2. Классификация и примеры программных ошибок

Классические примеры катастрофического ущерба

Авария ракеты-носителя Ариан 5 (4 июня 1996)

Материал из Википедии — свободной энциклопедии

Первый запуск новой ракеты-носителя Ариан 5, разработанной Европейским космическим агентством, был произведён 4 июня 1996 года. Запуск окончился неудачей — ракета разрушилась на 39-й секунде полёта из-за неверной работы бортового программного обеспечения. Этот неудачный запуск стал одной из самых дорогостоящих компьютерных ошибок в истории. В результате аварии был потерян спутник ЕКА Cluster.

Причина неудачи. В системе управления полётом новой ракеты Ариан 5 использовались фрагменты программного обеспечения ракеты Ариан 4, в частности системы инерциальной навигации. Однако при переносе этой системы для использования на новой ракете, разработчиками не были учтены все особенности. Из-за другой траектории выведения ракеты на 30-й секунде после запуска значение горизонтальной скорости превысило установленные в программе ограничения и вызвало сбой в работе компьютера. Это привело к выдаче ложной команды на отклонение сопел ускорителей, а позже и основного двигателя. В результате на 39-й секунде полёта ракета стала разрушаться под действием аэродинамических сил и самоуничтожилась.

При использовании этой программы на ракете Ариан 4 сбоя не происходило из-за отличий в характеристиках траектории полёта (для Ариан 4 такое поведение было правильным, то есть не являлось ошибкой).

Как уже было отмечено, термин **программная ошибка** будем разделять на **дефект –** код, приводящий к ошибке, как ее причину, и **сбой –** поведение программы вследствие исполнения дефекта. Поэтому часть ошибок ниже будут именоваться дефектами, т.к. они напрямую связаны с программным кодом, а часть ошибок, вызываемых отчасти внешними причинами, сохранит свое наименование.

Дефекты, связанные с особенностями языка, компилятора или системы исполнения кода

Сразу же оставим в стороне ошибки компиляции, поскольку это ошибки, связанные со **статической** структурой программного кода. Они по большому счету не являются ошибками, поскольку относятся к формальной структуре программы. Тем не менее, компилятор может просматривать **потенциально возможные варианты** исполнения кода (ветви программы) и контролировать для всех них обязательное исполнение действий, устраняющих потенциальные ошибки. Например, в Java к этому относятся:

- Синтаксический контроль источников синхронных исключений и передача полномочий на их обработку вызывающему модулю (спецификатор throws);
- Синтаксический контроль проверки инициализации локальных переменных при исполнении всех возможных ветвей и обработки всех исключений в методе.

Ошибки, возникающие во время исполнения кода (run-time) могут по-разному обнаруживаться и обрабатываться системой исполнения кода. Это зависит от «свободы и ответственности», которые предоставляются программе (и программисту) в среде исполнения, а также от принятой в языке **модели исполняемой программы** и ее особенностей. Например, в большинстве языков программирования переменные (и массивы) интерпретируются как области памяти, выход за пределы которых недопустим.

Вследствие этого выход за пределы индекса массива считается ошибкой времени исполнения. В Си/Си++ «благодаря» адресной арифметике любая переменная рассматривается как часть неограниченной области памяти, доступной программе. Поэтому сознательный, случайный или злонамеренный выход за ее переделы не является ошибкой.

Пример необнаруживаемой run-time ошибки в Си – возвращение указателя на локальную переменную, которая уничтожается в стеке при выходе из функции (метода). При этом впоследствии через указатель можно повредить произвольное содержимое стека.

```
int  *F() {
        int a=5;
        return &a;                  // Возвращается указатель на локальную переменную в стеке.
        }
```

Дефекты адресной арифметики в Си++

Адресная арифметика с Си/Си++ - инструмент работы с памятью на низком уровне. Любой указатель позволяет интерпретировать его содержимое и как адрес переменной, так и как адрес неограниченного массива (т.е. памяти) с произвольным значением индекса, содержащим элементы указуемого типа. Кроме того, возможно преобразования типа указателя (типа указуемых элементов). Все это позволяет работать с памятью на низком уровне – уровне физических адресов, вычисляемых размерностей и т.п..

Дефекты вычислений и преобразований

Операции с вещественными числами

Наиболее распространенная ошибка при работе с вещественными числами, игнорирование того факта, что вещественное число является **принципиально неточным,** т.е. имеет фиксированное количество точных разрядов и соответствующую погрешность представления. При выполнении операций в цикле эта погрешность будет накапливаться, в результате чего сравнение на равенство и известной суммой не произойдет. В следующем примере сумма из 33 значений 1/33. оказывается равной **0.9999999999999993** и цикл проскакивает через значение 1. В то же время произведение (1/33.)*33 оказывается в точности равным 1.

```
//-------------------------------------------------------------------------------------convertation.MyFloat
double dd=1/33.,vv=0;
System.out.println("dd="+dd);          // dd=0.030303030303030304
//--------- Сумма пролетает через 1
for (vv=0;vv!=1;vv+=dd){
   if (vv>1) break;
   }
System.out.println(vv);                // 1.0303030303030296
int i=0;
//---------- 33 раза по 1/33 меньше 1
for (vv=0,i=0;i<33;vv+=dd,i++);        // 0.9999999999999993
System.out.println(vv);
System.out.println(dd*33.);            // 1.0
```

Дефекты диапазонов представления данных

Иногда диапазон представления данных, соответствующий используемому типу, может оказаться недостаточным для входных данных. Например, метод **DataOutputStream.writeUTF** двоичного потока данных передает строку в формате

«счетчик длины – байты кода UTF8». Счетчик длины передается переменной типа **short,** что ограничивает длину последовательности значением 65535 (а с учетом кодировки со знаком 32867). Хотя на сам параметр метода - строку таких ограничений нет. Вызов метода для длинной строки сопровождается исключением, поэтому это можно рассматривать как ограничение метода, однако в самой программе, использующей этот метод, это уже будет дефектом, если такая же проверка не будет выполнена программой.

Дефекты явного и неявного преобразования типов данных

Явные и неявные преобразования примитивных типов в выражениях могут приводить к следующим ошибкам:

- Потеря значащих разрядов при уменьшении размерности целых;
- Заполнение 1 (знаковым разрядом) при увеличении размерности отрицательных чисел - в арифметических задачах вполне естественно, т.к. сохраняет значение переменной со знаком, а в задачах поразрядной обработки данных приводит в неправильному заполнению «лишних» разрядов.

```
byte a=(byte)0xFF;      // Значение со знаком =-1
int b=a;                // Будет FFFFFFFF
```

Про неявные преобразования программист иногда просто забывает (например, все короткие типы перед вычислением выражений в Java и Си автоматически удлиняются до стандартных), а сами ошибки преобразования никак себя не проявляют (не сопровождаются исключениями).

Дефекты структурирования и разработки кода

Дефекты этого вида связаны с процессом проектирования и кодирования программы и в большинстве своем выглядят как **опечатки,** т.е. исправляются редактированием одного выражения, условия или оператора. Тем не менее, обнаружение и локализация таких ошибок представляет собой сложную задачу.

Дефекты инициализации

Инициализация переменных, используемых в программе, отсутствует, либо производится не тем значением, либо не в том месте (что приводит к тому, что инициализация осуществляется не всегда, когда надо).

```
int i=0,j=0;                    // Неправильное место инициализации j
for(;i<n;i++){
        for(;j<n;j++){…}        // j=0 должно быть здесь for(j=0;…)
        }
int max;                        // Неправильное значение инициализации
for(max=0,i=0;i<n;i++)          // Надо max=A[0], ошибка при всех отрицательных в массиве
        if ()
```

В настоящее время многие компиляторы позволяют отслеживать наличие инициализации во всех ветвях программы, а отладчики или среда исполнения – использовать определенные значения переменных для установки их как неинициализированных.

Лишний или недостающий шаг цикла

Дефект можно назвать **«плюс-минус метр от столба».** Он обнаруживается на граничных условиях программы и связан с тем, что программист ошибается с первым, последним, предыдущим, текущим или последующим шагом цикла.

Дефект смещения на один шаг цикла

Аналогичная ситуация, когда требуемое действие делается с ошибкой на один шаг цикла или на один элемент последовательности. Иногда это зависит от различий в соглашениях по данным: например, в стеке указатель стека ссылается на последний записанный элемент (вершина стека), а количество элементов в массиве (в Си с индексацией от 0) является индексом первого свободного. В результате операции добавления в стек и в последовательной выглядят по-разному: **A[++sp]=v и A[n++]=v.**

Дефект начального или конечного шага цикла

Наиболее распространенным дефектом граничных условий является дефект первого и последнего шагов цикла. Например, практически 100% программ, использующих пары соседних элементы массива, имеют такой потенциальный дефект, поскольку первый и последний элементы не имеют соответствующей пары слева и справа.

В тексте программы должны быть условия или условные операторы, срабатывающие на первом ми последнем шаге и корректирующие поведение программы.

```
//--------------------------- Подсчет количества слов в строке
int ns=0;
for(i=0; s[i]!=0' i++){
        if (s[i]!=' ' && (i==0 || s[i-1]==' ')) ns++;          // Обнаружение начала слова ИЛИ
        if (s[i]!=' ' && s[i+1]==0 || s[i-1]==' ')) ns++;      // Обнаружение конца слова
        }
```

Дефекты размерностей данных

Дефекты, локализованные в коде, связанном с перераспределением памяти и других ресурсов, могут проявлять себя только при превышении определенной размерности входных данных.

Дефекты форматов входных данных

Программа, использующая входные данные, получает их в определенном формате. Однако по различным причинам, этот формат может быть нарушен. Причины могут быть разными: от программных до организационных. Примеры:

- Ошибка в программе, сгенерировавшей входной файл, что привело к нарушению его формата;
- Содержимое файла изменено случайно или преднамеренно;
- Предъявлен файл от устаревшей версии программы;
- Используется база данных с устаревшей структурой таблиц (от предыдущих версий);
- Программа открывает файл с другим содержимым (файл другого типа);
- Сетевое соединение закрывается раньше, чем будет передан весь файл.

В результате в программе возникают **наведенные ошибки**, которые могут приводить к зацикливанию, фатальному завершению и созданию некорректных структур данных. Последнее

От ошибок этого типа программа может и должна защищаться. Способы защиты:

- Внесение избыточности в структуры данных – контрольное суммирование данных и счетчики размерности, уникальные сигнатуры;

- Перехват исключений от наведенных ошибок и их «тонкий» анализ с целью определения причин.

Многие дефекты этого вида имеют отношение к технологическому процессу **управления конфигурациями.** При проектировании программного продукта необходимо во все виды входных данных (файлы, БД) закладывать информацию о текущей версии формата (в заголовке файла, в таблице конфигурационных параметров БД). Такие дефекты необходимо рассматривать во взаимосвязи с вопросами устойчивости и защищенности программы.

Пример. Почему перестал работать двоичный поиск

Система управления земельным кадастром использовала БД координат объектов, в котором каждому объекты был присвоен уникальный ID, генерируемый в порядке записи объектов в таблицу. Затем данные экспортировались в два двоичных файла – координатных записей переменной длины и файл, содержащий пары ID-адрес записи в первом файле. При поиске объекта по ID программа-«рисовалка» искала объект в таблице, загруженной в память, обычным двоичным поиском.

От одного из клиентов поступило сообщение, что «рисовалка» стало как-то странно случайным образом пропускать объекты при их изображении, хотя в базе они достоверно присутствовали. Не буду описывать прелести дистанционной отладки по межгороду по базе клиента (дело происходило в 90-х годах). Причина сбоя оказалась следующей. При восстановлении «сломанной» базы из архивов путем ручного копирования записей, куски таблиц были перепутаны и исходный порядок возрастания нарушился, хотя все записи были восстановлены. А поскольку компонента экспорта БД в двоичный файл открывала физическую таблицу, это привело к нарушению порядка возрастания ID-ов в двоичном файле. В результате двоичный поиск при делении пополам вел себя непредсказуемо. Для исправления дефекта понадобилось заменить обращение к физической таблице SQL-запросом с ORDER BY ID.

Дефекты форматов внутренних данных и соглашений по данным

Соглашения по данным – допустимые конфигурации элементов и их значений, которые обязанные соблюдаться всеми компонентами программного кода, работающими с ними (например, методами, работающими со структурами данных класса). Несоблюдение их в одной части кода может привести к появлению недопустимых сочетаний данных, которые при исполнении другой «правильной» части кода приведут к сбою. Ошибка может быть наведенной и мерцающей. Пример см. в гл. 4. «Дефект соглашений по данным. Двухуровневый массив ссылок»

Дефекты общего доступа и разделения данных

То, что программа хорошо организована в плане модульной структуры сущностей-классов, еще не гарантирует, что аналогичный «шоколад» будет присутствовать в структурах данных. Рассмотрим подробнее.

Связность, доступность, видимость, контекст. Данные, с которыми работает программа, представляют собой систему взаимосвязанных объектов. То, что один из объектов одного класса может работать с другим, возможно лишь при наличии «доступности» объекта второго класса для программного кода первого. Уровни доступности могут быть разными:

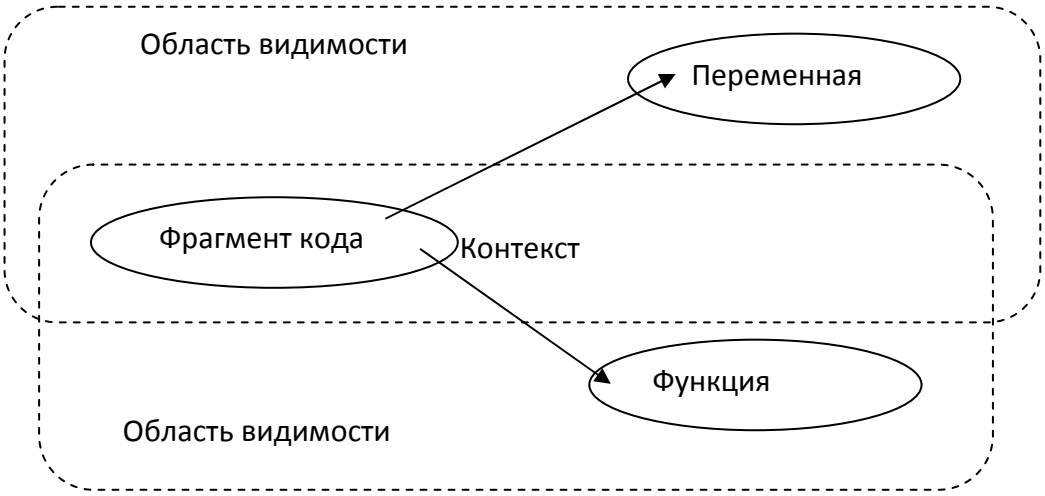
- **Прямая –** данные находятся в **контексте** исполняемого программного кода;
- **Косвенная –**по цепочке операций извлечения данных и вызова методов (точка,(),[]).

Исполняемый программный код всегда имеет неявное окружение – **контекст,** в котором ему доступны типы (классы), переменные и функции (методы) по непосредственным именам. Контекст является понятием, обратным **области видимости. Область видимости –** область программного кода, где переменная может быть использована по прямому имени.

Программный код имеет одновременно несколько контекстов (в порядке их «глобальности»):

- Глобальные данные и функции или текущее пространство имен (Си++);
- Имена импортированных классов;
- Статические методы и данные текущего класса;
- Методы и данные текущего класса (текущий объект);
- Незатененные данные и неперекрытые методы базового класса;
- Для вложенных и анонимных классов – данные и методы родительского класса;
- Для анонимных классов, создаваемых внутри методов, локальные переменные и формальные параметры с модификатором **final;**
- Формальные параметры текущего метода;
- Локальные переменные текущего метода.

Для данных некоторых контекстов имеет место **затенение –** если имя в контексте совпадает с именем в более глобальном, то оно «закрывает собой» (затеняет) второе. Это является причиной трудно обнаруживаемых ошибок, хотя синтаксически допустимо.



Программный код может использовать типы, переменные (объекты) и функции (методы), если они **достижимы из текущего контекста** при помощи таких операций над именами как [], «точка», new, ()-вызов метода. Поэтому добавление новых контекстов в программный код позволяет сделать его более компактным.

Дилемма ссылка-значение. Базовыми понятиями модульной организации программы являются способы передачи данных между **модулями по ссылке и по значению.** Языки по-разному поддерживают этот механизм. Так, в Си++ передача объекта по ссылке и по значению равноправны, в то время, как в Java основным для объектов является передача ссылки, а для значений используется явное клонирование. Каждый способ имеет свои преимущества и недостатки

Побочные эффекты и дефекты доступа по ссылке. Доступность ссылки на объект дает потенциальную возможность изменения его содержимого. Поэтому даже при хорошей модульной организации кода и его малой связности по управлению

Уровень 1 – закрытость данных. Первый уровень обеспечения модульности данных в ООП – безусловное закрытие прямого доступа к данным вне класса. Для этого они объявляются как **private** и доступ каждого из них реализуется через пару методов **get/set.** Это позволяет:

- при изменении внутреннего представления данных в классе избежать наведенных ошибок компиляции в других частях кода, работающих с ними по прямой ссылке;
- быстрее локализовать ошибку, связанную с доступом к этим данным;
- легко добавить средства ограничения доступа, сбора статистики для этих данных.

Уровень 2 – средства «раздачи» и контроля доступа в объектам. Объекты создаются и сохраняются в отдельном классе, сам класс «раздает» ссылки на эти объекты и подсчитывает их количество (имеются методы «получения» и «возвращения» ссылки). По-максимуму, класс сохраняет ссылки на «запросивших» и делает обратные вызовы при изменении данных, т.е. является контроллером, управляющим объектами.

Ошибки сборки, конфигурирования, размещения, настройки

При наличии различных версий или сборке программы в различных конфигурациях возможны ошибки, связанные с включением устаревшей версии кода, либо кода, не соответствующего конфигурации. То же самое имеет отношение к размещению программных компонент в распределенной системе, настройке их параметров.

Дефекты использования ресурсов

Внешние утечки памяти

В Си потеря ссылки (указателя) на динамические данные приводит к утечке памяти, поскольку процесс ее освобождения контролируется только программой. В Java/C# это исключено, т.к. такая ситуация контролируется сборщиком мусора.

Внутренние утечки памяти

Неиспользуемые объекты, на которые сохранены ссылки в структурах данных, являются внутренним мусором.

Зависшие ресурсы

Ресурс, который зарезервирован программой, но не освобожден (открытый файл, сетевое соединение), остается за программой в течение некоторого времени (до ее завершения, либо до операции «сбора мусора» над соответствующим объектом).

Ошибки, связанные с ограничениями по ресурсам

Любой ресурс, запрашиваемый программой, имеет физические ограничения, поэтому возможны сбои, связанные с его непредоставлением. Также большой объем используемого ресурса может приводить не к явным сбоям, а к деградации производительности (например, при работе с виртуальной памятью, превышающей физически доступный объем).

Ошибки восстановления и инициализации

Ошибки инициализации уже упоминались как ошибки в реализации отдельного модуля (метода, функции). На более высоком уровне (интегральное тестирование) эти ошибки могут давать эффект **«повторного запуска»,** когда при первом вызове модуль не срабатывает, но инициализирует структуры данных для корректного срабатывания последующих вызовов.

Сюда же относятся ошибки, связанные с сохнренением нужных данных и восстановлением программы после сбоев, например, при пропадании сетевых соединений и т.п..

Ошибки и дефекты реактивности и производительности

Ошибки проявляют себя как неприемлемые задержки при работе программы. Они могут возникать проявляться как торможение или подвисание программы при обработке больших массивов данных, а также как медленная реакция на интерактивные действия пользователей.

Зависание GUI

При синхронном выполнении продолжительных операций происходит блокирование интерфейса пользователя. Устраняется исполнением продолжительного действия в отдельном (фоновом) потоке. При этом возникает ряд проблем:

- Разделение общих данных: проще всего организационно разграничить доступные данные для основного и фонового, например, временно ограничив функционал GUI;
- Обычно действия с элементами GUI корректно выполняются только в специальном потоке (по умолчанию для методов GUI). При завершении фонового потока, возникновении ошибок или сообщений требуется взаимодействие с GUI, для этого нужно «вернуть» завершающий код в основной поток. Для этих целей в классах GUI имеются статические методы. Они также обеспечивают синхронизацию программного кода завершения с кодом основного потока.

Влияние фактора трудоемкости

При увеличении размерности обрабатываемых данных степень замедления программы зависит от трудоемкости алгоритма, которая, в свою очередь, оценивается как функция определенного вида – линейная, квадратичная и т.п.. От этого зависит масштабируемость программы по входным данным.

Чувствительность алгоритма к данным

На определенных сочетаниях данных производительность программы может деградировать. Это определяется свойством **чувствительности алгоритма к данным.** Иногда этот дает положительный эффект, но чаще – отрицательный.

Несоответствие размерностей данных

Программа должна адаптироваться под размерности данных, которые она обрабатывает. Например, перераспределение динамического массива следует производить в геометрической прогрессии, а размеры буферов – увеличивать пропорционально размеру файла. Иначе возникает эффект «вычерпывания бочки чайной ложкой».

Технологические ошибки – лишние циклы и повторные вычисления

Каждый лишний цикл дает еще одну дополнительную степень трудоемкости. Если этот цикл касается основной размерности входных данных, то это качественно снижает характеристики производительности. Типичный пример на Си:

```
char s[…]; for(int i=0; i<strlen(s);i++)…s[]…
```

функция определения длины строки использует аналогичный цикл просмотра строки до символа-ограничителя, при повторном вызове в основном цикле дает квадратичную трудоемкость.

«Повторные вычисления» – т.е. повторение некоторой части программы при тех же исходных данных, также могут приводить к снижению производительности в размерах, определяемых частотой этих повторений. Основным методическим приемом для исключения этого эффекта является **динамическое программирование –** создание кэша результатов решенных подзадач с ключевыми параметрами – исходными данными. В этом случае при обнаружении в кэш решения с требуемыми данными оно просто подставляется на нужном шаге (см.cprog- 7.7. Эффективность алгоритмов). В простейшем случае мы имеем итерационный цикл с вычислением результата текущего шага через предыдущие.

Методические ошибки реализации алгоритмов

Многие задачи могут быть решены как при помощи «тупого», так и при помощи «умного» алгоритма. Так, например, любая поисковая задача может быть решена при помощи примитивного линейного или комбинаторного перебора. Если размерность задачи позволяет решить ее в приемлемое время, то «тупой» алгоритм даже предпочтительней из-за простоты отладки и меньшего числа потенциальных ошибок. Тем не менее, решение задачи «в лоб» с определенного момента может перестать удовлетворять по производительности.

Незнание или отсутствие информации о внутренних процессах в среде исполнения

Программист использует программные решения, не имея информации или не удосужившись изучить таковую о механизмах реализации тех сервисом и средств, которыми он пользуется. Типичным примером является различие реализации классов **String и StringBuffer** в Java, которое проявляется при работе с длинными строками. Класс **String** представляет собой объект, содержащий неизменяемую строку, при любой операции над строками создаются новые объекты, а старые уходят «в мусор». Класс **StringBuffer** использует внутреннее перераспределение динамического массива в геометрической прогрессии (классический способ работы с динамическим массивом).Таким образом, при посимвольном чтении из файла в случае конкатенации символа с накопленной строкой в классе **String** мы будем иметь фактически квадратичную трудоемкость, связанную с созданием нового объекта и копированием в него старого содержимого, а при работе со **StringBuffer** трудоемкость будет практически линейной (см. **BugExamples - me.romanow.bug.performance.StringPerformance).**

Особенности среды исполнения, программного и аппаратного окружения

Вопросы производительности могут завести далеко «вглубь» программно-аппаратной среды исполнения. Типичным примером является зависимость производительности при обращении к кэш-памяти от распределения обращений по пространству адресов (линейное, случайное, локализованное). Для любой программы, работающей в ОС, имеется, как минимум, два уровня кэширования – подкачка страниц

виртуальной памяти в физическую (физическая память по существу играет роль кэша для виртуальной памяти) и процессорный кэш оперативной (физической) памяти.

Так, например, если с двумерным массивом работать по строкам и по столбцам, то производительность может реально отличаться на порядок. (**http://habrahabr.ru/post/211747/).** Воспроизведение этой ситуации на Java и на Си, а также использование одномерного массива как модели двумерного приводит к одинаковому качественному результату при количественных вариациях (проект **BugExamples - me.romanow.bug.performance.AnomalArray**), что исключает влияние программной платформы и среды исполнения (аналогичные результаты получены в Android). Очевидно, что при работе с массивом по строкам данные из кэш читаются полностью из каждого загруженного блока, а при работе по столбцам – по нескольку значений из одного блока с переходом на следующий).

Дефекты параллелизма и синхронизации

Сложность обнаружения дефектов параллелизма в том, что они приводят к **мерцающим и наведенным** ошибкам. Некорректная синхронизация данных может привести к нарушению целостности данных (соглашений по данным) в связи с непредвиденными последовательностями их изменений, невозможных в при последовательном исполнении методов. Некорректная синхронизация потоков – к зависанию (блокированию) некоторых из них.

Дефекты распределенных систем и протоколов

Дефекты в распределенных системах и поддерживающих их протоколах связаны с тем, что в них присутствует физический параллелизм и разделение данных (см. выше).

Ошибки пользовательского интерфейса

Проектирвоание графического интерфейса и Useability – отдельная тема, в которой имеются собственные критерии корректности работы компонент.