившихся «горячих точек».

Проводя оптимизацию, не следует забывать о ее цели. Фактически идеал недостижим, поэтому оптимизацию следует завершать когда:

1. Производительность программы признана удовлетворяющей.

2. В программе отсутствуют «горячие точки», то есть количество инструкций равномерно распределено по все программе, и дальнейшая оптимизация потребует переписывания большого количества кода.

3. Сложность алгоритма настолько высока, что не представляется возможным дальнейшая оптимизация без значительных временных затрат.

4. Критическая зависимость от платформы, когда дальнейшая машинно-зависимая оптимизация приведет к потере совместимости с одной из целевых платформ.

Ко всем методам оптимизации алгоритма предъявляются следующие требования:

1. оптимизация должна быть по возможности максимально машинно-независимой и переносимой на другие платформы (операционные системы) без существенных потерь эффективности;

2. оптимизация не должна увеличивать трудоемкость разработки (в том числе тестирования) приложения более чем на 10-15%;

3. оптимизирующий алгоритм должен давать выигрыш не менее чем на 20-25% в скорости выполнения;

4. оптимизация не должна допускать безболезненное внесение изменений.

Приемы оптимизации программы можно разделить на алгоритмические и машинно-зависимые способы. В случае использования алгоритмических приемов оптимизации используются различные математические и логические методы для улучшения параметров алгоритма. Такой способ оптимизации невозможно автоматизировать, успешность его применения зависит от программиста. Способность программиста к алгоритмической оптимизации программы зависит от его понимания предметной области: владения им базовых концепций применяемых алгоритмов и особенностей предметной области программы.

В первую очередь это замена алгоритмов на более быстродействующие. Часто бывает, что более простой алгоритм показывает низкую производительность по сравнению с более сложными. Тогда, возможна замена эквивалентных алгоритмов, например, замена пузырьковой сортировки массива на быструю сортировку.

В некоторых случаях возможна оптимизация программы за счет снижение точности. В зависимости от особенностей предметной области, возможно, уменьшить разрядность представления чисел или перейти от выполнения операций с числами с плавающей запятой к целым числам или числам с фиксированной запятой.

На практике используется весьма широкий набор машинно-независимых оптимизирующих преобразований, что связано с большим разнообразием неоптимальностей. К ним относятся:

- разгрузка участков повторяемости – это такой способ оптимизации, который состоит в вынесении вычислений из многократно исполняемых участков программы на участки программы, редко исполняемые. К этому виду преобразования относятся различные чистки зон, тел циклов и тел рекурсивных процедур, когда инвариантные по результату выполнения выражения, исполняемые при каждом прохождении участка повторяемости, выносятся из него. Если размещение осуществляется перед входом в участок повторяемости, то эту ситуацию называют чисткой вверх, если же за выходом из участка повторяемости, то чисткой вниз;
- упрощение действий – этот способ оптимизации ориентирован на улучшение программы за счет замены групп (как правило, удаленных друг от друга) вычислений на группу вычислений, дающий тот же результат с точки зрения всей программы, но имеющих меньшую сложность;
- чистка программы – данный способ повышает качество программы за счет удаления из нее ненужных объектов и конструкций. Набор преобразований этого типа включает в себя следующие оптимизации: удаление идентичных операторов, удаление из программы операторов, недостижимых по управлению от начального, удаление несущественных операторов, то есть операторов, не влияющих на результат программы, удаление процедур, к которым нет обращений, удаление неиспользуемых переменных и другие;
- экономия памяти и оптимизация работы с памятью – улучшения быстродействия возможно за счет уменьшения объема памяти, отводимой под информационные объекты программы в каждом ее исполнении;
- реализация действий – это способ повышения быстродействия программы за счет выполнения определенных ее вычислений на этапе трансляции;
- сокращение программы и другие методы.

Машинно-зависимые используют особенности устройства и работы конкретной системы. Ярким примером машинно-зависимой оптимизации является векторизация операций, т.е. использование потоковых расширений процессора, таких как MMX (MultiMedia eXtensions), SSE (Streaming SIMD Extensions) и т.п.

Машино-зависимую оптимизацию можно выполнять двумя различными способами. Первый способ основан на понимании работы кодогенератора компилятора, его алгоритма и рекомендуется для приложений, в которых компилятор выбирается в начале проекта и в дальнейшем не меняется. При использовании такого способа преобразуется исходный код программы, написанный на языке высокого уровня. Для тех проектов, в которых заранее не известен компилятор (OpenSource проекты, кроссплатформенные приложения) применятся другой способ, основанный на замещении ресурсоемких участков

кода ассемблерными вставками. При такой оптимизации ухудшается переносимость кода на другие платформы.

Машинно-зависимые способы оптимизации довольно хорошо автоматизируются и большую часть их выполняют оптимизирующие компиляторы. Однако всегда остаются моменты в программе, которые можно оптимизировать вручную.

9 Отладка и сопровождение программных продуктов

9.1 Ошибки программного обеспечения: причины, источники, классификация

Программирование сложный интеллектуальный процесс, сущность и основные закономерности которого еще недостаточно изучены. В программировании не может быть незначительных ошибок или несущественных погрешностей. Пропуск запятой или отсутствие пробела между символами делает программу неработоспособной.

Систематические характеры ошибок могут служить ориентиром для разработчиков при распределении усилий при создании комплекса программ. Кроме того характеристики ошибок в процессе проектирования программного комплекса помогают:

- оценивать реальное состояние проекта и планировать трудоемкость и срок до его завершения;
- рассчитывать необходимую эффективность средств защиты от не выявленных ошибок;
- оценивать требующиеся ресурсы компьютера по памяти и производительности с учетом затрат на устранение ошибок;
- проводить исследования и осуществлять адекватный выбор показательной сложности компонент и комплекса в целом, а также некоторые другие показатели качества.

Анализ первичных ошибок в программе производиться на двух уровнях детализации:

- дифференциально – с учетом типовых ошибок, сложности и степени автоматизации их обнаружения, затрат на корректировку;
- обобщенно – по суммарным характеристикам их обнаружения в зависимости от продолжительности разработки, эксплуатации и сопровождения программных средств.

Классификация ошибок:

- Технологические ошибки. В документации и фиксировании программ в памяти компьютера составляют 5-10% от общего числа ошибок обнаруживаемых при отладке. Большинство технологических ошибок выявляется автоматически формализованными методами.
- Программные ошибки по количеству и типам в первую очередь определяются степенью автоматизации программирования и глубиной

формализованного контроля текстов программ. Количество программных ошибок зависит от квалификации разработчика и от общего объема программного комплекса, от глубины логического и информационного взаимодействия модулей и от ряда других факторов.

- Алгоритмические ошибки значительно труднее поддаются обнаружению методами формализованного автоматического контроля. К алгоритмическим следует отнести, прежде всего, ошибки, обусловленные некорректной постановкой функциональных задач. Когда в спецификациях не полностью оговорены все условия, необходимые для получения правильного результата.

- Системные ошибки – сложный комплекс программ. которые определяются неполной информацией о реальных процессах происходящих в источниках и потребителях информации. На начальных стадиях проектирования не всегда удается точно сформулировать целевую задачу всей системы, а также целевые задачи основных групп программ и эти задачи уточняются в процессе проектирования. В соответствии с этим уточняются и конкретизируются техническое задание, описание программ, и выявляются отклонения от уточненного задания, которые могут квалифицироваться как системные ошибки.

10 Перспективы развития технологий программирования

Развитием событийно управляемой концепции объектно-ориентированного подхода стало появление в 90-х годах целого класса языков программирования, которые получили название языков сценариев или скриптов. В рамках данного подхода программа представляет собой совокупность возможных сценариев обработки данных, выбор которых инициируется наступлением того или иного события (щелчок по кнопке мыши, попадание курсора в определенную позицию, изменение атрибутов того или иного объекта, переполнение буфера памяти и т.д.). События могут инициироваться как операционной системой, так и пользователем. Существенным преимуществом языков сценариев является их совместимость с передовыми инструментальными средствами автоматизированного проектирования и быстрой реализации программного обеспечения, или так называемыми CASE- (Computer-Aided Software Engineering) и RAD- (Rapid Application Development) средствами.

Одним из наиболее передовых инструментальных комплексов, предназначенных для быстрой разработки приложений, является Microsoft Visual Studio .NET. Характерные примеры сценарных языков программирования: VBScript, PowerScript, LotusScript, JavaScript.

Еще один важный этап в развитии технологий программирования – это применение *языков поддержки параллельных вычислений*. Программы,

написанные на этих языках, представляют собой совокупность описаний процессов, которые могут выполняться как в действительности одновременно, так и в псевдопараллельном режиме. Языки параллельных вычислений позволяют достичь заметного выигрыша при обработке больших массивов информации, поступающих от одновременно работающих пользователей, либо имеющих высокую интенсивность (как, например, видео-информация или звуковые данные высокого качества). Примерами языков программирования с поддержкой параллельных вычислений могут служить Erlang, Modula-3 и др.

Помимо процедурного и объектно-ориентированного подходов набирает популярность функциональный подход в программировании. По точному определению академика А.П. Ершова: «Функциональное программирование – это способ составления программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции и задание для этого имени выражения, вычисляющего значение функции, а единственным правилом композиции – оператор суперпозиции функции. Никаких ячеек памяти, ни операторов присваивания, ни циклов, ни, тем более, блок-схем, ни передач управления» [1]. Функциональный подход представлен такими языками как Racket, Clojure, F#, Haskell и др.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гниденко, И.Г. Технология разработки программного обеспечения: учеб. пособие для СПО / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. – М.: Издательство Юрайт, 2019. – 235 с.

2. Черткова, Е.А. Программная инженерия. Визуальное моделирование программных систем: учебник для СПО / Е. А. Черткова. – 2-е изд., испр. и доп. – М.: Издательство Юрайт, 2019. – 147 с.

3. Казанский, А. А. Прикладное программирование на Excel 2013: учеб. пособие для СПО / А. А. Казанский. – М.: Издательство Юрайт, 2019. – 159 с.

4. Соколова, В.В. Разработка мобильных приложений: учеб. пособие для СПО / В. В. Соколова. – М.: Издательство Юрайт, 2019. – 175 с