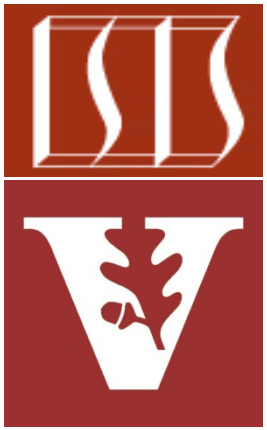
Android Concurrency: Java Synchronization & Scheduling Example (Part 1)

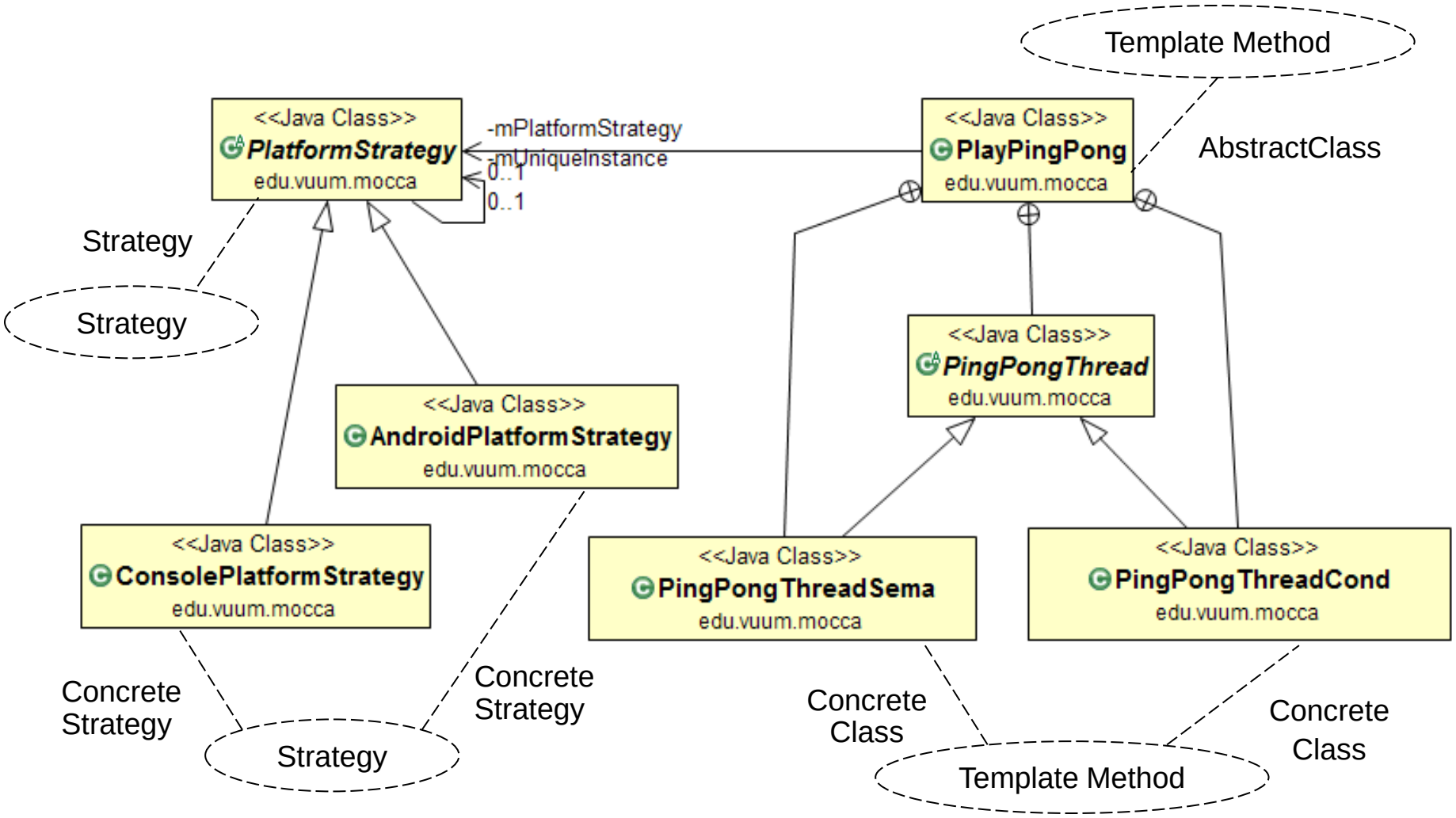


Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

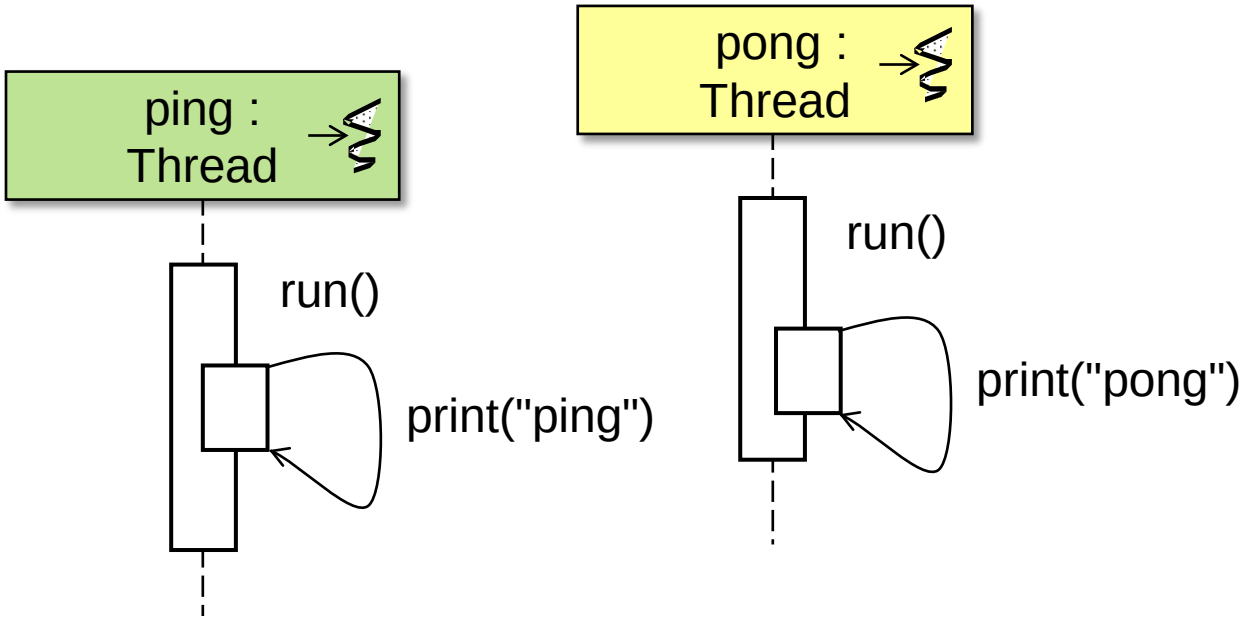
Learning Objectives in this Part of the Module

- Understand the design of a pattern-oriented framework schedules two Java threads that alternate printing “Ping” & “Pong” on the computer console



Outline of a Pattern-Oriented Framework Solution

- The earlier ping-pong program we analyzed was buggy since it lacked proper concurrency control



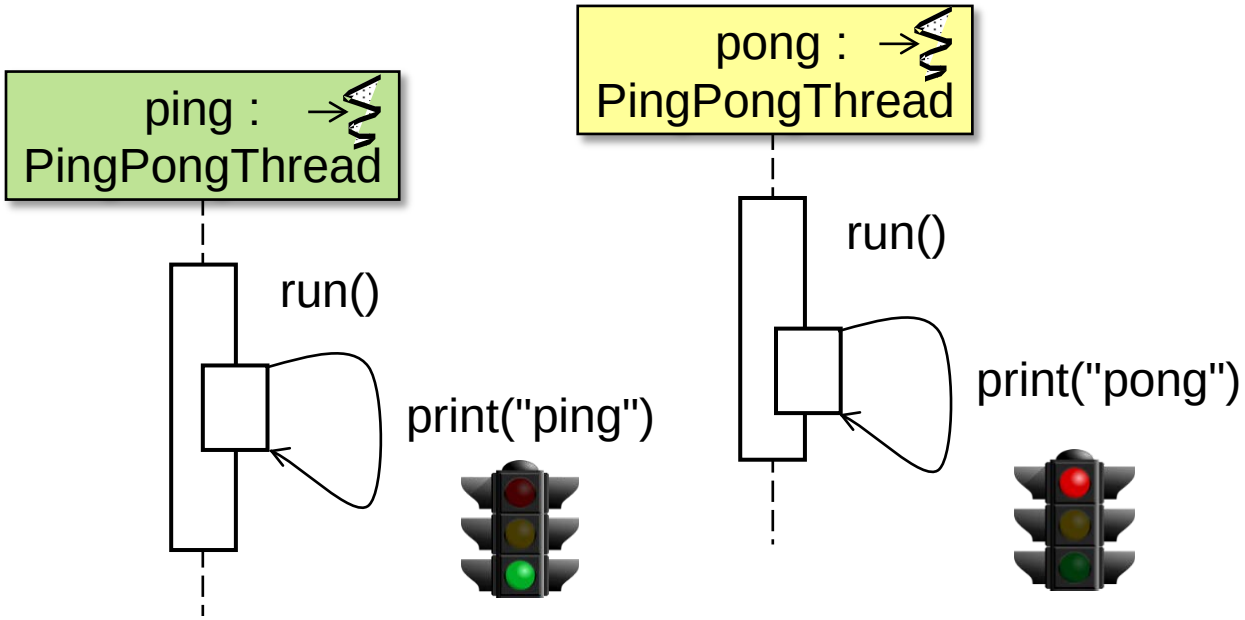

```
% java PingPongWrong
Ready...Set...Go!
Ping!(1)
Ping!(2)
Ping!(3)
Ping!(4)
Ping!(5)
Ping!(6)
Ping!(7)
Ping!(8)
Ping!(9)
Ping!(10)
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```

See prior part on “Motivating Java Synchronization & Scheduling Mechanisms”

Outline of a Pattern-Oriented Framework Solution

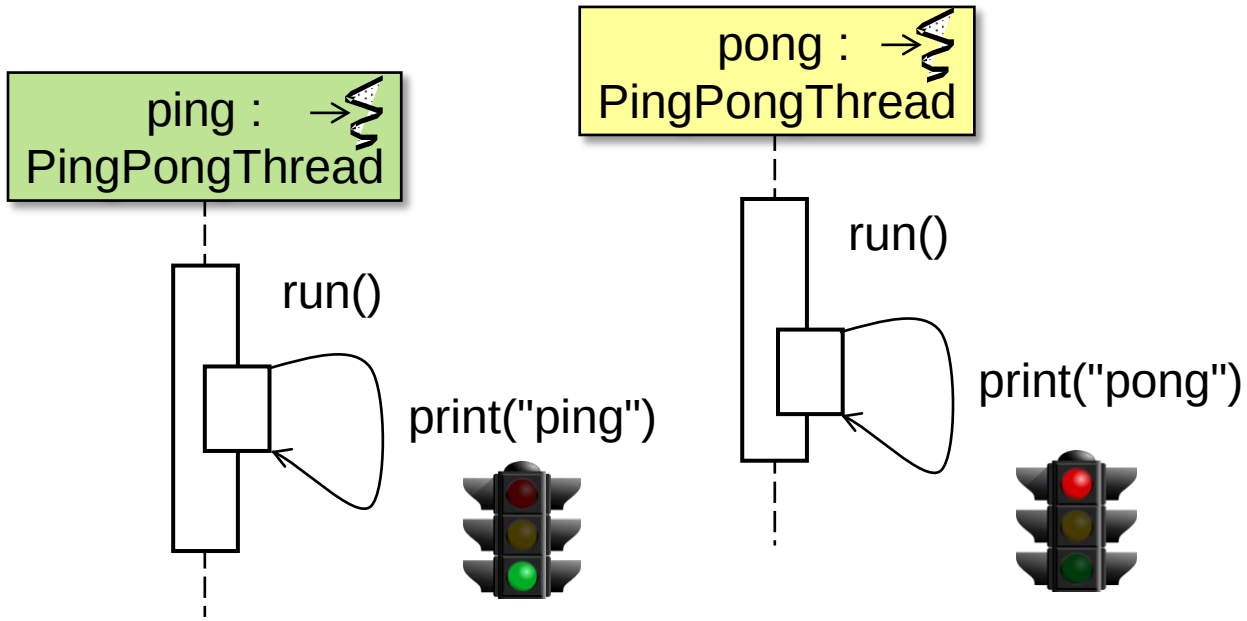
- The earlier ping-pong program we analyzed was buggy since it lacked proper concurrency control
  - This output's correct since it coordinates Thread interactions via Java synchronizers

```
% java PlayPingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Ping!(10)
Pong!(10)
Done!
```



Outline of a Pattern-Oriented Framework Solution

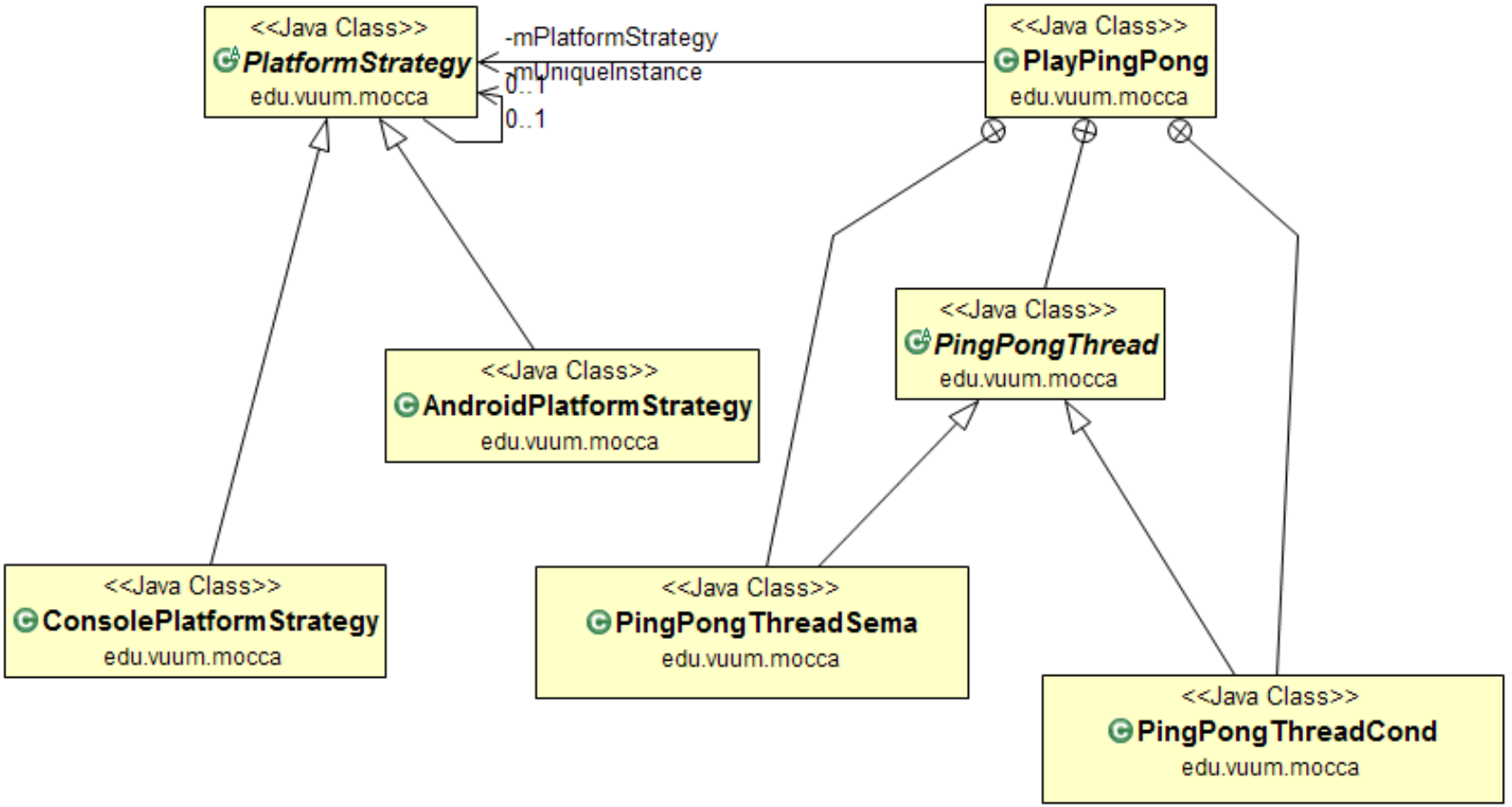
- The earlier ping-pong program we analyzed was buggy since it lacked proper concurrency control
  - This output's correct since it coordinates Thread interactions via Java synchronizers
    - e.g., Semaphores, ConditionObjects, & CountDownLatches




```
% java PlayPingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Ping!(10)
Pong!(10)
Done!
```

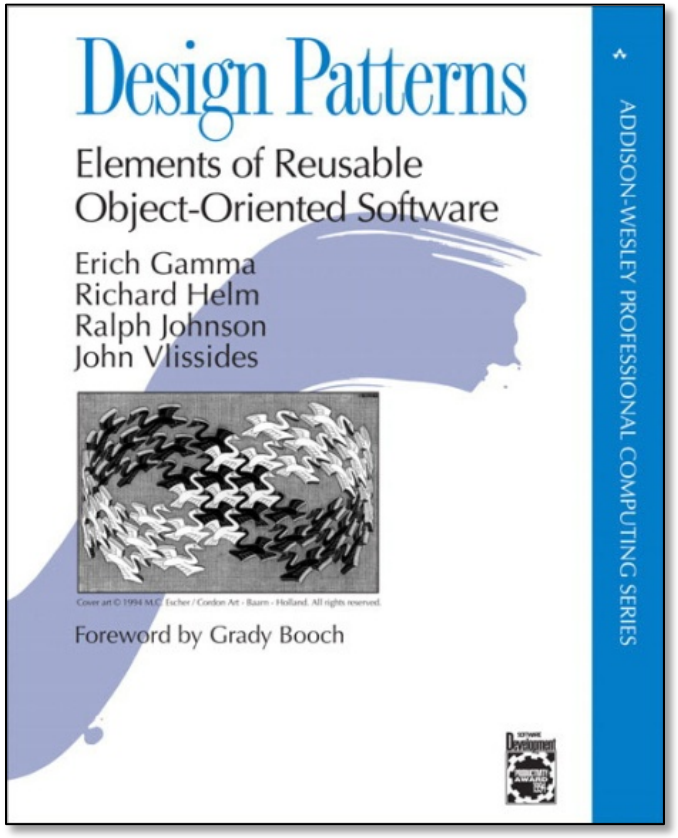
Outline of a Pattern-Oriented Framework Solution

- The earlier ping-pong program we analyzed was buggy since it lacked proper concurrency control
- This video examines the object-oriented design of the synchronized ping-pong program



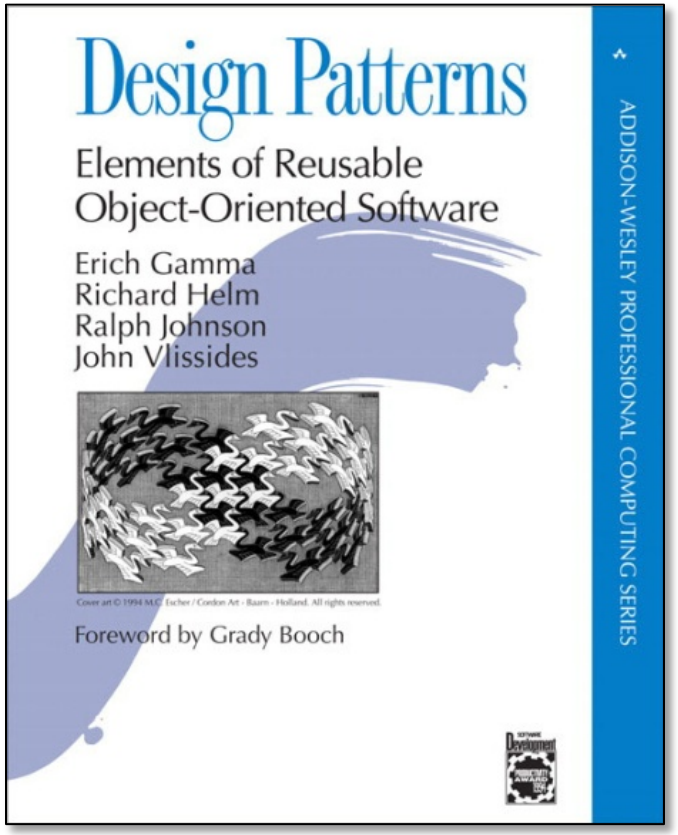
Outline of a Pattern-Oriented Framework Solution

- The earlier ping-pong program we analyzed was buggy since it lacked proper concurrency control
- This video examines the object-oriented design of the synchronized ping-pong program
- "Gang-of-Four" patterns are applied to enhance the framework-based solution



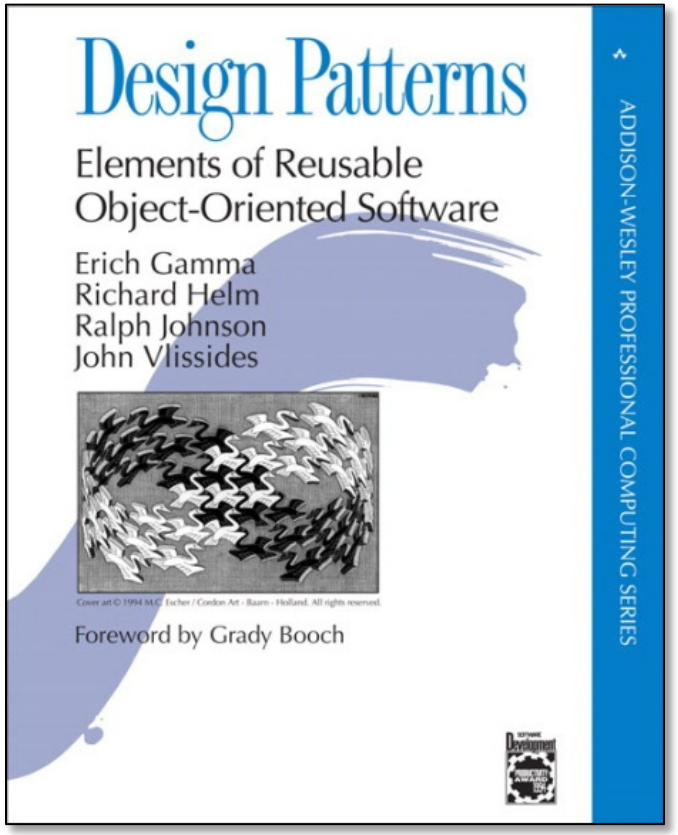
Outline of a Pattern-Oriented Framework Solution

- The earlier ping-pong program we analyzed was buggy since it lacked proper concurrency control
- This video examines the object-oriented design of the synchronized ping-pong program
- "Gang-of-Four" patterns are applied to enhance the framework-based solution
  - e.g., the *Template Method*, *Strategy*, & *Factory Method* patterns



Outline of a Pattern-Oriented Framework Solution

- The earlier ping-pong program we analyzed was buggy since it lacked proper concurrency control
- This video examines the object-oriented design of the synchronized ping-pong program
- "Gang-of-Four" patterns are applied to enhance the framework-based solution
  - e.g., the *Template Method*, *Strategy*, & *Factory Method* patterns
  - Benefits include greater reusability, flexibility, & portability



Outline of a Pattern-Oriented Framework Solution

- The earlier ping-pong program we analyzed was buggy since it lacked proper concurrency control
- This video examines the object-oriented design of the synchronized ping-pong program
- "Gang-of-Four" patterns are applied to enhance the framework-based solution
  - e.g., the *Template Method*, *Strategy*, & *Factory Method* patterns
  - Benefits include greater reusability, flexibility, & portability
    - Simplify porting to new platforms

```
% java PlayPingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Ping!(10)
Pong!(10)
Done!
```

Outline of a Pattern-Oriented Framework Solution

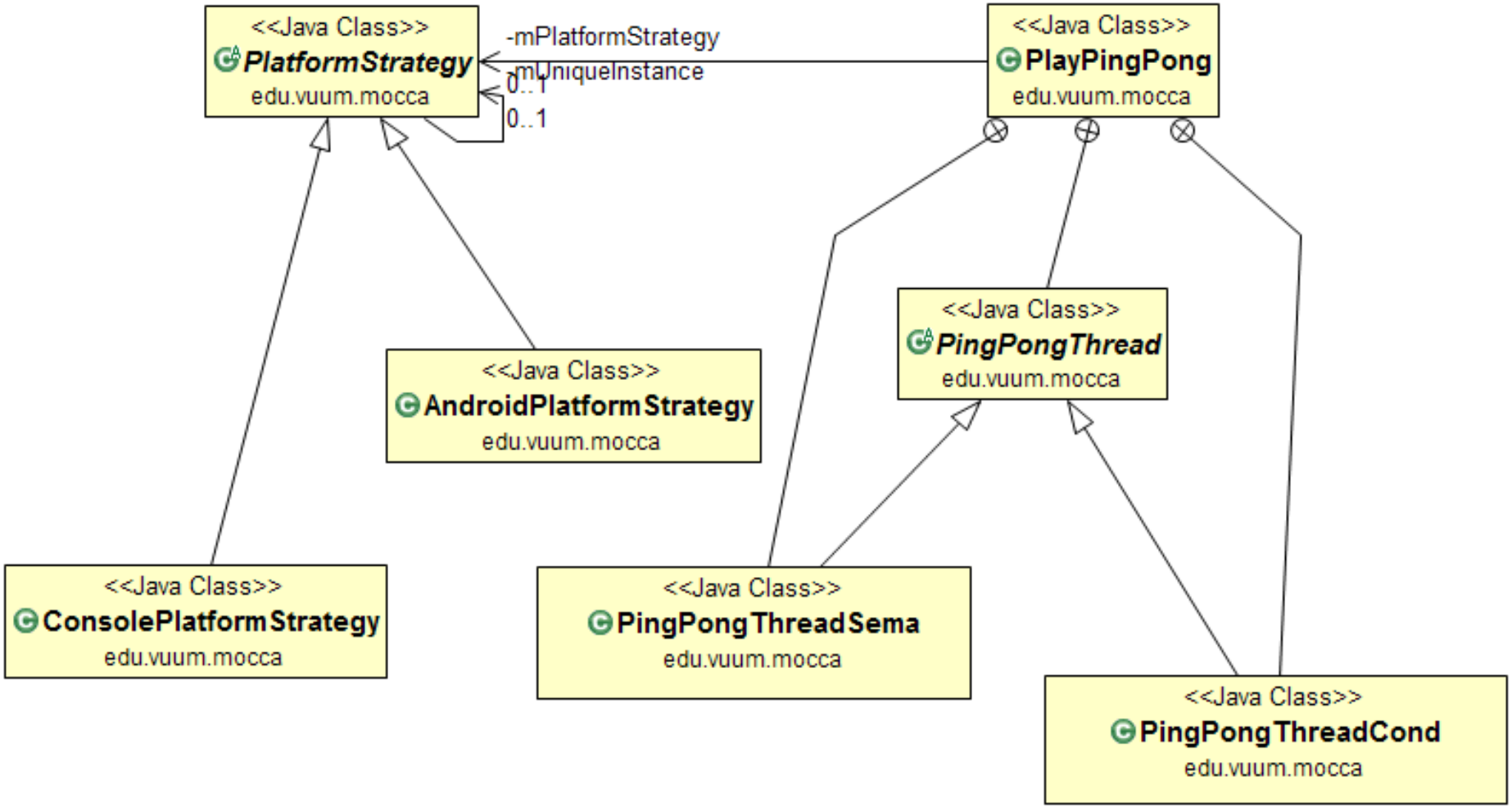
- The earlier ping-pong program we analyzed was buggy since it lacked proper concurrency control
- This video examines the object-oriented design of the synchronized ping-pong program
- "Gang-of-Four" patterns are applied to enhance the framework-based solution
  - e.g., the *Template Method*, *Strategy*, & *Factory Method* patterns
  - Benefits include greater reusability, flexibility, & portability
    - Simplify porting to new platforms

See upcoming section on "Android Concurrency Frameworks"

Object-Oriented Design of the Ping-Pong Program

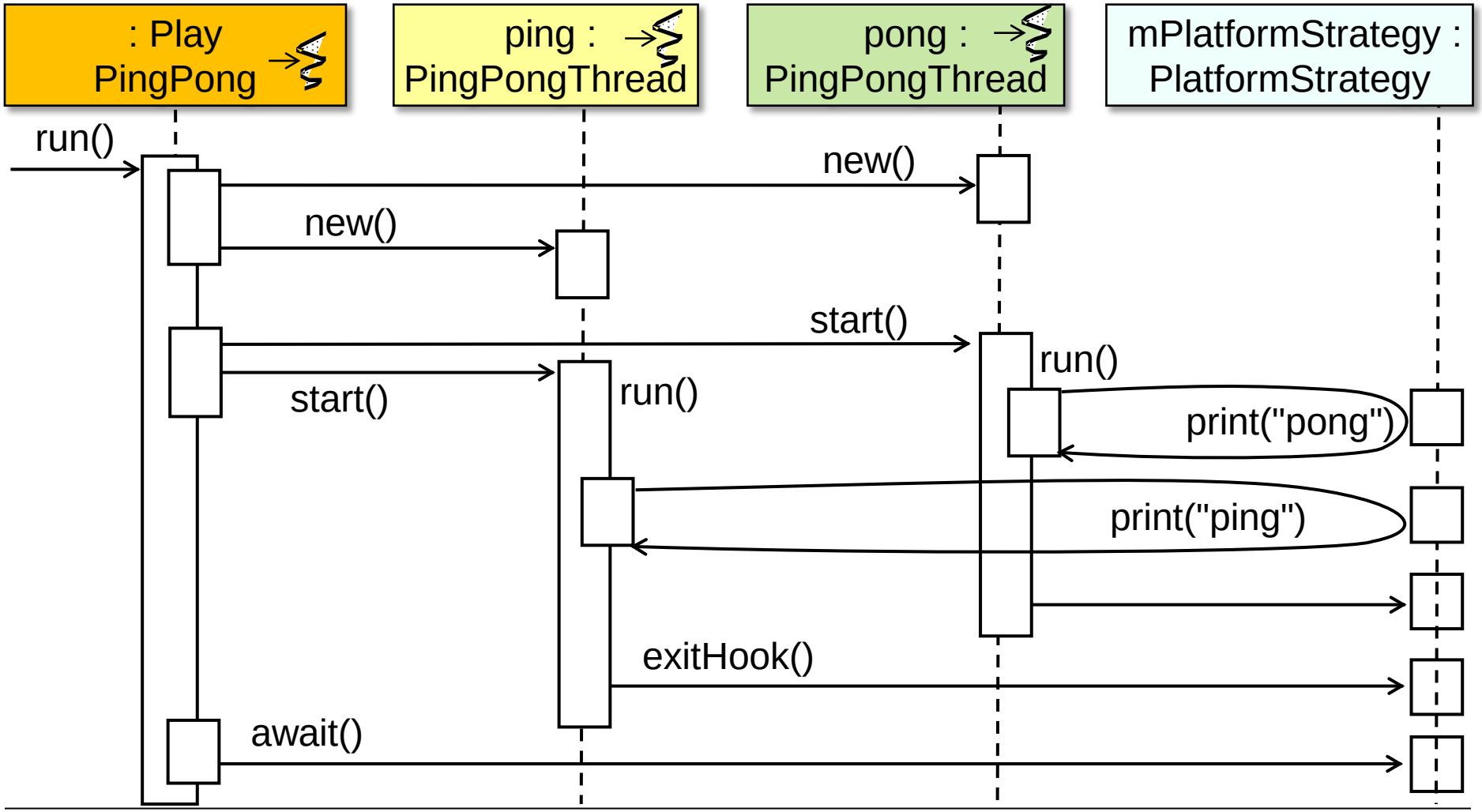
Design of the Ping-Pong Program

- UML class diagram for the ping-pong program



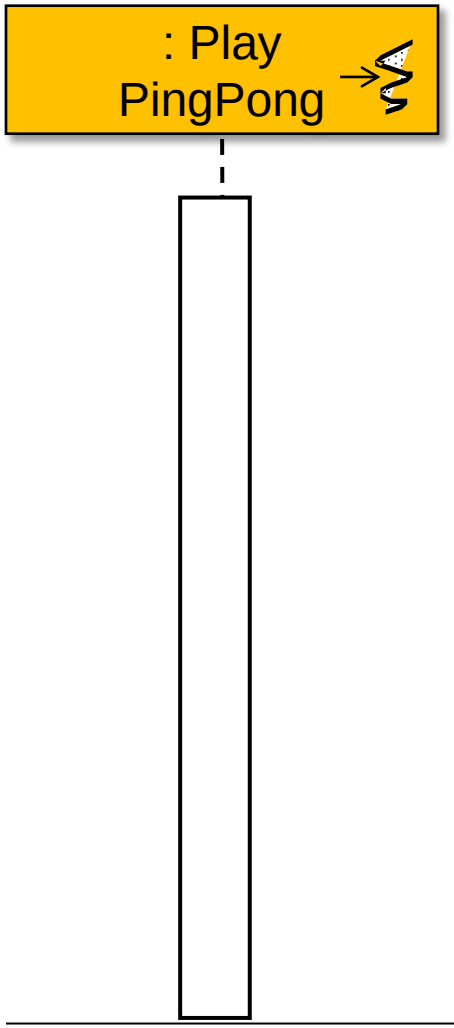
Design of the Ping-Pong Program

- UML sequence diagram for the ping-pong program



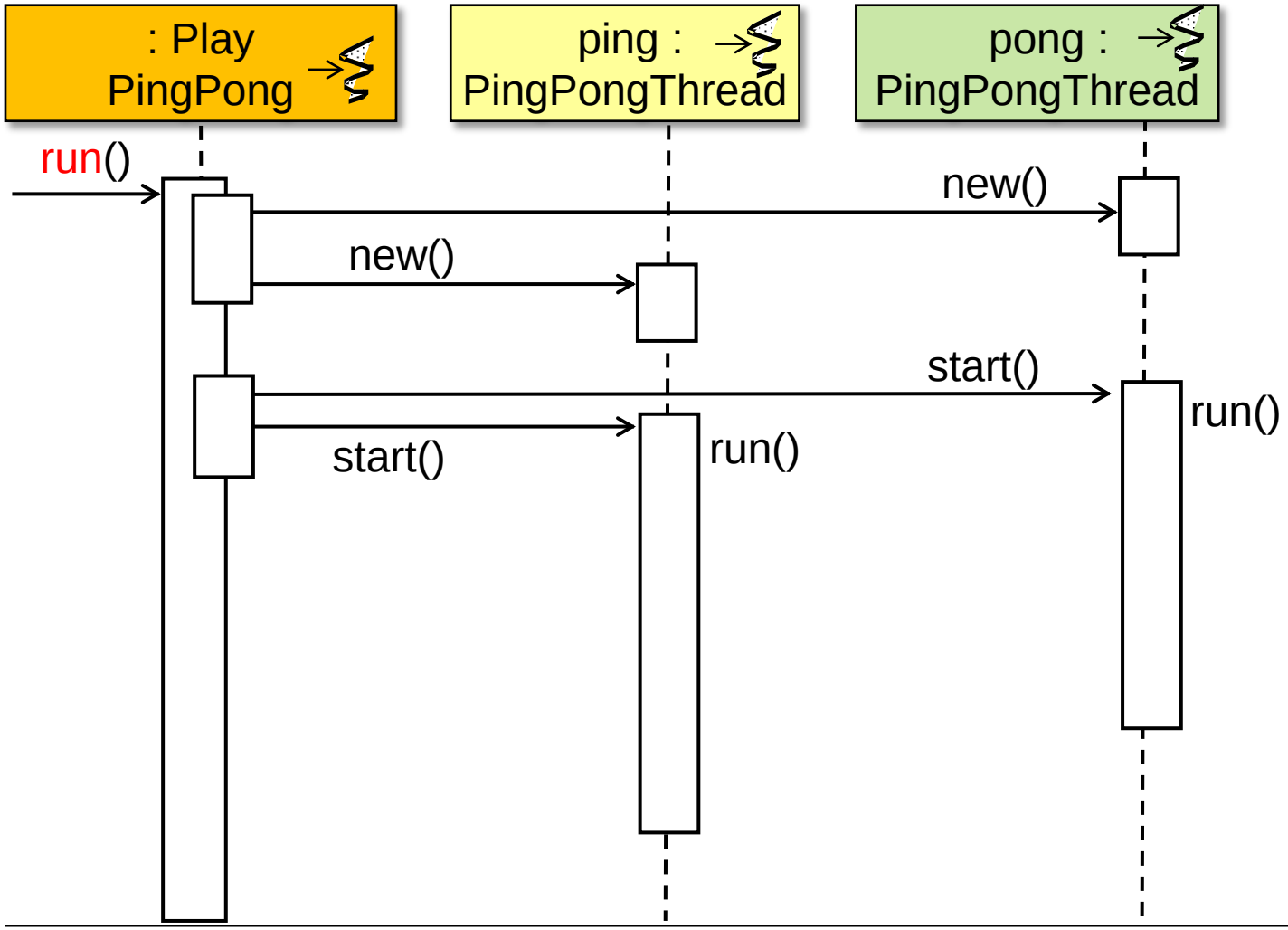
Design of the Ping-Pong Program

- UML sequence diagram for the ping-pong program



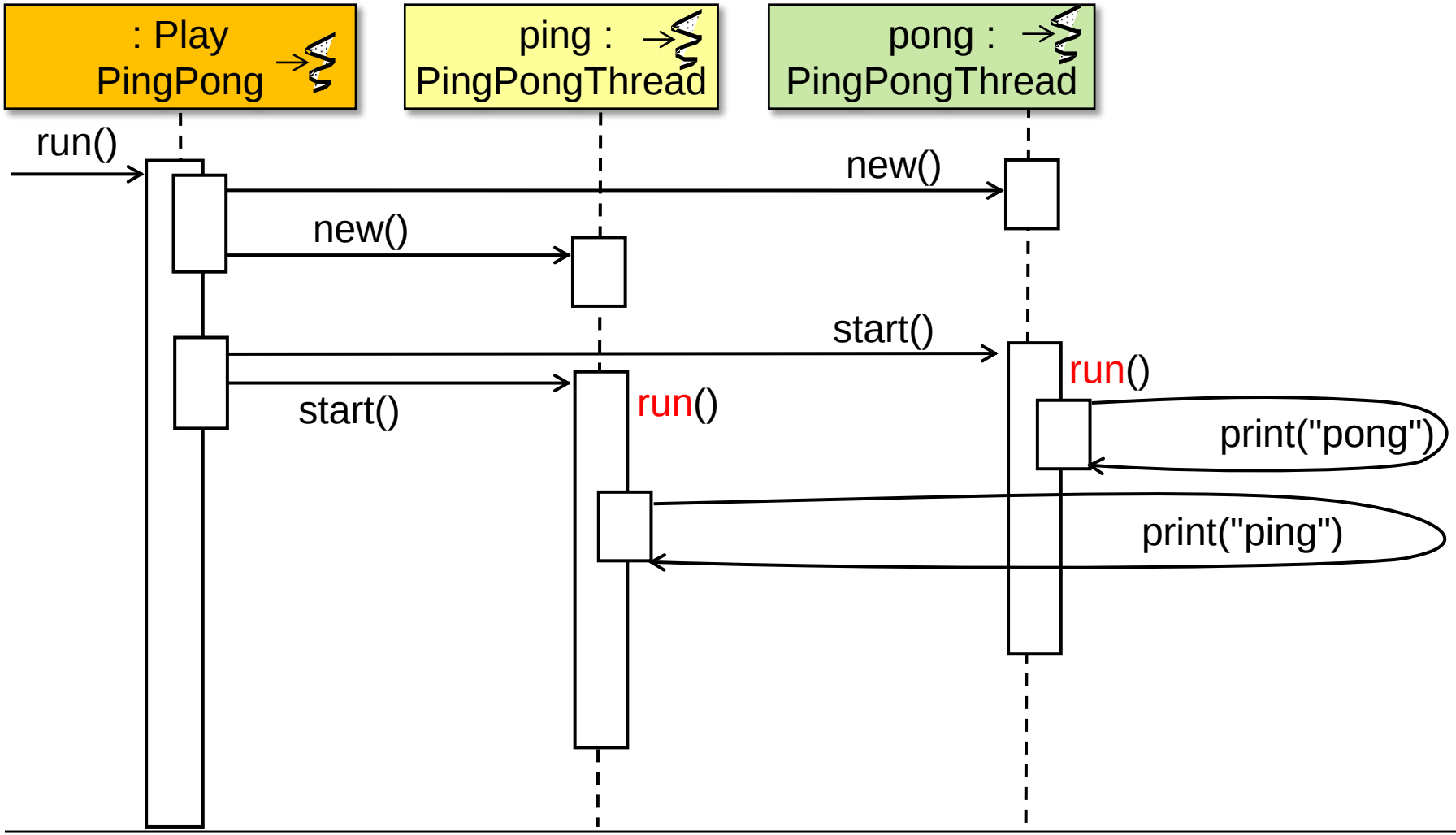
Design of the Ping-Pong Program

- UML sequence diagram for the ping-pong program



Design of the Ping-Pong Program

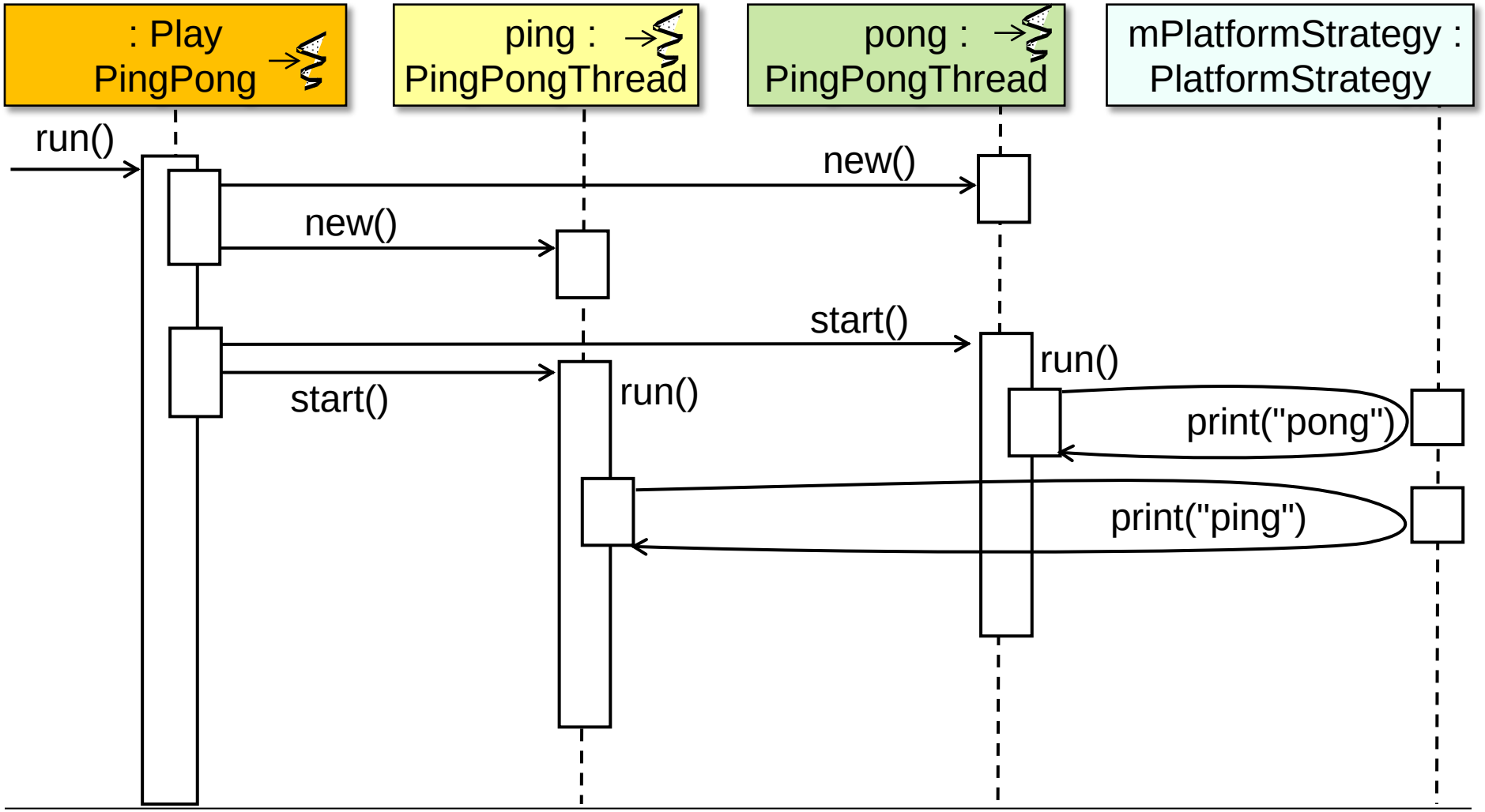
- UML sequence diagram for the ping-pong program



A pair of synchronizers controls the order in which “ping” & “pong” are called

Design of the Ping-Pong Program

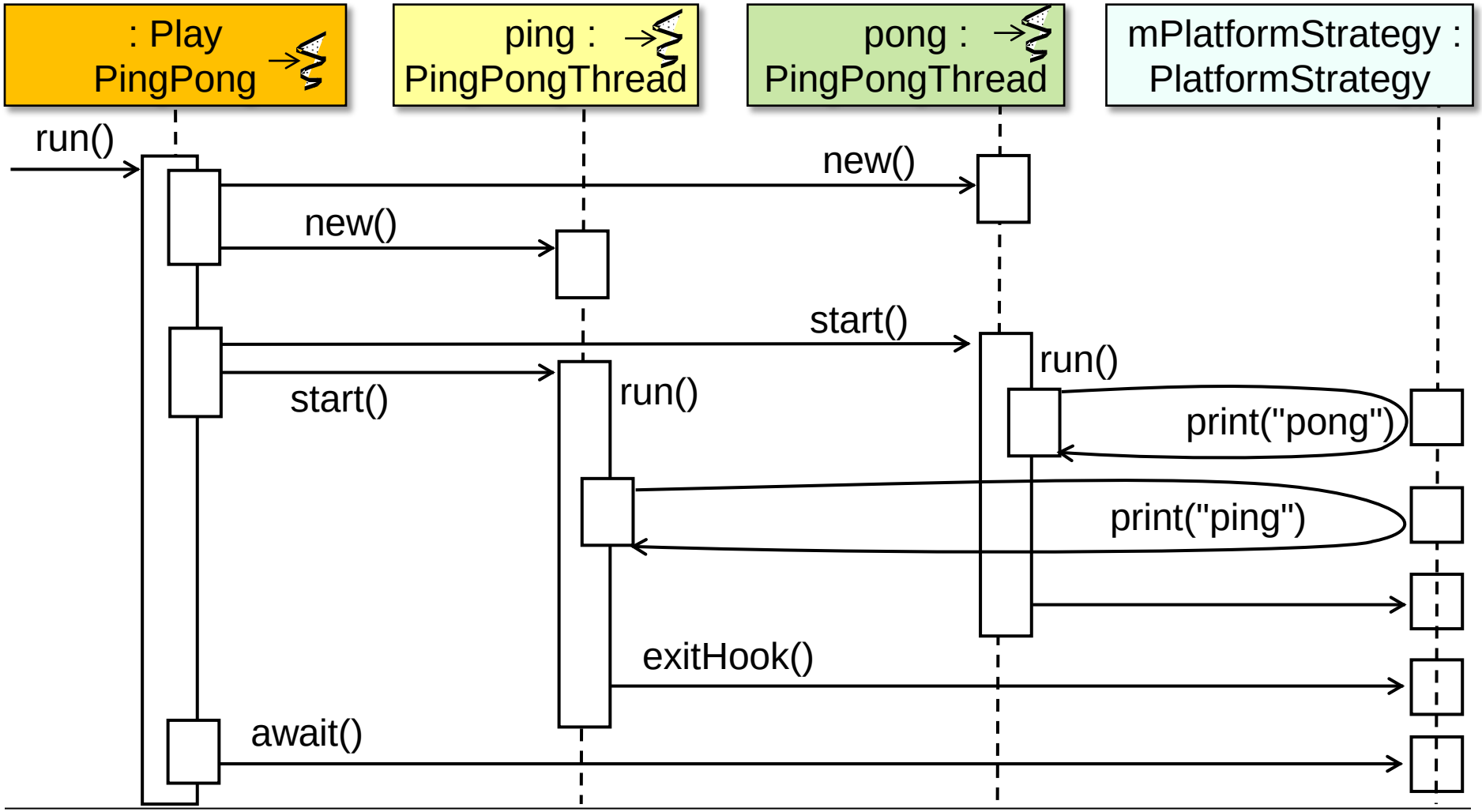
- UML sequence diagram for the ping-pong program



PlatformStrategy can be configured to support different runtime environments

Design of the Ping-Pong Program

- UML sequence diagram for the ping-pong program

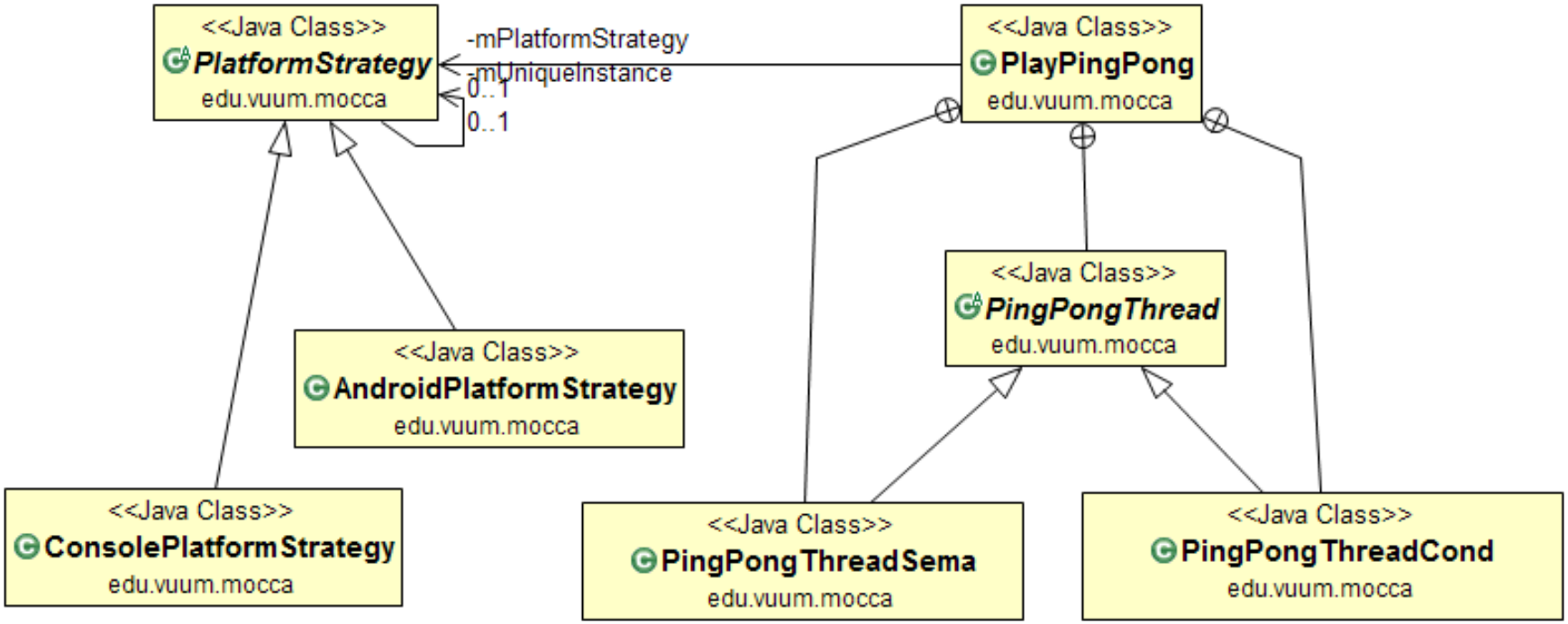


Extending this program is easy since it's pattern-oriented & framework-based

Pattern-Oriented Software Architecture of the Ping-Pong Program

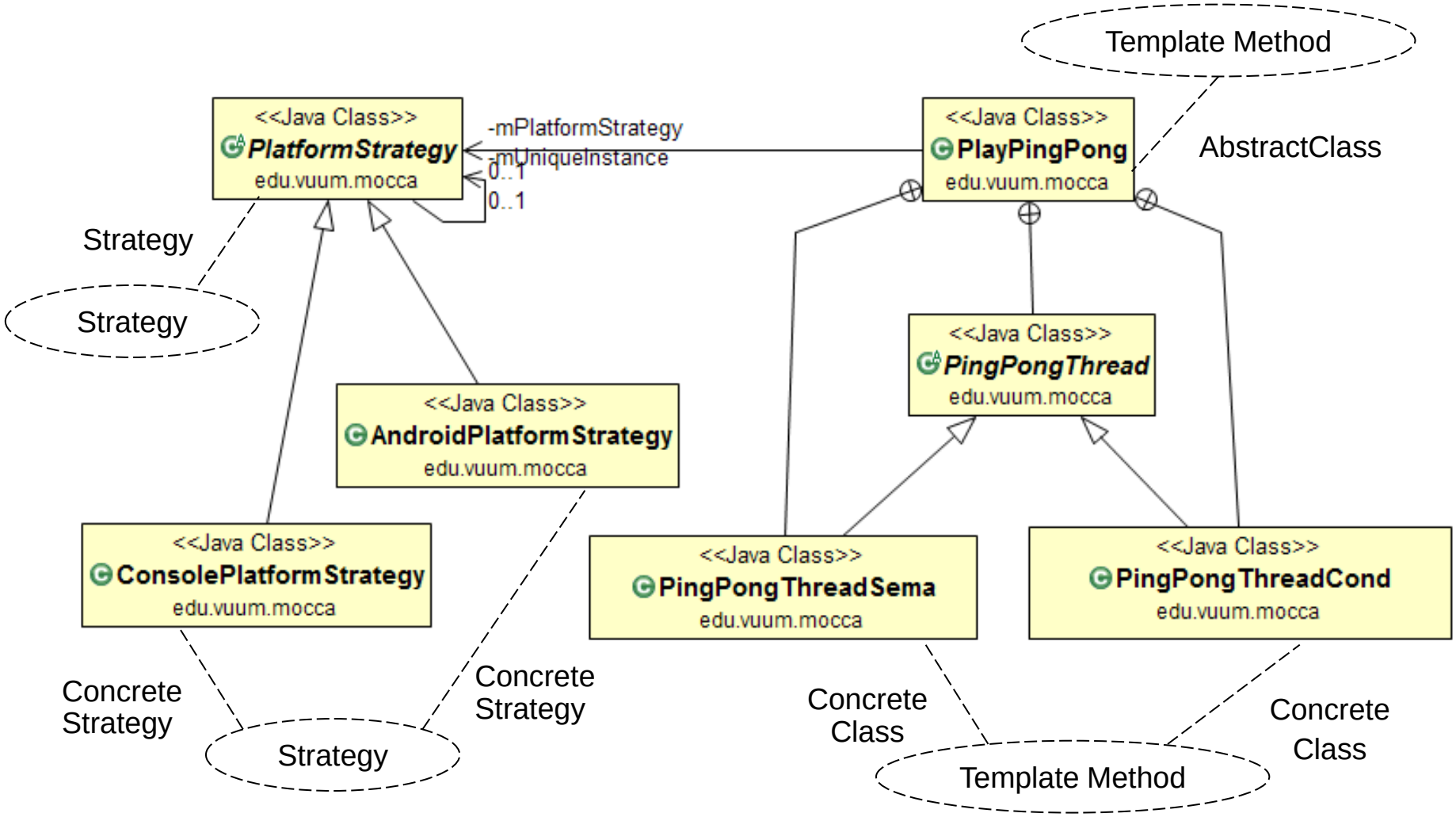
Ping-Pong Pattern-Oriented Software Architecture

- Patterns guide the design of the ping-pong program classes



Ping-Pong Pattern-Oriented Software Architecture

- Patterns guide the design of the ping-pong program classes



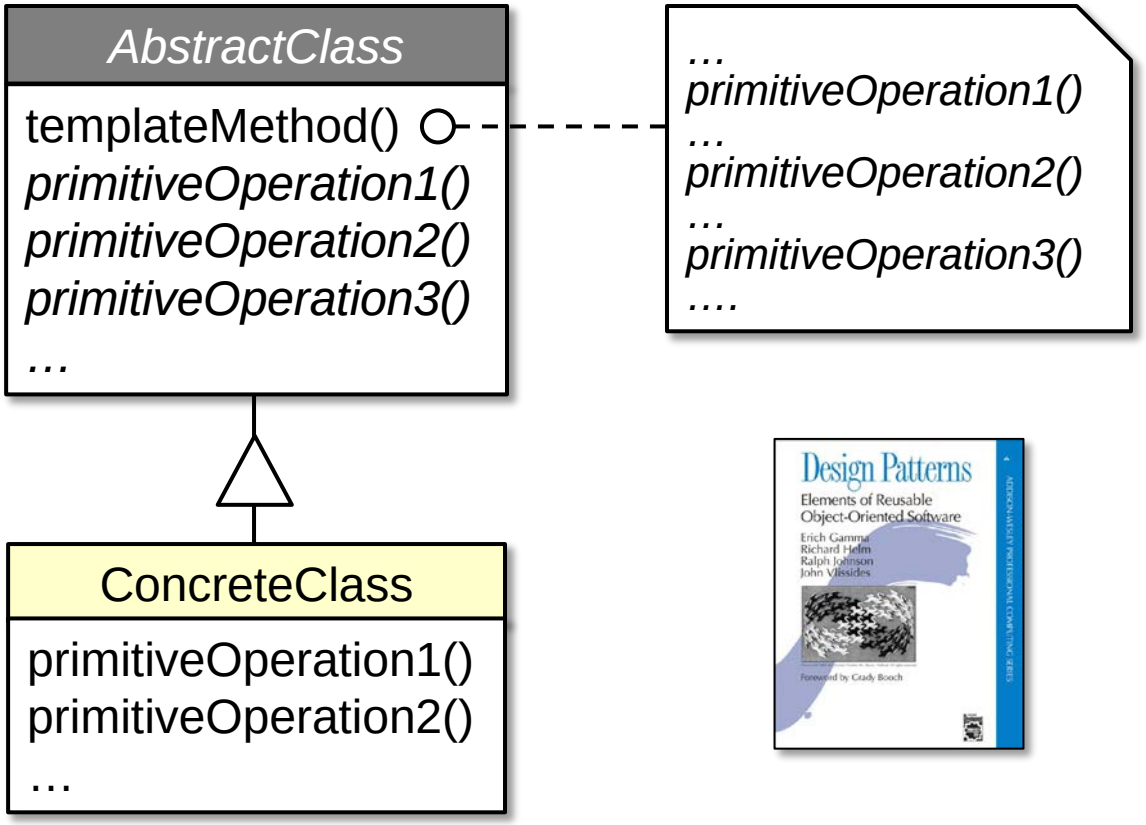
Motivation: Maximize Systematic Reuse

- Systematic reuse is a powerful method of producing quality software
  - Involves creating or acquiring reusable assets & then consistently using & evolving them

See earlier part on “Overview of Patterns & Frameworks (Part 1)”

Overview of the Template Method Pattern

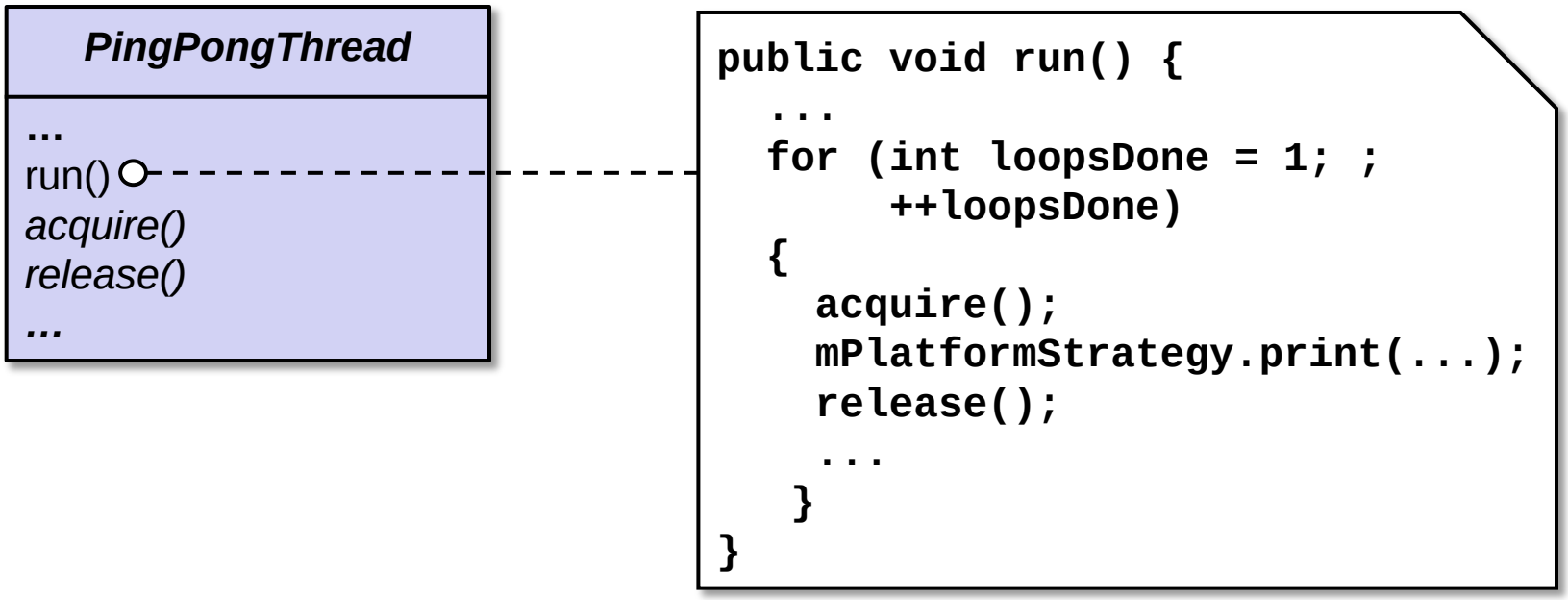
- *Template Method* provides a skeleton of an algorithm in a method, deferring some steps to subclasses



See en.wikipedia.org/wiki/Template_method_pattern for more info

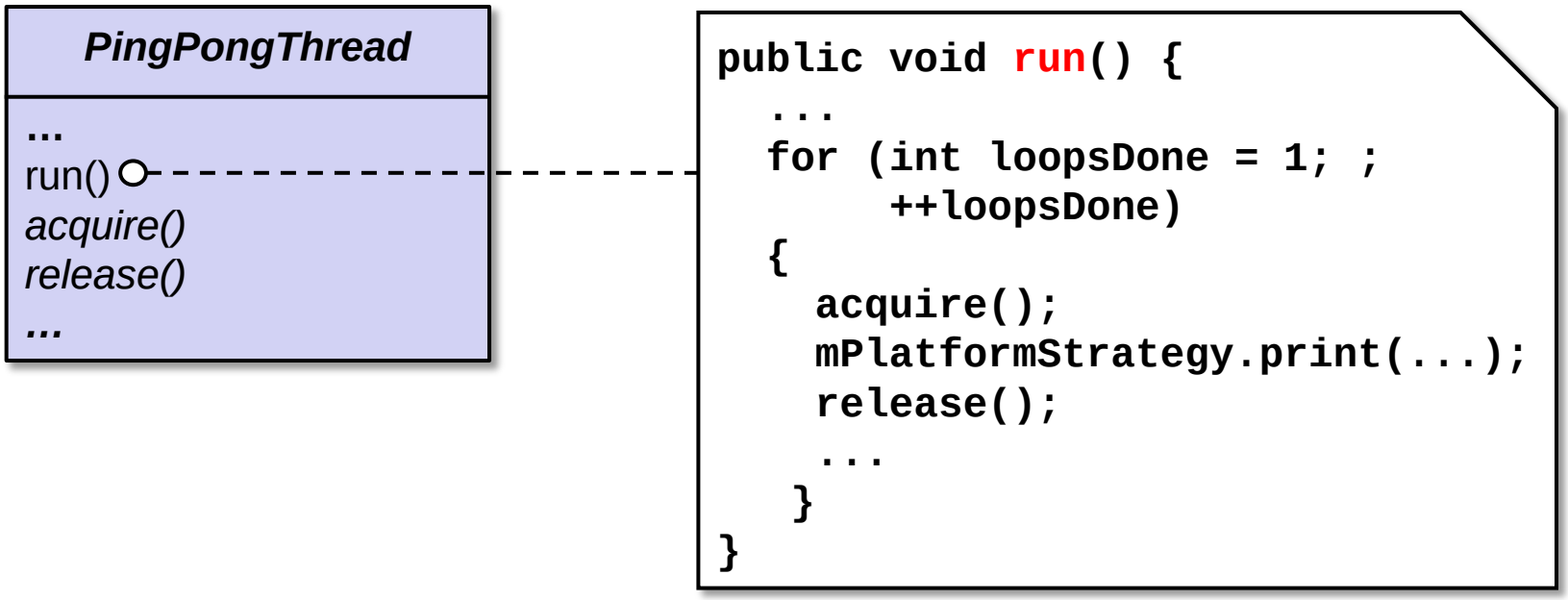
Applying the Template Method Pattern

- *Template Method* increases systematic reuse by implementing the ping/ping algorithm in the base class, but allowing subclasses to override its acquire() & release() hook methods to schedule printing via different mechanisms



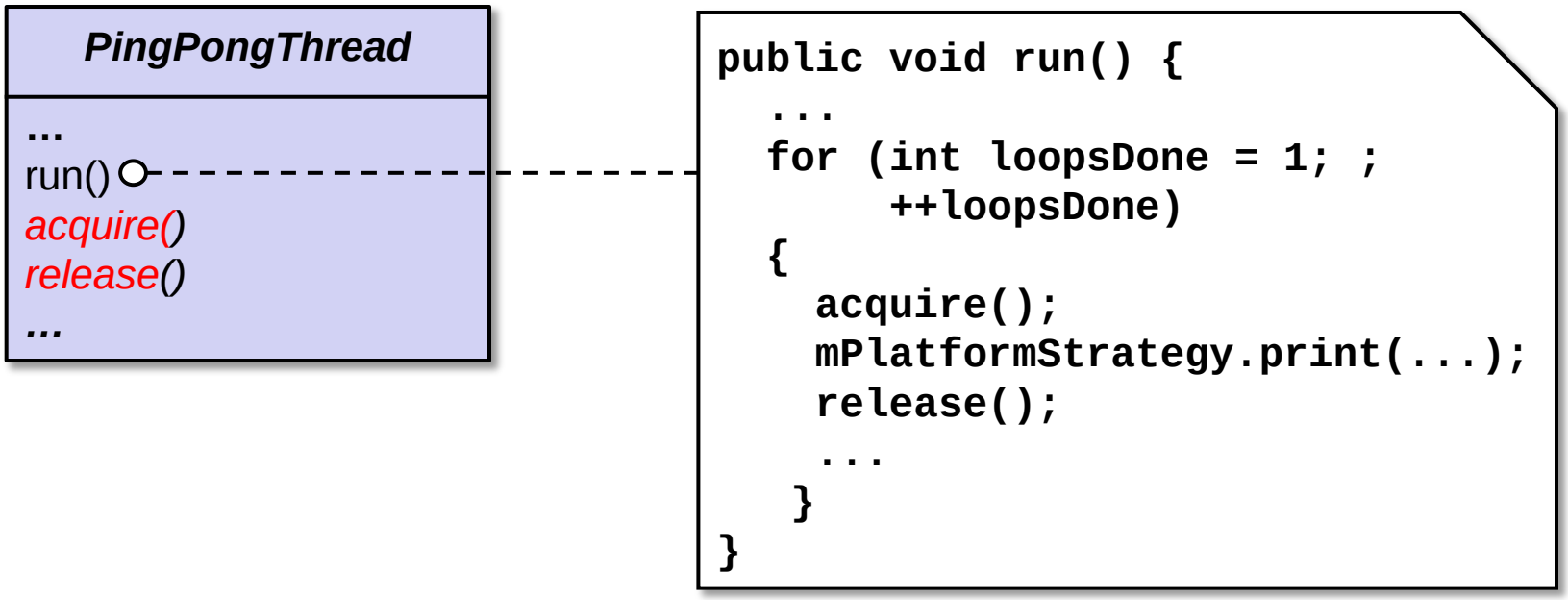
Applying the Template Method Pattern

- *Template Method* increases systematic reuse by implementing the ping/ping algorithm in the base class, but allowing subclasses to override its acquire() & release() hook methods to schedule printing via different mechanisms



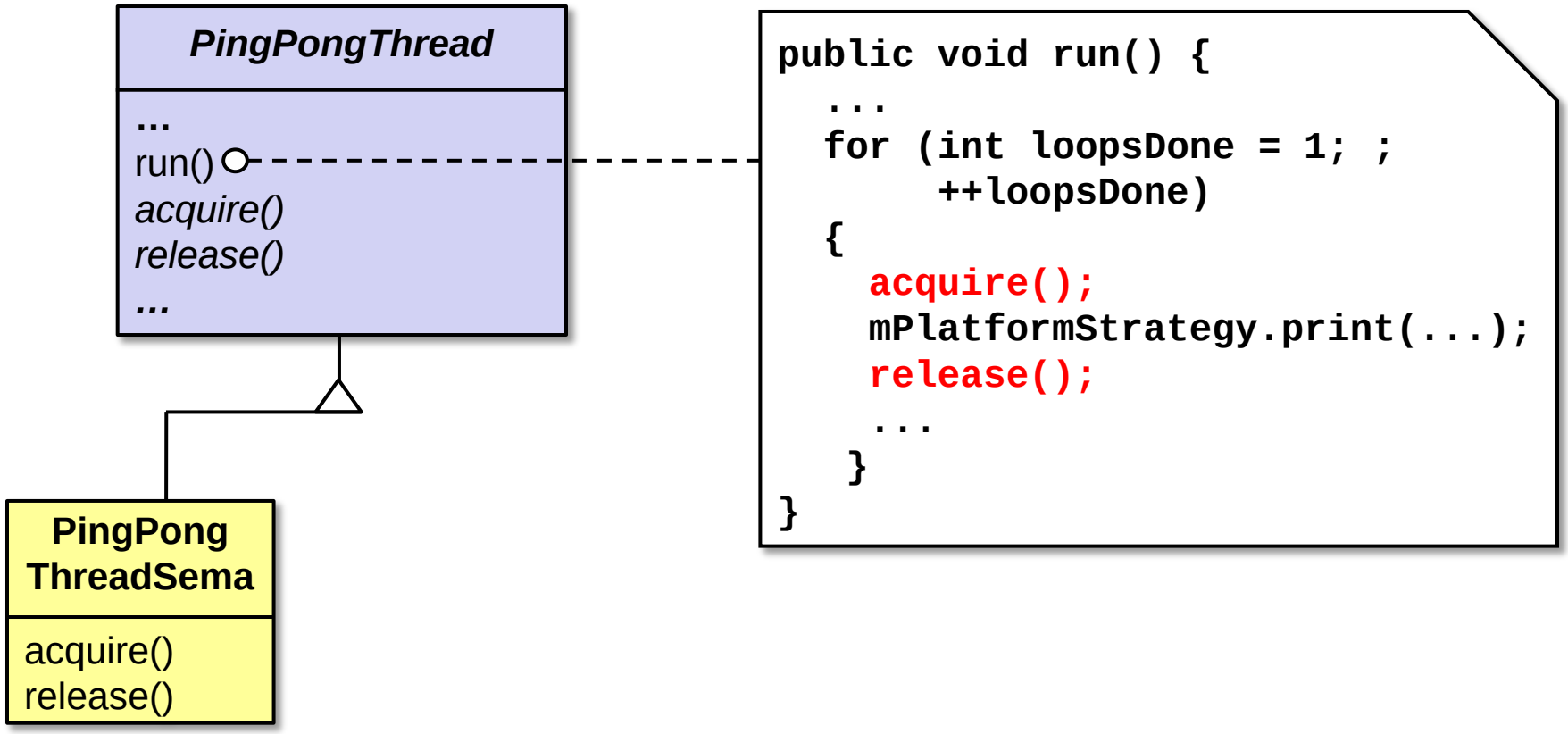
Applying the Template Method Pattern

- *Template Method* increases systematic reuse by implementing the ping/ping algorithm in the base class, but allowing subclasses to override its acquire() & release() hook methods to schedule printing via different mechanisms



Applying the Template Method Pattern

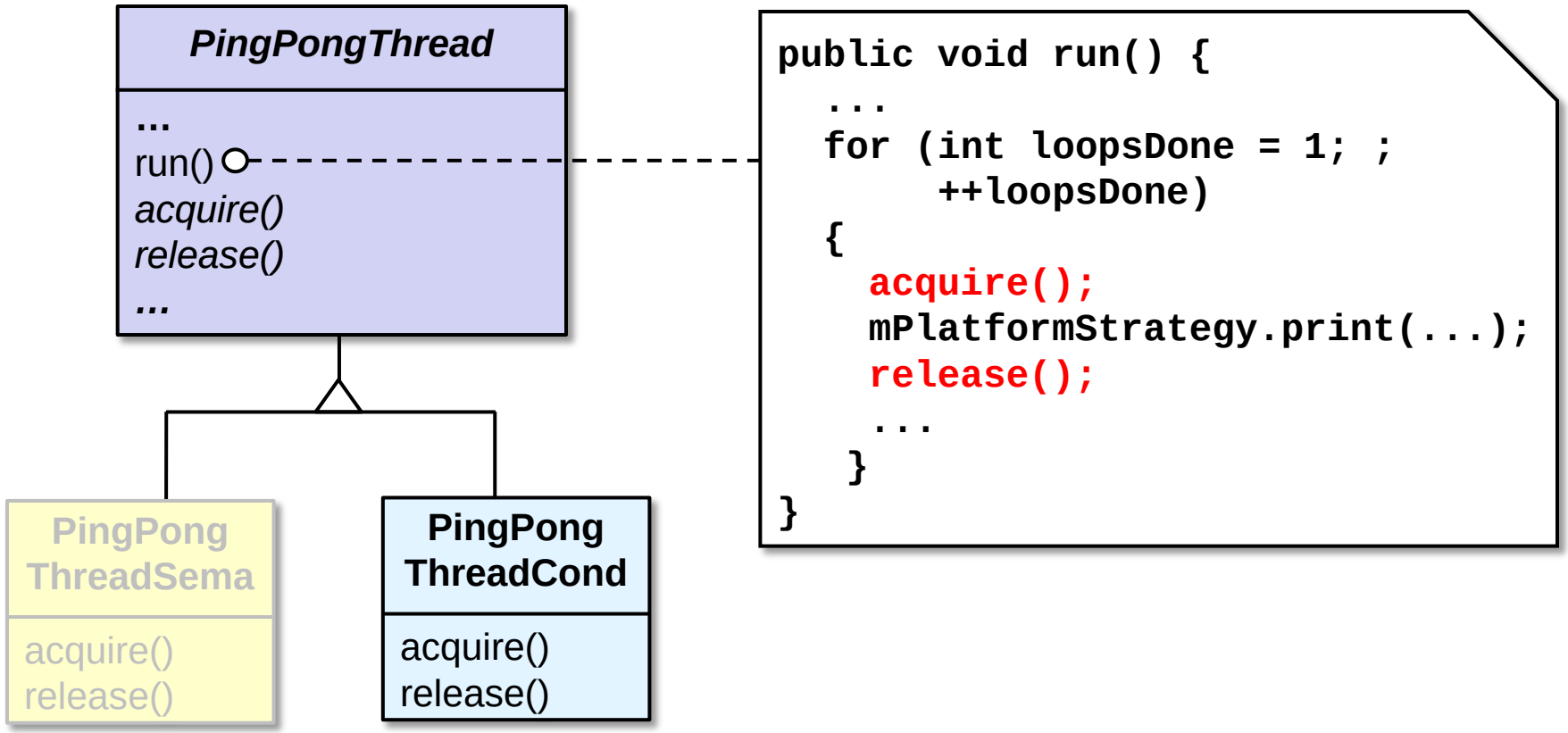
- *Template Method* increases systematic reuse by implementing the ping/ping algorithm in the base class, but allowing subclasses to override its acquire() & release() hook methods to schedule printing via different mechanisms



See next part on “Java Synchronization & Scheduling Example (Part 2)”

Applying the Template Method Pattern

- *Template Method* increases systematic reuse by implementing the ping/ping algorithm in the base class, but allowing subclasses to override its acquire() & release() hook methods to schedule printing via different mechanisms

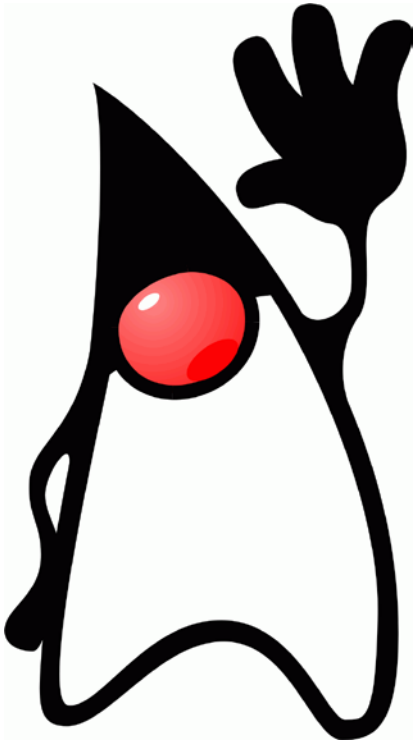


See next part on “Java Synchronization & Scheduling Example (Part 2)”

Applying the Strategy Pattern to the Ping-Pong Program

Motivation: Maximize Portability

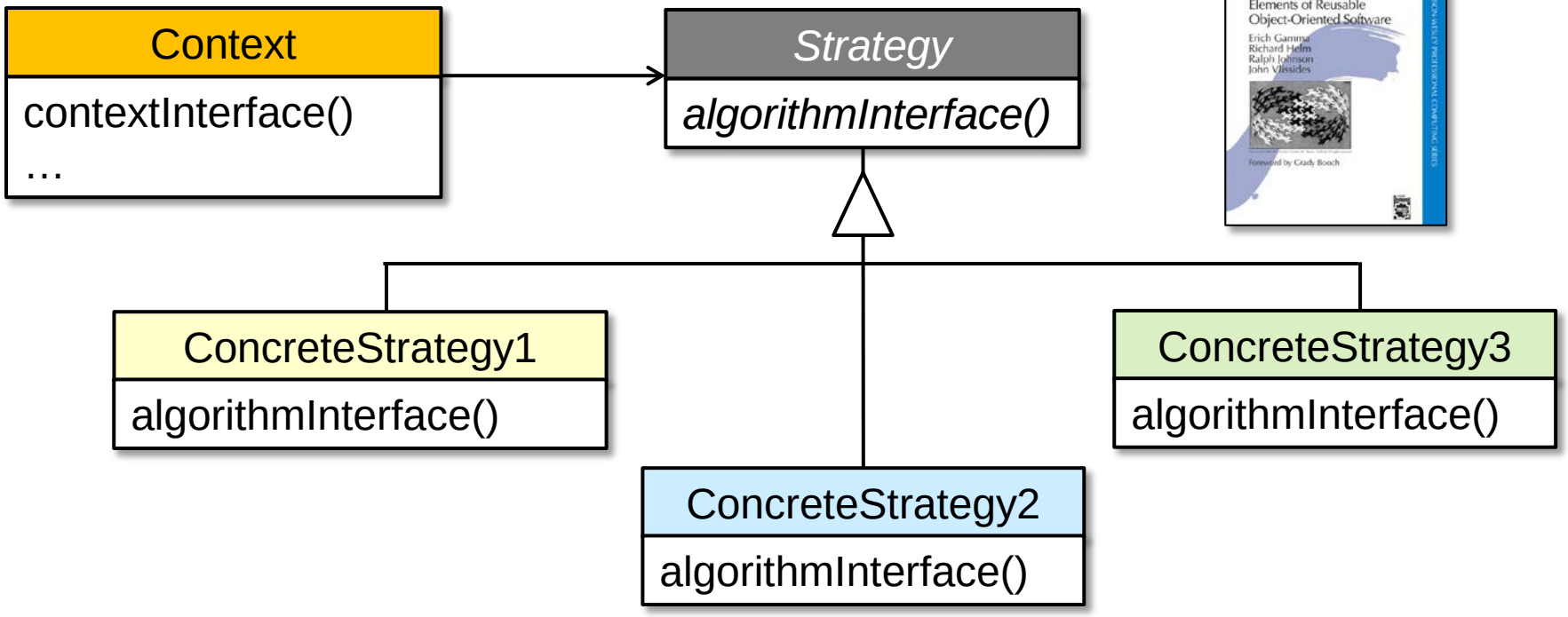
- Portability enables the same implementation to be applied in different operating environments
  - It increases the potential user base & market size



en.wikipedia.org/wiki/Software_portability has more on portability

Overview of the Strategy Pattern

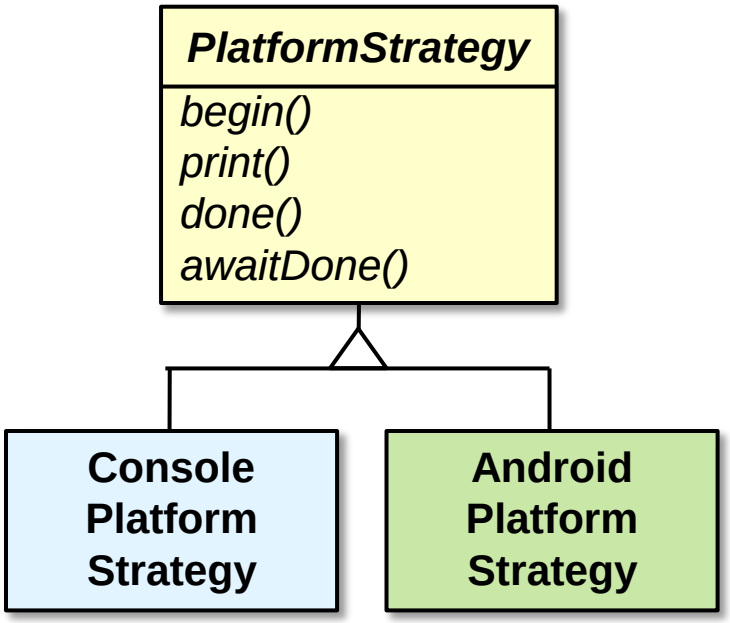
- *Strategy* defines a family of algorithms, encapsulate each one, & make them interchangeable to let clients & algorithms vary independently



See en.wikipedia.org/wiki/Strategy_pattern for more info

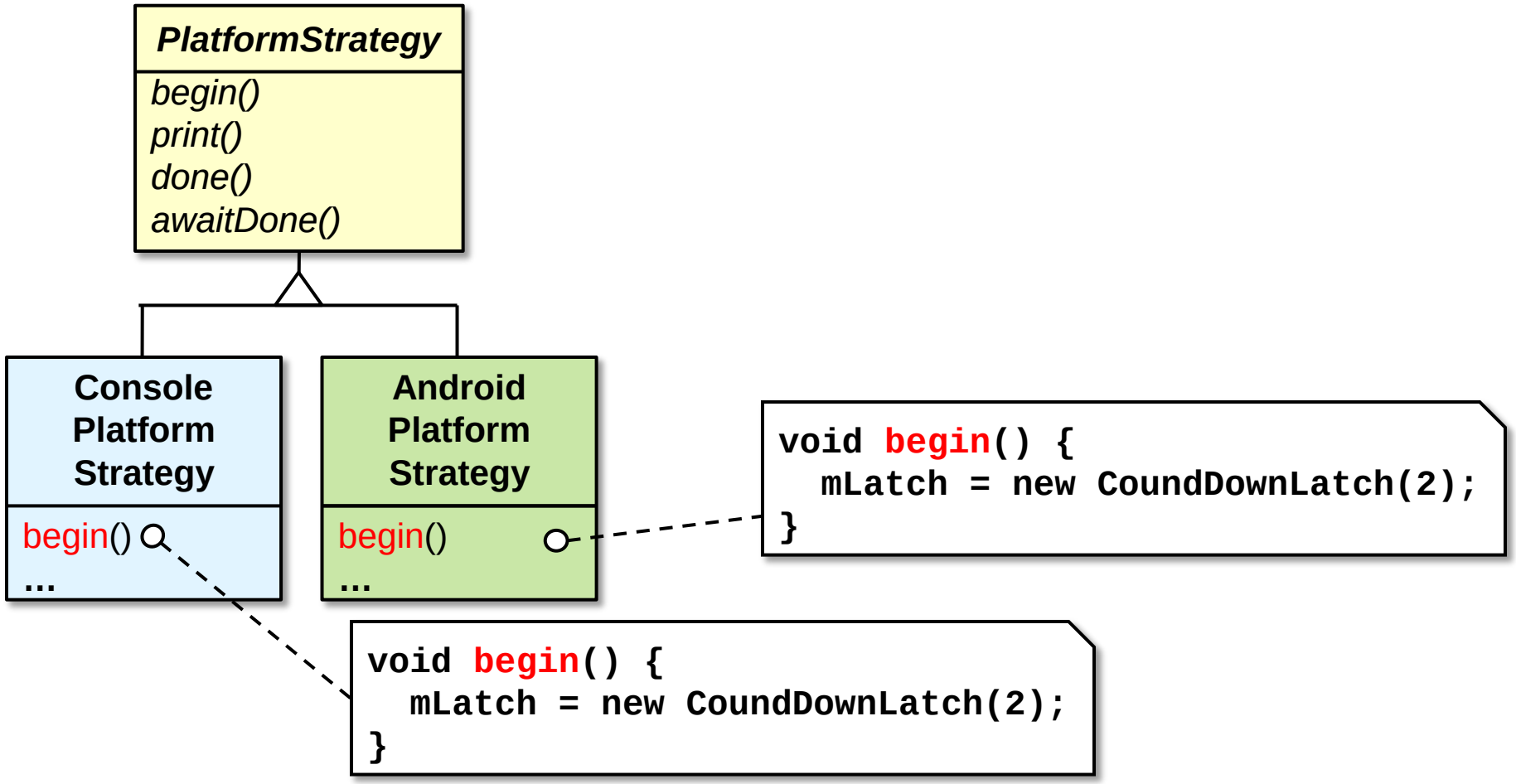
Applying the Strategy Pattern

- *Strategy* enhances portability by factoring out different ways of printing "ping" & "pong" on different types runtime platforms



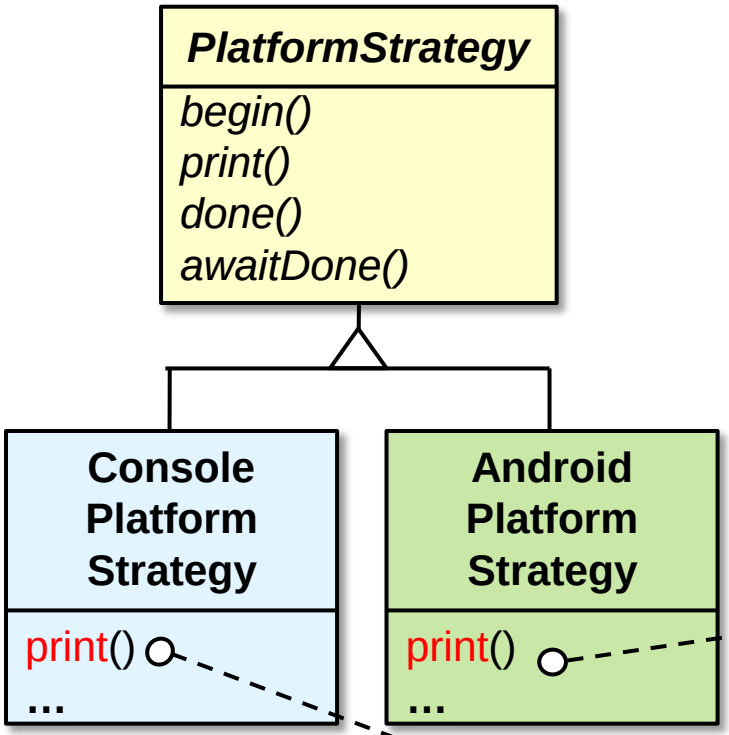
Applying the Strategy Pattern

- *Strategy* enhances portability by factoring out different ways of printing "ping" & "pong" on different types runtime platforms



Applying the Strategy Pattern

- *Strategy* enhances portability by factoring out different ways of printing "ping" & "pong" on different types runtime platforms

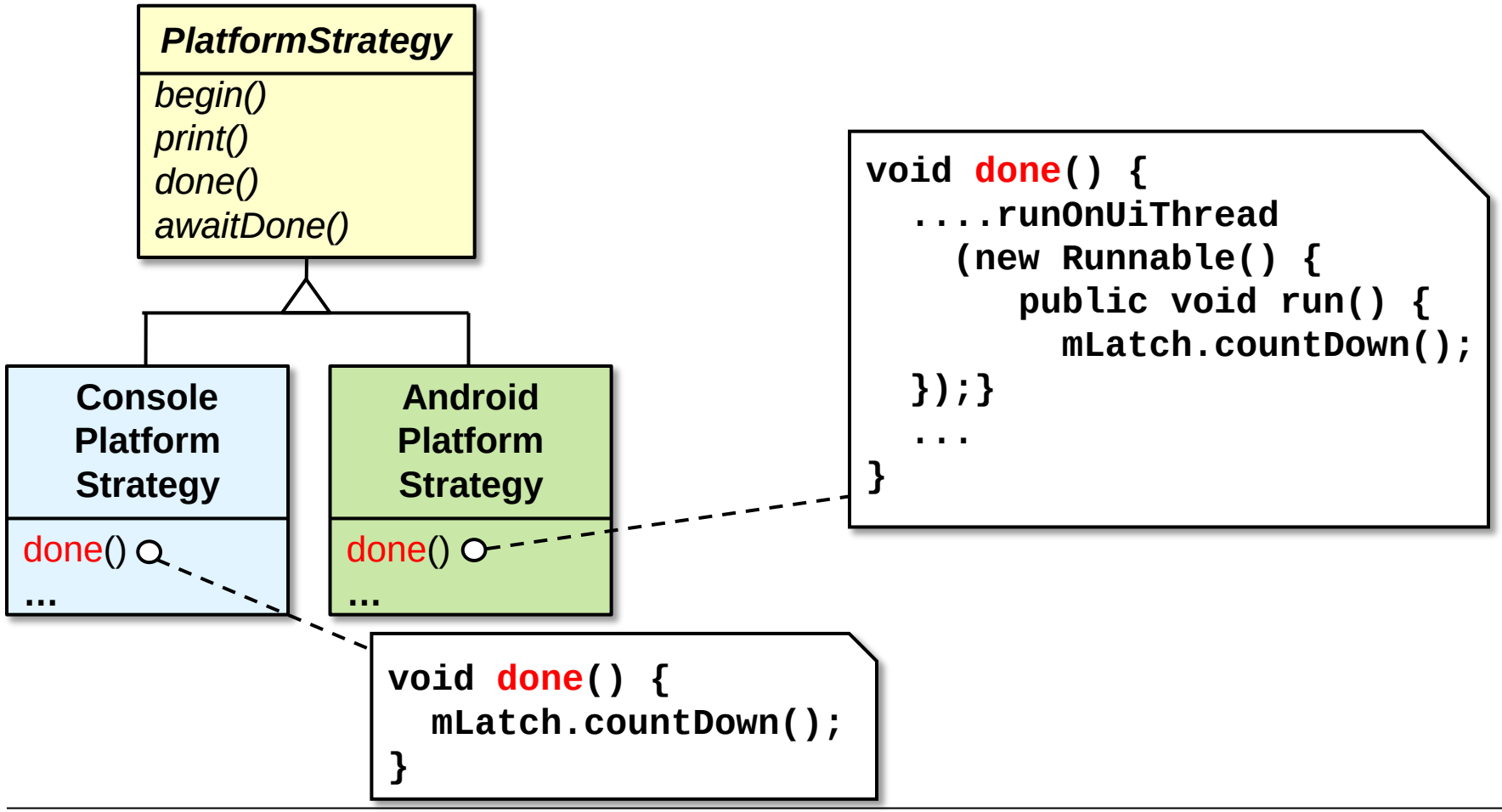



```
void print(outputString) {
  ....runOnUiThread
    (new Runnable() {
       public void run() {
         mTextView.append(outputString);
  }});
  ...
}
```

```
void print(String outputString) {
  System.out.println(outputString);
}
```

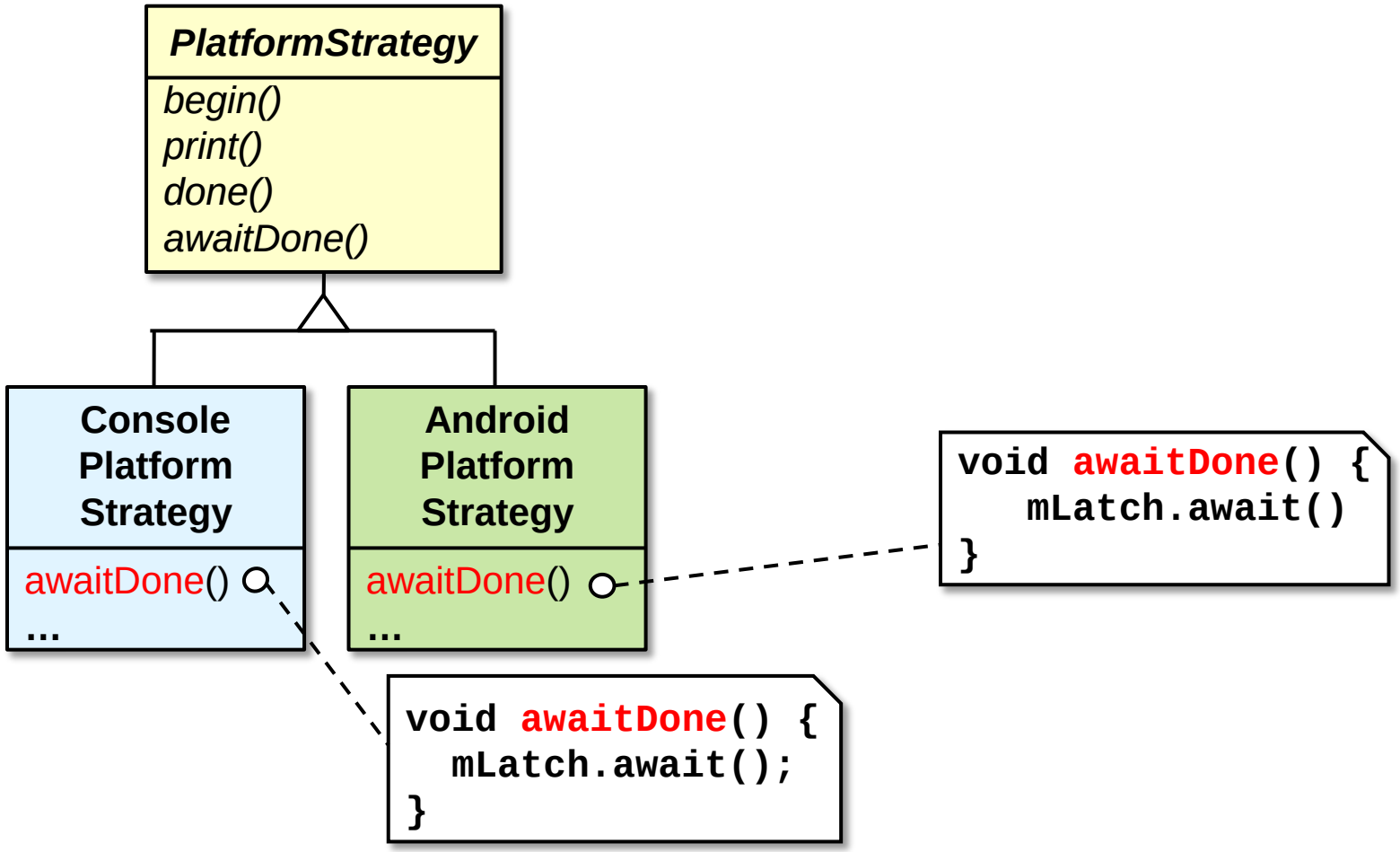
Applying the Strategy Pattern

- *Strategy* enhances portability by factoring out different ways of printing "ping" & "pong" on different types runtime platforms



Applying the Strategy Pattern

- *Strategy* enhances portability by factoring out different ways of printing "ping" & "pong" on different types runtime platforms



See earlier part on "Java CountDownLatch"

Applying the Factory Method Pattern to the Ping-Pong Program

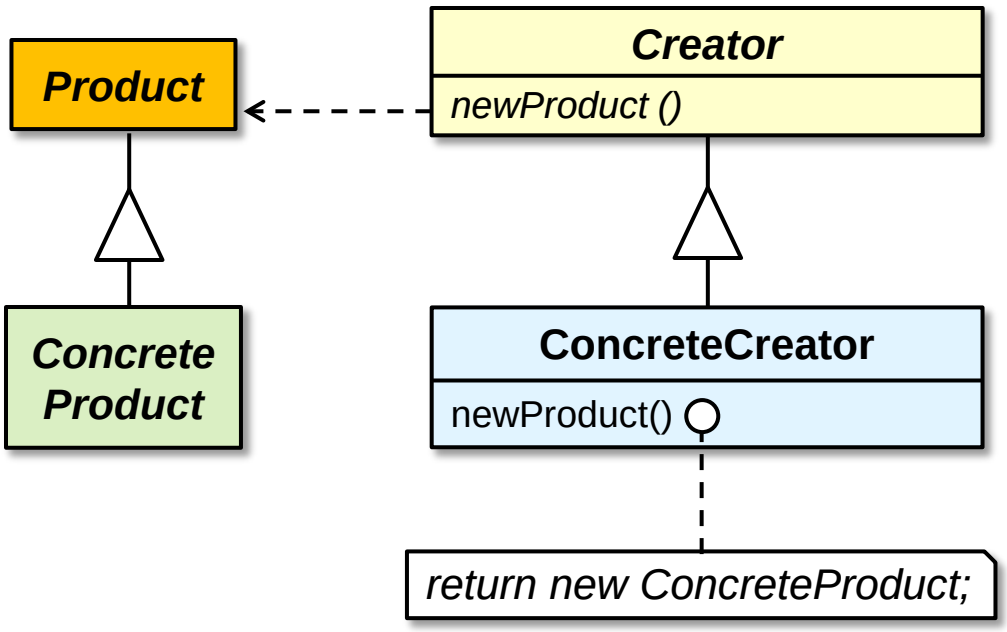
Motivation: Maximize Flexibility

- Flexibility enables designs to adapt more readily in response to external or internal changes in requirements or environments

See informativearchitecture.wordpress.com/2011/10/04/software-flexibility

Overview of the Factory Method Pattern

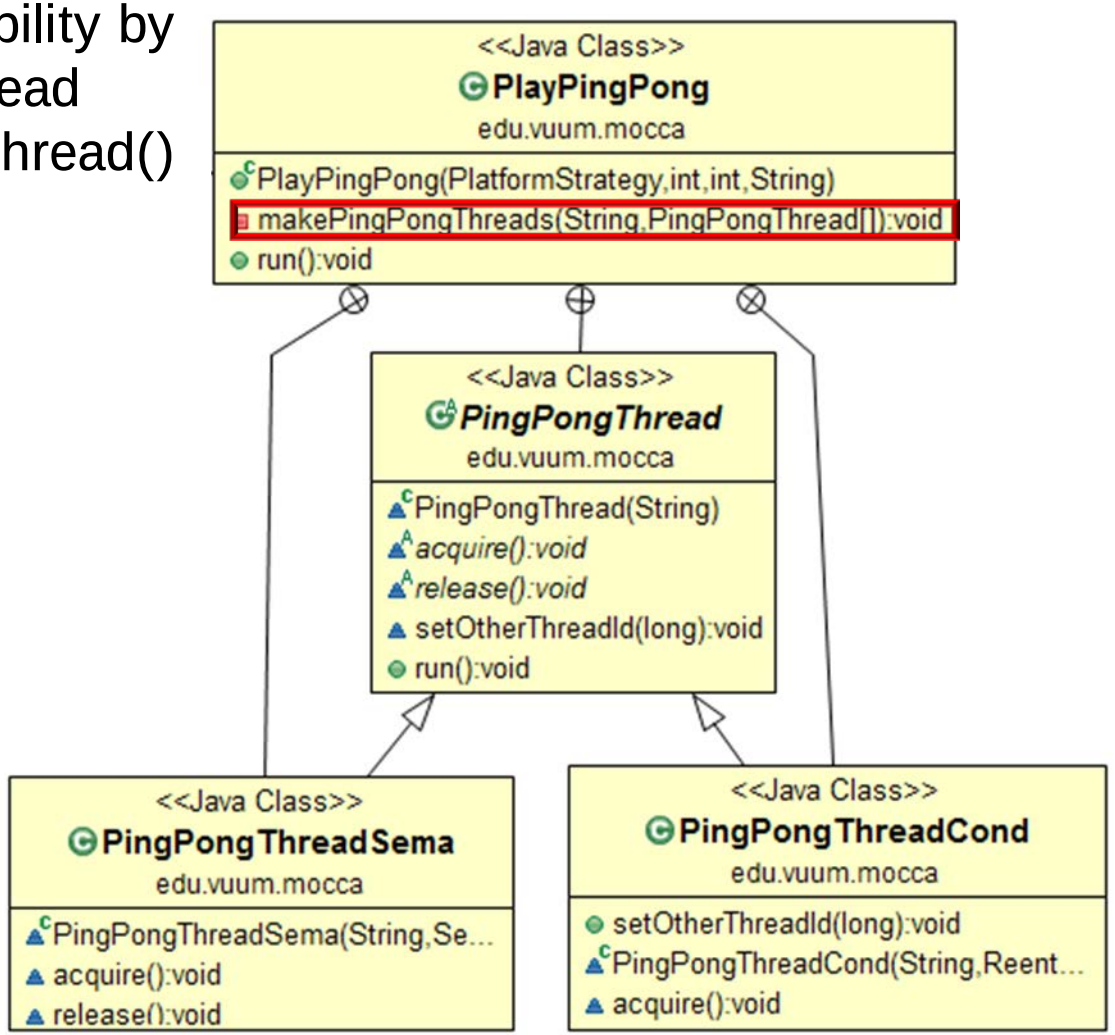
- *Factory Method* provides an interface for creating an object, but leaves the choice of the object's concrete type to a subclass



See en.wikipedia.org/wiki/Factory_method_pattern for more info

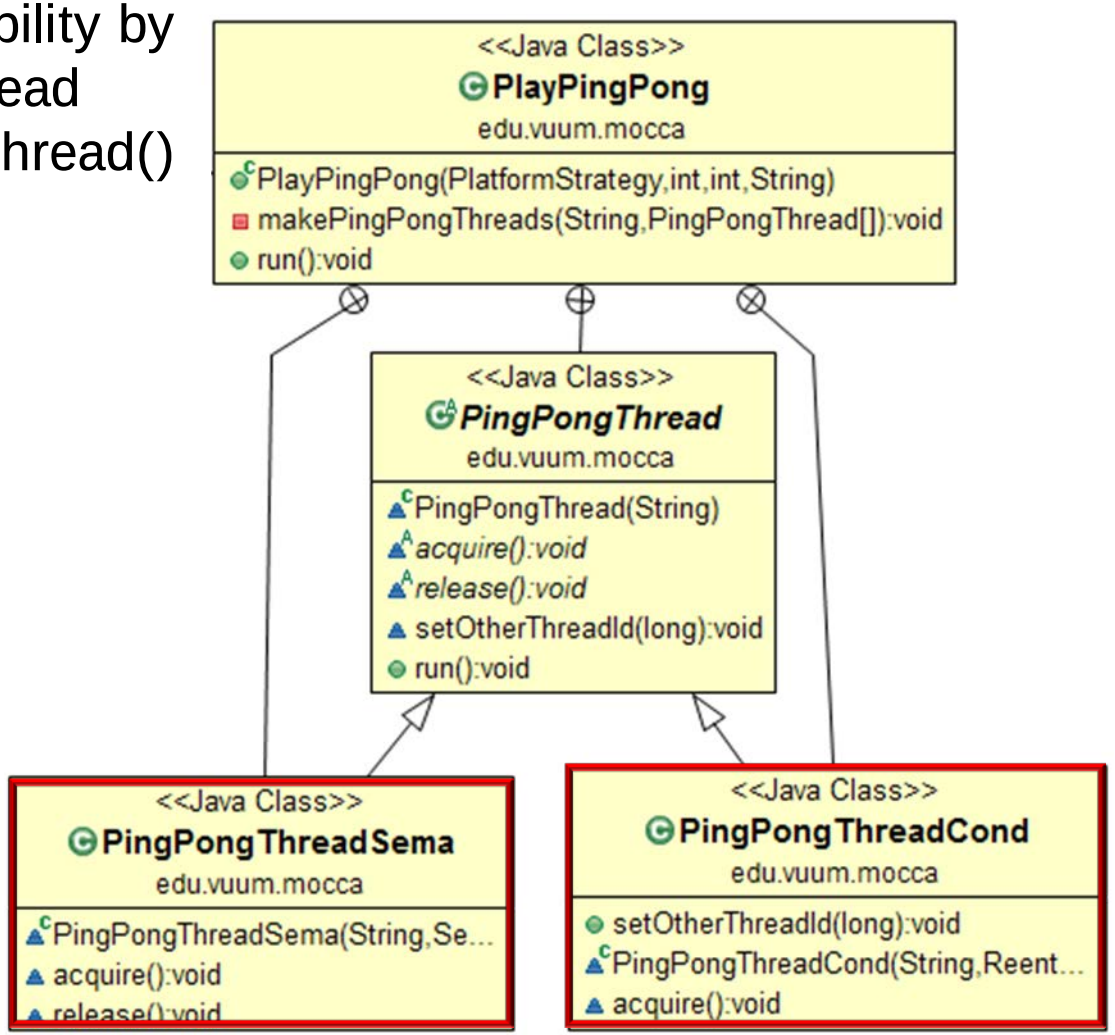
Applying the Factory Method Pattern

- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()



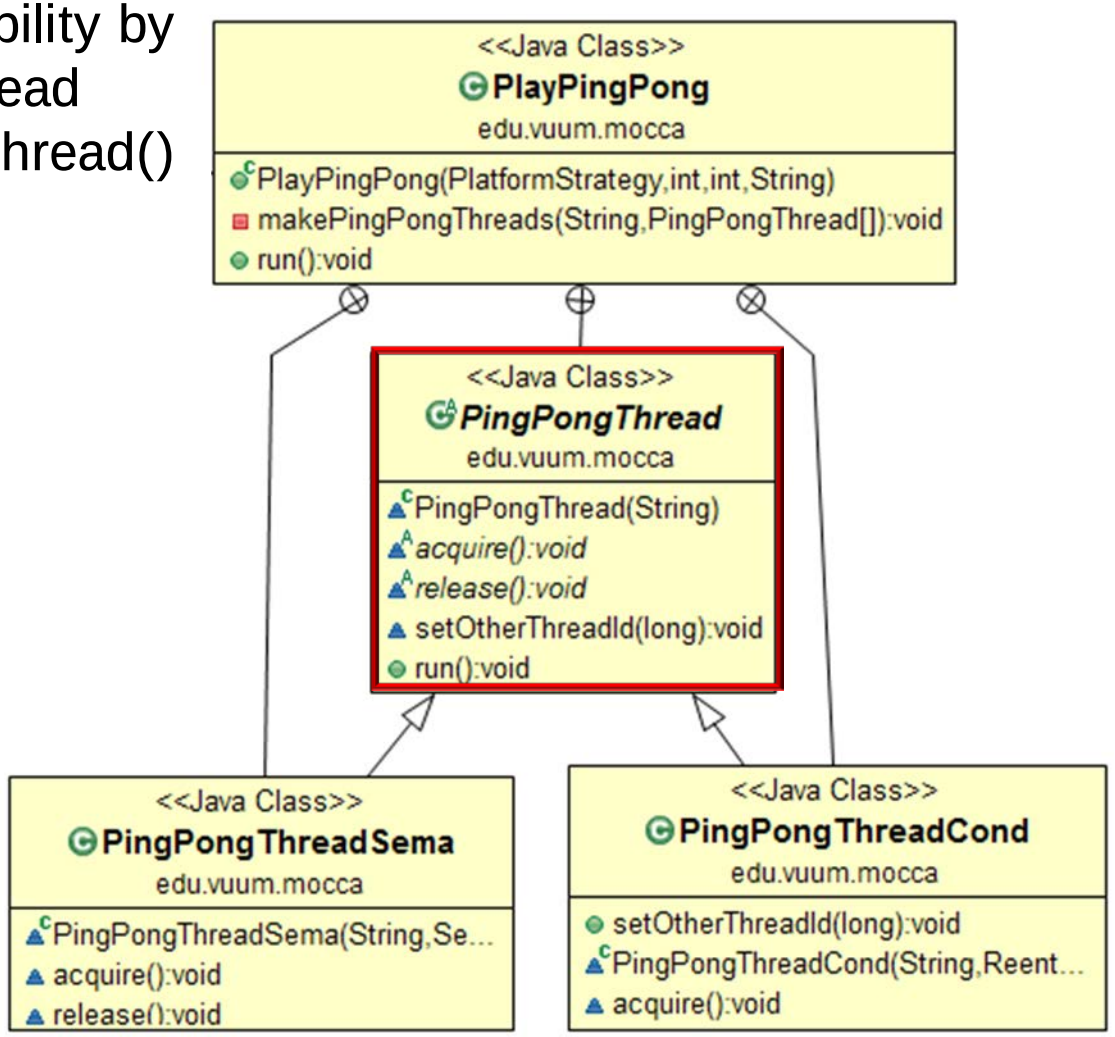
Applying the Factory Method Pattern

- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()



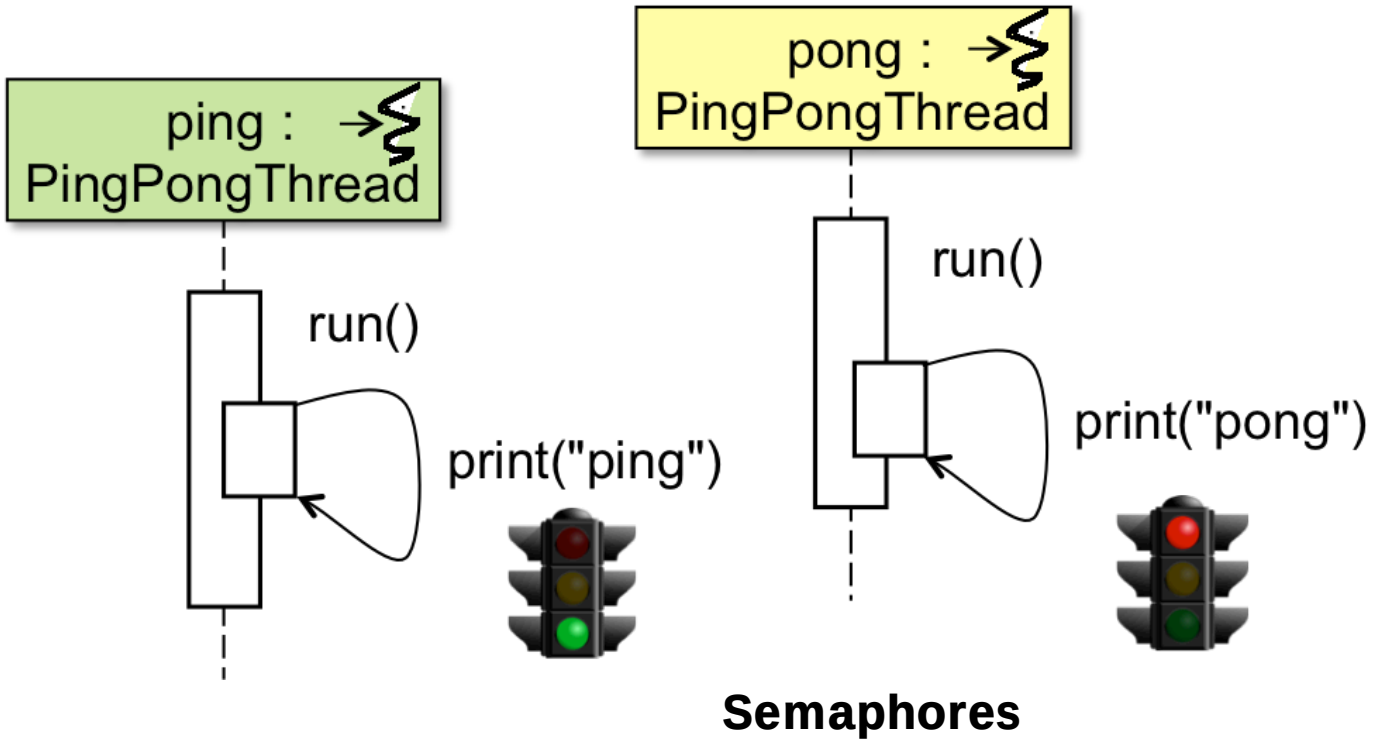
Applying the Factory Method Pattern

- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()



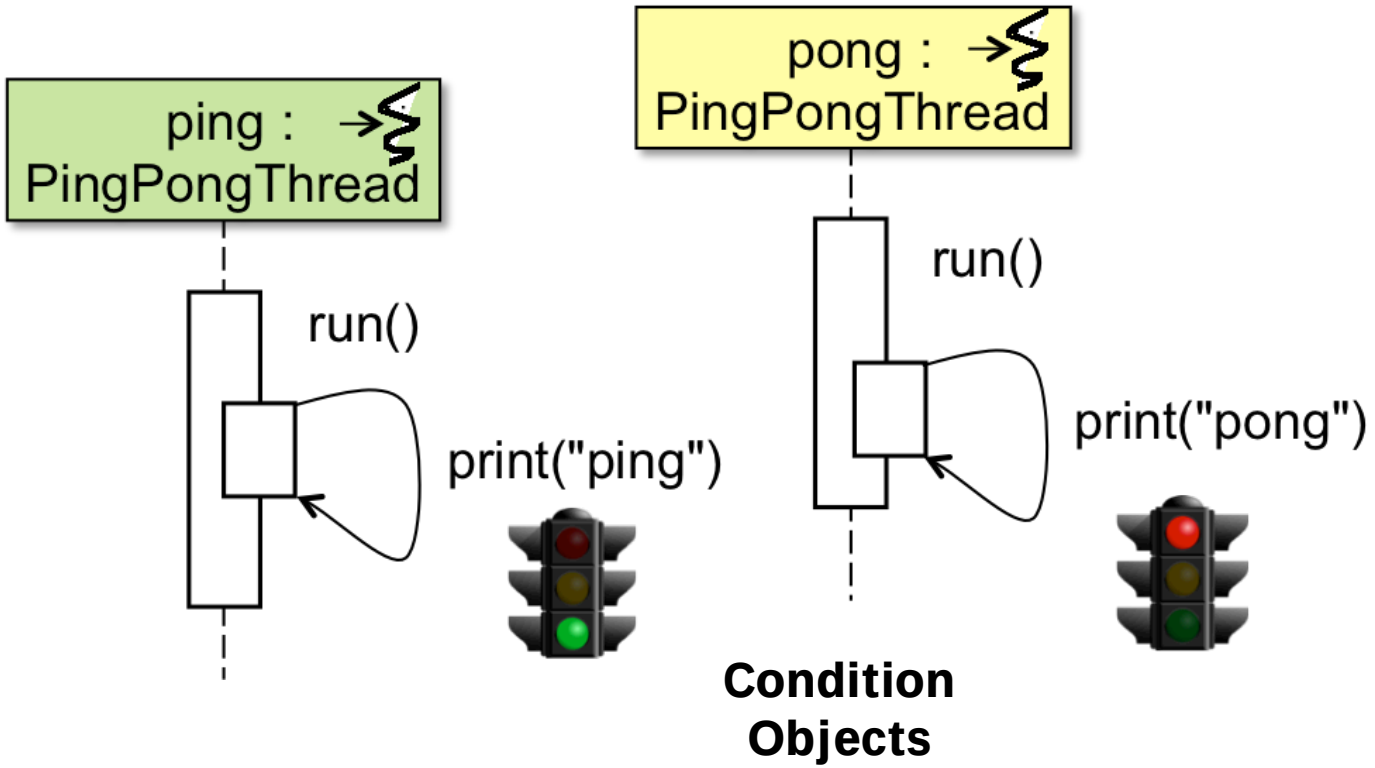
Applying the Factory Method Pattern

- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()



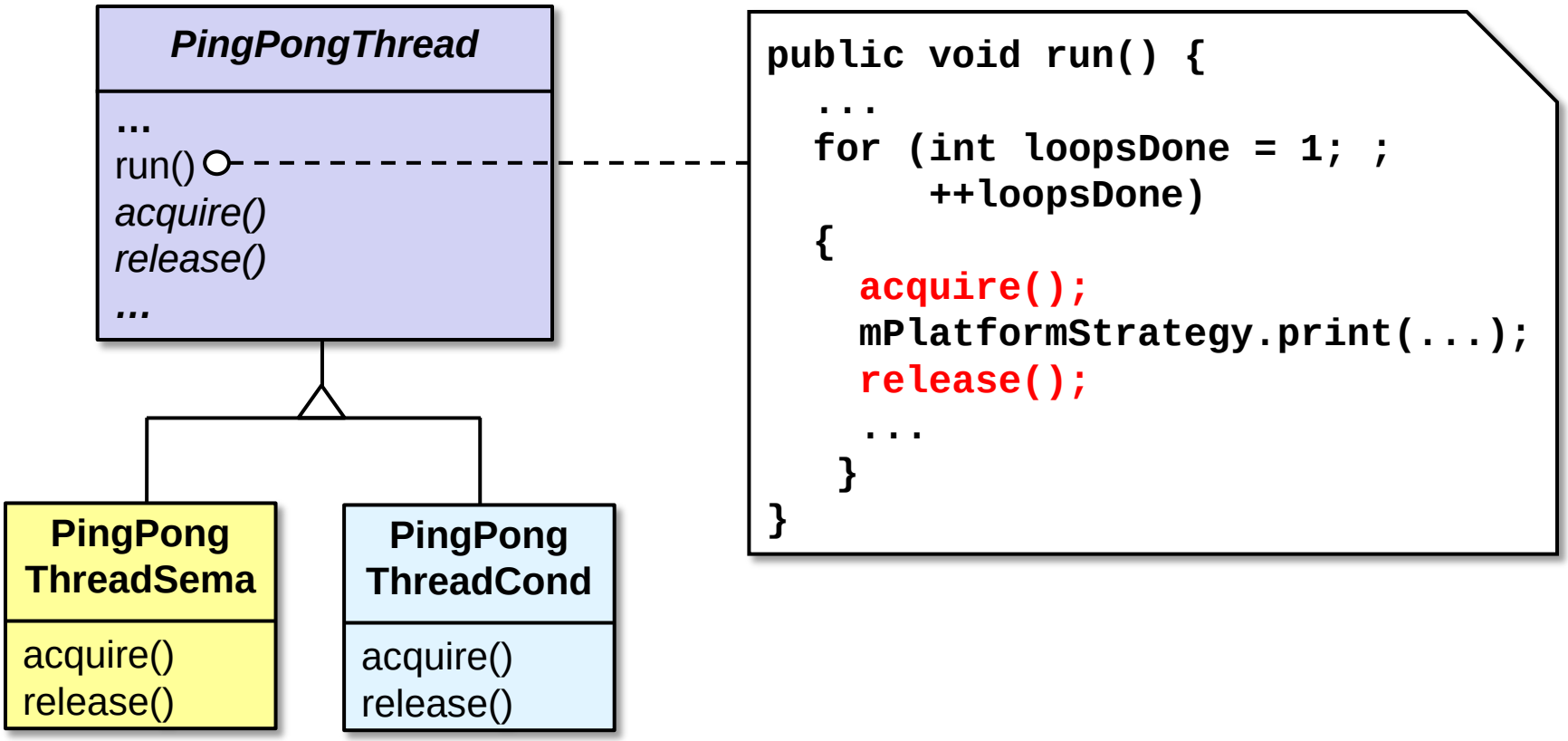
Applying the Factory Method Pattern

- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()



Applying the Factory Method Pattern

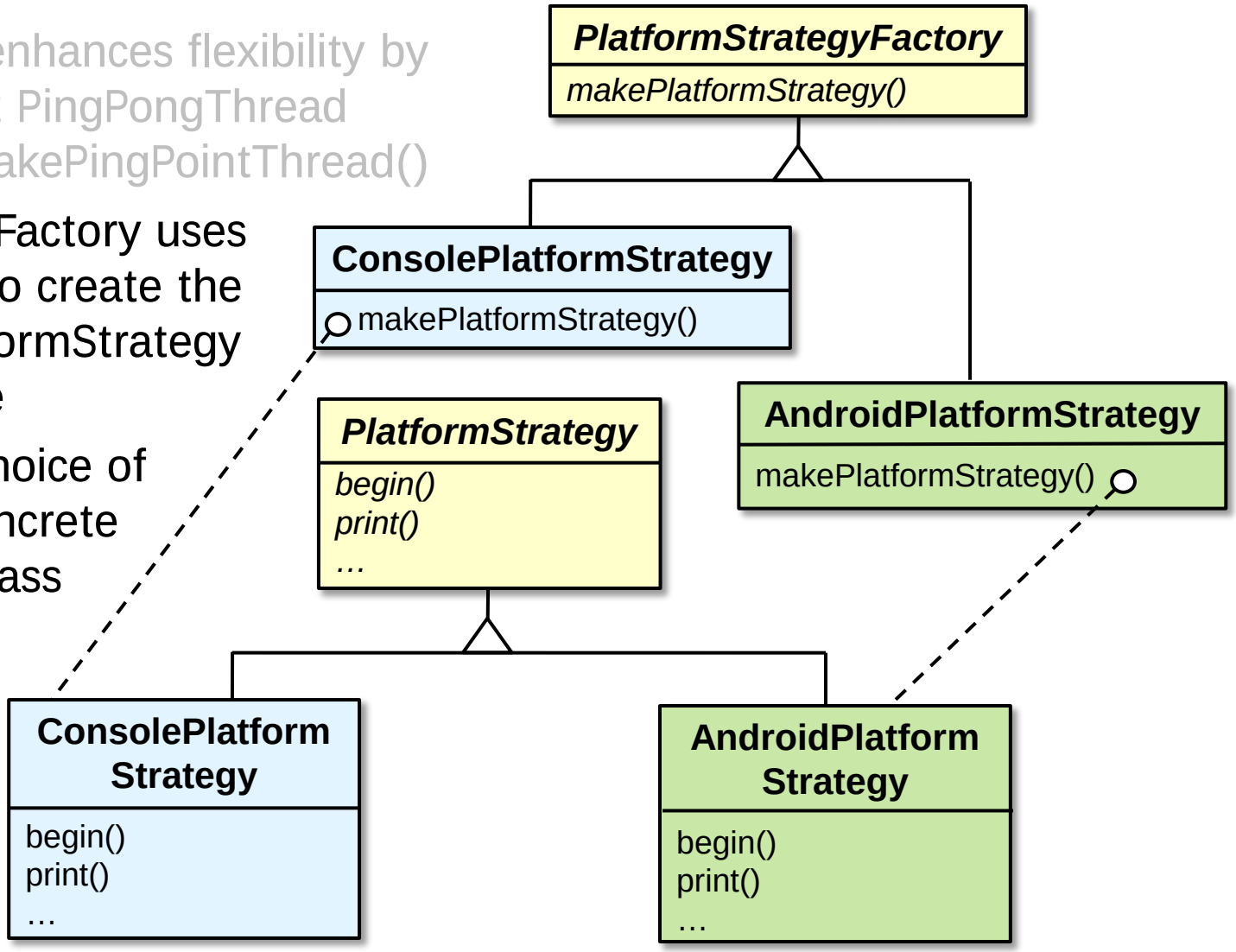
- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()



Supports new behavior & implementations without changing existing code

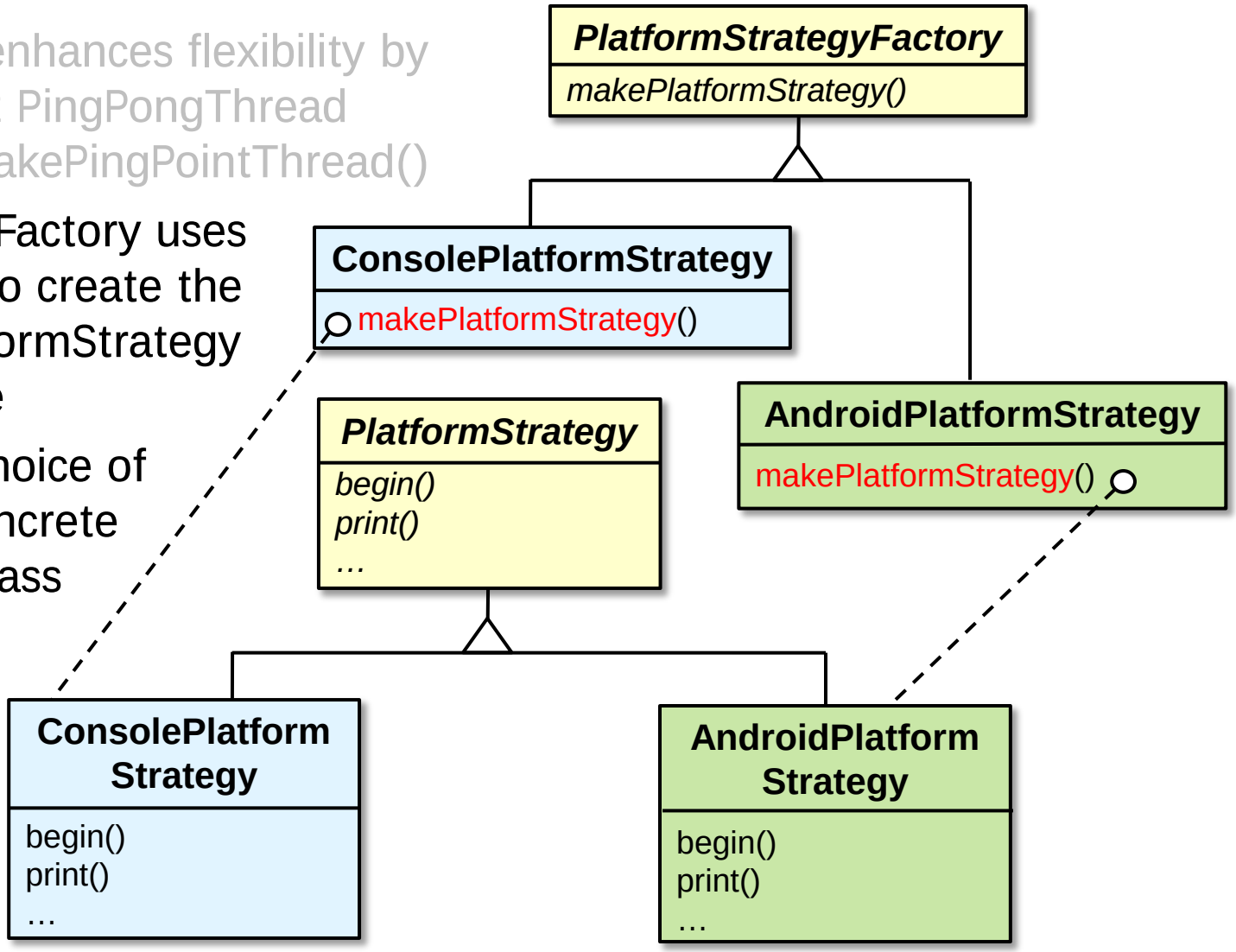
Applying the Factory Method Pattern

- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()
- PlatformStrategyFactory uses *Factory Method* to create the appropriate PlatformStrategy subclass instance
  - It leaves the choice of the object's concrete type to a subclass



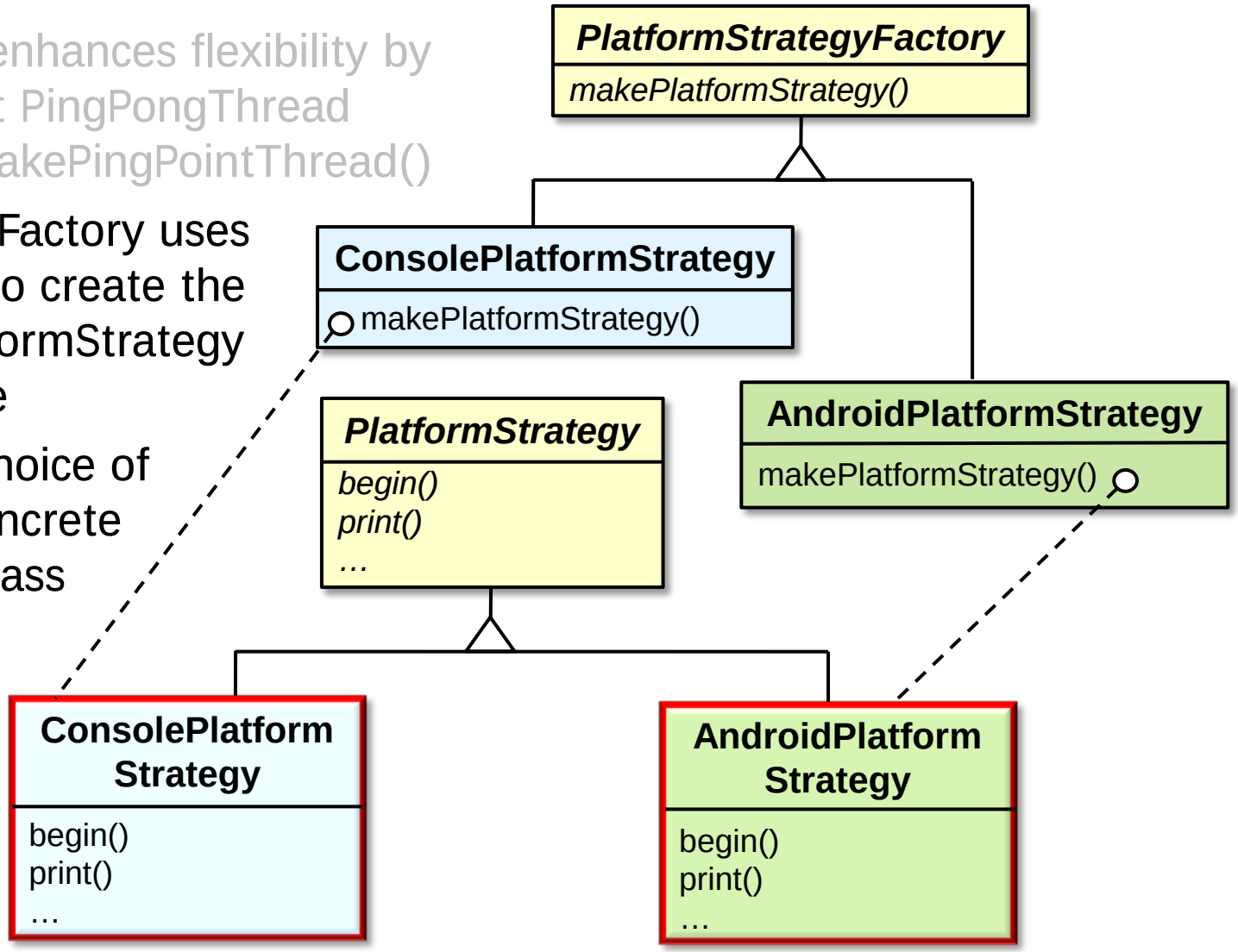
Applying the Factory Method Pattern

- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()
- PlatformStrategyFactory uses *Factory Method* to create the appropriate PlatformStrategy subclass instance
  - It leaves the choice of the object's concrete type to a subclass



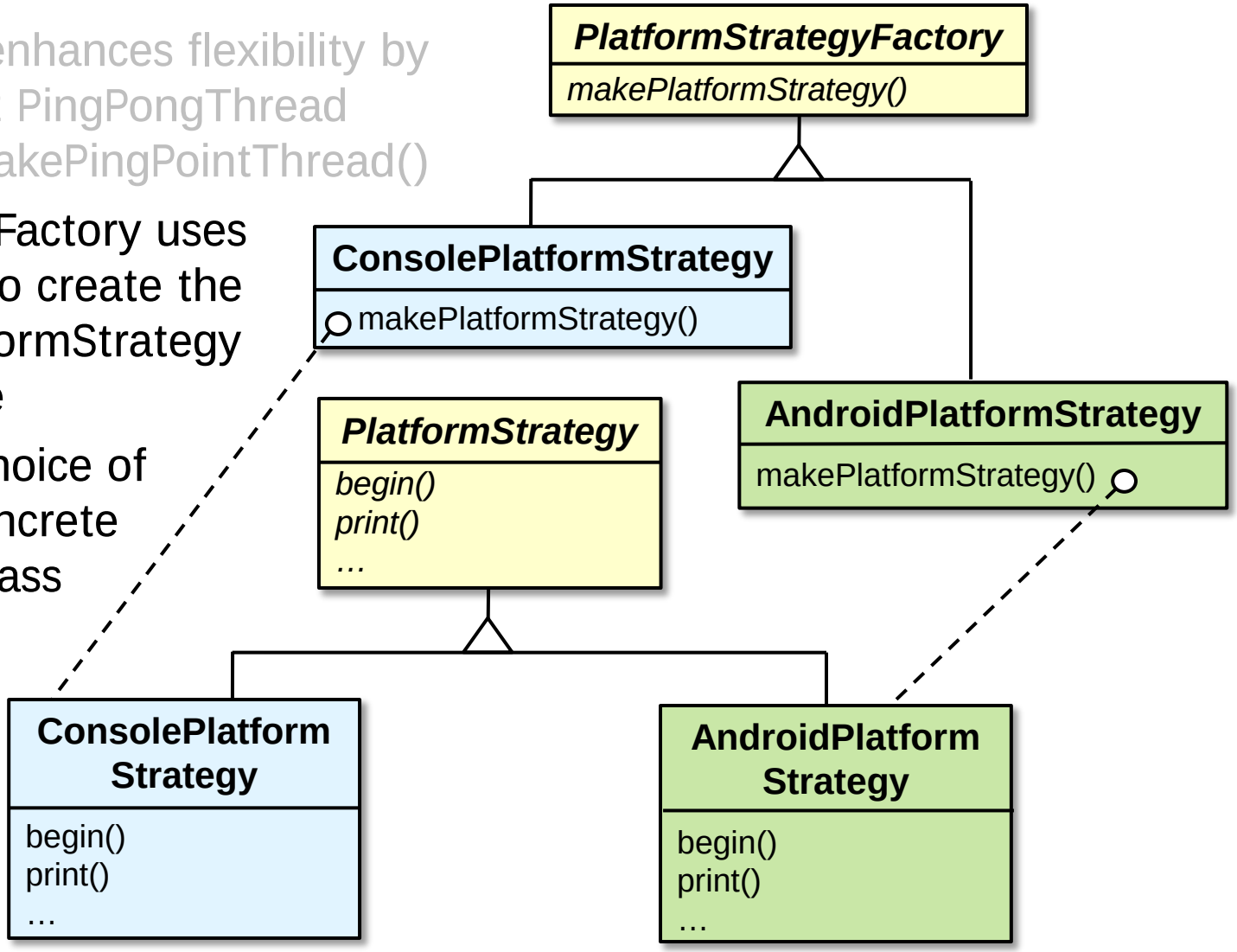
Applying the Factory Method Pattern

- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()
- PlatformStrategyFactory uses *Factory Method* to create the appropriate PlatformStrategy subclass instance
  - It leaves the choice of the object's concrete type to a subclass



Applying the Factory Method Pattern

- *Factory Method* enhances flexibility by creating different PingPongThread subclasses via makePingPointThread()
- PlatformStrategyFactory uses *Factory Method* to create the appropriate PlatformStrategy subclass instance
  - It leaves the choice of the object's concrete type to a subclass

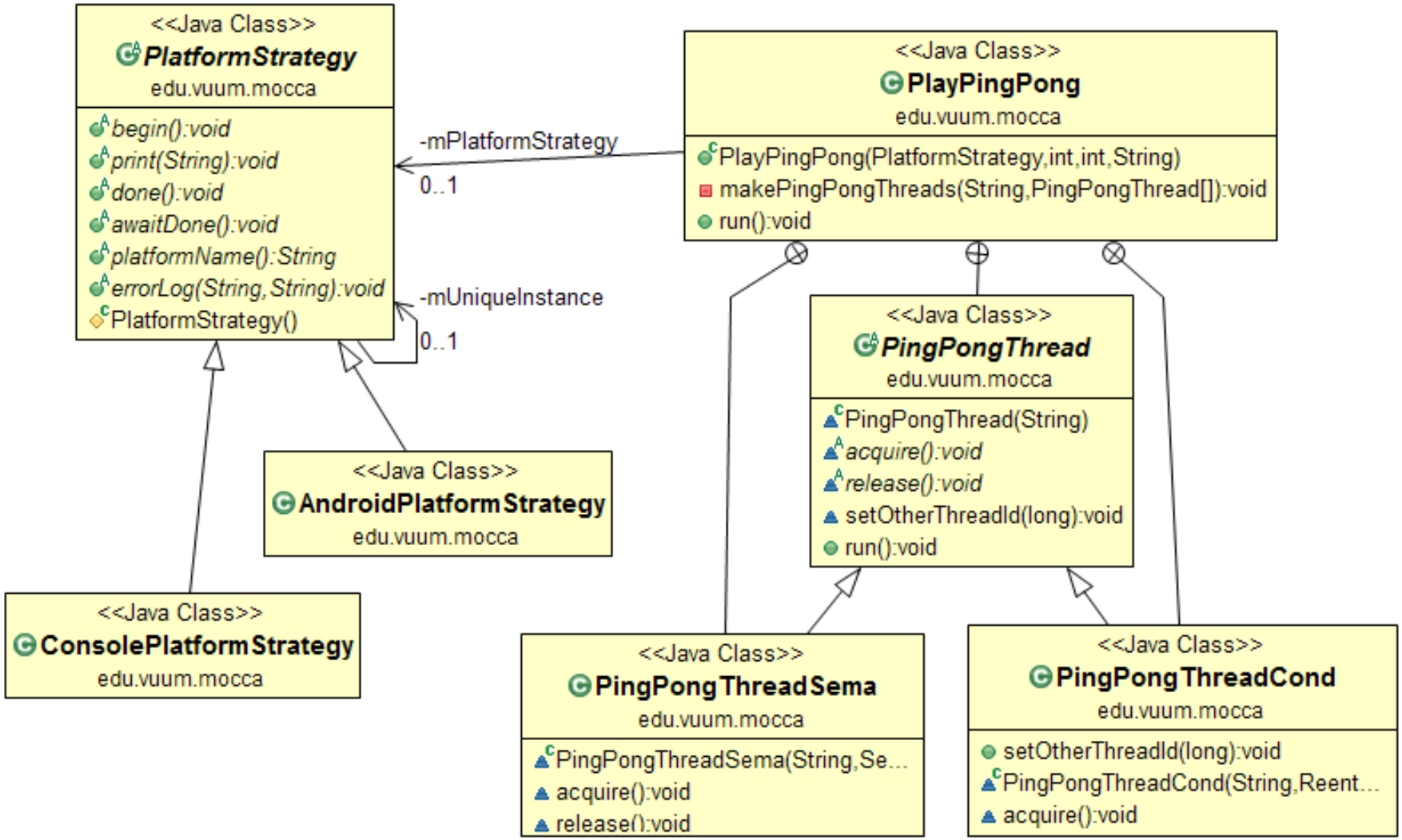


Factory Method also enhances systematic reuse & portability quality attributes

Summary

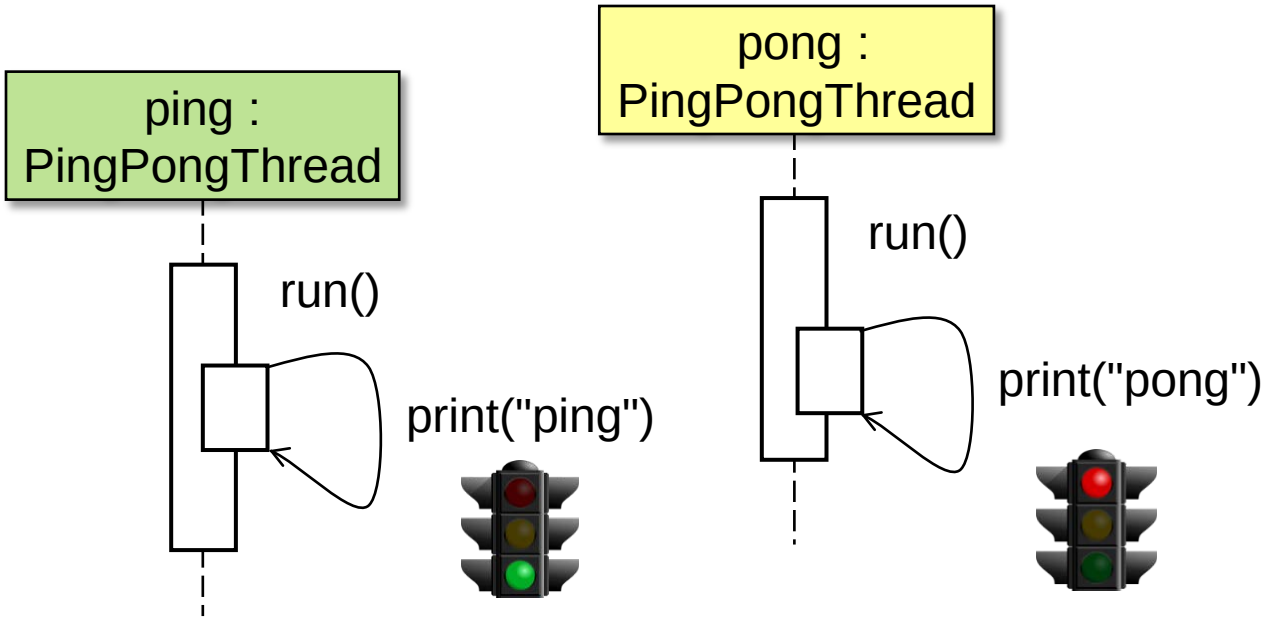
Summary

- The pattern-oriented framework supports a family of a concurrent ping-pong programs



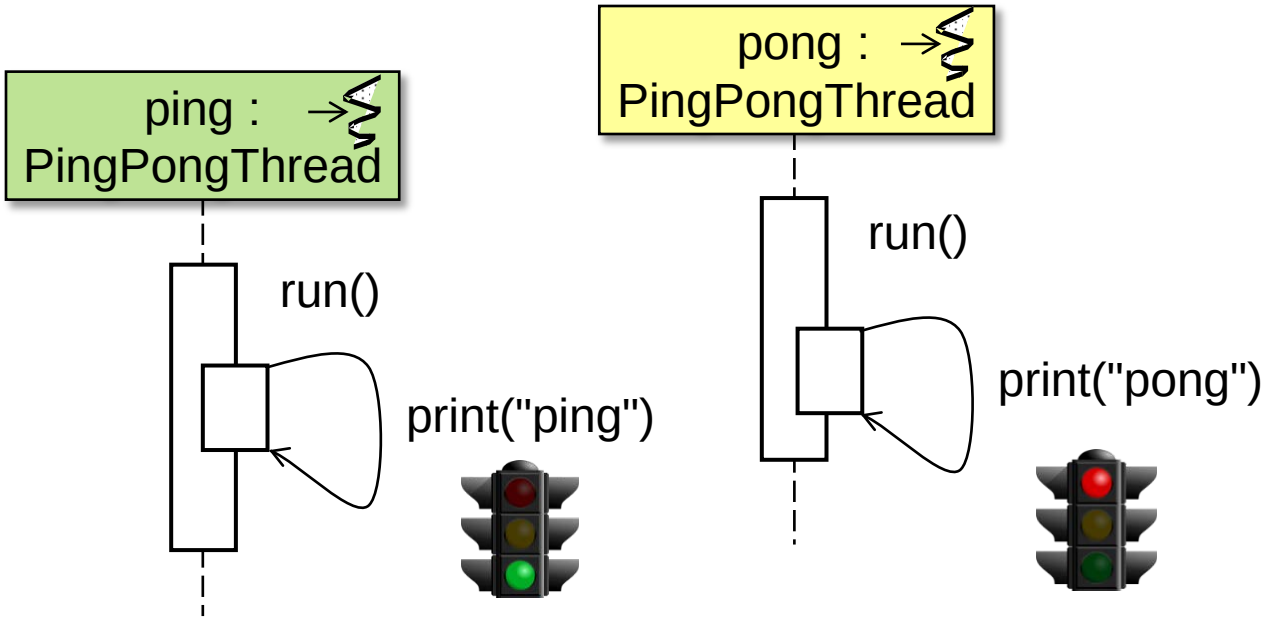
Summary

- The pattern-oriented framework supports a family of a concurrent ping-pong programs
  - We'll customize it to correctly alternate printing "ping" & "pong" to output by using Java synchronizers



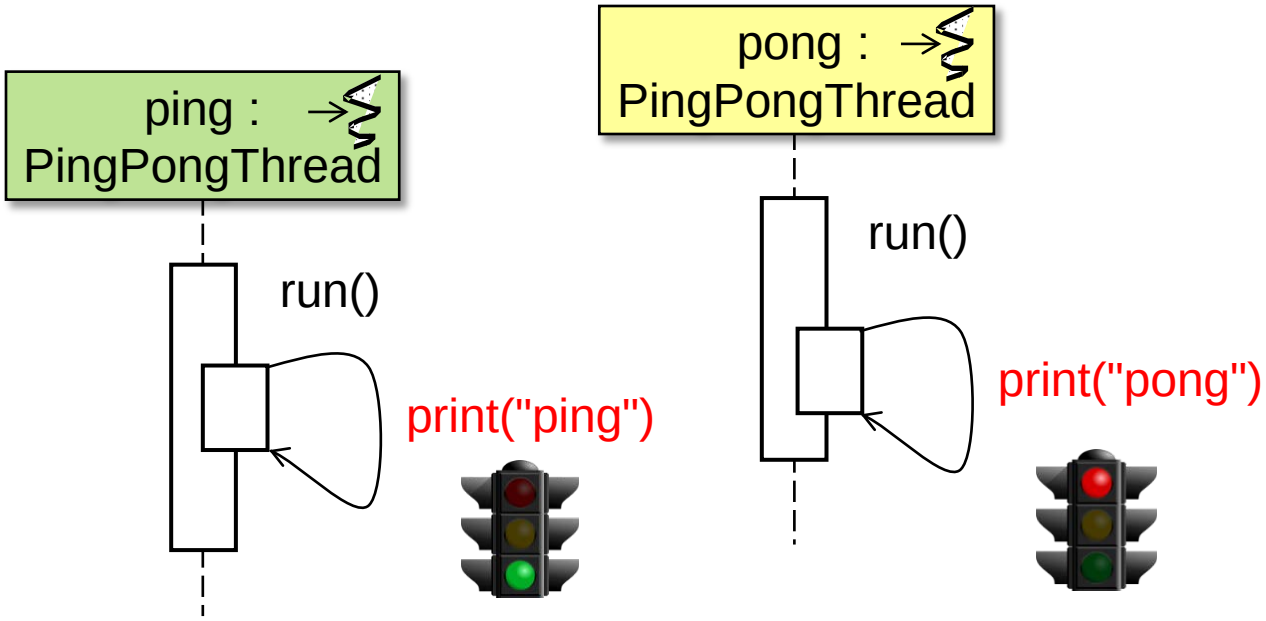
Summary

- The pattern-oriented framework supports a family of a concurrent ping-pong programs
  - We'll customize it to correctly alternate printing "ping" & "pong" to output by using Java synchronizers



Summary

- The pattern-oriented framework supports a family of a concurrent ping-pong programs
  - We'll customize it to correctly alternate printing "ping" & "pong" to output by using Java synchronizers

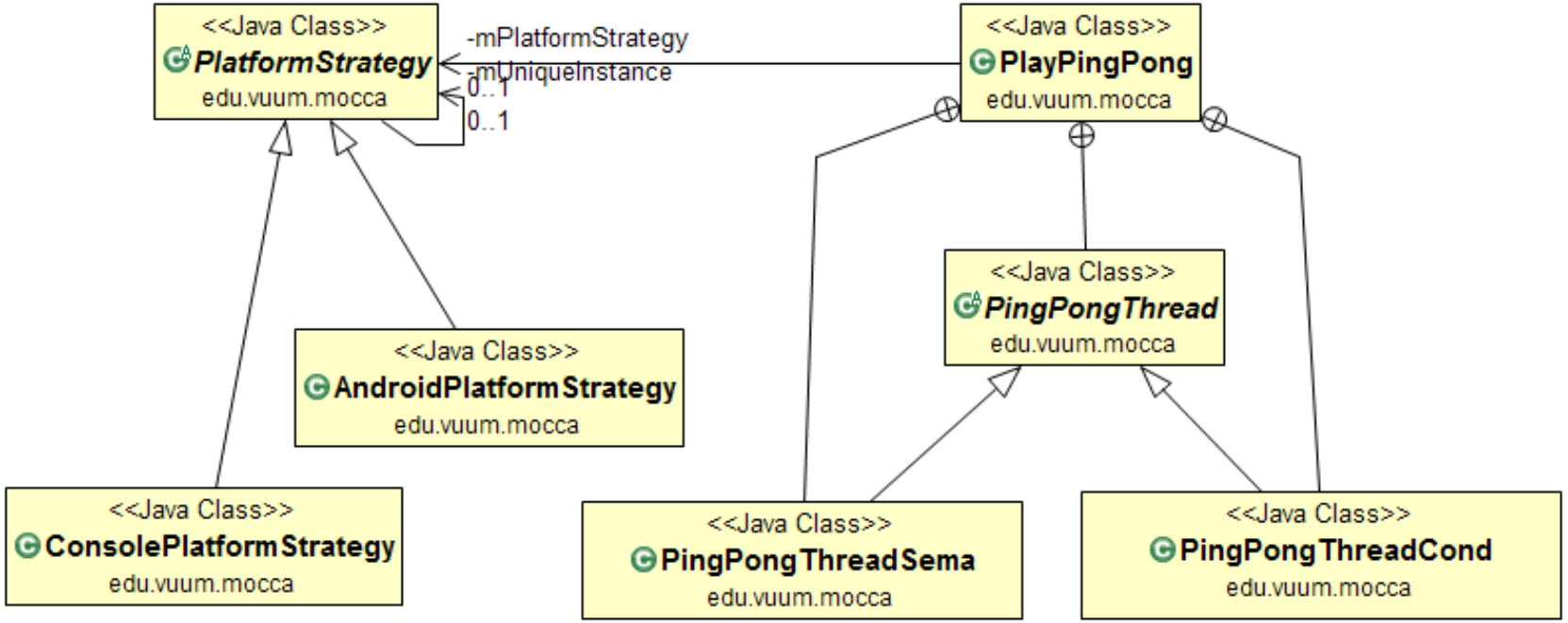



```
% java PlayPingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Ping!(10)
Pong!(10)
Done!
```

See earlier part on "Motivating Java Synchronization & Scheduling Mechanisms"

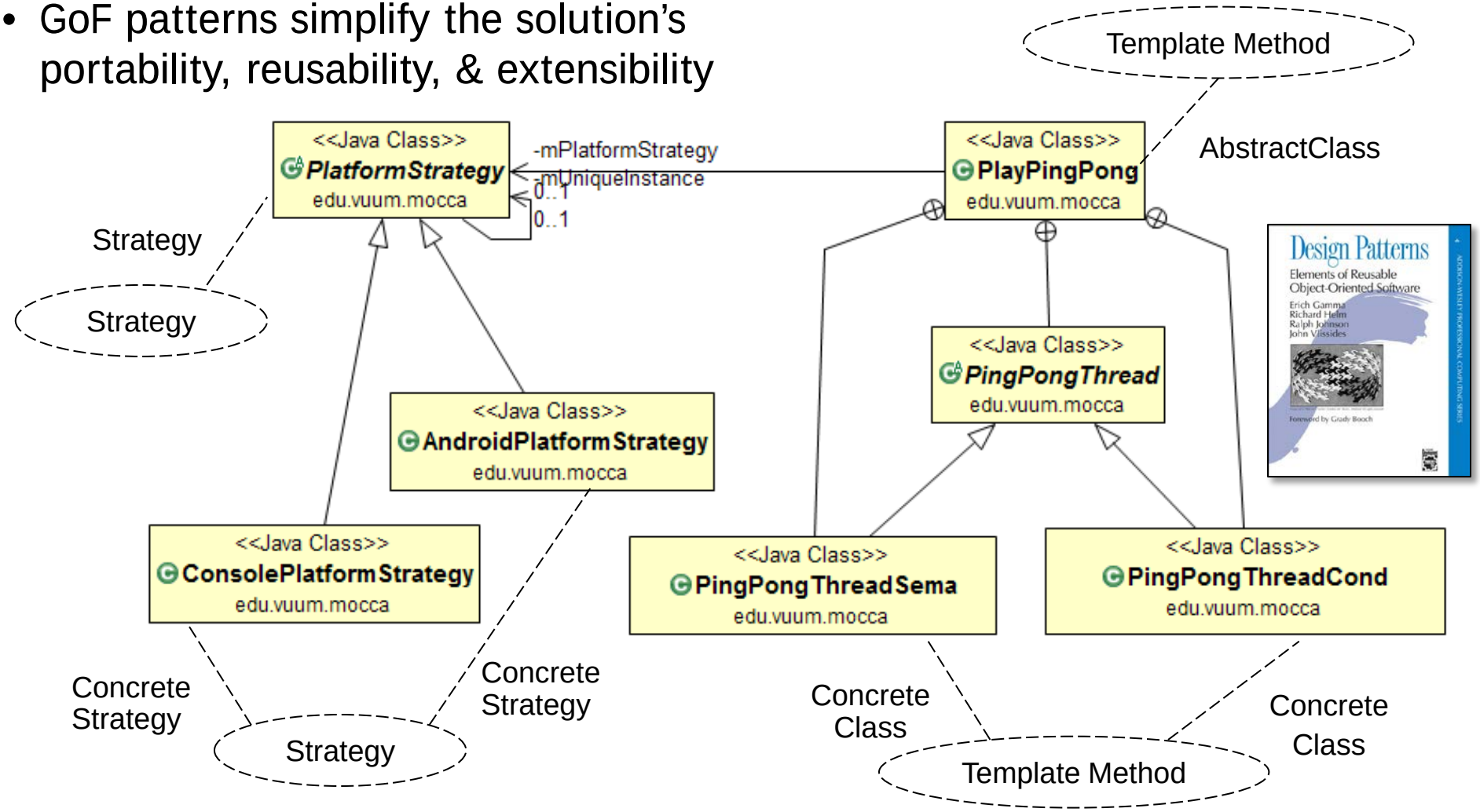
Summary

- The pattern-oriented framework supports a family of a concurrent ping-pong programs
- GoF patterns simplify the solution's portability, reusability, & extensibility



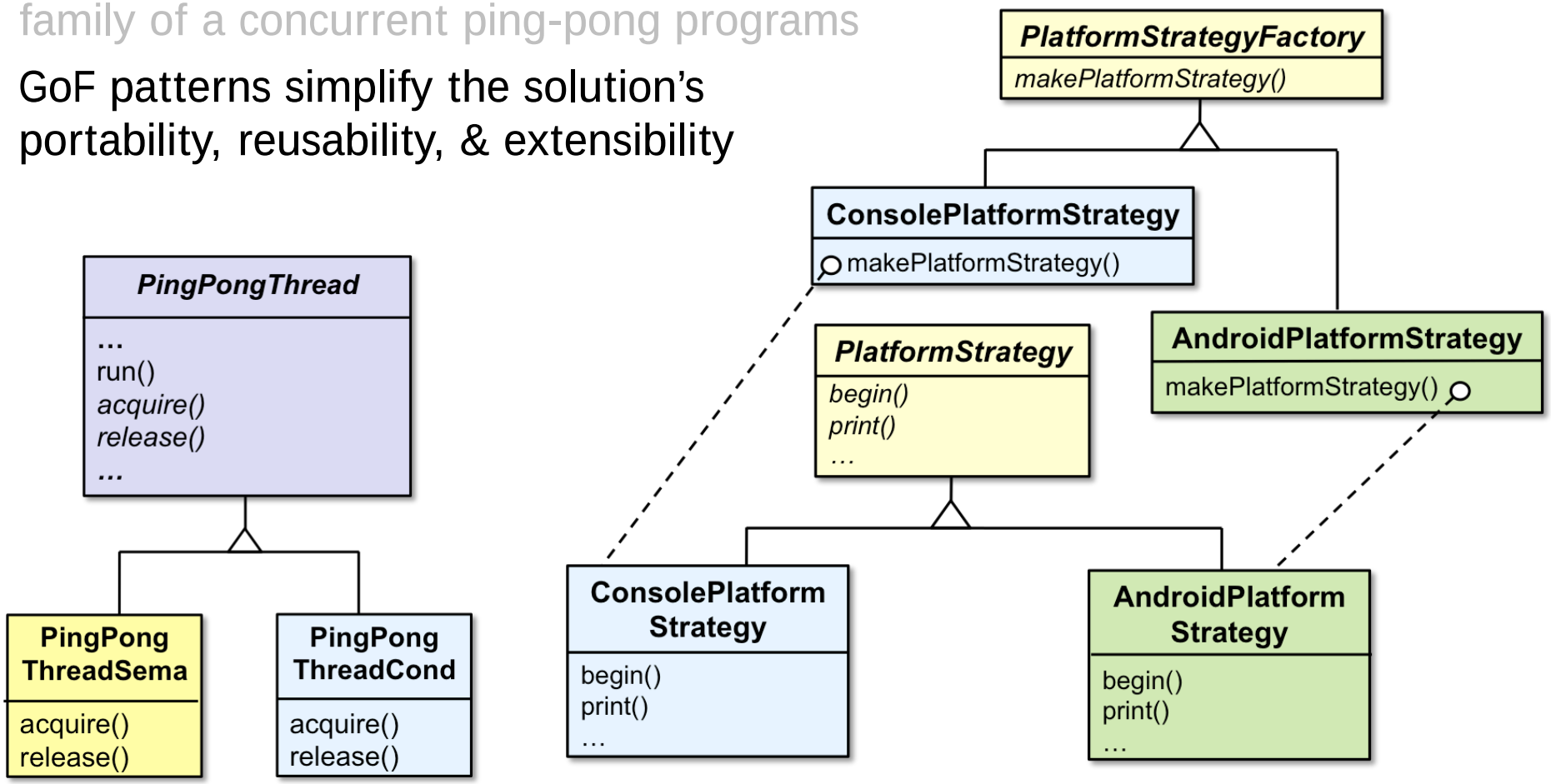
Summary

- The pattern-oriented framework supports a family of a concurrent ping-pong programs
- GoF patterns simplify the solution's portability, reusability, & extensibility



Summary

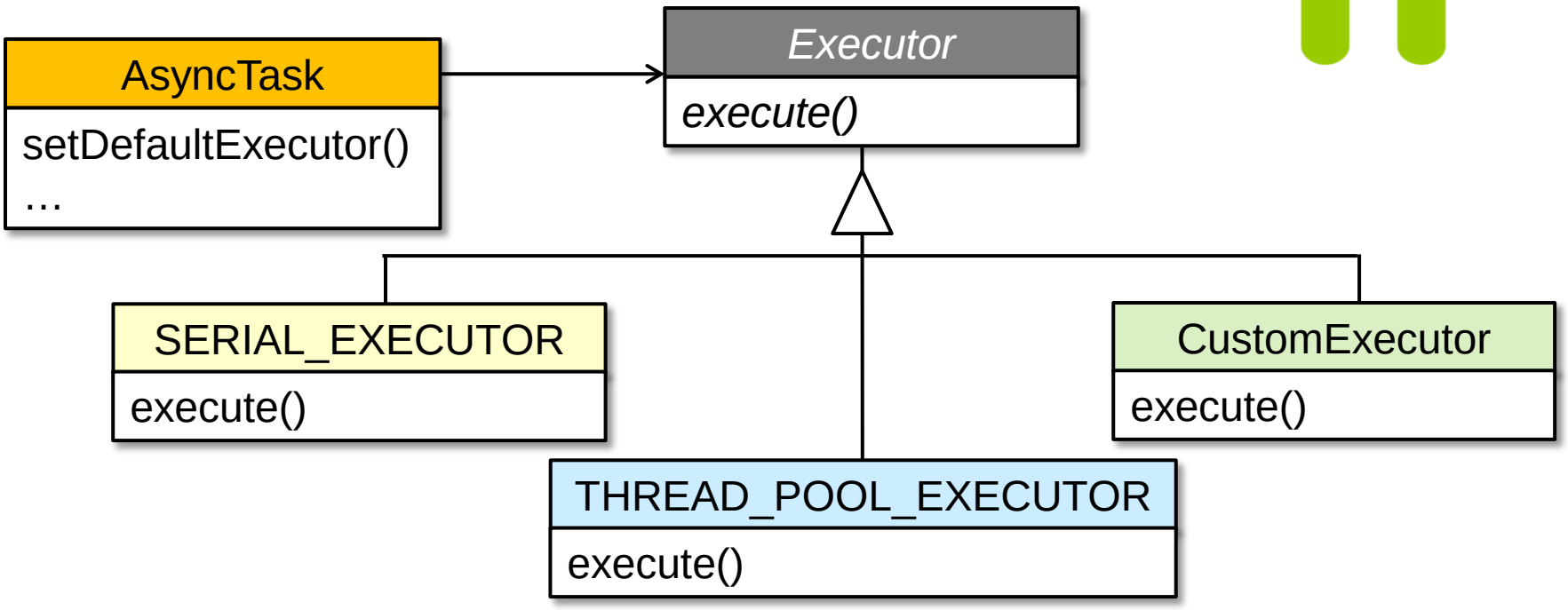
- The pattern-oriented framework supports a family of a concurrent ping-pong programs
- GoF patterns simplify the solution's portability, reusability, & extensibility



These patterns are broadly applicable to concurrent & non-concurrent software

Summary

- The pattern-oriented framework supports a family of a concurrent ping-pong programs
- GoF patterns simplify the solution's portability, reusability, & extensibility



See upcoming module on "Android Concurrency Frameworks"

Summary

- The pattern-oriented framework supports a family of a concurrent ping-pong programs
- GoF patterns simplify the solution's portability, reusability, & extensibility
- We'll use this framework to port our solution to Android