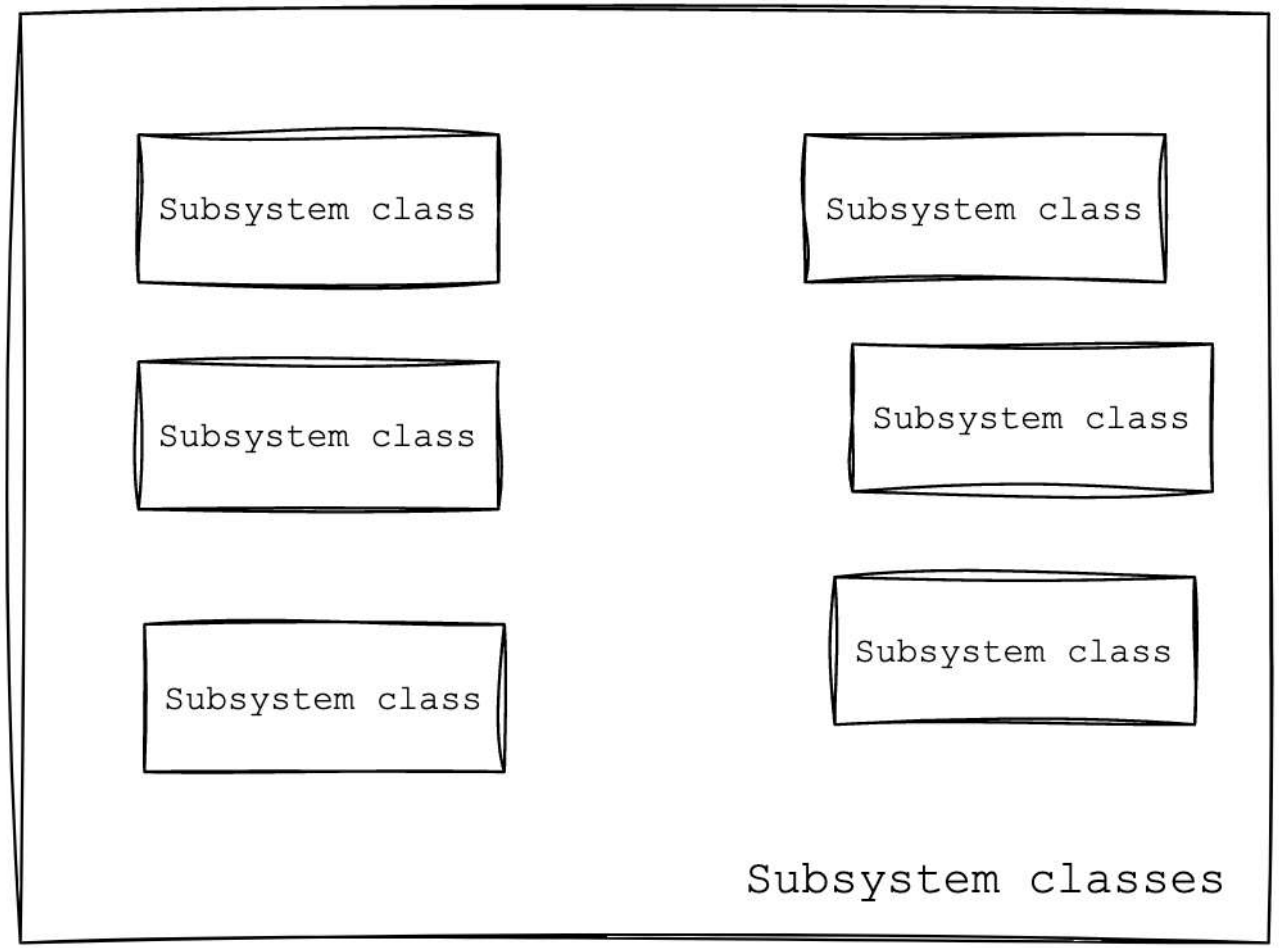
Façade

Artem Bakhtin

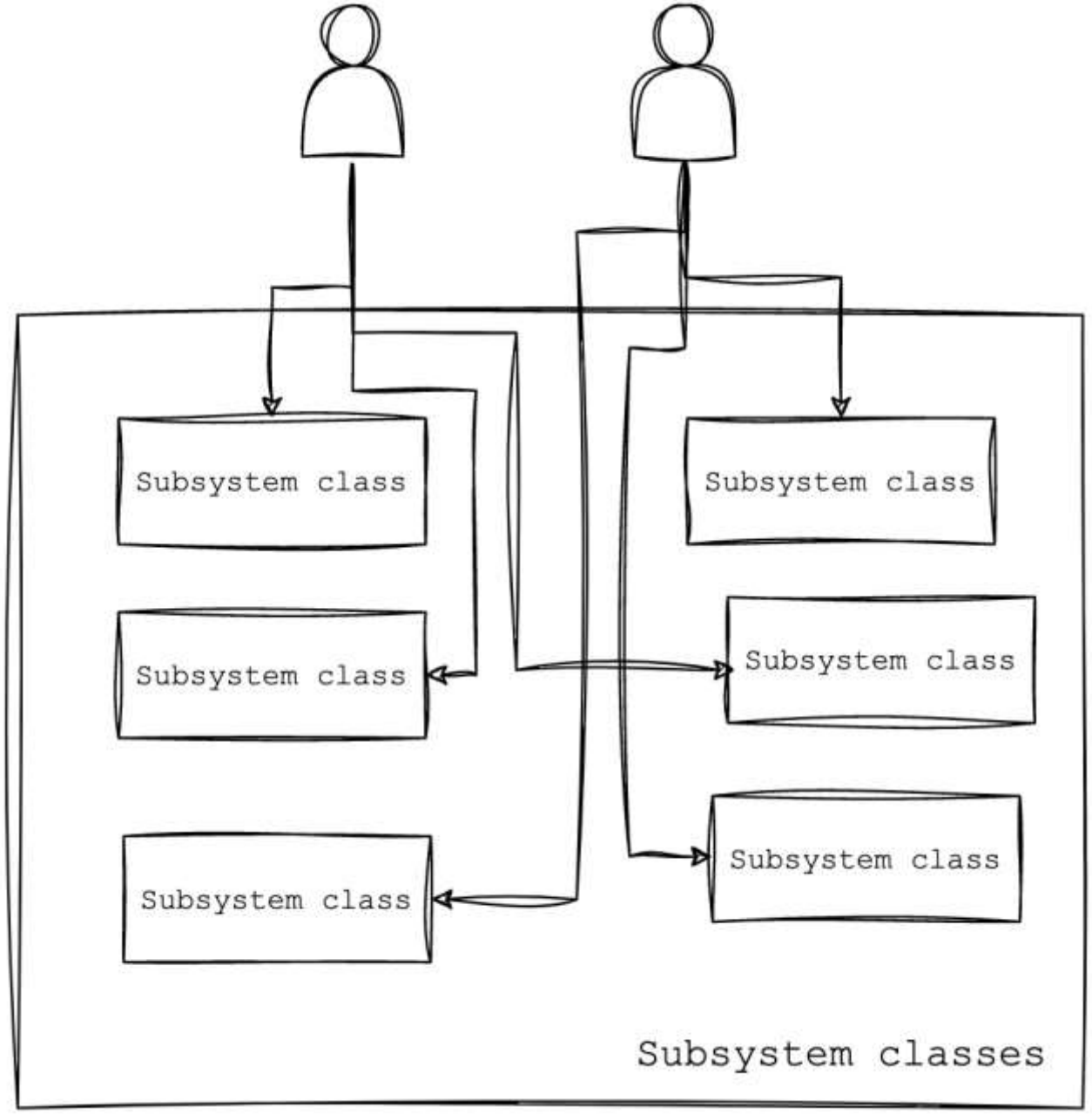
Motivation

- A system consisting of many subsystems



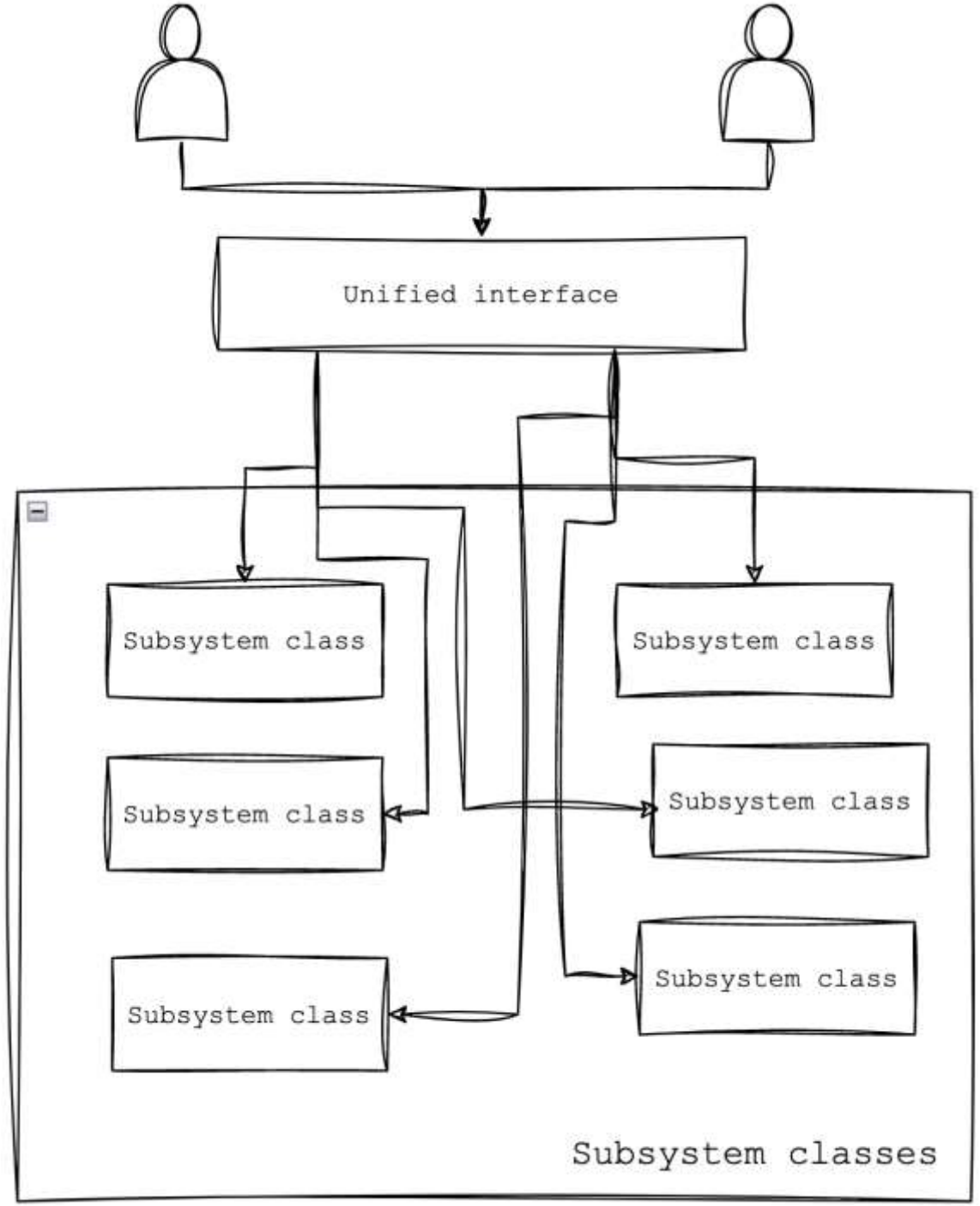
Motivation

- A subsystem consisting of many subsystem classes
- Manual control of subsystems
  - Object initialization
  - Dependency management
  - Exectuion flow management



Solution

- Unified interface



Facade

Facade intent to provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. (GoF)

Example

```
public class Engine {
    public void start() { … }
    void stop() { … }
}
```

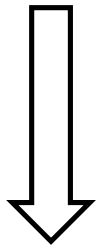
```
public class BrakePedal {
    public void brake() { … }
}
```

```
public class AcceleratorPedal {
    public void accelerate() { … }
}
```

```
public class Display {
    public void showContent(String content) { … }
}
```

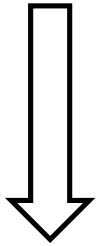
```
public class EntertainmentSystem {

    private Display display;

    public EntertainmentSystem(Display display) {
        this.display = display;
    }

    public void playMusic() { … }
    public Display getDisplay() { … }
}
```

```
class CarFacade {
    private Engine engine;
    private EntertainmentSystem entertainmentSystem;
    private AcceleratorPedal acceleratorPedal;
    private BrakePedal brakePedal;

    public CarFacade() {
        this.engine = new Engine();
        this.entertainmentSystem = new EntertainmentSystem(new Display());
        this.acceleratorPedal = new AcceleratorPedal();
        this.brakePedal = new BrakePedal();
    }

    public void startCar() {
        engine.start();
        entertainmentSystem.getDisplay().showContent("Show music library");
        entertainmentSystem.playMusic();
        acceleratorPedal.accelerate();
    }

    public void stopCar() {
        brakePedal.brake();
        engine.stop();
    }
}
```



The principle of least knowlede

- Also known as Law of Demeter
- Invoke methods that belong to
  - The object itself
  - Objects passed in the parameters
  - Any object the method creates or instantiates
  - Any components of the objects



```
public void startCar() {
    engine.start();
    entertainmentSystem.getDisplay().showContent("Show music library");
    entertainmentSystem.playMusic();
    acceleratorPedal.accelerate();
}
```

Apply principle

```
public class EntertainmentSystem {
    …
    public void showContent(String content) {
        …
        this.display.showContent(content);
    }
    …
}
```

```
class CarFacade {
    …
    public void startCar() {
        engine.start();
        entertainmentSystem.showContent("Show music library");
        entertainmentSystem.playMusic();
        acceleratorPedal.accelerate();
    }
    ….
}
```

Pitfalls and Best Practices

- Overusing Facades
- Facade Pattern is not a Silver Bullet
- Keep Facades Lightweight
- Loose Coupling

Abstract Factory

```
interface Engine {
    void start();
    void stop();
}
```

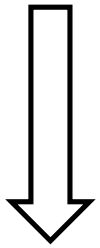
```
interface CarPartFactory {
    Engine createEngine();
    BrakesPedal createBrakesPedal();
    EntertainmentSystem createEntertaimentSystem();
    AccelerationPedal createAccelerationPedal();
}
```

```
class BasicEngine implements Engine {
    @Override
    public void start() { … }

    @Override
    public void stop() { … }
}
```

```
class LuxuryEngine implements Engine {
    @Override
    public void start() { … }

    @Override
    public void stop() { … }
}
```

```
class LuxuryCarPartFactory implements CarPartFactory {
    @Override
    public Engine createEngine() {
        return new LuxuryEngine();
    }

    @Override
    public AcceleratorPedal createAccelerationPedal() {
        return new LuxuryAccelerationPedal();
    }

    @Override
    public EntertainmentSystem createEntertainmentSystem() {
        return new LuxuryEntertainmentSystem(new LuxuryDisplay);
    }

    @Override
    public BrakePedal createBrakePedal() {
        return new LuxuryBreakPedal();
    }
}
```

Abstract Factory



```
class CarFacade {
    …

    public CarFacade(CarPartFactory factory) {
        this.engine = factory.createEngine();
        this.entertainmentSystem = factory.createEntertainmentSystem();
        this.acceleratorPedal = factory.createAccelerationPedal();
        this.brakePedal = factory.createBrakePedal();
    }

    …
}
```

The Facade Pattern in Real Word Projects

- JdbcTemplate

```java
@Override
@Nullable
public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws DataAccessException {
    Assert.notNull(sql, "SQL must not be null");
    Assert.notNull(rse, "ResultSetExtractor must not be null");
    if (logger.isDebugEnabled()) {
        logger.debug("Executing SQL query [" + sql + "]");
    }

    // Callback to execute the query.
    class QueryStatementCallback implements StatementCallback<T>, SqlProvider {
        @Override
        @Nullable
        public T doInStatement(Statement stmt) throws SQLException {
            ResultSet rs = null;
            try {
                rs = stmt.executeQuery(sql);
                return rse.extractData(rs);
            }
            finally {
                JdbcUtils.closeResultSet(rs);
            }
        }
        @Override
        public String getSql() {
            return sql;
        }
    }

    return execute(new QueryStatementCallback(), true);
}
```

Usage in nowadays frameworks

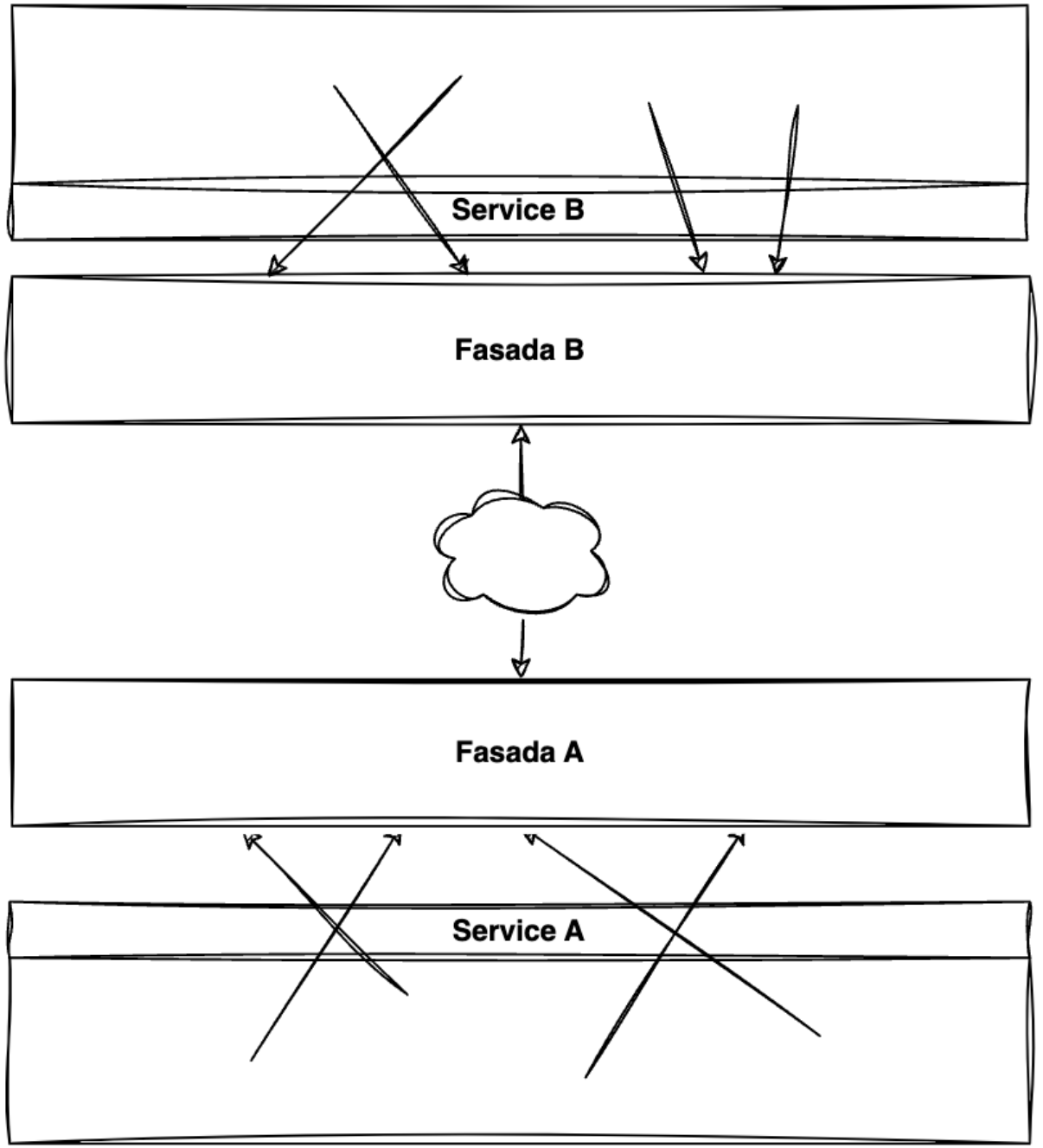
```
@Injectable({ providedIn: 'root' })
export class AuthFacade {
	isAuthenticated$ = this.store.pipe(select(AuthSelectors.isAuthenticated));

	constructor(private store: Store<fromAuth.AuthState>) {}

	login(auth: AuthenticateUser) {
		this.store.dispatch(login({ auth }));
	}

	logout() {
		this.store.dispatch(logout());
	}

	submitUserRegistration(auth: AuthenticateUser) {
		this.store.dispatch(register({ auth }));
	}
}
```

Relation with others DPs

- Adapter
- (Abstract) Factory
- Proxy
- Mediator
- Singleton

More use examples

- Microservices
- Legacy API



Conslusion