Khmelnytskyi National University
Department of Computer Engineering and Information Systems

Report

Laboratory work No 6

Discipline: “Object Oriented Programming”

Completed: 2st year student, group SE-24-1 **Mamchur Danylo**

Checked: **Boiko Viacheslav**

Khmelnytskyi, 2025

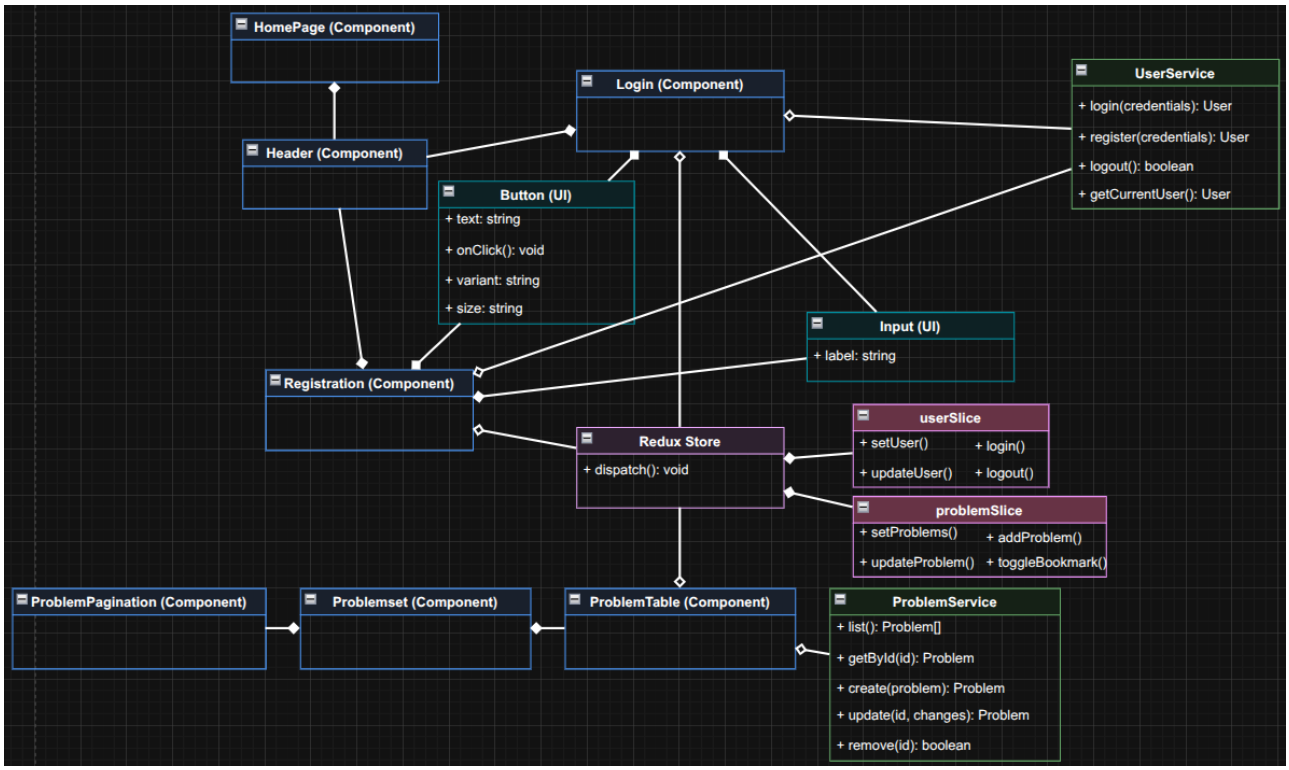
Laboratory work № 6 "DEPENDENCY MANAGEMENT"

Mamchur Danylo SE-24-1

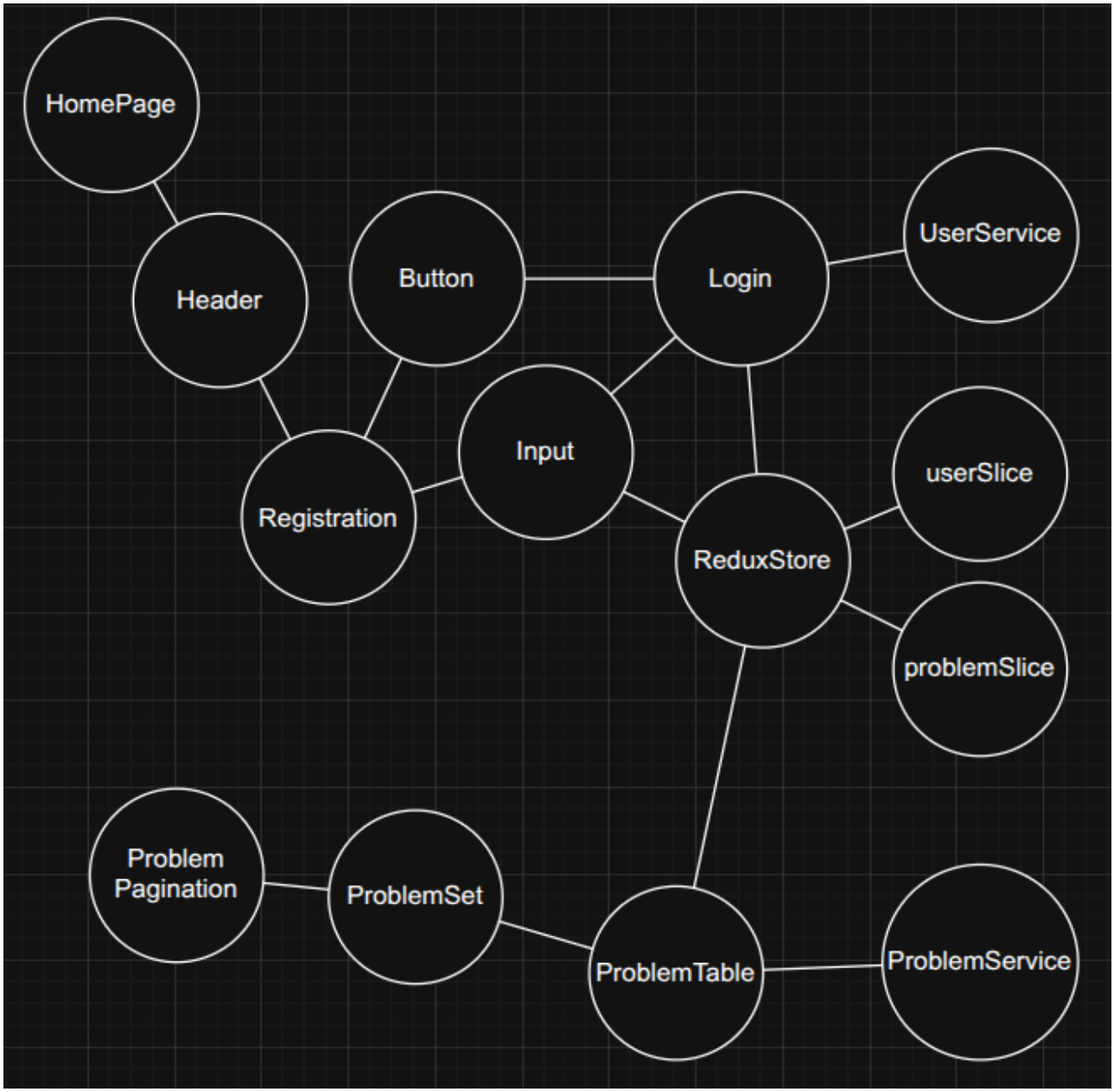
Purpose: provide knowledge and practice of how to handle dependencies in a way that makes code more modular, testable, and maintainable; understand how to apply dependency injection to decouple components; gain an experience with DI frameworks and best practices, which promote clean architecture principles.

Task execution

1. Class diagram w/ only composition and aggregation relations:



2. Clean dependency graph before decoupling:



3. The usage of three dependency management libraries:

```ts
import "reflect-metadata";
import { Container, Token } from "typedi";
import type { ProblemService } from "@/services/problem";
import { ProblemService as ProblemServiceClass } from "@/services/problem";
import type { UserService } from "@/services/user";
import { UserService as UserServiceClass } from "@/services/user";

const USER_TOKEN = new Token<UserService>("UserService");
const PROBLEM_TOKEN = new Token<ProblemService>("ProblemService");

// Using tokens and direct container access for this minimal demo

// Provide existing instances under tokens
Container.set(USER_TOKEN, new UserServiceClass());
Container.set(PROBLEM_TOKEN, new ProblemServiceClass());

export async function runTypeDiDemo(): Promise<string> {
  // Avoid constructor injection to keep demo resilient in bundler/dev mode
  const userService = Container.get(USER_TOKEN);
  const problemService = Container.get(PROBLEM_TOKEN);
  const user = await userService.getCurrentUser();
  const list = await problemService.list({ pageSize: 1 });
  const first = list.items[0]?.id ?? "-";
  return `user=${user?.username ?? ""}, auth=${Boolean(user?.isAuth)}, problems=${list.total}, first=${first}`;
}

export const TYPE_DI = {
  USER_TOKEN,
  PROBLEM_TOKEN,
  Container,
};

export function getTypeDiUserService() {
  return Container.get(USER_TOKEN);
}

export function getTypeDiProblemService() {
  return Container.get(PROBLEM_TOKEN);
}
```

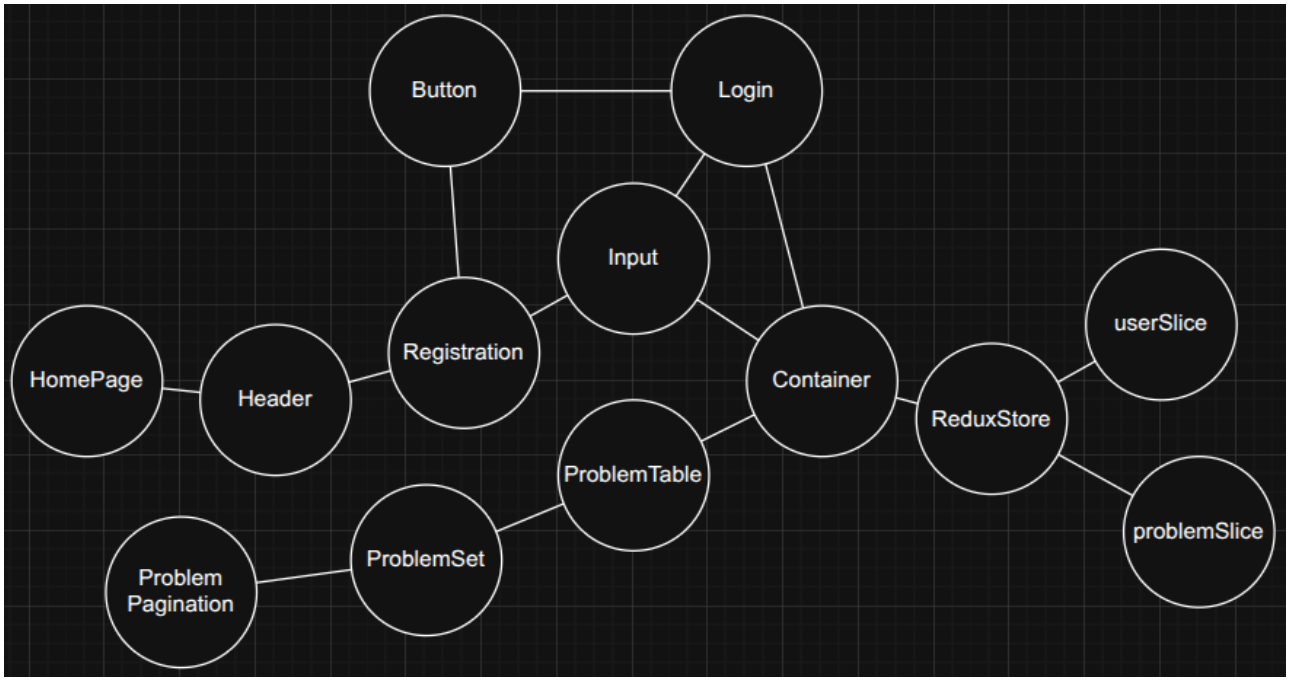
```ts
import type { UserService } from "@/services/user";
import { UserService as UserServiceClass } from "@/services/user";

const TOKENS = {
  UserService: "UserService",
  ProblemService: "ProblemService",
} as const;

@injectable()
class AppService {
  constructor(
    @inject(TOKENS.UserService) private readonly userService: UserService,
    @inject(TOKENS.ProblemService) private readonly problemService: ProblemService
  ) {}

  async summary(): Promise<string> {
    const user = await this.userService.getCurrentUser();
    const list = await this.problemService.list({ pageSize: 1 });
    const first = list.items[0]?.id ?? "-";
    return `user=${user?.username ?? ""}, auth=${Boolean(user?.isAuth)}, problems=${list.total}, first=${first}`;
  }
}

// Register existing instances so we don't have to decorate service classes
container.registerInstance<UserService>(TOKENS.UserService, new UserServiceClass());
container.registerInstance<ProblemService>(TOKENS.ProblemService, new ProblemServiceClass());

export async function runTsyringeDemo(): Promise<string> {
  const app = container.resolve(AppService);
  return app.summary();
}

export function getTsyringeUserService() {
  return container.resolve<UserService>(TOKENS.UserService as unknown as any);
}

export function getTsyringeProblemService() {
  return container.resolve<ProblemService>(TOKENS.ProblemService as unknown as any);
}
```

```ts
const TYPES = {
  UserService: Symbol.for("UserService"),
  ProblemService: Symbol.for("ProblemService"),
  AppService: Symbol.for("AppService"),
};

@injectable()
class AppService {
  constructor(
    @inject(TYPES.UserService) private readonly userService: UserService,
    @inject(TYPES.ProblemService) private readonly problemService: ProblemService
  ) {}

  async summary(): Promise<string> {
    const user = await this.userService.getCurrentUser();
    const list = await this.problemService.list({ pageSize: 1 });
    const first = list.items[0]?.id ?? "-";
    return `user=${user?.username ?? ""}, auth=${Boolean(user?.isAuth)}, problems=${list.total}, first=${first}`;
  }
}

// Create a local container and register existing service instances
const container = new Container({ defaultScope: "Singleton" });
container.bind<UserService>(TYPES.UserService).toConstantValue(new UserServiceClass());
container.bind<ProblemService>(TYPES.ProblemService).toConstantValue(new ProblemServiceClass());
container.bind<AppService>(TYPES.AppService).to(AppService);

export async function runInversifyDemo(): Promise<string> {
  const app = container.get<AppService>(TYPES.AppService);
  return app.summary();
}

export function getInversifyUserService() {
  return container.get<UserService>(TYPES.UserService);
}

export function getInversifyProblemService() {
  return container.get<ProblemService>(TYPES.ProblemService);
}
```

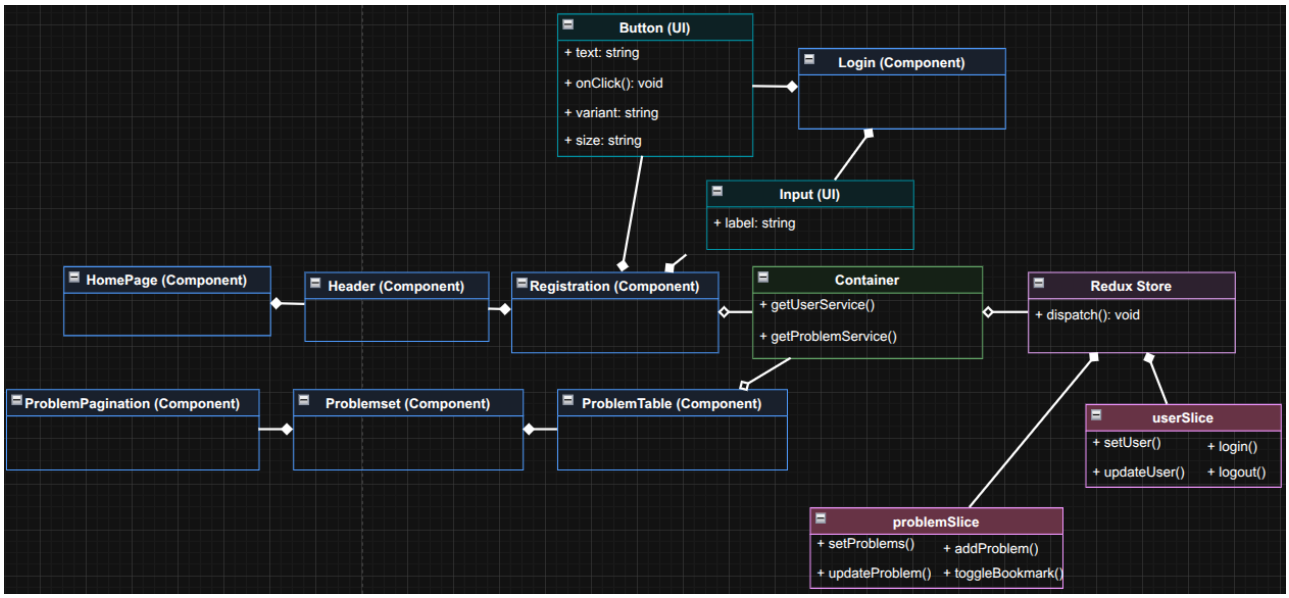
```ts
const onToggleBookmark = async (id: string) => {
  const problemService = getInversifyProblemService();
  await problemService.toggleBookmark(id);
  dispatch(toggleBookmark(id));
};
```

```ts
const onSubmit: SubmitHandler<LoginInputs> = async (data) => {
  const userService = getTsyringeUserService();
  const user = await userService.login({ username: data.nickname, password: data.password });
  dispatch(loginAction({ username: user.username, email: user.email, password: user.password }));
  if (!data.keepLoggedIn) reset();
  navigate("/");
};
```

4. Dependency diagram after applying the DI:



5. Class diagram w/ nly composition and aggragation relations (after applying the DI):



Questions

1. Dependency management is the practice of defining, versioning, wiring, and controlling the lifecycle of components and the external libraries they depend on to keep systems modular and reliable.
2. Loose coupling means components depend on abstractions and minimal knowledge of each other; high coupling is bad because small changes ripple widely, reducing testability and maintainability.
3. Unmanaged dependencies create hidden, hard-coded ties (e.g., new-ing services inside classes, global singletons), causing brittle code, version conflicts, and order-of-initialization bugs.
4. Instruments include DI frameworks/containers (InversifyJS, TSyringe, TypeDI), package managers (npm/yarn), build systems (Vite/Webpack), and module systems with semantic versioning.
5. Dependency injection is supplying a component's dependencies from the outside rather than creating them internally; types include constructor, setter/property, and method/parameter injection.
6. An IoC container is a runtime registry/factory that constructs objects, resolves their dependencies, and manages scopes/lifecycles to invert control from consumers to the framework.

7. A dependency graph is a directed graph of components and their dependency edges; a dependency tree is a rooted, acyclic view (often from a package manager) showing transitive dependencies per root.