

МЕТОДИЧЕСКИЙ ПАКЕТ №2

МДК.01.02 «Поддержка и тестирование программных модулей»

Виды ошибок. Методы отладки

Специальность 09.02.07 «Информационные системы и программирование»

Объём по рабочей программе: 4 академических часа

2026

Паспорт занятия

Цель занятия — научить студентов различать человеческую ошибку, дефект и отказ, воспроизводить проблему, локализовать её источник и применять базовые инструменты отладчика для поиска технической причины неправильного поведения программы.

Параметр	Содержание
Тема по рабочей программе	Виды ошибок. Методы отладки
Объём	4 академических часа (две пары по 90 минут)
Формат	Теоретическое объяснение + разбор примеров + практический фрагмент + контрольные вопросы
Профессиональные компетенции	ПК 1.3 — выполнять отладку программных модулей; ПК 1.4 — выполнять тестирование программных модулей.

Основа пакета. Материал рассчитан как самостоятельный пакет: его можно использовать независимо от второго занятия. Терминология согласована с рабочей программой МДК.01.02 и актуальным ISTQB CTFL v4.0.1.

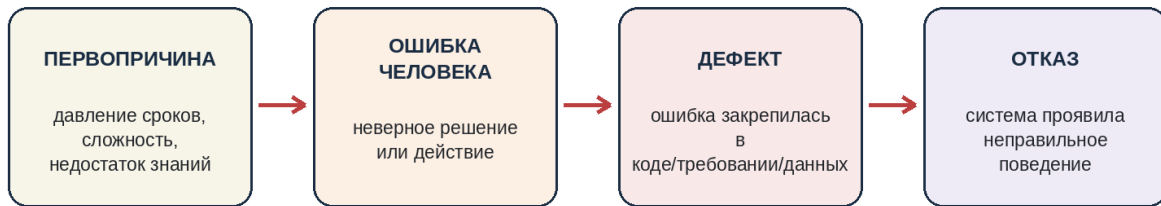
ЗАНЯТИЕ 2. Виды ошибок. Методы отладки

Главная задача занятия — научить студента не «чинить наугад», а мыслить как инженер: воспроизвести проблему, отделить симптом от причины, собрать факты, локализовать дефект, исправить минимально необходимое и проверить последствия изменения.

Пара	Время	Содержание
1-я пара	0–20 мин	Цепочка: первопричина → ошибка человека → дефект → отказ.
	20–50 мин	Практическая классификация программных ошибок.
	50–75 мин	Подход к диагностике: симптом, воспроизведение, минимальный пример.
	75–90 мин	Разбор коротких примеров и классификация.
2-я пара	0–30 мин	Методы отладки: логирование, debugger, breakpoints, пошаговое выполнение.
	30–55 мин	Watch, Locals, Call Stack, исключения, условные точки останова.
	55–80 мин	Демонстрация отладки дефектного C#-модуля.
	80–90 мин	Алгоритм действий и контрольные вопросы.

2.1. Ошибка, дефект и отказ — это разные вещи

Как человеческая ошибка превращается в отказ системы



Важно: дефект может существовать и никогда не проявиться как отказ, если соответствующий участок кода не будет выполнен или нужны редкие условия.

Ошибка (error) — ошибочное действие или решение человека. Например, разработчик неправильно понял формулу процента.

Дефект (defect, bug, fault) — конкретная проблема, появившаяся в рабочем продукте: коде, требовании, скрипте, настройке, документации. Например, вместо деления на 100 в коде вычитается число процентов.

Отказ (failure) — наблюдаемое неправильное поведение работающей системы. Например, при скидке 20% программа показывает 980 вместо 800.

Первопричина (root cause) — более фундаментальная причина, из-за которой ошибка вообще возникла: спешка, неоднозначное требование, отсутствие review, сложность архитектуры, недостаток знаний, слабая автоматизация проверок.

Ключевой тезис. Не каждый дефект немедленно приводит к отказу. Неверная ветка кода может годами существовать, но не выполняться на реальных данных. Поэтому «у нас никогда не падало» не доказывает отсутствия дефекта.

2.2. Практическая классификация ошибок

Существует много классификаций. На первом занятии полезнее дать не академически максимальный список, а категории, которые студент действительно сможет узнавать в коде.

Категория	Суть	Пример
Синтаксическая / компиляции	Код нарушает правила языка или не может быть корректно собран.	Пропущена скобка, неверный тип аргумента, неизвестное имя.
Времени выполнения	Программа собирается, но в процессе выполнения возникает исключение или аварийное состояние.	DivisionByZeroException, NullReferenceException, выход за границы массива.

Логическая	Программа выполняется, но алгоритм даёт неправильный результат.	Скидка 20% вычисляется как price - 20.
Граничная	Ошибка проявляется на нуле, минимуме, максимуме, пустом значении, ровно на границе условия.	age > 18 вместо age >= 18.
Работы с данными	Неверный формат, преобразование, округление, кодировка, состояние данных.	Денежное значение хранится в float и накапливает погрешность.
Интеграционная	Компоненты по отдельности работают, но неверно взаимодействуют.	Клиент ожидает поле userId, API отдаёт id.
Конфигурационная / окружения	Поведение зависит от настроек, версии, прав, путей, переменных окружения.	На компьютере разработчика файл есть, на сервере путь другой.
Производительности / ресурсов	Функционально результат правильный, но система работает неприемлемо медленно или расходует ресурсы.	Запрос выполняется 40 секунд; соединения с БД не освобождаются.
Конкурентности	Проблема зависит от порядка выполнения потоков или параллельных операций.	Два потока одновременно изменяют общий счётчик и теряют обновление.

2.3. Быстрые примеры для классификации

```
// Пример А
int[] values = { 10, 20, 30 };
Console.WriteLine(values[3]);
```

Это ошибка времени выполнения: индекс 3 выходит за диапазон массива из трёх элементов (допустимы 0, 1, 2).

```
// Пример В
bool IsAdult(int age)
{
    return age > 18;
}
```

Если по требованию взрослый — с 18 лет включительно, это логическая граничная ошибка. Код не падает, но возраст 18 обрабатывается неверно.

```
// Пример С
var path = "C:\app\config.json";
var json = File.ReadAllText(path);
```

На машине разработчика код может работать. На другом компьютере файла по такому пути нет — проблема окружения/конфигурации.

2.4. Симптом — не причина

Одна из самых важных привычек отладки — не исправлять первый подозрительный симптом. Если интерфейс показывает «0 товаров», это ещё не означает, что ошибка в интерфейсе. Причина может быть в запросе к БД, фильтре, API, десериализации ответа или правах доступа.

Полезная формула: «Что я наблюдаю?» отдельно от «Почему это происходит?»

Наблюдаемый симптом	Возможные причины
Кнопка ничего не делает	Не привязан обработчик; обработчик падает; запрос блокируется; состояние кнопки неверно.
В списке нет данных	БД вернула 0 строк; фильтр удалил элементы; API вернул ошибку; JSON не распарсился; UI не обновился.
Программа «зависла»	Бесконечный цикл; ожидание блокировки; синхронный долгий запрос; deadlock; очень тяжёлая операция.
Неверная сумма	Неправильная формула; неверный тип; округление; повторное применение операции; неверные входные данные.

2.5. Отладка начинается с воспроизведения

Практический цикл отладки



Отладка не начинается с «правки наугад». Сначала добейтесь воспроизводимости и соберите факты: входные данные, ожидаемый и фактический результат, стек вызовов, значения переменных и журналы.

Если проблему нельзя воспроизвести, её трудно исследовать. Поэтому первый вопрос не «где ошибка?», а «при каких условиях она стабильно возникает?»

Хорошее описание воспроизведения содержит:

- версию программы и окружение;

- точные начальные условия;
- входные данные;
- последовательность действий;
- ожидаемый результат;
- фактический результат;
- при необходимости — лог, stack trace, скриншот или запись сетевого запроса.

Практический приём. Чем меньше и стабильнее сценарий воспроизведения, тем легче отладка. Если дефект проявляется в большом приложении, полезно сузить его до минимального набора данных, модуля или функции.

2.6. Методы отладки: от простого к инструментальному

1) Анализ сообщений компилятора и исключений

Прежде чем ставить десятки breakpoint, прочитайте сообщение об ошибке. Тип исключения, сообщение и stack trace часто сразу указывают класс проблемы и место, откуда началось неправильное выполнение.

2) Логирование и диагностический вывод

Логи полезны, когда ошибка возникает не в интерактивной IDE, а на сервере, у пользователя, в долгом процессе или только в определённой последовательности событий. Печатать нужно не «мы здесь», а полезный контекст: идентификатор операции, входные параметры, ключевые состояния, длительность, ошибку и stack trace.

```
logger.LogInformation(
    "Calculate order {OrderId}: items={Count}, total={Total}",
    order.Id, order.Items.Count, order.Total);
```

Безопасность. Не логируйте пароли, токены доступа, полные банковские реквизиты и другие секреты. Диагностика не должна создавать новую уязвимость.

3) Пошаговая отладка в debugger

Отладчик позволяет остановить программу в выбранном месте и изучить её состояние. В Visual Studio базовые инструменты — breakpoint, Step Into, Step Over, Step Out, окна Locals/Autos/Watch и Call Stack.

Инструмент	Что делает	Когда применять
Breakpoint	Останавливает выполнение на выбранной точке.	Перед подозрительной веткой или операций.
Step Into (F11)	Выполняет следующий оператор и входит внутрь вызываемого метода.	Когда предполагается, что дефект внутри метода.
Step Over (F10)	Выполняет текущий оператор, но не заходит внутрь вызванного метода.	Когда метод не интересует или уже проверен.
Step Out (Shift+F11)	Продолжает выполнение до выхода из текущего метода.	Когда текущая функция уже исследована.

Watch	Позволяет постоянно наблюдать выбранные переменные и выражения.	Для отслеживания изменения ключевого состояния.
Locals / Autos	Показывает локальные и текущие значения.	Чтобы быстро увидеть состояние в точке останова.
Call Stack	Показывает цепочку вызовов, приведших к текущей точке.	Чтобы понять путь выполнения и источник вызова.
Conditional breakpoint	Останавливает программу только при выполнении условия.	Когда ошибка появляется, например, только при <code>id == 742</code> .

2.7. Отладка как проверка гипотез

Сильный разработчик не просто двигается по строкам. Он формулирует гипотезу: «Значение `total` становится отрицательным после применения скидки». Затем выбирает наблюдение, которое может подтвердить или опровергнуть гипотезу: breakpoint перед и после расчёта, Watch для `total`, просмотр входного `percent`.

1. Сформулируйте наблюдаемый факт: «ожидалось 800, получено 980».
2. Предложите вероятную область: расчёт скидки.
3. Сформулируйте проверяемую гипотезу: процент вычитается как абсолютное число.
4. Выберите наблюдение: посмотреть `price`, `percent` и результат выражения.
5. Подтвердите или отбросьте гипотезу.
6. После фикса создайте или сохраните тест, который не даст дефекту вернуться.

2.8. Сужение области поиска

Когда код большой, полезно искать границу между «ещё правильно» и «уже неправильно». Это напоминает двоичный поиск: вместо просмотра сотен строк по очереди проверяем состояние примерно в середине цепочки обработки и быстро сокращаем область.

Пример: данные проходят путь БД → репозиторий → сервис → API → клиент → UI. Если UI показывает неверное имя, проверяем ответ API. Если в API уже неверно — UI исключаем из поиска. Если API правильно, ищем на клиентской стороне. Каждая проверка делит пространство причин.

2.9. Assertions и автоматические тесты как инструменты диагностики

Assertion фиксирует предположение разработчика о состоянии программы. Если предположение нарушено, программа или тест останавливается ближе к месту возникновения проблемы, а не намного позже.

```
Debug.Assert(percent >= 0 && percent <= 100);
```

Unit-тест ещё полезнее: после обнаружения дефекта можно сначала написать тест, который воспроизводит проблему, увидеть его падение, исправить код и убедиться, что тест проходит. Такой тест становится защитой от регрессии.

2.10. Демонстрация отладки в Visual Studio

Исходный код для демонстрации:

```
using System;

class Program
{
    static decimal CalculateTotal(decimal price, int count, int discountPercent)
    {
        decimal subtotal = price * count;
        decimal discount = discountPercent / 100 * subtotal;
        return subtotal - discount;
    }

    static void Main()
    {
        decimal total = CalculateTotal(500m, 2, 20);
        Console.WriteLine($"Итого: {total}"); // ожидаем 800
    }
}
```

Проблема. discountPercent — int. Выражение discountPercent / 100 выполняет целочисленное деление: 20 / 100 даёт 0. Поэтому discount становится 0, а итог — 1000.

Пошаговый сценарий демонстрации

1. Запустить программу обычным способом и зафиксировать фактический результат: 1000 вместо ожидаемых 800.
2. Поставить breakpoint на строке decimal subtotal = price * count; и запустить Debug.
3. Посмотреть входные значения price=500, count=2, discountPercent=20.
4. Нажать F10 и убедиться, что subtotal=1000 — пока всё правильно.
5. Нажать F10 на строке расчёта discount и увидеть discount=0. Область дефекта локализована.
6. Добавить выражение discountPercent / 100 в Watch. Получить 0 и объяснить целочисленное деление.
7. Исправить выражение, например: decimal discount = discountPercent / 100m * subtotal;
8. Снова выполнить проверку: ожидаемый и фактический результат 800 совпадают.
9. Проверить соседние случаи: 0%, 100%, возможно отрицательный процент и процент >100. Решить, нужна ли валидация входа.

Что проговаривать вслух. Смысл демонстрации — не показать горячие клавиши как самоцель. Покажите причинную цепочку: тест обнаружил неправильный результат → debugger помог найти место, где состояние впервые стало неправильным → изменение исправило дефект → повторный тест подтвердил результат.

2.11. Плохие стратегии отладки и чем их заменить

Плохая привычка	Почему мешает	Чем заменить
-----------------	---------------	--------------

Менять несколько мест кода одновременно	Непонятно, какое изменение реально помогло; легко внести новый дефект.	Минимальные изменения и проверка после каждого логического шага.
Добавлять random print без плана	Получается поток шума без ответа на вопрос.	Сначала гипотеза, затем диагностический вывод нужных значений.
Игнорировать текст исключения	Теряется самая полезная диагностическая информация.	Читать тип, сообщение, внутреннее исключение и stack trace.
Считать, что проблема «точно в библиотеке»	Раннее предположение закрывает альтернативные причины.	Проверять границу: что передали библиотеке и что получили.
После фикса проверить только один сценарий	Исправление может сломать соседнюю функциональность.	Confirmation testing + небольшая регрессия.
Удалить симптом, не разобравшись в причине	Проблема возвращается в другой форме.	Понять механизм дефекта и при возможности устранить первопричину процесса.

2.12. Практический фрагмент: «найди точку, где данные испортились»

Предложите студентам код и попросите до запуска определить, где поставить breakpoints и какие значения добавить в Watch.

```
static int Average(int[] values)
{
    int sum = 0;
    for (int i = 0; i <= values.Length; i++)
    {
        sum += values[i];
    }
    return sum / values.Length;
}

Console.WriteLine(Average(new[] { 10, 20, 30 }));
```

Ожидаемые наблюдения:

- ошибка проявится при $i == 3$;
- условие цикла допускает $i == \text{values.Length}$;
- последний допустимый индекс — $\text{values.Length} - 1$;
- исправление: $i < \text{values.Length}$;
- после фикса нужно проверить пустой массив: возникнет деление на ноль, то есть обнаружится другой граничный случай.

Связь двух занятий. Это хороший пример того, что один исправленный дефект не делает функцию «полностью правильной». Отладка отвечает на конкретную проблему, а тестирование помогает системно искать следующие риски.

2.13. Как зафиксировать найденный дефект

Даже на вводном занятии полезно показать минимальную структуру сообщения о дефекте. Она дисциплинирует мышление и облегчает воспроизведение.

Поле	Пример
Заголовок	Скидка 20% не применяется при расчёте общей суммы.
Окружение	Windows 11, сборка 1.0.3, .NET 8.
Предусловие	Создан товар стоимостью 500 руб.
Шаги	1) Добавить 2 товара. 2) Установить скидку 20%. 3) Выполнить расчёт.
Ожидаемый результат	800 руб.
Фактический результат	1000 руб.
Дополнительные данные	discountPercent=20; discount=0 в debugger; приложен stack trace/лог при необходимости.

2.14. Контрольные вопросы по второму занятию

Чем дефект отличается от отказа?

Ответ: Дефект — проблема в рабочем продукте, например в коде. Отказ — наблюдаемое неправильное поведение работающей системы, которое может быть вызвано дефектом.

Почему первое действие при сложной ошибке — воспроизведение?

Ответ: Стабильный сценарий позволяет измерять состояние, сравнивать результаты и проверять гипотезы; без него непонятно, помогло ли изменение.

Когда лучше Step Into, а когда Step Over?

Ответ: Step Into используют, когда нужно исследовать внутреннюю работу вызываемого метода; Step Over — когда метод не представляет интереса и нужно продолжить на следующей строке текущего метода.

Что показывает Call Stack?

Ответ: Цепочку вызовов методов, которая привела к текущей точке выполнения.

Зачем нужен conditional breakpoint?

Ответ: Чтобы остановиться только при определённом условии и не проходить вручную множество одинаковых итераций.

Почему после исправления нужен регрессионный тест?

Ответ: Изменение может устранить исходный дефект, но вызвать проблемы в других частях системы.

Почему «поставить print везде» — слабая стратегия?

Ответ: Без гипотезы возникает много шума. Диагностические данные должны отвечать на конкретный вопрос о состоянии и пути выполнения.

Как связаны root cause analysis и профилактика дефектов?

Ответ: Если устранить первопричину — например, неоднозначные требования или отсутствие review — можно снизить вероятность целого класса похожих дефектов в будущем.

2.15. Готовый сценарий объяснения на доске

Можно завершить тему коротким алгоритмом, который студенты переписывают как памятку:

1. Зафиксируй, что должно было произойти и что произошло на самом деле.
2. Добейся воспроизводимости.
3. Собери контекст: входы, окружение, исключение, логи, стек вызовов.
4. Сузь область поиска: где данные ещё правильные, а где уже неправильные?
5. Сформулируй гипотезу о причине.
6. Проверь гипотезу debugger-ом, логами, тестом или минимальным примером.
7. Исправь минимально необходимое.
8. Подтверди исправление тем же сценарием.
9. Запусти соседние проверки, чтобы поймать регрессию.
10. Если проблема системная — подумай о первопричине и профилактике.

Итог занятия. Если студент запомнит этот алгоритм и перестанет «тыкать код до тех пор, пока заработает», занятие выполнило практическую цель.

Источники и нормативная основа

1. Рабочая программа профессионального модуля ПМ.01 «Разработка модулей программного обеспечения для компьютерных систем», АНО ПОО ММКЦТ «Академия ТОП», 2023. МДК.01.02, тема 56.
2. ISTQB. Certified Tester Foundation Level Syllabus v4.0.1, 15.09.2024. Используются понятия error, defect, failure, root cause, confirmation testing и regression testing.
3. Microsoft Learn. Visual Studio Debugger Documentation. Используются сведения о breakpoints, пошаговом выполнении, Watch/Locals и Call Stack.

Материал адаптирован для занятия СПО и не является официальным учебным курсом ISTQB или Microsoft.