

Министерство образования Республики Беларусь

Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра программного обеспечения информационных технологий

Дисциплина
«Объектно-ориентированные технологии программирования и
стандарты проектирования»

ОТЧЁТ
к лабораторной работе №1

ПОСТРОЕНИЕ ИЕРАРХИИ КЛАССОВ
В РАМКАХ ПРОЕКТА «NINJA-ARENA»

Студент группы
№351001
Ушаков А.Д.

Преподаватель
Деменковец Д. В.

Минск 2025

СОДЕРЖАНИЕ

Введение	3
1 Теоретические сведения	4
1.1 Вводная теория и определения (из лекционного курса)	4
1.2 Паттерны объектно-ориентированного программирования	5
1.3 Ответы на вопросы преподавателя	7
2 Организация проекта	8
2.1 Расположение модулей	8
2.2 Иерархия классов (директория models)	8
2.3 Директория components	10
2.4 Сборка и компиляция проекта	11
2.5 UI (директория assests)	13
3 Тестирование	16
3.1 Изменение контейнера	16
3.2 Нахождение объекта в контейнере	18
Заключение	19
Список использованных источников	20
Приложение А (обязательное) Абстрактный класс «Ninja»	21
Приложение Б (обязательное) Подкласс Ninja – «fireNinja»	22
Приложение В (справочное) Модуль fire_powers.ts	23
Приложение Г (обязательное) Подкласс fireNinja – класс «eyesClanNinja» ...	24
Приложение Д (обязательное) Подкласс eyesClanNinja – особый ниндзя «sensorNinja»	25

ВВЕДЕНИЕ

Современные технологии разработки программного обеспечения требуют от разработчиков не только глубоких знаний языков программирования, но и умения применять принципы объектно-ориентированного проектирования. В рамках дисциплины "Объектно-ориентированные технологии проектирования и стандарты проектирования" был создан проект "Ninja-Arena", целью которого является реализация иерархии классов, моделирующих персонажей из вселенной "Naruto". Проект направлен на изучение и применение ключевых концепций ООП, таких как наследование, инкапсуляция и полиморфизм, а также на освоение работы с модульной структурой проекта и взаимодействием с DOM [2].

Основной задачей первой части работы стало проектирование и реализация иерархии классов ниндзя. В основе иерархии лежит абстрактный класс "Ниндзя", от которого наследуются более специализированные классы: "Ниндзя стихии", "Ниндзя клана" и "Особый ниндзя клана". Каждый из этих классов обладает уникальными свойствами и методами, отражающими особенности персонажей из аниме-вселенной. Для хранения созданных объектов был разработан контейнер, позволяющий управлять коллекцией ниндзя и осуществлять их поиск.

Важной частью проекта стала реализация веб-интерфейса, который позволяет просматривать информацию о всех ниндзя в контейнере, а также выводить данные о конкретном персонаже. Для этого использовались технологии работы с DOM, что позволило динамически изменять содержимое страницы в зависимости от действий пользователя. Проект был организован с учётом модульной структуры, что обеспечивает удобство поддержки и расширения функциональности в будущем.

В перспективе планируется развитие проекта путём добавления функционала арены для проведения боёв между персонажами. Уже реализованные классы ниндзя содержат методы, которые будут использоваться для моделирования сражений. Это позволит не только углубить понимание принципов ООП, но и создать интерактивное приложение, демонстрирующее возможности объектно-ориентированного подхода в разработке сложных систем.

1 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Вводная теория и определения (из лекционного курса)

Базовые понятия структурного программирования:

Переменная – ячейка данных. Тип данных – допустимые значения и операции переменной. Указатель – переменная, содержащая адрес. Оператор – единица выполнения алгоритма. Процедура – вызываемая на выполнение подпрограмма. Процедурная переменная – указатель на процедуру. Модуль – единица разработки и доставки программы. Объект – динамический модуль. Исключение – событие и объект с информацией об ошибке.

Модуль — единица разработки, применения и поставки. Разграничение доступа к модулям. Разграничение доступа к модулю осуществляется с помощью ключевых слов: `public` – доступ к модулю получают все; `protected` – доступ к модулю получают данный модуль и модули расширения; `internal` – доступ к модулю получают данный модуль и программы, в которых данный модуль подключается на уровне исходного кода; `protected internal` – доступ к модулю получают модули расширения и программы, в которых данный модуль подключается на уровне исходного кода; `private` – доступ к модулю получают лишь процедуры этого модуля.

Объект – динамический модуль, у которого может быть много экземпляров. Описание динамического модуля – класс.

Исключение – событие и объект с информацией об ошибке. Исключение создается: аппаратно – в результате ошибки выполнения оператора; программно – с помощью оператора создания исключения: `throw new OutOfMemoryException("Не хватает памяти")`.

Базовые понятия объектно-ориентированного программирования:

Класс – тип данных для создания объектов. Объект – экземпляр класса. Метод – процедура над объектом. Конструктор и деструктор – особые методы. Свойство – виртуальное поле. Наследование – расширение класса. Виртуальный метод – переопределяемый метод. Делегат – ссылка на метод. Событие – список делегатов. Интерфейс – описание класса без реализации. Шаблон – параметризованный класс. Атрибут – метаданные, механизм рефлексии.

Конструктор и деструктор – особые методы, связанные с инициализацией объекта.

Свойство – виртуальное поле. Работа со свойством внешне не отличается от работы с полем. Назначение свойств – создание побочных эффектов при обращении, например, открытие файла при установке свойства `Active` в `true`.

1.2 Паттерны объектно-ориентированного программирования

Инкапсуляция. Одним из самых важных факторов при проектировании компонентов приложения является сокрытие внутренних данных компонента и деталей его реализации от других компонентов приложения и предоставление набора методов для взаимодействия с ним (API). Этот принцип является одним из четырёх фундаментальных принципов ООП и называется инкапсуляцией.

Правильная инкапсуляция важна по многим причинам:

1 Она способствует переиспользованию компонентов: поскольку в этом случае компоненты взаимодействуют друг с другом только посредством их API и безразличны к изменениям внутренней структуры, они могут использоваться в более широком контексте.

2 Инкапсуляция ускоряет процесс разработки: слабо связанные друг с другом компоненты (то есть компоненты, чей код как можно меньше обращается или использует код других компонентов) могут разрабатываться, тестироваться и дополняться независимо.

3 Правильно инкапсулированные компоненты более легки для понимания и процесса отладки, что упрощает поддержку приложения.

Во многих языках инкапсуляция реализована с помощью системы классов, которые позволяют собрать информацию об объекте в одном месте; пакетов, которые группируют классы по какому-либо критерию, и модификаторов доступа, которыми можно пометить весь класс или его поле или метод.

Наследование [3] является одним из важнейших принципов объектно-ориентированного программирования, поскольку оно позволяет создавать иерархические структуры объектов. Используя наследование, можно создать общий класс, который будет определять характеристики и поведение, свойственные какому-то набору связанных объектов. В дальнейшем этот класс может быть унаследован другими, более частными классами, каждый из которых будет добавлять уникальные, свойственные только ему характеристики и дополнять или изменять поведение базового класса. В терминах многих объектно-ориентированных языков такой общий класс называется суперклассом (superclass), или базовым классом (base class), или классом-родителем (parent class), а класс, его наследующий, - подклассом (subclass), или дочерним классом (child class), или классом-потомком (derived class).

Полиморфизм [3] — это свойство функции обрабатывать различные типы объектов и классов.

Основная задача полиморфизма — оптимизация кода и удаление дублирующих друг друга команд. Если проще, функции из разных классов будут выполнять одну и ту же команду. Для этого не нужно прописывать отдельные команды для каждого участка кода.

Рассмотрим пример. Допустим, у нас в магазине есть два товара: один вызывается методом `AddToCartBoll`, а другой `AddToCartChess`. Нам не нужно прописывать каждый метод по отдельности, потому что достаточно одного `AddToCart`. При этом если у нас появится третий товар, мы просто чуть изменим код, чтобы работа `AddToCart` распространялась и на него.

Совершенно не важно, что это за объекты — они будут выполнять одну и ту же команду. При этом реакция на команду будет разной. К примеру, у нас есть два объекта с разными свойствами: с помощью этого способа можно добиться, чтобы они обрабатывались одним методом или функцией.

Простыми словами, полиморфизм — это то, что позволяет использовать один и тот же метод для разных объектов.

Абстракция — это процесс выделения общих характеристик и функциональности объектов или системы, игнорируя детали реализации. Данный принцип позволяет разрабатывать программы на различных языках программирования, скрывая сложность и детали нижележащего кода. Это делается для упрощения сложных систем и концепций, чтобы разработчики могли фокусироваться на основных аспектах проблемы и легче понимали код.

Абстракция позволяет создавать абстрактные классы и интерфейсы, которые определяют общие свойства и методы, не зависящие от конкретной реализации. Преимущества абстракции ООП включают:

- 1 Упрощение сложности: абстракция в программировании позволяет скрыть детали реализации и сосредоточиться на ключевых аспектах системы. Это помогает упростить понимание и поддержку кода.

- 2 Модульность: возможность разбить систему на модули или классы, которые могут работать независимо друг от друга. Это способствует повторному использованию кода и улучшает масштабируемость проекта.

- 3 Повышение безопасности: абстракция позволяет скрыть некоторые детали реализации, что делает код более безопасным и защищенным. Внешние компоненты не имеют прямого доступа к внутренним деталям объекта или системы.

1.3 Ответы на вопросы преподавателя

Связанность. В объектно-ориентированном программировании (ООП) под связанностью понимается степень прямой осведомленности одного элемента о другом. Другими словами, как часто изменения в классе А приводят к соответствующим изменениям в классе В. Связанность можно рассматривать как горизонтальную характеристику, так как она описывает взаимодействие между элементами, находящимися на одном уровне абстракции. Например, два класса, которые взаимодействуют друг с другом, находятся на одном уровне иерархии.

Связность. В ООП связность определяет то, как спроектирован отдельный класс. Она гарантирует, что класс разработан с единой и четко сформулированной целью. Другими словами, если все составляющие класса похожи по многим аспектам, то он обладает сильной связностью. Связность можно рассматривать как вертикальную характеристику, так как она описывает внутреннюю организацию и целостность одного элемента системы. Это относится к тому, как элементы внутри класса или модуля организованы вокруг общей цели.

В хорошем дизайне системы стремятся к низкой связанности (чтобы элементы системы были максимально независимы) и высокой связности (чтобы каждый элемент системы был целостным и выполнял одну четкую задачу).

Ответственность класса – это его обязанность выполнять определенную функциональность или управлять определенными данными. Каждый класс должен иметь четко определенную зону ответственности, чтобы система была modular, понятной и легко поддерживаемой.

В ООП ответственность – это ключевое понятие, которое определяет, за что именно отвечает конкретный класс или объект. Ответственность связана с тем, какие задачи или функции должен выполнять класс, и какие данные он должен обрабатывать.

Класс должен иметь только одну причину для изменения, то есть он должен отвечать только за одну задачу или функциональность. Это означает, что класс не должен быть "универсальным" и пытаться делать слишком много. Вместо этого он должен фокусироваться на одной конкретной задаче.

2 ОРГАНИЗАЦИЯ ПРОЕКТА

2.1 Расположение модулей

Все файлы проекта, относящиеся к реализации, располагаются в директории `src`, которая разветвляется на `asests`, `components`, `models` (см. рисунок 2.1.1).

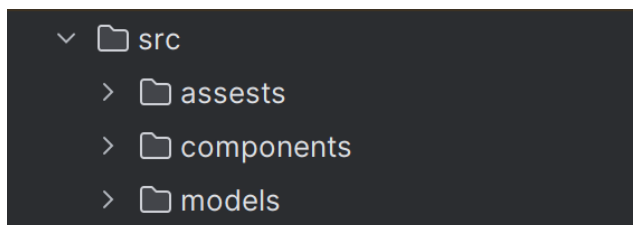


Рисунок 2.1.1 – Главная директория `src`

В директории `asests` хранятся файлы, реализующие пользовательский интерфейс в формате веб-страниц, а также определений базовых элементов для их формирования.

Папка `components` содержит непосредственные объявление и создание объектов (персонажей-ниндзя), связанные с ними информацию и файлы (например, изображения `.jpg`). Также в `components` происходит инициализация контейнера, хранящего в себе все объекты.

Директория `models` хранит всю иерархию классов, на которых базируются созданные объекты (см. следующий подраздел).

2.2 Иерархия классов (директория `models`)

Как упоминалось ранее, проект был реализован по мотивам вселенной «Naruto», где вся фабула выстроена вокруг жизни «шиноби». Ниндзя обладают разными стихиями, принадлежат к различным кланам и живут во множестве деревень. Иерархия была составлена следующим образом: создаётся абстрактный, базовый для всех, класс «Ниндзя». Затем от него наследуются ниндзя, владеющие разными стихиями. На этом моменте как раз таки отчётливо виден принцип **полиморфизма**: у базового класса есть методы атаки, регенерации и суперспособностей, которые ниндзя каждой стихии реализует по-своему (эти функции определены в отдельных файлах, например, `fire_powers`). Далее реализован принцип **наследования**: ниндзя стихий могут состоять в разных кланах – у персонажей каждого клана свои поля и методы для будущих сражений на арене. Последним, третим уровнем вложенности являются «специальные ниндзя», которые наверняка найдутся в каждом клане.

На рисунке 2.2.1 проиллюстрировано, каким образом организована иерархия классов.

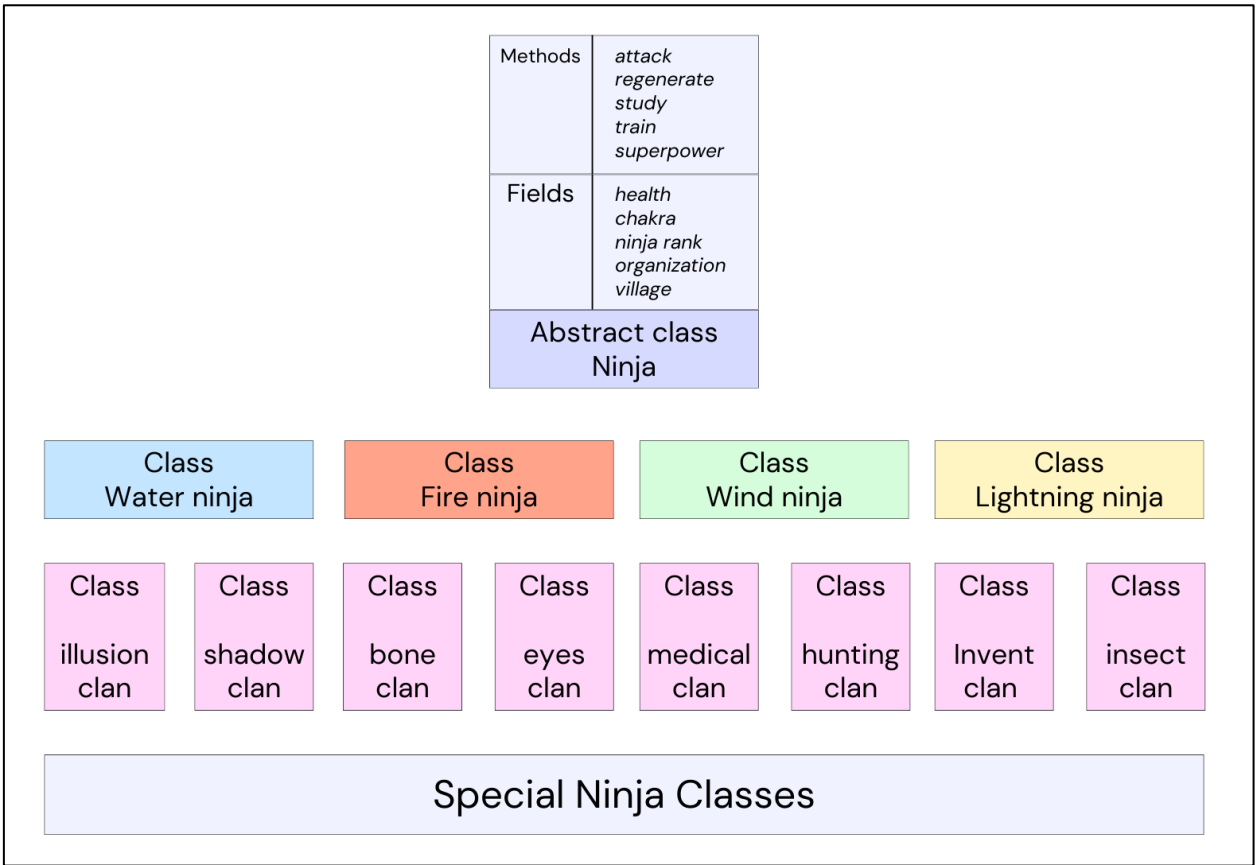


Рисунок 2.2.1 – Иерархичная структура типов проекта

Ниже, на рисунке 2.2.2, представлена детальная файловая организация иерархии классов.

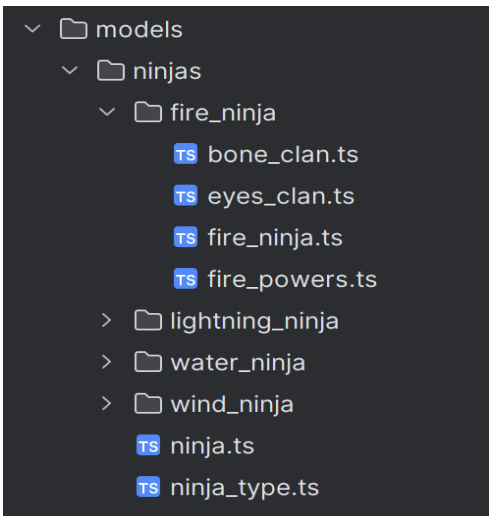


Рисунок 2.2.2 – Файловая структура директории models

В приложениях А-Ч можно ознакомиться с непосредственной реализацией классов на языке TypeScript.

2.3 Директория components

Директория components, как упоминалось в подразделе 2.1, содержит создание и инициализацию объектов проекта, а также сборный контейнер, куда они помещаются (см. рисунок 2.3.1).

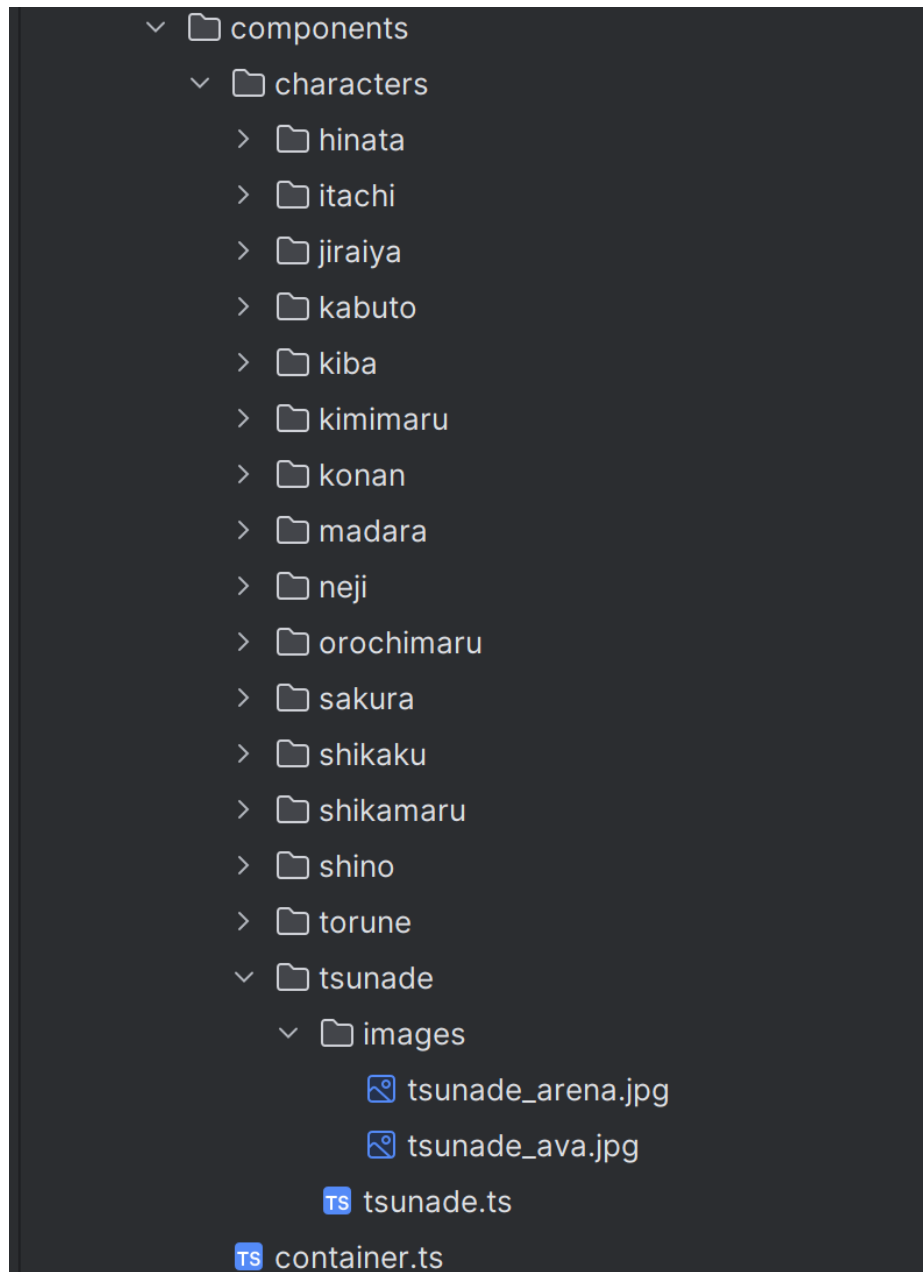


Рисунок 2.3.1 – Файловая организация объектов приложения

Объект инициализируется следующим образом [1]:

```
import ...

export const Tsunade = new neurosurgeon("Tsunade", 1500, 1000,
Rank.K,
"sanins", "Konohagakure",
`${baseImagesPath}/tsunade/images/tsunade_ava.jpg`,
`${baseImagesPath}/tsunade/images/tsunade_arena.jpg`, 1.5, 3);
```

Затем он помещается в контейнер в отдельном модуле `container.ts`:

```
import {Shikamaru} from "../characters/shikamaru/shikamaru";

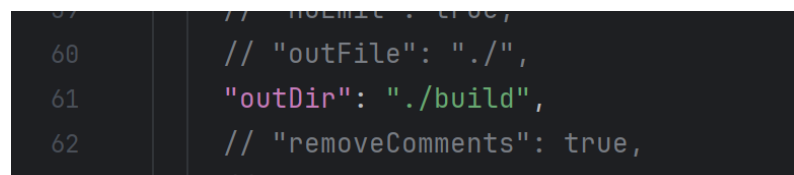
export let Container: Array<NinjaType> = [];

Container.push(Shikamaru);
```

2.4 Сборка и компиляция проекта

В ходе работы над проектом пришлось столкнуться с большими трудностями в реализации визуального интерфейса (следующий подраздел), так как браузер не интерпретирует typescript файлы, а работает только с javascript. Стало необходимым скомпилировать все typescript-файлы в js – директория `build`, а также учесть некоторые тонкости сборки, например, соотнести разные стандарты для модулей (CommonJS, ES6 и т.д.), настроить конфигурацию проекта `tsconfig.json` и т.д.

Для того, чтобы скомпилировать ts-файлы, необходимо в `tsconfig` указать выходную директорию `“./build”` (рис. 2.4.1), после чего в командной строке запустить компилятор `tsc` (рис. 2.4.2).



```
60 // "outFile": "./",
61 "outDir": "./build",
62 // "removeComments": true,
```

Рисунок 2.4.1 – Главная настройка `tsconfig`



```
PS D:\bsuir\00TPiSP> npx tsc
```

Рисунок 2.4.2 – Запуск компиляции

Важно упомянуть, что перед данными действиями в первую очередь необходимо инициализировать корневой-каталог как проект typescript.

После процесса компиляции формируется папка build, которая имеет такую же структуру, как и src (см. рисунок 2.4.3), только с js-файлами.

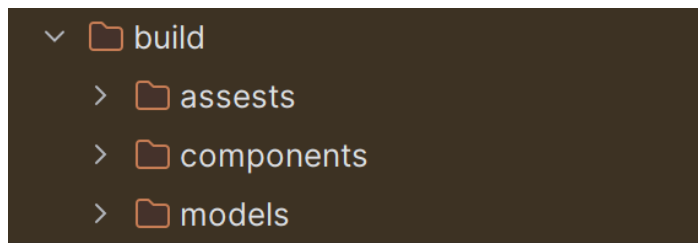


Рисунок 2.4.3 – Структура директории build

Одной из проблем при настройке проекта стало некорректное формирование путей импорта и экспорта js-модулей. В typescript-файлах пути взаимодействия модулей задаются строковыми литералами, которые содержат полные названия файлов без указания расширения. А для того, чтобы скомпилированные js-модули могли корректно подключать друг друга, необходим постфикс `-.js` в строковых литералах импорта и экспорта. Это необходимо делать при каждой компиляции (например, после добавления объектов) вручную, прописывая команду «под значком лампочки» (см. рисунок 2.4.4).

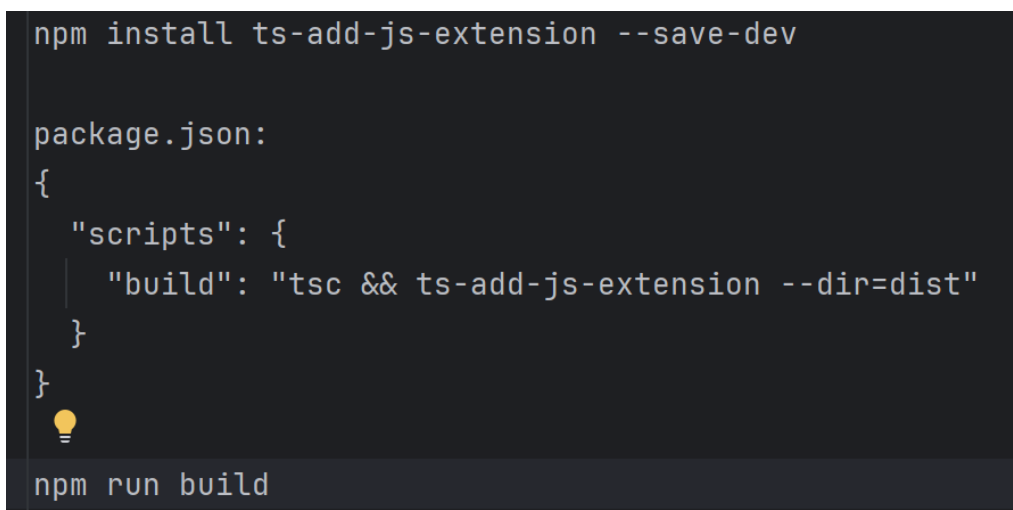


Рисунок 2.4.4 – Настройка взаимодействия js-модулей

2.5 UI (директория assests)

В разделе про сборку и компиляцию проекта было упомянуто о сложности создания веб-страницы под typescript-файлы. И тем не менее практически все используемые модули имеют расширение ts. Есть только один файл расширения js из папки build, который связывает backend-часть и frontend.

Для формирования html-элементов, которые получают данные классов с backend-части, были реализованы специальные «гибридные» классы для интерфейса (см. код ниже). Стоит отметить, что принципы ООП также были соблюдены и на уровне визуализации: есть отдельный «гибридный» класс для карточки персонажа и для контейнера на веб-странице.

```
export class Card {
  _character: NinjaType;
  _photo: string;
  _name: string;
  _health: string;
  _chakra: string;
  _rank: string;
  _organization: string;
  _village: string;

  constructor(character: NinjaType) {
    this._character = character;
    this._photo = `
      <div class="character-card__photo">
        
      </div>`;
    this._name = `
      <div class="character-card__name">${character.name}</div>
    `;
    this._health = `
      <div class="character-card__field character-
card__field_property">Health</div>
      <div class="character-card__field character-card__field_value">
        ${character.health}pt
      </div>
    `;
    this._chakra = `
      <div class="character-card__field character-
card__field_property">Chakra</div>
      <div class="character-card__field character-card__field_value">
        ${character.chakra}pt
      </div>
    `;
    this._rank = `
      <div class="character-card__field character-card__field_property">Ninja
rank</div>
      <div class="character-card__field character-card__field_value">
        ${character.rank}
      </div>
    `;
  }
}
```

```

    </div>
  `;
  ...
}

makeCard(): string {
  let result: string;

  result =
    `

Ответ на вопрос, почему данный класс-интерфейс был назван «гибридным»: он принимает на вход объект с backend-части, который не имеет ничего, кроме свойств, и формирует из него карточку персонажа в формате html, как своеобразный мост между логической частью и визуальной. Метод makeCard() объединяет все свойства карточки в один html-блок. Таким образом был продемонстрирован принцип инкапсуляции, так как данный элемент возможно будет использовать неоднократно.



На текущем этапе развития проекта в соответствии с требованиями лабораторной работы было реализовано 2 веб-страницы, на одной из которых выводится контейнер, содержащий все созданные объекты (см. рис. 2.5.1), а на другой пользователь самостоятельно, введя в поисковую строку, может найти интересующий его объект (см. рис. 2.5.2).



| Character  | Health | Chakra | Ninja rank | Organization | Village      |
|------------|--------|--------|------------|--------------|--------------|
| Hinata     | 800pt  | 400pt  | 1          | none         | Konohagakure |
| Itachi     | 1200pt | 600pt  | 3          | akatsuki     | Konohagakure |
| Jiraiya    | 1400pt | 800pt  | 4          | sanins       | Konohagakure |
| Kabuto     | 1000pt | 500pt  | 3          | none         | Konohagakure |
| Kiba       | 900pt  | 500pt  | 2          | none         | Konohagakure |
| Kimimaru   |        |        |            |              |              |
| Konan      |        |        |            |              |              |
| Madara     |        |        |            |              |              |
| Neji       |        |        |            |              |              |
| Orochimaru |        |        |            |              |              |



Рисунок 2.5.1 – Страница созданных персонажей



14


```

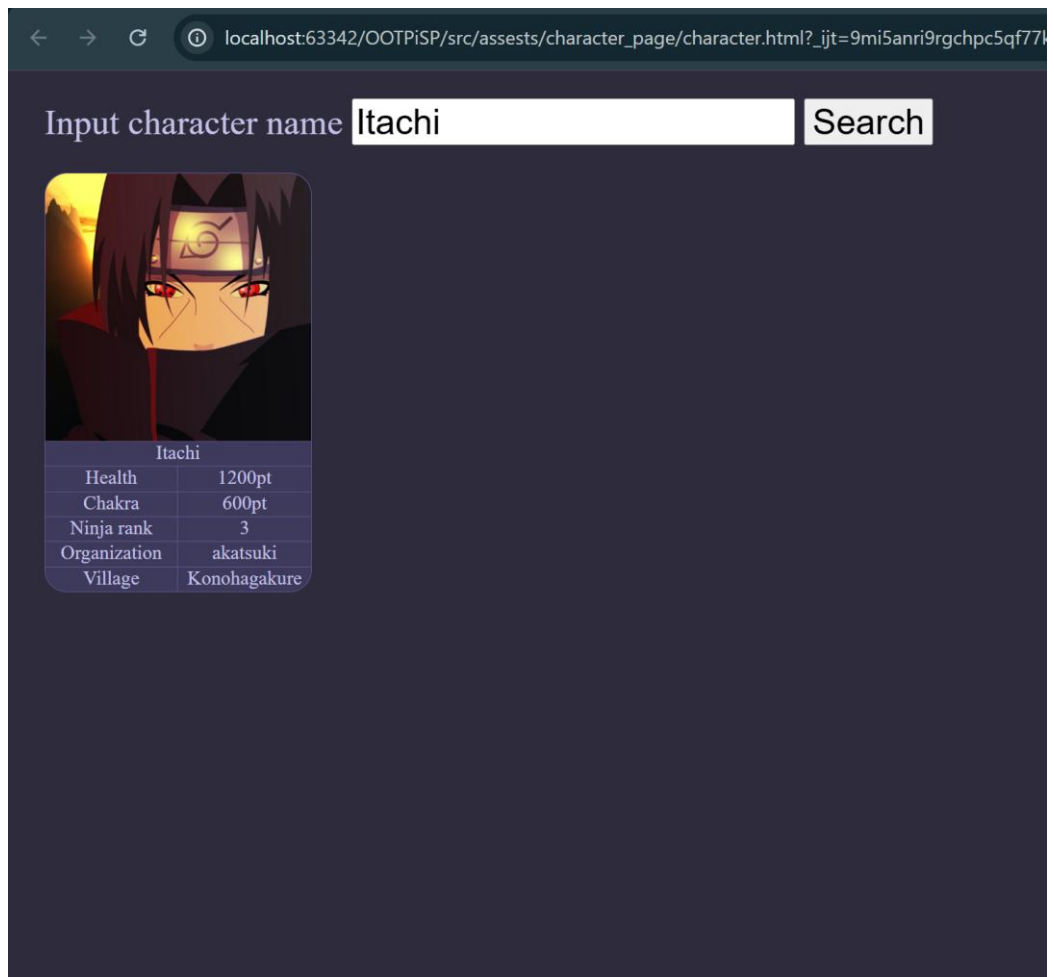


Рисунок 2.5.2 – Поиск интересующего объекта

3 ТЕСТИРОВАНИЕ

3.1 Изменение контейнера

На рисунке 3.1.1 показан изначальный состав контейнера, в который были добавлены все существующие объекты. В свою очередь, на рисунке 3.1.2 показан внешний вид контейнера на нижней части страницы, которая впоследствии будет изменена.

```
21 Container.push(Hinata);
22 Container.push(Itachi);
23 Container.push(Jiraiya);
24 Container.push(Kabuto);
25 Container.push(Kiba);
26 Container.push(Kimimaru);
27 Container.push(Konan);
28 Container.push(Madara);
29 Container.push(Neji);
30 Container.push(Orochimaru);
31 Container.push(Sakura);
32 Container.push(Shikaku);
33 Container.push(Shikamaru);
34 Container.push(Shino);
35 Container.push(Torune);
36 Container.push(Tsunade);
```

Рисунок 3.1.1 – Изначальный состав контейнера

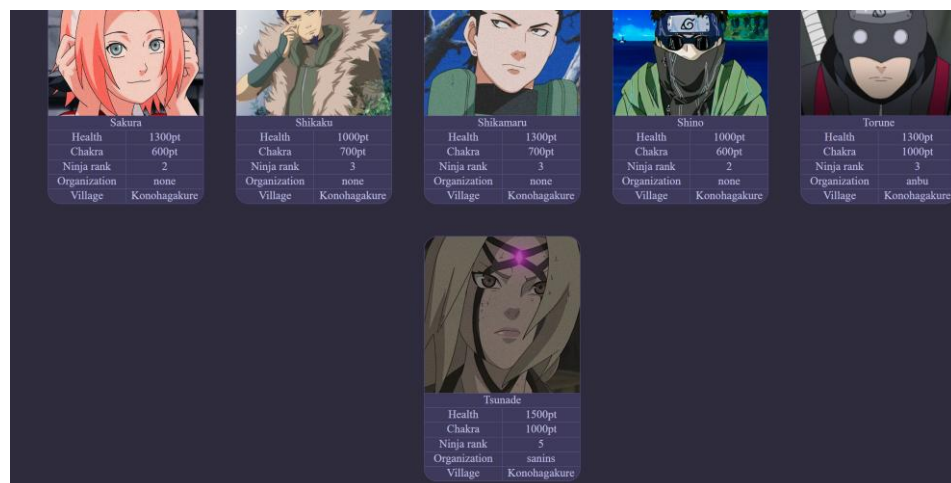


Рисунок 3.1.2 – Нижняя часть страницы

Теперь уберём из контейнера последнего персонажа с именем Tsunade (см. рисунок 3.1.3), после заново скомпилируем проект и откроем веб-страницу в браузере (рис. 3.1.4).

```

21 Container.push(Hinata);
22 Container.push(Itachi);
23 Container.push(Jiraiya);
24 Container.push(Kabuto);
25 Container.push(Kiba);
26 Container.push(Kimimaru);
27 Container.push(Konan);
28 Container.push(Madara);
29 Container.push(Neji);
30 Container.push(Orochimaru);
31 Container.push(Sakura);
32 Container.push(Shikaku);
33 Container.push(Shikamaru);
34 Container.push(Shino);
35 Container.push(Torune);
36 //Container.push(Tsunade);

```

Рисунок 3.1.3 – Контейнер без последнего персонажа

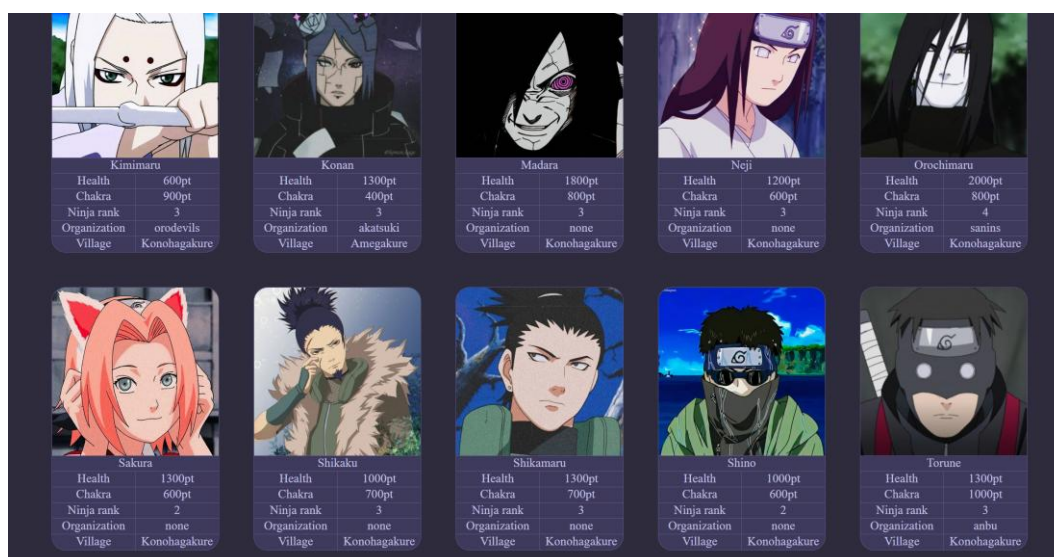


Рисунок 3.1.4 – Нижняя часть страницы

Исходя из внешнего вида страницы, на которой больше не отображается персонаж Tsunade, после изменения состава контейнера, можно сделать вывод, что тест пройден успешно.

3.2 Нахождение объекта в контейнере

При вводе пустой строки или невалидного имени, как и предполагалось при разработке, на странице не происходит никаких изменений – тест пройден успешно.

Если ввести текст, который соответствует имени существующего в контейнере персонажа, то его карточка отображается на странице (см. рис. 3.2.1). Изменив поле ввода на имя другого персонажа и нажав на кнопку, мы получим новую отобразившуюся карточку (см. рис. 3.2.2). Данные тесты пройдены успешно.



Рисунок 3.2.1 – Поиск персонажа в контейнере

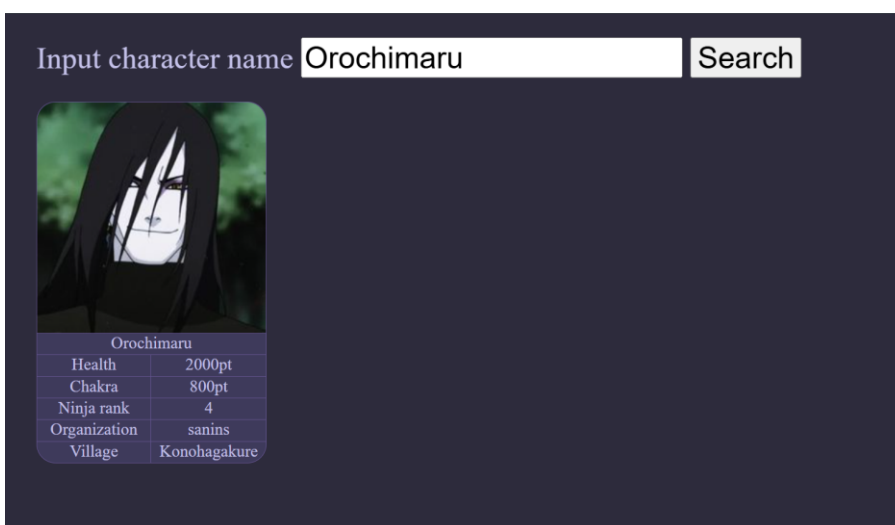


Рисунок 3.2.2 – Изменение параметра поиска

ЗАКЛЮЧЕНИЕ

Важной частью проекта стала реализация веб-интерфейса, который позволяет пользователю просматривать информацию о всех ниндзя в контейнере, а также находить конкретного персонажа по имени. Для этого были использованы технологии работы с DOM [2], что позволило динамически изменять содержимое страницы в зависимости от действий пользователя. Визуальная часть проекта была реализована с использованием «гибридных» классов, которые связывают логическую часть приложения с интерфейсом, демонстрируя принцип инкапсуляции.

Тестирование проекта подтвердило корректность работы всех компонентов. Были успешно протестированы изменения в контейнере, а также функциональность поиска персонажей. Проект показал свою устойчивость к изменениям и готовность к дальнейшему развитию.

В перспективе планируется расширение функциональности проекта путём добавления арены для проведения боёв между персонажами. Это позволит не только углубить понимание принципов ООП, но и создать интерактивное приложение, демонстрирующее возможности объектно-ориентированного подхода в разработке сложных систем.

Таким образом, в ходе выполнения лабораторной работы были достигнуты все поставленные цели, а также получены практические навыки работы с объектно-ориентированными технологиями и стандартами проектирования. Проект «Ninja-Arena» является наглядным примером применения принципов ООП в реальной разработке программного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] TypeScript Documentation [Электронный ресурс]. – Режим доступа : <https://www.typescriptlang.org/docs>.

[2] MDN Web Docs [Электронный ресурс]. – Режим доступа : <https://developer.mozilla.org>.

[3] Tproger [Электронный ресурс]. – Режим доступа : <https://tproger.ru>.

ПРИЛОЖЕНИЕ А
(обязательное)
Абстрактный класс «Ninja»

```
export abstract class Ninja {
  _name: string;
  _health: number;
  _chakra: number;
  "_ninja rank": Rank;
  _organization: Organizations;
  _village: Villages;
  _appearance: string;
  "_arena view": string;

  constructor(
    name: string,
    health: number,
    chakra: number,
    rank: Rank,
    organization: Organizations,
    village: Villages,
    appearance: string,
    arena_view: string,
  ) {
    this._name = name;
    this._health = health;
    this._chakra = chakra;
    this["_ninja rank"] = rank;
    this._organization = organization;
    this._village = village;
    this._appearance = appearance;
    this["_arena view"] = arena_view;
  }

  get name(): string { return this._name; }
  get health(): number { return this._health; }
  get chakra(): number { return this._chakra; }
  get rank(): Rank { return this["_ninja rank"]; }
  get organization(): string { return this._organization; }
  get village(): string { return this._village; }
  get appearance(): string { return this._appearance; }
  get arenaView(): string { return this["_arena view"]; }

  study(requirement: boolean): any {
  };
  train(score: number): any {
  };

  abstract attack(arena: Opponents): Opponents;
  abstract regenerate(arena: Opponents): Opponents;
  abstract superpower(arena: Opponents): Opponents;
}
```

ПРИЛОЖЕНИЕ Б
(обязательное)
Подкласс Ninja – «fireNinja»

```
export class fireNinja extends Ninja {
  "_flame power": number;

  constructor (
    name: string,
    health: number,
    chakra: number,
    rank: Rank,
    organization: Organizations,
    village: Villages,
    appearance: string,
    arena_view: string,
    flame_power: number,
  ) {
    super(name, health, chakra, rank, organization,
      village, appearance, arena_view);
    this["_flame power"] = flame_power;
  }

  get flamePower() { return this["_flame power"]; }

  attack(arena: Opponents): Opponents {
    return fireBall(arena);
  }

  regenerate(arena: Opponents): Opponents {
    return heartOfHell(arena);
  }

  superpower(arena: Opponents): Opponents {
    return fireDestruction(arena);
  }
}
```

В классе присутствует новое свойство `flame_power` (мощность огня), а также определены абстрактные методы `attack`, `regenerate`, `superpower`, которые задаются функциями из отдельного модуля `fire_powers.ts` (приложение В).

ПРИЛОЖЕНИЕ В
(справочное)
Модуль fire_powers.ts

```
import {Opponents} from
"../../../../../asests/dom_elements/arena/opponents";

export function fireBall(arena: Opponents): Opponents {
    let result: Opponents;

    result = arena;
    result.enemy.health -= 280;
    result.you.chakra -= 100;
    result.enemy.chakra += 150;

    return result;
}

export function heartOfHell(arena: Opponents): Opponents {
    let result: Opponents;

    result = arena;
    result.you.health += 150;
    result.you.chakra -= 120;

    return result;
}

export function fireDestruction(arena: Opponents): Opponents {
    let result: Opponents;

    result = arena;
    result.enemy.health -= 600;
    result.you.chakra -= 450;
    result.enemy.chakra += 600;

    return result;
}
```

В данном модуле представлены способности ниндзя стихии огня, которые в дальнейшем будут использоваться на арене сражений, а пока что задействуются как функции внутри методов данного класса.

ПРИЛОЖЕНИЕ Г
(обязательное)
Подкласс fireNinja – класс «eyesClanNinja»

```
export class eyesClanNinja extends fireNinja {
  _eye_level: number;

  constructor(
    name: string,
    health: number,
    chakra: number,
    rank: Rank,
    organization: Organizations,
    village: Villages,
    appearance: string,
    arena_view: string,
    flame_power: number,
    eye_level: number,
  ) {
    super(name, health, chakra, rank, organization,
      village, appearance, arena_view, flame_power);
    this._eye_level = eye_level;
  }

  get eyeLevel(): number { return this._eye_level; }

  repeatEnemyAttack(arena: Opponents): Opponents {
    let result: Opponents;
    result = Action(arena);
    return result;
  }
}
```


ПРИЛОЖЕНИЕ Д
(обязательное)
Подкласс eyesClanNinja – особый ниндзя «sensorNinja»

```
export class sensorNinja extends eyesClanNinja {  
  
  constructor(  
    name: string,  
    health: number,  
    chakra: number,  
    rank: Rank,  
    organization: Organizations,  
    village: Villages,  
    appearance: string,  
    arena_view: string,  
    flame_power: number,  
    eye_level: number,  
  ) {  
    super(name, health, chakra, rank, organization, village,  
      appearance, arena_view, flame_power, eye_level);  
  }  
  
  evadeAttack(arena: Opponents): Opponents {  
    let result: Opponents;  
    result = Action(arena);  
    return result;  
  }  
}
```

Класс особого ниндзя будет иметь свои индивидуальные методы для сражений на арене.