Design & Implementation

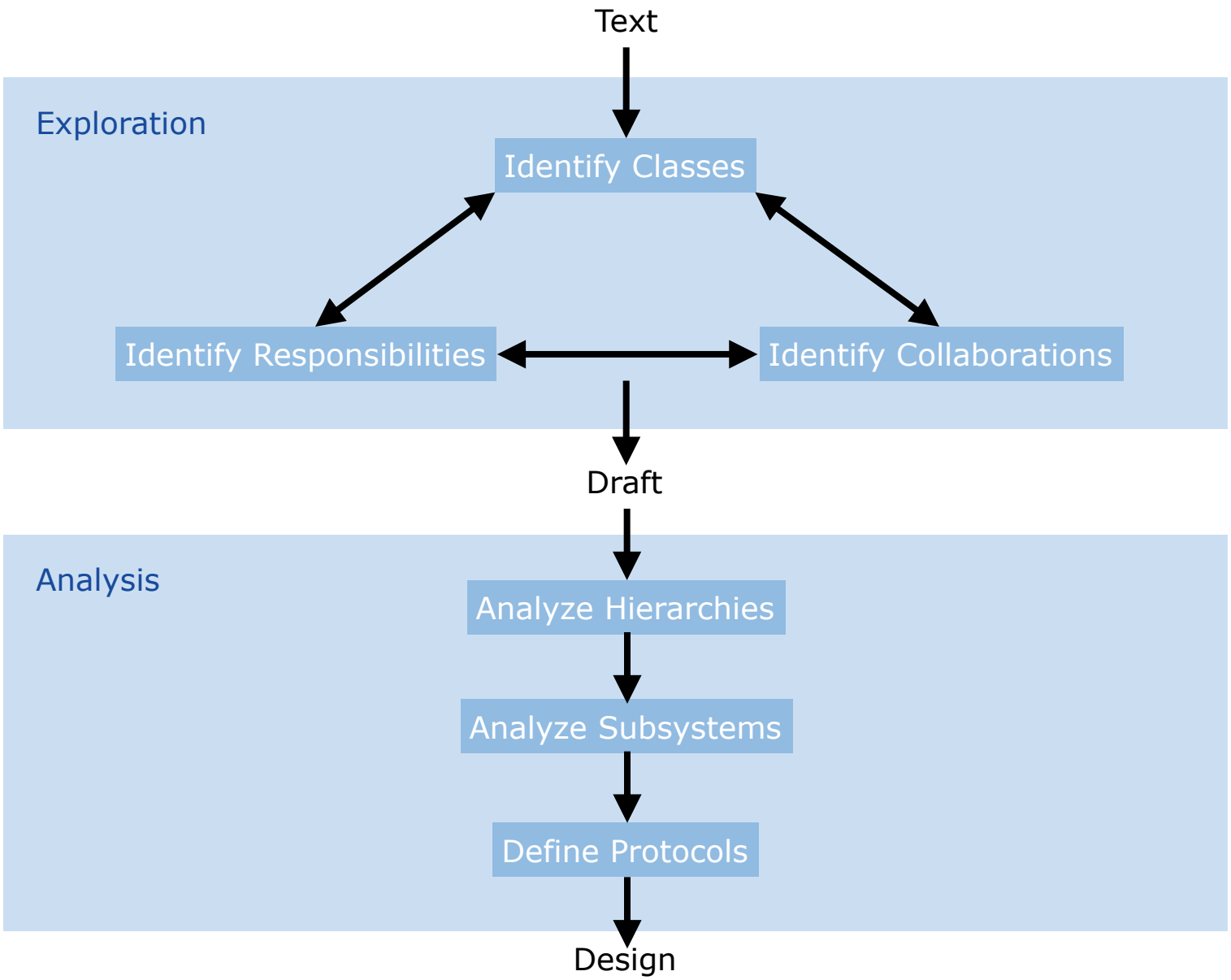
Overview

Contents & Goals

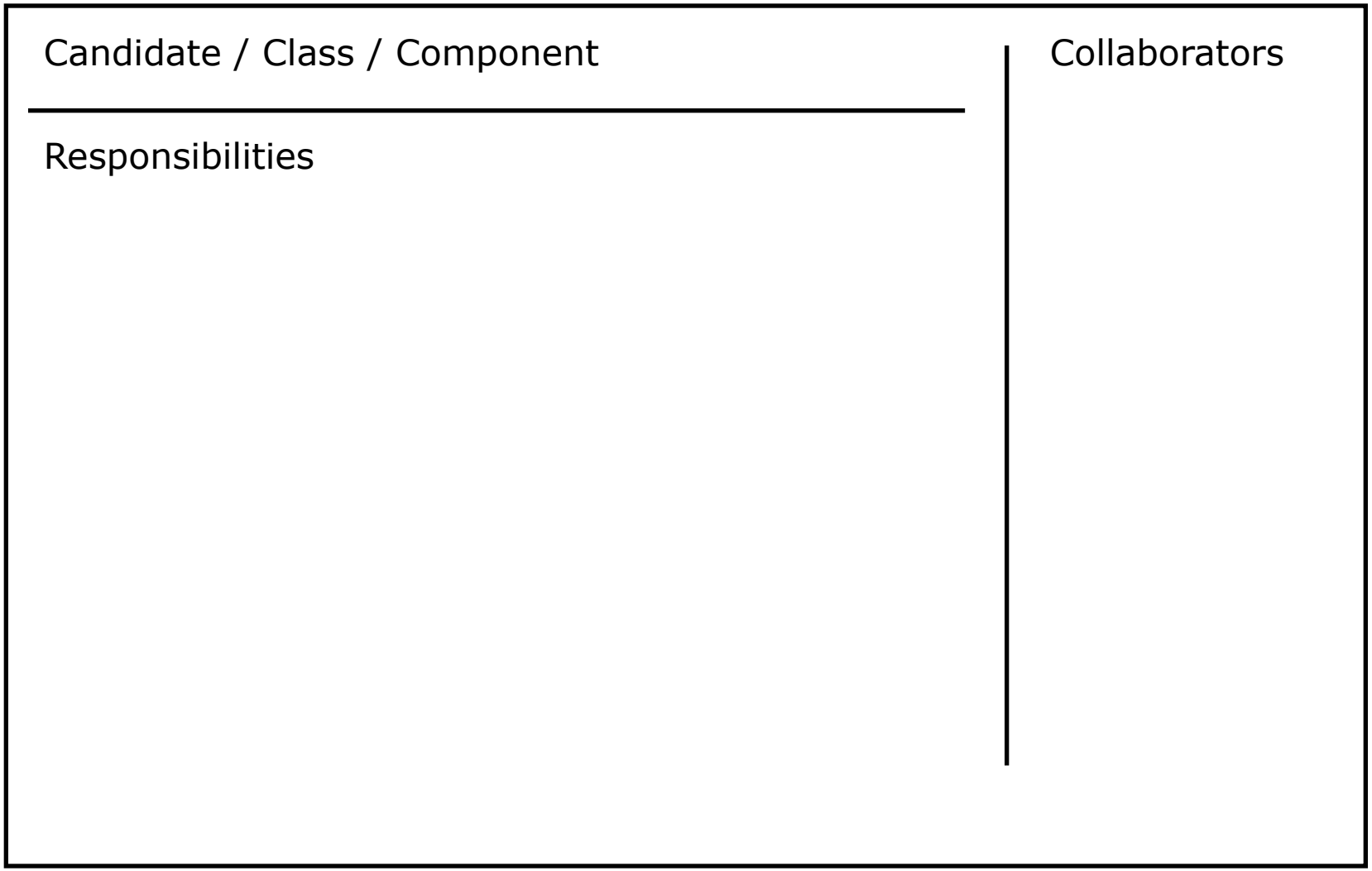
- CRC – Classes Responsibilities Collaborators
- JBricks – Design and Implementation

- Application of object-oriented design
- Having fun with design & implementation

Responsibility-Driven Design



CRC Cards


Candidate / Class / Component
Collaborators
Responsibilities

CRC Cards – Approach

Write up a description of the problem domain

Identify objects (nouns in description)

Identify responsibilities (verbs in description)

Assign responsibilities to classes

Refine responsibilities

Identify collaborations among classes

Classes without collaborators are put aside

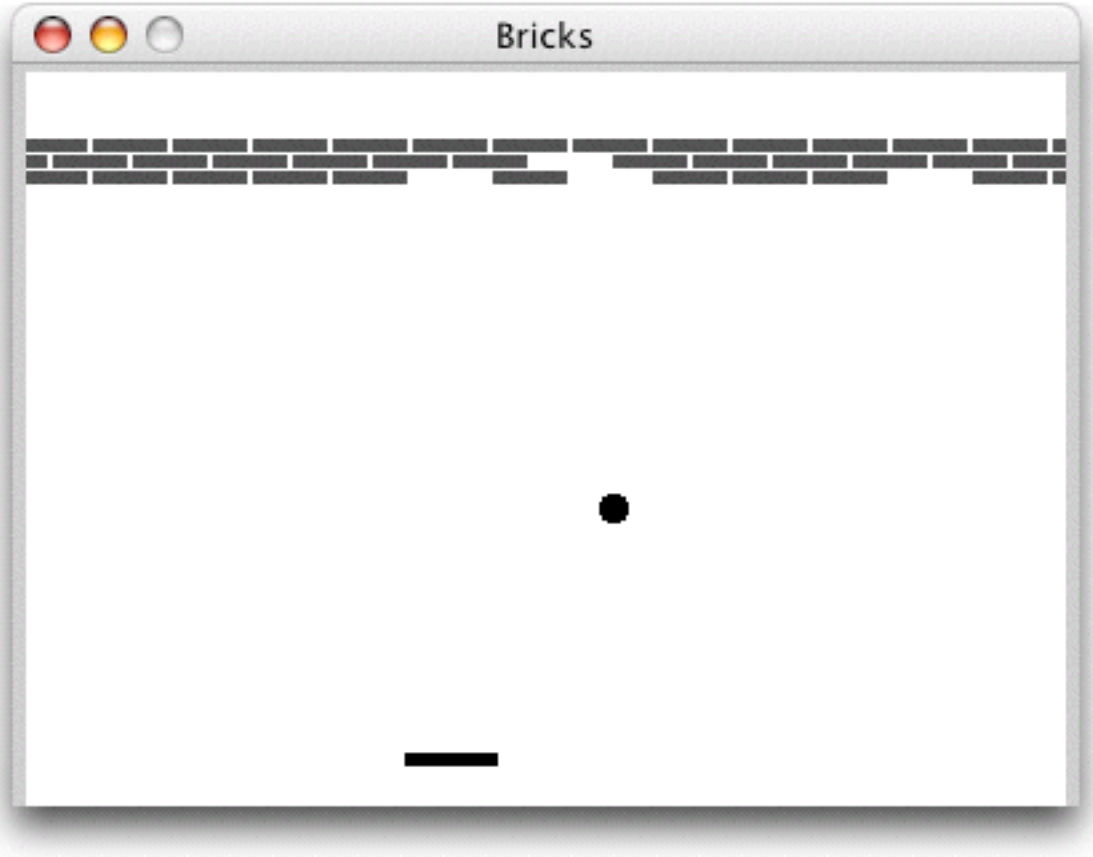
Refine structures

Group responsibilities in contracts (interfaces?)

Create subsystems based on collaborations

Add details

Bricks



Bricks – Description

The goal of the video game Bricks is to make all bricks vanish by hitting a ball against them. The bricks are horizontally aligned in three rows in the upper third of the rectangular game area. The game area is bordered on the top and on the sides, but open at the bottom.
The user can deflect the ball with a paddle. The paddle can be moved horizontally between the game borders by means of the mouse. The ball bounces off the bricks, the game borders, and the paddle. If the ball drops to the bottom of the game area it is lost. If a brick is hit by the ball it vanishes. The user gets three balls per game.

Bricks – Description

The goal of the video game Bricks is to make all bricks vanish by hitting a ball against them. The bricks are horizontally aligned in three rows in the upper third of the rectangular game area. The game area is bordered on the top and on the sides, but open at the bottom.
The user can deflect the ball with a paddle. The paddle can be moved horizontally between the game borders by means of the mouse. The ball bounces off the bricks, the game borders, and the paddle. If the ball drops to the bottom of the game area it is lost. If a brick is hit by the ball it vanishes. The user gets three balls per game.

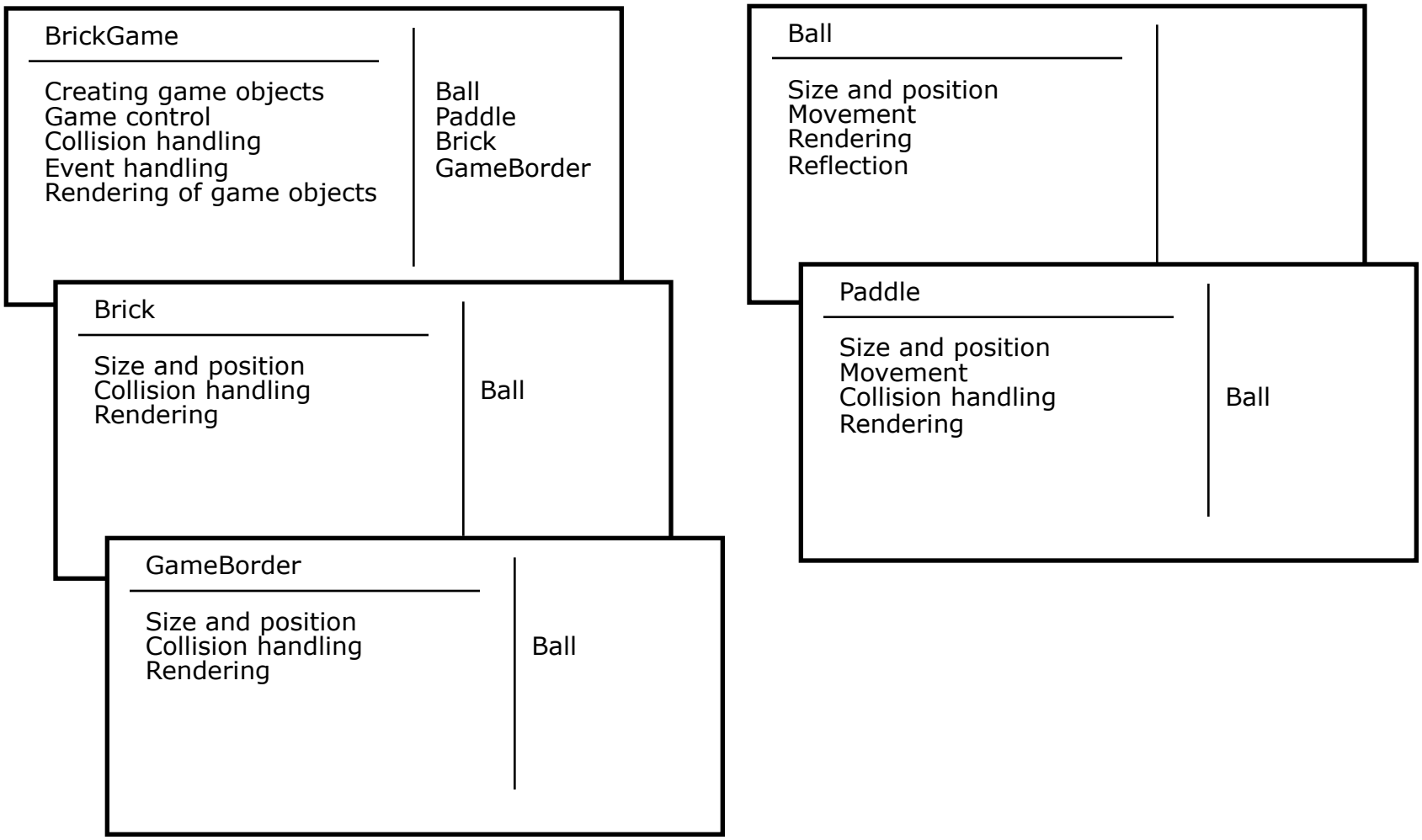
Bricks – Description

The goal of the video game Bricks is to make all bricks vanish by hitting a ball against them. The bricks are horizontally aligned in three rows in the upper third of the rectangular game area. The game area is bordered on the top and on the sides, but open at the bottom.
The user can deflect the ball with a paddle. The paddle can be moved horizontally between the game borders by means of the mouse. The ball bounces off the bricks, the game borders, and the paddle. If the ball drops to the bottom of the game area it is lost. If a brick is hit by the ball it vanishes. The user gets three balls per game.

Bricks – Objects

Objects (Nouns)	Responsibilities (Verbs)
Brick Game	Vanish
Ball	Hit
Brick	Deflect
Paddle	Move
Game Border	Bounce Off
	Lose

Bricks – CRC-Cards



Refinements

- Creating game objects
  - List of bricks, game borders, paddle
- Game control
  - Is the ball still inside the game border?
  - How many balls are still available?
  - Paddle movement
  - Moving the ball
- Rendering game objects
  - Drawing bricks, game borders, paddle, ball
- Collision handling
  - Is there a collision between the ball and game border/brick/paddle)
  - Brick has to be removed on a collision
  - Reflecting the ball

Refinements

- Event handling
  - Pressing any key terminates the game
  - Paddle moves according to the mouse
- Game object position
  - Brick, Paddle, Ball, GameBorder
  - Bounding rectangle is sufficient
- Moving the game objects (paddle, ball)
  - Setting a new game object position
  - Paddle is moved by the mouse
  - Ball is moved at regular intervals (velocity, direction)

Abstractions

- GameObject
  - Abstract class for GameBorder, Brick, Ball, and Paddle
  - Maintains bounding rectangle

- MovingGObject
  - Derived from GameObject
  - Object can be moved to a new position
  - Ball and Paddle are derived from MovingGObject

Scenarios (Use Cases)

- Gameplay
  - Create game objects (bricks, game borders, paddle)
    While balls are available
        Wait for releasing the next ball (by pressing mouse button)
        Play ball

  - Play ball
      Create ball
      While game is not yet terminated
        Move ball
        Check for collisions
        Move paddle

  - Termination condition
      Key pressed
      Ball leaves game area
      All bricks are removed

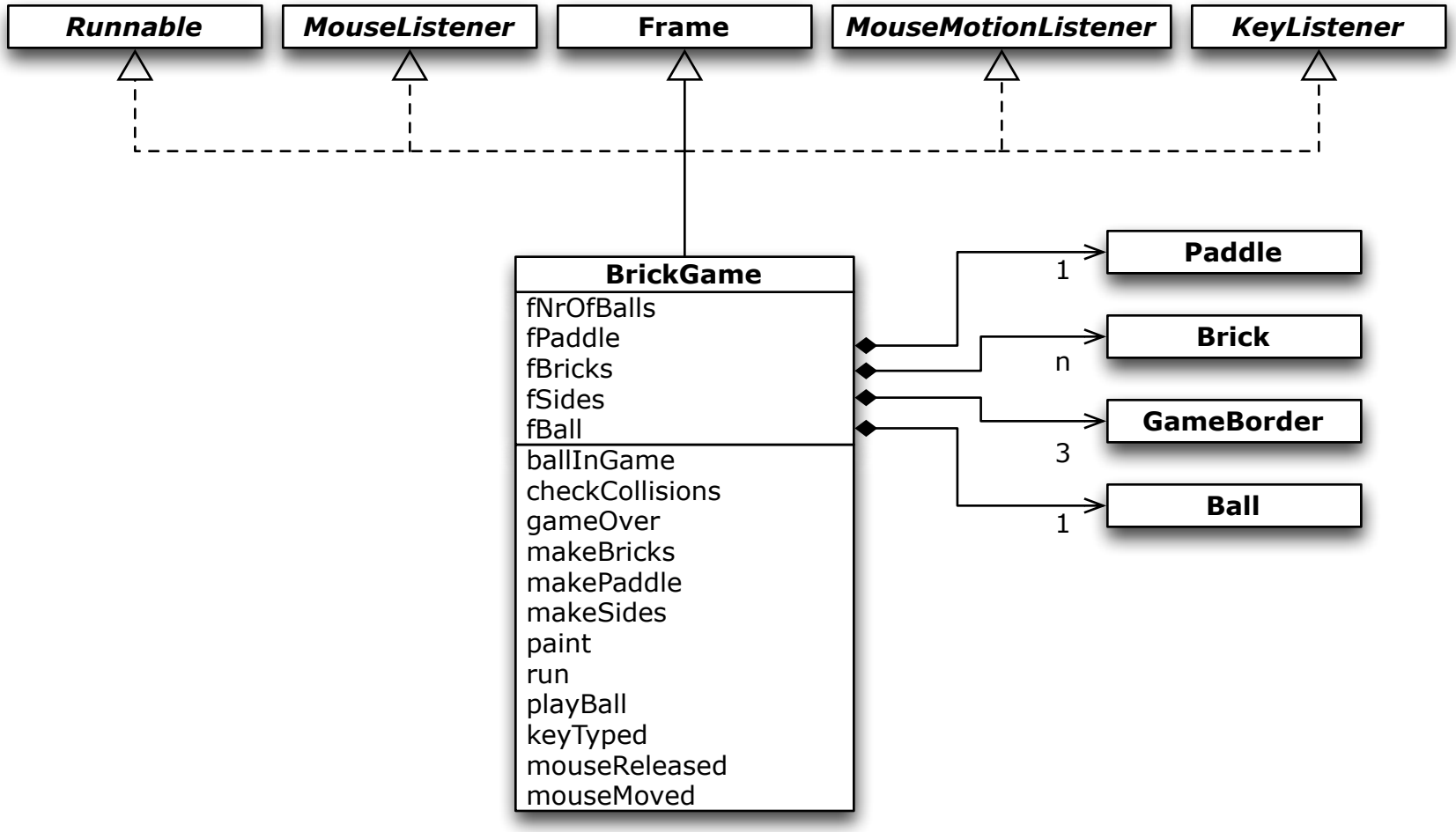
Scenarios (Use Cases)

- Collision control
  - Collision: Bounds of two game objects are overlapping
  - Collision control is executed between erasing and rendering
  - Who knows the obstacles?
    - BrickGame
  - If object is a brick it must be removed
  - Collision makes the ball reflect
    - Reflection is defined by the obstacle
    - ➜ GameObject gets the attribute reflection

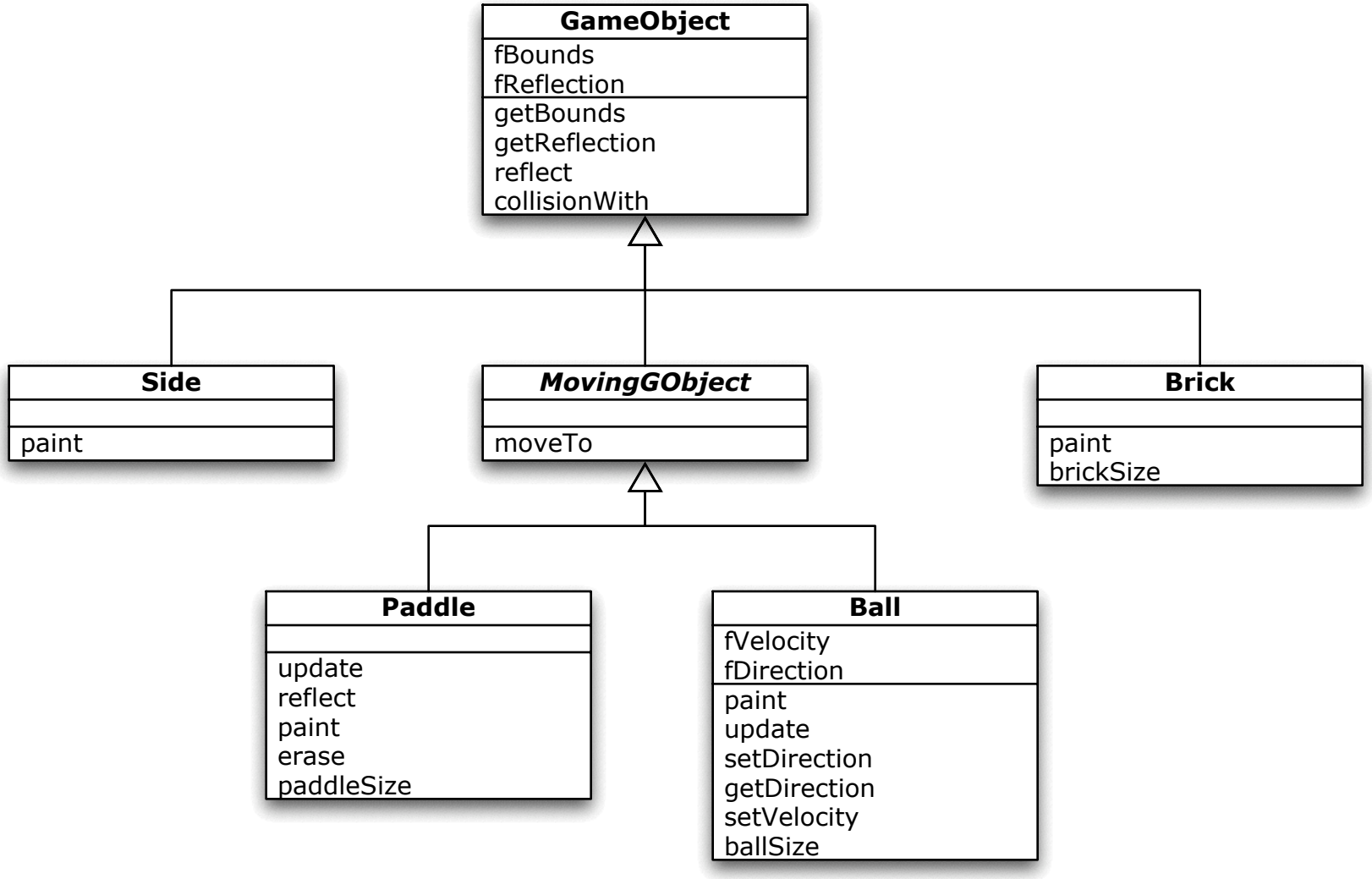
Implementation in Java

- Bricks is the „main class“
  - Starts and stops BrickGame
  - Creates user interface for starting and stopping game
- BrickGame is derived from Frame
- Event handling is not straightforward
  - Methods keyPressed() and mouseMoved() are called from Java
  - Game control runs in the EDT (event dispatching thread)
  - Does not release control
    ➜ no events can be delivered
  - Solution: BrickGame is a thread of its own and releases control regularly

Design


Runnable
MouseListener
Frame
MouseMotionListener
KeyListener
BrickGame
fNrOfBalls
fPaddle
fBricks
fSides
fBall
ballInGame
checkCollisions
gameOver
makeBricks
makePaddle
makeSides
paint
run
playBall
keyTyped
mouseReleased
mouseMoved
1
Paddle
n
Brick
3
GameBorder
1
Ball

Design


GameObject
fBounds
fReflection
getBounds
getReflection
reflect
collisionWith
Side
paint
MovingGObject
moveTo
Brick
paint
brickSize
Paddle
update
reflect
paint
erase
paddleSize
Ball
fVelocity
fDirection
paint
update
setDirection
getDirection
setVelocity
ballSize

Implementation – Issues

- Ball
  - Velocity depends on angle
  - Ball is mostly too fast
- Paddle
  - Damages game border
- Collision control is inefficient and imprecise
  - Sometimes the ball just disappears
  - Collision control is always executed even if the ball is far away from any obstacle

Design – Issues

- Responsibilities are not where they belong to
  - BrickGame has too many responsibilities
    - As a consequence too many parameters have to be passed
  - Collision control is in BrickGame and not in Ball
- GameObjects
  - No distinction between simple and composite objects
  - BrickGame must distinguish
- Reflection behavior is inflexible
  - Cannot be configured
  - New reflection behavior cannot be added easily
- Few abstractions
  - ➜ Hardly flexible und reusable

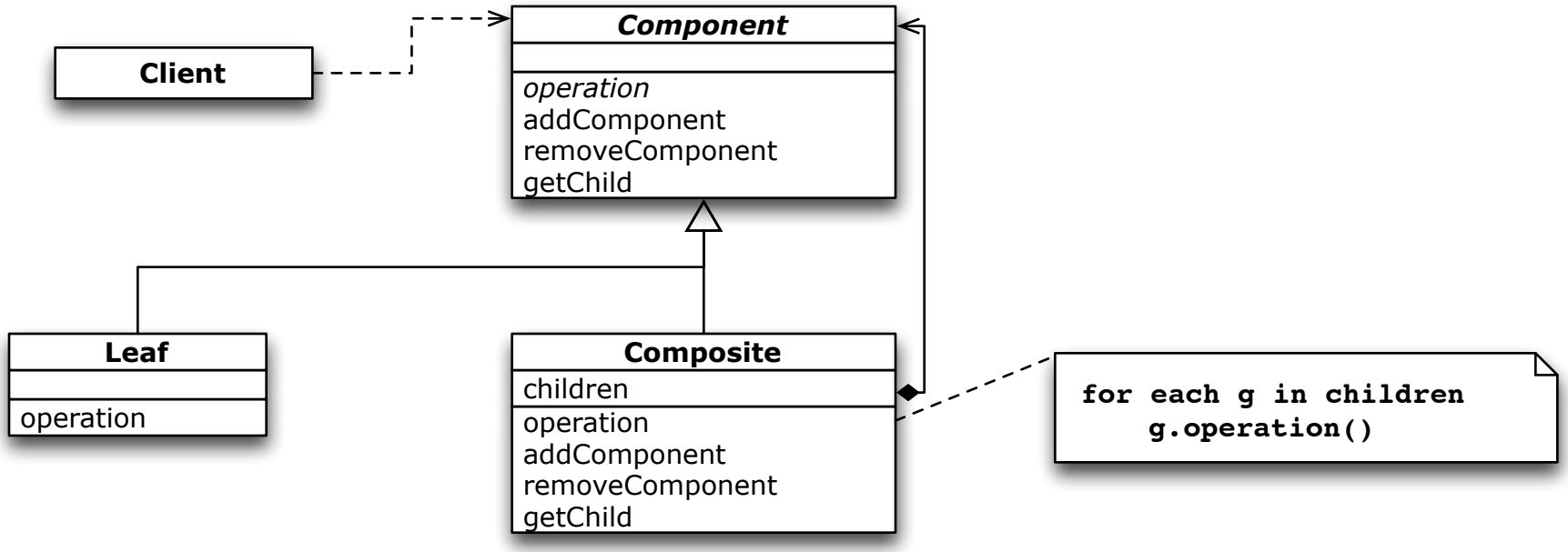
- Many implementation issues result from design issues

Composite

- Intent
  - Treat individual objects and compositions of objects uniformly. Objects are composed into tree structures.
- Motivation
  - JBricks: treat rows of bricks in the same way as a single brick
- Applicability
  - Representation of object hierarchies (dynamically)
    - Object consists of a set of objects
    - Object is contained in some other object (whole-part hierarchies)
  - Clients should be able to ignore the difference between a single object and a set of objects

Composite

- Structure



Composite

- Participants
  - Component — declares interface for objects in the composition
implements default behavior
declares interfaces for accessing child/parent objects
  - Leaf — represents leaf objects in the composition
defines behavior for primitive objects
  - Composite — defines behavior for components with child objects
stores child components
implements child-related operations
  - Client — manipulates objects in the composition through the component interface
- Consequences
  - Complex object structures can be built dynamically and recursively
  - Clients don’t need to differentiate between simple and complex objects
  - Makes it easy to add new kinds of objects
  - Restricting object structure is not easy

Composite

- Implementation
  - Explicit references to parent object
  - Sharing components
  - Maintaining child objects (Component or Composite)
  - Child ordering
  - Caching
- Known uses
  - Can be found in almost any object-oriented system
  - JDK: java.awt.Container, java.io.SequenceInputStream
- Related patterns
  - Decorator and composite are often used together
  - Iterator pattern can be used for traversing a composite
  - Visitor pattern localizes operations in object hierarchies

Redesign (Composite)

- CompositeGObject
  - Composite
  - GameObject does not maintain child objects
  - Reference to parent object
  - Two subclasses:
    - Wall is a composite of Bricks
    - GameBorder is a composite of Sides (formerly known as GameBorder)

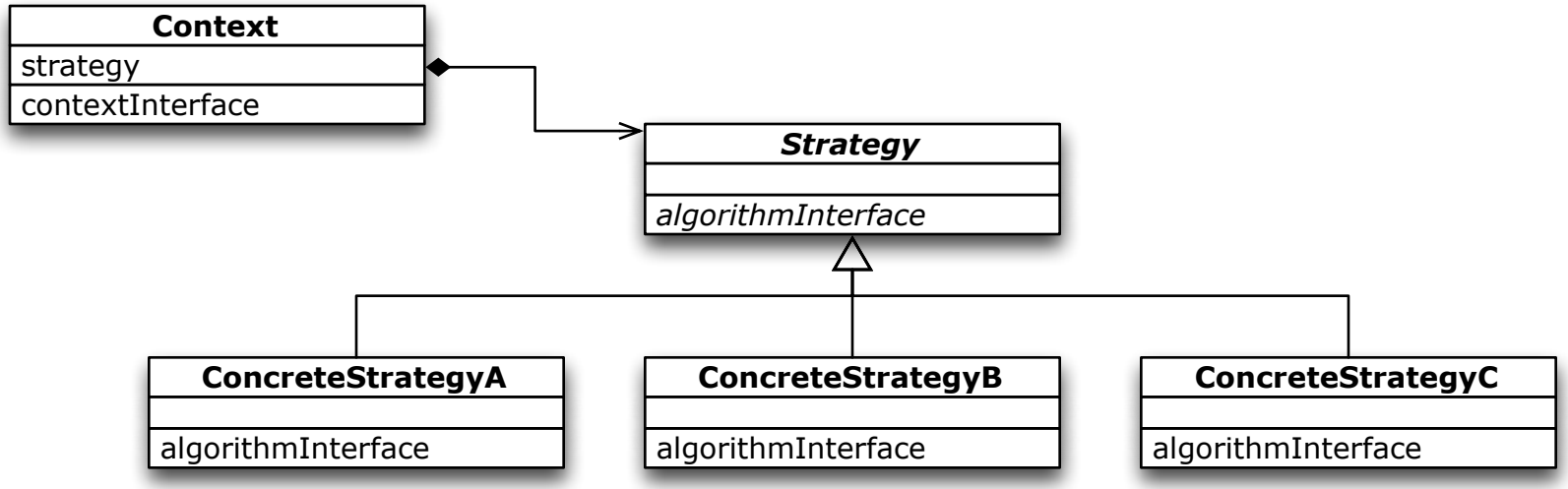
- Consequences
  - BrickGame can shed some responsibilities
    - Wall and GameBorder are creating themselves
  - Collision control can be moved to GameObject (Ball)

Strategy

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Also known as: Policy
- Motivation
  - Reflection as part of a GameObject is inflexible
  - Reflection can be modeled as an object
- Applicability
  - Several classes differ only in their behavior
  - Several variants of an algorithm are needed
  - Clients are shielded from algorithms
  - A class defines behavior variations and uses them depending on the context (case/switch statement)

Strategy

- Structure



Strategy – Code Initial Position

- Example

```
public void execute {
    switch (condition) {
        case condition1:
            executeAlgorithm1();
            break;
        case condition2:
            executeAlgorithm2();
            break;
        case condition3:
            executeAlgorithm3();
            break;
        default:
    }
}
```

- Alternative example
  - The algorithm variations are realized in different implementations of execute() along the inheritance hierarchy

Strategy – Code Target Solution

- Context

```
public class Context {
    private Strategy  fStrategy;

    public void setStrategy(Strategy strategy) {
        fStrategy = strategy;
    }

    public void execute() {
        fStrategy.executeAlgorithm()
    }
}
```

- Usage

```
Context context = new Context();
context.setStrategy(new MyStrategy());
context.execute();
```

Strategy – Code Target Solution

- Strategy implementations

```
public class Strategy {
    public void executeAlgorithm() {
        algorithmImplementation1()
    }
}
```

```
public class MyStrategy extends Strategy {
    public void executeAlgorithm() {
        algorithmImplementation2()
    }
}
```

Strategy

- Participants
  - Strategy — declares interface common to all supported algorithms. Context uses this interface
  - ConcreteStrategy — implements algorithm using the Strategy interface
  - Context — is configured with a ConcreteStrategy
maintains reference to Strategy object
may define an interface for the Strategy object to access its data
- Consequences
  - Families of related algorithm
  - Alternative to subclassing
  - Eliminating case/switch statements
  - Implementation alternatives (e.g. time vs. memory)
  - Communication overhead between Context and Strategy
  - Increased number of objects

Strategy

- Known uses
  - JDK: java.awt.LayoutManager, Collections.sort (Comparator)
  - ET++: line breaking algorithms

- Related patterns
  - Strategies are often implemented as flyweights
  - Strategies use delegation to vary an entire algorithm, whereas Template Methods use subclassing to vary parts of an algorithm

Redesign (Strategy)

- Reflection
  - Strategy
    - Specifies general and specific reflection behavior
    - Subclasses: HReflection, VReflection, and RandomReflection
  - RandomReflection: Decorator
  - HReflection and VReflection are implemented as Singletons

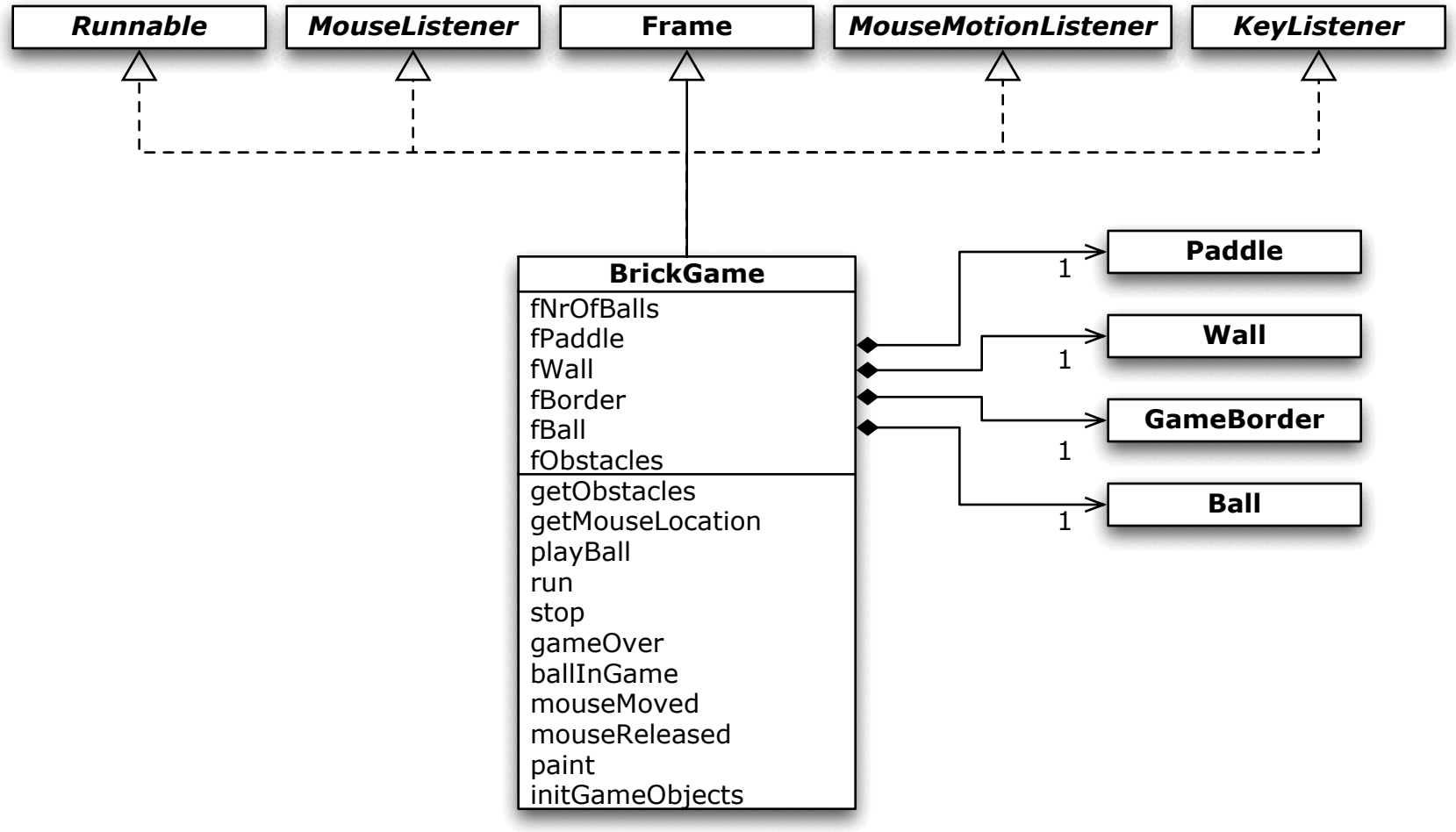
```
public abstract class Reflection {

    public void reflect(Ball ball, GameObject reflector) {
        Point r = reflector.adaptReflection(reflector.getReflection());
        ball.setDirection(new Point(ball.getDirection().x*r.x,
                                    ball.getDirection().y*r.y));
    }
}
```

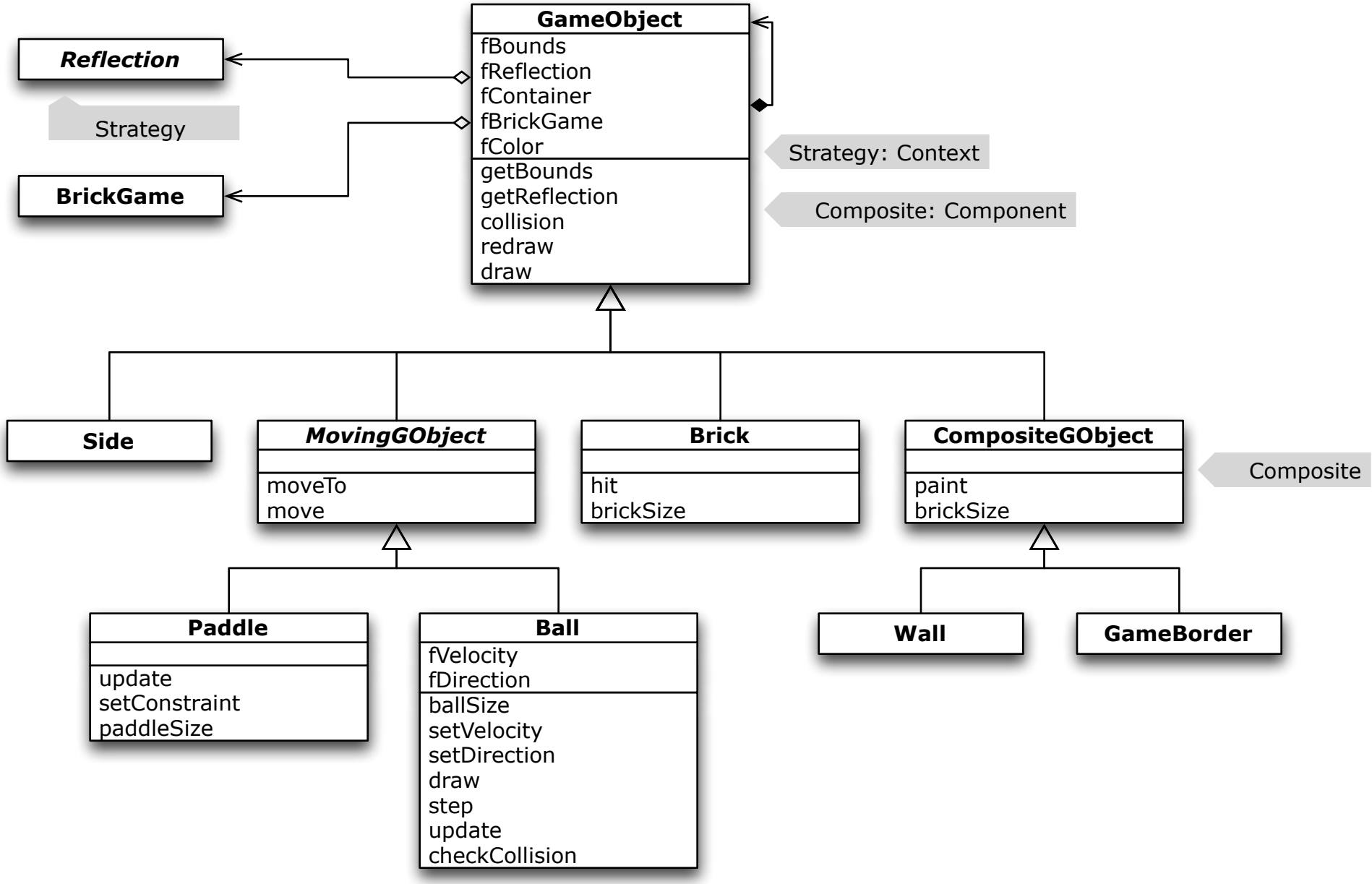
Consequences of Redesign

- More sensible distribution of responsibilities
  - BrickGame is still the manager
    ➜ Considerably less responsibilities, rather coordinating
- Better abstractions
  - GameObject
  - CompositeGObject
  - MovingGObject
  - Reflection
- Methods are simpler with less parameters
- Easier adaptability (small inheritance interface)

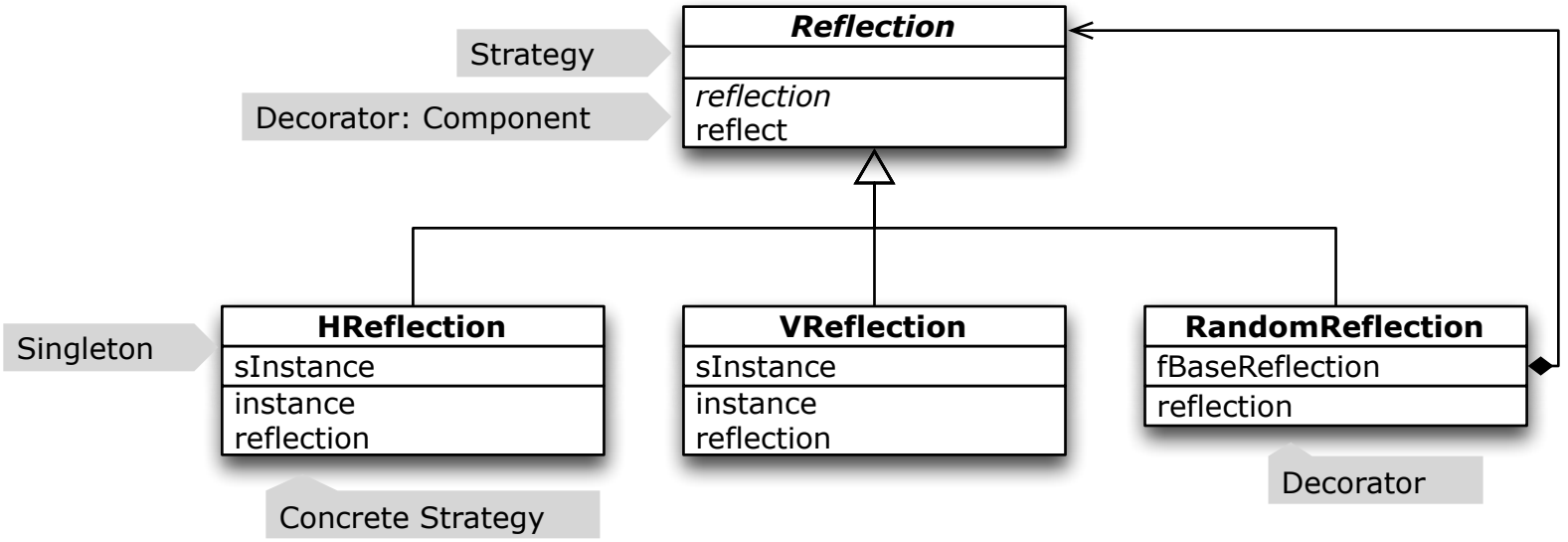
Redesign (BrickGame)



Redesign (GameObject)



Redesign (Reflection)



Summary

- Good design can hardly be planned
- Good design can help to optimize
  - Algorithm ➜ factor 2, Design ➜ factor 5, Architecture ➜ factor 10
- CRC-Cards are cheap and effective
- Design methods are no guarantee for good design
  - Support for a design process
  - Means of communication
  - Essential are experience and creativity
- Software development is (among others) an iterative and creative process
- Architecture and abstractions are crucial

Literature

- Rebecca Wirfs-Brock: Object Design: Roles, Responsibilities, and Collaborations
- Simone, Bellin: The CRC Card Book