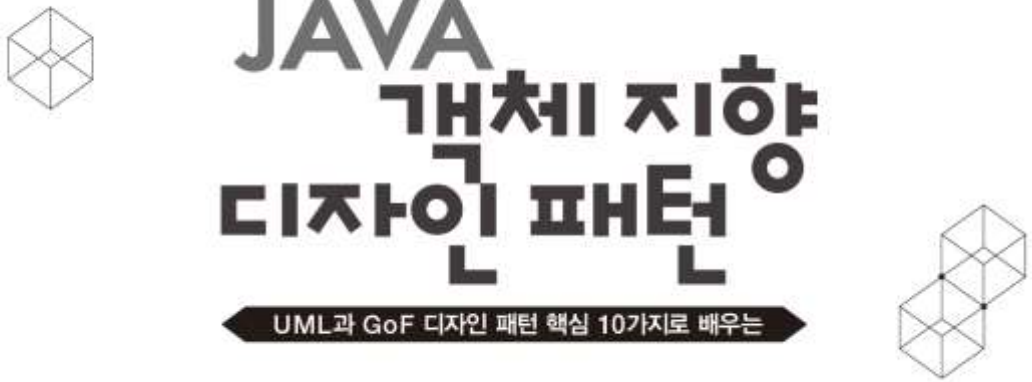
11. 템플릿 메서드 패턴



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

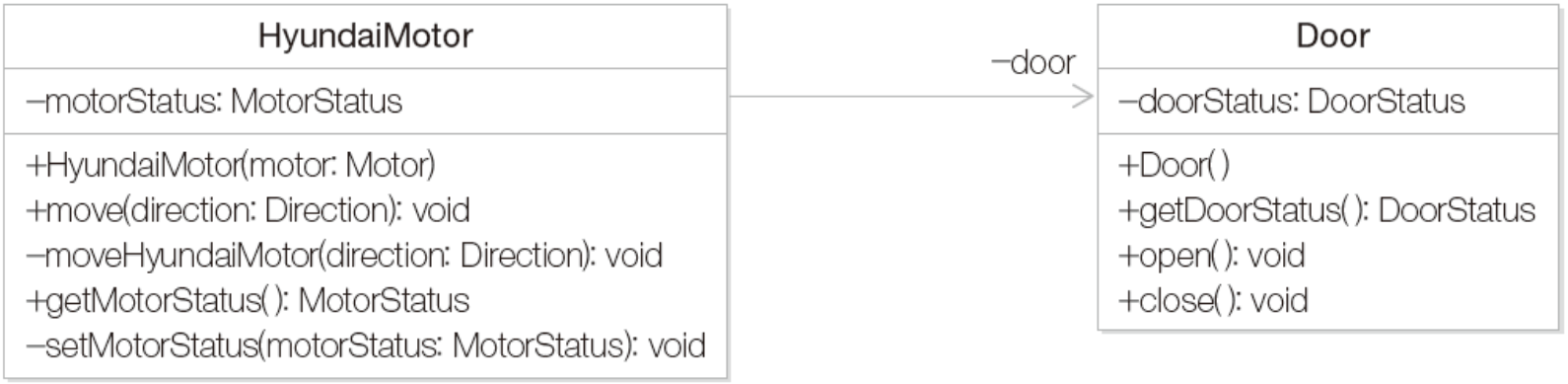
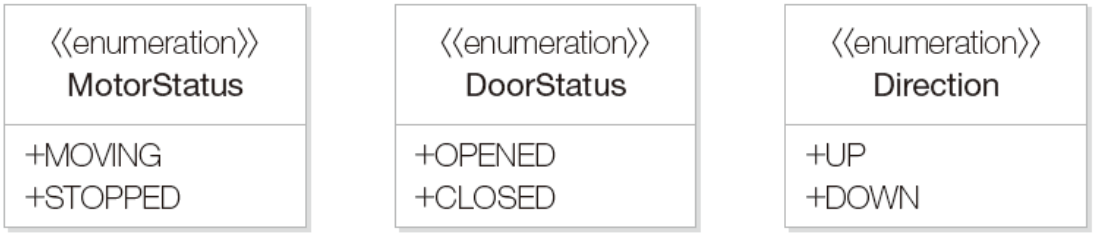
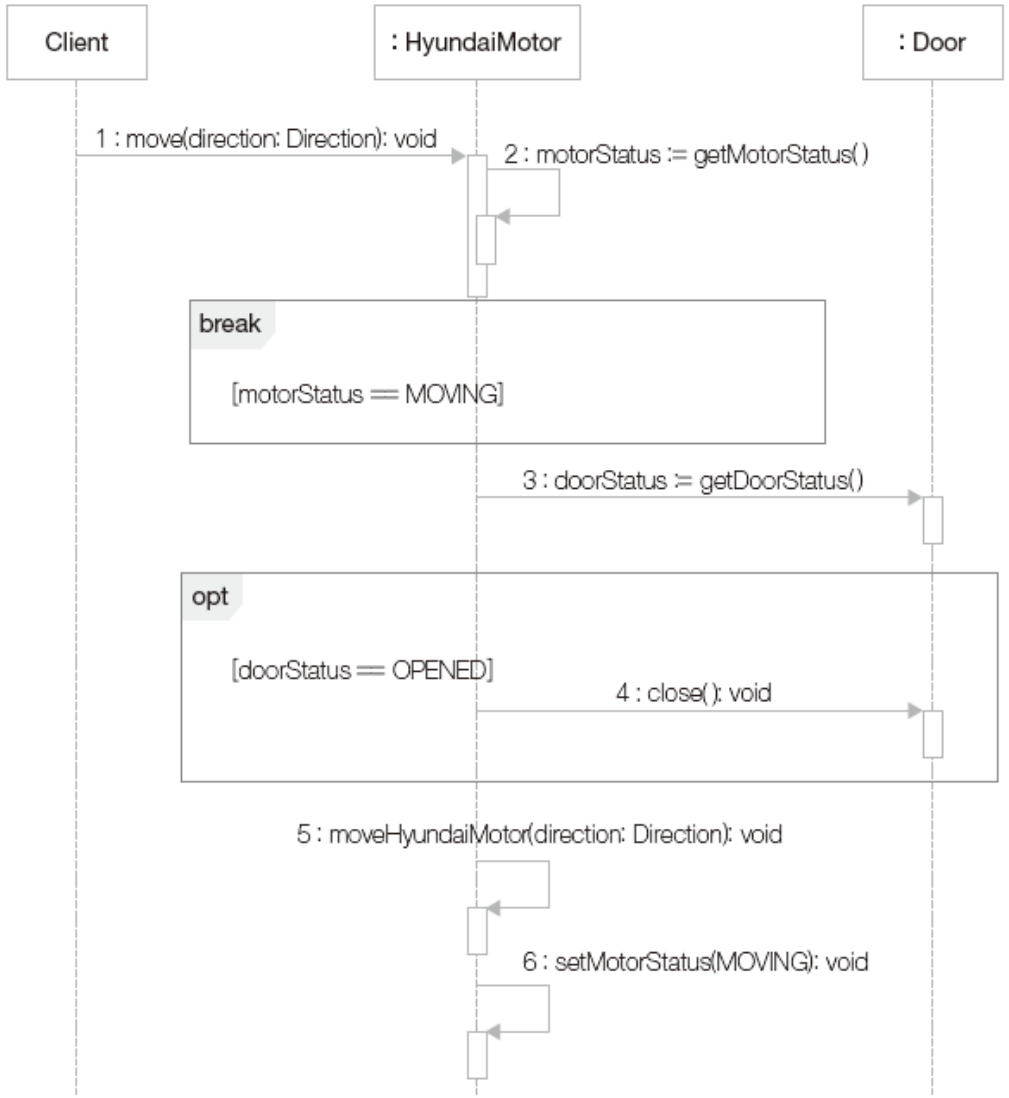


그림 11-2 Enumeration type 인 MotorStatus, DoorStatus, Direction의 설계



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 Direction { UP, DOWN }
```

```
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?

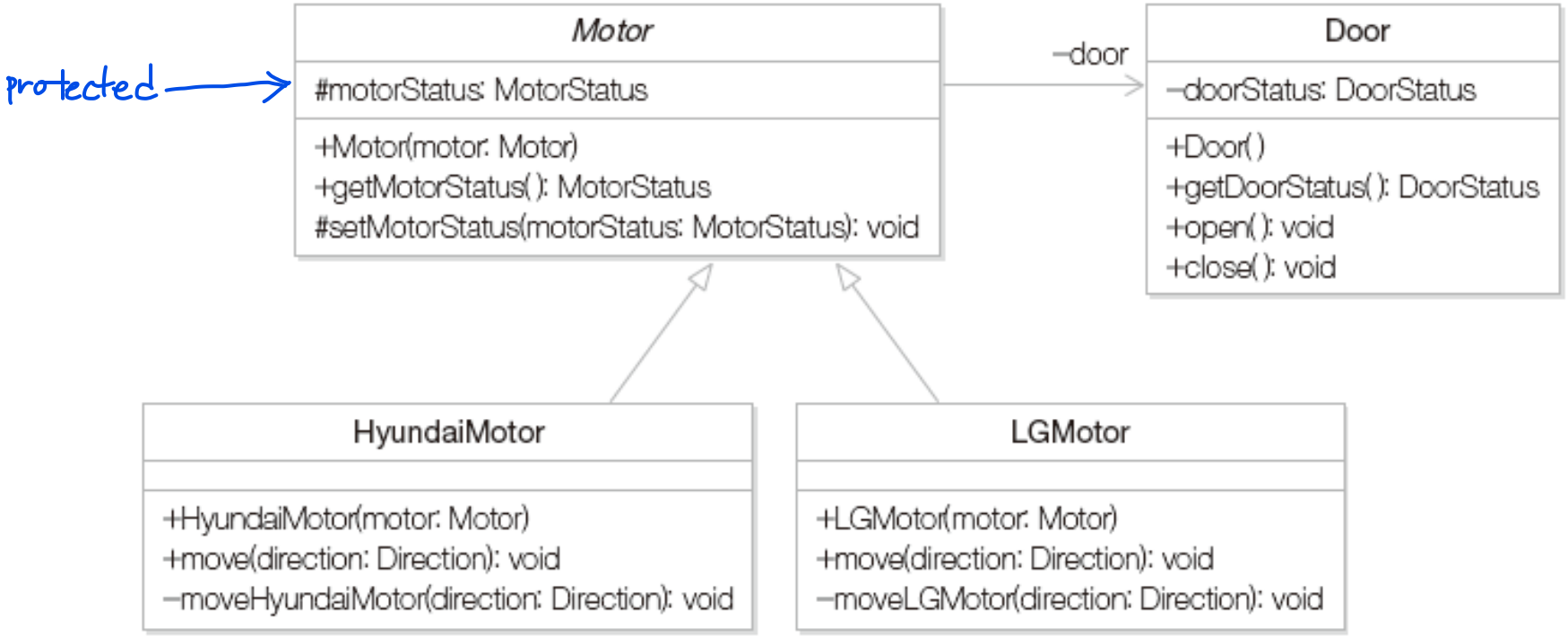
LGMotor 클래스

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
   private Door door ;
   protected 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

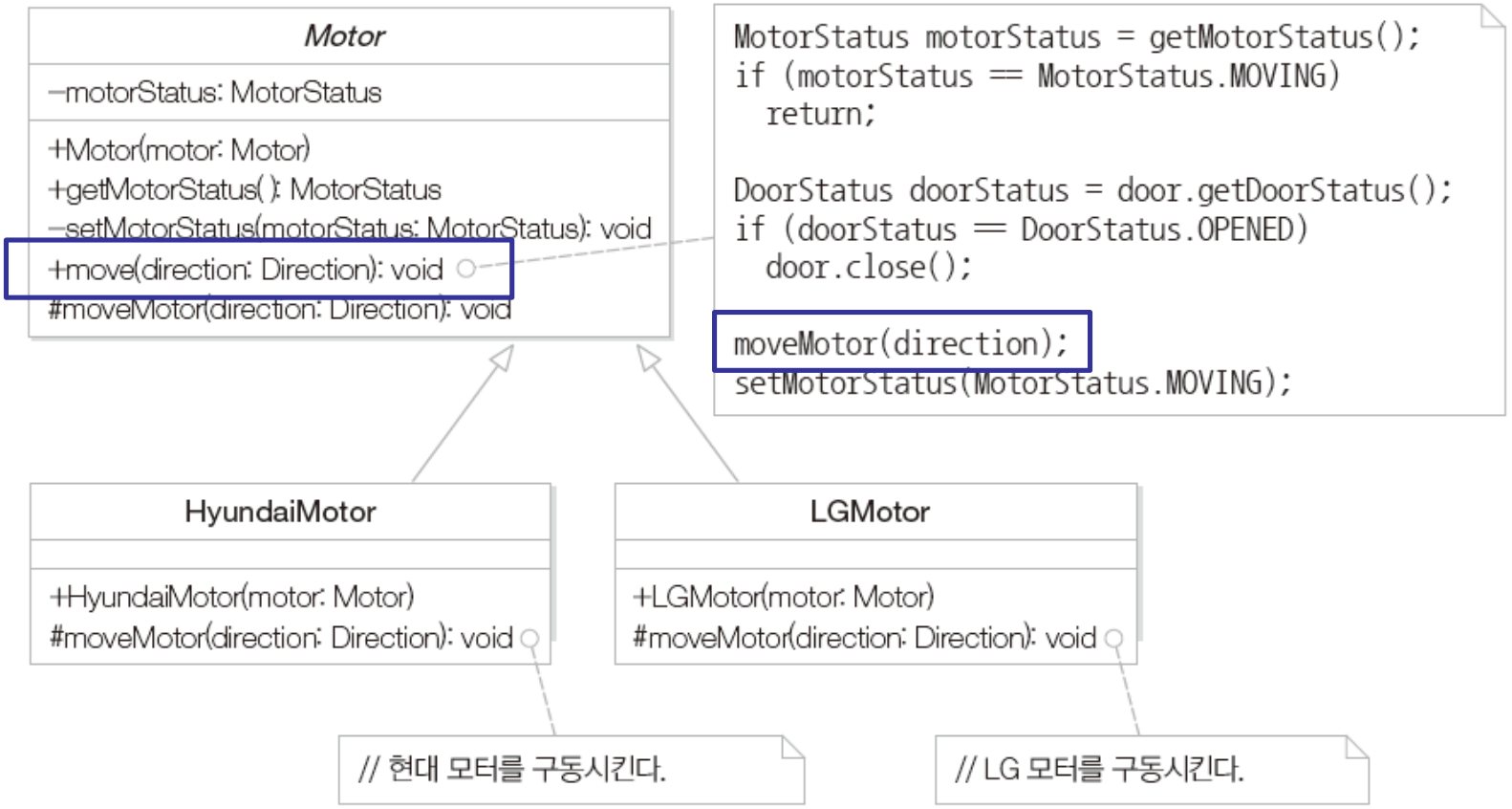
```
public void move(Direction direction) {
   MotorStatus motorStatus = getMotorStatus() ;
   if (  motorStatus == MotorStatus.MOVING )
      return ;

   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 클래스에서 구현할 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

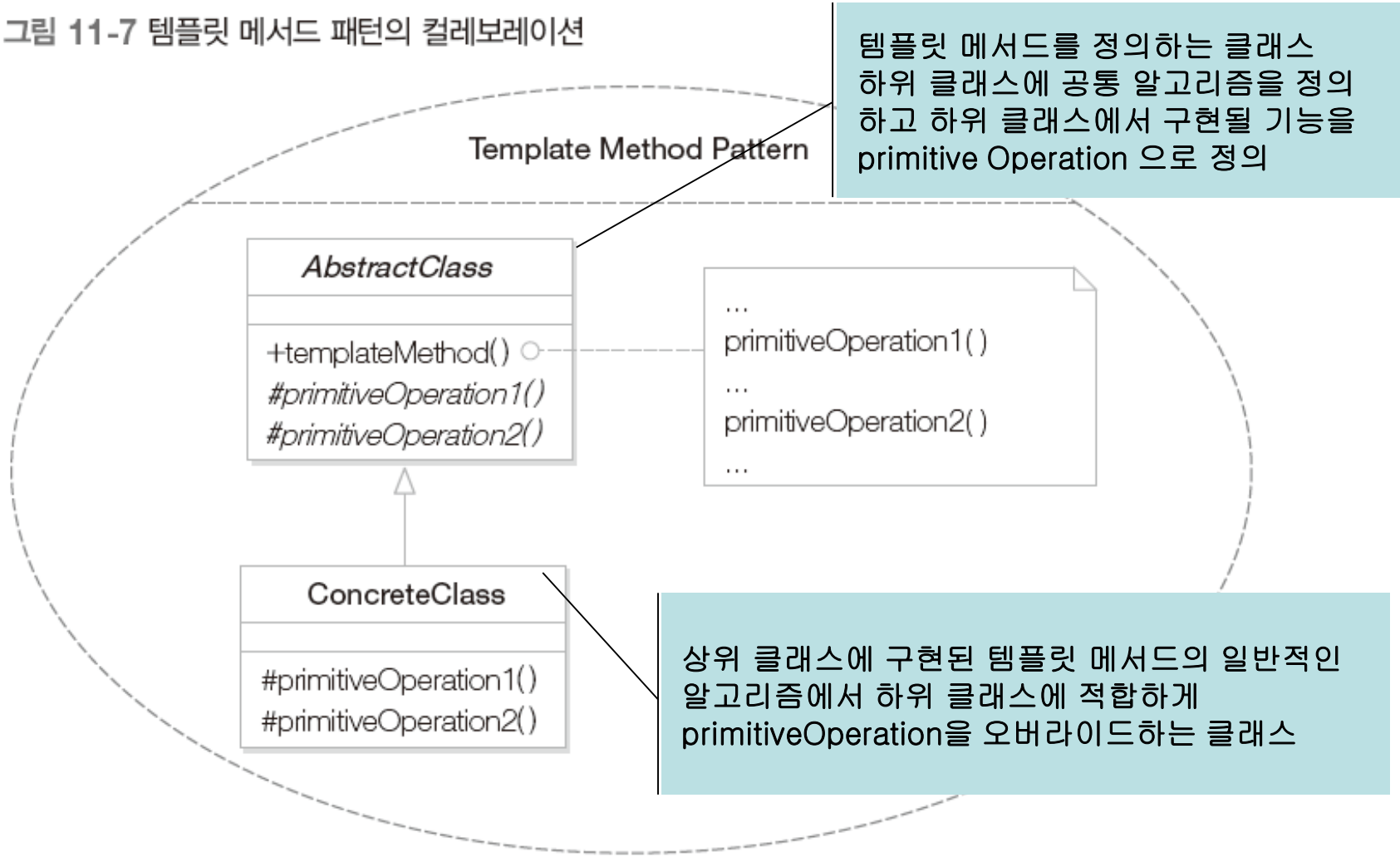
Primitive 메서드 또는 hook 메서드

[참고] hook 메서드란?

- 후크(Hook)
  - 추상클래스에서 구현되는 메서드긴 하지만, 기본적인 내용만 구현되어 있거나 아무 코드도 들어있지 않은 메서드
  - 서브클래스(자식클래스)에서 오버라이드하여 사용

11.4 템플릿 메서드 패턴

그림 11-7 템플릿 메서드 패턴의 컬레보레이션

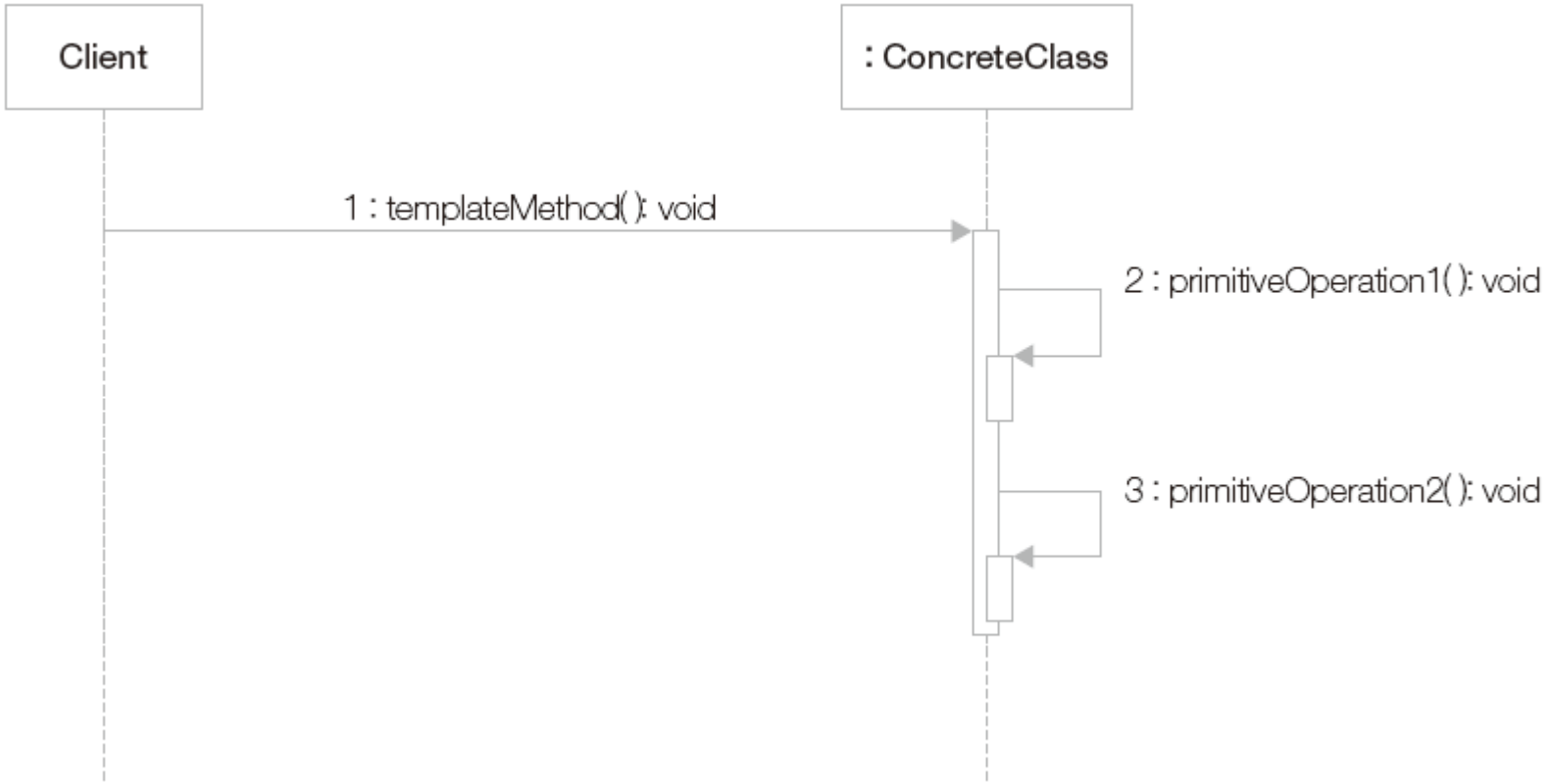


템플릿 메서드를 정의하는 클래스
하위 클래스에 공통 알고리즘을 정의하고 하위 클래스에서 구현될 기능을 primitive Operation 으로 정의

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

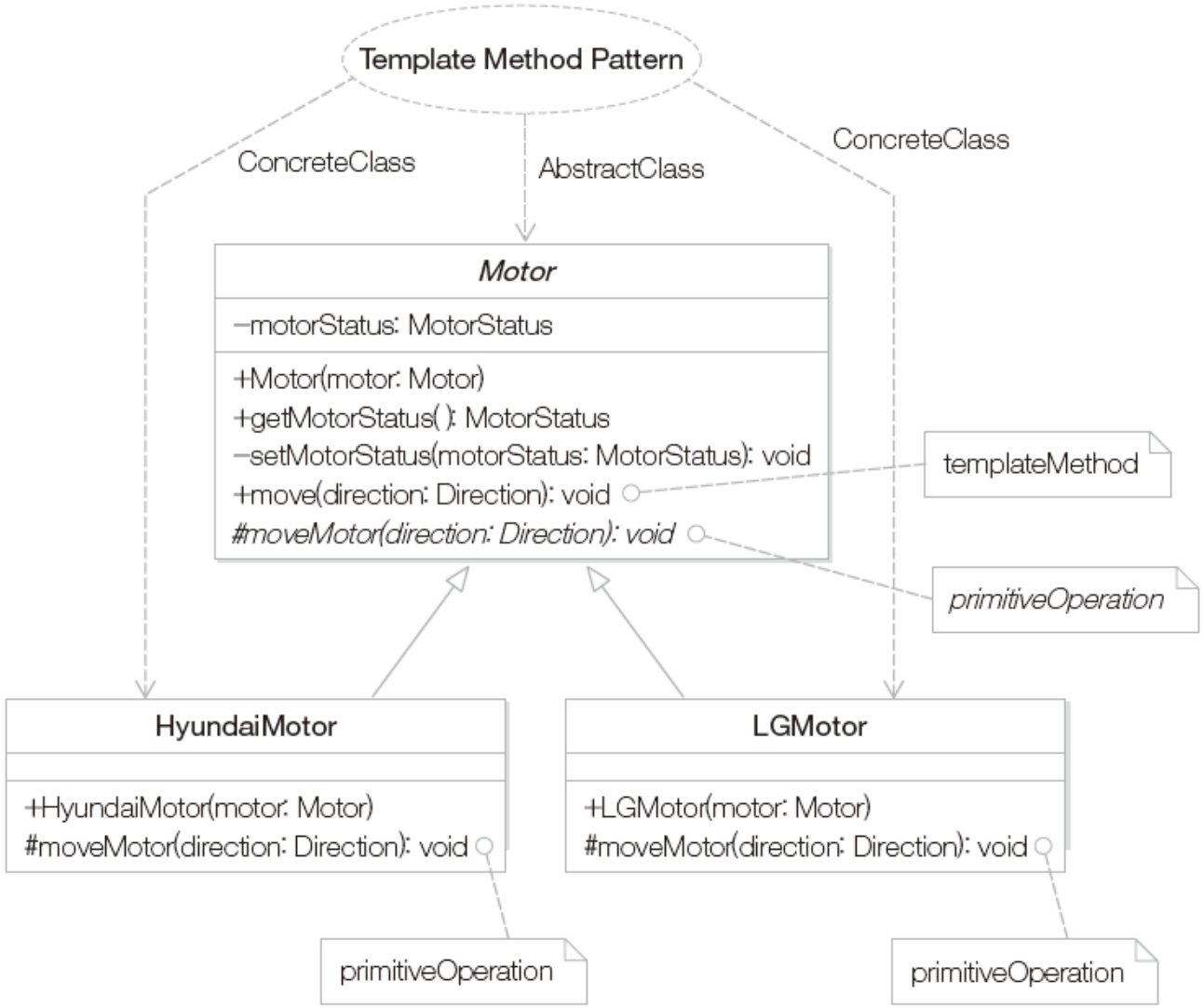
11.4 템플릿 메서드 패턴

그림 11-8 템플릿 메서드 패턴의 순차 다이어그램



템플릿 메서드 패턴의 적용

그림 11-9 템플릿 메서드 패턴을 모터 예제에 적용한 경우



또 다른 예제

- 뜨거운 음료 만들기
  - 다음과 같은 과정을 모든 음료 제조에 포함
    - 물 끓이기
    - 컵에 붓기
    - 음료 내리기
    - 기호에 따라 설탕과 같은 첨가물을 추가하기

*출처 : 얄팍한 코딩사전, 템플릿 매소드(Template Method) 패턴, https://www.youtube.com/watch?v=JgOtZfly6DE&t=139s

음료 추상 클래스

찻잎 우릴 때
따라해야 하는 순서를 강제함

```
abstract class Beverage{
 final void prepareRecipe(){
   boilWater();
   brew();
   pourInCup();
   addCondiments();
 }
void boilWater(){
 System.out.println("물이 끓습니다.");
 }
void pourInCup(){
 System.out.println("컵에 붓습니다.");
 }
 abstract void brew();
 abstract void addCondiments();
}
```

차, 커피 클래스

```
class Tea extends Beverage{
  void brew() {
    System.out.println("Stepping the tea");
  }
  void addCondiments(){
    System.out.println("Adding lemon");
  }
}
```

```
class Coffee extends Beverage{
  void brew() {
    System.out.println("Dripping coffee through filter");
  }
  void addCondiments() {
    System.out.println("Adding sugar and milk");
  }
}
```

클라이언트 클래스

```
public class Client {
  public static void main(String[] args) {
    Beverage tea = new Tea();
    Beverage coffee = new Coffee();
    System.out.println("₩nMaking tea...");
    tea.prepareRecipe();
    System.out.println("₩nMaking coffee...");
    coffee.prepareRecipe();
  }
}
```

예제 (3)

- 다양한 형식의 파일을 읽어들여 처리한 후 다른 형태로 저장하는 프로그램 생성하기
  - 파일을 로드
  - 올바른 형식인지 확인
  - 정해진 프로세스로 데이터를 처리
  - 새로운 파일로 저장

*출처 : 얄팍한 코딩사전, 템플릿 매소드(Template Method) 패턴, https://www.youtube.com/watch?v=JgOtZfly6DE&t=139s

조건문을 사용하는 템플릿 메소드

```
abstract class DataProcessor {
  public final void process(String data) {
     loadData(data);
     if (isValidData(data)) {
       processData(data);
       saveData(data);
     } else {
       System.out.println("Data is invalid, processing aborted.");
     }
   }

  protected abstract void loadData(String data);
  protected abstract Boolean isValidData(String data);
  protected abstract void processData(String data);
  protected abstract void saveData(String data);
}
```

문서 종류별 클래스 구현

```
class CSVDataProcessor extends DataProcessor {
  @Override
  protected void loadData(String data) {
    System.out.println("Loading data from CSV file: " + data);
  }

  @Override
  protected Boolean isValidData(String data) {
     return data != null && data.contains("CSV");
  }

  @Override
  protected void processData(String data) {
    System.out.println("Processing CSV data");
  }
  @Override
  protected void saveData(String data) {
    System.out.println("Saving CSV data to database");
  }
}
```

문서 종류별 클래스 구현

```
class JSONDataProcessor extends DataProcessor {
  @Override
  protected void loadData(String data) {
    System.out.println("Loading data from JSON file: " + data);
  }

  @Override
  protected Boolean isValidData(String data) {
     return data != null && data.contains("JSON");
  }

  @Override
  protected void processData(String data) {
    System.out.println("Processing JSON data");
  }

  @Override
  protected void saveData(String data) {
    System.out.println("Saving JSON data to database");
  }
}
```

클라이언트 코드 구현

```
public class Client {
  public static void main(String[] args) {
    DataProcessor csvProcessor = new CSVDataProcessor();
      csvProcessor.process("CSV data");

      System.out.println();

      DataProcessor jsonProcessor = new JSONDataProcessor();
      jsonProcessor.process("XML data");
  }
}
```

템플릿 메소드를 사용할 때

- 구현할 알고리즘의 구조는 고정되어 있지만, 세부 과정 각각은 변경 혹은 확장 가능할 때 사용 가능

템플릿 메서드 패턴의 장점

- 알고리즘의 일관성 유지
  - 상위 클래스에서 알고리즘의 전체 흐름을 정의하고, 하위 클래스에서 일부 단계를 구현하기 때문에 알고리즘의 일관성이 보장
- 중복 코드 제거
  - 알고리즘의 공통 부분을 상위 클래스에 두고, 변하는 부분만 하위 클래스에서 구현하게 하여, 중복 코드를 줄이고 재사용성을 높임
- 하위 클래스에서 알고리즘의 세부 구현을 수정 가능
  - 하위 클래스는 템플릿 메서드를 상속하여 일부 메서드를 오버라이드함으로써 알고리즘의 세부 단계를 자신에게 맞게 수정가능
- 알고리즘 변경의 용이성
  - 알고리즘의 흐름을 상위 클래스에서 정의하므로, 알고리즘의 큰 틀을 변경하려면 상위 클래스만 수정하면 됨
- 가독성 및 확장성
  - 템플릿 메서드 패턴은 알고리즘의 단계들을 명확하게 구분하고, 각 단계를 하위 클래스에서 정의할 수 있기 때문에 코드가 더 직관적이고 가독성이 좋음
  - 또한, 새로운 하위 클래스를 추가해도 기존 코드에 큰 영향을 주지 않기 때문에 확장성이 뛰어남

템플릿 메서드 패턴의 단점

- 상위 클래스의 변경이 모든 하위 클래스에 영향을 미침
- 상위 클래스의 복잡성 증가
  - 알고리즘의 공통된 골격을 상위 클래스에 두면 상위 클래스가 매우 복잡해질 수 있음.
  - 특히, 상위 클래스가 너무 많은 하위 클래스를 지원해야 할 경우, 템플릿 메서드가 복잡해져 코드의 유지보수가 어려워질 수 있음
- 하위 클래스가 너무 많아지면 관리하기 어려움
- 기능 확장이 어려울 수 있음
  - 템플릿 메서드는 알고리즘의 구조를 정의하는 데 중점을 두기 때문에, 기능을 확장하거나 변경하는 데 제약이 있는 경우도 존재
- 하위 클래스가 너무 많은 책임을 맡을 수 있음
  - 템플릿 메서드에서 세부 구현을 하위 클래스에 위임하므로, 하위 클래스가 많은 책임을 떠맡을 수 있음
  - 하위 클래스가 지나치게 복잡해질 수 있고, 한 클래스가 너무 많은 작업을 처리하게 되어 응집력이 떨어질 수 있습니다.