1. ООП на C#
2. Проектирование архитектуры приложений
3. Принципы S.O.L.I.D.
4. Паттерны проектирования
5. UI Framework = WPF
6. Углубленные аспекты C# (reflections, dynamic compilation, dynamic libraries, modern languages features.)

База данных(SQL) теория, проектирование таблиц, язык SQL.

Вопросы проектирования объектно–ориентированной архитектуры.

ООП - есть проблемы с соответствием ожидания заказчика в условиях меняющихся требований.

Стандарты проектирования П.О.

IDEF0, IDEF1x, DFD, UML, …

(Графические нотаций для описания различных этапов проектирования)

В первую очередь 100% проектируют – Базу данных – графическая нотация называется ER – диаграммы (entity - relation)

------------------------------------------------------------------------------------

Проблемы в/д с заказчиком ПО

1. Заказчик часто не знает, что желает

(какую задачу решает?) – решается выявлением бизнес процессов (IDEF0)

1. Заказчик часто не является техническим специалистом – требуется общий язык. (Язык находят и как правило он графический - диаграммы) DFD, ER
2. Регулярно и быстро меняющиеся требования заказчика. (Переделка – ЛОМАТЬ - существующего кода) – решение этой проблемы:

А) **Гибкие** методологии разработки (Scrum, agile, …) – Вопросы организации труда.

Б) **Гибкая** разработка кода: solid + паттерны ООП.

В) Автоматическая тестирование кода (Unit test) TDD – Test Driven Development.

Г) Системы использования систем контроля версий (git)

C#. Интерфейсы (новый тип данных).

Interface.

Опр. – абстрактный тип данных предназначенный для описания функциональности объектов.

// Интерфейс – это важнейший тип для соблюдения принципов solid. (Гибкой разработки)

Для получения хорошего представления об интерфейсах (используем сравнение).

Abstract class versus interface

Общие черты

1) Очень похожий синтаксис (запрещено создавать объект - экземпляры)

2) И те и другие участвуют в наследовании, и не просто – а созданы именно для наследования!

3) Наличие абстрактных методов

4) Наличие публичных методов

5) Контракт – обязательность исполнения.

Отличия

1. Синтаксис похожий, но разные ключевые слова:

abstract class <=> interface

+ **Самое заметное отличие – В интерфейсах Запрещены ПОЛЯ.**

2) Класс может наследовать другой 1 класс и/или интерфейс, или даже несколько интерфейсов (множественное наследование интерфейса), а вот интерфейс не может наследовать класс, но может другие интерфейсы в любом кол-ве.

3) В интерфейсах все методы 100% абстрактные (др быть не может) а в абстрактном классе могут быть обычные методы и абстрактные. => в интерфейсах слово abstract использовать нельзя.

4) В интерфейсах все методы 100% публичные (других быть не может), а в классе публичные и не публичные. => в интерфейсах писать public нельзя

5) Абстр. Классы – это “оч. жесткий контракт” в программировании абстрактные классы играю роль “семечко” – хранилище ДНК. Из класса вырастает дерево с определенными узлами.

Интерфейс – “гибкий контракт” описывают “ЧИСТУЮ ФУНКЦИОНАЛЬНОСТЬ” без данных (без ДНК). Благодаря интерфейсам можно “скрещивать” классы, принадлежащие разным деревьям.

Итого:

“Интерфейсы” – это “рафинированные” абстрактные классы

Правило наименования интерфейсов (работают почти во всех ЯП)

1. Всегда начинается с I
2. Название должно быть действием/глаголом с суффиксом able(**ну или несколько существит выраж функцию**)

Д З класс Student перечисление ТипСортировкиСтудента: ПоРосту, Повесу, ПоФИо, Поуспеваемость

Нужные поля

+ статическое свойство – ТипСортировкиСтудента

Создать список студентов

Выполнить разные сортировки (с выводом на экран)

С указанием типа сортировки(через статическое свойство).

Написать свитч и сделать в зависимости от сортировки

Принципы S.O.L.I.D

Нацелены на “Гибкую разработку” (легко и **безболезненно модифицируемый** код). “ООП”

1. S = SRP = Single Responsibility Principle = Принцип единственной ответственности => “у вас должна быть только одна причина – единственная вносить изменения в класс”: каждый класс должен нести ответственность за 1 задачу (в крайнем случае несколько тесно-связанных). [не должно быть класса типа “человек-оркестр”]
2. O = OCP = Open-Closed Principle = “Принцип открытости-закрытости” => Класс должен быть открыт для расширения функционала, но закрыт для модификации. (Добавлять можно, переписывать НЕЛЬЗЯ) [в C# идеально подходят “методы расширения **Extensions**”, частично согласуется с Инкапсуляцией]
3. L = LSP = Liskov’s Substitution Principle = “Принцип подстановки Лисков(ой)”(Барбара Лисков) => класс нужно проектировать с учетом наследования таким образом чтобы в системе всегда можно было заменить предка его потомком (и ничего не должно сломаться).
4. I = ISP = interface Segregation Principle = “Принцип разделения интерфейса” => в интерфейсе не должно быть много методов (обычно рекомендуется 1-5 ) если получается больше – скорее всего можно разделить на несколько других интерфейсов.

“МФУ” IMFU {Print(); Scan(); Fax(); Email(); Copy();}

IPrintable, IScanable и т.д.

1. D = DIP = Dependency Inversion Principle = “Принцип инверсии зависимостей” => Классы верхнего уровня не должны зависеть от классов нижнего, и классы нижнего ТОЖЕ не должны зависеть от верхнего: и те и другие Должны зависеть только от **АБСТРАКЦИИ**.

Принцип – последний по списку, но не по значению!!!

[**Почти все** паттерны ООП следуют этому принципу!]

**DIP**

Что это за уровни с классами?

Чисто технически в любом приложении можно выделить минимум ДВА, а обычно ТРИ уровня (слоя)

1. Нижний уровень (слой ”Детали”)
2. Средний уровень (слой “Механизмы”/ “Механики”)
3. Верхний/высший уровень (слой “Бизнес-логика” / “Игровая логика”)

(Эти слои получаются сами-собой) как строительство здании (фундамент – стены - крыша)

Обычно разработка программ происходит (сама-собой) от нижнего слоя к верхнему (от деталей к бизнес-логике) => такой подход к разработке и проектированию =>

Проектирование/разработка “снизу-вверх”.

1. Нижний слой – детали – низкоуровневые функции (работы с жестким диском, оперативной памятью, сетевыми протоколами, с видеокартой, с операционной системой, …)
2. Механики – прослойка основных функций программы.
3. Бизнес-логика – (игровые правила, бизнес-процессы)

При таком подходе возникает “жесткая зависимость” классов из верхних уровней с классами нижними.

Структура “Карточный домик”

Для игр есть проблема “**Портирование**” – когда у приложения нужно поменять самый нижний слой

С этой проблемой (портироваться) ДАВНО пытаются бороться!!!

В играх – движки (Unity, unreal engine)

В бизнесе – спец. Языки(+платформой) Java, C#

Java – JVM (Java virtual Machine)

C# - .NET Framework => с 2010 => .NET Core

Особое место занимает JavaScript – работает в браузере – а браузеры по сути “нижний слой”

ECMA5/6 JS + class

Есть другой путь решения проблемы проектирования – гибкая архитектура за счет SOLID => DIP

При таком подходе DIP = проектирование и разработка происходит сверху-вниз(т.е. переворачивается с ног на голову) красиво и легко решаются проблемы портирования.

Шаги

1. Выделяется абстракция (чистое описание возможностей без конкретики) для верхнего уровня (на C# самая лучшая абстракция - **Интерфейс**)
2. На основе этой абстракций пишется верхний слой
3. На основе этой абстракции (или родственной) строится нижний слой.

Требуется немного больше времени на разработку и доп. Время на первоначальное проектирование.

C# .NET micro (Framework)

C# .Net Nano (Framework)

Первый Паттерн (шаблон - рецепт) ОО программирования - “инъекция зависимости”

**Dependency injection**

Если в классе есть гибкая зависимость – то в реальной работе Объекту этого класса нужно… иметь возможность “Подкинуть” реального представителя этой зависимости. (Для кнопки нужна реальная лампа!!!)

Три способа “подкинуть” реальный объект в зависимости

1. Через конструктора класса
2. Через публичный метод класса
3. Через пропертью

(в некоторых случаях можно организовать автоматическую инъекцию)

Специально для таких полей (с инъекцией) в c# добавили специальный модификатор

**Readonly**

Обозначает что данному полю значение можно задать ТОЛЬКО в конструкторе.

S.O.L.I.D. => следовать, +паттерн.

На самом деле принципов Больше.

+KISS = keep it simple, stupid

+DRY = don’t repeat yourself

* Сборник рецептов – как следовать этим принципам.

“Паттерны” ОО программирование.

Не законы – это рекомендации / рецепты.

Много = на три группы:

1. Порождающие паттерны (creational)
2. Структурные (structural)
3. Поведенческие (behavior)
4. Одиночка. / Singleton
5. Группа? Порождающие
6. Зачем? Используется в случаях когда необходимо обеспечить 100% гарантию, что какой то объект класса будет существовать в памяти работающей программы в единств. Экземпляре.

= new Object();

1. Многие разработчики называют – **антипаттерн !!!**
2. **Почти в каждой 1 есть.**
3. **Архитектура / устройство:**
4. Публичный класс в котором создаем **закрытое статическо**е поле (тип – обычно совпадает с классом)

Private static

Б)Закрытый констр класса private

В) Публичный статический метод/ пропертя возвр объект (поле из а).

Известный кейс – Logger

Спец объект осуществляющий журналирование(логи). В логах встречаются записи об исключениях (Exception), ключевые этапы выполнения кода (действий пользователя).

Создание и использование библиотек классов (ClassLibrary)

Появились в 1980-х – 1990-х в windows.

1) \*.dll = DLL = Dynamic Linked Library

2) Ранее существовали (до сих пор исп.) библиотеки статической компоновки – весь код библиотеки целиком включается в тело программы которая его использует! (приводит к увеличению объема программ) во многих программах оказывается куча “одинакового кода”.

Dll – код библиотеки хранится автономно и когда требуется какой-то программе загружается автоматически в оперативную память – несколько программ может использовать библиотеку из памяти! Одновременно! (экономия на жестком диске и оперативной памяти)

directX - \*.dll direct3d.dll directaudio.dll ddraw.dll

C++ = компиляция = в машинный двоичный код = dll имеют специальный формат придуманный Microsoft (точка хода, заголовок с информацией: точка входа, список функций, …)

Dll-Hell Windows 95 неустойчива.

Win 2000 = WinXp

Dll доработали => COM (Component object Model)

В несколько раз сложнее чем простые dll, но зато безопаснее.

.NET Framework = компиляция не в машинный язык в ПРОМЕЖУТОЧНЫЙ байт-код.

Dll – C# очень сильно отличаются от стандартных Windows (по сути общее только название \*.dll)

Обычные программы на C++ и программы на C# могут использовать библиотеки классич. И .NET через механизм “Interop” .Net Framework.

Dll библиотеки C# более безопасные и простые в создание чем COM библиотеки Windows (например, им не нужен реестр – уникальный ключ сразу хранится внутри библиотеки).

В C# есть 3 способа подключения и использования библиотек.

Динамические библиотеки.

COM – объекты (C++) =>

Продвинутые dll важнейшая цель - предоставить доступ к функционалу стороннему ПО.

Используем автоматизацию пакета Microsoft Office. => Генерация отчетов (Документов).

1. Обозреватель решений – ссылки добавить => Раздел использует COM => MS WordObjectLibrary.

(COM объекты НИКОГДА не копируются в проект!!!)

Использование “Отражений”

Reflections.

В .NET при компиляции каждого Типа

В генерируемый код авто добавляется дополнительная информация – метаданные – они объектно-ориентированные (представляют собой ОБЪЕКТ с методами и свойствами).

Это объект – метаданные – “отражение”: хранит в себе “внутреннее описание” того объекта, к которому привязан.

В режиме Runtime всегда можно получить полную информацию об устройстве объекта любого класса через его “Отражение”.

Когда применяется:

1. Прилетают “неизвестные данные” извне
2. Автоматизация обработки множества свойств для множества объектов (рефлексия поможет сильно сократить объем кода и устранить дублирование кода)
3. Необходимость использовать “обходные пути” – приватный метод.
4. Загрузка неизвестных библиотек Dll в режиме Runtime.
5. Динамическая (runtime) генерация нового КОДА.

Паттерн “Итератор”

(в C# - “перечислитель” - Enumerator)

[на этом паттерне построена работа всех функции Linkq]

1. Относится к группе: “Поведенческих”
2. Зачем? Цель? – Позволяет организовать последовательный ПЕРЕБОР всех элементов коллекции ЕДИНООБРАЗНО в независимости от внутреннего устройства самой коллекции.

* В Программировании ЕСТЬ МНОЖЕСТВО СОВЕРШЕННО различных коллекции(простые списки, связанные списки, очереди, стеки, бинарные деревья, графы,…) совершенно различное внутренне устройство. Разные методы и способы обхода элементов.

У дерева = .GetRoot(), .GetChildren(), .NextChild(),…

Очень часто для выполнения простого действия – ПЕРЕБОР всех элементов – приходится использовать кучу разных Функции.

[В C# этот паттерн является частью синтаксиса ЯЗЫКА – цикл foreach работает на основе этого паттерна, язык Linlq – также работает на его основе]

1. Устройство:
2. Первый IEnumerable – один метод . GetEnumerator()
3. Первый интерфейс – IEnumerator (enumetate + or) – Я перечислитель (циклом **foreach**)

В этом интерфейсе есть 3 элемента: .MoveNext(), .Reset(), .Current

1. Класс коллекция с интерфейсом из (1), + класс перечислитель с интерфейсом из (2)
2. Цикл foreach() – использует оба этих интерфейса

Коллекция – связанный список (односвязанный, двусвязанный, кольцо) – Достоинство: Очень быстрая(дешевые) операции вставки и удаления элемента из середины.

В C# есть готовая реализация – LinkedList<T> методы все такие же как в обычном List<T> потому что общий интерфейс IList, но внутри все совершенно иначе

В C# 2.0 был добавлен новый оператор yield

Употребляется совместно с оператором break или return и служит для супер-компактной записи Паттерна Итератор.

(Синтаксический сахар над большим объемом кода).

C# 3.0 Features (Linq – killer-feature) – 2007 год

* +9 новых особенностей и большинство из них 6/7 предназначены для реализации нового языка Linq.

Язык - встроен в язык C#

Linq = language integrated query – интегрированный (в C#) язык запросов (к любым структурированным перечислимым ДАННЫМ)

Благодаря Linq C# приобрел черты Декларативных языков.

1. Implicitly typed local variables – неявно типизированные локальные переменные

var

1. Var

Тип переменной задается не слева от имени (как привыкли), а тип определяется автоматически из факта присвоения исходя из их типа данных результата выражения СПРАВА.

ЭТОТ тип определяется до КОМПИЛЯЦИИ

1. Automatic properties – автоматические свойства

- для удобства работы с Linq.

-Разрешается не прописывать полностью геттер и сеттер для Проперти + можно не прописывать Приватное поле для проперти

- авто создаст КОМПИЛЯТОР.

1. Object initializer – инициализатор объекта – позволяет задавать значение пропертей сразу при создании объекта без конструктора.