### Предмет МДК.05.02 Разработка кода информационных систем

### Группа. 2Б

### Дата 02.05.2025

### Тема: Отладка приложений. Организация обработки исключений.

### Задание: сделать конспект лекционного материала

**Отладка приложений. Организация обработки исключений.**

*Корректность* и *устойчивость* - два основных качества программной системы, без которых все остальные ее достоинства не имеют особого смысла. Понятие *корректности* программной системы имеет смысл только тогда, когда задана ее *спецификация*. В зависимости от того, как формализуется *спецификация*, уточняется понятие *корректности*.

***Корректность*** - это способность программной системы работать в строгом соответствии со своей *спецификацией*. ***Отладка*** - процесс, направленный на достижение *корректности*.

Во время работы системы могут возникать ситуации, выходящие за пределы, предусмотренные *спецификацией*. Такие **ситуации** называются *исключительными*. ***Устойчивость*** - это способность программной системы должным образом реагировать на *исключительные ситуации*. *Обработка* ***исключительных ситуаций*** - процесс, направленный на достижение *устойчивости*.

Почему так трудно создавать корректные и *устойчивые* программные системы? Все дело в сложности разрабатываемых систем. Когда в 60-х годах прошлого века фирмой IBM создавалась операционная система OS-360, то на ее создание потребовалось 5000 человеко-лет, и проект по сложности сравнивался с проектом высадки первого человека на Луну. Сложность нынешних сетевых операционных систем, систем управления хранилищами данных, прикладных систем программирования на порядки превосходит сложность OS-360, так что, несмотря на прогресс, достигнутый в области технологии программирования, проблемы, стоящие перед разработчиками, не стали проще.

#### Жизненный цикл программной системы

Под "*жизненным циклом*" понимается период от замысла программного продукта до его

"кончины". Обычно рассматриваются следующие фазы этого процесса: Проектирование - Разработка - Развертывание - Сопровождение

Все это называется циклом, поскольку после каждой фазы возможен возврат к предыдущим этапам. В объектной технологии этот процесс является бесшовным, все этапы которого тесно переплетены. Не следует рассматривать его как однонаправленный - от проектирования к сопровождению. Чаще всего, ситуация обратная: уже существующая реализация системы, прошедшая сопровождение, и существующие библиотеки компонентов оказывают решающее влияние на то, какой будет новая система, каковы будут ее *спецификации*.

Вот некоторые типовые правила, характерные для процесса разработки ПО:

* Уделяйте этапу проектирования самое пристальное внимание. Успех дела во многом определяется первым этапом. Нет смысла торопиться с переходом на последующие этапы, пока не составлены ясные и четкие *спецификации*. Ошибки этого этапа - самые дорогие и трудно исправляемые.
* Помните о тех, для кого разрабатывается программный продукт. Идите "в люди", чтобы понять, что нужно делать. Вместе с тем, не следует полностью полагаться на пользователей - их опыт консервативен, новые идеи могут часто приходить от разработчиков, а не от пользователей.
* Разработка не начинается "с нуля". Только используя уже готовые компоненты, можно своевременно создать новую систему. Работая над проектом, думайте о будущем, Не создавайте компоненты, допускающие их *повторное использование* в других проектах.
  + - Создавайте как можно раньше прототип свой системы и передавайте его пользователям в опытную эксплуатацию. Это поможет устранить множество недостатков и ошибок в заключительной версии программного продукта.
    - Какие бы хорошие *спецификации* не были написаны, какими бы хорошими технологиями и инструментами не пользовались разработчики, какими бы профессионалами они ни были - этого еще не достаточно для успеха дела. Необходимым условием является управление проектом, наличие специальных средств управления. Но и этого не достаточно. Третьим важным фактором является существование команды. Коллектив разработчиков должен представлять собой единый коллектив. Умение работать в команде так же важно, как и профессиональные навыки разработчика.

#### Три закона программотехники

##### Первый закон (закон для разработчика)

*Корректность* **системы - недостижима. Каждая последняя найденная ошибка является предпоследней.**

Этот *закон* отражает сложность нетривиальных систем. Разработчик всегда должен быть готов к тому, что в работающей системе имеются ситуации, в которых система работает не в точном соответствии со своей *спецификацией*, так что от него может требоваться очередное изменение либо системы, либо ее *спецификации*.

##### Второй закон (закон для пользователя)

**Не бывает некорректных систем. Каждая появляющаяся ошибка при эксплуатации системы - это следствие незнания** *спецификации* **системы.**

Есть два объяснения справедливости второго *закона*. Несерьезное объяснение состоит в том, что любая система, что бы она ни делала, при любом постусловии *корректна* по отношению к предусловию False, поскольку невозможно подобрать ни один набор входных данных, удовлетворяющих этому предусловию. Так что все системы *корректны*, если задать False в качестве их предусловия. Если вам пришлось столкнуться с системой, предусловие которой близко к False, то лучшее, что можно сделать, это отложить ее в сторону и найти другую.

Более поучительна реальная ситуация, подтверждающая второй *закон* и рассказанная мне в былые годы Виталием Кауфманом - специалистом по тестированию трансляторов. В одной серьезной организации была разработана серьезная прикладная система, имеющая для них большое значение. К сожалению, при ее эксплуатации сплошь и рядом возникали ошибки, из-за которых организация вынуждена была отказаться от использования системы. Разработчики обратились к нему за помощью. Он, исследуя систему, не внес в нее ни строчки кода. Единственное, что он сделал, это описал точную *спецификацию* системы, благодаря чему стала возможной нормальная эксплуатация.

Обратите внимание на философию, характерную для этих законов: при возникновении ошибки разработчик и пользователь должны винить себя, а не кивать друг на друга. Так что часто встречающиеся фразы "Ох уж эта фирма Чейтософт - вечно у них ошибки!" характеризует, мягко говоря, непрофессионализм говорящего.

**Третий закон (закон чечако)**

**Если** *спецификацию* **можно нарушить, - она будет нарушена. Новичок (чечако)**

**способен "подвесить" любую систему.**

Неквалифицированный пользователь в любом контексте всегда способен выбрать наименее подходящее действие, явно не удовлетворяющее *спецификации*, которая ориентирована на "разумное" поведение пользователей. Полезным практическим следствием этого *закона*

является привлечение к этапу тестирования системы неквалифицированного пользователя

* "человека с улицы".

### Отладка

Что должно делать для создания *корректного* и *устойчивого* программного продукта? Как минимум, необходимо:

* + создать *надежный код*, *корректность* которого предусматривается с самого начала;
  + отладить этот код;
  + предусмотреть в нем *обработку исключительных ситуаций*.

#### Создание надежного кода

Большинство вопросов, затрагиваемых в этой лекции, в том числе и проблемы создания *надежного кода*, заслуживают отдельного и глубокого рассмотрения. К сожалению, придется ограничиться лишь высказыванием ряда тезисов.

Для повышения надежности нужно уменьшить сложность системы, и главное в этом процессе - это *повторное использование*. В идеале большая часть системы должна быть собрана из уже готовых компонентов. Объектная технология проектирования вносит свой вклад в повышение надежности кода. **Наследование и универсализация** позволяют, не изменяя уже существующие классы, создать новые классы, новые типы данных, придающие проектируемой системе новые свойства при минимальных добавлениях нового кода. *Статический контроль типов* позволяет выявить многие ошибки еще на этапе компиляции. *Динамическое связывание* и *полиморфизм* позволяют автоматически включать объекты классов-потомков в уже существующие схемы работы - методы родителя могут вызывать методы потомков, ничего не зная о появлении этих новых потомков.

*Автоматическая сборка мусора* позволяет снять с разработчика обязанности управления освобождением памяти и предотвратить появление крайне неприятных и опасных ошибок, связанных с некорректным удалением объектов.

Крайне важную роль в создании *надежного кода* играют *спецификации* **методов** класса, класса в целом, системы классов. *Спецификации* являются частью документации, встроенной в проект, и вообще важной его частью. Их существование облегчает не только создание *корректного* кода, соответствующего *спецификации*, но и создание системы тестов, проверяющих *корректность* кода. Нужно сказать, что существуют специальные инструментальные средства, поддерживающие автоматическое создание тестов на основе *спецификаций*. Незаменима роль *спецификаций* на этапе сопровождения и *повторного использования* **компонентов**. Невозможно повторно использовать компонент, если у него нет ясной и полной *спецификации*.

#### Искусство отладки

Нужно стараться создавать *надежный код*. Но без *отладки* пока обойтись невозможно. Роль

*тестеров* в современном процессе разработки ПО велика.

*Отладка* - это некоторый детективный процесс. Программа, в которую внесены изменения, подозревается в том, что она работает некорректно. Презумпция невиновности здесь не применима. Если удается предъявить тест, на котором программа дает неверный результат, то доказано, что подозрения верны. Втайне мы всегда надеемся, что программа работает правильно. Но цель тестирования другая - попытаться опровергнуть это предположение.

*Отладка* может доказать некорректность программы, но она не может доказать ее правильность. *Отладка* не гарантирует *корректности программы*, даже если все тесты пройдены успешно. Искусное тестирование создает высокую степень уверенности в *корректности программы*.

Часть ошибок программы ловится автоматически еще на этапе компиляции. Сюда относятся все синтаксические ошибки, ошибки несоответствия типов и некоторые другие. Это простые ошибки и их исправление, как правило, не вызывает трудностей. В *отладке* нуждается синтаксически *корректная программа*, результаты вычислений которой получены, но не соответствуют требуемым *спецификациям*. Чаще всего еще не отлаженная программа на одних исходных данных работает правильно, на других - дает ошибочный результат. Искусство *отладки* состоит в том, чтобы обнаружить все ситуации, в которых работа программы приводит к ошибочным вычислениям.

Как и во всякой детективной деятельности, в ходе *отладки* необходим сбор улик, для чего применяется *две группы средств*. Первая позволяет контролировать ход вычислительного процесса: порядок следования операторов в методах, порядок вызова самих методов, условия окончания циклов, правильность переходов. Вторая отслеживает изменение состояния вычислительного процесса (значения свойств объектов) в процессе выполнения.

Есть и другая классификация. Средства, используемые при *отладке*, можно разделить на **инструментарий**, предоставляемый средой разработки Visual Studio .Net, и *программные средства*, предоставляемые языком и специальными классами библиотеки FCL. Начнем рассмотрение с программных средств.

#### Отладочная печать и условная компиляция

Одним из основных средств *отладки* является *отладочная печать*, позволяющая получить данные о ходе и состоянии процесса вычислений. Обычно разрабатываются специальные отладочные методы, вызываемые в критических точках программы - на входе и выходе программных модулей, на входе и выходе циклов и так далее. Искусство *отладки* в том и состоит, чтобы получить нужную информацию о прячущихся ошибках, проявляющихся, возможно, только в редких ситуациях.

Хотелось бы иметь легкий механизм управления отладочными методами, позволяющий включать при необходимости те или иные инструменты. Для этого можно воспользоваться *механизмом условной компиляции*, встроенным в язык C#. Этот механизм состоит из двух частей. К проекту, точнее, к конфигурации проекта можно добавить специальные *константы условной компиляции*. Вызов отладочного метода может быть сделан условным. Если соответствующая *константа компиляции* определена, то происходит компиляция вызова метода и он будет вызываться при выполнении проекта. Если же константа не определена (выключена), то вызов метода даже не будет компилироваться и никаких динамических проверок - вызывать метод или нет - делаться не будет.

Как задавать константы компиляции? Напомню, что проекты в Visual Studio существуют в нескольких конфигурациях. В ходе работы с проектом можно легко переключаться с одной конфигурации на другую, после чего она становится активной, можно изменять настройки конфигурации, можно создать собственные конфигурации проекта. По умолчанию проект создается в двух конфигурациях - Debug и Release, первая из которых предназначена для *отладки*, вторая - для окончательных вычислений. Первая не предполагает оптимизации и в ней определены две *константы условной компиляции* - DEBUG и TRACE, во второй - определена только константа TRACE. Отладочная версия может содержать вызовы, зависящие от константы DEBUG, которые будут отсутствовать в финальной версии.

Используя страницу свойств, к конфигурации проекта можно добавлять новые константы  
 компиляции.