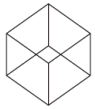
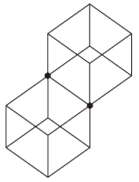
11. 템플릿 메서드 패턴



JAVA 객체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



학습목표

학습목표

- 공통 코드의 재사용 방법을 부분적으로 이해하기
- 템플릿 메서드 패턴을 이용한 코드 재사용 방법 이해하기
- 사례 연구를 통한 템플릿 메서드 패턴의 핵심 특징 이해하기

11.1 여러 회사의 모터를 지원하자

❖ 엘리베이터 제어 시스템에서 모터를 구동시키는 기능

- HyundaiMotor 클래스: 모터를 제어하여 엘리베이터를 이동시키는 클래스
- Door 클래스: 문을 열거나 닫는 기능을 제공하는 클래스

그림 11-1 현대 모터를 구동시키는 HyundaiMotor 클래스 설계

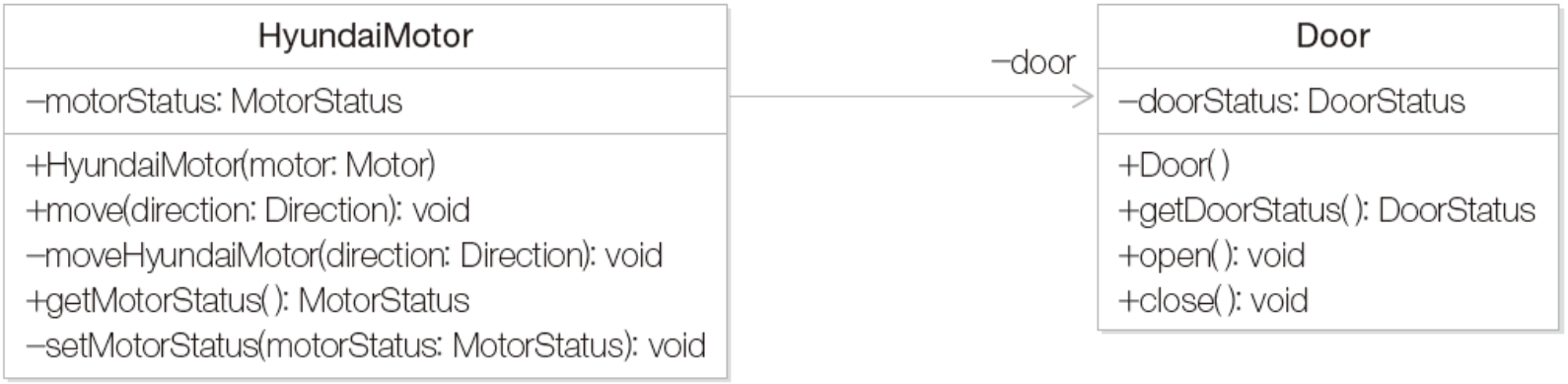
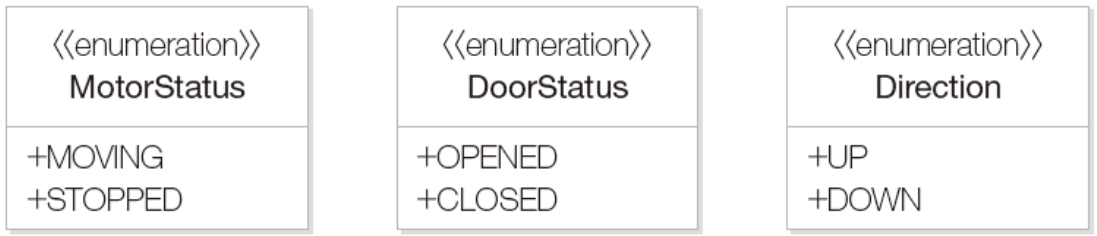
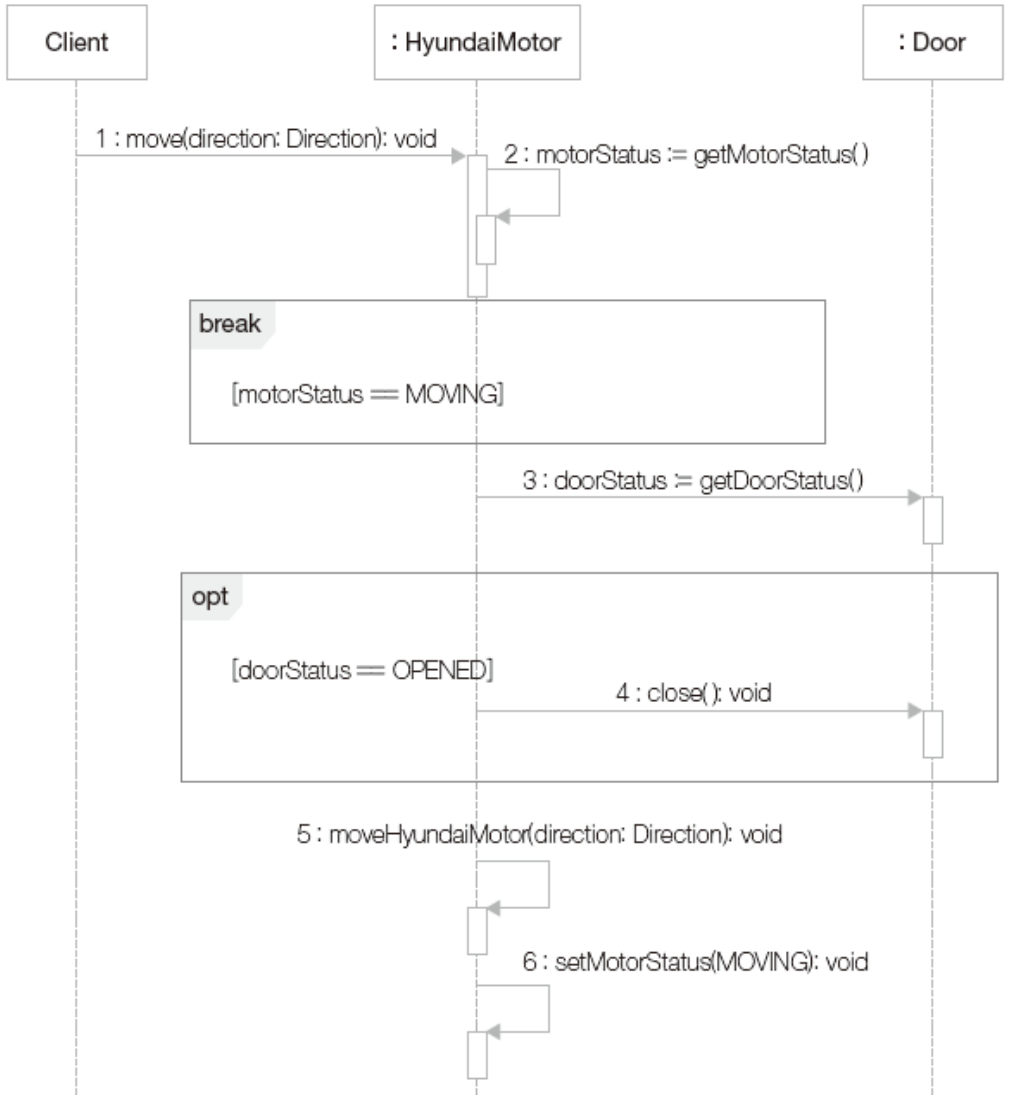


그림 11-2 Enumeration 인터페이스인 MotorStatus, DoorStatus, Direction의 설계



11.1 여러 회사의 모터를 지원하자

그림 11-3 HyundaiMotor 클래스의 move 메서드 설계



소스 코드

코드 11-1

```
public enum DoorStatus { CLOSED, OPENED }
public enum MotorStatus { MOVING, STOPPED}
public enum MotorStatus { MOVING, STOPPED}
public class Door {
    private DoorStatus doorStatus ;
    public Door() {
        doorStatus = DoorStatus.CLOSED ;
    }
    public DoorStatus getDoorStatus() {
        return doorStatus ;
    }
    public void close() {
        doorStatus = DoorStatus.CLOSED ;
    }
    public void open() {
        doorStatus = DoorStatus.OPENED ;
    }
}
```

소스 코드

코드 11-1

```
public class HyundaiMotor {
    private Door door ;
    private MotorStatus motorStatus ;
    public HyundaiMotor(Door door) {
        this.door = door ;
        motorStatus = MotorStatus.STOPPED ;  // 초기에는 멈춘 상태
    }
    private void moveHyundaiMotor(Direction direction) {
        // Hyundai Motor를 구동시킨다.
    }
    public MotorStatus getMotorStatus() { return motorStatus; }
    private void setMotorStatus(MotorStatus motorStatus) { this.motorStatus = motorStatus; }
    public void move(Direction direction) {
        MotorStatus motorStatus = getMotorStatus() ;
        if (  motorStatus == MotorStatus.MOVING ) return ;  // 이미 이동 중이면 아무 작업을 하지 않음

        DoorStatus doorStatus = door.getDoorStatus() ;
        if ( doorStatus == DoorStatus.OPENED ) door.close() ;  // 만약 문이 열려 있으면 먼저 문을 닫음

        moveHyundaiMotor(direction) ;  // 모터를 주어진 방향으로 이동
        setMotorStatus(MotorStatus.MOVING) ;  // 모터 상태를 이동 중으로 변경함
    }
}
```

소스 코드

코드 11-1

```
public class Client {
    public static void main(String[] args) {
        Door door = new Door() ;
        HyundaiMotor hyundaiMotor = new HyundaiMotor(door) ;
        hyundaiMotor.move(Direction.UP) ;
    }
}
```

11.2 문제점

- HyundaiMotor 클래스는 현대모터를 구동시킨다. 만약 다른 회사의 모터를 제어해야 한다면? 예를 들어 LG모터를 구동시키기 위해서는 어떻게 해야 할까?

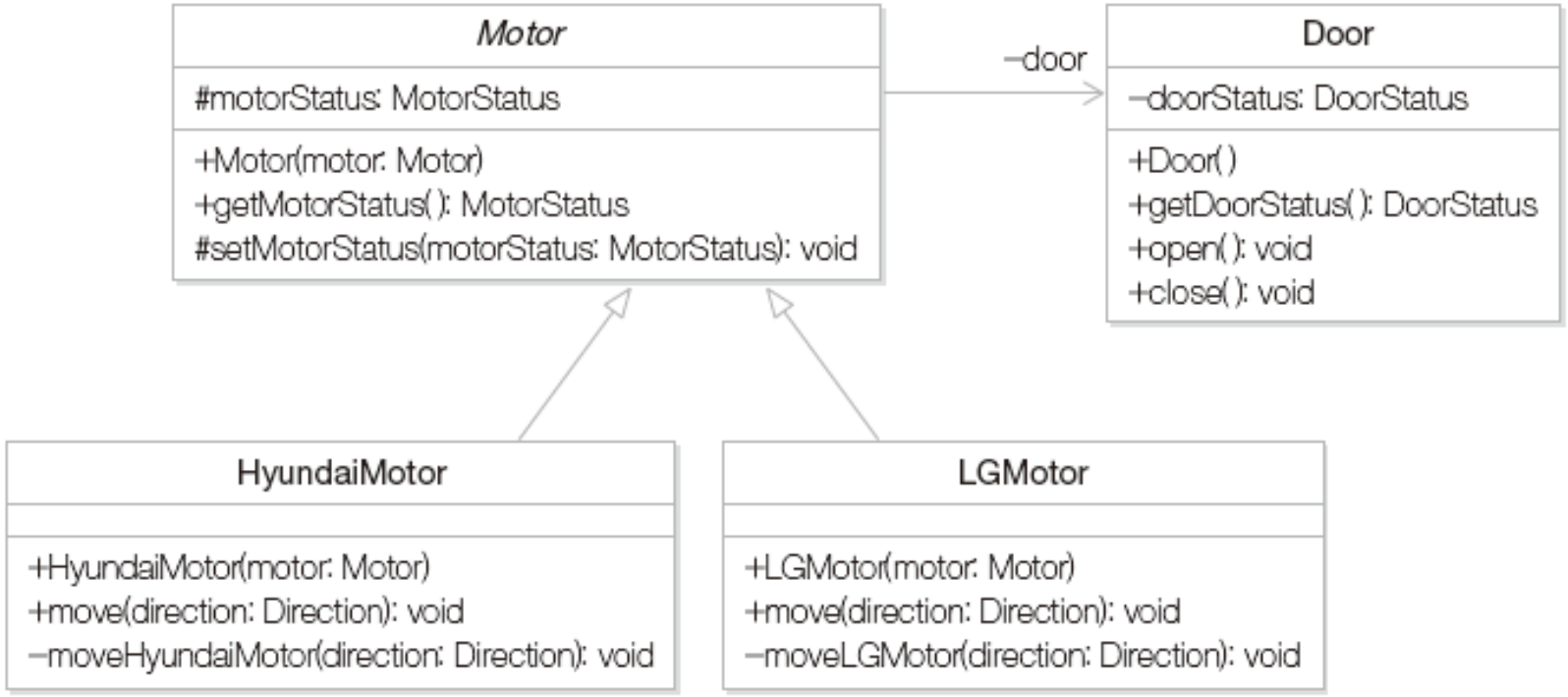
LGMotor 클래스

코드 11-2

```
public class LGMotor {
    private Door door ;
    private MotorStatus motorStatus ;
    public LGMotor(Door door) {
        this.door = door ; motorStatus = MotorStatus.STOPPED ;
    }
    private void moveLGMotor(Direction direction) {
        // LG Motor를 구동시킴
    }
    public MotorStatus getMotorStatus() { return motorStatus; }
    private void setMotorStatus(MotorStatus motorStatus) {
        this.motorStatus = motorStatus;
    }
    public void move(Direction direction) {
        MotorStatus motorStatus = getMotorStatus() ;
        if (  motorStatus == MotorStatus.MOVING ) return ;
        DoorStatus doorStatus = door.getDoorStatus() ;
        if ( doorStatus == DoorStatus.OPENED ) door.close() ;
        moveLGMotor(direction) ;  // move 메서드는 이 문장을 제외하면 HyundaiMotor와 동일함
        setMotorStatus(MotorStatus.MOVING) ;
    }
}
```

Motor 클래스 설계

그림 11-4 HyundaiMotor와 LGMotor 클래스의 상위 클래스인 Motor의 정의



Motor 클래스 소스 코드

코드 11-3

```
public abstract class Motor { // HyundaiMotor와 LGMotor에 공통적인 기능을 구현하는 클래스
   protected Door door ;
   private MotorStatus motorStatus ;

   public Motor(Door door) {
      this.door = door ;
      motorStatus = MotorStatus.STOPPED ;
   }
   public MotorStatus getMotorStatus() {
      return motorStatus;
   }
   protected void setMotorStatus(MotorStatus motorStatus) {
      this.motorStatus = motorStatus;
   }
}
```

HyundaiMotor 클래스 소스 코드

코드 11-3

```
public class HyundaiMotor extends Motor { // Motor를 상속받아서 HyundaiMotor를 구현함
   public HyundaiMotor(Door door) {
      super(door) ;
   }
   private void moveHyundaiMotor(Direction direction) {
      // Hyundai Motor를 구동시킨다.
   }
   public void move(Direction direction) {
      MotorStatus motorStatus = getMotorStatus() ;
      if (  motorStatus == MotorStatus.MOVING ) return ;

      DoorStatus doorStatus = door.getDoorStatus() ;
      if ( doorStatus == DoorStatus.OPENED )
         door.close() ;

      moveHyundaiMotor(direction) ;  // move 메서드는 이 구문을 제외하면 LGMotor와 동일함

      setMotorStatus(MotorStatus.MOVING) ;
   }
}
```

LGMotor 클래스 소스 코드

코드 11-3

```
public class LGMotor extends Motor {
   public LGMotor(Door door) {
      super(door) ;
   }
   private void moveLGMotor(Direction direction) {
      // LG Motor를 구동시킨다.
   }
   public void move(Direction direction) {
      MotorStatus motorStatus = getMotorStatus() ;
      if (  motorStatus == MotorStatus.MOVING ) return ;

      DoorStatus doorStatus = door.getDoorStatus() ;
      if ( doorStatus == DoorStatus.OPENED )
         door.close() ;

      moveLGMotor(direction) ; // move 메서드는 이 구문을 제외하면 HyundaiMotor와 동일함

      setMotorStatus(MotorStatus.MOVING) ;
   }
}
```

HyundaiMotor와 LGMotor의 move 메서드

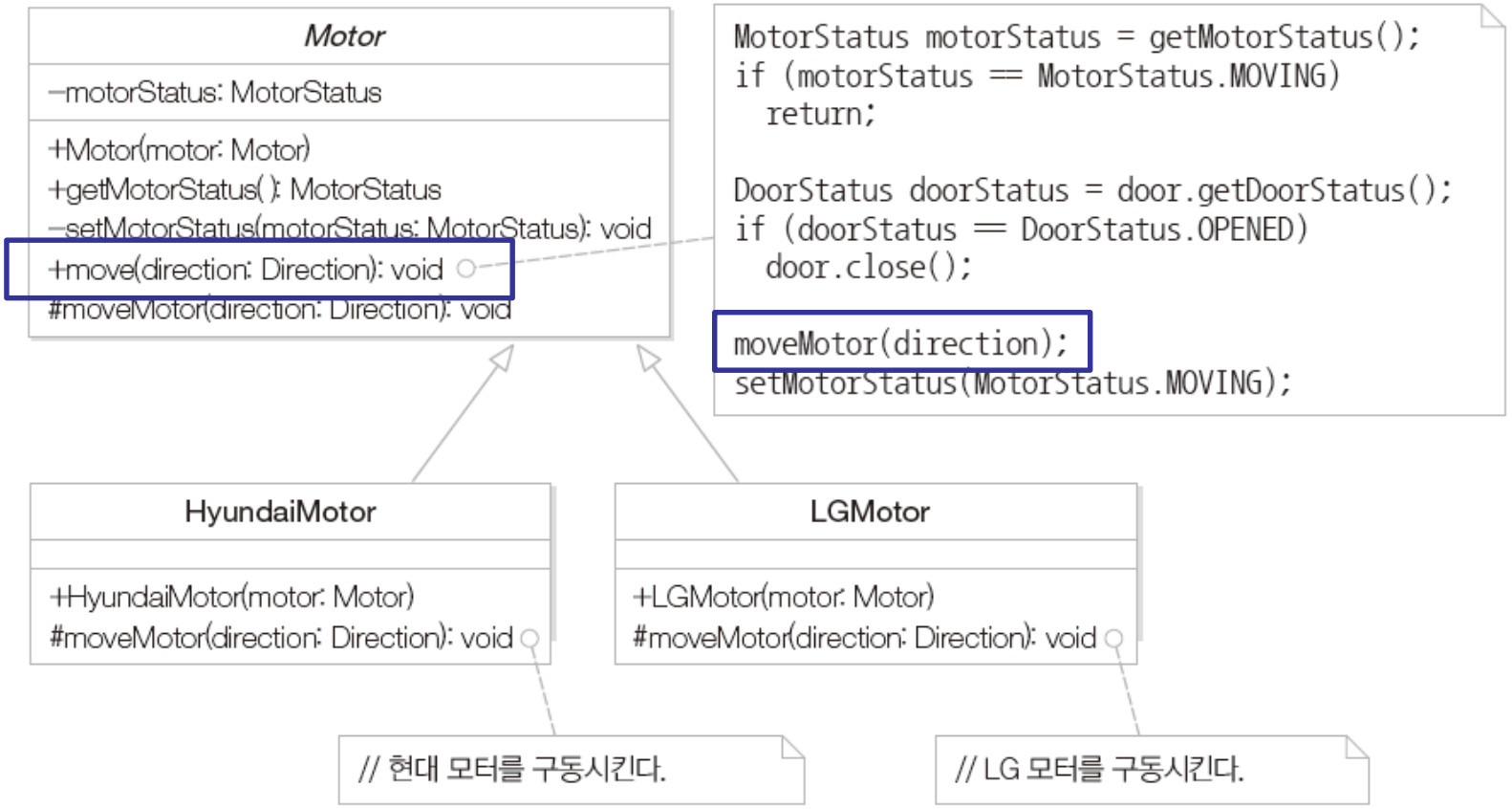
```
public void move(Direction direction) {
   MotorStatus motorStatus = getMotorStatus() ;
   if (  motorStatus == MotorStatus.MOVING )
      return ;

   DoorStatus doorStatus=door.getDoorStatus() ;
   if ( doorStatus == DoorStatus.OPENED )
     door.close() ;

   moveHyundaiMotor(direction) ;

   setMotorStatus(MotorStatus.MOVING) ;
}
```

```
public void move(Direction direction) {
   MotorStatus motorStatus = getMotorStatus() ;
   if (  motorStatus == MotorStatus.MOVING )
      return ;

   DoorStatus doorStatus=door.getDoorStatus() ;
   if ( doorStatus == DoorStatus.OPENED )
      door.close() ;

   moveLGMotor(direction) ;

   setMotorStatus(MotorStatus.MOVING) ;
}
```

11.3. 해결책

❖ move 메서드에서 공통적인 부분을 상위 클래스 Motor로 이동

그림 11-5 move 메서드의 중복 코드를 최소화한 설계



11.3. 해결책: 소스 코드

코드 11-4

```
public abstract class Motor {
   private Door door ;
   private MotorStatus motorStatus ;

   public Motor(Door door) {
      this.door = door ;
      motorStatus = MotorStatus.STOPPED ;
   }
   public MotorStatus getMotorStatus() { return motorStatus; }
   private void setMotorStatus(MotorStatus motorStatus) {
      this.motorStatus = motorStatus;
   }
   public void move(Direction direction) { // LGMotor와 HyundaiMotor의 move에서 공통만을 가짐
      MotorStatus motorStatus = getMotorStatus() ;
      if (  motorStatus == MotorStatus.MOVING ) return ;

      DoorStatus doorStatus = door.getDoorStatus() ;
      if ( doorStatus == DoorStatus.OPENED ) door.close() ;

      moveMotor(direction) ; // 하위 클래스에서 override됨
      setMotorStatus(MotorStatus.MOVING) ;
   }
   protected abstract void moveMotor(Direction direction) ;
}
```

11.3. 해결책: 소스 코드

코드 11-4

```
public class HyundaiMotor extends Motor {
   public HyundaiMotor(Door door) {
      super(door) ;
   }
   protected void moveMotor(Direction direction) {
      // Hyundai Motor를 구동시킨다.
   }
}
```

```
public class LGMotor extends Motor {
   public LGMotor(Door door) {
      super(door) ;
   }
   protected void moveMotor(Direction direction) {
      // LG Motor를 구동시킨다.
   }
}
```

11.4 템플릿 메서드 패턴

- 전체적으로 동일하면서 부분적으로 상이한 문장을 가지는 메소드의 코드 중복을 최소화할 때 유용

템플릿 메소드 패턴은 전체적인 알고리즘을 구현하면서 상이한 부분은 하위 클래스에서구현할 수 있도록 해 주는 디자인 패턴으로서 전체적인 알고리즘의 코드를 재사용하는 데 유용하다.

11.4 템플릿 메서드 패턴

그림 11-6 템플릿 메서드의 개념

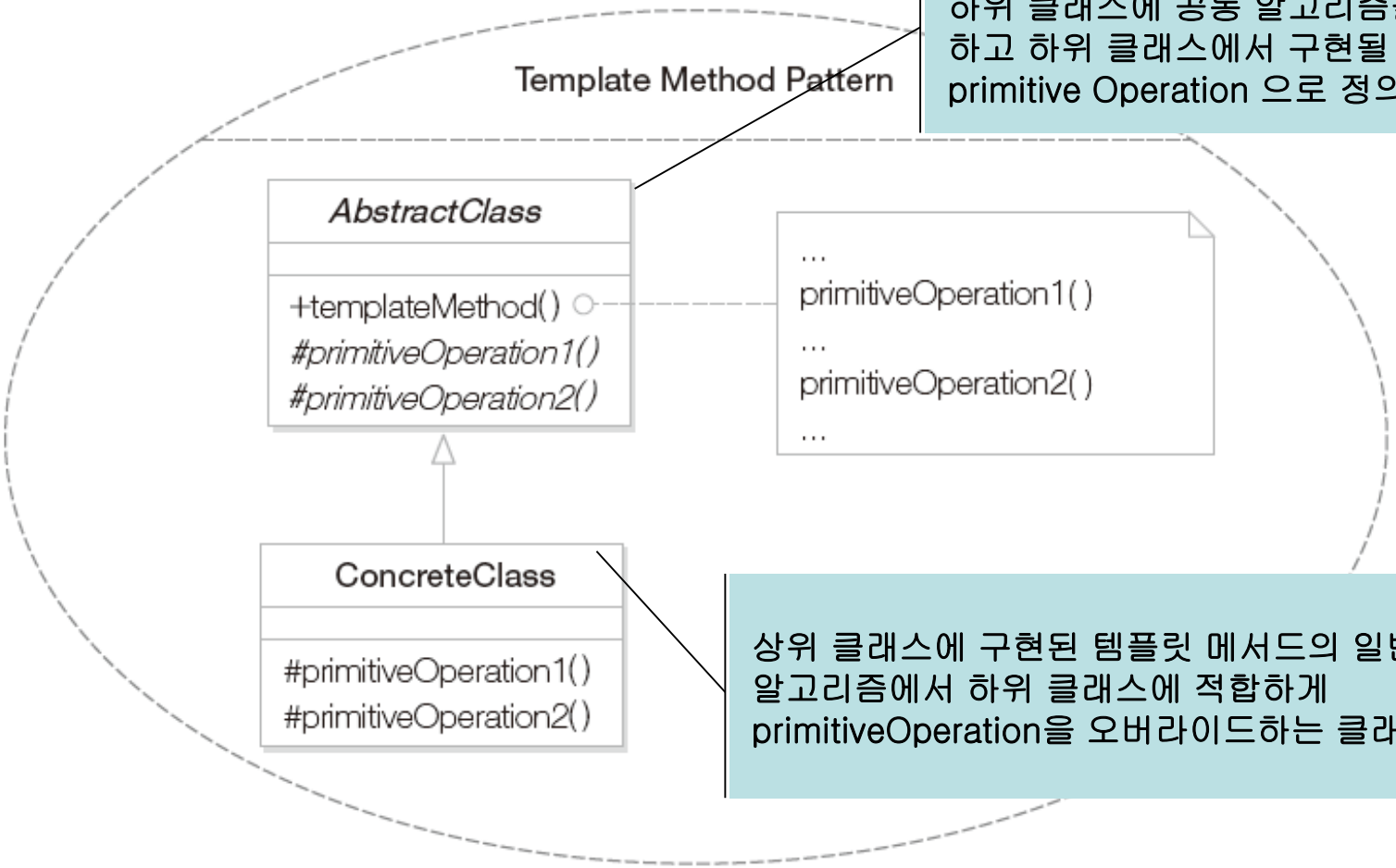
```
public void move(Direction direction) {
  MotorStatus motorStatus = getMotorStatus();
  if (motorStatus == MotorStatus.MOVING)
    return;

  DoorStatus doorStatus = door.getDoorStatus();
  if (doorStatus == DoorStatus.OPENED)
    door.close();

  moveMotor(direction);

  setMotorStatus(MotorStatus.MOVING);
}
```

템플릿 메서드

Primitive 메서드 또는 hook 메서드

11.4 템플릿 메서드 패턴

그림 11-7 템플릿 메서드 패턴의 컬레보레이션

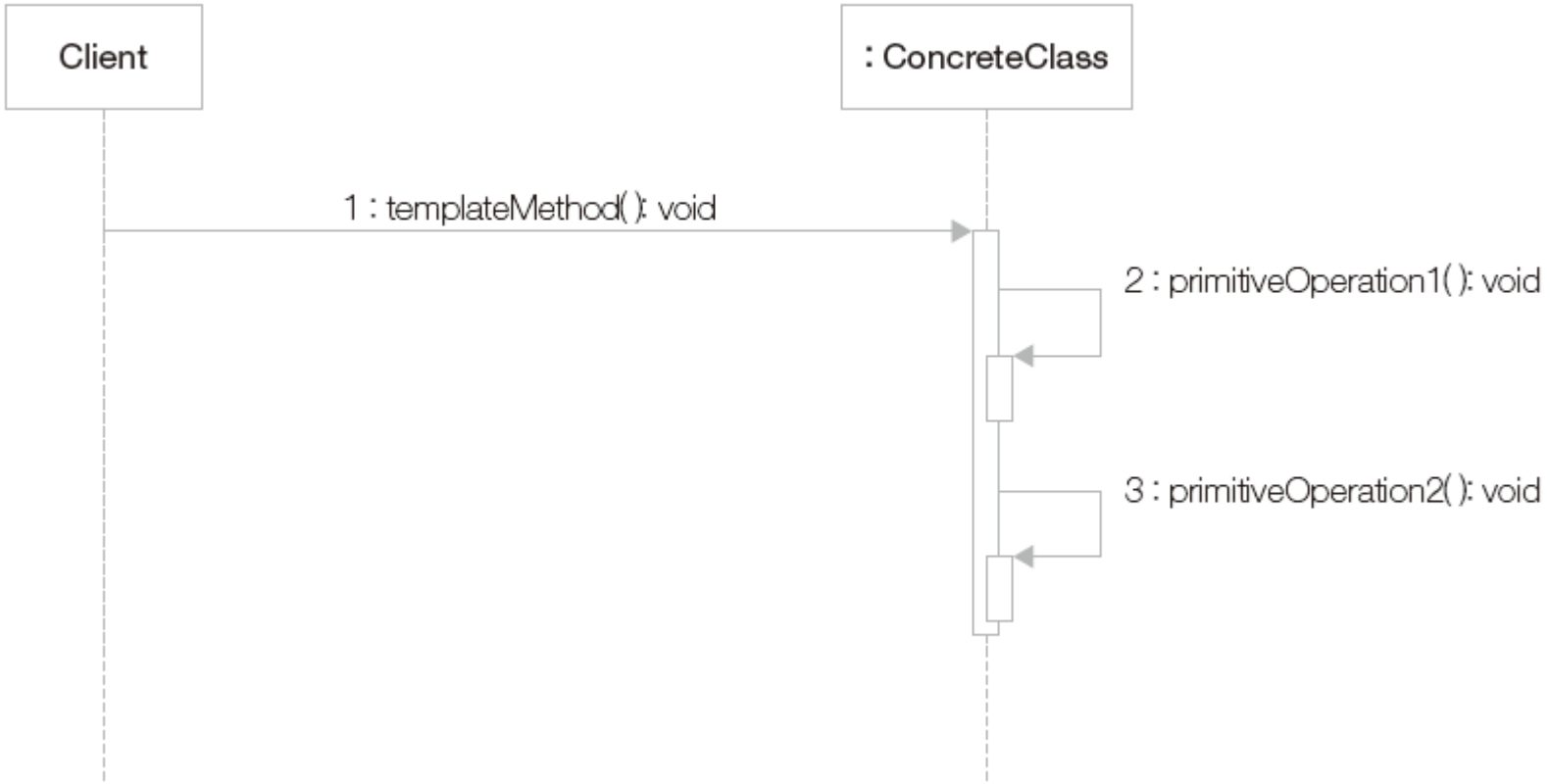


템플릿 메서드를 정의하는 클래스
하위 클래스에 공통 알고리즘을 정의하고 하위 클래스에서 구현될 기능을 primitive Operation 으로 정의

상위 클래스에 구현된 템플릿 메서드의 일반적인 알고리즘에서 하위 클래스에 적합하게 primitiveOperation을 오버라이드하는 클래스

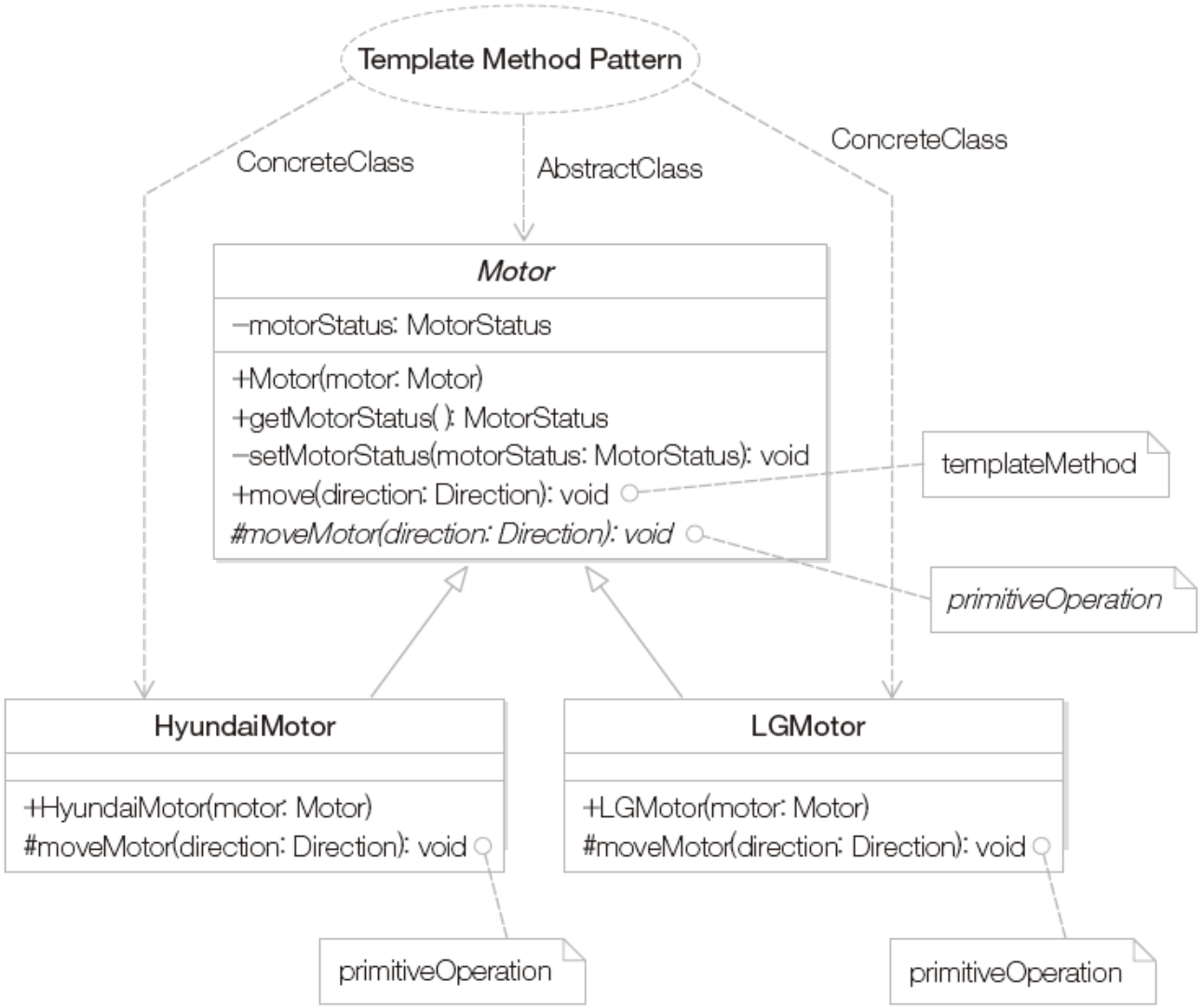
11.4 템플릿 메서드 패턴

그림 11-8 템플릿 메서드 패턴의 순차 다이어그램



템플릿 메서드 패턴의 적용

그림 11-9 템플릿 메서드 패턴을 모터 예제에 적용한 경우



다음은 Adder 클래스와 Multiplier 클래스이다, Template Method 패턴을 사용하여 프로그램을 재작성하라.

```
public class Adder {
    public int add(int n) {
        int tot = 0;
        for (int i=1; i <= n; i++) {
            tot = tot + i;
        }

        return tot;
    }
}
```

```
public class Multiplier {
    public int multi(int n) {
        int tot = 1;
        for (int i=1; i <= n; i++) {
            tot = tot * i;
        }
        return tot;
    }
}
```

참고: 스트림 프로그래밍

- 스트림: 데이터 흐름(순열)
  - Java 8이상 사용 가능
  - 배열 또는 컬렉션에 함수 여러 개를 조합해서 원하는 결과를 필터링하고 가공된 결과를 얻을 수 있음
  - 람다를 이용해서 코드의 양을 줄이고 간결하게 표현
- 스트림 프로그래밍
  - 생성하기 : 스트림 인스턴스 생성.
  - 가공하기 : filter나 map 등을 사용하여 원하는 결과를 만들어가는 중간 작업
  - 결과 만들기 : collect/reduce 등을 사용하여 최종적으로 결과를 만들어내는 작업

예제 1

- ArrayList<string> list = new ArrayList<>(Arrays.asList("Apple","Banana","Melon","Grape","Strawberry"));
- Stream 함수를 이용하여 스트림 생성후 map 함수를 이용하여 대문자로 변환
  - list.stream().map(s->s.toUpperCase());
    list.stream().map(String::toUpperCase);
- Filter 함수를 이용하여 특정 요소만 걸러냄
  - list.stream().filter(t->t.length()>5)
- Reduce 함수를 이용하여 결과만들기
  - list.stream().filter(t->t.length()>5).reduce(“” , (a, b)->a+” “+ b);

실습 2

❖ 다음 **add** 함수를 스트림을 이용하여 다시 재작성하시오.

❖
```
public class Adder {
    public int add(int n) {
        int tot = 0;
        for (int i=1; i <= n; i++) {
            tot = tot + i;
        }

        return tot;
    }
}
```

실습 3

- 다음 **multi** 함수를 스트림을 이용하여 다시 재작성하시오. 또한 **BigInteger**를 이용하여 **overflow**가 나지 않도록 하시오.

```
public class Multiplier {
    public int multi(int n) {
        int tot = 1;
        for (int i=1; i <= n; i++) {
            tot = tot * i;
        }
        return tot;
    }
}
```