DESIGN PATTERN

OVERVIEW

Introduction

Patterns

Categories

Additional Category

Design Pattern

- Design Pattern
  - Design Patterns are language independent solutions to common programming problems
  - The designs are object-oriented
- GoF
  - Stands for the Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - Who introduced Design Patterns were introduced by Gamma et al. in their 1994 Book “Design Patterns: Elements of Reusable Object-Oriented Software”
- Categories
  - Design patterns are contained within three categories: Creational, Behavioural and Structural

Design Patterns for Java

Creational

1) Singleton
2) Factory
3) Abstract Factory
4) Builder
5) Prototype

Structural

1) Template Method
2) Mediator
3) Chain of Responsibility
4) Observer
5) Strategy
6) Command
7) State
8) Visitor
9) Interpreter
10) Iterator
11) Memento

Behavioural

1) Adapter
2) Composite
3) Proxy
4) Flyweight
5) Façade
6) Bridge
7) Decorator

Each design pattern is a solution to a specific problem

Three GoF Categories

- Creational patterns
  - Focus on how to solve a problem by applying a pattern for Instantiating an Object
- Structural patterns
  - Focus on different patterns in creating a class, e.g. making use of inheritance
- Behavioural patterns
  - Focus on interaction between objects, with attention paid to responsibilities

Miscellaneous Design Patterns

- This category includes patterns applicable to Java but not by the GoF.
  - Formally the included patterns are approved by Sun and then Oracle
  - Three notable patterns

Data Access Object (DAO) Pattern

- Allows the application/business layer to be separated from data storage layer, e.g. database layer
- The pattern deploys an abstract API to hide complex details behind CRUD operations in the storage layer.
- https://www.baeldung.com/java-dao-pattern

Dependency Injection Pattern

- When a class has fields which are objects of different classes, we have dependency
- The pattern aims to make objects available on demand
- https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/

Model View Controller (MVC) Pattern

- Separates an application into three main components: Model, View, and Controller
- In UML terms: Entity, Boundary and Controller
- https://www.geeksforgeeks.org/mvc-design-pattern/

CREATIONAL CATEGORY OF DESIGN PATTERNS

Overview

Builder pattern

Builder design

Overview of Creational Design Patterns

Class constructors can be inflexible

- Only one instance of an object is created at a time
- Inability to return a value
- Inability to control when and how many objects are instantiated

Creational patterns aim to overcome such issues

- And at the same time increasing flexibility and reuse of existing code

Common Creational Patterns

- Singleton, Factory method, Abstract factory, Builder, Object Pool, Prototype

Activity 1

Work in a group of two to search and define the following terms:

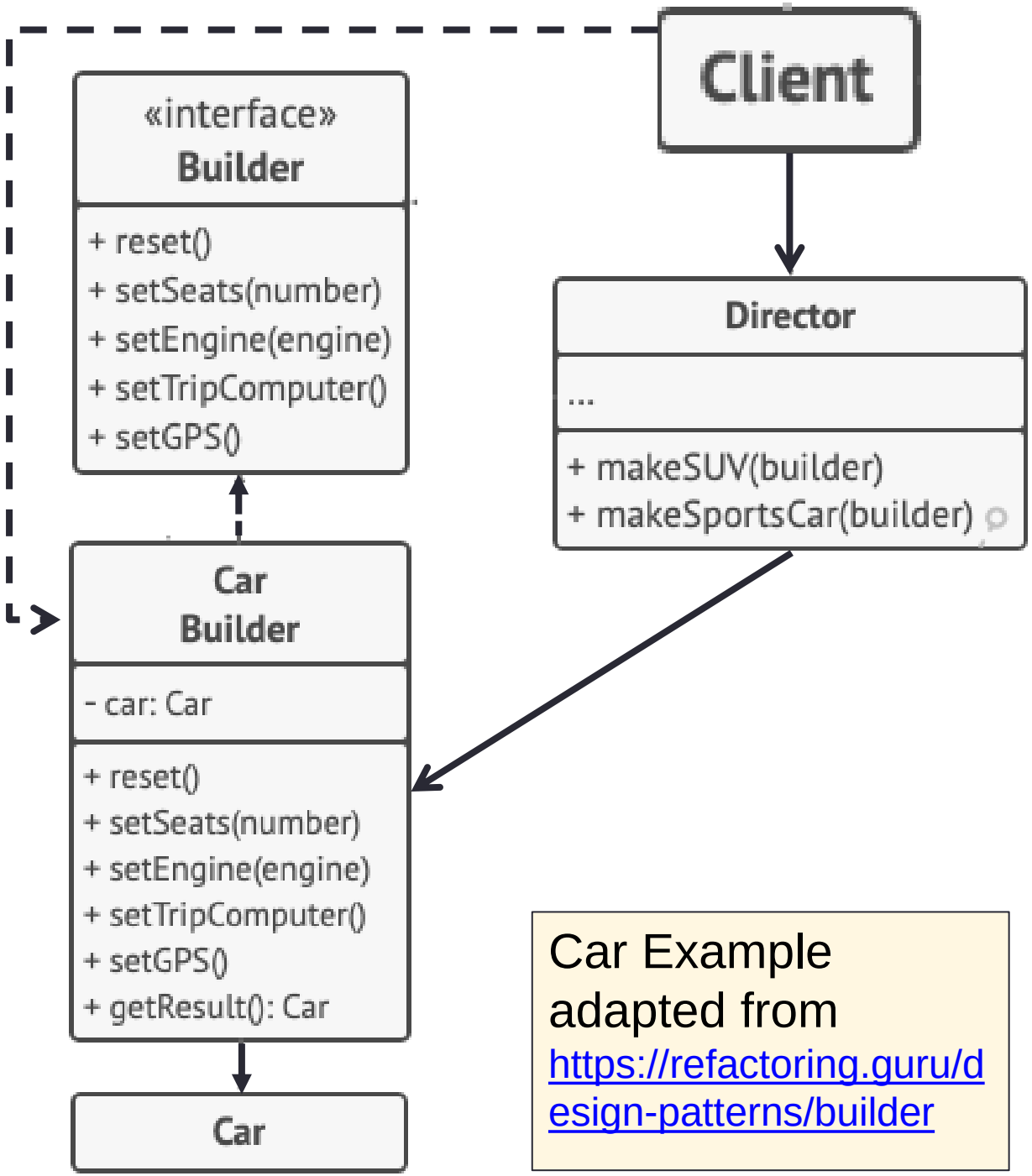
- Singleton
- Factory method
- Abstract factory
- Builder
- Object Pool
- Prototype

Reflect on how useful these concepts for java development.

Builder Pattern

- Underlying Problem
  - The main class (the client) needs customisable object that may include or omit properties
  - This required the object's class to have numerous fields and several overloaded constructors
  - Some fields may have dependencies
  - Other fields may be objects
  - Result is a gigantic class with lots of parameters, constructors and methods
- Solution: Builder pattern specifies that field initialisation bypasses constructor via:
  1. A Base Interface (the **Builder**)
  2. An implementation of the builder interface (i.e. **concrete Builder** class)
  3. A **Director** class to deploy a recipe to set fields for the object

Example Builder Design



Car Example adapted from https://refactoring.guru/design-patterns/builder

- CarBuilder implements Builder Interface
  - The field in CarBuilder Class is a Car object, which consists of various objects, e.g. seat.
  - It has methods to customise how these objects fit into the car
- The Director has various methods
  - Each method is a recipe for a specific type of car, e.g., `makeSportsCar()`
  - Each recipe calls methods from CarBuilder, with different values, e.g. `builder.setSeats(2)`
- Client creates CarBuilder (`builder`) and Director (`director`) objects
  - Client invokes a recipe method from `director` and passes `builder` as a parameter to the recipe method
  - Once recipe is complete, the client create Car (`car`) object by

```
Car car = builder.getResult()
```

STRUCTURAL CATEGORY OF DESIGN PATTERNS

Overview
Adaptor Pattern
Example Problem
Data Classes
Adaptor Class

Overview of Structural Design Patterns

Overview

- In terms of efficiency, the developer aims to reuse code
- The same applies to reusing objects and classes from one problem for a different problem

Structural patterns

- Are used to create large structures by reusing objects and classes
- The reuse is based on examining relationships between the smaller structures

Common structural patterns

- Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Private class data, Proxy

Where Adaptor Patten is used

- An adaptor is used where different devices have different interfaces
  - A real-life example are different power sockets when travelling
  - Solved by using a Power Adaptor.
- In Java, the problem relates to different data classes that may have same type of data, but may have
  - Different field names
  - Different amount of fields
  - Different method names for corresponding methods
- Adaptor pattern can be used to allow fields of one class to be accessed
  - Using the corresponding method name of the other class

Example Problem

The Client

- The client is a seller of hospitality ticket packages for UK Squash Tournaments
- Its existing data class is `SquashTicket`

New Data

- The client buys from a competing, the sale of hospitality passes for UK Men's Padel Tournaments
- Where the data class is `PadelPass`

The Problem

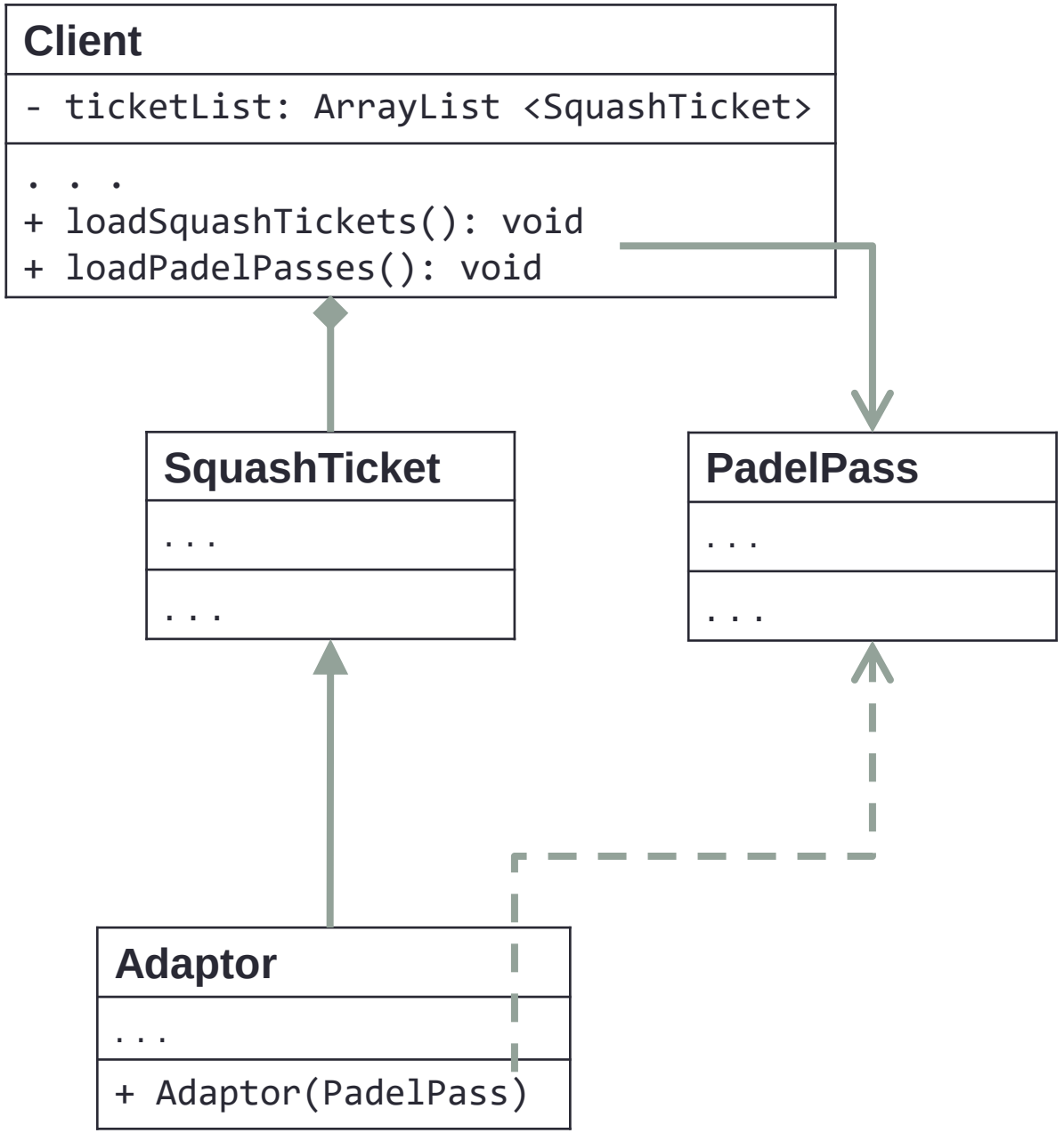
- The client application cannot directly add `PadelPass` objects into its central data structure
- And there is no time to fully modify the application or merge the separate datasets

Data Classes and Data

SquashTicket
- event: String - month: String - club: String - gender: String - round: String - price: double - available: int
. . .

PadelPass
- id: String - tournament: String - day: String - month: String - club: String - location: String - passCost: double - minimumPasses: int - passesLeft: int
. . .

Adaptor Class Relationships



- Client is the application that wishes to make use of additional data
  - i.e., include `PadelPass` Data
- To do so we create an **`Adaptor`** class which inherits from the existing data class in the client
  - `Adaptor extends SquashTicket`
- Separate data files means separate load methods in the Client:
  - `loadSquashTickets()`
  - `loadPadelPasses()`
- `loadSquashTickets()` method will create an `SquashTicket` object
  - And add each object to `ticketList`
- `loadPadelPass()` method will create **`Adaptor`** objects from padel pass data
  - And add these objects to `ticketList`
  - i.e., Polymorphism

Different ways to design Adaptor class

Initialise Inherited Fields

Adaptor Class constructor has a PadelPass object as a parameter

The inherited fields from SquashTicket are initialised using appropriate fields of PadelPass parameter

Feasible to merge PadelPass fields into one SquashTicket field and vice-a-versa

Ignore any field in PadelPass class that is not an equivalent of a SquashTicket field

Override Inherited Methods

Adaptor Class has a PadelPass object as a field

Adaptor class Constructor sets null, empty or default values for inherited SquashTicket fields

Adaptor Class constructor uses parameters to instantiate the PadelPass object

Inherited SquashTicket methods are overridden using equivalent method(s) from the PadelPass object

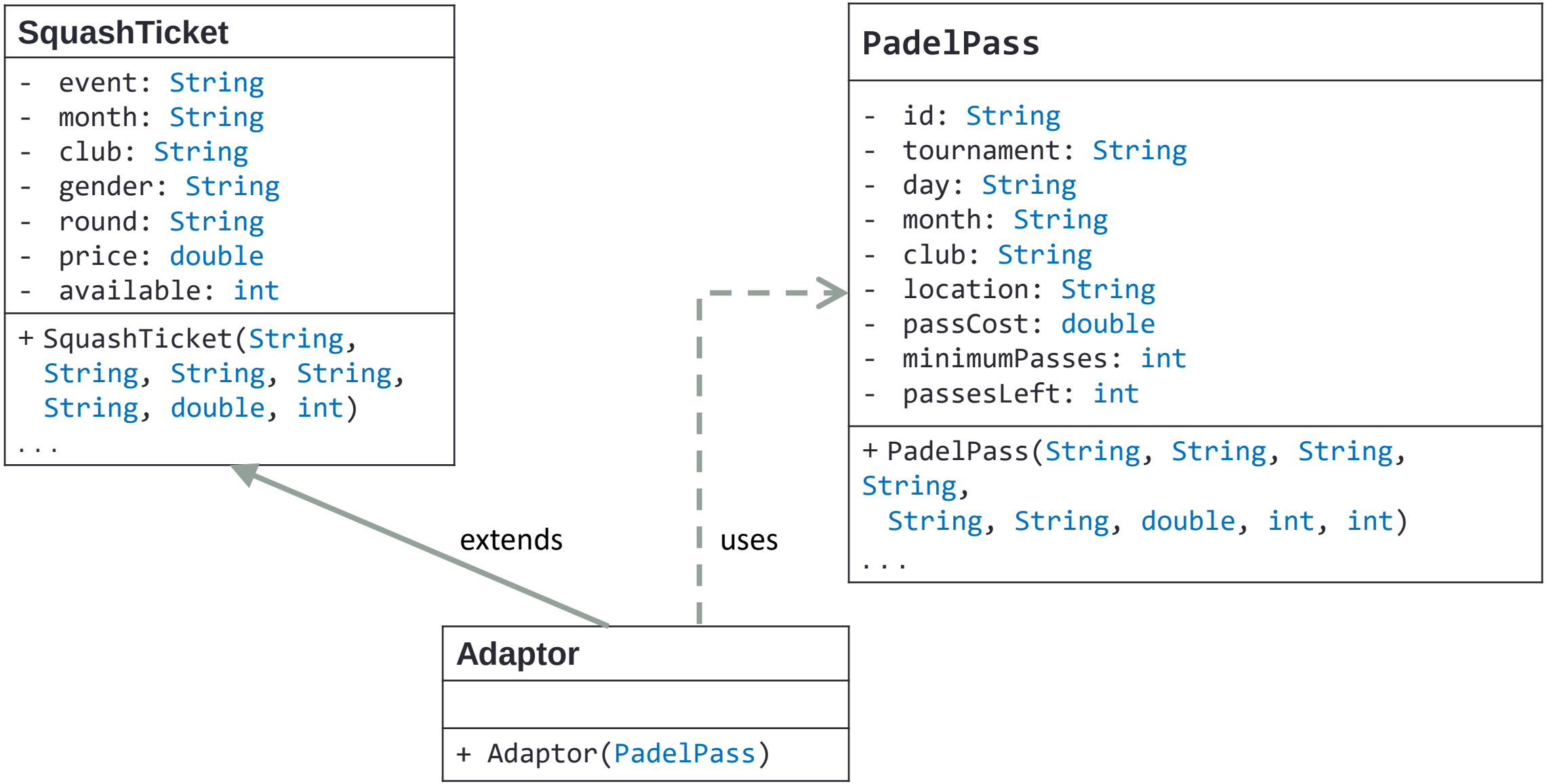
Hybrid Approach

Combination of fields and methods approach

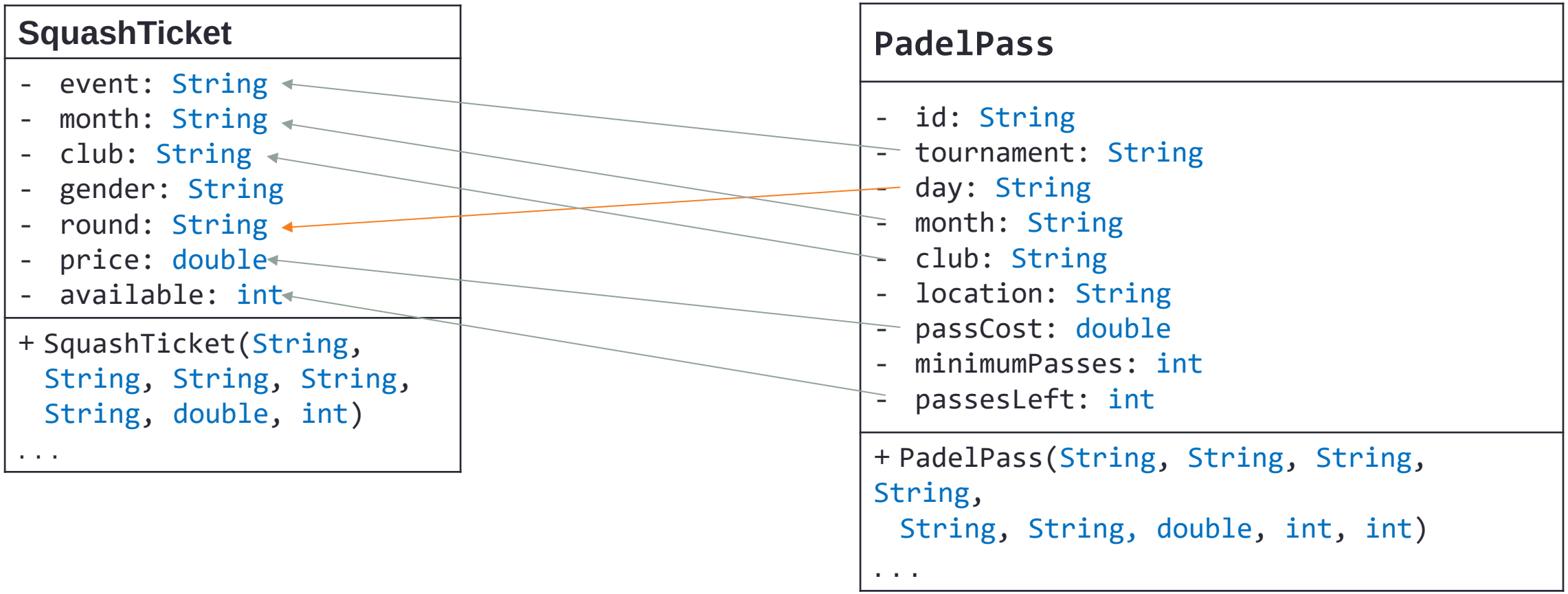
Deployed when we need to manipulate external object

Used to ensure that as much of external data and business rules are fulfilled.

Initialise Inherited Fields: Class Relationships



Field mappings



BEHAVIOURAL CATEGORY OF DESIGN PATTERNS

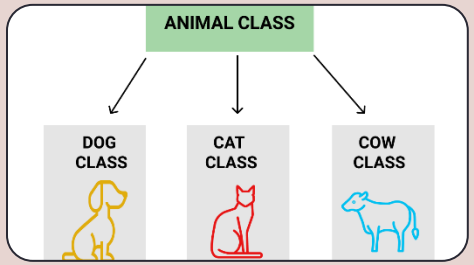
Overview

Memento pattern

Memento design

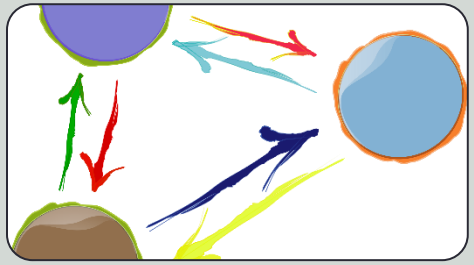
Memento example application runtime

Overview of Behavioural Design Patterns



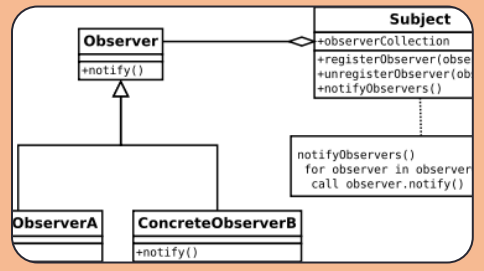
Inheritance

- Inheritance is used create specialised sub classes with additional behaviours
- If various classes need to interact, then what is best mechanism to enable communication



Behavioural Patterns

- Focus on how objects communicate with each other.
- Behavioural object patterns use object composition rather than inheritance
- This involves changing the underlying design and algorithm of a class



Common Behavioural Patterns include

- Command, Interpreter, Mediator, Observer, State, Visitor, Memento

Activity 2

Work in a group of two to search and define the following terms:

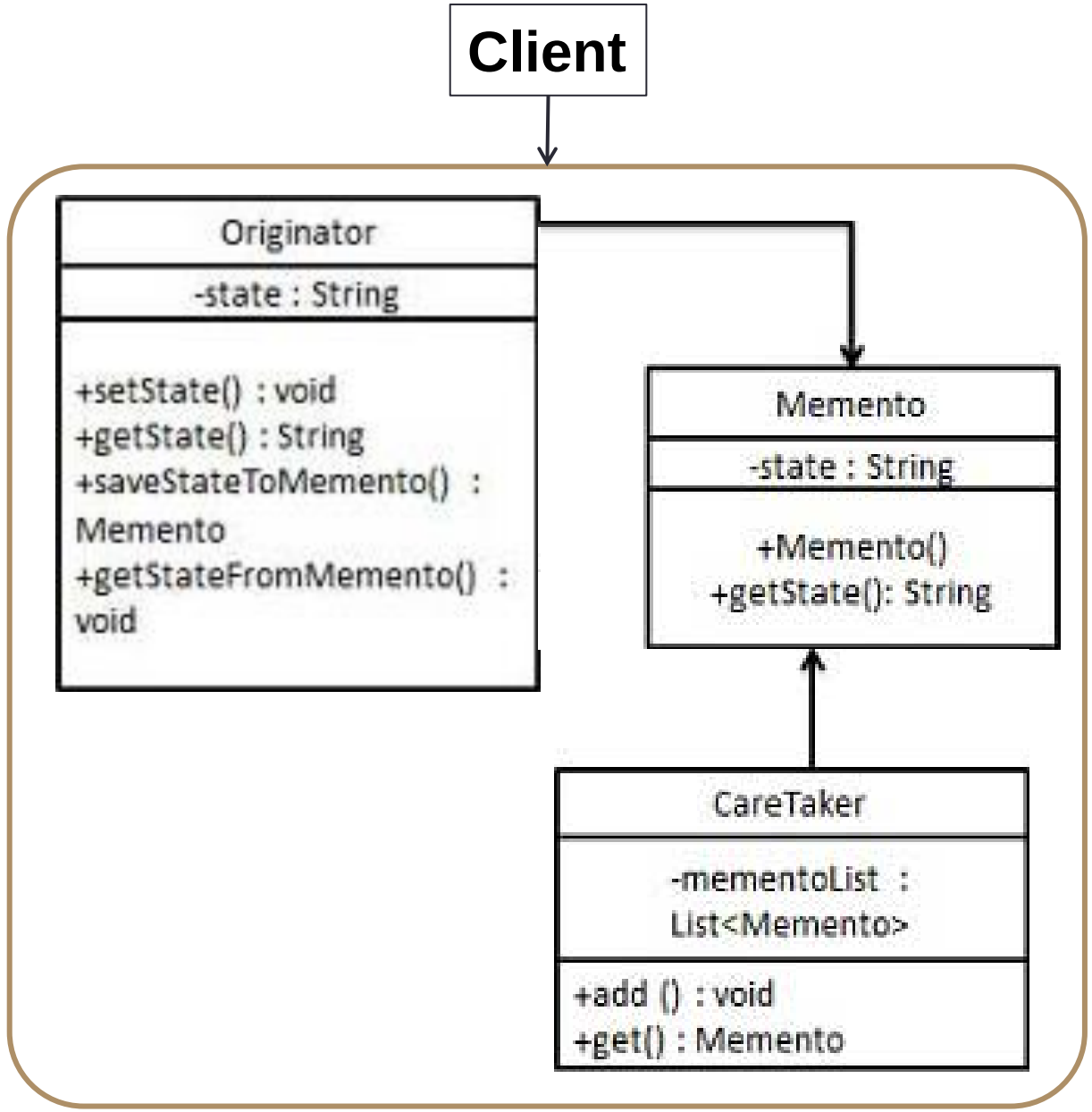
- Command
- Interpreter
- Mediator
- Observer
- State
- Visitor
- Memento

Reflect on how useful these concepts for java development.

Memento Design Pattern

- Underlying Problem
  - Application requires an undo / rollback feature, to return to a previous state
  - The feature must follow principle of encapsulation and not reveal any of stored states outside the application
- Memento design pattern
  - Memento design pattern captures snapshots of an application's state and provides rollback through four classes
    1) Client - the part of the application requesting rollback
    2) Originator - creating and managing the state of an application
    3) Memento - stores the state of the Originator at a particular point in time
    4) Caretaker - responsible for keeping track of Memento object
  - Can also be used to provide transaction/rollback functionality

Example Memento Design



- The client initiates the process by requesting some save or rollback operation
- The originator acts on the operation
- A save operation involves the Originator creating a Memento object to save state
- The memento object is added to a collection in the Caretaker
- A rollback operation requires the Originator to obtain correct memento object from caretaker
- Application is restored to state saved in the retrieved memento object

Example adapted from: https://www.tutorialspoint.com/design_pattern/memento_pattern.htm