Дефекты

Ошибка, дефект, сбой

Наглядно понять отличие между ошибкой, дефектом и сбоем можно на следующей истории.

Жил-был мастер. Он шил платья на заказ. Однажды он допустил **ошибку** -- забыл прошить нижний край у кармана платья. Результатом ошибки стал **дефект**. Платье висело на вешалке и выглядело абсолютно нормально, но оно было с дефектом. Маленькая девочка увидела платье и сразу влюбилась. Она купила платье и носила его повсюду. И все было хорошо, платье сидело замечательно, дефект никак не проявлялся. Пока новая хозяйка не решила положить в карман ключ. Девочка опустила руку в карман, отпустила ключ... У-у-у-упс, ключ выпал на пол! Произошел **сбой** в системе — проявился ранее скрытый дефект.

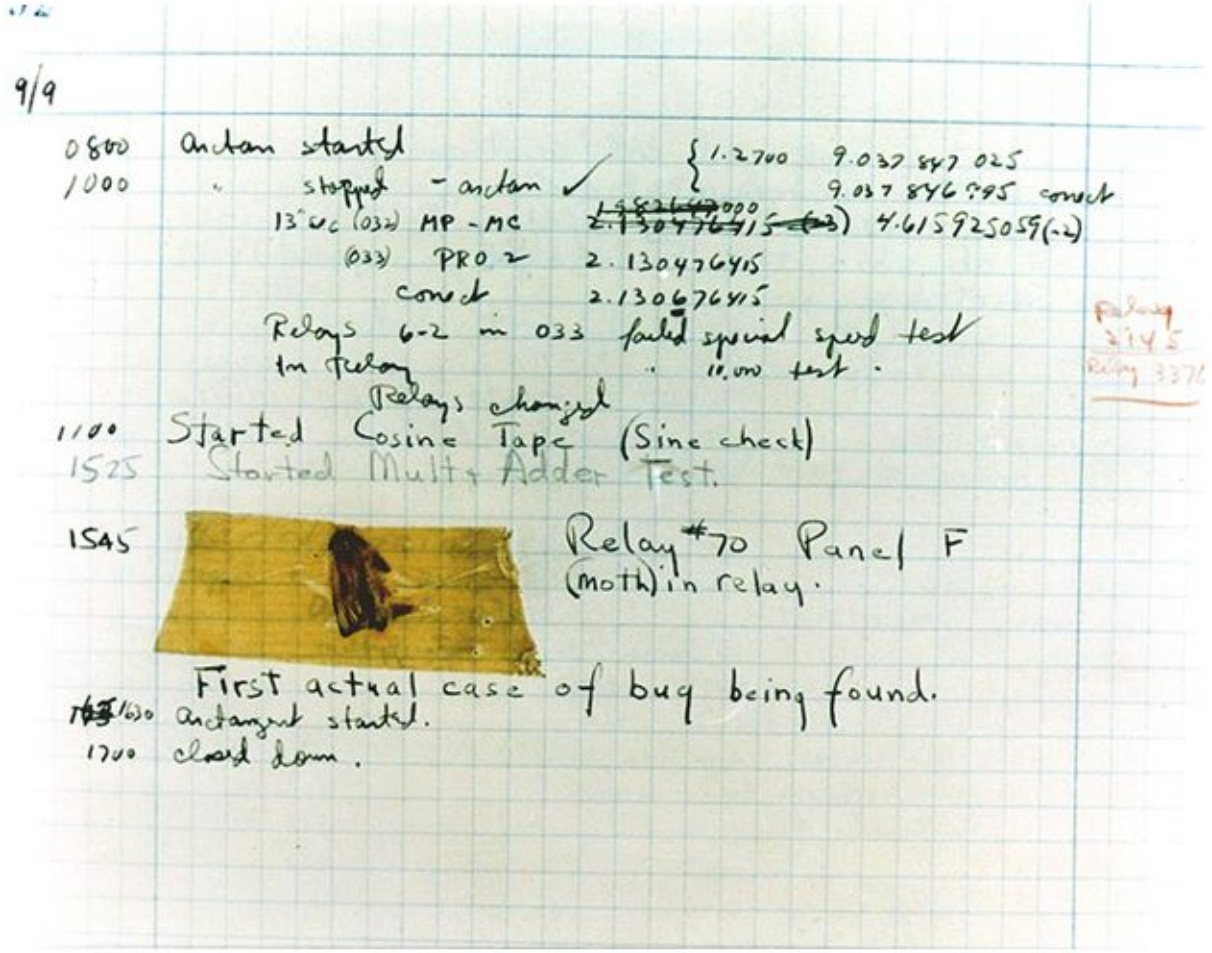
Точно так же бывает и в ПО: разработчики допускают ошибки при написании кода и в программе затаивается дефект. И даже если дефект не нашли и о нем никто не знает, он все равно есть! Сидит и ждет своего часа. И когда пользователь натыкается на ошибочный код, происходит сбой.

Баг (англ. bug -- жучок) -- слово, обозначающее ошибку в программе или системе, из-за которой программа выдает неожиданное поведение и, как следствие, результат. Говоря терминами выше, баг -- он же дефект.

В значении неуловимой технической ошибки слово “bug” употреблялось задолго до появления компьютеров персоналом телеграфных и телефонных компаний в отношении неполадок с электрооборудованием и радиотехникой. Ещё в 1878 году Томас Эдисон писал: “Так было со всеми моими изобретениями. Первый шаг -- интуиция, которая приходит как вспышка, затем возникают трудности -- устройство отказывается работать, и именно тогда проявляются “жучки” -- как называют эти мелкие ошибки и трудности -- и требуются месяцы пристального наблюдения, исследований и усилий, прежде чем дело дойдет до коммерческого успеха или неудачи”.

По одной из версий, в отношении программной ошибки этот термин впервые был применен 9-го сентября 1946-го года Грейс Хоппер, которая работала в Гарвардском университете с вычислительной машиной Harvard Mark II. Проследив возникшую

ошибку в работе программы до электромеханического реле машины, она нашла между замкнутыми контактами сгоревшего мотылька. Извлеченное насекомое было вклеено скотчем в технический дневник с сопроводительной иронической надписью: “Первый реальный случай обнаружения жучка” (англ. “First actual case of bug being found”).



Сейчас, понятно, бумажный журнал никто не использует для фиксации дефектов. Вместо этого применяют специализированные средства -- bug/issue trackers.

Системы управления дефектами (bug/issue trackers)

При разработке программного обеспечения как большие, так и маленькие софтверные компании используют системы управления дефектами. Это прикладная программа, которая помогает разработчикам учитывать и контролировать дефекты и неполадки, найденные в программах, пожелания пользователей, а также следить за процессом устранения этих дефектов и выполнения или невыполнения пожеланий. И от того как ими пользуются, может во многом зависеть продуктивность проекта. Приведем основные свойства таких систем.

- Хранят и предоставляют доступ к БД дефектов
- Управляют правами доступа
- Отправляют нужные нотификации
    - изменение жизненного цикла дефекта
    - почта/IM/SMS

- Собирают статистику и формируют отчеты

Все они могут отличаться исполнением, дизайном, возможностями, но всегда будет то общее, сама суть, что их объединяет -- наличие сообщения о найденной ошибке.

Характеристики дефектов:

- Идентификатор дефекта
- Состояние дефекта
- Содержание дефекта, шаги воспроизведения
- Контекст
- Сложность
- Серьезность/важность
- Автор
- Ответственный за исправление
- Ответственный за проверку
- Зависимость
- Временные параметры устранения дефекта
- Дополнительные материалы
- ...

Рассмотрим некоторые из них подробнее.

Идентификатор

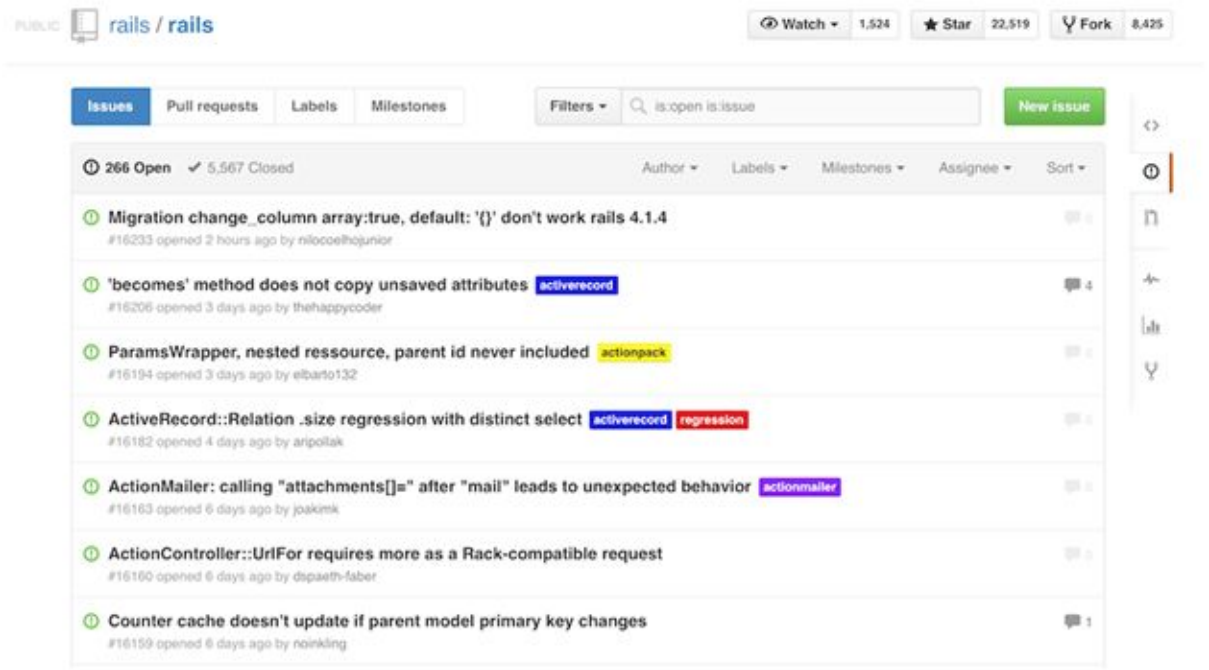
Уникальный идентификатор, выданный дефекту. Система должна иметь легкий механизм поиска дефекта по идентификатору. Идентификатор не должен повторяться ни в одном из проектов предприятия и не должен изменяться в процессе жизненного цикла проекта, так как на него могут быть ссылки в документации, в зависимостях других других дефектов и т.п.

Заголовок

Заголовок должен быть, по возможности, коротким и емким. Желательно, чтобы каждый, кто посмотрит на него первый раз, сразу понял о чем может идти речь. Т.е. правильный заголовок должен отражать суть дефекта, но в то же время не забывайте, что заголовок -- это всё-таки краткое описание, так что писать развернутое сообщение тоже не стоит. Если это исключение, значит, пишете: “исключение при...”, а не “появляется сообщение с исключением…”

Иногда люди пишут всего два-три слова в заголовок. И это неправильно. Например, “Incorrect Site Search”. О чем нам это говорит? Можно понять только то, что дефект связан с некорректной работой поиска по сайту. Но что именно не так с поиском? В чем заключается некорректное поведение? Загадка. При каких условиях воспроизводится дефект? Тоже загадка.

На картинке ниже приведен хороший список заголовков в проекте Ruby on rails.



Описание дефекта

Описание дефекта -- это самое ключевое поле. При его составлении следует придерживаться некоторых правил.

Указывайте шаги по воспроизведению дефекта. Здесь важно четко описать последовательность действий (с упоминанием всех вводимых данных), по которым можно легко воспроизвести ситуацию, приводящую к сбою. Также сохраняйте все материалы для исследования бага сразу, как только вы его обнаружили, а не ждите следующей попытки. В следующий раз дефект может не проявиться и выловить его впоследствии может оказаться довольно сложно.

Указывайте ожидаемый и фактический результаты. Т.е. вы пишете, что вы ожидали увидеть, и что на самом деле увидели.

Пишите коротко и ясно. Не нужно длинных историй с бесполезными деталями. Но и в крайности впадать тоже не надо, потому что в этом случае есть риск сделать отчет неполным и неясным.

Старайтесь взглянуть на дефект с точки зрения пользователя (не разработчика) и именно с этой точки зрения ее описать. Писать лучше в третьем лице, не нужно писать от себя, вроде: “я сделал то, а получил это...”, правильнее: “было сделано то, получилось это...”.

Один отчет о дефекте ссылается только на одну проблему. Избегайте совмещения нескольких дефектов в один. Если дефект вытекает из какого-нибудь другого, то создайте новый баг и сошлитесь на него, указав его идентификатор и название.

Во многих случаях имеет смысл прикладывать информацию, которая у вас есть: ссылки, скриншоты, видео... Это довольно удобно и позволяет:

- избежать длинного и, часто, туманного изложения сути дела пользователем;
- получателю сообщения быстрее ее понять, пользуясь визуальными средствами.

Контекст

Дефект обычно связан с каким-либо проектом или задачей. Здесь вы должны указать, к чему конкретно этот дефект имеет отношение:

- Проект/программа/задача
- Номер версии, ревизии
- Модуль, компонент

В процессе жизненного цикла контекст дефекта может уточняться.

Тип

В системах по управлению дефектами могут заноситься задачи самых разных типов:

- Ошибка функционирования
- Ошибка документации
- Предложение по усовершенствованию
- Предложение по изменению
- Новая функциональность
- ...

Серьёзность (severity)

Серьезность -- это степень негативного влияния дефекта на работоспособность приложения. Ее выставляет автор бага, тем самым показывая, насколько критические последствия этого дефекта.

Градация серьезности устанавливается индивидуально для каждой компании. Наиболее распространена следующая пятиуровневая система.

- *Блокирующий (Blocker)*. Дефект приводит приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее основными функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы.
- *Критический (Critical)*. Неправильно работает ключевая бизнес-логика, нарушается нормальная работа критичных для бизнес-модели функций. Решение проблемы необходимо для дальнейшей работы с основными функциями.
- *Значительный (Major)*. Часть основной бизнес-логики работает некорректно. Осложняется использование базовых функций.
- *Незначительный (Minor)*. Не нарушается бизнес-логика тестируемой части приложения, но осложняется использование дополнительных функций. Имеются очевидные обходные пути.
- *Тривиальный (Trivial)*. Не оказывает никакого влияния на функциональность системы, но ухудшает впечатление пользователя о ней. Например, грамматические, орфографические, пунктуационные ошибки в текстах, малозаметные ошибки в отображении страниц, зависящие от браузера и настроек пользователя и т.п.

Приоритет (priority)

Приоритет -- это атрибут, указывающий на очередность устранения дефекта. Он определяется менеджером проекта. Чем выше стоит приоритет, тем скорее нужно исправить дефект. Градации приоритета:

- *Наивысший (Top)*. Дефект крайне негативно влияет на бизнес компании и требует безотлагательного исправления, до реализации любых новых задач.
- *Высокий (High)*. Как только дефекты с наивысшим приоритетом были устранены, дефект, имеющий приоритет *High*, является следующим кандидатом на исправление. Он является критическим для проекта.
- *Обычный (Normal)*. Наличие дефекта не является критичным, но требует обязательного решения. Исправление может быть отложено до ближайших этапов/спринтов разработки. Большинство дефектов получает именно такой приоритет.
- *Низкий (Low)*. Дефект с низким приоритетом указывает на то, что проблема определенно существует, но ее решение может подождать.

Важно понимать разницу между *severity* и *priority*. Если у вас есть куча багов с высокой серьезностью -- это показатель того, что что-то идет не так. При этом большое количество дефектов с высоким приоритетом вовсе не означает, что в продукте все плохо.

Например, ситуация, когда на главной странице портала не загружаются логотип и кнопки меню из-за неправильного написания пути к файлам, должна быть исправлена безотлагательно, в то время как сам дефект имеет trivial severity. Или наоборот, вы обнаружили, что при определенной последовательности действий ваше приложение падает, но анализ проблемы показал, что дефект существовал всегда, просто раньше его не замечали, и никто из пользователей за все время жизни проекта не столкнулся с такой ситуацией. Это дефект с низким приоритетом, пока можно не исправлять, хоть и severity у него будет критический.

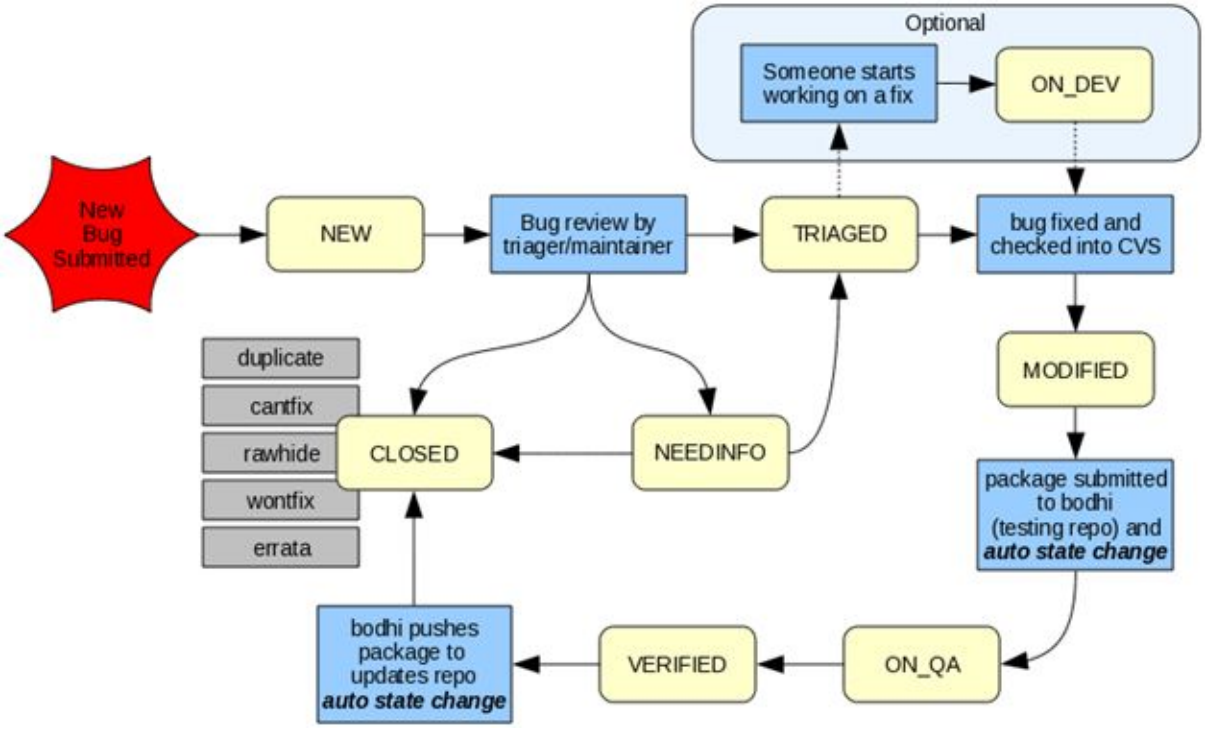
Статус

Статус дефекта отражает жизненный цикл дефекта -- последовательность этапов, которые проходит дефект на своём пути с момента его создания до окончательного закрытия. Перечислим возможные состояния.

- *Новый* -- дефект только найден и зарегистрирован в системе.
- *Взятый на исправление* -- менеджер проекта, тимлид или кто-то ещё назначает ответственного за устранение дефекта.
- *Исправленный* -- этот статус присваивается после того, как дефект был исправлен. Затем ответственный за проверку (как правило, тестировщик) должен удостовериться, что дефект действительно был устранен.
- *Закрытый* -- исправленный и проверенный.
- *Незакрытый* -- исправленный, но проваливший проверку.
- *Отклоненный* -- пишется комментарий разработчика или менеджера о причине отказа в рассмотрении.
- *Дубликат* -- в системе уже зарегистрирован аналогичный баг, поэтому этот дефект избыточен и работы по его исправлению будут проводиться на основании ранее заведенного дефекта.
- *Заново открытый* -- при повторном обнаружении дефекта после его предварительного устранения.

- *Временно приостановленный* -- не нужно исправлять на данной итерации.
- *Требующий пояснения* -- дефект описан неясно или недостаточно полно.
- *Невоспроизводимый* -- запрос дополнительной информации об условиях, в которых дефект проявляется.
- *“Не баг, а фича”* -- это не баг в программе, а одна из функций в неё заложенных. Например, за дефект принято внедренное нововведение, о котором автор бага не знает.
- …

Как видно, в течение жизни дефект может поменять кучу статусов, и вариант, при котором жизненный цикл состоит лишь из четырех стадий: “Новый” → “Взятый на исправление” → “Исправленный” → “Закрытый” -- считается идеальным и встречается достаточно редко. Обычно разработчикам приходится “развлекаться” с дефектами, перемещаясь от одного статуса к другому. На схеме ниже показан пример жизненного цикла дефекта со всевозможными поворотами.



Зависимость

Показывает зависимости исправления данного дефекта от исправления других дефектов. Зависимости представляются в виде списка идентификаторов дефектов, от которых зависит данный дефект.

Автор

Автор – лицо, обнаружившее дефект. Автором может быть тестировщик, заказчик, пользователь, автор кода или какой-то другой разработчик. Автором может быть любой, имеющий права добавлять дефекты для данного проекта.

Ответственный за исправление дефекта

Ответственный за исправление дефекта -- лицо, в задачу которого входит устранение дефекта. В зависимости от политики управления ответственный за исправление дефекта может назначаться автоматически (например, в зависимости от компонента, в котором найден баг) или может явно назначаться вручную (например, тимлидом).

Ответственный за проверку дефекта

Ответственный за проверку дефекта -- лицо, в задачу которого входит проверка успешности устранения дефекта. Ответственный за проверку -- не обязательно автор дефекта. Опять же в зависимости от политики управления ответственный за проверку дефекта может назначаться автоматически или вручную.

Временн*ы*е параметры

Временные параметры устранения -- желаемое время, когда требуется устранить дефект, либо желаемая версия проекта, к которой требуется устранить дефект.

Для больших и закрытых проектов часто приходится делать несколько баг-трекеров. Например, у проекта есть внутренний баг-трекер, к которому доступа нет ни у кого, кроме разработчиков. А для пользователей выделяется отдельный баг-трекер, в котором они добавляют всё, что хотят. В него периодически заходит кто-нибудь из разработчиков и переносит во внутренний баг-трекер то, что ему показалось актуальным. Т.е. используется так называемый прокси-баг-трекер. Но нужно понимать, что это серьезный объем ресурсов, который требуется от команды, поскольку нужно поддерживать два баг-трекера одновременно.

Отладка

Начинающий программист, как правило, переоценивает свои возможности и, проводя разработку программы, исходит из того, что в его программе ошибок не будет. А говоря про только что созданную программу, готов уверять, что она на 99% правильна, и ее остается только для большей уверенности один раз запустить с какими-нибудь произвольными данными. Исходя из такого представления о своих способностях, этот программист и строит свою работу над программой, планирует время и все такое, и каждый неверный результат, каждая найденная ошибка вызывают у него изумление и считается, конечно, последней. Вследствие такого подхода получения надежного корректного кода откладывается на длительный и весьма неопределенный срок. Только приобретя достаточный опыт, программист понимает справедливость древнего высказывания "человеку свойственно ошибаться". Оказывается, практически невозможно составить реальную сложную программу без ошибок, и почти невозможно для достаточно сложной программы быстро найти и устранить в ней **все** ошибки.

Учитывая это грустное наблюдение, разумно уже при разработке программы на этапах алгоритмизации и программирования готовиться к обнаружению ошибок на стадии отладки, принимать профилактические меры по обнаружению ошибок

(например, продумывать и реализовывать систему логгирования). Также на этом этапе следует продумать систему и методику тестирования -- определить наиболее критичные куски кода, которые будут покрываться тестами, спроектировать тесты, обдумать их корректность и полноту. У не особо опытных программистов это может вызывать нехилый когнитивный диссонанс, поскольку приходится думать о тестировании и отладке еще несуществующей программы. Однако, другого способа прокачать скилл нет. Вообще оптимизм и самоуверенность для программиста на стадии разработки строго противопоказаны. Опыт показывает, что на фазу тестирования и отладки может приходиться 30-60% всего времени, отводимого на проект.

Отладка -- этап разработки ПО, на котором воспроизводят, локализуют и устраняют ошибки. Отладка, бывает двух видов:

- *Синтаксическая отладка*. Синтаксические ошибки выявляет компилятор, поэтому исправлять их достаточно легко.
- *Семантическая (смысловая) отладка*. Ее время наступает тогда, когда синтаксических ошибок не осталось, но результаты программа выдает неверные. Здесь компилятор сам ничего выявить не сможет, хотя в среде программирования обычно существуют вспомогательные средства отладки, о которых мы еще поговорим.

Как бы тщательно мы ни писали код, отладка почти всегда занимает больше времени, чем программирование.

Исследования показали, что опытным программистам для нахождения тех же ошибок требовалось примерно в 20 раз меньше времени, чем неопытным. Более того, некоторые программисты находят больше дефектов и исправляют их грамотнее. Так что понимать методы отладки нужно и полезно.

Воспроизведение дефекта

При поиске ошибок огромную роль играет способность стабильно воспроизвести ошибку за детерминированное количество шагов. Чем меньше таких шагов, тем лучше (а совсем хорошо, если эти шаги можно тем или иным способом автоматизировать), поэтому их выявление -- это первоочередная задача. В большинстве случаев 90% времени тратится на поиск ошибки и 10% на ее исправление. В некоторых случаях этот этап может быть вполне нетривиальным, особенно если приложение работает с сетью, железом или состоит из нескольких параллельных процессов/потоков. Также ошибка может проявиться лишь при определенном состоянии окружения, длительном использовании или определенном нетривиальном случае использования. Если ошибку удается стабильно воспроизвести, полдела по исправлению уже сделано, т.к. к этому времени уже начинают формироваться первые гипотезы о причинах возникновения проблемы.

Невоспроизводимая ошибка -- неисправляемая ошибка. Если вы не можете повторить ошибку со 100%-ным эффектом, то вы никогда и никому не докажете (включая себя), что вы ее поправили, даже если вы ее поправили. Единственный способ убедиться, что ошибка исправлена -- иметь 100%-ный сценарий, который эту ошибку воспроизводит, и после изменения запустить его и в 100% случаях эту ошибку не получить.

Когда вы обнаружили сценарий воспроизводящий ошибку, то следующая стадия -- минимизация числа шагов. Чем меньше таких шагов, тем лучше. Если у вас ошибка воспроизводится за 10 шагов, вы выкинули второй шаг, а ошибка все равно повторяется, то это дает вам дополнительную информацию о том, что, скорее всего, этот шаг к ошибке не вел, и вообще неважно, что там происходило. Но здесь нужно быть аккуратным, поскольку вы можете выкинуть что-нибудь полезное.

Идеально, если вы автоматизируете ваш сценарий вплоть до скрипта и какого-нибудь автоматического теста. Вообще, хорошая практика: если вы встретили ошибку в вашей программе, напишите сразу для нее тест и добавьте его в общую систему тестирования. Возможно, вы поправили эту ошибку, и больше она никогда не возникнет. Но часто так бывает, что вы её исправили, а через месяц она опять вылезла. А если у вас написан тест, то вы всегда можете удостовериться: проявилась эта ошибка снова или нет.

Также на этом этапе очень важно сохранять непредвзятость. Выводы делать пока рано. Разумеется, воспроизведение ошибки дает вам некоторую информацию, вы ее накапливаете, но не делайте каких-либо поспешных заключений. Если вы на этом этапе начнете строить выводы и что-нибудь исправлять, то, возможно, вы что-нибудь и пофиксите, но не то, и тогда сделаете только хуже.

Локализация ошибки

Локализация — это нахождение места ошибки в программе. В процессе поиска ошибки мы обычно выполняем одни и те же действия:

- прогоняем программу и получаем результаты;
- сверяем результаты с эталонными и анализируем несоответствие;
- выявляем наличие ошибки, выдвигаем гипотезу о ее характере и месте в программе;
- проверяем текст программы, исправляем ошибку, если мы нашли ее правильно.

Для того чтобы найти ошибку, необходимо попытаться как можно сильнее упростить входные данные, действия пользователя или другие факторы, оказывающие прямое воздействие на ПО. Например, самый тупой прием -- один за другим убирать куски проблемного кода и следить за тем, что проблема от этого не пропала. Но как вы понимаете, такой “метод научного тыка” отнимает много времени и, если есть возможность, надо применять более эффективные стратегии.

Существует два способа обнаружения ошибки. Каждый из них по-своему удобен и обычно их используют совместно.

- **Аналитический**. Имея достаточное представление о структуре программы, просматриваем ее текст вручную, без прогона.
- **Экспериментальный**. Существуют две взаимодополняющие технологии.
    - *Использование отладчиков* -- программ, которые способны осуществлять пошаговое выполнение программы оператор за оператором с остановками на некоторых строках исходного кода или при достижении определённого условия. Отладчик удобен, но не всегда доступен (например, при отладке сетевых приложений или приложений реального времени просто нет времени на то, чтобы

висеть в отладчике и думать). К тому же есть языки (например, Go), где отладчиками вообще не принято пользоваться.
    - Вывод текущего состояния программы с помощью расположенных в критических точках программы операторов вывода. Получать информацию о выполнении каждого оператора тоже небезынтересно. Но здесь мы снова сталкиваемся со слишком большими объемами информации. Кроме того, мы здорово загромождаем программу добавочными операторами (которые могут даже влиять на эффективность работы основного кода), получая мало читабельный текст, да еще рискуем внести десяток новых ошибок. Так что осторожно и все такое.

Оба способа по-своему удобны и обычно используются совместно.

При локализации ошибки могут использовать следующие подходы:

- **Метод индукции** — анализ программы от частного к общему. Просматриваем симптомы ошибки и определяем данные, которые имеют к ней хоть какое-то отношение. Затем, используя тесты, исключаем маловероятные гипотезы, пока не остается одна, которую мы пытаемся уточнить и доказать.
- **Метод дедукции** — от общего к частному. Выдвигаем гипотезу, которая может объяснить ошибку, пусть и не полностью. Затем при помощи тестов эта гипотеза проверяется и доказывается.
- **Обратное движение по алгоритму**. Отладка начинается там, где впервые встретился неправильный результат. Затем работа программы прослеживается (мысленно или при помощи тестов) в обратном порядке, пока не будет обнаружено место возможной ошибки.

Место проявления ошибки != место ошибки

Важно понимать, что если у вас произошло исключение или, скажем, ошибка сегментации, то это вовсе не означает, что ошибочный код находится именно в месте, на котором приложение завершилось. Программа может испортить память в одном месте, а ошибка проявляться совсем в другом. Именно неаккуратное обращение с памятью часто приводит к мистическим ошибкам, которые начинают списывать на баги в ОС или компиляторе. Искать такую ошибку в проекте, в котором более 100 тысяч строк кода, -- занятие не для слабонервных. Важно уметь отследить проблему назад к причинам ее возникновения, потому как если пытаться исправить симптомы, самой проблемы это не исправит, и она может проявиться в дальнейшем как-нибудь еще. Для этого люди придумали профайлеры.

Беспристрастность и “замыливание”

Вы должны смотреть на всё со стороны, даже если это ваш код. Исправление -- это такой процесс, к которому нужно подходить еще более спокойно, чем к программированию. Здесь важно не спешить. Иначе вы эту ошибку поправите, но поправите ее так, что лучше бы не правили.

Если отладка зашла в тупик и обнаружить ошибку не удается, лучше отложите программу. Когда вы сидите часами и не понимаете, что происходит, полезно просто встать и прогуляться или хотя бы заняться чем-нибудь другим. Смена обстановки дает возможность по-новому на все это посмотреть. А когда глаз “замылен”, эффективность работы упорно стремится к нулю.

Вредные советы по отладке

Cargo cult programming

Заключается в добавлении кусков кода, не особо понимая их значения. Сюда же относится попытки реализации очистки памяти в языках со сборщиком мусора, создание фабрик для генерации простых объектов и т.д. Чаще всего проявляется у программистов, плохо понимающих суть происходящего, и пишущих программы методом копипаста. Либо при поддержке незнакомого или legacy кода. Либо при мифологическом мировосприятии. Также к этому может приводить чрезмерное злоупотребление паттернами или стандартами кодирования, опять же без понимания того, зачем и почему. Не скатиться в карго культ сильно помогает постоянный контроль полного осознания всех производимых действий. Карго культ программирование стало гораздо чаще проявляться с ростом функциональности сред разработки. Есть мнение, что если открыть IDEA или Visual Studio и просто ударить несколько раз головой по клавиатуре, то все равно получится компилирующийся код.

Shotgun programming

Заключается во внесении рандомных изменений в код в надежде, что баг от этого пофиксится. В стиле: “ммм, метод не работает. Ну-ка я изменю его аргумент с true на false и посмотрю, что будет!”. Почти никогда не работает и чуть менее чем всегда приводит к появлению новых багов. С другой стороны, внесение рандомных изменений может помочь формализовать симптомы. Также может сработать в многопоточных приложениях, когда добавление кода в один из потоков замедляет/ускоряет его, и приводит к изменению состояния гонки. Но опять же, увлекаться тут не стоит и каждое изменение нужно вносить подконтрольно и сознательно. Сюда же относится уже упоминавшееся хаотичное или последовательное комментирование/раскомментирование кода. Суть проста -- надо начинать комментировать код до тех пор, пока не пропадет “фантастическая” ошибка. Затем надо понемногу убирать комментарии пока не будет локализован участок, который вызывает ошибку. Не увлекайтесь этой методикой и используйте ее только в крайнем случае, когда уже ничего не помогает. Хотя часто, как первый подход к локализации ошибки, это хорошо работает.

Programming by accident

Очень распространенный стиль программирования. Проявляется тогда, когда разработчик не особо понимает, как код работает, и при этом он почему-то работает. Тогда программист дописывает еще немного кода, и он опять продолжает работать. Т.к. это происходит случайно, рано или поздно наступает момент, когда код работать перестает, и разработчик не понимает, как его исправить. И тут у него появляется два пути:

- остановиться и разобраться наконец уже в коде;
- воспользоваться shotgun debugging для отладки этой ошибки.

Не поддавайтесь второму варианту!

Least effort programming

Этот стиль очень распространен у начинающих программистов. Предположим, что вам нужно исправить возникновение NullPointerException, вы идете в ту строку, где оно проявляется, и окружаете ее условием if (pointer != null) … Ну да, в этом месте данное исключение больше бросаться не будет, но проблему-то вы не исправили совсем. Дефект вы, может быть, конкретно этот и поправили, но ошибка может проявляться не только в этом дефекте. Поэтому она обязательно вылезет в другом месте, и последствия, скорее всего, будут гораздо хуже. Фактически, своим исправлением вы маскируете реальную ошибку.

Surgical programming

Это противоположное ко всему предыдущему. Когда surgical programmer пытается исправлять ошибку, он анализирует ее причину. И причину возникновения причины. Потом он исследует последствия изменения кода, который заставляет другой код приводить к тому, что другой код работает неправильно. Потом он ищет по всему проекту использования класса из этого кода, на всякий случай. И для каждого найденного вхождения он запускает новый поиск, анализируя использования этих использований. Потом он пишет юнит-тесты для 30 разных возможных сценариев, даже для тех, которые не имеют ничего общего с той ошибкой, которую он исправляет. На всякий случай и во имя справедливости. И в итоге, полный уверенности в своей хирургической точности и гордости за проведенное с пользой рабочее время, он исправляет опечатку в имени переменной.

За это время, другой, нормальный программист уже исправил пять других ошибок.

Исправление ошибок

Принципы исправления ошибок похожи на закон Мерфи (если есть вероятность того, что какая-нибудь неприятность может случиться, то она обязательно произойдёт):

- *Там, где найдена одна ошибка, возможно, есть и другие*. Часто это связано с тем, что когда человек писал код, то был не очень сосредоточенным в этот момент, и поэтому он совершил ошибку не один раз, а несколько. Либо этому участку кода просто уделили меньше внимания.
- *Вероятность, что ошибка найдена правильно, никогда не равна ста процентам.* Действительно, вы ведь исправляете дефект, а дефект может быть вызван несколькими ошибками сразу. Вы могли устранить одну ошибку, а другую -- нет. К тому же часто исправление одной ошибки приводит к появлению новых. Более того, вы могли, например, смерджить ветку, и ошибка исчезла или перестала проявляться, но на самом деле, она где-то есть. Поэтому полезно самому себе задавать следующие вопросы:
    - Исправлена ли ошибка?
    - Исправлена ли ошибка вашим исправлением?
    - А та ли ошибка исправлена?
    - Не появились ли от этого другие ошибки?

- *Ваша задача -- найти саму ошибку, а не ее симптом*. Если программа упорно выдает результат 0.1 вместо эталонного нуля, то простым округлением вопрос не решить. Если результат получается отрицательным вместо эталонного положительного, бесполезно брать его по модулю -- вы получите вместо решения задачи ерунду с подгонкой.

Виды трудно обнаружимых ошибок (“плавающих багов”)

- Неинициализированная переменная -- переменной не присвоено значение, но в отладчике она попадает на область памяти, заполненную нулями, а в реальной работе в памяти по тому же адресу находится произвольное значение.
- Неправильный порядок инициализации -- инициализация может происходить на более поздней стадии работы, чем первое использование, при этом после инициализации проблема не проявляется (в то время как на стадии отладки порядок инициализации совпадает с порядком использования).
- Ошибка синхронизации в многозадачной среде или многопоточном приложении (состояние гонки) -- всевозможные ошибки от неправильной установки семафоров, до ошибок, связанных с взаимными приоритетами работы потоков (приоритеты при отладке и в реальной работе могут отличаться). Это серьезные ошибки, которые очень сложно отлаживать, потому что когда у вас есть два параллельно исполняющихся потока или процесса, то даже добавление отладочной печати в один из них может все сильно поменять. Действительно, если у вас состояние гонки между двумя потоками и один просто не успевает, пока второй там что-то делает, то при добавлении отладочной печати он может начать успевать -- и тогда у вас вообще все поедет.
- Аппаратная ошибка или ошибка на стыке программно-аппаратного интерфейса. На эмуляторе все работает, но с реальным железом все всегда совсем немного по-другому.

Эти ошибки очень сложно искать, поскольку проявляются в зависимости от случайных факторов и воспроизводятся нестабильно.

Важно также осознавать, что код, скомпилированный под дебагом -- это совсем не тот код, который скомпилирован под релизом. Дебажная версия содержит в себе большое количество различной отладочной информации, также в ней содержится код, соответствующий ассертам (о них мы поговорим ниже). Некоторые переменные могут храниться на стеке вместо регистров. В релизной версии ассертов и отладочной информации нет, к тому же компилятор может проводить над кодом разного рода оптимизации, что, собственно говоря, меняет код.

Защитное программирование

Мы поговорили о дефектах, выяснили, как с ними бороться, а теперь поговорим о том, как их не допускать. Очень многие программисты пишут код, напоминающий непрочные башни из кубиков: одного легкого толчка в основание достаточно, чтобы обрушить всю конструкцию. Код должен строиться последовательными слоями и

нужно применять технологии, гарантирующие прочность каждого уровня, чтобы на нем можно было возводить следующий.

Есть большая разница между кодом, который на первый взгляд работает, правильным кодом и хорошим кодом:

- легко написать код, который почти всегда работает. Вводишь в программу обычные данные и получаешь обычные результаты. Но стоит подать на вход нечто необычное, и все может рухнуть;
- правильный код не рухнет. Для любого набора входных данных результат будет корректен. Однако обычно количество всевозможных комбинаций входных данных оказывается невероятно большим, и все их трудно протестировать;
- однако не всякий правильный код оказывается хорошим -- например, его логику трудно проследить, код непонятен, его практически невозможно сопровождать.

Следуя этим определениям, вы должны стремиться к созданию хорошего кода. Перечислим основные требования, которым он должен удовлетворять:

- Надёжный
- Эффективный
- Понятный
- Правильный
    - делает то, что должен
    - не делает того, чего не должен
    - не боится непредвиденных ситуаций
        - не делает предположений относительно окружения
        - готов всегда и ко всему
        - способен пережить некорректные ситуации

Одно дело писать такой хороший код, находясь в удобных домашних условиях, которыми вы вполне управляете. Совершенно иная перспектива -- заниматься этим в бурных условиях производства, когда обстановка непрерывно меняется, объем кода быстро растет и постоянно приходится сталкиваться с фантастическим старым кодом -- архаичными программами, авторов которых найти невозможно. Попробуйте написать хороший код, когда все направлено против вас!

Как в таких условиях обеспечить профессиональное качество кода? На помощь приходит защитное программирование.

Что такое защитное программирование?

При всем многообразии методов разработки кода (объектно-ориентированное программирование, модели, основанные на компонентах, структурное проектирование, экстремальное программирование и т.д.) защитное программирование оказывается универсальным подходом. Оно представляет собой не столько формальную технологию, сколько неформальный набор основных принципов. Защитное программирование -- не волшебное средство от всех болезней, а практический способ предотвратить множество проблем кодирования.

Во время написания кода невольно делаются какие-то допущения о том, как он будет выполняться, как будет происходить его вызов, какие входные данные допустимы и т.д. Можно даже не заметить сделанные допущения, потому что они

кажутся очевидными. Проходят месяцы беззаботного кодирования, и прежние предположения все более стираются из памяти.

Допущения служат причиной появления кода с ошибками. Очень легко предположить следующее:

- функцию никогда не станут вызывать таким способом. Ей всегда будут передаваться только допустимые параметры;
- этот фрагмент кода всегда будет работать, он никогда не сгенерирует ошибку;
- никто не станет пытаться обратиться к этой переменной, если я напишу в документации, что она предназначена только для внутреннего употребления.

В защитном программировании нельзя делать никаких допущений. Нельзя предполагать, что некое событие никогда не случится. Никогда нельзя предполагать, что все в мире будет происходить так, как вам бы того хотелось.

Чем больше объем написанного вами кода и чем быстрее вы его проглядываете, тем выше вероятность допустить ошибку. Нельзя написать устойчивый код, не потратив время, необходимое для проверки каждого предположения. К сожалению, в спешке, в которой создаются программы, редко есть возможность притормозить, критически оценить сделанное и поразмыслить над фрагментом кода. Слишком быстро происходят события в мире, и программистам нельзя отставать. Следовательно, нужно использовать любые возможности для сокращения числа ошибок, и защитные меры оказываются одним из главных видов нашего оружия

Защитное программирование -- это метод профилактики, а не способ лечения. Этим оно отличается от отладки, т.е. устранения ошибок после того, как они дали о себе знать. Отладка -- это именно поиск способа лечения. Защитное программирование нужно в первую очередь для того, чтобы предотвратить отказ (или обнаружить ошибки на раннем этапе, пока они не проявили себя недопустимым образом, который потребует от вас отлаживать программу всю ночь). Защитное программирование чем-то похоже на анализ рисков: вы пытаетесь предугадывать, тратить время и усилие на предотвращение проблем до того, как они произойдут.

Действительно ли защитное программирование стоит этих хлопот? Рассмотрим аргументы за и против.

Против

Защитное программирование требует дополнительных ресурсов -- ваших и компьютера.

- Каждая защитная мера требует дополнительной работы. Вы тратите больше времени и усилий на написание кода.
- Снижается эффективность кода: даже небольшой дополнительный код требует выполнения лишних операций. Снижение эффективности может быть незаметным, когда касается одной функции или класса, но если в системе 100 000 функций, могут возникнуть проблемы.
- Снижается читаемость кода, поскольку для всех предположений делаются явные проверки в коде.

За

- Мы можем заранее предвидеть возможные дефекты или хотя бы пытаться их предугадать путем выяснения неприятностей, способных произойти на каждом этапе выполнения кода, и принять защитные меры.

- Намного сокращается время отладки, и мы можем заняться чем-то более интересным в освободившееся время. Вспомните Мёрфи: если вашим кодом можно воспользоваться некорректно, так оно и случится.
- Код, который выполняется правильно, но немного медленнее, значительно предпочтительнее кода, который почти всегда работает правильно, но временами с большим треском падает.
- Иногда защитный код проектируется так, что его можно удалить из окончательной версии, и таким образом проблема быстродействия снимается.
- Повышается безопасность программ, предохраняя от умышленного злоупотребления. Взломщики и создатели вирусов не упускают случая воспользоваться неаккуратно написанным кодом, чтобы получить контроль над приложением и реализовать свои зловредные планы. Это серьезная угроза в современном мире программных разработок, которая приводит к огромному ущербу, наносимому продуктивности, финансам и конфиденциальности.

Общие принципы

Защитное программирование предполагает соблюдение ряда разумных правил. Существует масса простых приемов, которые неизмеримо повысят надежность вашего кода. Несмотря на свою очевидность, они часто игнорируются, откуда и вытекает низкое качество большей части существующего программного обеспечения. Большей безопасности и надежной разработки можно достичь на удивление легко, если только программисты проявят бдительность и компетентность.

Хороший стиль кодирования. Значительной части ошибок можно избежать, придерживаясь добротного стиля кодирования. Такие простые вещи, как выбор осмысленных имен переменных и разумная расстановка скобок делают код понятнее и уменьшают шансы пропустить ошибку.

Хорошая архитектура. Точно так же очень важно изучить проект в целом, прежде чем погружаться в написание кода. “Лучшая документация компьютерной программы -- это ее четкая структура”. Если вы начнете с того, что реализуете понятные API, определите логичную структуру системы и четкие роли и задачи компонент, то избавитесь от возможной головной боли в будущем.

Чем больше спешки, тем меньше скорость. Сплошь и рядом программы пишут сломя голову. Программист быстренько создаёт функцию, пропускает ее через компилятор для проверки синтаксиса, запускает, чтобы убедиться в ее работоспособности, и переходит к очередной задаче. Такой подход чреват опасными последствиями.

Правило: необходимо обдумывать каждую новую строку. Какие ошибки могут в ней возникнуть? Все ли возможные повороты логики вы рассмотрели? Медленное и методичное программирование может показаться скучным, но оно действительно сокращает количество ошибок в программе. Пример ловушки, которая поджидает спешащих программистов, пишущих на языках семейства C -- это ошибочный ввод “=” вместо “==”. Компилятор не покажет вам, что программа будет выполняться не так, как вам хотелось.

Дисциплина. Дисциплинированность -- это привычка, которую надо усвоить и укреплять. Не бросайтесь вперед, пока не выполните все задачи, необходимые для

завершения работы с каким-то разделом кода. Например, если вы собираетесь сначала написать основной код, а потом заняться проверкой и обработкой ошибок, вы должны твердо решить, что сделаете и то, и другое. Никогда не откладывайте на будущее проверку ошибок ради того, чтобы написать основной код еще нескольких разделов. Ваше намерение вернуться к обработке ошибок позже может быть вполне искренним, но “позже” может превратиться в “гораздо позже”, когда вы в значительной мере забудете контекст, что в результате потребует еще большего труда и времени.

Не верьте никому. Разработка хороших программ требует недоверия к человеческой натуре. Даже добропорядочные пользователи могут вызвать проблемы у вашей программы. У вас могут возникнуть проблемы по следующим причинам:

- обычный пользователь случайно введет в программу неверные данные или воспользуется ею некорректно;
- злоумышленник сознательно попытается заставить программу вести себя некорректно;
- клиентский код вызовет вашу функцию, неправильно передав ей параметры или задав им недопустимые значения;
- операционная среда не сможет предоставить программе необходимый сервер;
- внешние библиотеки окажутся некорректными и не выполнят те контракты по интерфейсам, на которые вы полагались.

А может быть, вы сами сделаете глупую ошибку в какой-то функции или забудете, как работает код, написанный вами три года назад, и неправильно воспользуетесь им. Не рассчитывайте, что все будет хорошо и весь код будет работать корректно. Ставьте проверки везде, где только можно. Все время ищите слабые точки и помещайте в них дополнительные средства защиты.

Стремитесь к ясности, а не к краткости. Когда встает выбор между кратким (но непонятным) и ясным (но скучным) кодом, делайте его в пользу того кода, смысл которого понятен, даже если он менее элегантен. Например, сложные арифметические выражения разбивайте на последовательность отдельных операторов, логика которых понятнее.

Подумайте о тех, кто будет читать ваш код. Возможно, его будет сопровождать менее опытный кодировщик, и если он не разберется в логике, то может сделать ошибки. Сложные конструкции или оригинальные фокусы с языком могут свидетельствовать о ваших энциклопедических знаниях в области приоритетов операторов, но сильно ослабят возможность сопровождения кода. Будьте проще.

Код, который нельзя сопровождать, не может быть безопасным. В отдельных случаях чрезмерно сложные выражения приводят к генерации компилятором некорректного кода -- так часто обнаруживаются ошибки в оптимизации, проводимой компилятором.

Максимальная закрытость. То, что является внутренним делом, должно оставаться внутри. Ваши “личные вещи” должны храниться под замком. Несмотря на любые ваши просьбы, стоит только отвернуться, и люди начнут копаться в ваших данных, если вы оставите им для этого малейшую возможность, и станут вызывать “внутренние” функции по своему усмотрению. Не позволяйте им этого делать!

Включайте вывод всех предупреждений при компиляции. Большинство компиляторов выдает массу сообщений об ошибках, когда их что-то не устраивает в коде. Они также выводят различные предупреждения, если им кажется, что в коде

может быть ошибка, например в C или C++ использование переменной до того, как ей присваивается значение. Обычно вывод таких предупреждений можно избирательно включать или отключать.

Если ваш код изобилует опасными конструкциями, такие предупреждения могут занять многие страницы. К сожалению, по этой причине очень часто отключают вывод предупреждений компилятором или просто не обращают на них внимания. Так поступать не следует.

Всегда включайте вывод предупреждений компилятором, и если ваш код генерирует предупреждения, немедленно исправьте его, чтобы таких сообщений больше не было. Нельзя успокаиваться, пока компиляция не станет проходить гладко при включенном выводе предупреждений. Они существуют не зря. Даже когда вам кажется, что какое-то предупреждение несущественно, добейтесь его исчезновения, потому что в один прекрасный день из-за него вы не заметите того предупреждения, которое окажется действительно важным. Вообще, во многих проектах сборка устроена так, что предупреждения компиляции валят сборку, просто билд считается не пройденным.

Пользуйтесь средствами статистического анализа. Предупреждения компилятора представляют собой результат частичного статического анализа кода -- контроля кода, проводимого перед запуском программы. Существует ряд самостоятельных инструментов статического анализа, например lint или его более современные потомки для C и FxCop для сборок .NET. Вашей постоянной практикой должна стать проверка своего кода с помощью этих средств. Они обнаруживают гораздо больше ошибок, чем компилятор.

Пользуйтесь хорошими средствами регистрации диагностических сообщений. Когда пишут новый код, то часто включают в него много операторов вывода диагностики, чтобы разобраться в происходящем в программе. Одна из самых распространенных практик -- логирование. С помощью него удобно следить за процессом выполнения бизнес-логики проекта.

Проверяйте все возвращаемые значения. Чаще всего коварные ошибки возникают тогда, когда программист не проверяет возвращаемые значения. Если функция возвращает значение, то делает это не зря.

Проверяйте все данные из внешних источников. Получив данные из файла, от пользователя, из сети или любого другого внешнего интерфейса, удостоверьтесь, что все значения попадают в допустимый интервал. Проверьте, что числовые данные имеют разрешенные значения, а строки достаточно коротки, чтобы их можно было обработать. Если строка должна содержать определенный набор значений (скажем, идентификатор финансовой транзакции или что-либо подобное), проконтролируйте, что это значение допустимо в данном случае, если же нет -- как-нибудь обработайте эту ситуацию.

Проверяйте значения всех входных параметров метода. Проверка значений входных параметров метода практически то же самое, что и проверка данных из внешнего источника, за исключением того, что данные поступают из другого метода, а не из внешнего интерфейса.

Ограничения (утверждения)

Как уже говорилось, при составлении программы мы делаем ряд неявных предположений. Но как физически учесть эти предположения в коде, чтобы они не привели к возникновению неуловимых проблем? Нужно просто написать некоторый дополнительный код для проверки всех этих условий. Такой код выступает в роли документации каждого предположения, переводя его в явный вид. В результате мы приводим в систему *ограничения*, налагаемые на функциональность и поведение программы.

Утверждение/ограничение (assertion) -- это код (обычно метод или макрос), используемый во время разработки, с помощью которого программа проверяет правильность своего выполнения. Если утверждение истинно, то все работает так, как ожидалось. Если ложно -- значит, в коде обнаружена ошибка и программа завершается.

Обычно утверждение принимает два аргумента: логическое выражение, описывающее предположение, которое должно быть истинным, и сообщение, выводимое в противном случае. Вот как будет выглядеть утверждение на языке Java, если переменная denominator должна быть ненулевой:

```
assert denominator != 0 : “denominator is unexpectedly equal to 0”
```

Утверждения не предназначены для показа сообщений в релизной версии, они в основном применяются при разработке и поддержке. Обычно их добавляют при компиляции кода во время разработки и удаляют при компиляции релизной версии. В период разработки утверждения выявляют противоречивые допущения, непредвиденные условия, некорректные значения, переданные методам, и т.п. При компиляции релизной версии они могут быть удалены и, таким образом, не повлияют на производительность системы.

Общие принципы использования утверждений

Используйте утверждения для документирования и проверки предусловий, постусловий и инвариантов. *Предусловия* -- это соглашения, которые клиентский код, вызывающий метод или класс, обещает выполнить до вызова метода или создания экземпляра объекта. Предусловия -- это обязательства клиентского кода перед кодом, который он вызывает.

Постусловия -- это соглашения, которые метод или класс обещает выполнить при завершении своей работы. Постусловия -- это обязательства метода или класса перед кодом, который их использует.

Инварианты -- это условия, которые должны быть выполнены при достижении в ходе выполнения программы определенной точки, например между проходами цикла, при вызове методов и т. п.

Используйте процедуры обработки ошибок для ожидаемых событий и утверждения для событий, которые происходить не должны. Утверждения проверяют условия событий, которые никогда не должны происходить. Обработчик ошибок проверяет внештатные события, которые могут и не происходить слишком

часто, но были предусмотрены писавшим код программистом и должны обрабатываться и в релизной версии. Обработчик ошибок обычно проверяет некорректные входные данные, утверждения -- ошибки в программе.

Если для обработки аномальной ситуации служит обработчик ошибок, он позволит программе адекватно отреагировать на ошибку. Если же в случае аномальной ситуации сработало утверждение, для исправления просто отреагировать на ошибку мало -- необходимо изменить исходный код программы, перекомпилировать и выпустить новую версию ПО.

Будет правильно рассматривать утверждения как выполняемую документацию -- работать программу с их помощью вы не заставите, но вы можете документировать допущения в коде более активно, чем это делают комментарии языка программирования.

Старайтесь не помещать выполняемый код в утверждения. Если в утверждении содержится код, возникает возможность удаления этого кода компилятором при отключении утверждений. Допустим, у вас есть следующее утверждение:

```
Debug.Assert( PerformAction() ) “Невозможно выполнить действие”
```

Проблема здесь в том, что, если вы не компилируете утверждения, вы не компилируете и код, который выполняет указанное действие. Вместо этого поместите выполняемые выражения в отдельных строках, присвойте результаты статусным переменным и проверяйте значения этих переменных.

Для большей устойчивости кода проверяйте утверждения, а затем все равно обработайте возможные ошибки. Каждая потенциально ошибочная ситуация обычно проверяется или утверждением, или кодом обработчика ошибок, но не тем и другим вместе. Некоторые эксперты утверждают, что необходим только один тип проверки.

Однако реальные программы и проекты бывают слишком запутанными, чтобы можно было полагаться на одни лишь утверждения. В больших, долгоживущих системах различные части могут разрабатываться несколькими проектировщиками 5–10 лет и более. Разработка будет производиться в разное время и в разных версиях продукта. Эти проекты будут основаны на разных технологиях и сосредоточены на различных вопросах разработки системы. Проектировщики могут быть удалены друг от друга географически, особенно если элементы системы приобретались у независимых компаний. Программисты будут использовать различные стандарты кодирования в разное время жизни системы. В большой команде разработчиков некоторые неминуемо будут добросовестнее других, поэтому часть кода будет проверяться более тщательно, чем остальная. В любом случае, когда тестовые команды работают в нескольких географических регионах, а требования бизнеса приводят к изменению тестового покрытия от версии к версии, рассчитывать на всестороннее низкоуровневое тестирование системы нельзя.

Способы обработки ошибок

Утверждения применяют для обработки ошибок, которые никогда не должны происходить. А что делать с возможными ошибками? Рассмотрим основные приемы.

Вернуть нейтральное значение. Иногда наилучшей реакцией на неправильные данные будет продолжение выполнения и возврат заведомо безопасного значения. Численные расчеты могут возвращать 0. Операция со строкой может вернуть пустую строку, а операция с указателем -- пустой указатель. Метод рисования в видеоигре, получивший неправильное исходное значение цвета, может по умолчанию использовать цвет фона или изображения. Однако в методе рисования рентгеновского снимка ракового больного вряд ли стоит применять “нейтральное значение”. В таких случаях лучше прекратить выполнение программы, чем показать пациенту неправильные результаты.

Заменить следующим корректным блоком данных. Условия обработки потока данных иногда таковы, что следует просто вернуть следующие допустимые данные. Если при чтении информации из базы данных встречена испорченная запись, можно просто продолжить считывание, пока не будут найдены корректные данные. Если вы считываете показания термометра 100 раз в секунду и один раз не получили достоверного измерения, можно просто подождать 1/100 секунды и обратиться к следующему показанию.

Вернуть тот же результат, что и в предыдущий раз. Если программа считывания показаний термометра один раз не получила измерение, она может просто вернуть то же значение, что и в предыдущий раз. В зависимости от приложения температура, скорее всего, не сильно изменится за 1/100 секунды. Если в видеоигре запросу на прорисовку части экрана передано неверное значение цвета, вы можете просто вернуть тот же цвет, что и раньше. Но, авторизуя транзакции в банкомате, вы, пожалуй, не захотите использовать “то же значение, что и в предыдущий раз” -- ведь это будет номер счета предыдущего клиента!

Подставить ближайшее допустимое значение. В некоторых случаях вы можете вернуть ближайшее допустимое значение. Часто это обоснованный подход для получения показаний откалиброванных инструментов. Так, термометр мог бы быть откалиброван от 0 до 100 градусов по Цельсию. Если вы получаете значение меньше 0, можно заменить его на 0, как ближайшее допустимое значение. Если же значение больше 100, можно подставить 100. Если в операции со строкой ее длина заявлена меньшей 0, можно принять ее за 0. Например, автомобиль использует этот подход к обработке ошибок, когда он двигается задним ходом. Так как спидометр не показывает отрицательную скорость, то при движении задним ходом, скорость просто равна 0 -- ближайшему допустимому значению.

Записать предупреждающее сообщение в лог. Обнаружив неверные данные, вы можете решить записать предупреждение в файл журнала и продолжить работу. Этот подход можно сочетать с другими способами, такими как подстановка ближайшего допустимого значения или замена следующим корректным блоком данных. Используя такой журнальный файл, задумайтесь, можно ли его безопасно сделать общедоступным или же его надо зашифровывать либо защищать каким-либо иначе.

Вернуть код ошибки. Вы можете решить, что только определенные части системы будут обрабатывать ошибки. Другие же не будут обрабатывать ошибки локально, а будут просто сообщать, что обнаружена ошибка, и надеяться, что какой-либо другой вышестоящая в иерархии вызовов метод эту ошибку обработает. Конкретный механизм оповещения остальной системы об ошибке может быть следующим:

- установить значение статусной переменной;
- вернуть статус в качестве возвращаемого значения функции;
- сгенерировать исключение, используя встроенный в язык программирования механизм обработки исключений.

В этом случае не столь важно выбрать механизм обработки ошибок, как решить, какая часть системы будет обрабатывать ошибки напрямую, а какая -- только сообщать об их возникновении. Если система должна быть безопасной, убедитесь, что вызывающие методы всегда проверяют коды возврата.

Прекратить выполнение. Некоторые системы прекращают работу при возникновении любой ошибки. Этот подход оправдан в приложениях, критичных к безопасности. Например, какая реакция на ошибку будет наилучшей, если ПО, контролирующее радиационное оборудование для лечения рака, получит некорректное значение радиационной дозы? Надо ли использовать то же значение, что и в предыдущий раз? А может, ближайшее допустимое или нейтральное значение? В этом случае остановка работы -- наилучший вариант. Мы охотнее предпочтем перезагрузить машину, чем рискнуть применить неправильную дозу.

Устойчивость против корректности

Как нам показали примеры с видеоигрой и рентгеновской установкой, выбор подходящего метода обработки ошибки зависит от приложения, в котором эта ошибка происходит. Кроме того, обработка ошибок в общем случае может стремиться либо к большей корректности, либо к большей устойчивости кода. Разработчики привыкли применять эти термины неформально, но, строго говоря, эти термины находятся на разных концах шкалы.

Корректность предполагает, что нельзя возвращать неточный результат; лучше не вернуть ничего, чем неточное значение. *Устойчивость* требует всегда пытаться сделать что-то, что позволит программе продолжить работу, даже если это приведет к частично неверным результатам.

Приложения, требовательные к безопасности, часто предпочитают корректность устойчивости. Лучше не вернуть никакого результата, чем неправильный результат. Радиационная машина -- хороший пример применения такого принципа.

В потребительских приложениях устойчивость, напротив, предпочтительнее корректности. Какой-то результат всегда лучше, чем прекращение работы.

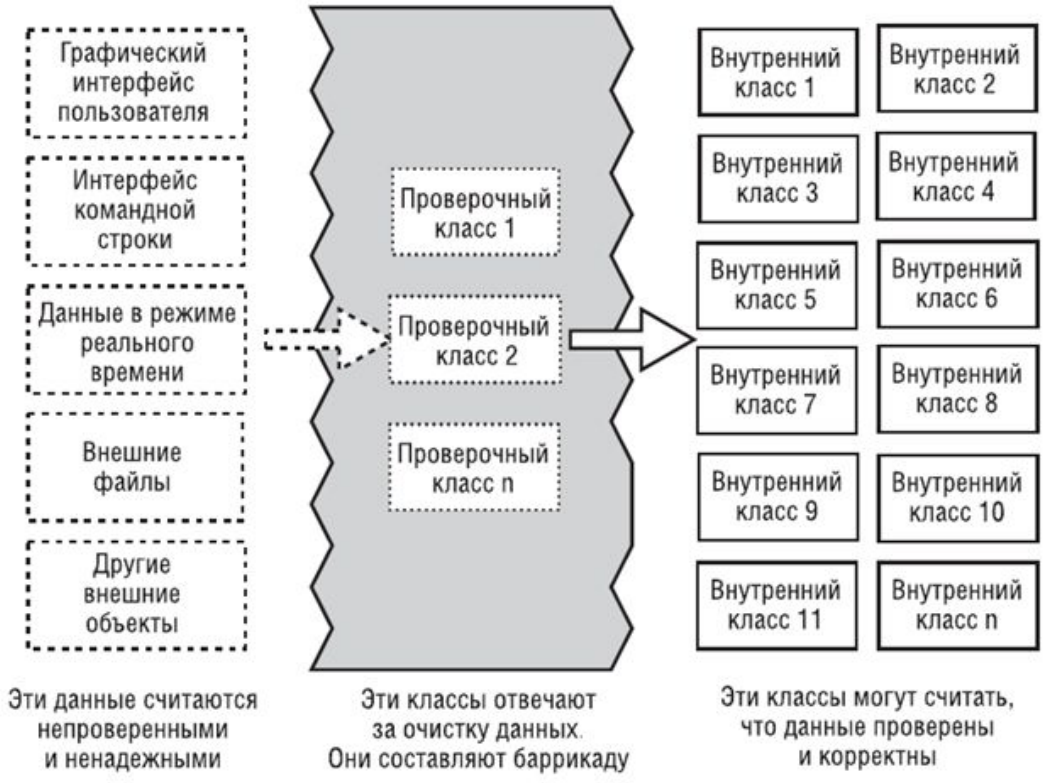
Устойчивость vs корректность -- это всегда альтернатива, которую вы должны выбрать для себя сами. И что важно, это альтернатива, которую вы выбираете на самом раннем этапе разработки вашего софта, при проектировании архитектуры.

Изоляция повреждений, вызванных ошибками

Изоляция повреждений, или *баррикада* -- это стратегия, сходная с тем, как изолируются отсеки в трюме корабля. Если корабль налетает на айсберг и в днище появляется пробоина, отсеки задраиваются, и остальная часть корабля не страдает. Баррикады также аналогичны брандмауэрам, предотвращающим распространение огня из одной части здания в другую.

Один из способов изоляции в целях защитного программирования состоит в разработке набора интерфейсов в качестве оболочки для “безопасных” частей кода. Проверяйте корректность данных, пересекающих границу безопасной области, и реагируйте соответственно, если данные неправильные.

Выделение части кода для работы с непроверенными данными, а части -- для работы с только корректными данными, может быть эффективным способом освободить большую часть программы от ответственности за проверку допустимости данных.



Тот же подход применим и на уровне класса. Открытые методы класса предполагают, что данные небезопасны и отвечают за их проверку и исправление. Если данные были проверены открытыми методами класса, закрытые методы могут считать, что данные безопасны.

Связь между баррикадами и утверждениями

Применение баррикад делает отчетливым различие между утверждениями и обработкой ошибок. Методы с внешней стороны баррикады должны использовать обработчики ошибок, поскольку небезопасно делать любые предположения о данных.

Методы внутри баррикад должны использовать утверждения, т.к. данные, переданные им, считаются проверенными при прохождении баррикады. Если один из методов внутри баррикады обнаруживает некорректные данные, это следует считать ошибкой в программе, а не в данных.

Использование баррикад также иллюстрирует значимость принятия решения об обработке ошибок на уровне архитектуры. Решение, какой код находится внутри, а какой -- снаружи баррикады, принимается на уровне архитектуры.

Доля защитного программирования в промышленной версии

Один из парадоксов защитного программирования состоит в том, что во время разработки вы бы хотели, чтобы ошибка была заметной: лучше пусть она надоедает, чем будет существовать риск ее пропустить. Но во время эксплуатации вы бы предпочли, чтобы ошибка была как можно более ненавязчивой, чтобы программа могла элегантно продолжить или прекратить работу. Далее перечислены основные принципы для определения, какие инструменты защитного программирования следует оставить в релизной версии, а какие -- убрать.

Оставьте код, которые проверяет существенные ошибки. Решите, какие части программы могут содержать скрытые ошибки, а какие -- нет. Скажем, разрабатывая электронную таблицу, вы можете скрывать ошибки, касающиеся обновления экрана, так как в худшем случае это приведет к неправильному изображению. А вот в вычислительном модуле скрытых ошибок быть не должно, поскольку такие ошибки могут привести к неверным расчетам в электронной таблице. Большинство пользователей предпочтут помучиться с некорректным выводом на экран, чем с неправильным расчетом налогов и аудитом налоговой службы.

Удалите код, проверяющий незначительные ошибки. Если последствия ошибки действительно незначительны, удалите код, который ее проверяет. При этом “удалить” означает не физически убрать код, а использовать управление версиями, переключатели прекомпилятора или другую технологию для компиляции программы без этого кода.

Удалите код, приводящий к прекращению работы программы. Если на стадии разработки ваша программа обнаруживает ошибку, ее надо сделать заметнее, чтобы ее могли исправить. Часто наилучшим действием при выявлении ошибки будет печать диагностического сообщения и прекращение работы. Это полезно даже для незначительных ошибок.

Во время эксплуатации пользователям нужна возможность сохранения своей работы, прежде чем программа рухнет. И поэтому они, вероятно, будут согласны терпеть небольшие отклонения в обмен на поддержание работоспособности программы на достаточное для сохранения время. Пользователи не приветствуют ничего, что приводит к потере результатов их работы, независимо от того, насколько это помогает отладке и, в конце концов, улучшает качество продукта. Если ваша программа содержит отладочный код, способный привести к потере данных, уберите его из промышленной версии.

Регистрируйте ошибки для отдела технической поддержки. Обдумайте возможность оставить отладочные средства в промышленной версии, но изменить их поведение на более подходящее. Если вы заполнили ваш код утверждениями,

прекращающими выполнение программы на стадии разработки, на стадии эксплуатации можно не удалять их совсем, а настроить процедуру утверждения на запись сообщений в файл.

Все функции программы, которые могут дать сбой, должны иметь код ошибки, который должен отображаться в сообщении об ошибке. Благодаря таким кодам обычно удается собрать достаточно информации, которую можно передать ответственным за исправление ошибки специалистам. Документируйте коды ошибок и в сообщение включите информацию о том, где находится эта документация, чтобы помочь техподдержке.

Убедитесь, что оставленные сообщения об ошибках дружелюбны. Если вы оставляете в программе внутренние сообщения об ошибках, проверьте, что они дружественны к пользователю.

Сообщение об ошибке должно:

- идентифицировать программу, которая его выводит;
- сообщать пользователю о конкретной проблеме;
- предлагать определенные способы решения этой проблемы;
- предоставлять информацию о том, куда пользователь может обратиться за помощью;
- предоставлять дополнительную информацию человеку, оказывающему помощь пользователю;
- предоставлять уникальный идентификационный код, который поможет отличить это сообщение от других, похожих на него.

Оно не должно:

- содержать информацию, которая не поможет в решении проблемы и на которую пользователь зря потратит время;
- содержать бесполезную, избыточную, неполную или неверную информацию.

Очень важно при написании сообщений помнить о конечном пользователе; о том, что программой будут пользоваться другие люди; о том, что каждое сообщение об ошибке программист составляет не только для себя. Пользователь не обязан знать программу во всех деталях. Предоставляемая информация должна быть подробной и понятной. Сообщение не должно ставить пользователя в тупик.