Лекция 3.

Архитектура и оформление консольного PHP-приложения

Поставленная задача

1. Нужно написать консольную игру на PHP, данные будут храниться в БД SQLite.
2. В дальнейшем приложение придется перевести в другую среду выполнения (веб-браузер) и переписать на языке программирования JavaScript. Данные будут храниться в базе данных IndexedDB

Нужно спроектировать архитектуру приложения для упрощения выполнения этих задач.

Требования к архитектуру приложения

1. **Бизнес-логика игры** была максимально независима от:
    - среды выполнения (консоль vs браузер),
    - языка программирования (PHP vs JS),
    - способа хранения данных (SQLite vs IndexedDB / localStorage / API).
2. **Интерфейс (ввод/вывод)** и **доступ к данным** были вынесены в отдельные слои.

Требования к архитектуру приложения

🎯 **Цель архитектуры:** "Ядро игры" + "Адаптеры"

- **Ядро (Core / Domain)** — чистая логика: правила, состояние, ходы, победа/поражение.
- **Адаптеры** — обёртки под конкретную среду:
  - `ConsoleView` / `WebView`
  - `SQLiteRepository` / `LocalStorageRepository`
  - `PHPRunner` / `JSRunner`

Такой подход называется **Hexagonal Architecture** (Ports and Adapters) или **Clean Architecture**.

Создание приложения

Нужно написать консольную игру на PHP, данные будут храниться в БД SQLite.
В дальнейшем приложение придется перевести в другую среду выполнения (веб-браузер) и переписать на языке программирования JavaScript.

1. **Настройка рабочей среды**

2. **Программирование задачи**

Создание приложения

Нужно написать консольную игру на PHP, данные будут храниться в БД SQLite.
В дальнейшем приложение нужно перевести на другую среду выполнения (веб-браузер) и ЯП (JavaScript).

1. **Настройка рабочей среды**
   - Настройка Git и GitHub, создание репозитория
   - Установка СУБД SQLite
   - Установка PHP CLI
   - Установка менеджера зависимостей Composer
   - Публикация игры на Packagist
   - Выбор IDE или редактора
   - Настройка линтера кода (*PHP Code Sniffer*)

2. **Программирование задачи**

Создание приложения

Нужно написать консольную игру на PHP, данные будут храниться в БД SQLite.
В дальнейшем приложение нужно перевести на другую среду выполнения (веб-браузер) и ЯП (JavaScript).

1. **Настройка рабочей среды**

2. **Программирование задачи**
   - Разработка алгоритма
   - Выбор технологии и языка программирования
   - Проектирование архитектуры приложения
   - Кодирование и оформление кода
   - Тестирование

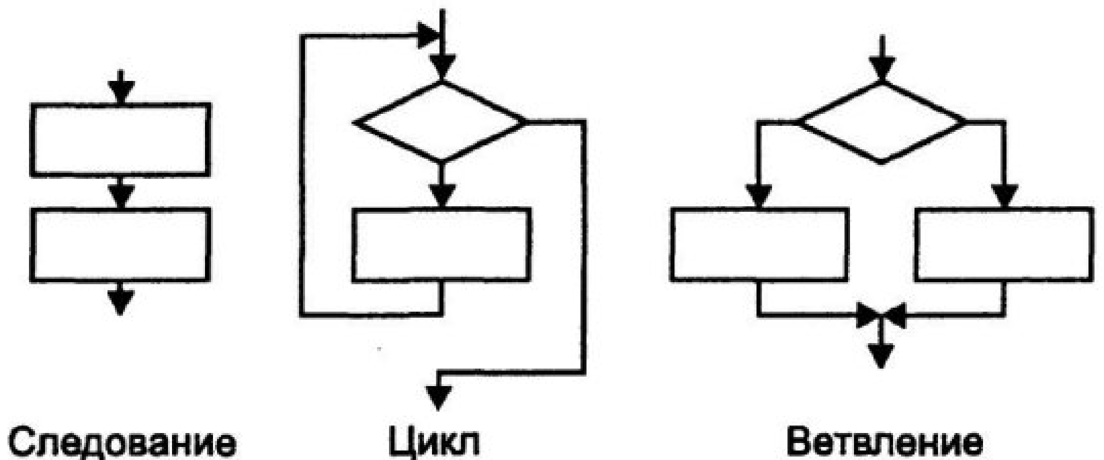
Программирование задачи

- Разработка алгоритма
- **Выбор технологии и языка программирования**
- Проектирование архитектуры приложения
- Кодирование и оформление кода
- Тестирование

Выбор технологии программирования

Будем писать приложение в императивном процедурном стиле, придерживаясь структурного подхода.

- Проектирование сверху вниз
- Структурное кодирование
- Модульное программирование



Базовые конструкции структурного программирования

Структурный подход

Проектирование сверху вниз

- Сначала проектируется общая структура программы.
- Затем каждая часть детализируется до уровня конкретных операторов.
- Кодирование следует этой иерархии — сначала основной модуль, затем вспомогательные функции.

Структурный подход - базовые структуры управления

Базовые структуры управления

- **Последовательность** – операторы, перечисленные в определенном порядке. Ее выполнение состоит в выполнении каждого из этих операторов в том же порядке.
- **Цикл**, который содержит последовательность операторов, выполняемую многократно.
- **Выбор**, состоящий из условия и двух последовательностей операторов.

Дополнительные структуры управления

- **Оператор перехода (goto).** Любая программа, использующая его, имеет эквивалент, выраженный в терминах стандартных структур управления.

- **Обработка исключений**, обеспечивающая способы восстановления после возникновения событий, прерывающих нормальный поток управления (переполнение, void-вызовы и т.п.) . Try … Except …

Последовательность

- Главный принцип организации последовательного кода — упорядочение зависимостей.
- Зависимости должны быть сделаны явными с помощью хороших имен методов, списков параметров, комментариев и вспомогательных переменных.
- Если порядковые зависимости в коде отсутствуют, старайтесь размещать взаимосвязанные выражения как можно ближе друг к другу.

Программа должна быть написана так, чтобы ее можно было читать сверху вниз, а не перескакивая с места на место.

Последовательность

Неудачный код (C++)

```
MarketingData marketingData;
SalesData salesData;
TravelData travelData;

travelData.ComputeQuarterly();
salesData.ComputeQuarterly();
marketingData.ComputeQuarterly();

salesData.ComputeAnnual();
marketingData.ComputeAnnual();
travelData.ComputeAnnual();

salesData.Print();
travelData.Print();
marketingData.Print();
```

Как рассчитывается marketingData?

Последовательность

Хороший последовательный код (C++)

```
MarketingData marketingData;
marketingData.ComputeQuarterly();
marketingData.ComputeAnnual();
marketingData.Print();

SalesData salesData;
salesData.ComputeQuarterly();
salesData.ComputeAnnual();
salesData.Print();

TravelData travelData;
travelData.ComputeQuarterly();
travelData.ComputeAnnual();
travelData.Print();
```

Упоминания каждой переменной располагаются вместе — они «локализованы».

Код можно разбить на отдельные методы для данных по маркетингу, продажам и поездкам.

Выбор

- Для последовательностей `if … then …else` и операторов `case` выбирайте порядок, позволяющий улучшить читабельность
- Избегайте вложенных проверок и сложных выражений в условиях (лучше определить для них функции)

Циклы

- Циклы сложны для понимания, поэтому они должны быть максимально простыми. Следует избегать экзотических видов циклов, минимизировать вложенность, делать очевидными входы и выходы цикла.

- Называйте индексы цикла понятно и используйте только с одной целью.

Структурный подход – модульность кода

Структурный подход

2. **Модульность кода:**

   - Программа разбивается на **логически независимые модули** — функции, процедуры, подпрограммы.
   - Каждый модуль решает **одну конкретную задачу** (принцип единственной ответственности).
   - Модули имеют **чётко определённый интерфейс** (входные/выходные данные).

Подпрограммы

Почему нужно писать подпрограммы и когда это нужно делать?

- **Снижение сложности за счет формирования понятной промежуточной абстракции.** Выделение фрагмента кода в удачно названный метод — один из лучших способов документирования его цели.

Подпрограммы

Почему нужно писать подпрограммы и когда это нужно делать?

- **Снижение сложности за счет формирования понятной промежуточной абстракции.** Выделение фрагмента кода в удачно названный метод — один из лучших способов документирования его цели.

```
Assign(f, "my.ini");
Open(f);
While (not eof(f)) do
  Begin
    Readln(f, s);
    If substr(s, 1,5) = "user" then UserName:= substr(s,
6,len(s)-5);
  End;
```

```
UserName := GetUserNameFromIniFile;
```

Подпрограммы

Почему нужно писать подпрограммы и когда это нужно делать?

- Снижение сложности за счет формирования понятной промежуточной абстракции.
- **Предотвращение дублирования кода.** Проще изменять код, выше надежность (меньше вероятность ошибиться).

Подпрограммы

Почему нужно писать подпрограммы и когда это нужно делать?

- Снижение сложности за счет формирования понятной промежуточной абстракции.
- Предотвращение дублирования кода.
- **Упрощение переноса приложения на другие платформы.** Выделение и изоляция непортируемого кода (нестандартные возможности языка, зависимости от оборудования и операционной системы и т. д.).

Подпрограммы

Почему нужно писать подпрограммы и когда это нужно делать?

- Снижение сложности за счет формирования понятной промежуточной абстракции.
- Предотвращение дублирования кода.
- Упрощение переноса приложения на другие платформы.
- **Упрощение сложных логических проверок.** Сокрытие деталей проверок + описательное имя метода позволяет лучше охарактеризовать суть проверки.

Аргументы функций

Количество аргументов нужно минимизировать.

Идеальный (самый выразительный) случай – аргументов нет.
Проще понимать и тестировать.

Следует избегать использования аргументов-флагов.

Разделение команд и запросов

Функция должна что-то делать или отвечать на какой-то вопрос, но не одновременно.

Либо функция изменяет состояние объекта, либо возвращает информацию об этом объекте.

```
function Set(attribute: string, value: string): boolean;
…
if Set("username", "popov-av") …   // ???
```

Разделение команд и запросов

Функция должна что-то делать или отвечать на какой-то вопрос, но не одновременно.

Либо функция изменяет состояние объекта, либо возвращает информацию об этом объекте.

```
function Set(attribute: string, value: string): boolean;
…
if Set("username", "popov-av") …   // ???
```

Лучше:

```
if attributeExists("username") then setAttribute("username",
"popov-av");
```

Модульность кода - специфика PHP

- Настройка автозагрузки модулей с помощью Composer

Структурный подход – кодирование и оформление кода

Структурный подход

3. **Читаемость и самодокументируемость:**

- Использование осмысленных имён переменных и функций.
- Чёткая структура отступов и форматирования.
- Минимизация побочных эффектов и глобальных переменных.

Проектирование архитектуры приложения

Общий подход

- Разделение ответственности между данными, логикой и визуальным представлением
- Разделение кода на модули, содержащие описания функций, и скрипты с исполняемым кодом
- Создание файловой структуры приложения

Проектирование архитектуры приложения

Общий подход

- Разделение ответственности между данными, логикой и визуальным представлением
- Разделение кода на модули, содержащие описания функций, и скрипты с исполняемым кодом
- Создание файловой структуры приложения

Специфика PHP

- Настройка автозагрузки модулей с помощью Composer

Программирование задачи

- Разработка алгоритма
- Выбор технологии и языка программирования
- **Проектирование архитектуры приложения**
- Кодирование и оформление кода
- Тестирование

Проектирование архитектуры приложения

- **Разделение ответственности между данными, логикой и визуальным представлением**
- Выделение модулей и подпрограмм
- Разделение кода на модули, содержащие описания функций, и скрипты с исполняемым кодом
- Создание файловой структуры приложения

Архитектурный паттерн MVC

1. Model (Модель)

- Хранит данные и бизнес-логику.
- Не знает ничего о том, как данные отображаются или как пользователь взаимодействует с ними.

2. View (Представление)

- Отвечает за **вывод в консоль** (`echo` , `print`) и **чтение ввода** (`fgets(STDIN)` или `readline()`).
- Не содержит логики — только форматирование и общение с пользователем.

3. Controller (Контроллер)

- Координирует взаимодействие между Model и View.
- Принимает решения: что делать при выборе пользователя, какие методы модели вызывать и что показывать через View.

MVC в консольных PHP-приложениях

- **Чёткая структура**: легко понять, где что находится.
- **Легко тестировать**: модель можно тестировать без ввода/вывода.
- **Гибкость**: если позже вы захотите сделать веб-интерфейс — модель останется той же, нужно будет только заменить View и, возможно, адаптировать Controller.
- **Поддержка CLI**: PHP отлично работает в консоли, особенно в современных версиях.

Проектирование архитектуры приложения

- Разделение ответственности между данными, логикой и визуальным представлением
- **Выделение модулей и подпрограмм**
- Разделение кода на модули, содержащие описания функций, и скрипты с исполняемым кодом
- Создание файловой структуры приложения

Проектирование архитектуры приложения

- Разделение ответственности между данными, логикой и визуальным представлением
- Выделение модулей и подпрограмм
- **Разделение кода на модули, содержащие описания функций, и скрипты с исполняемым кодом**
- Создание файловой структуры приложения

Модули с описаниями функций vs Исполняемые скрипты

Разделение на **модули (логика)** и **исполняемые скрипты (запуск)** — это не просто "правило хорошего тона", а **практическая необходимость** для создания:

- надёжного,
- масштабируемого,
- тестируемого,
- переиспользуемого кода.

Модули с описаниями функций vs Исполняемые скрипты

1. Повторное использование кода

Если функции лежат в отдельных модулях (например, `math_utils.php`, `file_handler.py`), их можно **импортировать** в разные скрипты без дублирования.

Модули с описаниями функций vs Исполняемые скрипты

× **Плохо** (всё в одном файле):

```php
// script.php
function add($a, $b) { return $a + $b; }
echo add(2, 3); // исполняемый код
```

✅ **Хорошо:**

```php
// math.php
function add($a, $b) {
    return $a + $b;
}

// main.php
require 'math.php';
echo add(2, 3);
```

Теперь `add()` можно использовать в десятках скриптов — просто подключив `math.php`.

Модули с описаниями функций vs Исполняемые скрипты

2. Тестируемость

Если модуль содержит **только определения**, его легко протестировать **автоматически**.

Например, в PHPUnit или PyTest вы можете:

- импортировать модуль,
- вызвать функцию с разными аргументами,
- проверить результат.

Модули с описаниями функций vs Исполняемые скрипты

3. Избежание побочных эффектов при импорте

Когда вы подключаете файл (через `require`, `import`, `include` и т.п.), вы **ожидаете**, что он просто определит функции/классы, а **не начнёт что-то делать** (писать в файл, отправлять запросы, выводить текст).

Если модуль сразу выполняет код — это **побочный эффект**, который:

- усложняет отладку,
- может привести к ошибкам при повторном подключении,
- нарушает принцип "импорт = безопасная операция".

Модули с описаниями функций vs Исполняемые скрипты

4. Чёткое разделение ответственности

- **Модули** — отвечают за **логику**: "как что-то сделать".
- **Исполняемые скрипты** — отвечают за **запуск**: "что запустить и когда".

Это соответствует принципу **единой ответственности (SRP)** из SOLID.

Пример структуры проекта:

```
1  project/
2  ├── lib/
3  │   ├── calculator.php      ← только функции
4  │   └── logger.php          ← только классы/функции
5  └── app.php                 ← исполняемый скрипт: использует lib/
```

Модули с описаниями функций vs Исполняемые скрипты

5. Безопасность и контроль над выполнением

Если весь код "размазан" по файлам и запускается при подключении, сложно контролировать:

- что запускается при тестировании,
- что запускается в production,
- не происходит ли инициализация БД в момент импорта библиотеки.

Когда исполняемый код находится **только в одном месте** (например, в `main.php` или внутри `if __name__ == "__main__":` в Python), вы точно знаете, **где начинается программа**.

Модули с описаниями функций vs Исполняемые скрипты

6. Поддержка CLI и веб-интерфейсов

Один и тот же модуль с логикой можно использовать:

- в консольном скрипте,
- в веб-приложении (через HTTP-запрос),
- в API,
- в фоновом процессе.

Если логика завязана на `echo` или `input()` прямо в модуле — такая гибкость теряется.

Программирование задачи

- Разработка алгоритма
- Выбор технологии и языка программирования
- Проектирование архитектуры приложения
- **Кодирование и оформление кода**
- Тестирование

Кодирование и оформление кода

Программирование - раздел лингвистики.
ЯП - это искусственный язык, на котором мы общаемся с машиной (10%) и другими программистами (90%).

Код должен быть самодокументируемым и понятным

Основная задача - **создание кода, читая который можно воспроизвести задачу в исходных терминах**

Кодирование и оформление кода

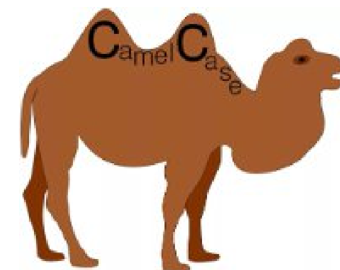
- Осмысленные и произносимые имена сущностей
- Только английские слова
- Не должно быть магических чисел (нужны константы)

Именование переменных

Схемы именования переменных

- Визуальное выделение составных частей имени
- Понимание сути переменной, ее назначение и тип (именованная константа; элементарная переменная; тип, определенный пользователем, или класс).

Схемы именования переменных



CamelCase (CamelCase, PascalCase). Применяется для имен классов Java, в методах Windows API и .NET.

camelCase. Применяется для членов классов в Java, для переменных в JavaScript.

snake_case (змеиная нотация). Принят в PHP.

kebab-case (шашлычная нотация)

UPPERCASE_WITH_UNDERSCORES. Для констант.

_name. Переменная объявлена как private или protected и недоступна вне класса.

Имена переменных

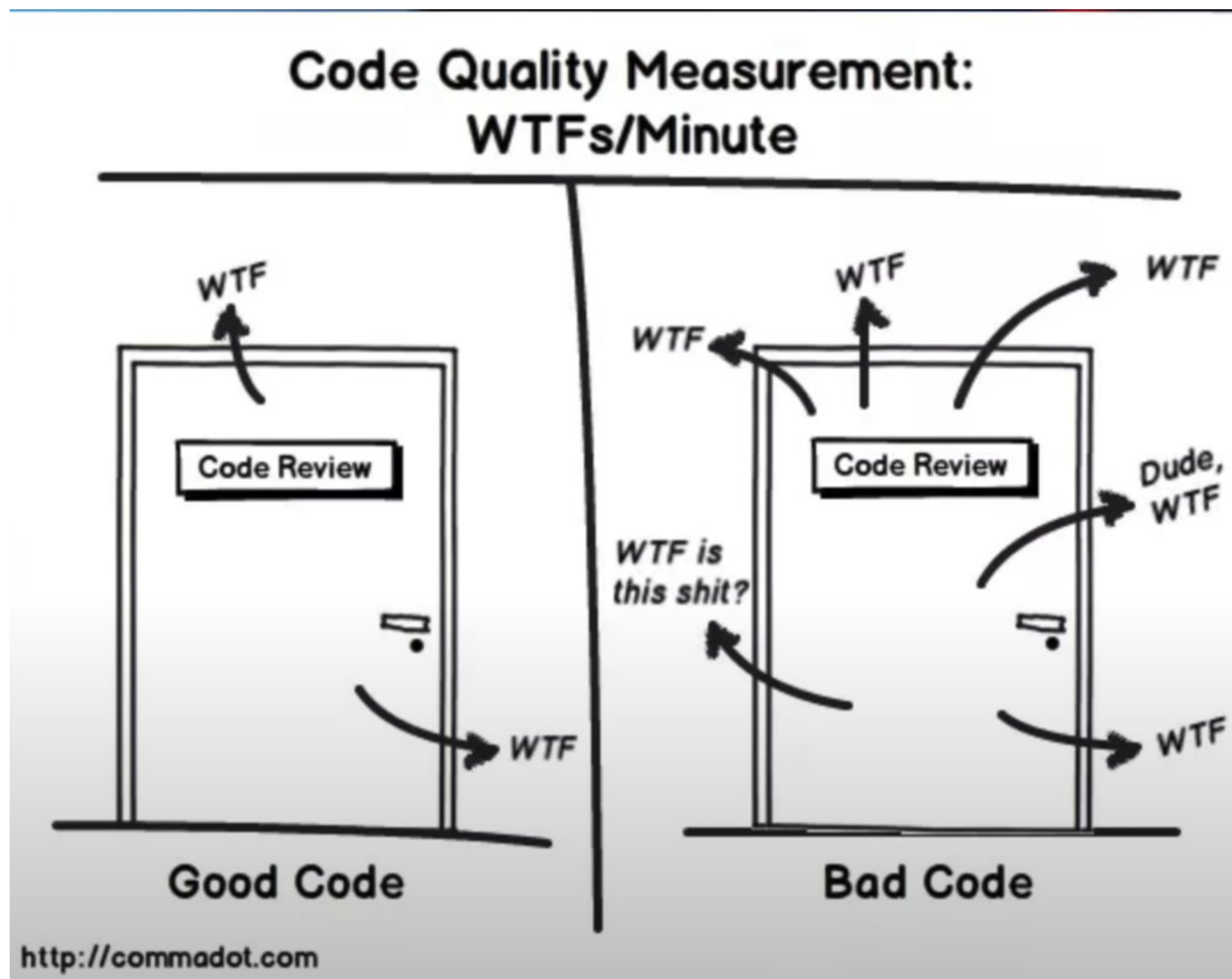
- **Имена должны быть максимально конкретны.** Для важных переменных следует избегать имен x, j, temp, … . Короткие имена допустимы в качестве счетчиков в циклах и т.п.

- **Следует отказаться от сокращения имен.** Нужно выбирать имена так, чтобы они облегчали чтение кода, даже за счет удобства его написания.

- Пример: *В переменной хранится текущая дата. Как назвать переменную?*

  `CD`, `CurDate`, `CurrentDate`

Принцип наименьшего удивления



Имена переменных

Хорошее имя чаще всего описывает проблему, а не ее решение (отвечает на вопрос ***что?*** а не ***как?***)

- Запись данных о сотруднике: **`inputRecord`** или **`employeeData`**
- Статус принтера: **`bitFlag`** или **printerReady**

Имена переменных

Следует избегать имен:

- Общие термины
  `data`, `value`, `temp`
- Синонимы
  `number` и `nomer`
- Имеют разную суть, но похожие имена
  `clientRecs`, `clientReps`
- Содержат цифры (лучше массивы)
- Отличаются регистром символов
- Совпадают со стандартными типами, переменными и методами
  `integer`, `string`
- Не связаны со смыслом переменных.

Никогда не использовать символы `I`, `l`, `O` как однобуквенные идентификаторы!

Имена переменных

Хорошие программисты...

- Понимают важность выбора имен и занимаются этой проблемой
- Не забывают о выборе правильного имени для каждого создаваемого объекта
- Учитывают многие факторы: длину имени, понятность, контекст и т. д.
- Видят общую картину и выбирают имена в рамках проекта (или проектов)

Плохие программисты...

- Не заботятся о понятности своего кода
- Пишут *одноразовый* код, плохо продумывая его в погоне за скоростью
- Игнорируют естественную идиоматику языка
- Не стремятся к единообразию в именах
- Не заботятся об общей картине и не интересуются согласованностью своего кода с проектом в целом

Имена переменных

Хорошо ли выбраны имена?

a. `int apple_count`
b. `char foo`
c. `bool apple_count`
d. `char *string`
e. `int loop_counter`

Имена переменных

Что делает этот код?

```
void bsrt(int a[], int n)
{
    for (int i = 0; i < n1; i++)
        for (int j = n1; j > i; j)
            if (a[j1] > a[j])
            {
                int tmp = a[j1];
                a[j1] = a[j];
                a[j] = tmp;
            }
}
```

Имена переменных

Что делает этот код?

```
void bsrt(int a[], int n)
{
    for (int i = 0; i < n1; i++)
        for (int j = n1; j > i; j)
            if (a[j1] > a[j])
            {
                int tmp = a[j1];
                a[j1] = a[j];
                a[j] = tmp;
            }
}
```

```
void swap(int *first, int *second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}

void bubbleSort(int items[], int size)
{
    for (int pos1 = 0; pos1 < size1; pos1++)
        for (int pos2 = size1; pos2 > pos1; pos2)
            if (items[pos21] > items[pos2])
                swap(&items[pos21],
&items[pos2]);
}
```

Именование функций

Именование функций

Функция - это действие, название должно быть глаголом

Действие-Объект
Глагол-Существительное

`getUser`, `printPage`

Командлет Get-Verb в PowerShell

Именование функций

- **Описывайте все, что метод выполняет.**

```
Date newDate = date.add(5);
```

Именование функций

- **Описывайте все, что метод выполняет.**

```
Date newDate = date.add(5);
```

1. Что прибавляется к дате (дни, часы, недели)?

2. Изменяется ли экземпляр date, или функция возвращает новое значение Date без изменения старого?

Именование функций

- **Описывайте все, что метод выполняет**.

```
Date newDate = date.add(5);
```

Если функция прибавляет пять дней с изменением date, то она должна называться `addDaysTo` или `increaseByDays`.

Если функция возвращает новую дату, смещенную на пять дней, но не изменяет исходного экземпляра `date`, то она должна называться `daysLater` или `daysSince`.

Именование функций

- Описывайте все, что метод выполняет.
- **Избегайте невыразительных глаголов**.

`HandleCalculation()`, `PerformServices()`, `OutputUser(), ProcessInput()`

HandleOutput() – непонятно. Лучше FormatAndPrintOutput().

Именование функций

- Описывайте все, что метод выполняет.
- Избегайте невыразительных глаголов.
- **Для именования функции используйте описание возвращаемого значения**.

  `customerId.Next()`, `printer.IsReady()`, `pen.CurrentColor()`

Именование функций

- **Аккуратно используйте антонимы.**

```
add / remove
increment / decrement
open / close
begin / end
insert / delete
show / hide
create / destroy
lock / unlock
source / target
first / last
min / max
start / stop
get / put
next / previous
up / down
get / set
old / new
```

Именование функций

- **Аккуратно используйте антонимы.**
- **Определяйте правила именования часто используемых операций и придерживайтесь их.**

Метод, возвращающий уникальный идентификатор объекта:

```
employee.id.Get()
dependent.GetId()
candidate.id()
```

Оформление кода – специфика PHP

- Соблюдение стандартов кодирования PSR-1, PSR-12 (PHP Code Sniffer, PHP Coding Standard Fixer)