1. Программные ошибки

Ошибки надо не признавать. Их надо смывать. Кровью! «**Кавказская пленница, или Новые приключения Шурика**»

1.1. Что такое программная ошибка

Казалось бы простое понятие – программная ошибка, при ближайшем рассмотрении порождает ряд вопросов – от риторических до философских. Рассмотрим несколько определений.

Определение по принципу «само собой разумеется»: «Ошибка, когда программа работает неправильно». Если программа не компилируется, зацикливается или падает, то она, естественным образом содержит ошибку. В остальных случаях можно задать резонный вопрос «А судьи кто?». Одно дело, если результат программы можно явно оценить как ошибочный. Обратные примеры:

- результатом поискового алгоритма является решение, для оценки оптимальности которого требуется такой же, но заведомо правильно реализованный алгоритм? Т.е. оценка достоверности результата соизмерима с затратами на его достижение;
- в одних случаях периодическое «падение» программы считается ошибкой, ведущей к катастрофическим последствиям, а в других случаях с этим можно смириться, т.е. она не выходит за пределы потребительских качеств программы.

Определение технико-бюрократическое: «Программа содержит ошибку, если ее работа не соответствует спецификации, техническому заданию». Не все требования, касающиеся корректности работы программы, могут быть отражены в задании, представления о результатах работы программы может быть неадекватно сформулированы, возможность проверки некоторых требований может оказаться под вопросом, да и сама спецификация может быть неполной.

Определение технологическое: «Программа содержит ошибки компиляции, сборки, или времени выполнения». Слишком слабое требование, сводящееся к тому, что любая программа, которая транслируется и выдает результат – правильная. Тем не менее, языковая среда является **первым уровнем тестирования –** она накладывает ограничения на возможные формы записи программы и контролирует процесс ее исполнения. В связи с этим надо заметить, что чем больше свободы имеет программист в синтаксисе языка, чем меньше ограничений и контроля со стороны системы исполнения программы, тем больше вероятность **необнаруженных ошибок** на этом уровне, которые «перекладываются» на последующие этапы. Здесь классическим примером является Си с его архитектурно-ориентированной моделью исполнения программы – адресная арифметика, указатели и т.п..

Общий термин «программная ошибка» можно разделить на два технологических элемента:

- Причина ошибки – **дефект,** фрагмент кода, по вине которого возникает ошибка;
- Следствие – **сбой,** некорректное поведение программы, выражающееся в фатальной ошибке, зацикливании, нарушении целостности данных, дефектах структур данных, ошибках в выходных данных и т.п.

Философия ошибок

Если понимать программную ошибку как **отклонение в поведении программы от ожидаемого,** то следует, прежде всего, определить, а где находится тот идеал, с которым производится сравнение. Вообще-то он должен быть сформулирован в различных **требованиях** к программе на различных этапах ее проектирования, невыполнение которых может рассматриваться как ошибка. А основным элементом поведения является результат ее работы в виде данных, корректность которого проверяется теми или иными методами.

Формальная корректность результата. Если рассматривать программу как простое преобразование входного набора данных в выходной, то формальная правильность работы программы определяется соответствием ее результата ожидаемому набору значений. При этом возможны различные варианты, влияющие на процесс тестирования – оценки результатов и автоматизации:

- Результат очевиден или может быть вычислен вручную. Такой метод применяется, например, при отладке – путем выполнения программы со статическими тестовыми данными ограниченной размерности. Другой пример - поисковая задача, если решение всегда имеется и требуется нахождение первого подходящего;
- Результат удовлетворяет формальным критериям и проверяется, затраты на проверку соизмеримы с затратами на тестирование. Например, при тестировании сортировки результат тестирования может быть проверен формально – по расположению элементов в порядке возрастания (попарное сравнение соседей – линейный процесс) и соответствия элементов входа и выхода.
- Результат не очевиден. Для его проверки необходимо решить эту же задачу другим средством (алгоритмом). Например, поисковая задача должна возвращать оптимальное решение при возможном его отсутствии и наличии других (субоптимальных) решений.
- В области тестирования имеется такой термин как **ожидаемое поведение.** Не всегда ожидаемое поведение и, как следствие, результат теста (выходные данные) легко распределить по принципу «ошибка – терпимый недостаток».

Требования по производительности. Программа должна обеспечивать не только достоверный результат, но и получать его за приемлемое время, что соответствует требованиям производительности. При формулировании требований производительности нужно учитывать следующее:

- Производительность зависит от архитектуры аппаратных средств и системного программного обеспечения;
- Производительность программ является производной от трудоемкости используемых алгоритмов. Она, в свою очередь может иметь различную зависимость от размерности входных данных (от логарифмической до экспоненциальной). Поэтому в требованиях к производительности следует указывать диапазоны размерностей данных, на которых она обеспечивается.
- Может иметь место чувствительность алгоритмов, реализуемых в программе – различная трудоемкость (и производительность) на разных наборах входных данных.

Требования реального времени. Некоторые компоненты программы должны реагировать на заданные внешние воздействия в течение ограниченного интервала времени. Несоблюдение этих требований является ошибкой, последствия которой могут

быть вплоть до катастрофических. К требованиям реального времени также относится **реактивность программы**, т.е. способность давать отклик на действия пользователя в ограниченное время. Требования этого вида сильно зависят от архитектурных (аппаратных и программных) решений.

Требования надежности, устойчивости и безопасности. Также могут составлять существенную часть требований для программ, работающих в условиях повышенной ответственности за возможный сбой.

Требования эргономики и «человеческий фактор». Большинство общих требований графического интерфейса, а также часть требований производительности и реактивности базируется на субъективных (в лучшем случае, экспертных) оценках в категориях «удобный», «дружественный», объединяемых общим термином **Useability.** Формализация и количественная оценка требований здесь затруднена, ошибки в этой области проявляются не только в виде снижения потребительских свойств программы, но и более серьезно:

- Возможность случайно нанести ущерб случайным исполнением последовательности команд (отсутствие «защиты от дурака»);
- Снижением общей производительности работы пользователя из-за излишних действий, неудобного интерфейса, возможной недопустимой последовательности команд и т.п..

Все перечисленные требования могут быть положены в основу идеальной модели, в сравнении с которой и определяется факт наличия ошибок. Отсюда следует определение тестирования: **тестирование – целенаправленный процесс испытания программы в различных условиях с целью выявления ошибок путем сравнения ее поведения с эталонной моделью.**

Кроме того, программа функционирует не в вакууме, а в рамках аппаратно-программной архитектуры («среда исполнения – операционная система-железо»), которая может иметь значительное количество нюансов предоставления ресурсов, форматов данных, их кодирования, производительности, параллелизма, вплоть до интенсивности потоков данных и внешних событий, включая действия пользователя. Все это также может приводить к **программным ошибкам, вызванных окружением.**

Таким образом, область тестирования может расширяться бесконечно как за счет требований, так и за счет учета факторов, которые могут повлиять на ее работоспособность. Аналогично, граница между ошибкой и «милым недостатком», с которым приходится мириться, также может варьироваться.

В связи с этим следует обсудить известный тезис: «Любая программа содержит ошибку». Он является следствием **невозможности полного тестирования программы** путем простого перебора всех возможных вариантов ее исполнения в различных условиях, в число которых входят:

- возможные сочетания входных данных;
- возможные последовательности входных данных и команд;
- возможные архитектурные комбинации аппаратно-программного окружения;
- возможные временные сочетания событий, влияющих на поведение программы.

Поэтому тестирование не может гарантировать отсутствие ошибок в программе, а стремится выявить их и понизить вероятность их появления за счет следующих процессов:

- выявление факторов, влияющих на появление ошибок в программе;

- определение возможных видов ошибок, которые могут появиться в программе;
- разработка методики тестирования, сценариев и тестов, проведение тестирования;
- включение в программу кода, повышающего устойчивость программы к ошибкам.

Защита от ошибок и устойчивость программы

То, что программу нельзя тотально протестировать, означает, что она сама должна иметь средства для анализа и нейтрализации ошибок, как собственных, так и вызванных окружением. В целом это можно назвать «защитой от дурака», если под дураком понимать всю окружающую физическую и логическую реальность. Поскольку большинство ошибок окружения может быть обнаружено вызовом соответствующих функций (методов) проверки, либо сопровождается исключениями, то задача программиста состоит в том, чтобы не лениться заниматься этой рутиной в программном коде. Что же касается собственных багов, то те из них, которые приводят к исключениям, также могут быть обнаружены, зафиксированы и нейтрализованы. Перечислим основные виды ошибок, от которых следует защищаться программе:

- «защита от дурака» - проверка действий пользователя, возможно приводящих к некорректным результатам или деградации производительности (копирование больших массивов, недопустимые форматы ввода, перезапись файлов и т.п.);
- ошибки форматов – недопустимые форматы файлов или сообщений программ вследствие ошибок открытия/соединения или намеренного искажения;
- ошибки предоставления ресурсов – отсутствие ресурса или его ограниченность;
- сбои и аварийные завершения (закрытие сетевого потока до окончания передачи файла и т.п..).

Кроме средств самой защиты необходимы **средства фиксации ошибок:** сохранение максимально возможной информации об ошибке (дата, время, версия программы, окружение, стек вызова, текущие данные). Запись данных в лог-файл, отправка, по возможности, на сервер сопровождения.

1.2. Тестирование, инспекция, верификация и отладка

Если тестирование – процесс, хотя и важный, но необязательный, то отладка – обязательный начальный этап тестирования, связанный с получением минимально работоспособной версии программы. Скажем так, **отладка – бессистемное начальное тестирование программы с целью получения исходной работоспособной версии программы (альфа-версии),** годной для последующего тестирования. Кроме того, задачей тестирования является, прежде всего, обнаружение **факта ошибки,** а задачей отладки – ее **локализация и исправление.** Поэтому **отладка – процесс локализации и исправления известной ошибки.**

В настоящее время правилом хорошего тона является, как минимум, модульное тестирование, исполняемой параллельно с модульной разработкой (изменением) кода, входящей в процедуру отладки.

Верификация – способ формального доказательства правильности программного кода путем анализа утверждений о состоянии данных (предикаты) и их преобразовании операциями и операторами программы.

Основной недостаток верификации – затраты на верификацию значительно превосходят затраты на ее разработку. Тем не менее, в идеях верификации есть здравый смысл, который следует переносить на этап конструирования программы и ее отдельных модулей:

- утверждения о состоянии структур данных (**соглашения по данным).** При проектировании структур данных оговариваются допустимые конфигурации элементов и их значений, которые обязанные соблюдаться всеми компонентами программного кода, работающими с ними (например, методами, работающими со структурами данных класса). Соглашения по данным должны минимизировать возможные сочетания представлений данных (например, не использовать для представления пустой строки NULL-указателя в Си, а только указателя на пустую строку с нулевым байтом-ограничителем;
- утверждения о сохраняемых условиях на разных шагах алгоритма (**инварианты).** При конструировании и тестировании циклов и рекурсивных алгоритмов весьма продуктивным является формулировка условий, сохраняемого на каждом шаге цикла или рекурсивного алгоритма. Естественно, что они являются **параметризуемыми,** т.е. зависят от параметров, используемых в программе на текущем шаге цикла или рекурсивном вызове.

Инспекция кода

Инспекция кода – анализ работоспособного кода на предмет возможных ошибок. По поводу анализа программ см. **http://ermak.cs.nstu.ru/cprog гл.2. Анализ программ**.

Специфика процесса отладки

Специфика процесса отладки состоит в обнаружении условий и закономерностей появления дефекта и в его локализации. Далее – набор общеизвестных банальностей.

Минимизация тестовых данных. Для упрощения поиска дефекта необходимо уменьшить набор входных данных до того минимума, на котором он еще проявляется. Лучше всего для этого использовать статические данные тестовых модулей.

Эксперименты над программой. Статистика «черного ящика». Предположения об ошибке. Если минимизировать данные не удается, необходимо набрать статистику ошибочного и правильного исполнения программы над разными наборами данных, т.е. выполнить некоторый вариант функционального тестирования по методу «черного ящика». Тестовые наборы следует готовить, исходя либо из предположений о характере ошибки, либо из анализа функционала программы (наборы граничных значений).

Локализация дефекта. Метод половинного деления. Локализация дефекта производится с помощью отладчика. Участок программы или последовательность шагов ее работы делится на две части точкой останова, в которой проверяется корректность промежуточных результатов. В зависимости от этого выбирается часть кода (или последовательность шагов), содержащая дефект.

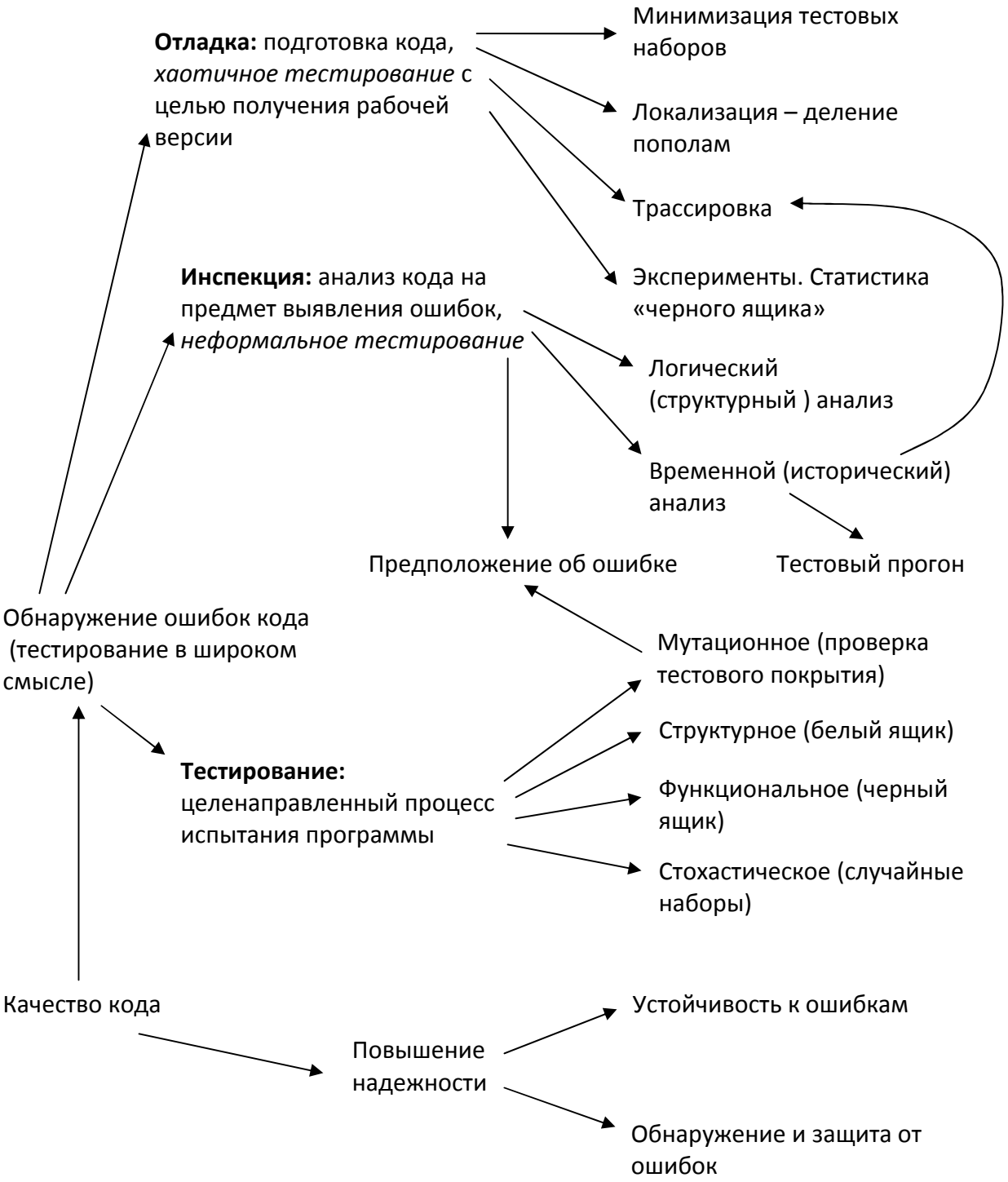
Трассировка против отладчика. Для многих ошибок необходим **временной анализ поведения программы,** цели которого могут быть разными:

- обнаружение аномальных закономерностей неправильно работающей программы;
- локализация дефекта – определение шага программы или диапазона данных, на котором он проявляется;

Для таких целей отладчик не просто не нужен, а вреден. Полезным средством является log-файл, либо обычная функция вывода в консоль нужных данных в нужных точках. Кроме того, трассировка необходима, если дефект проявляется при сложных сочетаниях данных, которые трудно уловить обычным отладчиком.

Предположение об ошибке. Инспекция кода. Локализация дефекта при отладке основывается на анализе результатов исполнения программы, а также на анализе внутренних структур данных (т.к. при дефекте они могут не удовлетворять принятым

соглашениям по данным). На основании этого делается предположение об ошибке и более детально анализируется тот код, в котором предположительно она может содержаться.



Особенности анализа программ, содержащих ошибки

Анализ поведения программы, содержащей ошибку, может быть значительно более сложным, чем правильно работающей. Для второй известно, что установленные **инварианты и соглашения по данным** соблюдаются, для первой это недостоверно. Т.е. «законы», принятые при ее проектировании, нарушены, следовательно, исходных данных для ее анализа, меньше. Само поведение может быть довольно парадоксальным. Подробнее см**. 4. Примеры отладки и тестирования**.

1.3. Характеристики ошибок

Условия появления ошибки

Ошибки могут проявляться при различных условиях, связанных как с самой программой, так и с ее окружением.

Ошибка при определенном значении или наборе значений. Ошибка детерминированно проявляется при некотором значении или наборе значений, имеющих отношение к данным одного вида.

Ошибка при определенной последовательности значений. Ошибка детерминированно проявляется при определенном сочетании значений, имеющих отношение к данным различных видов, либо вводимых в разное время в определенной последовательности.

Ошибка при определенных временных сочетаниях (гонки). Если в программе имеется внутренний параллелизм (потоки), то некоторые ошибки синхронизации могут проявляться только в определенных сочетаниях последовательности исполнения кодов из разных потоков. А так как потоки могут исполняться с произвольными скоростями, то ошибки такого рода являются **мерцающими,** т.е. они могут возникать в случайные моменты времени без каких-либо закономерностей.

Ошибки со случайным временем появления. Случайным временем появления могут характеризоваться многие другие ошибки:

- код, содержащий дефекты, может вызываться внешними событиями, происходящими в случайные моменты времени;
- для выполнения некоторых операций может не оказаться необходимых ресурсов;
- дефект вызывается при накоплении определенного количества данных;
- дефект вызывается при определенной последовательности команд пользователя или последовательности вызова методов, сама последовательность возникает случайным образом.

Момент проявления

Фактическое проявление программной ошибки не всегда может совпадать **с выполнением ошибочного кода.** Часто ошибка приводит к нарушению логической структуры данных (соглашений по данным), которые приводят к ошибкам при их последующем использовании. Например, некорректное освобождение динамической памяти в Си приводит к краху программы при последующих операциях резервирования памяти. В этом случае вторичную ошибку можно назвать **наведенной,** хотя фактически она ошибкой не является**.** *Примечание:* Сказанное не относится к ситуации, когда ошибочно вычисленные данные хранятся до момента их использования или визуализации.

Характер последствий, уровень ущерба

Снижение потребительских свойств программы. Программа может иметь неудобный интерфейс, снижающий производительность пользователя при работе с ней, подвисать при исполнении длительных операций, иметь невысокую «защиту от дурака».

Локальная устранимая ошибка. Ошибочное действие или результат легко идентифицируются и для их устранения достаточно повторения тех же или альтернативных действий.

Частичный восполнимый ущерб. Потеря или искажение результатов и данных, которые могут быть восстановлены повторением некоторой последовательности действий (например, не сохраняется файл изменений, производимых последовательностью команд). Неисполнение действий, которые могут быть повторены через некоторое время (например, перевод денежных средств).

Частичный невосполнимый ущерб. Потеря или искажение результатов и данных, неисполнение действий, для восстановления которых или повторения требуются неординарные процедуры, либо такие процедуры отсутствуют.

Катастрофа. Последствия программной ошибки связаны со значительным материальным ущербом, опасностью для здоровья и жизни, переходом в аварийный режим работы организации или предприятия.

Воздействие на программу

Ошибки оказывают влияние как **на результат** работы программы, так и на ее работоспособность.

Наблюдаемый или тестируемый ошибочный результат – несоответствие результата программы ожидаемому.

Нарушение целостности внутренних данных (дефект структуры данных). Программные ошибки могут приводить к тому, что конфигурация внутренних данных не будет соответствовать тем **соглашениям,** которые приняты при проектировании для этой структуры данных. В момент возникновения дефекта это может никак не проявиться, но зато привести к появлению **наведенных** ошибок при исполнении формально правильного кода.

Зацикливание, внешне проявляющееся как зависание программы. Зацикливание рекурсивного алгоритма приводит к фатальной ошибке, связанной с переполнением стека.

Фатальная ошибка - неуправляемое завершение программы.

Фатальная ошибка - перехваченное исключение, приводящее к авариному завершению отдельного этапа исполнения программы с последующим восстановлением, либо к завершению программы с сообщением об ошибке, либо к ее перезагрузке.

Деградация по производительности или ресурсам. Программа снижает свою производительность, либо использует значительно большее, чем обычно, количество ресурсов (например, памяти). Деградация может быть обусловлена утечками памяти, «зависшими» - неосвобожденными ресурсами (сетевыми соединениями, открытыми файлами, потоками), чувствительностью алгоритма к определенным наборам данных. Деградация может быть моментальной и накапливаемой. В последнем случае производительность постепенно снижается с течением времени, что обычно имеет место в серверных компонентах, работающих продолжительное время.

С точки зрения тестирование зацикливание и фатальная ошибка даже являются более предпочтительными, т.к. позволяют сразу идентифицировать ошибку, а неправильный результат не всегда может быть замечен сразу.

Виды ошибок по отношению к структуре алгоритма

Опечатка. Ошибка, внесенная бессознательно или автоматически в одно выражение или оператор. Исправляется редактированием. Как правило, такая ошибка обнаруживается при отладке программы, либо при **структурном тестировании** (исполнении фрагмента кода с ошибкой).

Крайняя ситуация (граничное условие). Ошибка, связанная с пропуском одного из **граничных условий** – сочетании данных, с которыми работает программа (например, пустая строка, строка, состоящая из одних пробелов и не содержащая слов). Либо ошибка, связанная с пропуском **одного шага алгоритма (цикла, рекурсии).** Основной причиной появления таких ошибок является то, что программа разрабатывается на основе анализа поведения **образной модели,** отражающей **частный случай работы программы.** Обнаруживается инспекцией кода и **функциональным тестированием** при достаточном тестовом покрытии. Исправляется путем вставки фрагмента, работающего с этим условием.

Методологическая ошибка алгоритма. Алгоритм, положенный в основу программы, не во всех случаях дает решение, либо в определенных случаях дает неверное решение. Но в отличие от ошибок граничных условий, эта частичная неработоспособность не может быть локализована и устранена. Иногда выбранный алгоритм не подходит по производительности и трудоемкости (не масштабируется по данным). Ошибка устраняется разработкой нового алгоритма решения задачи.

Методологическая ошибка архитектуры. Выбранное архитектурное решение не обеспечивает выполнение требований, предъявляемых к системе, т.е. является ошибкой проектирования.

Затраты на исправление

В зависимости от вида ошибок и структуры кода могут потребоваться различные действия:

- исправление выражения/оператора;
- исправление фрагмента кода в отдельном методе;
- рефакторинг – изменение заголовка метода, интерфейса, заголовка класса, доступа, сопровождающееся формальным изменением кода в разных частях проекта;
- реинжиниринг – изменение структуры значительной части программного кода (группы классов, характера их взаимодействия);
- изменения программной архитектуры проекта.

Причины появления ошибок

"Я понял. Оказывается, это неправильные программисты. И они, наверное, делают неправильный код". ***Винни Пух***

Причиной большинства ошибок является пресловутый «человеческий фактор». Это не означает, что во всем нужно винить автора кода, но соблюдение некоторых принципов позволяет сократить количество ошибок в программной коде, хотя бы на уровне модулей (классов) или их взаимодействия.

Незнание или отсутствие/недоступность информации. Причиной ошибки является неадекватное понимание механизма работы сервиса или службы, незнание семантики языка или особенностей среды исполнения.

Стиль редактирования или структурирования кода, способствующий появлению ошибок. При редактировании кода легко допустить ошибку, когда при формальной замене одной конструкции на другую меняется синтаксическая структура программы.

Такие ошибки формально относятся к «опечаткам», но от этого процедура их устранения не становится проще.

Типичным примером здесь является **вынесение за тело цикла** отдельных операторов или, наоборот, **попадание в тело цикла** операторов, не являющихся его частью. Чтобы их избегнуть, необходимо соблюдать правила структурного редактирования кода. В нашем случае (речь идет о Си-подобном синтаксисе) это:

- форматирование текста кода с равным отступом для всех строк одного уровня вложенности (например, того же тела цикла);
- если тело цикла – оператор, то перед добавлением в тело цикла других операторов, его нужно взять в фигурные скобки, а затем производить изменения;
- «правило парных скобок». При вводе синтаксической конструкции, содержащей компоненты в скобках, сначала вводить заголовок конструкции с пустыми парными скобками, а затем вписывать в них содержимое, например **if(){}** или **a.addActionListener(new ActionListener(){});**

Указанные замечания являются проекцией требований технологии структурного программирования (нисходящего, пошагового и пр.) на процесс ввода и редактирования кода.

Технологический стиль разработки. Все правила «хорошего тона» технологии ООП – сокрытие данных, независимость интерфейса от реализации, абстрагирование, использование шаблонов проектирования и т.п.. являются основой для потенциального снижения ошибок межмодульного взаимодействия.

Несоблюдение соглашений по данным или инвариантов. Выше в пункте «верификация» были даны определения инвариантов программного кода и соглашений по данным. Отсутствие их четкого определения или размытость на этапе проектирования может привести к ошибкам граничных ситуаций:

- Нарушение соглашений по данным - код не проверяет одно из возможных значений (например, допустимое по принятым соглашениям значение null-ссылки, что приводит к исключению);
- Нарушение инвариантов кода - одна из ветвей фрагмента кода не формирует правильных начальных значений для следующей исполняемой по времени ветви (например, одна из ветвей тела цикла не формирует правильных начальных значений следующего шага).

1.4. Тестирование на различных этапах жизненного цикла программы

Все перечисленное выше имеет отношение к технологическому процессу конструирования в жизненном цикле разработки программного продукта. В других технологических процессах также должны быть целенаправленные действия по проверке корректности разрабатываемых компонент или моделей, т.е. тестирование. Естественно, что содержание процедур тестирования (или инспекции) будет разным:

- Тестирование модели предметной области (фаза исследования);
- Тестирование требований (фазы исследования и анализа);
- Тестирование прецедентов и сценариев (фаза анализа);
- Тестирование модели классов анализа (фаза анализа);
- Тестирование архитектурного прототипа (фаза анализа);

- Тестирование модели графического интерфейса пользователя (GUI) (фаза анализа-уточнения);
- Тестирование документации (фаза распространения).

И наконец, готовый программный продукт (или прототип) могут подвергаться различному тестированию в зависимости от достигаемых при этом целей на этапе **системного тестирования**:

- Приемочное тестирование при передаче заказчику;
- Установочное тестирование для проверки правильности инсталляции;
- Альфа- и бета-тестирование при пробной эксплуатации продукта (внутри организации или на группе пользователей);
- Тестирование на соответствие требованиям спецификаций;
- Тестирование надежности и устойчивости к сбоям;
- Регрессионное тестирование – повторное тестирование после внесения изменений;
- Тестирование производительности;
- Нагрузочное тестирование (стресс-тестирование) – тестирование при повышенной рабочей нагрузке;
- Сравнительное тестирование различных версий;
- Восстановительное тестирование для проверки работоспособности после восстановления;
- Конфигурационное тестирование;
- Тестирование Useability – пользовательских характеристик системы.