OODP– Session 4

Session times

PT group 1 – Monday	18:00-21:00	room: Malet 403
PT group 2 – Thursday	18:00-21:00	room: Malet 407
FT - Tuesday	13:30-17:00	room: Malet 404

Email: oded@dcs.bbk.ac.uk

Web Page: http://www.dcs.bbk.ac.uk/~oded

Visiting Hours: Tuesday 17:00 to 19:00

Previously on OODP

- …
- GRASP design Patterns
  - **Information Expert**
  - **Creator**
  - **Low Coupling**
  - **High Cohesion**
  - **Controller**

(don’t think in patterns, think patterns)

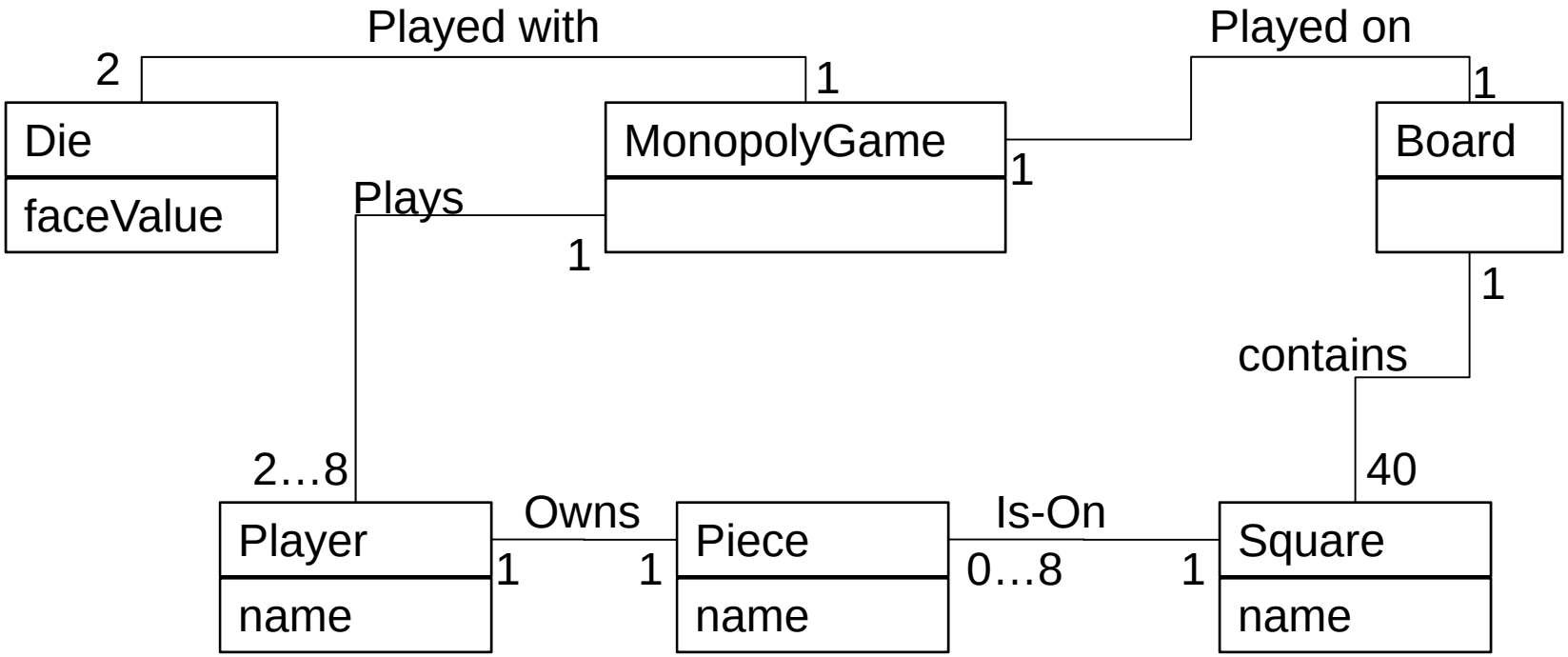
Use Case Realization

Monopoly game simulation

Monopoly Use-Cases

- The game starts, first player roles the dice, first player looks at the value on the dice, first player moves his piece value positions. Second…, i'th player…, first player,…, and this is all repeated n times.
- Instantiation

Monopoly Domain Model



Monopoly: choosing a controller class

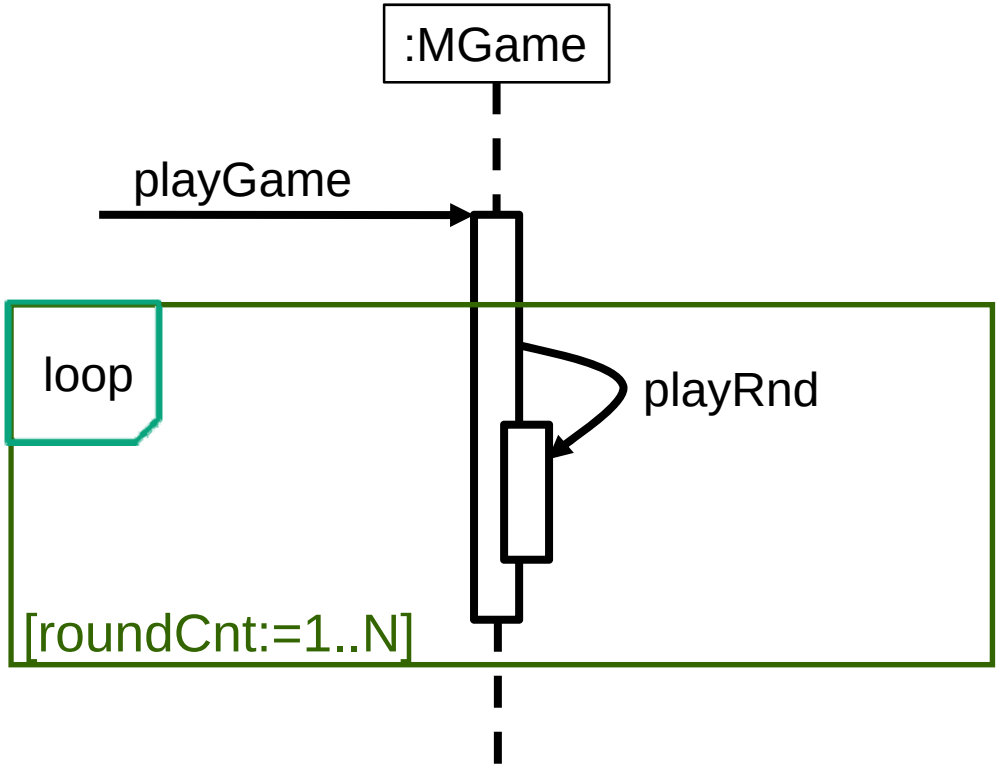
Two options

- MonopolyGame – one object to handle, simple controller pattern application

- MonopolyGameHandler – special object according to an advanced pattern

Monopoly Game Loop

- Who is responsible for Controlling the game loop? (Expert)

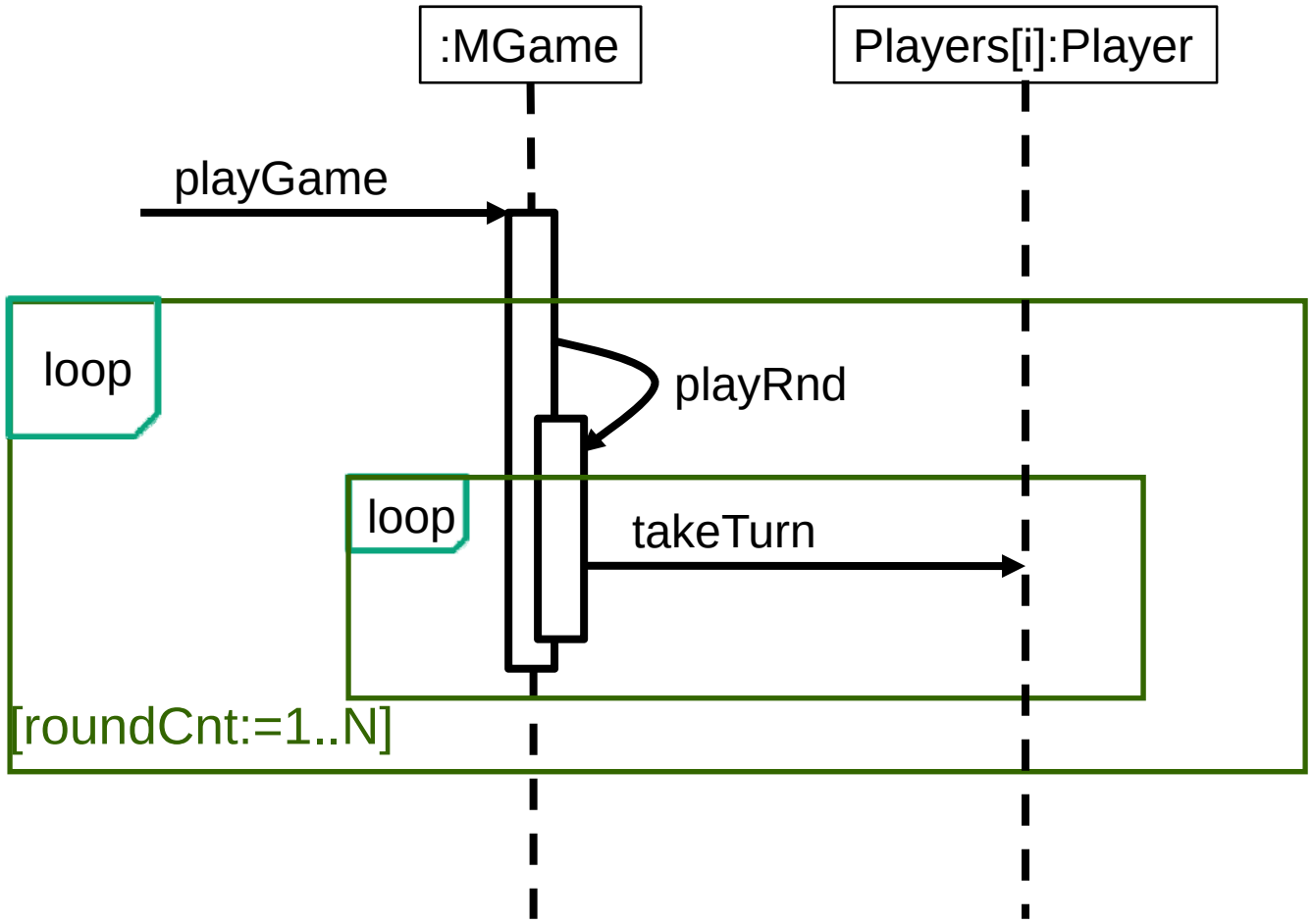


Monopoly Game Loop

- Who takes a turn? (Expert)
- Who has the information needed

Information Needed	Who has it?
Current location of player	**A player knows its piece and a piece knows it square (domain model)**
The die object (for rolling etc.)	**MGame**
All the squares	**Board**

Monopoly Player Takes a Turn



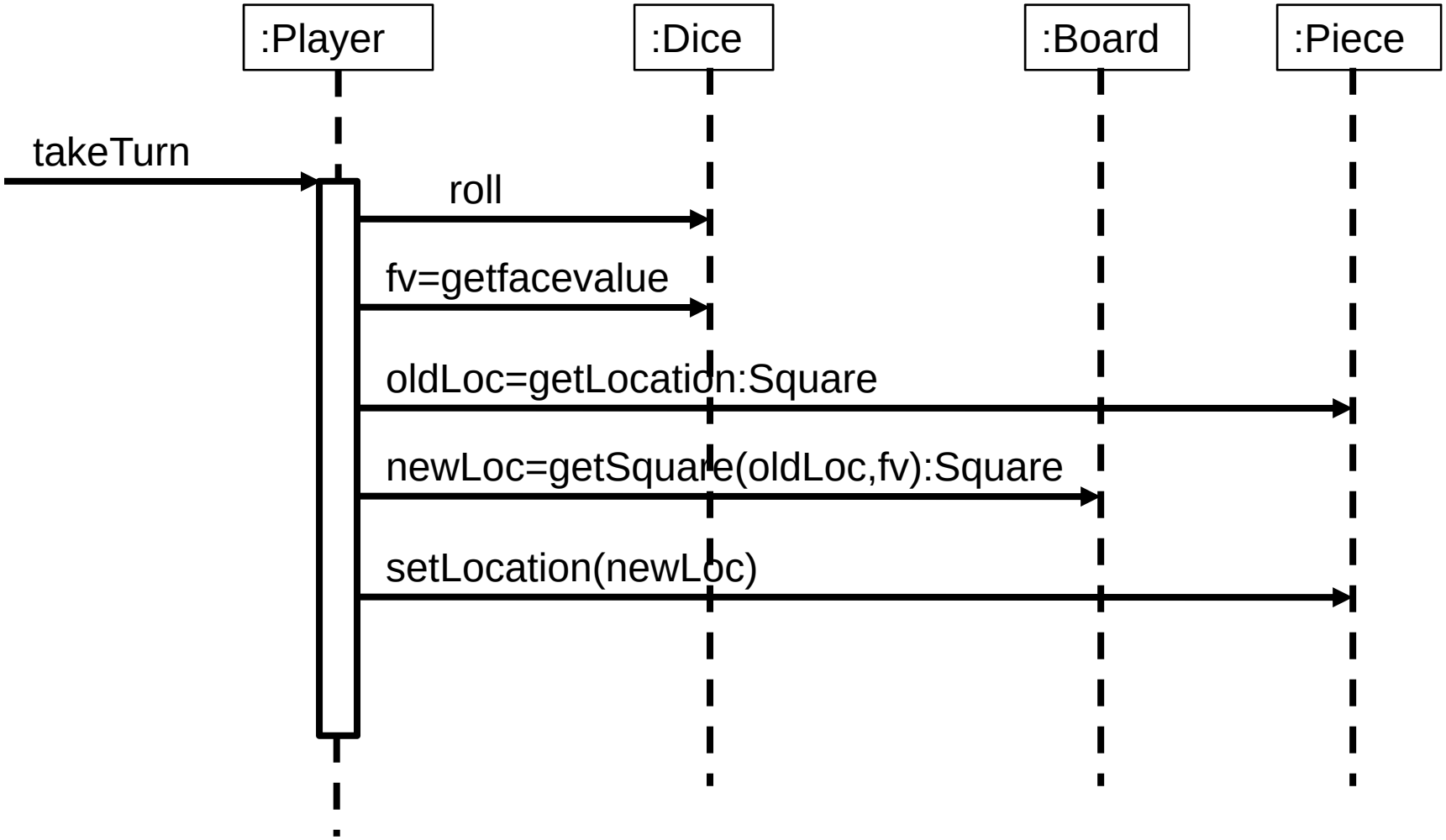
Monopoly Taking a Turn

1. Calculating a random number in the range 2…12
2. Calculating the new square location
3. Moving players piece

- Who computes random number? die class
- Who computes new square position? Board, knows all its squares
- Who sets new location? piece class
- Who coordinates all of this?

Remember: these objects appeared in the domain model, but we are dealing with the design model

Monopoly Player Takes a Turn



Advanced Patterns

GoF pattern description template

- Name & classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Pattern

Classification (GOF)

- Creational Patterns
  - Abstract object instantiation
- Structural Patterns
  - Compose and organise objects into larger structures
- Behavioural Patterns
  - Algorithms, interactions and control flow between objects.

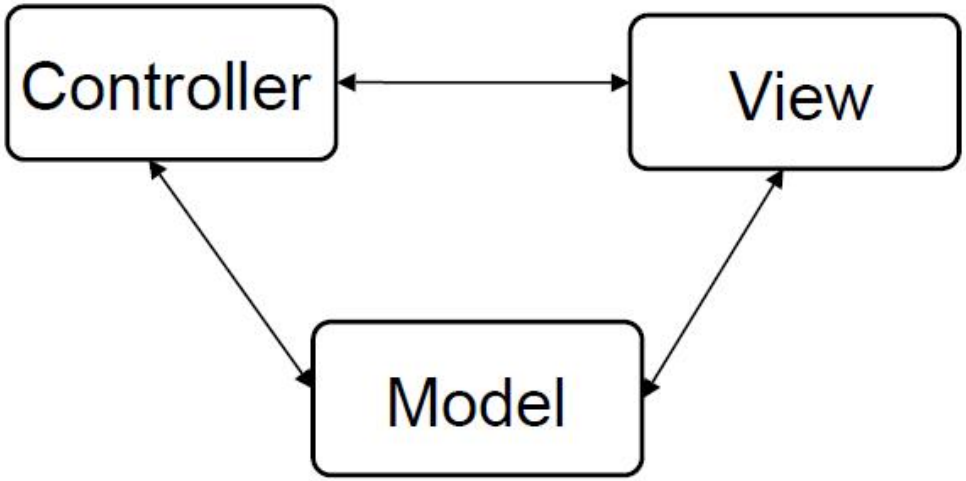
Example: Model-View-Controller (MVC)

Model-View-Controller group of classes used to build interfaces (originally in Smalltalk, Java Swing, EPOC uses MVC).

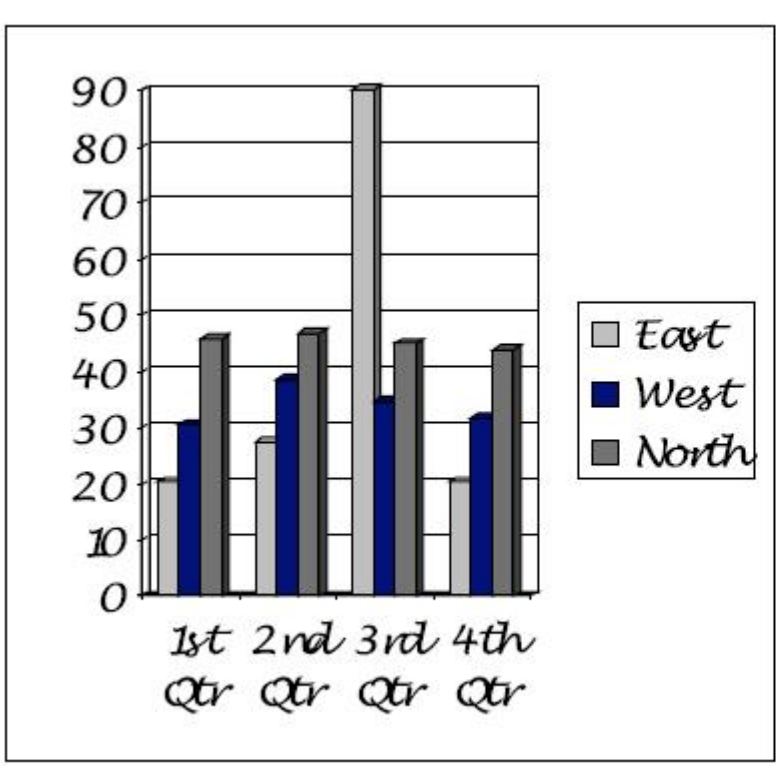
- Model - application object,
- View - presentation on screen,
- Controller - how user input controls the interface

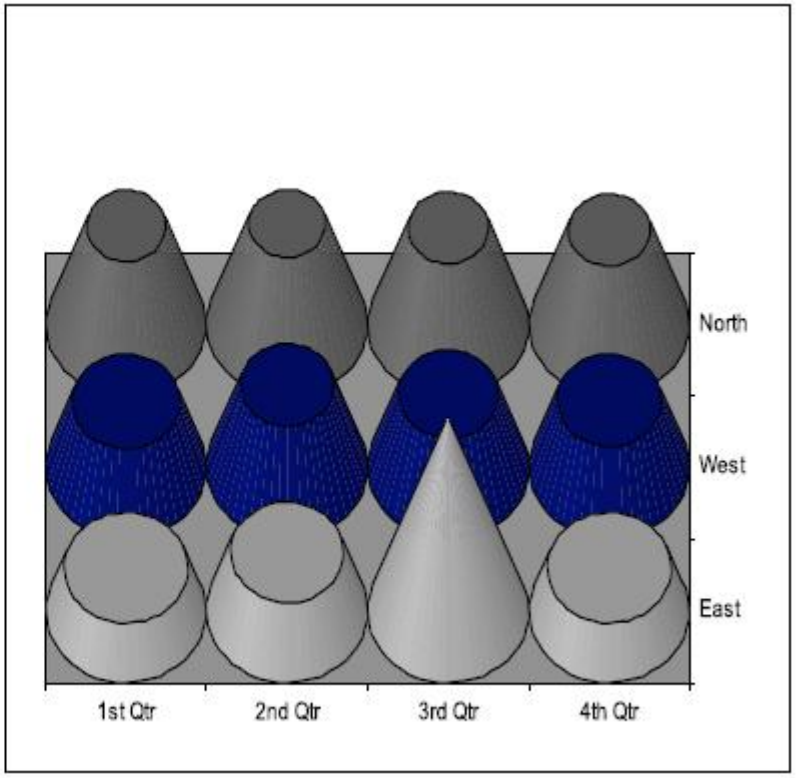
MVC

- Model must notify views of change.
- Views must keep themselves up to date.
- Several controllers e.g. command keys, pop-up menus can be used. Usually controllers organised in a class hierarchy -> sub-classing



Two views of the data





MVC Advantages

- MVC decouples the model management (data) from the representation (views) and the reaction to user input (controllers). Increased flexibility.
- Allows to have multiple (synchronized views) on the same data.
- Allows to associate different controllers with each of the views if needed. Change the way in which the interface reacts without changing the interface.
- Views can be easily composed.

Advanced Patterns

Next Step

Creational Patterns

- *Singleton - for creating classes which must have only a* single instance (e.g. a printer spooler)
- *Factory method - used when a class can't anticipate the* class of objects it must create but it wants its subclasses to specify the objects it creates
- *Abstract factory - provides an interface for creating* families of related objects without the need to specify their concrete classes
- *Prototype - creating objects from object instances*

Structural Patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Patterns

- Behavioural patterns are concerned with algorithms and the assignment of responsibilities.

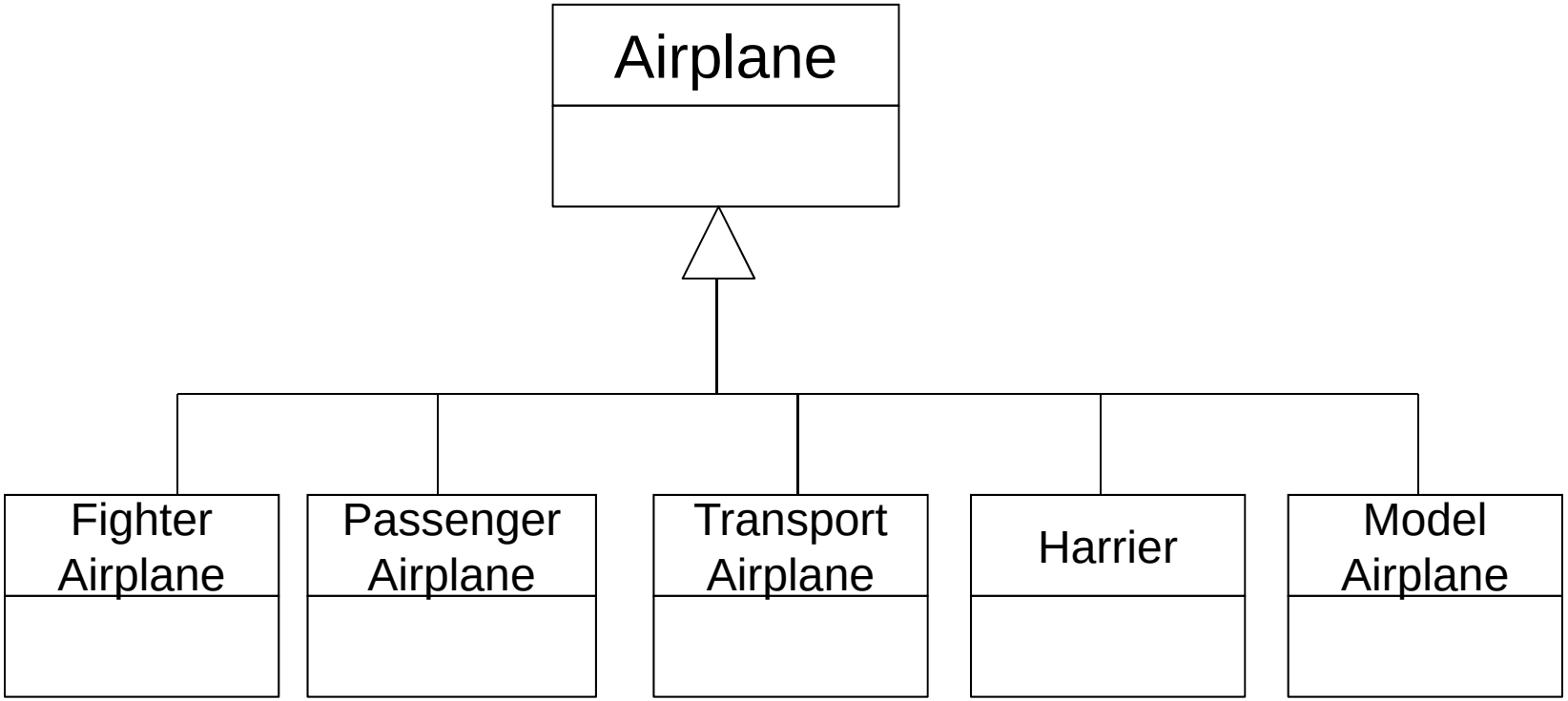
- Chain of Responsibility
- Command
- Interpreter
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Underlying Ideas:Programming to an interface

- Programming to an interface, not to an implementation!
- Don’t declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class.
- You have to instantiate concrete classes (specify an implementation) at some point in your system, and the creational patterns let you do just that.

Strategy

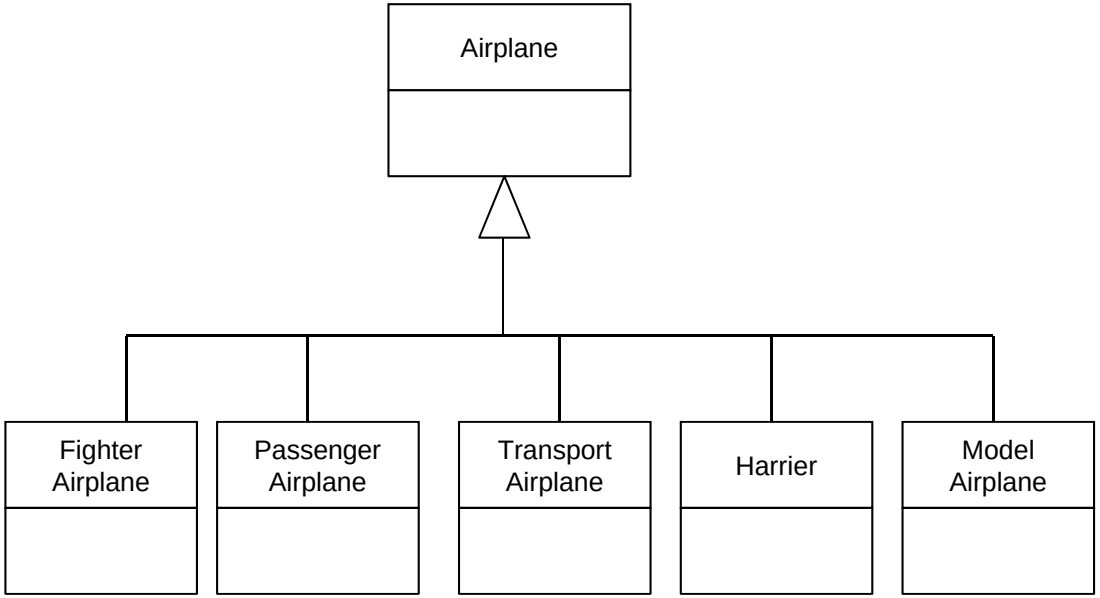
Airplanes, Domain Model



Airplanes, Use case Model

According to the use case model

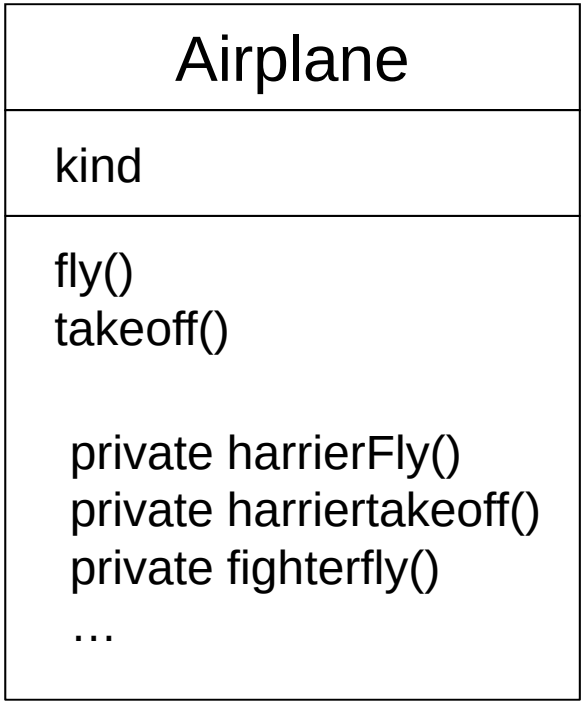
- All airplanes can fly except for the model airplane
- All airplanes can takeoff except for the model airplane
- Even among the airplanes that can fly there is a difference in how they takeoff() and fly().



Airplanes, Naïve Idea

Recall: a conceptual classes does not necessarily translate to a design class.

One big class that has an attribute “*kind*” and methods which choose the right option according to the attributes value.



Airplanes, Naïve Idea

Pros

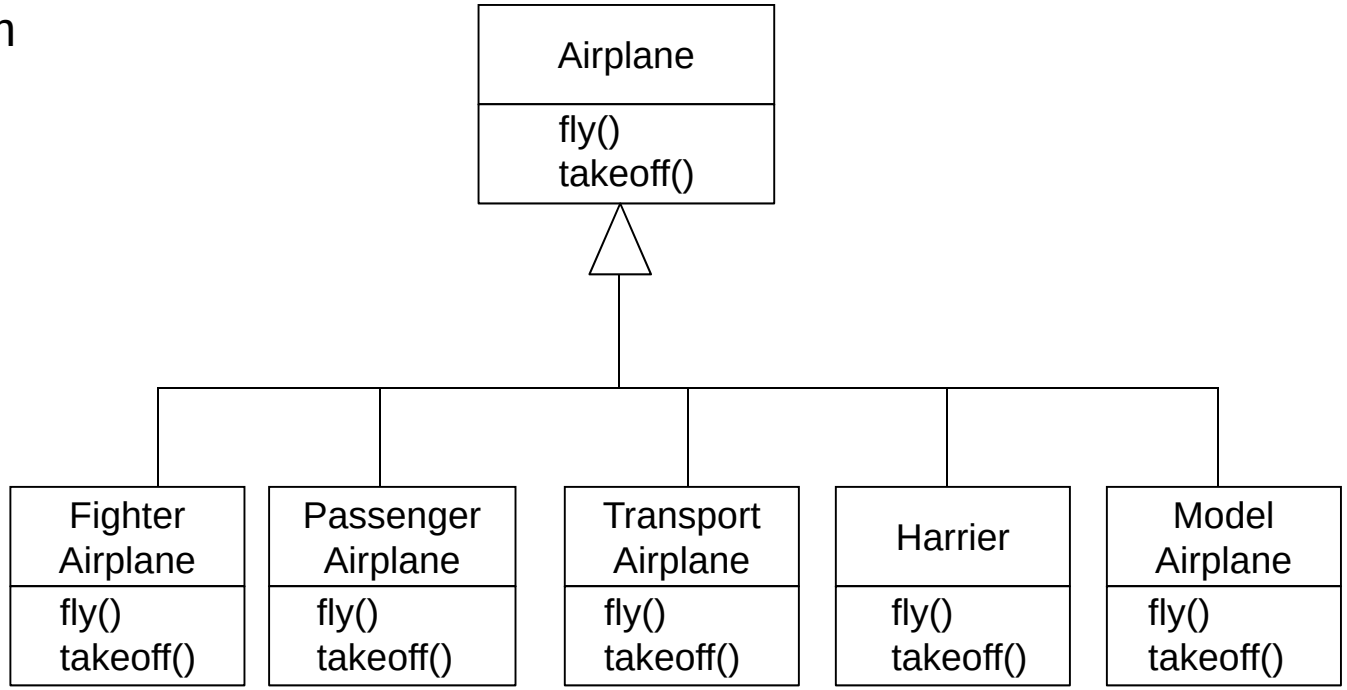
- Low coupling
- Small number of classes
- Code reuse

Cons

- Low Cohesion
- Messy code
- Messy maintenance

Airplanes, Polymorphism

Polymorphism



Airplanes, Polymorphism

Pros

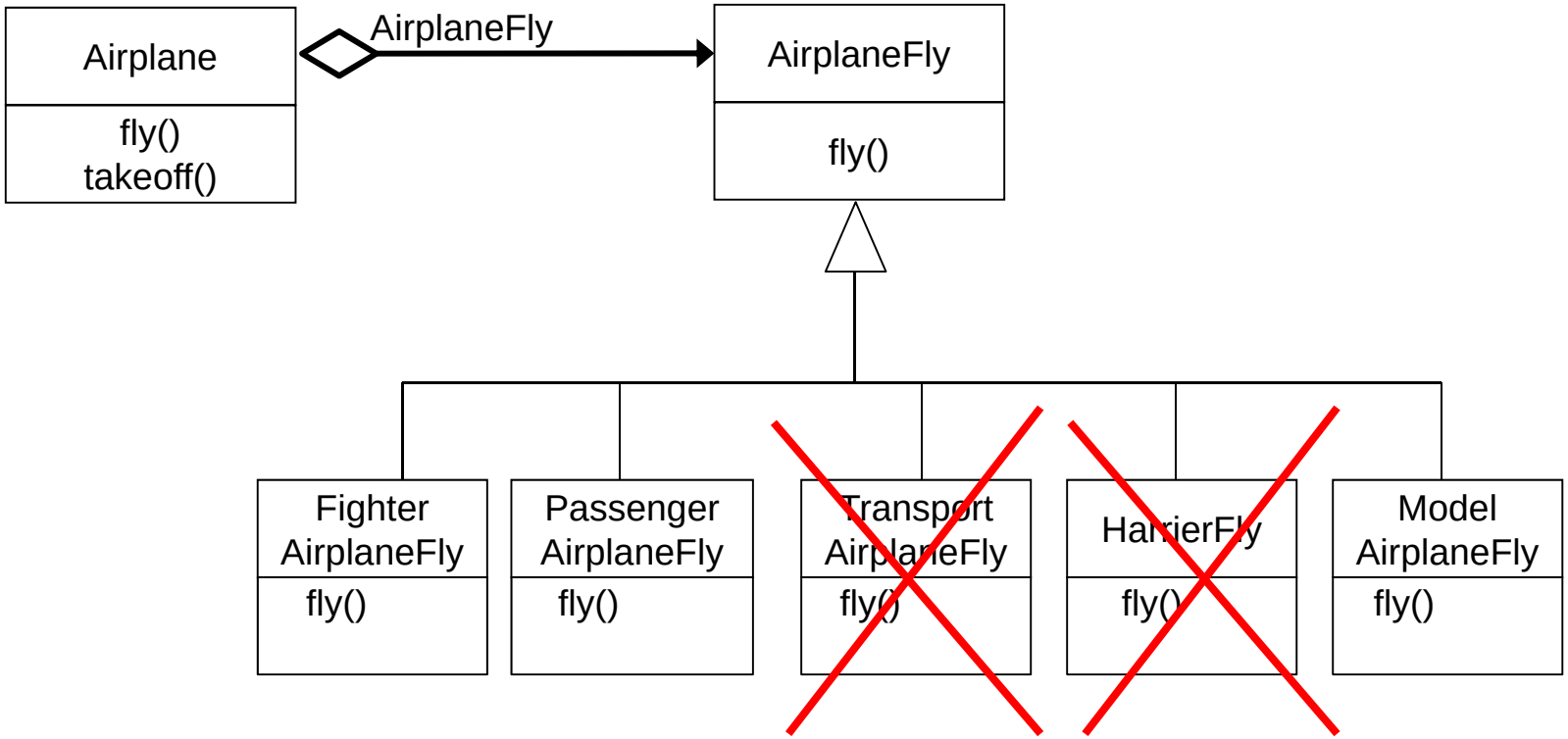
- Cleaner code

Cons

- Higher coupling
- What if some of the classes fly the same way?
- Low Cohesion

Airplanes, Delegate, Encapsulate

- Airplane class delegates the responsibility for fly



Airplanes, Delegate Encapsulate

Pros

- High cohesion
- More flexibility
- Easy to add new possible ways to fly
- Better coupling

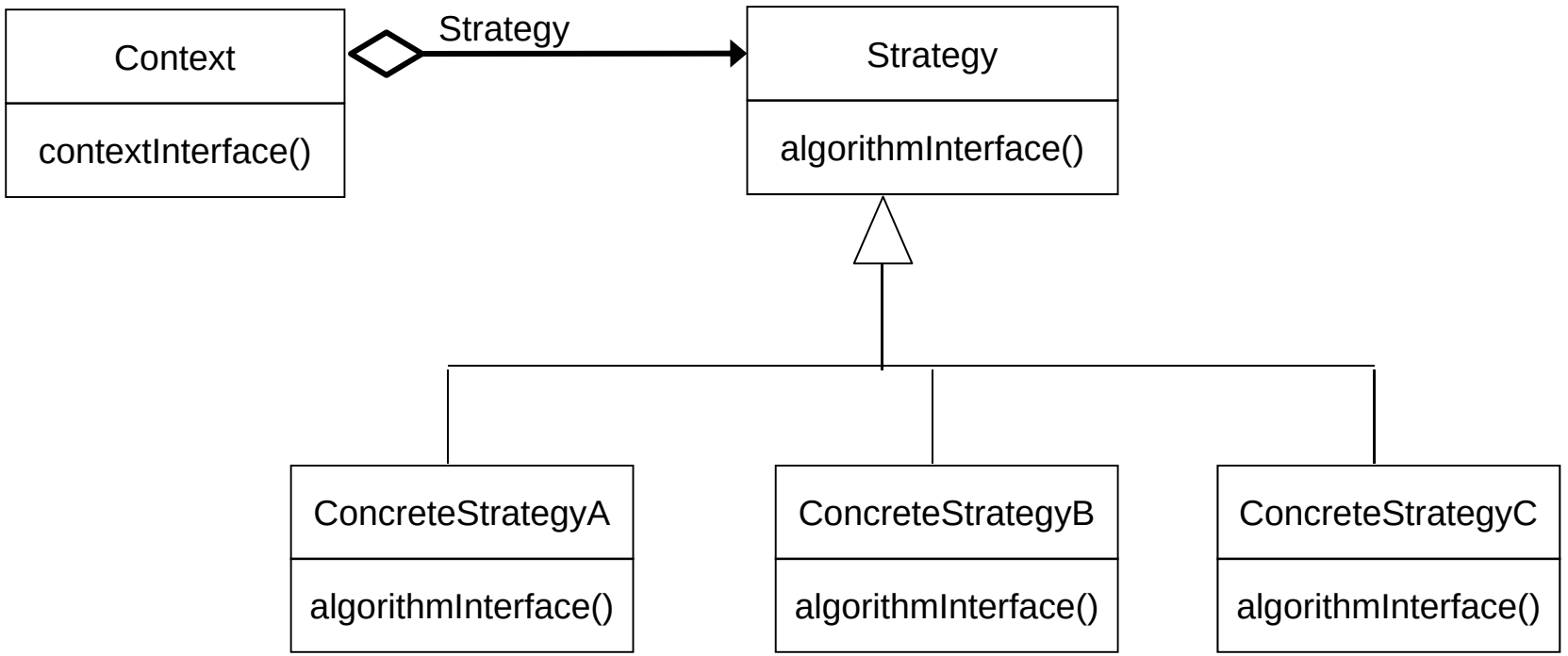
Cons

- More classes
- More work

Strategy Pattern

Problem: How do you make a family of algorithms interchangeable

Solution:



Strategy Pattern

Pros

- Simpler code
- High Cohesion
- Easy to add new possible ways to fly
- Better coupling

Cons

- More classes
- More coupling
- More communication between objects

Delegation

- It is a way of making composition as powerful for re-use as inheritance. Two objects are involved in handling a request: a receiving object delegates operations to its delegate.