**22 вопрос. Описать любой "неприятный запах" кода, и как его можно избежать.**

**Комментарии**

**C1: Неуместная информация**

В комментариях неуместно размещать информацию, которую удобнее хранить в других источниках: в системах управления исходным кодом, в системах контроля версий и в других системах протоколирования. Например, история изменений только загромождает исходные файлы длинным историческим и малоинтересным текстом. Метаданные (авторы, дата последней модификации и т. д.) в общем случае также неуместны в комментариях. Комментарии должны быть зарезервированы для технической информации о коде и его архитектуре.

**C2: Устаревший комментарий**

Комментарий, содержимое которого потеряло актуальность, считается устаревшим. Комментарии стареют довольно быстро. Не пишите комментарии, которые с течением времени устареют. Обнаружив устаревший комментарий, обновите его или избавьтесь от него как можно быстрее. Устаревшие комментарии часто «отрываются» от кода, который они когда-то описывали. Так в вашем коде появляются плавучие островки недостоверности и бесполезности.

**C3: Избыточный комментарий**

Избыточным считается комментарий, описывающий то, что и так очевидно. Например:

i++; // Увеличение переменной i

Или другой пример — комментарий Javadoc, который содержит не больше (а вернее, меньше) полезной информации, чем простая сигнатура функции:

/\*\*

\* @param sellRequest

\* @return

\* @throws ManagedComponentException

\*/

public SellResponse beginSellItem(SellRequest sellRequest) throws ManagedComponentException

Комментарии должны говорить то, что не может сказать сам код.

**C4: Плохо написанный комментарий**

Если уж вы беретесь за написание комментария, напишите его хорошо. Не жалейте времени и позаботьтесь о том, чтобы это был лучший комментарий, который вы способны создать. Тщательно выбирайте слова. Следите за правильностью орфографии и пунктуации. Не пишите сумбурно. Не объясняйте очевидное. Будьте лаконичны.

**C5: Закомментированный код**

Фрагменты закомментированного кода выводят меня из себя. Кто знает, когда был написан этот код? Кто знает, есть от него какая-нибудь польза или нет? Однако никто не удаляет закомментированный код — все считают, что он понадобится кому-то другому. Этот код только попусту занимает место, «загнивая» и утрачивая актуальность с каждым днем. В нем вызываются несуществующие функции. В нем используются переменные, имена которых давно изменились. В нем соблюдаются устаревшие конвенции. Он загрязняет модуль, в котором он содержится, и отвлекает людей, которые пытаются его читать. Закомментированный код отвратителен! Увидев закомментированный код, удалите его! Не беспокойтесь, система управления исходным кодом его не забудет. Если кому-то этот код действительно понадобится, то он сможет вернуться к предыдущей версии. Не позволяйте закомментированному коду портить вам жизнь.

**Рабочая среда**

**E1:** **Построение состоит из нескольких этапов**

Построение проекта должно быть одной тривиальной операцией. Без выборки многочисленных фрагментов из системы управления исходным кодом. Без длинных серий невразумительных команд или контекстно-зависимых сценариев для построения отдельных элементов. Без поиска дополнительных файлов в формате JAR, XML и других артефактов, необходимых для вашей системы. Сначала вы проверяете систему одной простой командой, а потом вводите другую простую команду для ее построения.

svn get mySystem

cd mySystem

ant all

**E2:** **Тестирование состоит из нескольких этапов**

Все модульные тесты должны выполняться всего одной командой. В лучшем случае все тесты запускаются одной кнопкой в IDE. В худшем случае одна простая команда вводится в командной строке. Запуск всех тестов — настолько важная и фундаментальная операция, что она должна быть быстрой, простой и очевидной.

**Функции**

**F1: Слишком много аргументов**

Функции должны иметь небольшое количество аргументов. Лучше всего, когда аргументов вообще нет; далее следуют функции с одним, двумя и тремя аргументами. Функции с четырьмя и более аргументами весьма сомнительны; старайтесь не использовать их в своих программах (см. «Аргументы функций» на с. 64).

**F2: Выходные аргументы**

Выходные аргументы противоестественны. Читатель кода ожидает, что аргументы используются для передачи входной, а не выходной информации. Если ваша функция должна изменять чье-либо состояние, пусть она изменяет состояние объекта, для которого она вызывалась (см. «Выходные аргументы», с. 70).

**F3: Флаги в аргументах**

Логические аргументы явно указывают на то, что функция выполняет более одной операции. Они сильно запутывают код. Исключите их из своих программ (см. «Аргументы-флаги», с. 66).

**F4: Мертвые функции**

Если метод ни разу не вызывается в программе, то его следует удалить. Хранить «мертвый код» расточительно. Не бойтесь удалять мертвые функции. Не забудьте, что система управления исходным кодом позволит восстановить их в случае необходимости.

**Разное**

**G1: Несколько языков в одном исходном файле**

Современные среды программирования позволяют объединять в одном исходном файле код, написанный на разных языках. Например, исходный файл на языке Java может содержать вставки XML, HTML, YAML, JavaDoc, English, JavaScript и т. д. Или, скажем, наряду с кодом HTML в файле JSP может присутствовать код Java, синтаксис библиотеки тегов, комментарии на английском языке, комментарии Javadoc, XML, JavaScript и т. д. В лучшем случае результат получается запутанным, а в худшем — неаккуратным и ненадежным. В идеале исходный файл должен содержать код на одном — и только одном! — языке. На практике без смешения языков обойтись, скорее всего, не удастся. Но по крайней мере следует свести к минимуму как количество, так и объем кода на дополнительных языках в исходных файлах.

**G2: Очевидное поведение не реализовано**

Согласно «принципу наименьшего удивления1 », любая функция или класс должны реализовать то поведение, которого от них вправе ожидать программист. Допустим, имеется функция, которая преобразует название дня недели в элемент перечисления, представляющий этот день.

Day day = DayDate.StringToDay(String dayName);

Логично ожидать, что строка "Monday" будет преобразована в Day.MONDAY. Также можно ожидать, что будут поддерживаться стандартные сокращения дней недели, а регистр символов будет игнорироваться. Если очевидное поведение не реализовано, читатели и пользователи кода перестают полагаться на свою интуицию в отношении имен функций. Они теряют доверие к автору кода и им приходится разбираться во всех подробностях реализации.

**G3: Некорректное граничное поведение**

Код должен работать правильно — вроде бы очевидное утверждение. Беда в том, что мы редко понимаем, насколько сложным бывает правильное поведение. Разработчики часто пишут функции, которые в их представлении работают, а затем доверяются своей интуиции вместо того, чтобы тщательно проверить работоспособность своего кода во всех граничных и особых ситуациях. Усердие и терпение ничем не заменить. Каждая граничная ситуация, каждый необычный и особый случай способны нарушить работу элегантного и интуитивного алгоритма. Не полагайтесь на свою интуицию. Найдите каждое граничное условие и напишите для него тест.

**G4: Отключенные средства безопасности**

Авария на Чернобыльской станции произошла из-за того, что директор завода отключил все механизмы безопасности, один за другим. Они усложняли проведение эксперимента. Результат — эксперимент так и не состоялся, а мир столкнулся с первой серьезной катастрофой в гражданской атомной энергетике. Отключать средства безопасности рискованно. Ручное управление serialVersionUID бывает необходимо, но оно всегда сопряжено с риском. Иногда отключение некоторых (или всех!) предупреждений компилятора позволяет успешно построить программу, но при этом вы рискуете бесконечными отладочными сеансами. Не отключайте сбойные тесты, обещая себе, что вы заставите их проходить позднее, — это так же неразумно, как считать кредитную карту источником бесплатных денег.

**G5: Дублирование**

Это одно из самых важных правил в книге и к нему следует относиться очень серьезно. Практически каждый автор, пишущий о проектировании программного обеспечения, упоминает это правило. Дэйв Томас (Dave Thomas) и Энди Хант (Andy Hunt) назвали его принципом DRY («Don’t Repeat Yourself», то есть «не повторяйтесь») [PRAG]. Кент Бек сделал его одним из основных принципов экстремального программирования в формулировке «Один, и только один раз». Рон Джеффрис (Ron Jeffries) ставит это правило на второе место, после требования о прохождении всех тестов. Каждый раз, когда в программе встречается повторяющийся код, он указывает на упущенную возможность для абстракции. Возможно, дубликат мог бы стать функцией или даже отдельным классом. «Сворачивая» дублирование в подобные абстракции, вы расширяете лексикон языка программирования. Другие программисты могут воспользоваться созданными вами абстрактными концепциями. Повышение уровня абстракции ускоряет программирование и снижает вероятность ошибок. Простейшая форма дублирования — куски одинакового кода. Программа выглядит так, словно у программиста дрожат руки, и он снова и снова вставляет один и тот же фрагмент. Такие дубликаты заменяются простыми методами. Менее тривиальная форма дублирования — цепочки switch/case или if/else, снова и снова встречающиеся в разных модулях и всегда проверяющие одинаковые наборы условий. Вместо них надлежит применять полиморфизм. Еще сложнее модули со сходными алгоритмами, но содержащие похожих строк кода. Однако дублирование присутствует и в этом случае. Проблема решается применением паттернов ШАБЛОННЫЙ МЕТОД или СТРАТЕГИЯ [GOF]. В сущности, большинство паттернов проектирования, появившихся за последние 15 лет, представляет собой хорошо известные способы борьбы с дублированием. Нормальные формы Кодда устраняют дублирование в схемах баз данных. Само объектно-ориентированное программирование может рассматриваться как стратегия модульной организации кода и устранения дубликатов. Естественно, это относится и к структурному программированию. Надеюсь, я достаточно четко выразил свою мысль. Ищите и устраняйте дубликаты повсюду, где это возможно.

**G6: Код на неверном уровне абстракции**

В программировании важную роль играют абстракции, отделяющие высокоуровневые общие концепции от низкоуровневых подробностей. Иногда эта задача решается созданием абстрактных классов, содержащих высокоуровневые концепции, и производных классов, в которых хранятся низкоуровневые концепции. Действуя подобным образом, необходимо позаботиться о том, чтобы разделение было полным. Все низкоуровневые концепции должны быть сосредоточены в производных классах, а все высокоуровневые концепции объединяются в базовом классе. Например, константы, переменные и вспомогательные функции, относящиеся только к конкретной реализации, исключаются из базового класса. Базовый класс не должен ничего знать о них. Правило также относится к исходным файлам, компонентам и модулям. Качественное проектирование требует, чтобы концепции разделялись на разных уровнях и размещались в разных контейнерах. Иногда такими контейнерами являются базовые и производные классы; в других случаях это могут быть исходные файлы, модули или компоненты. Но какое бы решение ни было выбрано в конкретном случае, разделение должно быть полным. Высокоуровневые и низкоуровневые концепции не должны смешиваться. Рассмотрим следующий фрагмент:

public interface Stack {

Object pop() throws EmptyException;

void push(Object o) throws FullException;

double percentFull();

class EmptyException extends Exception {}

class FullException extends Exception {}

}

Функция percentFull находится на неверном уровне абстракции. Существует много реализаций стека, в которых концепция заполнения выглядит разумно, однако другие реализации могут не знать, до какой степени заполнен стек. Следовательно, эта функция должна располагаться в производном интерфейсе — например, BoundedStack. Возможно, вы думаете, что для неограниченного стека реализация может просто вернуть 0? Проблема в том, что абсолютно неограниченного стека не существует. Вам не удастся предотвратить исключение OutOfMemoryException, проверив условие stack.percentFull() < 50.0. Если ваша реализация функции возвращает 0, то она попросту врет. Суть в том, что ложь и фикции не способны компенсировать неверного размещения абстракций. Разделение абстракций — одна из самых сложных задач, решаемых разработчиками. Если выбор сделан неверно, не надейтесь, что вам удастся найти простое обходное решение.

**G7: Базовые классы, зависящие от производных**

Самая распространенная причина для разбиения концепций на базовые и производные классы состоит в том, чтобы концепции базового класса, относящиеся к более высокому уровню, были независимы от низкоуровневых концепций производных классов. Следовательно, когда в базовом классе встречаются упоминания имен производных классов, значит, в проектировании что-то сделано не так. В общем случае базовые классы не должны ничего знать о своих производных классах. Конечно, у этого правила имеются свои исключения. Иногда количество производных классов жестко фиксировано, а в базовом классе присутствует код для выбора между производными классами. Подобная ситуация часто встречается в реализациях конечных автоматов. Однако в этом случае между базовым и производными классами существует жесткая привязка, и они всегда размещаются вместе в одном файле jar. В общем случае нам хотелось бы иметь возможность размещения производных и базовых классов в разных файлах jar. Размещение производных и базовых классов в разных файлах jar, при котором базовые файлы jar ничего не знают о содержимом производных файлов jar, позволяет организовать развертывание систем в формате дискретных, независимых компонентов. Если в такие компоненты будут внесены изменения, то они развертываются заново без необходимости повторного развертывания базовых компонентов. Такая архитектура значительно сокращает последствия от вносимых изменений и упрощает сопровождение систем в условиях реальной эксплуатации.

**G8: Слишком много информации**

Хорошо определенные модули обладают компактными интерфейсами, позволяющими сделать много минимальными средствами. Для плохо определенных модулей характерны широкие, глубокие интерфейсы, которые заставляют пользователя выполнять много разных операций для решения простых задач. Хорошо определенный интерфейс предоставляет относительно небольшое количество функций, поэтому степень логической привязки при его использовании относительно невелика. Плохо определенный интерфейс предоставляет множество функций, которые необходимо вызывать, поэтому его использование сопряжено с высокой степенью логической привязки. Хорошие разработчики умеют ограничивать интерфейсы своих классов и модулей. Чем меньше методов содержит класс, тем лучше. Чем меньше переменных известно функции, тем лучше. Чем меньше переменных экземпляров содержит класс, тем лучше. Скрывайте свои данные. Скрывайте вспомогательные функции. Скрывайте константы и временные переменные. Не создавайте классы с большим количеством методов или переменных экземпляров. Не создавайте большого количества защищенных переменных и функций в субклассах. Сосредоточьтесь на создании очень компактных, концентрированных интерфейсов. Сокращайте логические привязки за счет ограничения информации.

**G9: Мертвый код**

Мертвым кодом называется код, не выполняемый в ходе работы программы. Он содержится в теле команды if, проверяющей невозможное условие. Он содержится в секции catch для блока try, никогда не инициирующего исключения. Он содержится в маленьких вспомогательных методах, которые никогда не вызываются, или в никогда не встречающихся условиях switch/case. Мертвый код плох тем, что спустя некоторое время он начинает «плохо пахнуть». Чем древнее код, тем сильнее и резче запах. Дело в том, что мертвый код не обновляется при изменении архитектуры. Он компилируется, но не соответствует более новым конвенциям и правилам. Он был написан в то время, когда система была другой. Обнаружив мертвый код, сделайте то, что положено делать в таких случаях: достойно похороните его. Удалите его из системы.

**G10: Вертикальное разделение**

Переменные и функции должны определяться вблизи от места их использования. Локальные переменные должны объявляться непосредственно перед первым использованием и должны обладать небольшой вертикальной областью видимости. Объявление локальной переменной не должно отдаляться от места ее использования на сотню строк. Приватные функции должны определяться сразу же после первого использования. Приватные функции принадлежат области видимости всего класса, но вертикальное расстояние между вызовами и определениями все равно должно быть минимальным. Приватная функция должна обнаруживаться простым просмотром кода от места первого использования.

**G11: Непоследовательность**

Если некая операция выполняется определенным образом, то и все похожие операции должны выполняться так же. Это правило возвращает нас к «принципу наименьшего удивления». Ответственно подходите к выбору новых схем и обозначений, а если уж выбрали — продолжайте следовать им. Если в функцию включена переменная response для хранения данных HttpServletResponse, будьте последовательны и используйте такое же имя переменной в других функциях, работающих с объектами HttpServletResponse. Если метод называется processVerificationRequest, присваивайте похожие имена (например, processDeletionRequest) методам, обрабатывающим другие запросы. Последовательное соблюдение подобных схем и правил существенно упрощает чтение и модификацию кода.

**G12: Балласт**

Какой прок от конструктора по умолчанию, не имеющего реализации? Он только попусту загромождает код. Неиспользуемые переменные, невызываемые функции, бессодержательные комментарии — все это бесполезный балласт, который следует удалить. Поддерживайте чистоту в своих исходных файлах, следите за их структурой и не допускайте появления балласта.

**G13: Искусственные привязки**

То, что не зависит друг от друга, не должно объединяться искусственными привязками. Например, обобщенные перечисления не должны содержаться в более конкретных классах, потому что в этом случае информация о конкретном классе должна быть доступна в любой точке приложения, в которой используется перечисление. То же относится и к статическим функциям общего назначения, объявляемым в конкретных классах. В общем случае искусственной считается привязка между двумя модулями, не имеющая явной, непосредственной цели. Искусственная привязка возникает в результате размещения переменной, константы или функции во временно удобном, но неподходящем месте. Главные причины для появления искусственных привязок — лень и небрежность. Не жалейте времени — разберитесь, где должно располагаться объявление той или иной функции, константы или переменной. Слишком часто мы размещаем их в удобном месте «под рукой», а потом оставляем там навсегда.

**Имена**

**N1: Используйте содержательные имена**

Не торопитесь с выбором имен. Позаботьтесь о том, чтобы имена были содержательными. Помните, что смысл может изменяться в ходе развития программного продукта; почаще переосмысливайте уместность выбранных вами имен. Не рассматривайте это как дополнительный «фактор комфортности». Имена в программных продуктах на 90% определяют удобочитаемость кода. Не жалейте времени на то, чтобы выбрать их осмысленно, и поддерживайте их актуальность. Имена слишком важны, чтобы относиться к ним легкомысленно. Возьмем следующий код. Что он делает? Когда я представлю вам тот же код с нормально выбранными именами, вы моментально поймете его смысл, но в этом виде он представляет собой мешанину из символов и «волшебных чисел».

public int x() {

int q = 0;

int z = 0;

for (int kk = 0; kk < 10; kk++) {

if (l[z] == 10) {

q += 10 + (l[z + 1] + l[z + 2]);

z += 1;

} else if (l[z] + l[z + 1] == 10) {

q += 10 + l[z + 2];

z += 2;

} else {

q += l[z] + l[z + 1]; z += 2;

}

}

return q;

}

А вот как должен был выглядеть этот код. Вообще говоря, этот фрагмент чуть менее полон, чем приведенный выше. И все же вы сразу догадаетесь, что мы пытаемся сделать, и с большой вероятностью сможете написать отсутствующие функции, основываясь на своих предположениях. «Волшебные числа» перестали быть волшебными, а структура алгоритма радует своей очевидностью.

public int score() {

int score = 0;

int frame = 0;

for (int frameNumber = 0; frameNumber < 10; frameNumber++) {

if (isStrike(frame)) {

score += 10 + nextTwoBallsForStrike(frame);

frame += 1;

} else if (isSpare(frame)) {

score += 10 + nextBallForSpare(frame); frame += 2;

} else {

score += twoBallsInFrame(frame);

frame += 2;

}

}

return score;

}

Сила хорошо выбранных имен заключается в том, что они дополняют структуру кода описаниями. На основании этих описаний у читателя формируются определенные предположения по поводу того, что делают другие функции модуля. Взглянув на приведенный код, вы сможете представить себе примерную реализацию isStrike(). А при чтении метода isStrike() становится очевидно, что он делает «примерно то, что предполагалось».

private boolean isStrike(int frame) {

return rolls[frame] == 10;

}

**N2: Выбирайте имена на подходящем уровне абстракции**

Не используйте имена, передающие информацию о реализации. Имена должны отражать уровень абстракции, на котором работает класс или функция. Сделать это непросто — и снова потому, что люди слишком хорошо справляются со смешением разных уровней абстракции. При каждом просмотре кода вам с большой вероятностью попадется переменная, имя которой выбрано на слишком низком уровне. Воспользуйтесь случаем и измените его. Чтобы ваш код хорошо читался, вы должны серьезно относиться к его непрерывному совершенствованию. Возьмем следующий интерфейс Modem:

public interface Modem {

boolean dial(String phoneNumber);

boolean disconnect();

boolean send(char c);

char recv();

String getConnectedPhoneNumber();

}

На первый взгляд все хорошо — имена функций выглядят разумно. В самом деле, во многих приложениях они точно соответствуют выполняемым операциям. А если для установления связи используется не коммутируемое подключение, а какой-то другой механизм? Например, модемы могут связываться на физическом уровне (как кабельные модемы, обеспечивающие доступ к Интернету во многих домах). А может быть, связь устанавливается посредством отправки номера порта коммутатору через интерфейс USB. Разумеется, концепция телефонных номеров в таких случаях относится к неверному уровню абстракции. Более правильная стратегия выбора имен в таких сценариях может выглядеть так:

public interface Modem {

boolean connect(String connectionLocator);

boolean disconnect();

boolean send(char c);

char recv();

String getConnectedLocator();

}

Теперь имена функций никак не ассоциируются с телефонными номерами. Они могут использоваться как для подключения по телефонной линии, так и для любой другой стратегии подключения.

**N3: По возможности используйте стандартную номенклатуру**

Имена проще понять, если они основаны на существующих конвенциях или стандартных обозначениях. Например, при использовании паттерна ДЕКОРАТОР можно включить в имена декорирующих классов слово Decorator. Например, имя AutoHangupModemDecorator может быть присвоено классу, который дополняет класс Modem возможностью автоматического разрыва связи в конце сеанса. Паттерны составляют лишь одну разновидность стандартов. Например, в языке Java функции, преобразующие объекты в строковые представления, часто называются toString. Лучше следовать подобным стандартным конвенциям, чем изобретать их заново. Группы часто разрабатывают собственные стандартные системы имен для конкретного проекта. Эрик Эванс (Eric Evans) называет их всеобщим языком проекта1 . Широко используйте термины этого языка в своем коде. Чем больше вы используете имена, переопределенные специальным смыслом, относящимся к вашему конкретному проекту, тем проще читателю понять, о чем идет речь в вашем коде.

**N4: Недвусмысленные имена**

Выбирайте имена, которые максимально недвусмысленно передают назначение функции или переменной. Рассмотрим пример из FitNesse:

private String doRename() throws Exception {

if(refactorReferences)

renameReferences();

renamePage();

pathToRename.removeNameFromEnd();

pathToRename.addNameToEnd(newName);

return PathParser.render(pathToRename);

}

Имя функции получилось слишком общим и расплывчатым; оно ничего не говорит о том, что делает функция. Ситуацию усугубляет тот факт, что в функции с именем doRename находится функция renamePage! Что можно сказать о различиях между этими функциями по их именам? Ничего. Функцию было бы правильнее назвать renamePageAndOptionallyAllReferences. На первый взгляд имя кажется слишком длинным, но функция вызывается только из одной точки модуля, поэтому ее документирующая ценность перевешивает длину.

**N5: Используйте длинные имена для длинных областей видимости**

Длина имени должна соответствовать длине его области видимости. Переменным с крошечной областью видимости можно присваивать очень короткие имена, но у переменных с большей областью видимости имена должны быть длинными. Если область видимости переменной составляет всего пять строк, то переменной можно присвоить имя i или j. Возьмем следующий фрагмент из старой стандартной игры «Bowling»:

private void rollMany(int n, int pins) {

for (int i=0; i < n; i++)

g.roll(pins);

}

Смысл переменной i абсолютно очевиден. Какое-нибудь раздражающее имя вида rollCount только затемнило бы смысл этого тривиального кода. С другой стороны, смысл коротких имен переменных и функций рассеивается на длинных дистанциях. Таким образом, чем длиннее область видимости имени, тем более длинным и точным должно быть ее имя.

**N6: Избегайте кодирования**

Информация о типе или области видимости не должна кодироваться в именах. Префиксы вида m\_ или f бессмысленны в современных средах. Кроме того, информация о проекте и/или подсистеме (например, префикс vis\_ для подсистемы визуализации) также отвлекает читателя и является избыточной. Современные среды разработки позволяют получить всю необходимую информацию без уродования имен. Поддерживайте чистоту в своих именах, не загрязняйте их венгерской записью.

**N7: Имена должны описывать побочные эффекты**

Имена должны описывать все, что делает функция, переменная или класс. Не скрывайте побочные эффекты за именами. Не используйте простые глаголы для описания функции, которая делает что-то помимо этой простой операции. Для примера возьмем следующий код из TestNG:

public ObjectOutputStream getOos() throws IOException {

if (m\_oos == null) {

m\_oos = new ObjectOutputStream(m\_socket.getOutputStream());

}

return m\_oos;

}

Функция не ограничивается простым получением m\_oos; она создает объект m\_oos, если он не был создан ранее. Таким образом, эту функцию было бы правильнее назвать createOrReturnOos.

**Тесты**

**T1: Нехватка тестов**

Сколько тестов должен включать тестовый пакет? К сожалению, многие программисты руководствуются принципом «Пожалуй, этого хватит». Тестовый пакет должен тестировать все, что может сломаться. Если в системе остались условия, не проверенные тестами, или вычисления, правильность которых не подтверждена, значит, количество тестов недостаточно.

**T2: Используйте средства анализа покрытия кода**

Средства анализа покрытия сообщают о пробелах в вашей стратегии тестирования. Они упрощают поиск модулей, классов и функций с недостаточно полным тестированием. Во многих IDE используются визуальные обозначения: строки, покрытые тестами, выделяются зеленым цветом, а непокрытые — красным. Это позволяет легко и быстро обнаружить команды if или catch, тело которых не проверяется тестами.

**T3: Не пропускайте тривиальные тесты**

Тривиальные тесты пишутся легко, а их информативная ценность превышает затраты времени на их создание.

**T4: Отключенный тест как вопрос**

Иногда мы не уверены в подробностях поведения системы, потому что неясны сами требования к программе. Вопрос о требованиях можно выразить в виде теста — закомментированного или помеченного аннотацией @Ignore. Выбор зависит от того, компилируется или нет код, к которому относится неопределенность.

**T5: Тестируйте граничные условия**

Особенно тщательно тестируйте граничные условия. Программисты часто правильно реализуют основную часть алгоритма, забывая о граничных ситуациях. T6: Тщательно тестируйте код рядом с ошибками Ошибки часто собираются группами. Если вы обнаружили ошибку в функции, особенно тщательно протестируйте эту функцию. Может оказаться, что ошибка была не одна.

**T7: Закономерности сбоев часто несут полезную информацию**

Иногда анализ закономерностей в сбоях тестовых сценариев помогает выявить причины возникших проблем. Это еще один аргумент в пользу максимальной полноты тестовых сценариев. Всесторонние наборы тестовых сценариев, упорядоченные логичным образом, выявляют закономерности. Простой пример: вы заметили, что все тесты с входными данными, длина которых превышает пять символов, завершаются неудачей? Или что любой тест, который передает во втором аргументе функции отрицательное число, не проходит? Иногда простая закономерность чередования красного и зеленого в тестовом отчете заставляет нас воскликнуть «Ага!» на пути к правильному решению. Интересный пример такого рода приведен на с. 303.

**T8: Закономерности покрытия кода часто несут полезную информацию**

Анализ того, какой код выполняется или не выполняется в ходе тестирования, иногда подсказывает причины возможных сбоев в ходе тестирования.

**T9: Тесты должны работать быстро**

Медленный тест не выполняется за разумное время. Если время поджимает, из тестового пакета первыми будут удалены медленные тесты. Сделайте все необходимое для того, чтобы ваши тесты работали быстро.