KAUNO TECHNOLOGIJOS UNIVERSITETAS
FAKULTETAS

T120B516 Objektinis programų projektavimas

Projektinis darbas

Žaidimas „Laivų mūšis“

Atliko:

Šarūnas Palianskas

Matas Rutkauskas

Saulė Virbičianskaitė

Vilmantas Pieškus

Priėmė:

lekt. BARISAS Dominykas

lekt. VALINČIUS Kęstutis

KAUNAS 2022

Turinys

1 Projekto aprašymas

Projekto metu bus kuriama žaidimo „Laivų mūšis“ implementacija su papildomomis funkcijomis ir žaidimo galimybėmis, skirta žaisti dviems žaidėjams naudojant serverį komunikacijai tarp jų. Žaidimo metu kiekvienas iš žaidėjų turi žaidimo figūrėlių – laivų – rinkinį, kurį išdėsto žaidimo lentoje taip, jog kitas žaidėjas apie šį išdėstymą nežino. Tuomet žaidėjai paeiliui renkasi žaidimo lentos langelius, taip juos pažymėdami kaip atakuotus. Jei pasirinktame langelyje buvo vieno iš priešo laivų dalis, ši pažymima kaip pažeista, laivas, kurio visos dalys tampa pažeistos yra laikomas nuskandintu. Žaidimo tikslas yra nuskandinti visus priešo laivus greičiau nei tai padarys oponentas.

1.1 Žaidimo reikalavimai

1.1.1 Žaidimo lygiai

Įprastas

Šiame lygyje žaidėjai žaidžia klasikinę žaidimo versiją. Žaidimo lentos dydis yra 10x10 langelių, žaidėjai turi po penkias figurėles. Žaidėjai paeiliui renkasi lentos koordinates, kurias atakuoti, o po atakos matomi rezultatai – pataikyta ar ne.

Skirtingi amunicijos tipai

Šiame lygyje žaidimo figurėlių dalys turi gyvybės taškus, todėl laivas tampa nuskandintas tik kai visų laivo dalių gyvybės išsenka. Žaidimo metu taip pat naudojami skirtingi amunicijos tipai, kurių kievienas turi savų pranašumų ir trūkumų. Žaidimo lenta šiame lygyje yra 20x20 langelių.

Karo rūkas

Šiame lygyje žaidėjams suteikiama galimybė judinti figurėles pagal nustatytas judėjimo taisykles. Taip pat įvedamas apribojimas, jog atakuoti galima tik tuos lentos langelius, kurie patenka į žaidėjo nenuskandintų laivų matymo zoną. Žaidimo lenta šiame lygyje yra 30x30 langelių.

1.1.2 Žaidimo figūrėlės

Žaidėjai pradžioje turi po 5 figūrėlės

- Lėktuvnešis (5x1 dydžio)
  - Karo rūko lygyje šios figūrėlės matymo spindulys yra 7 langeliai.
- Drednoutas (4x1 dydžio)
  - Karo rūko lygyje šios figūrėlės matymo spindulys yra 5 langeliai.
- Kreiseris (3x1 dydžio)
  - Karo rūko lygyje šios figūrėlės matymo spindulys yra 4 langeliai.
- Povandeninis laivas (3x1 dydžio)
  - Karo rūko lygyje šios figūrėlės matymo spindulys yra 4 langeliai.
  - Skirtingų amunicijos tipų lygyje šią figurėlę galima pažeisti tik su sprogstamaisiais šoviniais
- Kateris (2x1 dydžio)
  - Karo rūko lygyje šios figūrėlės matymo spindulys yra 3 langeliai.
- Lėktuvas
  - Prieinamas tik karo rūko lygyje
  - Galima naudoti tol, kol žaidėjas turi nesunaikintą lėktuvnešio figurėlę
  - Ėjimo pradžioje figurėlės pozicija yra tokia pati kaip lėktuvnešio
  - Gali judėti į bet kurią lentos koordinatę 7 langelių spinduliu aplink lėktuvnešį
  - Matymo spindulys yra 4 langeliai

Žaidimo lygyje su skirtingais amunicijos tipais kievienas figūrėlės langelis turi 10 gyvybės taškų.

1.1.3 Amunicijos tipai

Žaidimo lygyje su skirtingais amunicijos tipais žaidėjas atakos metu gali rinktis iš trijų amunicijos tipų:

- Standartinis
    - Atakos plotas yra vienas langelis.
    - Daroma žala yra 4 gyvybės taškai, jei laivo dalis nepažeista, 3 taškai jei dalis pažeista.
- Sprogstamasis
    - Atakos plotas yra 3x3 langeliai, aplink pasirinktą atakos langelį.
    - Daroma žala yra 2 gyvybės taškai kiekviename langelyje.
- Anti-šarviniai
    - Atakos plotas yra vienas langelis.
    - Daroma žala yra 10 gyvybės taškų.
    - Panaudojus žaidėjas praleidžia kitą savo ėjimą.

1.1.4 Žaidimo figūrėlių judėjimas

Karo rūko žaidimo lygyje ėjimo metu žaidėjas gali ne tik atakuoti, bet tuo pat metu dar atlikti ir vieną judesio veiksmą kiekvienam iš laivų, kurių bendras gyvybių taškų kiekis yra bent pusė pilnų jo taškų.

Jūdėjimo veiksmai gali būti:

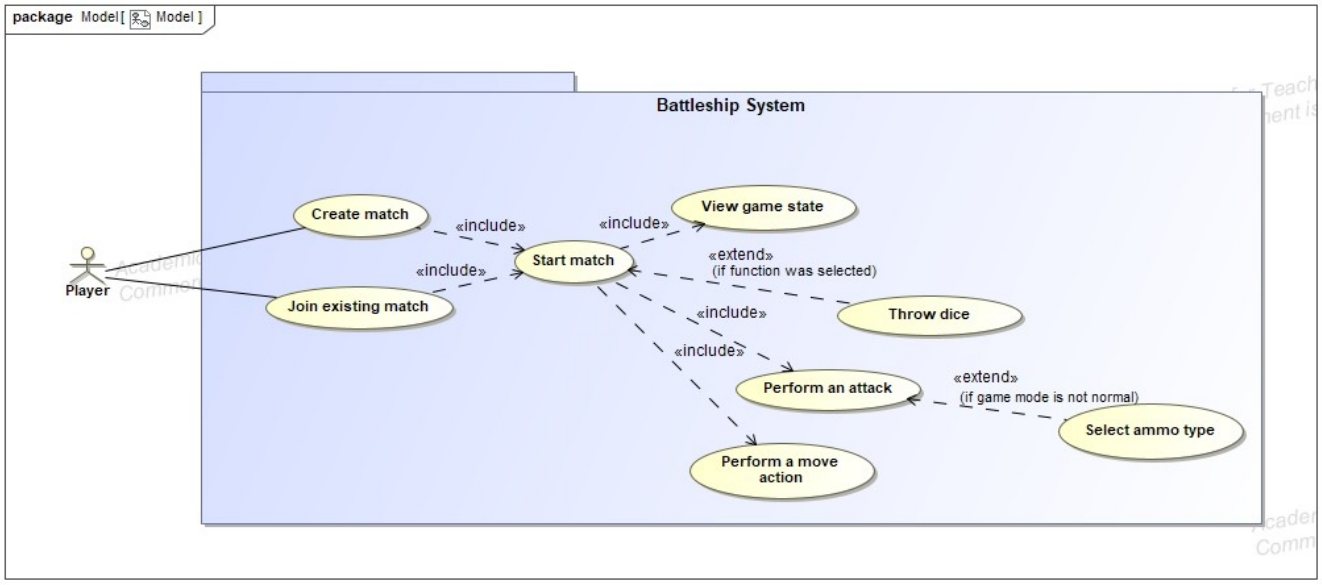
- Jūdėjimas į priekį
    - Laivo figūrėlė perstatoma vienu langeliu į priekį
- Pasisukimas
    - Laivo figūrėlė gali pasisukti 90, 180 ar 270 laipsnių.

Jūdėjimo veiksmus laivo figūrėlė gali atlikti tik tuomet, jei po veiksmo figūrėlės pozicija nesikirs su kitomis žaidėjo figūrėlėmis ar nebus už žaidimo lentos ribų.

1.1.5 Papildomos funkcijos

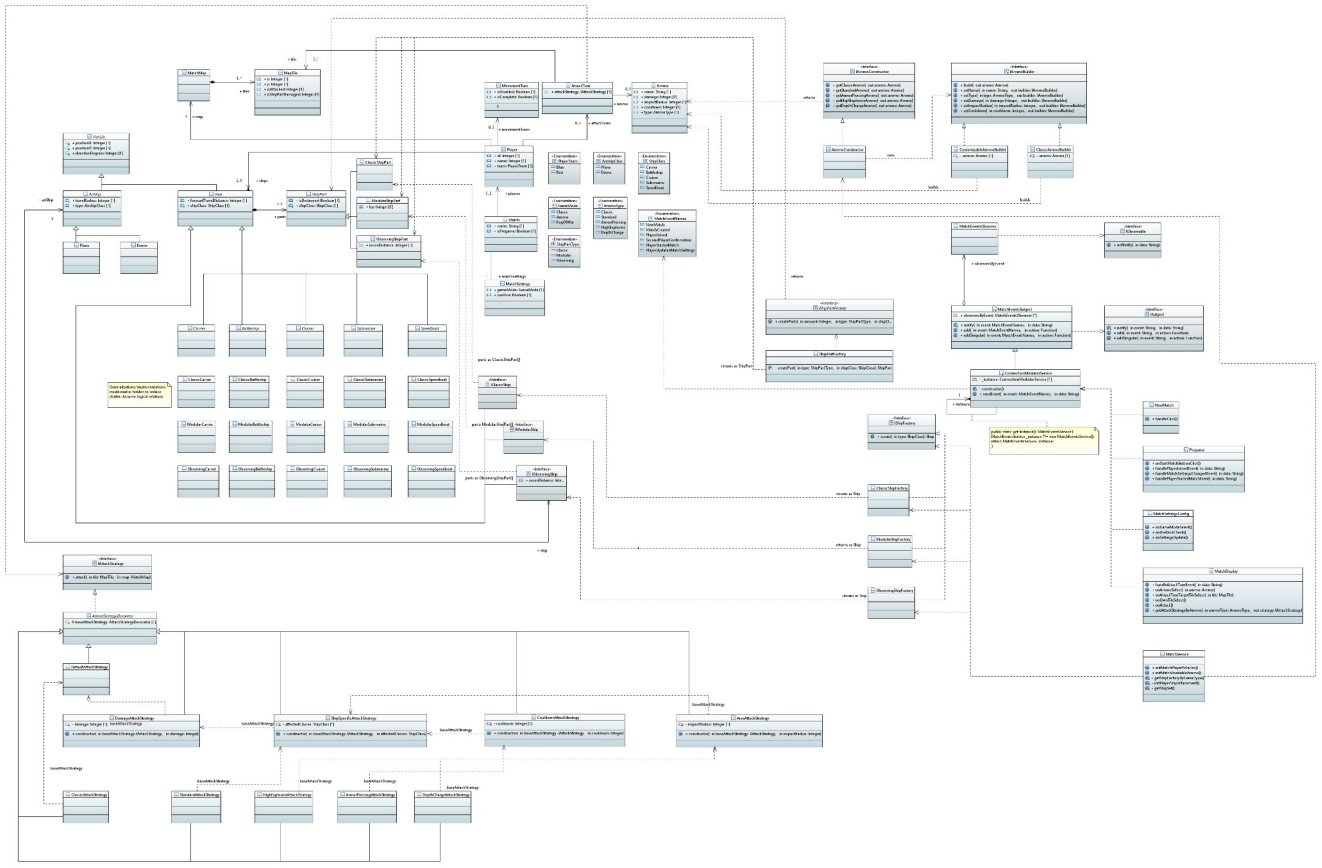
Žaidėjai gali pasirinkti žaisti naudojant žaidimo kauliuką. Tokiu atveju prieš ėjimą žaidėjas „meta“ kauliuką, o šis nurodo kiek papildomų ėjimų iš eilės žaidėjas gali atlikti. Kauliuko reikšmės gali būti nuo 0 iki 2. Žaidėjas prieš papildomus ėjimus kauliuko nebemeta.

1.2 Panaudojimo atvejų diagrama



Figūra 1: Žaidimo panaudojimo atvejų diagrama

1.3 Klasių diagrama



Figūra 2: Žaidimo klasių diagrama

2 Pritaikyti šablonai

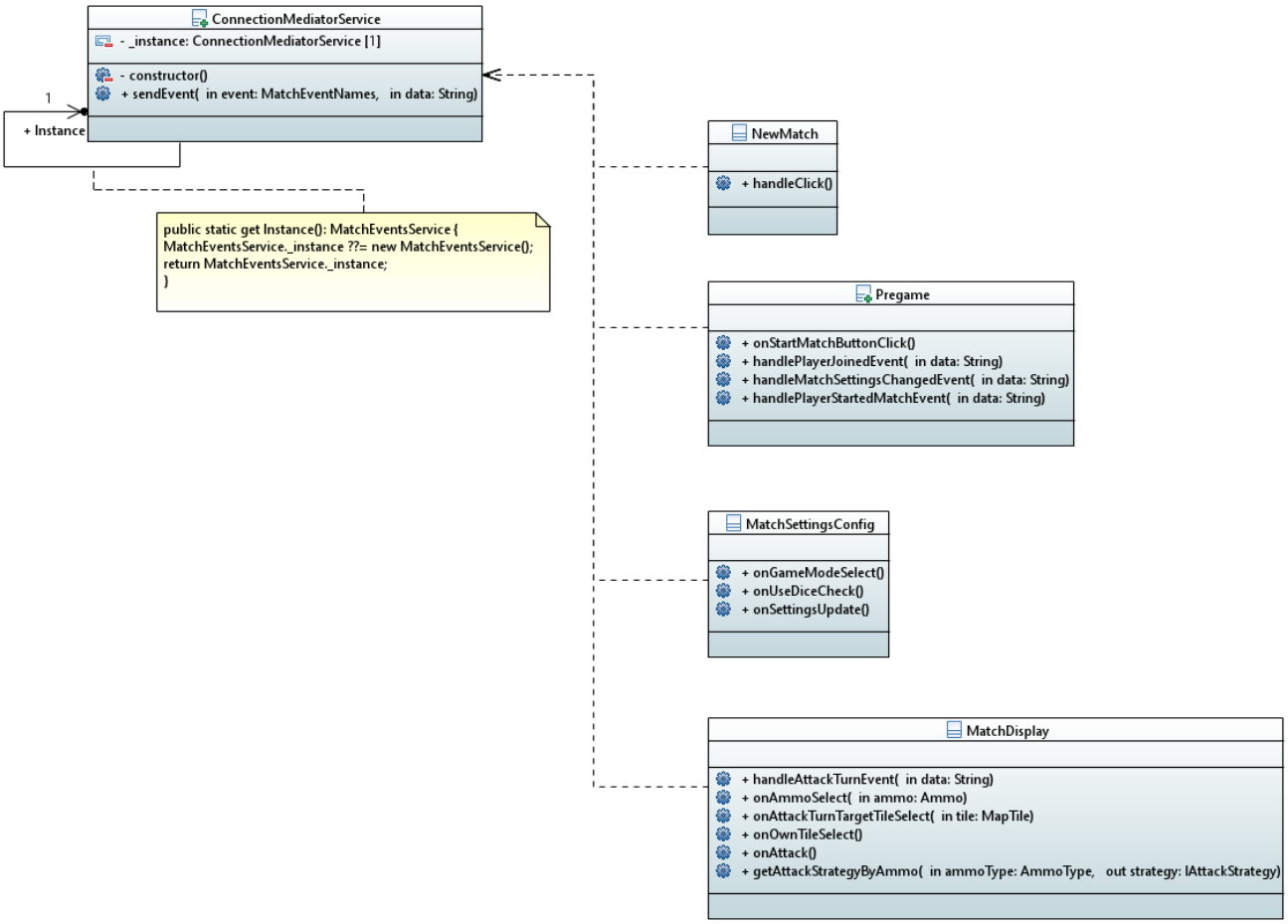
2.1 Singleton – ConnectionMediatorService

Atsakingas komandos narys: Šarūnas Palianskas

Programoje žaidėjų veiksmams koordinuoti naudojama *ConnectionMediatorService* kuri palaiko komunikaciją su serveriu naudojant SignalR. Dėl šios priežasties yra svarbu išlaikyti tik vieną šios klasės objektą programos veikimo metu.

Šablonas implementuotas *Typescript* kalba, todėl lenktynių situacija bandant gauti objektą nesusidarys dėl *Javascript* kalbos naršyklėje naudojamo variklio kuris naudoja vieną giją.

Šablonas buvo taikomas nuo pradžių, todėl klasių diagramos prieš taikymą nėra.



Figūra 3: Singleton šablono klasių diagrama

Šablono kodo fragmentas:

```
export default class ConnectionMediatorService extends MatchEventsSubject {
  private static _instance: ConnectionMediatorService;
  // other props and methods
  private constructor() {
    super();
    // connection init
  }
```

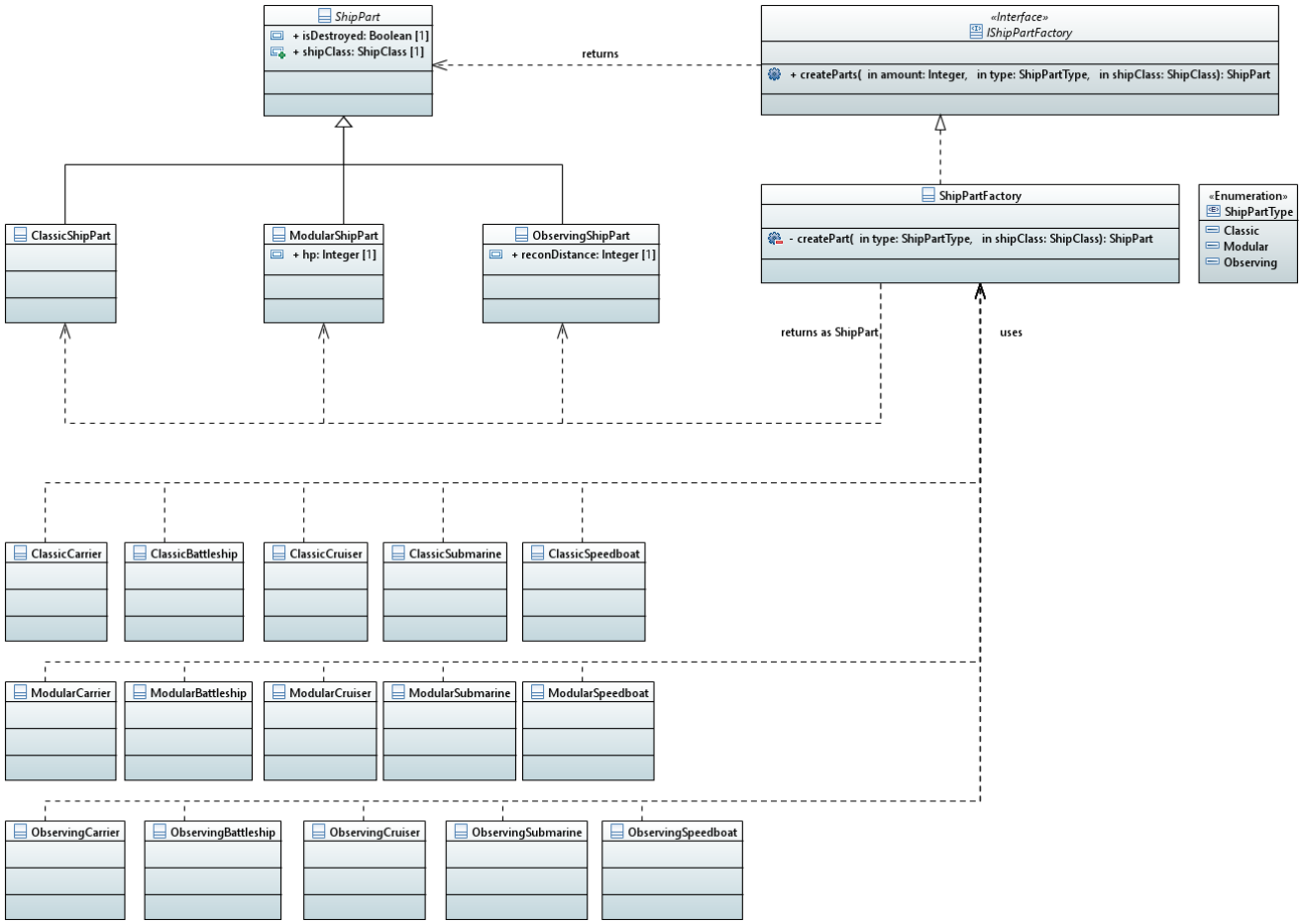
```
  public static get Instance(): ConnectionMediatorService {
    ConnectionMediatorService._instance ??= new ConnectionMediatorService();
    return ConnectionMediatorService._instance;
  }
}
```

2.2 Factory – ShipPartFactory

Atsakingas komandos narys: Šarūnas Palianskas

Laivo modeliai yra sudaryti iš keleto laivo dalių. Šioms dalims kurti naudojamas *ShipPartFactory*.

Šablonas buvo taikomas nuo pradžių, todėl klasių diagramos prieš taikymą nėra.



Figūra 4: Factory šablono klasių diagrama

Šablono kodo fragmentas:

```
export enum ShipPartType {
  Classic,
  Modular,
  Observing,
}
```

```
export interface IShipPartFactory {
  createParts(
    amount: number,
    type: ShipPartType,
    shipClass: ShipClass
  ): ShipPart[];
}

export default class ShipPartFactory implements IShipPartFactory {
  createParts(
    amount: number,
    type: ShipPartType,
    shipClass: ShipClass
  ): ShipPart[] {
    const result: ShipPart[] = [];

    for (let i = 0; i < amount; i++) {
      result.push(this.createPart(type, shipClass));
    }

    return result;
  }

  private createPart(type: ShipPartType, shipClass: ShipClass) {
    switch (type) {
      case ShipPartType.Classic: {
        return new ClassicShipPart(shipClass);
      }
      case ShipPartType.Modular: {
        return new ModularShipPart(shipClass);
      }
      case ShipPartType.Observing: {
        return new ObservingShipPart(shipClass);
      }
    }
  }
}

export abstract class ShipPart {
  constructor(shipClass: ShipClass) {
```

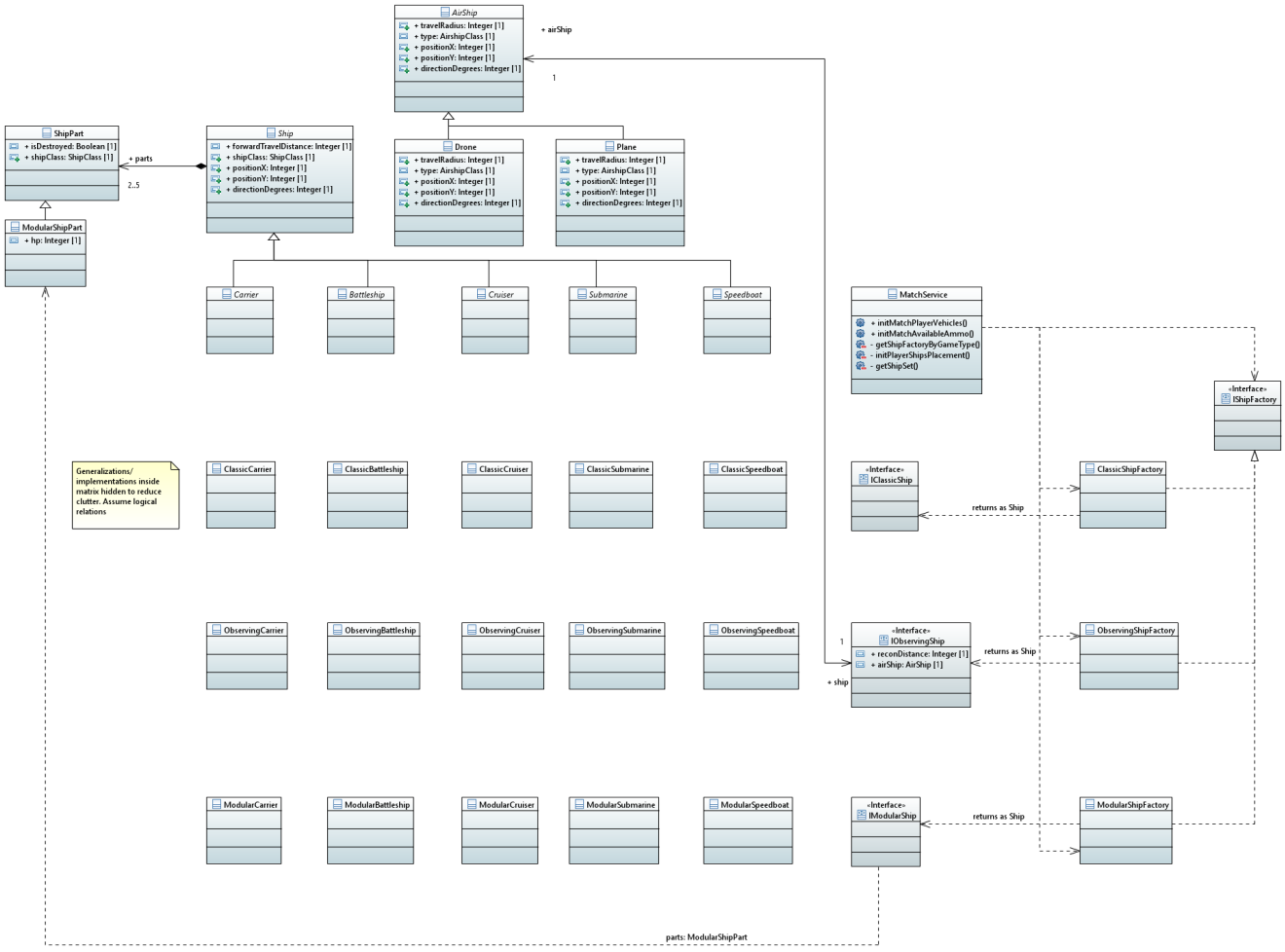
```
    this.shipClass = shipClass;
  }

  isDestroyed = false;
  shipClass: ShipClass;
}

export class ClassicShipPart extends ShipPart {}

export class ModularShipPart extends ShipPart {
  hp = 10;
}

export class ObservingShipPart extends ShipPart {
  reconDistance!: number;
}
```

2.3 Abstract factory – ShipFactory

Atsakingas komandos narys: Šarūnas Palianskas

Laivo modeliai programoje gali būti keleto skirtingų klasių (pvz. lėktuvnešis), taip pat laivai gali būti skirtingų tipų priklausomai nuo žaidimo tipo (pvz. turintys laivo dalis, kurios turi gyvybės taškus – nėra nuskandinamos iš karto į jas pataikius). Tokiu atveju programoje realizuotas *Abstract factory* šablonas laivų kūrimui.

Šablonas buvo taikomas nuo pradžių, todėl klasių diagramos prieš taikymą nėra.



Figūra 5: Abstract factory šablono klasių diagrama

Šablono kodo fragmentas:

```
export interface IShipFactory {
  create(type: ShipClass): Ship;
}

export default class ClassicShipFactory implements IShipFactory {
  // other props and methods

  create(type: ShipClass): IClassicShip {
    switch (type) {
      case ShipClass.Carrier: {
        return new ClassicCarrier();
      }
      case ShipClass.Battleship: {
        return new ClassicBattleship();
      }
```

```
      case ShipClass.Cruiser: {
        return new ClassicCruiser();
      }
      case ShipClass.Submarine: {
        return new ClassicSubmarine();
      }
      case ShipClass.Speedboat: {
        return new ClassicSpeedboat();
      }
    }
  }
}

export default class ObservingShipFactory implements IShipFactory {
  // other props and methods

  create(type: ShipClass): IObservingShip {
    switch (type) {
      case ShipClass.Carrier: {
        return new ObservingCarrier();
      }
      case ShipClass.Battleship: {
        return new ObservingBattleship();
      }
      case ShipClass.Cruiser: {
        return new ObservingCruiser();
      }
      case ShipClass.Submarine: {
        return new ObservingSubmarine();
      }
      case ShipClass.Speedboat: {
        return new ObservingSpeedboat();
      }
    }
  }
}

export default class ModularShipFactory implements IShipFactory {
  // other props and methods
```

```
  create(type: ShipClass): IModularShip {
    switch (type) {
      case ShipClass.Carrier: {
        return new ModularCarrier();
      }
      case ShipClass.Battleship: {
        return new ModularBattleship();
      }
      case ShipClass.Cruiser: {
        return new ModularCruiser();
      }
      case ShipClass.Submarine: {
        return new ModularSubmarine();
      }
      case ShipClass.Speedboat: {
        return new ModularSpeedboat();
      }
    }
  }
}

export class MatchService {
// other props and methods
  static initMatchPlayerVehicles(): void {
    const match = MatchProvider.Instance.match;

    const factory = this.getShipFactoryByGameType(match);

    match.players.forEach((player) => {
      player.ships = this.getShipSet(factory);
    });
  }

  private static getShipFactoryByGameType(match: Match): IShipFactory {
    switch (match.settings.gameMode) {
      case GameMode.Classic: {
        return ClassicShipFactory.Instance;
      }
```

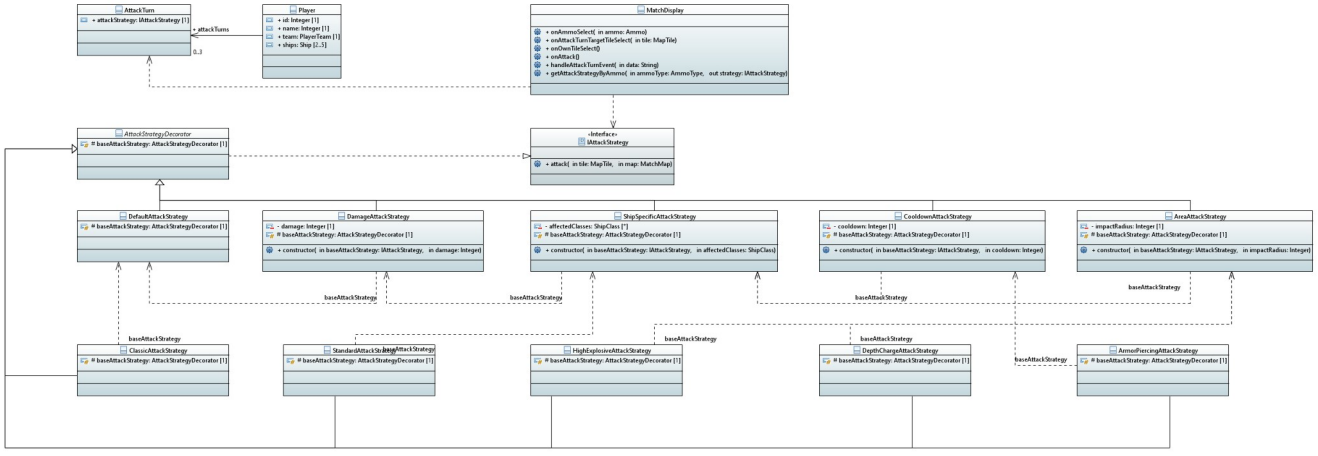
```
      case GameMode.Ammo: {
        return ModularShipFactory.Instance;
      }
      case GameMode.FogOfWar: {
        return ObservingShipFactory.Instance;
      }
    }
  }

  private static getShipSet(factory: IShipFactory): Ship[] {
    return [
      factory.create(ShipClass.Carrier),
      factory.create(ShipClass.Battleship),
      factory.create(ShipClass.Cruiser),
      factory.create(ShipClass.Submarine),
      factory.create(ShipClass.Speedboat),
    ];
  }
}
```

2.4 Strategy – AttackStrategy

Atsakingas komandos narys: Šarūnas Palianskas

Programoje naudojami keletas skirtingų amunicijos tipų, nuo kurių priklauso kaip turėtų būti vykdoma ataka (pvz. sprogstamoji amunicija daro žalą ir aplink esančiuose langeliuose). Žaidėjui pasirinkus atakos ėjimo amuniciją programos veikimo metu nustatoma, kokį atakos algoritmą naudoti. Kartu su *Strategy* šablonu panaudotas ir *Decorator* šablonas.

Šablonas buvo taikomas nuo pradžių, todėl klasių diagramos prieš taikymą nėra.



Figūra 6: Strategy šablono klasių diagrama

Šablono kodo fragmentas:

```
export interface IAttackStrategy {
  attack(tile: MapTile, map: MatchMap): void;
}

export abstract class AttackStrategyDecorator implements IAttackStrategy {
  protected baseAttackStrategy?: AttackStrategyDecorator;

  abstract attack(tile: MapTile, map: MatchMap): void;
}

export class DefaultAttackStrategy extends AttackStrategyDecorator {
  attack(tile: MapTile, map: MatchMap): void {
    tile.isAttacked = true;

    console.log(`default attack on ${tile.x}-${tile.y}`);
  }
}

export class DamageAttackStrategy extends AttackStrategyDecorator {
  private damage: number;

  constructor(baseAttackStrategy: IAttackStrategy, damage: number) {
    super();

    this.baseAttackStrategy = baseAttackStrategy;
    this.damage = damage;
  }

  attack(tile: MapTile, map: MatchMap): void {
    this.baseAttackStrategy!.attack(tile, map);

    if (!!tile.shipPart) {
      const shipPart = tile.shipPart as ModularShipPart;

      shipPart.hp -= this.damage;

      if (shipPart.hp <= 0) {
```

```
        shipPart.hp = 0;
        shipPart.isDestroyed = true;
        tile.isShipPartDestroyed = true;
      }
    }

    console.log(`damage(${this.damage}) attack on ${tile.x}-${tile.y}`);
  }
}

export class AreaAttackStrategy extends AttackStrategyDecorator {
  private impactRadius: number;

  constructor(baseAttackStrategy: IAttackStrategy, impactRadius: number) {
    super();

    this.baseAttackStrategy = baseAttackStrategy;
    this.impactRadius = impactRadius;
  }

  attack(tile: MapTile, map: MatchMap): void {
    for (
      let i = tile.x - (this.impactRadius - 1);
      i < tile.x + this.impactRadius;
      i++
    ) {
      for (
        let j = tile.y - (this.impactRadius - 1);
        j < tile.y + this.impactRadius;
        j++
      ) {
        if (i < 0 || i > map.tiles.length || j < 0 || j > map.tiles.length) {
          continue;
        }

        const tile = map.tiles[i][j];
        this.baseAttackStrategy!.attack(tile, map);
        console.log(`area(${this.impactRadius}) attack on ${tile.x}-${tile.y}`);
      }
```

```
    }
  }
}

export class CooldownAttackStrategy extends AttackStrategyDecorator {
  private cooldown: number;

  constructor(baseAttackStrategy: IAttackStrategy, cooldown: number) {
    super();

    this.baseAttackStrategy = baseAttackStrategy;
    this.cooldown = cooldown;
  }

  attack(tile: MapTile, map: MatchMap): void {
    this.baseAttackStrategy!.attack(tile, map);

    const player = MatchProvider.Instance.match.players[0];

    if (player.attackTurns.length > this.cooldown) {
      player.attackTurns.reverse().splice(0, this.cooldown).reverse();
    } else {
      player.turnOverDraw += this.cooldown;
    }

    console.log(`cooldown(${this.cooldown}) attack on ${tile.x}-${tile.y}`);
  }
}

export class ShipSpecificAttackStrategy extends AttackStrategyDecorator {
  private affectedClasses: ShipClass[];

  constructor(
    baseAttackStrategy: IAttackStrategy,
    affectedClasses: ShipClass[]
  ) {
    super();

    this.baseAttackStrategy = baseAttackStrategy;
```

```
    this.affectedClasses = affectedClasses;
  }

  attack(tile: MapTile, map: MatchMap): void {
    if (
      !tile.shipPart ||
      this.affectedClasses.includes(tile.shipPart.shipClass)
    ) {
      this.baseAttackStrategy!.attack(tile, map);

      console.log(
        `ship specific(${this.affectedClasses}) attack on ${tile.x}-${tile.y}`
      );
    }
  }
}

export default function MatchDisplay() {
  function handleAttackTurnEvent(data: any): void {
    // other statements
    turn.attackStrategy = getAttackStrategyByAmmo(ammoType);
    turn.attackStrategy.attack(mapTile, defencePlayer!.map);
    // other statements
  }

  function getAttackStrategyByAmmo(ammoType: AmmoType): IAttackStrategy {
    const ammo = match.availableAmmoTypes.find(
      (ammo) => ammo.type === ammoType
    );
    switch (ammoType) {
      case AmmoType.Classic:
        return new ClassicAttackStrategy();
      case AmmoType.Standard:
        return new StandardAttackStrategy(ammo!.damage);
      case AmmoType.ArmorPiercing:
        return new ArmorPiercingAttackStrategy(ammo!.cooldown, ammo!.damage);
      case AmmoType.HighExplosive:
        return new HighExplosiveAttackStrategy(
          ammo!.damage,
```

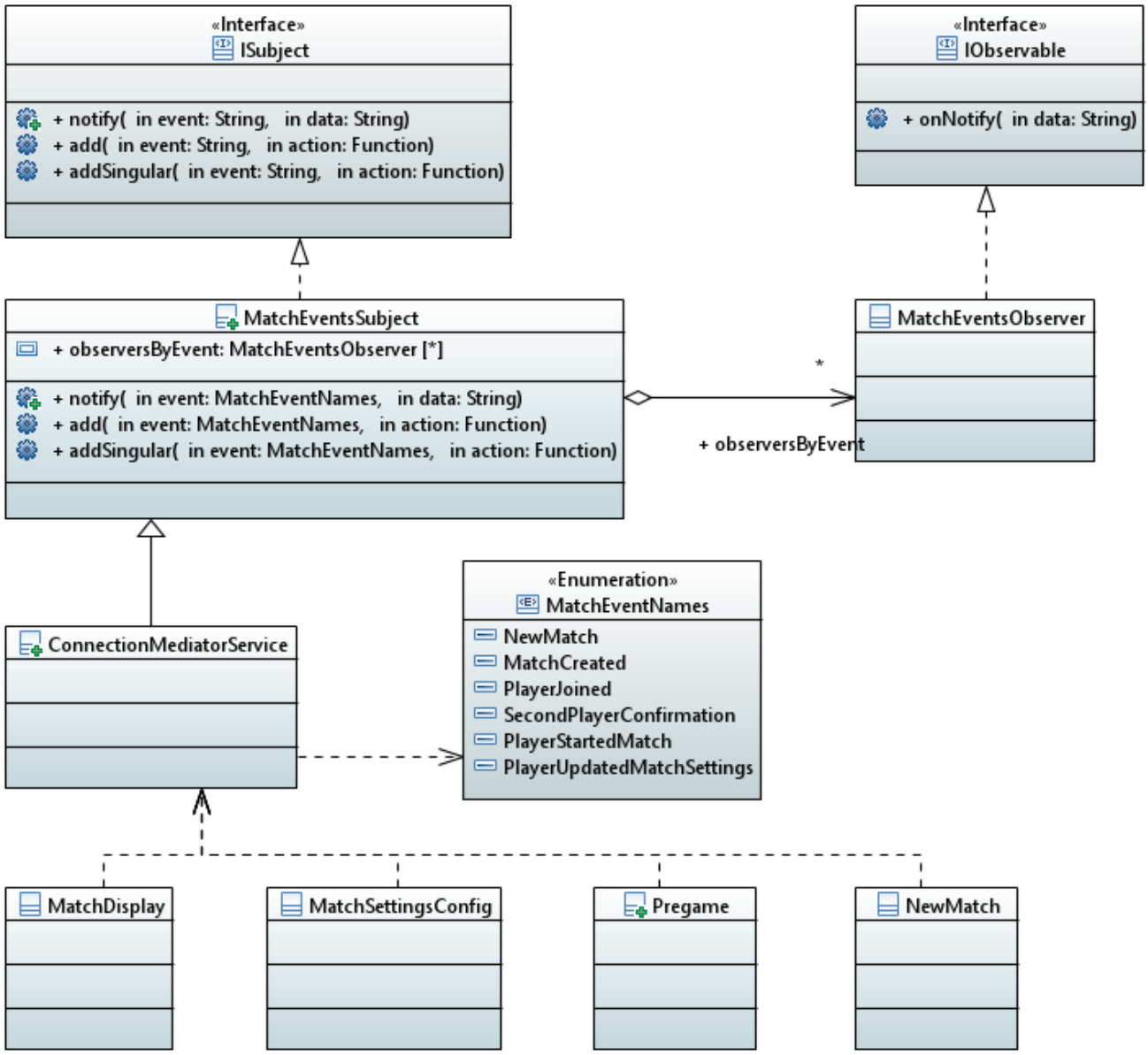
```
          ammo!.impactRadius
        );
      case AmmoType.DepthCharge:
        return new DepthChargeAttackStrategy(ammo!.damage, ammo!.impactRadius);
    }
  }
}
```

2.5 Observer – MatchEventsSubject/MatchEventsObserver

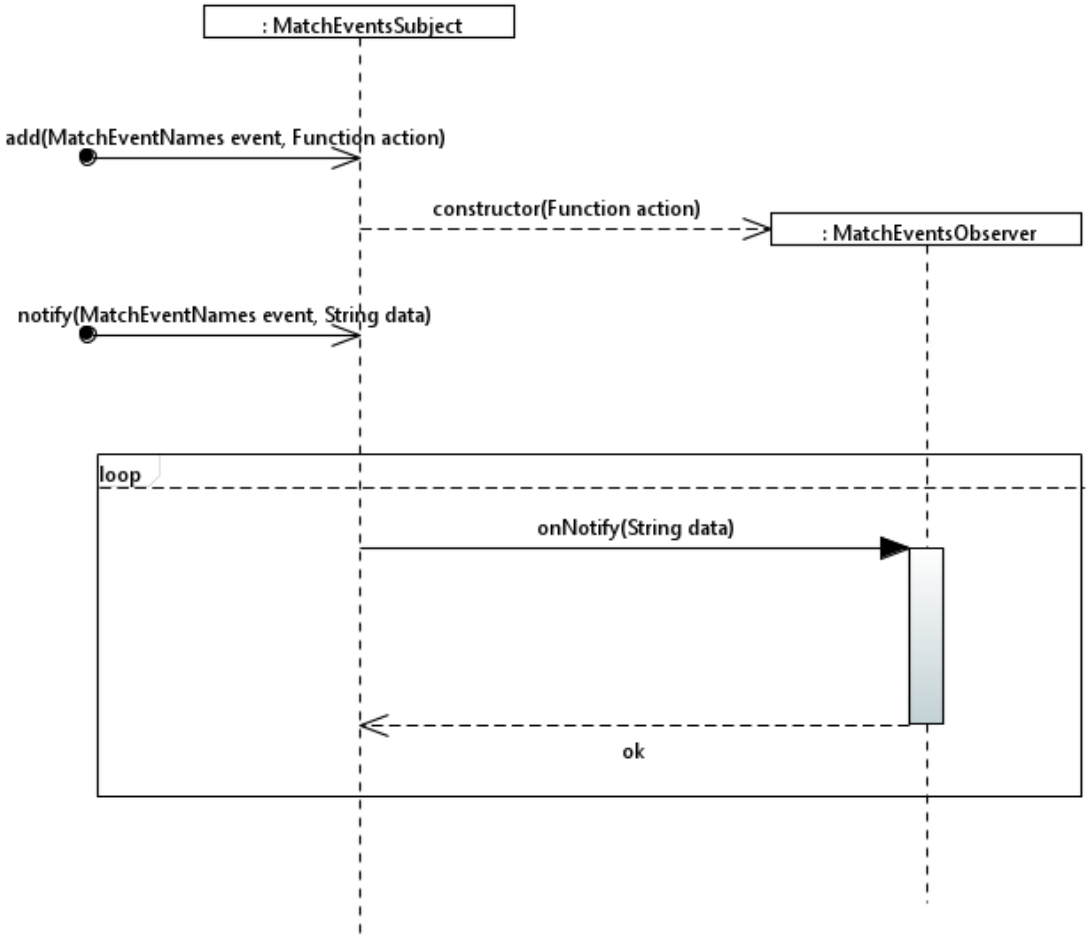
Atsakingas komandos narys: Šarūnas Palianskas

Programoje naudojama įvykiais paremta architektūra. *Observer* šablonas panaudotas norint priskirti *callback'us* įvykiams.

Šablonas buvo taikomas nuo pradžių, todėl klasių diagramos prieš taikymą nėra.



Figūra 7: Observer šablono klasių diagrama



Figūra 8: Observer šablono sekų diagrama

Šablono kodo fragmentas:

```
export interface ISubject {
  notify(event: any, data: any): void;
  add(event: any, action: Function): void;
  addSingular(event: any, action: Function): void;
}

export default class MatchEventsSubject implements ISubject {
  protected observersByEvent: { [event: number]: MatchEventsObservable[] } = {};

  constructor() {
    for (const event in MatchEventNames) {
      // iterator contains `number` keys, then `string` values
      if (isNaN(Number(event))) {
        break;
      }
```

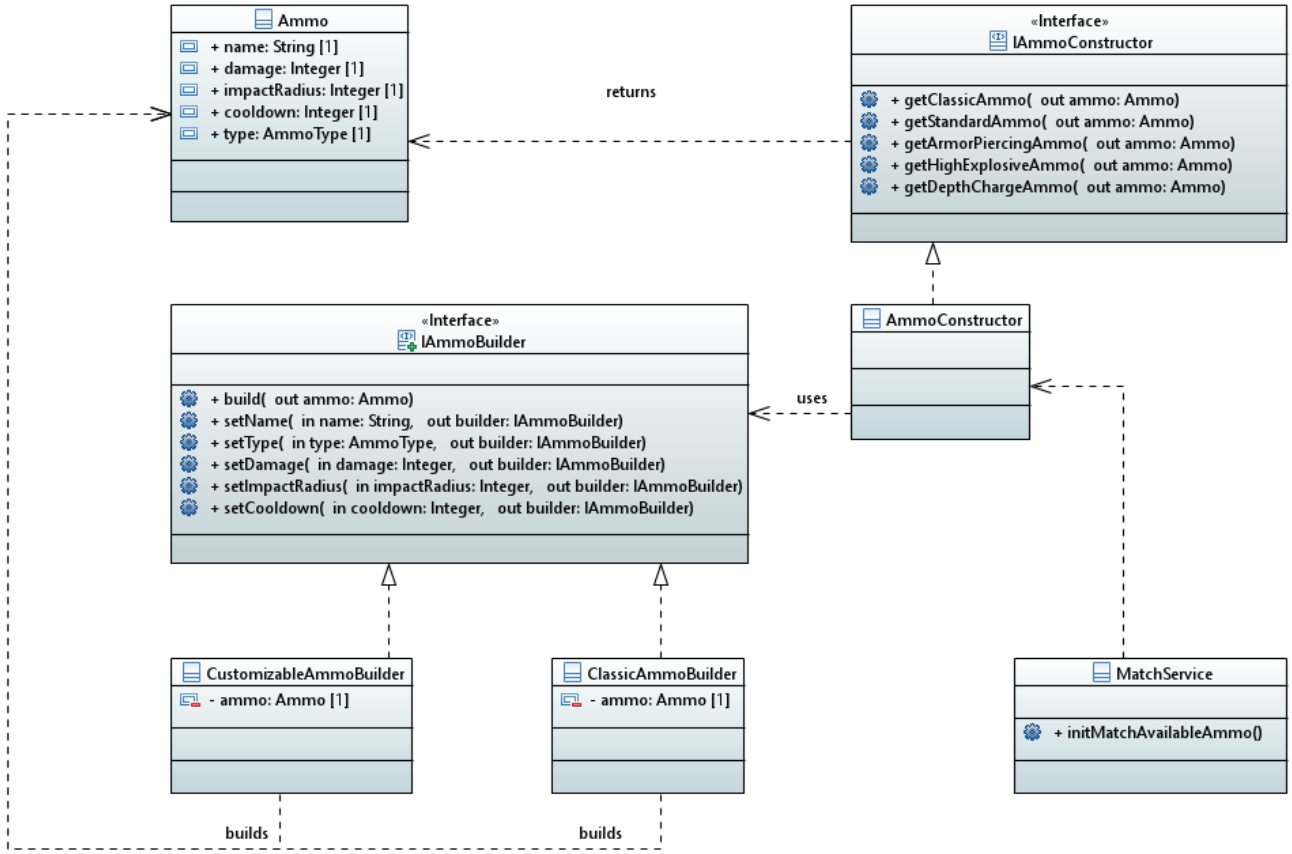
```
      this.observersByEvent[event] = [];
    }
  }

  public notify(event: MatchEventNames, data: any): void {
    const eventObservers = this.observersByEvent[event];

    eventObservers?.forEach((observer) => observer.onNotify(data));
  }

  public add(event: MatchEventNames, action: (data: string) => void) {
    this.observersByEvent[event].push(new MatchEventsObservable(action));
  }

  public addSingular(event: MatchEventNames, action: (data: string) => void) {
    if (this.observersByEvent[event].length === 0) {
      this.observersByEvent[event].push(new MatchEventsObservable(action));
    }
  }
}

export interface IObservable {
  onNotify(data: string): void;
}

export default class MatchEventsObservable implements IObservable {
  constructor(onNotify: (data: string) => void) {
    this.onNotify = onNotify;
  }

  onNotify: (data: string) => void;
}

export default class ConnectionMediatorService extends MatchEventsSubject {
  // other props and methods
}
```

2.6 Builder – AmmoBuilder

Atsakingas komandos narys: Šarūnas Palianskas

Programoje naudojama keletas skirtingų amunicijos tipų kurie turi nustatytus parametrus (pvz. žalos kiekis). Amunicijos modelių kūrimui panaudotas *Builder* šablonas kuris supaprastina modelių kūrima.

Šablonas buvo taikomas nuo pradžių, todėl klasių diagramos prieš taikymą nėra.



Figūra 9: Builder šablono klasių diagrama

Šablono kodo fragmentas:

```
export interface IAmmoBuilder {
  build(): Ammo;
  setName(name: string): IAmmoBuilder;
  setType(type: AmmoType): IAmmoBuilder;
  setDamage(damage: number): IAmmoBuilder;
  setImpactRadius(impactRadius: number): IAmmoBuilder;
  setCooldown(cooldown: number): IAmmoBuilder;
}

export class ClassicAmmoBuilder implements IAmmoBuilder {
  private ammo = new Ammo();

  constructor() {
    this.ammo.cooldown = 0;
```

```
    this.ammo.damage = 1;
    this.ammo.impactRadius = 1;
    this.ammo.type = AmmoType.Classic;
  }

  build(): Ammo {
    return this.ammo;
  }
  setName(name: string): IAmmoBuilder {
    this.ammo.name = name;

    return this;
  }

  setType(): IAmmoBuilder {
    return this;
  }
  setDamage(): IAmmoBuilder {
    return this;
  }
  setImpactRadius(): IAmmoBuilder {
    return this;
  }
  setCooldown(): IAmmoBuilder {
    return this;
  }
}

export class CustomizableAmmoBuilder implements IAmmoBuilder {
  private ammo = new Ammo();

  build(): Ammo {
    return this.ammo;
  }
  setName(name: string): IAmmoBuilder {
    this.ammo.name = name;

    return this;
  }
```

```
  setType(type: AmmoType): IAmmoBuilder {
    this.ammo.type = type;

    return this;
  }
  setDamage(damage: number): IAmmoBuilder {
    this.ammo.damage = damage;

    return this;
  }
  setImpactRadius(impactRadius: number): IAmmoBuilder {
    this.ammo.impactRadius = impactRadius;

    return this;
  }
  setCooldown(cooldown: number): IAmmoBuilder {
    this.ammo.cooldown = cooldown;

    return this;
  }
}

export interface IAmmoConstructor {
  getClassicAmmo(): Ammo;
  getStandardAmmo(): Ammo;
  getArmorPiercingAmmo(): Ammo;
  getHighExplosiveAmmo(): Ammo;
  getDepthChargeAmmo(): Ammo;
}

export class AmmoConstructor implements IAmmoConstructor {
  getClassicAmmo(): Ammo {
    const builder: IAmmoBuilder = new ClassicAmmoBuilder();

    return builder.setName('Classic').build();
  }

  getStandardAmmo(): Ammo {
    const builder: IAmmoBuilder = new CustomizableAmmoBuilder();
```

```
  const damage = 3;
  const impactRadius = 1;
  const cooldown = 0;

  return builder
    .setName('Standard')
    .setType(AmmoType.Standard)
    .setDamage(damage)
    .setImpactRadius(impactRadius)
    .setCooldown(cooldown)
    .build();
}

getArmorPiercingAmmo(): Ammo {
  const builder: IAmmoBuilder = new CustomizableAmmoBuilder();

  const damage = 10;
  const impactRadius = 1;
  const cooldown = 1;

  return builder
    .setName('Armor Piercing')
    .setType(AmmoType.ArmorPiercing)
    .setDamage(damage)
    .setImpactRadius(impactRadius)
    .setCooldown(cooldown)
    .build();
}
getHighExplosiveAmmo(): Ammo {
  const builder: IAmmoBuilder = new CustomizableAmmoBuilder();

  const damage = 2;
  const impactRadius = 2;
  const cooldown = 0;

  return builder
    .setName('High Explosive')
    .setType(AmmoType.HighExplosive)
```

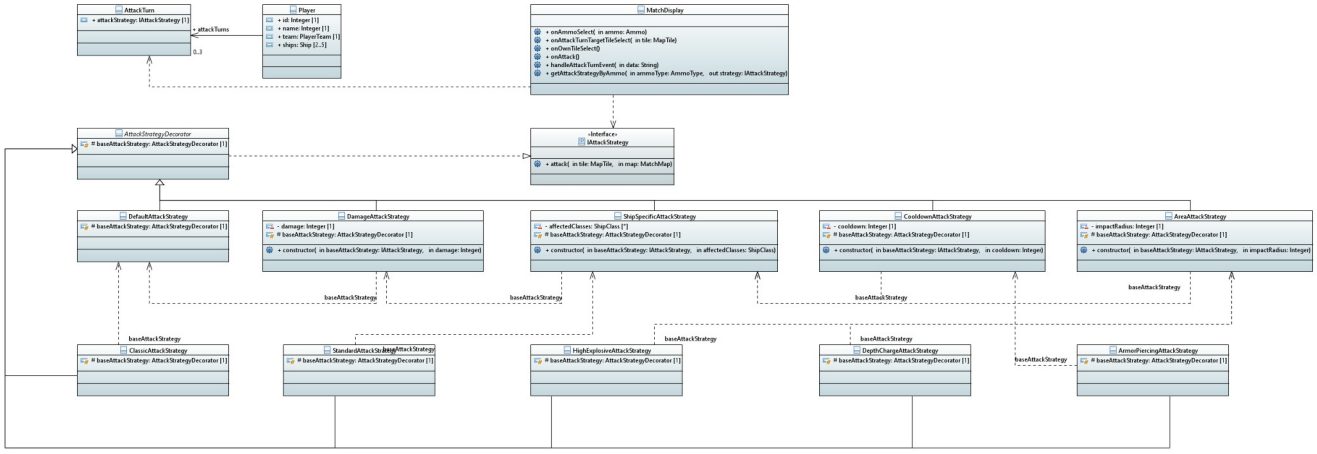
```
      .setDamage(damage)
      .setImpactRadius(impactRadius)
      .setCooldown(cooldown)
      .build();
  }
  getDepthChargeAmmo(): Ammo {
    const builder: IAmmoBuilder = new CustomizableAmmoBuilder();

    const damage = 4;
    const impactRadius = 2;
    const cooldown = 0;

    return builder
      .setName('Depth Charge')
      .setType(AmmoType.DepthCharge)
      .setDamage(damage)
      .setImpactRadius(impactRadius)
      .setCooldown(cooldown)
      .build();
  }
}

export class MatchService {
  // other props and methods
  static initMatchAvailableAmmo(): void {
    const match = MatchProvider.Instance.match;

    const ammoConstructor = new AmmoConstructor();

    if (match.settings.gameMode == GameMode.Ammo) {
      match.availableAmmoTypes.push(ammoConstructor.getStandardAmmo());
      match.availableAmmoTypes.push(ammoConstructor.getArmorPiercingAmmo());
      match.availableAmmoTypes.push(ammoConstructor.getHighExplosiveAmmo());
      match.availableAmmoTypes.push(ammoConstructor.getDepthChargeAmmo());
    } else {
      match.availableAmmoTypes.push(ammoConstructor.getClassicAmmo());
    }
  }
}
```

2.7 Decorator – AttackStrategyDecorator

Atsakingas komandos narys: Šarūnas Palianskas

Programoje naudojamos atakos strategijos turi tarpusavyje persidengiančių dalių (pvz. standartinė ir sprogramosios amunicijos atakos daro žalą veikiamuose langeliuose). Pasikartojančios dalys buvo išskirstos į atskiras klases, tokiu būdu leidžiant jas lengvai pakartotinai naudoti be kodo kartojimo.

Šablonas buvo taikomas nuo pradžių, todėl klasių diagramos prieš taikymą nėra.



Figūra 10: Decorator šablono klasių diagrama

Šablono kodo fragmentas:

```
export interface IAttackStrategy {
  attack(tile: MapTile, map: MatchMap): void;
}

export abstract class AttackStrategyDecorator implements IAttackStrategy {
  protected baseAttackStrategy?: AttackStrategyDecorator;

  abstract attack(tile: MapTile, map: MatchMap): void;
}

export class DefaultAttackStrategy extends AttackStrategyDecorator {
  attack(tile: MapTile, map: MatchMap): void {
    tile.isAttacked = true;

    console.log(`default attack on ${tile.x}-${tile.y}`);
  }
}
```

```
export class DamageAttackStrategy extends AttackStrategyDecorator {
  private damage: number;

  constructor(baseAttackStrategy: IAttackStrategy, damage: number) {
    super();

    this.baseAttackStrategy = baseAttackStrategy;
    this.damage = damage;
  }

  attack(tile: MapTile, map: MatchMap): void {
    this.baseAttackStrategy!.attack(tile, map);

    if (!!tile.shipPart) {
      const shipPart = tile.shipPart as ModularShipPart;

      shipPart.hp -= this.damage;

      if (shipPart.hp <= 0) {
        shipPart.hp = 0;
        shipPart.isDestroyed = true;
        tile.isShipPartDestroyed = true;
      }
    }

    console.log(`damage(${this.damage}) attack on ${tile.x}-${tile.y}`);
  }
}

export class AreaAttackStrategy extends AttackStrategyDecorator {
  private impactRadius: number;

  constructor(baseAttackStrategy: IAttackStrategy, impactRadius: number) {
    super();

    this.baseAttackStrategy = baseAttackStrategy;
    this.impactRadius = impactRadius;
  }
```

```
  attack(tile: MapTile, map: MatchMap): void {
    for (
      let i = tile.x - (this.impactRadius - 1);
      i < tile.x + this.impactRadius;
      i++
    ) {
      for (
        let j = tile.y - (this.impactRadius - 1);
        j < tile.y + this.impactRadius;
        j++
      ) {
        if (i < 0 || i > map.tiles.length || j < 0 || j > map.tiles.length) {
          continue;
        }

        const tile = map.tiles[i][j];
        this.baseAttackStrategy!.attack(tile, map);
        console.log(`area(${this.impactRadius}) attack on ${tile.x}-${tile.y}`);
      }
    }
  }
}

export class CooldownAttackStrategy extends AttackStrategyDecorator {
  private cooldown: number;

  constructor(baseAttackStrategy: IAttackStrategy, cooldown: number) {
    super();

    this.baseAttackStrategy = baseAttackStrategy;
    this.cooldown = cooldown;
  }

  attack(tile: MapTile, map: MatchMap): void {
    this.baseAttackStrategy!.attack(tile, map);

    const player = MatchProvider.Instance.match.players[0];

    if (player.attackTurns.length > this.cooldown) {
```

```
      player.attackTurns.reverse().splice(0, this.cooldown).reverse();
    } else {
      player.turnOverDraw += this.cooldown;
    }

    console.log(`cooldown(${this.cooldown}) attack on ${tile.x}-${tile.y}`);
  }
}

export class ShipSpecificAttackStrategy extends AttackStrategyDecorator {
  private affectedClasses: ShipClass[];

  constructor(
    baseAttackStrategy: IAttackStrategy,
    affectedClasses: ShipClass[]
  ) {
    super();

    this.baseAttackStrategy = baseAttackStrategy;
    this.affectedClasses = affectedClasses;
  }

  attack(tile: MapTile, map: MatchMap): void {
    if (
      !tile.shipPart ||
      this.affectedClasses.includes(tile.shipPart.shipClass)
    ) {
      this.baseAttackStrategy!.attack(tile, map);

      console.log(
        `ship specific(${this.affectedClasses}) attack on ${tile.x}-${tile.y}`
      );
    }
  }
}

export class ClassicAttackStrategy extends AttackStrategyDecorator {
  constructor() {
    super();
```

```
    this.baseAttackStrategy = new DefaultAttackStrategy();
  }

  attack(tile: MapTile, map: MatchMap): void {
    this.baseAttackStrategy!.attack(tile, map);

    if (!!tile.shipPart) {
      tile.shipPart.isDestroyed = true;
      tile.isShipPartDestroyed = true;
    }

    console.log(`classic attack on ${tile.x}-${tile.y}`);
  }
}

export class StandardAttackStrategy extends AttackStrategyDecorator {
  constructor(damage: number) {
    super();

    const defaultAttackStrategy = new DefaultAttackStrategy();
    const damageAttackStrategy = new DamageAttackStrategy(
      defaultAttackStrategy,
      damage
    );

    const affectedClasses = [
      ShipClass.Carrier,
      ShipClass.Battleship,
      ShipClass.Cruiser,
      ShipClass.Speedboat,
    ];
    const shipSpecificAttackStrategy = new ShipSpecificAttackStrategy(
      damageAttackStrategy,
      affectedClasses
    );

    this.baseAttackStrategy = shipSpecificAttackStrategy;
  }
```

```
  attack(tile: MapTile, map: MatchMap): void {
    this.baseAttackStrategy!.attack(tile, map);
    console.log(`standard attack on ${tile.x}-${tile.y}`);
  }
}

export class HighExplosiveAttackStrategy extends AttackStrategyDecorator {
  constructor(damage: number, impactRadius: number) {
    super();

    const defaultAttackStrategy = new DefaultAttackStrategy();
    const damageAttackStrategy = new DamageAttackStrategy(
      defaultAttackStrategy,
      damage
    );

    const affectedClasses = [
      ShipClass.Carrier,
      ShipClass.Battleship,
      ShipClass.Cruiser,
      ShipClass.Speedboat,
    ];
    const shipSpecificAttackStrategy = new ShipSpecificAttackStrategy(
      damageAttackStrategy,
      affectedClasses
    );
    const areaAttackStrategy = new AreaAttackStrategy(
      shipSpecificAttackStrategy,
      impactRadius
    );

    this.baseAttackStrategy = areaAttackStrategy;
  }

  attack(tile: MapTile, map: MatchMap): void {
    this.baseAttackStrategy!.attack(tile, map);
    console.log(`high explosive attack on ${tile.x}-${tile.y}`);
  }
```

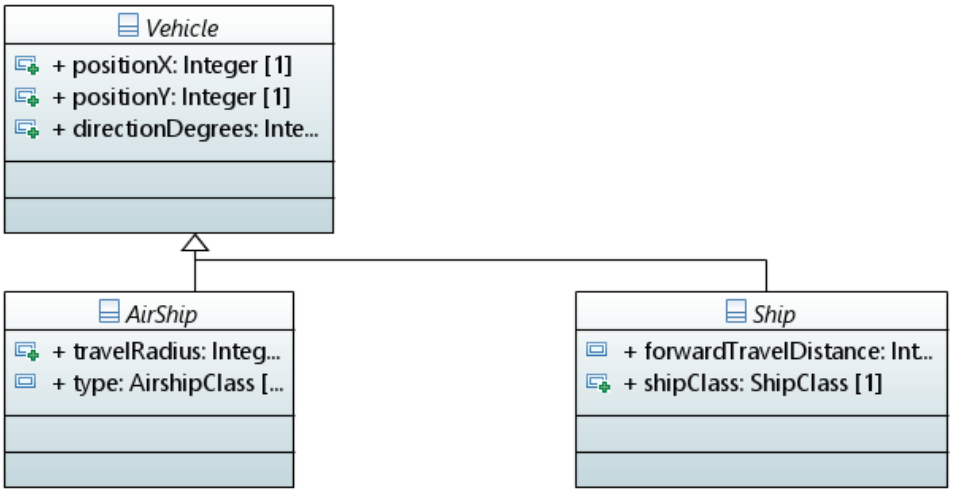
```
}

export class DepthChargeAttackStrategy extends AttackStrategyDecorator {
  constructor(damage: number, impactRadius: number) {
    super();

    const defaultAttackStrategy = new DefaultAttackStrategy();
    const damageAttackStrategy = new DamageAttackStrategy(
      defaultAttackStrategy,
      damage
    );

    const affectedClasses = [ShipClass.Submarine];
    const shipSpecificAttackStrategy = new ShipSpecificAttackStrategy(
      damageAttackStrategy,
      affectedClasses
    );
    const areaAttackStrategy = new AreaAttackStrategy(
      shipSpecificAttackStrategy,
      impactRadius
    );

    this.baseAttackStrategy = areaAttackStrategy;
  }

  attack(tile: MapTile, map: MatchMap): void {
    this.baseAttackStrategy!.attack(tile, map);
    console.log(`depth charge attack on ${tile.x}-${tile.y}`);
  }
}

export class ArmorPiercingAttackStrategy extends AttackStrategyDecorator {
  constructor(cooldown: number, damage: number) {
    super();

    const defaultAttackStrategy = new DefaultAttackStrategy();
    const damageAttackStrategy = new DamageAttackStrategy(
      defaultAttackStrategy,
      damage
```

```
    );

    const affectedClasses = [
      ShipClass.Carrier,
      ShipClass.Battleship,
      ShipClass.Cruiser,
      ShipClass.Speedboat,
    ];
    const shipSpecificAttackStrategy = new ShipSpecificAttackStrategy(
      damageAttackStrategy,
      affectedClasses
    );
    const cooldownAttackStrategy = new CooldownAttackStrategy(
      shipSpecificAttackStrategy,
      cooldown
    );

    this.baseAttackStrategy = cooldownAttackStrategy;
  }

  attack(tile: MapTile, map: MatchMap): void {
    this.baseAttackStrategy!.attack(tile, map);
    console.log(`armor piercing attack on ${tile.x}-${tile.y}`);
  }
}
```

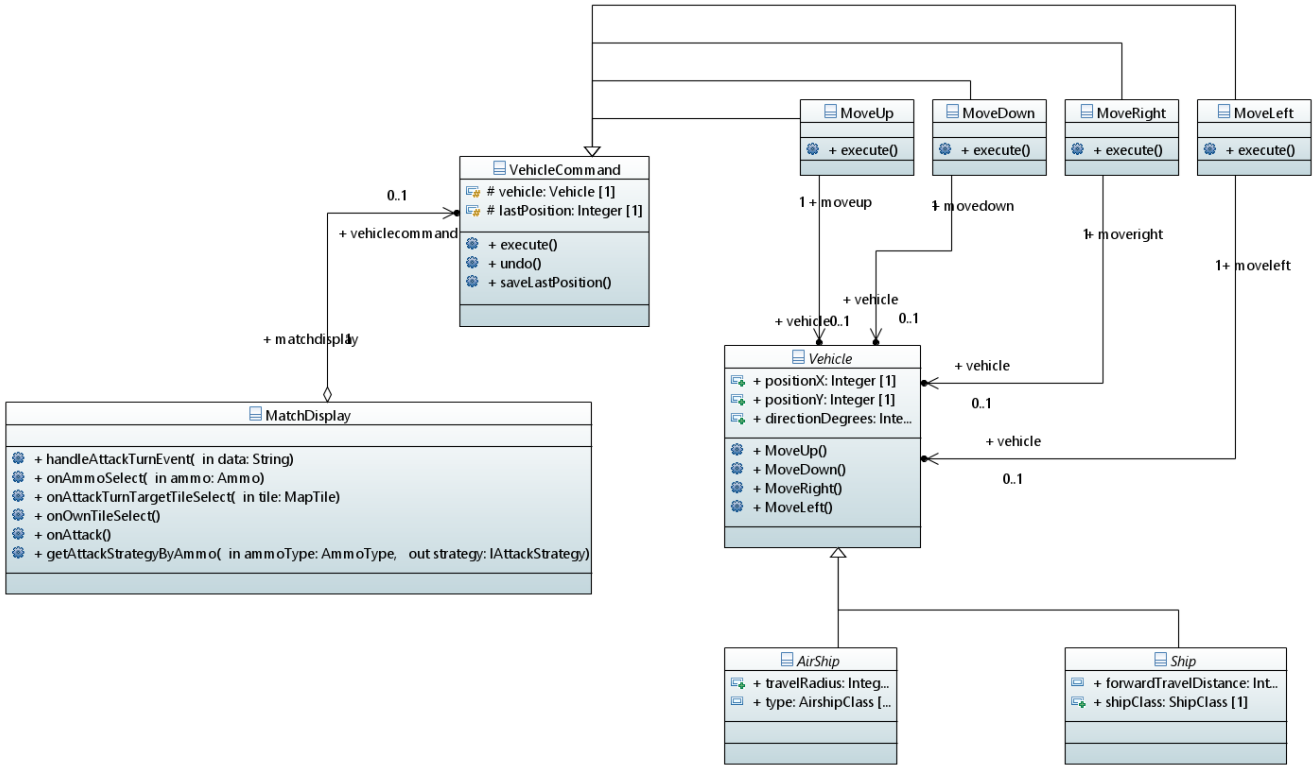
2.8 Command – VehicleCommand

Atsakinga komandos narė: Saulė Virbičianskaitė

Command šablonas tinka laivų statymui, kuomet žaidėjas pasirinktą laivą galės statyti taip, kad prieš tai buvusi pozicija būtų įsiminta sistemos.



Figūra 11: Klasių diagramos fragmentas iki šablono implementacijos



Figūra 12: Command šablono klasių diagrama

Šablono kodo projektas:

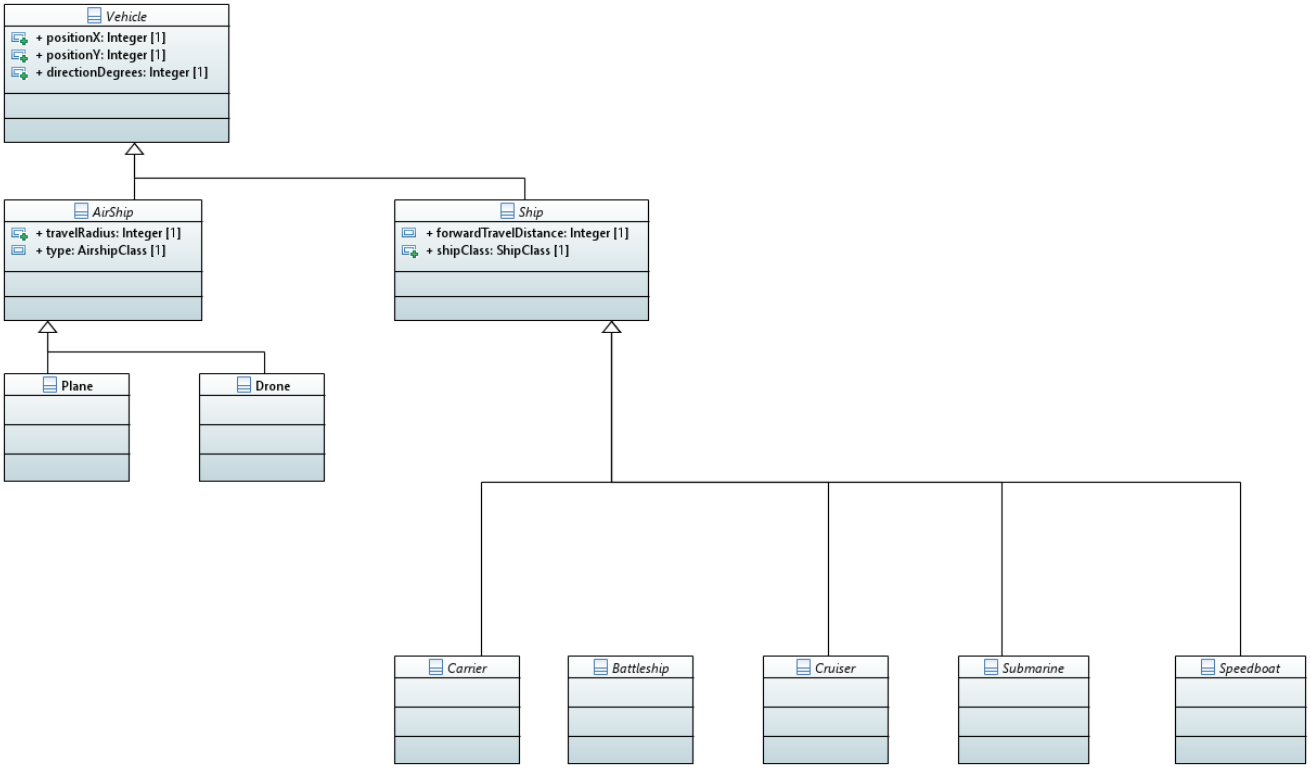
```
abstract class VehicleCommand {
    protected vehicle: Vehicle;
    protected lastPosition: { positionX: number, positionY: number }

    protected constructor(vehicle: Vehicle) {
        this.vehicle = vehicle;
        this.lastPosition = { positionX: vehicle.positionX, positionY: vehicle.po-
sitionY }
    }
```

```
    saveLastPosition() {
        this.lastPosition = { positionX: this.vehicle.positionX, positionY: thi-
s.vehicle.positionY }
    }

    undo() {
        this.vehicle.positionX = this.lastPosition.positionX;
        this.vehicle.positionY = this.lastPosition.positionY;
    }

    abstract execute(): Vehicle;
}

class MoveUp extends VehicleCommand {
    execute() {
        this.vehicle.MoveUp
        return this.vehicle
    }
}

class MoveDown extends VehicleCommand {
    execute() {
        this.vehicle.MoveDown
        return this.vehicle
    }
}

class MoveRight extends VehicleCommand {
    execute() {
        this.vehicle.MoveRight
        return this.vehicle
    }
}

class MoveLeft extends VehicleCommand {
    execute() {
        this.vehicle.MoveLeft
        return this.vehicle
    }

 }
```

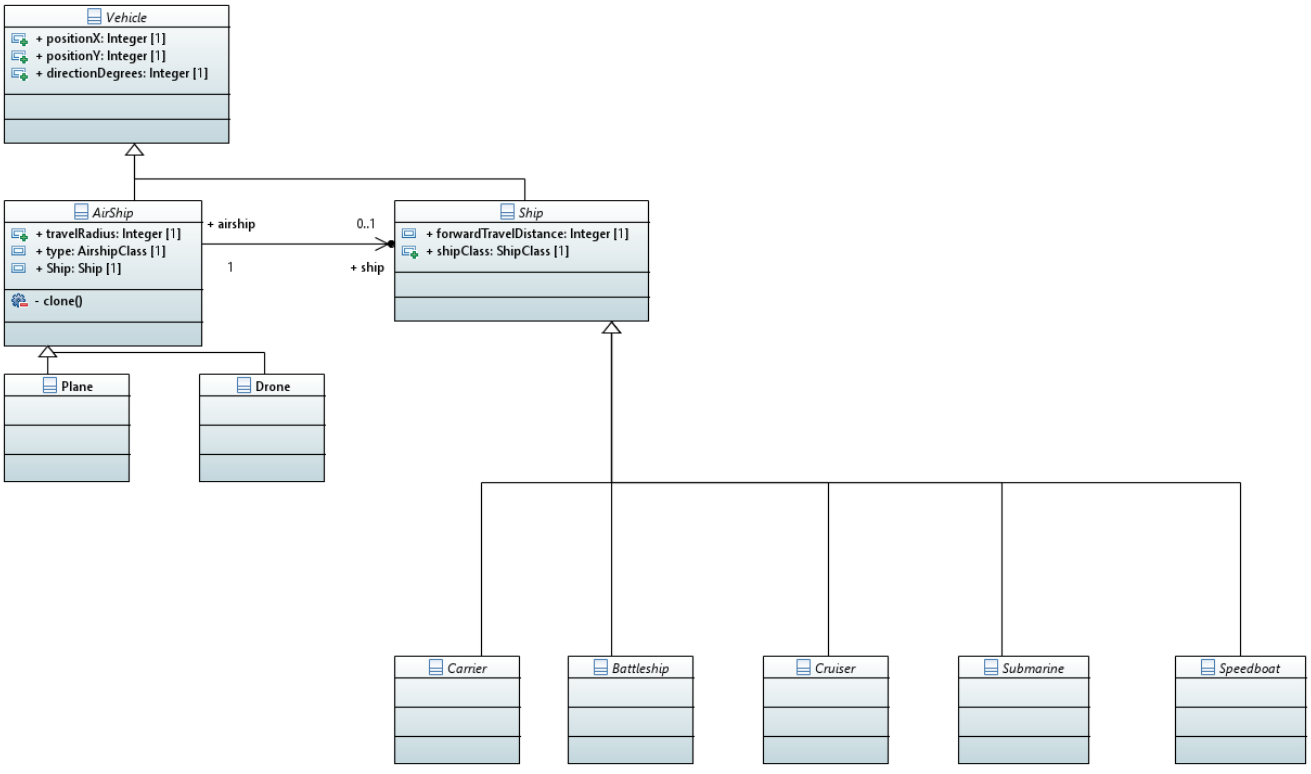
2.9 Bridge – Airship

Atsakingas komandos narys – Vilmantas Pieškus

Bridge šablonas buvo panaudotas, kad atskirti abstrakciją nuo implementacijos, kad galėtume atnaujinti kodą nepriklausomai nuo vienas kito. Dabar galime Ship koordinačių savybes pritaikyti prie visų Airship objektų.



Figūra 13: Klasių diagramos fragmentas iki šablono implementacijos



Figūra 14: Bridge šablono klasių diagrama

Šablono kodo projektas:

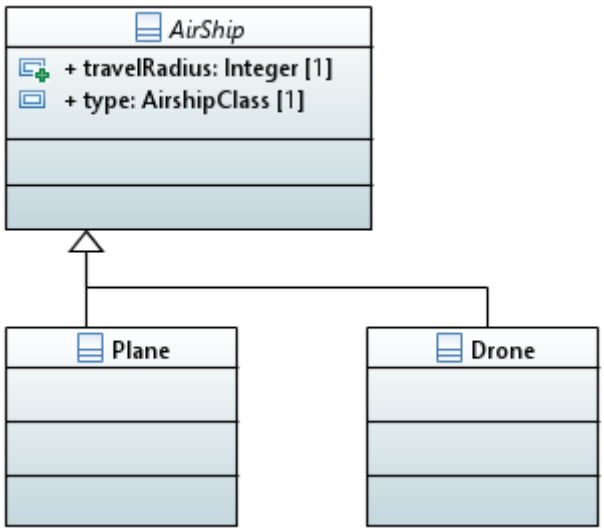
```
import Vehicle from '../Vehicle';
import Ship from '../Ships/Ship';

export enum AirshipClass {
```

```
  Plane,
  Drone,
}

export abstract class Airship extends Vehicle {
  readonly travelRadius!: number;
    readonly type!: AirshipClass;
    readonly ship!: Ship;
    constructor(ship: Ship) {
        super(ship.positionX, ship.positionY, ship.directionDegrees);
        this.ship = ship;
    }
}

export class Plane extends Airship {
  readonly travelRadius = 7;
  readonly type = AirshipClass.Plane;
}

export class Drone extends Airship {
  readonly travelRadius = 4;
  readonly type = AirshipClass.Drone;
}
```

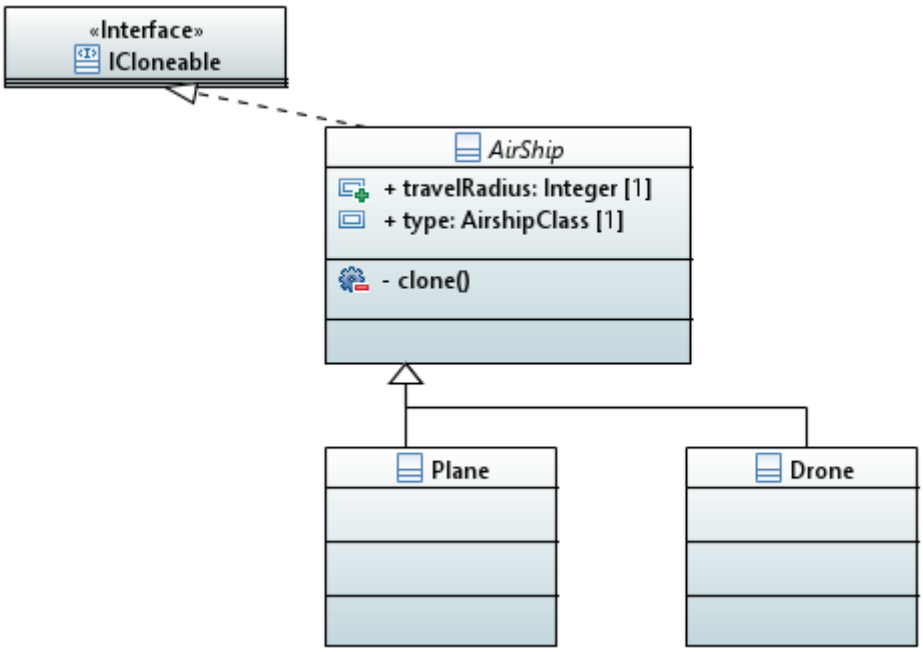
2.10 Prototype – Airship

Atsakingas komandos narys: Vilmantas Pieškus

Prototype šablonas buvo panaudotas, kad galėtume kopijuoti objektus ir nekurti juos kiekvieną iškvietimą. Pritaikyta yra Airship klasėje, kurioje mes galime kviesti lėktuvą arba droną, ir sukūrus vieną objektą mes vis kopijuojame tuos objektus, kurie yra nepriklausomi nuo jų klasių.



Figūra 15: Klasių diagramos fragmentas iki šablono implementacijos



Figūra 16: Prototype šablono klasių diagrama

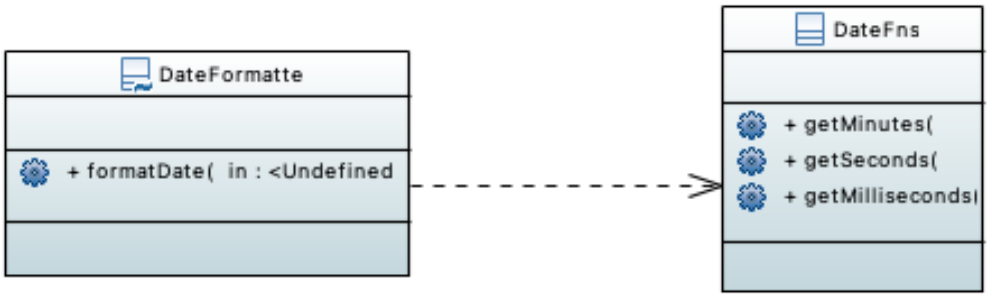
Šablono kodo projektas:

```
export enum AirshipClass {
    Plane,
    Drone,
}

export abstract class Airship extends Vehicle implements Cloneable {
    readonly travelRadius!: number;
    readonly type!: AirshipClass;
    readonly ship!: Ship;
    constructor(ship: Ship) {
        super(ship.positionX, ship.positionY, ship.directionDegrees);
        this.ship = ship;
    }
    clone(): this {
        const objCloneByJsonStringfy = JSON.parse(JSON.stringify(this));
        return objCloneByJsonStringfy;
    }
}

export class Plane extends Airship {
    readonly travelRadius = 7;
    readonly type = AirshipClass.Plane;
}

export class Drone extends Airship {
    readonly travelRadius = 4;
    readonly type = AirshipClass.Drone;
}
interface Cloneable {
    clone(): this;

  }
```

2.11 Facade – DateFormatter

Atsakingas komandos narys: Matas Rutkauskas

Facade šablonas buvo panaudotas, kad sukurti sluoksnį tarp aplikacijos ir datų bibliotekos, kadangi iš jos naudosime tik vieną ar kelias funkcijas, papildoma klasė mums leidžia atskleisti tik reikalingus metodus. Norint pakeisti biblioteką, užtektų pakeisti tik metodų implementaciją.



Figūra 17: Facade šablono klasių diagrama

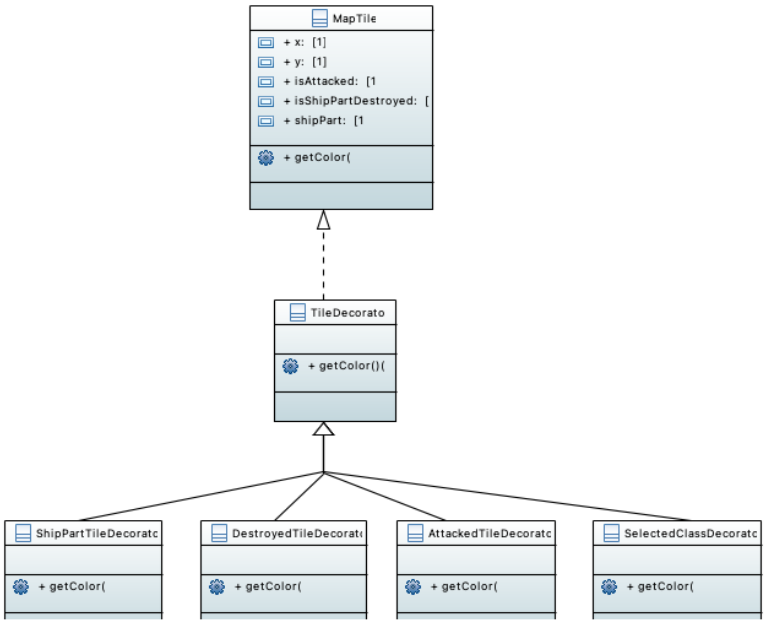
Šablono kodas:

```
export class SelectedTileDecorator extends TileDecorator {
  getColor(){
    return TileColor.grey;
  }
}
```

2.12 MapTileDecorator

Atsakingas komandos narys – Matas Rutkauskas

Buvo panaudotas decorator šablonas, kad lengviau atskirti žemėlapio langelių funkcionalumą, dabar kiekvienas decorator grąžina atitinkamą langelio spalvą, bet ateityje funkcionalumas gali prasiplėsti.



Figūra 18: Decorator šablono klasių diagrama

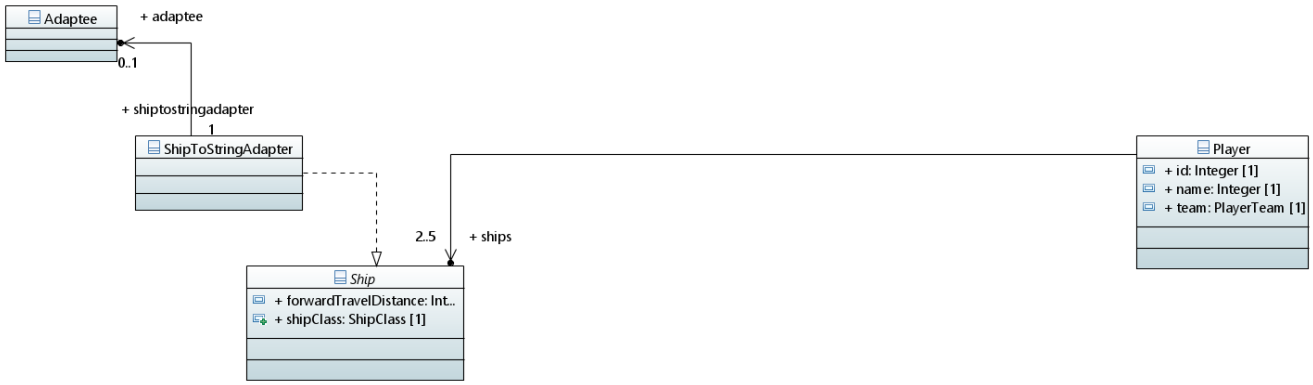
Šablono kodas:

```
export abstract class TileDecorator implements MapTile {
  isAttacked: boolean;
  isShipPartDestroyed: boolean;
  x: number;
  y: number;

  constructor(tile: MapTile) {
    this.isAttacked = tile.isAttacked;
    this.isShipPartDestroyed = tile.isShipPartDestroyed;
    this.x = tile.x;
    this.y = tile.y;
  }

  abstract getColor(): TileColor;

}

export class ShipPartTileDecorator extends TileDecorator {
  getColor(){
    return TileColor.blue;
  }
}

export class AttackedTileDecorator extends TileDecorator {
  getColor(){
    return TileColor.yellow;
  }
}

export class DestroyedTileDecorator extends TileDecorator {
  getColor(){
    return TileColor.red;
  }
}

export class SelectedTileDecorator extends TileDecorator {
  getColor(){
    return TileColor.grey;
  }
}
```

2.13 Adapter – ShipToStringAdapter

Atsakinga komandos narė: Saulė Virbičianskaitė

Adapter šablonas leidžia dviems sąsajoms bendrauti. Čia Ship objektas paverčiamas į string, kuris, prireikus, gali būti panaudojamas API.



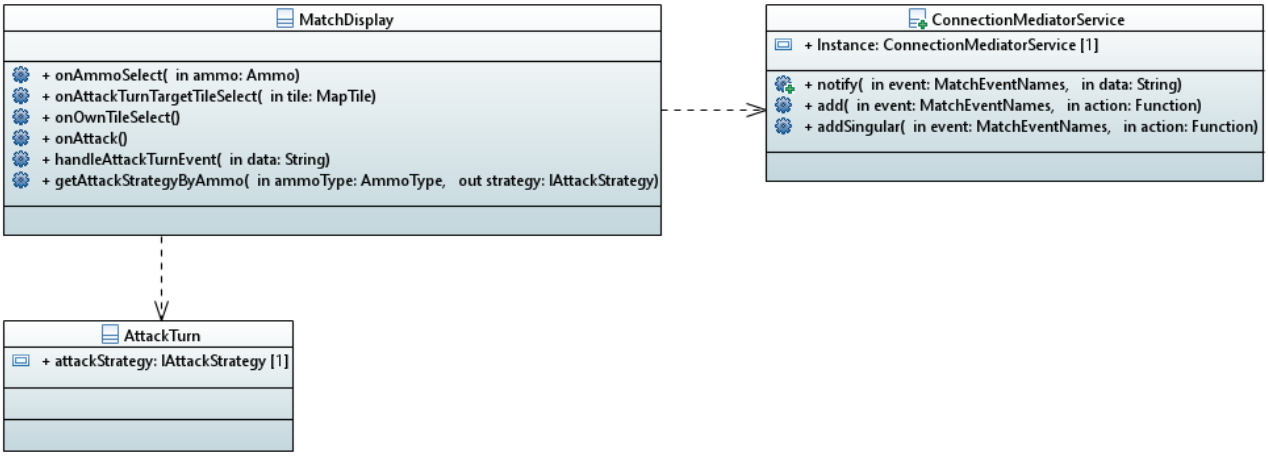
Figūra 19: Adapter šablono klasių diagrama

Šablono kodo fragmentas:

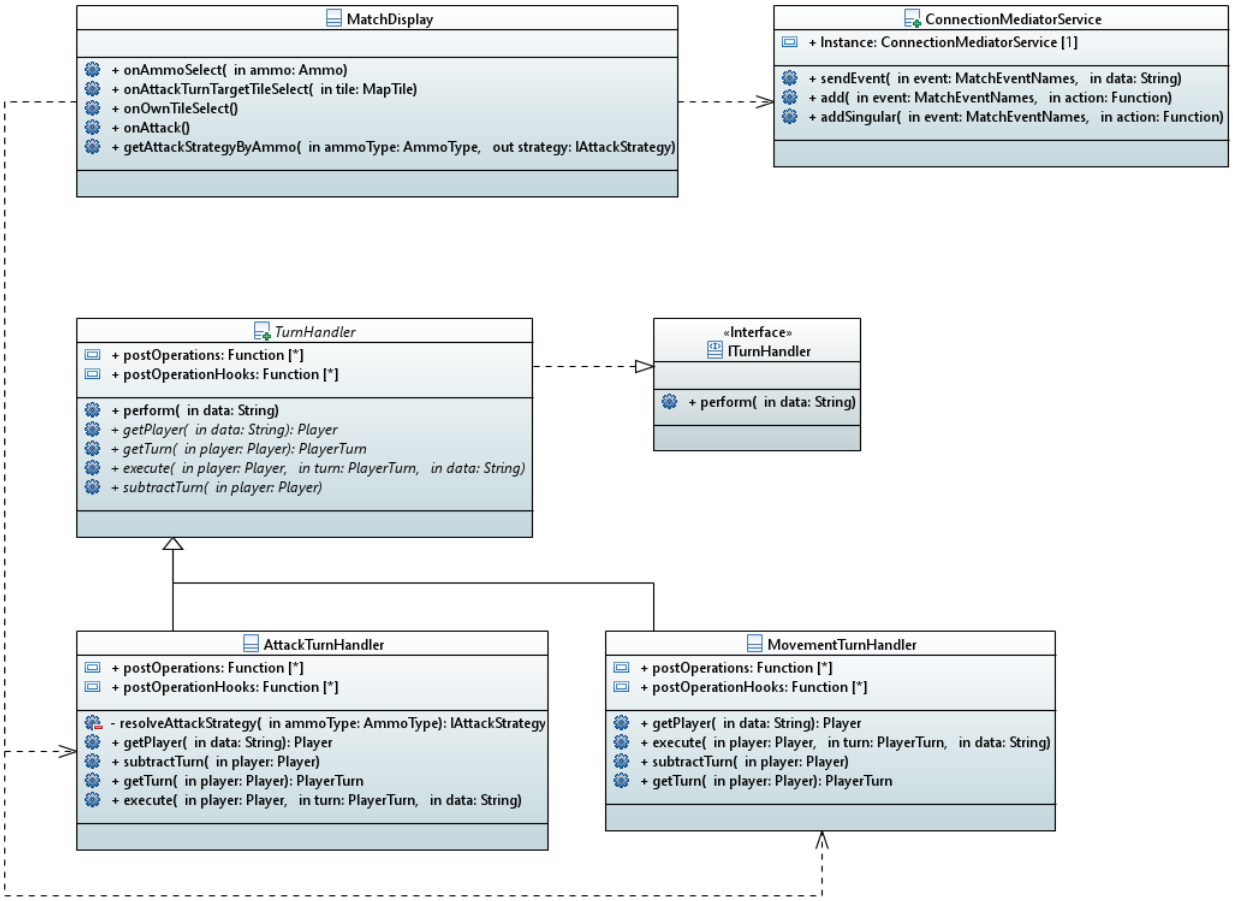
```
export class ShipToStringAdapter extends Ship {
    public ship: Ship;
    shipToString: string;
    constructor(ship: Ship) {
        super();
        this.ship = ship;
        this.shipToString = JSON.stringify(ship);
    }
    create() {
        var logger = LoggerService.Instance.getLogger(PatternTypes.Adapter);
        logger.log(`Ship to string: `, this.shipToString);
    }
}
```

2.14 Template – TurnHandler

Atsakingas komandos narys: Šarūnas Palianskas



Figūra 20: Klasių diagrama prieš šablono pritaikymą



Figūra 21: Template šablono klasių diagrama

Šablono kodo fragmentas:

```
export default function MatchDisplay() {

  useEffect(() => {
    const attackTurnHandler = new AttackTurnHandler(
      getAttackStrategyByAmmo,
      [(_) => rerender()],
      [(_) => true]
    );

    ConnectionMediatorService.Instance.addSingular(
      MatchEventNames.AttackPerformed,
      attackTurnHandler.perform
    );
  }, []);
}
```

```
import { AmmoType } from '../../models/Ammo';
import { MapTile } from '../../models/MatchMap';
import { Player } from '../../models/Player';
import { AttackTurn, PlayerTurn } from '../../models/Turns/AttackTurn';
import MatchProvider from '../MatchProvider/MatchProvider';
import { IAttackStrategy } from
'../Strategies/AttackStrategies/AttackStrategies';

export interface ITurnHandler {
  perform(data: any): void;
}

export abstract class TurnHandler implements ITurnHandler {
  readonly perform = (data: any): void => {
    const player = this.getPlayer(data);
    const turn = this.getTurn(player);

    this.execute(player, turn, data);
    this.subtractTurn(player);

    this.postOperationHooks.forEach((hook, index) => {
      if (hook(data)) {
        this.postOperations[index](data);
      }
    });
  };

  protected abstract getPlayer(data: any): Player;
  protected abstract getTurn(player: Player): PlayerTurn;
  protected abstract execute(player: Player, turn: PlayerTurn, data: any): void;
  protected abstract subtractTurn(player: Player): void;

  protected abstract postOperations: { (data: any): void }[];
  protected abstract postOperationHooks: { (data: any): boolean }[];
}

export class AttackTurnHandler extends TurnHandler {
  constructor(
    private resolveAttackStrategy: { (ammoType: AmmoType): IAttackStrategy },
    postOperations: { (data: any): void }[] = [],
```

```
    postOperationHooks: { (data: any): boolean }[] = []
  ) {
    super();

    this.postOperations = postOperations;
    this.postOperationHooks = postOperationHooks;
  }

  protected getPlayer(data: any): Player {
    const { offencePlayerId } = data as AttackTurnEventProps;

    return MatchProvider.getPlayer(offencePlayerId)!;
  }
  protected getTurn(player: Player): PlayerTurn {
    return player.attackTurns[0];
  }
  protected execute(player: Player, turn: PlayerTurn, data: any): void {
    const { defencePlayerId, tile, ammoType } = data as AttackTurnEventProps;

    const defencePlayer = MatchProvider.getPlayer(defencePlayerId)!;
    const mapTile = defencePlayer.map.tiles[tile.x][tile.y];

    const attackTurn = turn as AttackTurn;
    attackTurn.attackStrategy = this.resolveAttackStrategy(ammoType);
    attackTurn.attackStrategy.attack(mapTile, defencePlayer.map);
  }
  protected subtractTurn(player: Player): void {

    if (player.attackTurns.length > 0) {
      player.attackTurns.shift();
    } else {
      player.turnOverDraw++;
    }
  }

  protected postOperations: ((data: any) => void)[];
  protected postOperationHooks: ((data: any) => boolean)[];
}
```

```
export class MovementTurnHandler extends TurnHandler {
  protected getPlayer(data: any): Player {
    throw new Error('Method not implemented.');
  }
  protected getTurn(player: Player): PlayerTurn {
    throw new Error('Method not implemented.');
  }
  protected execute(player: Player, turn: PlayerTurn, data: any): void {
    throw new Error('Method not implemented.');
  }
  protected subtractTurn(player: Player): void {
    throw new Error('Method not implemented.');
  }
  protected postOperations: ((data: any) => void)[] = [];
  protected postOperationHooks: ((data: any) => boolean)[] = [];
}

interface AttackTurnEventProps {
  offencePlayerId: number;
  defencePlayerId: number;
  tile: MapTile;
  ammoType: AmmoType;
}
```

3 Išvados

Atlikus projektinio darbo pirmąją dalį buvo:

- suprojektuotas ir įgyvendintas žaidimas „Laivų mūšis“ panaudojant programinės įrangos projektavimo šablonus,
- praktiškai įsisavinti programinės įrangos projektavimo šablonų naudojimo ypatumai.