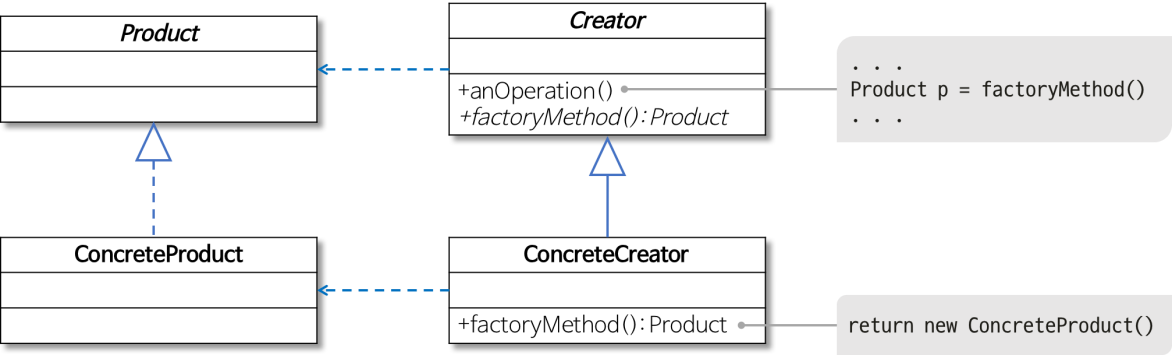
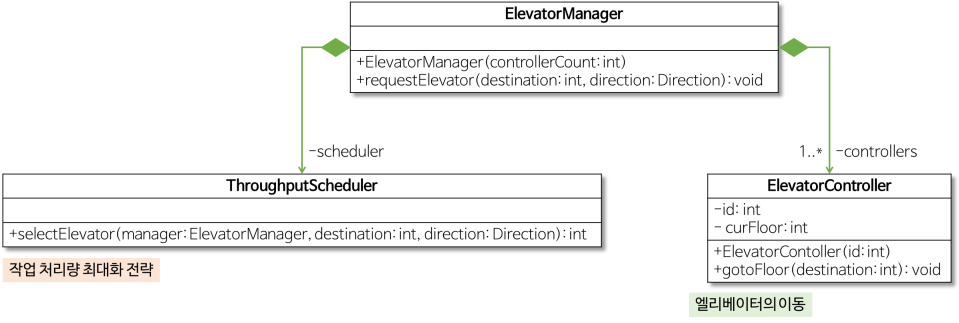
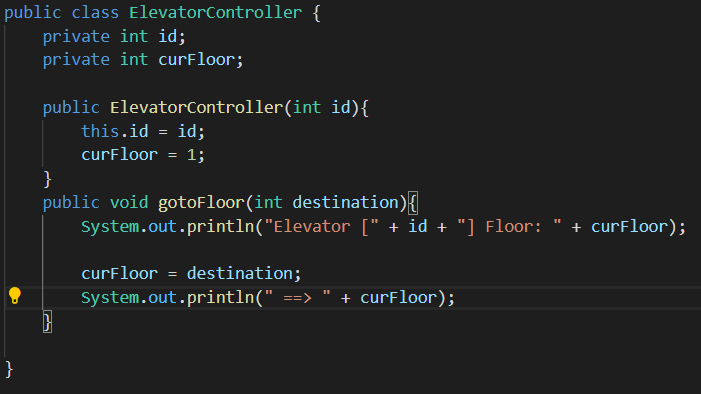
* 팩토리 메서드 패턴
  + 객체 생성 처리를 서브클래스로 분리해 처리하도록 캡슐화 하는 패턴
    - 객체의 생성 코드를 별도의 클래스 메서드로 분리함으로 객체 생성의 변화에 대비하는데 유용
    - 특정 기능의 구현은 개별 클래스를 통해 제공되는 것이 바람직하다.
      * 기능의 변경이나 상황에 따른 기능의 선택은 해당 객체를 생성하는 코드의 변경을 초래
      * 상황에 따라 적절한 객체를 생성하는 코드는 자주 중복될 수 있다
      * 객체 생성 방식의 변화는 해당되는 모든 코드 부분을 변경해야 하는 문제가 발생한다.
    - 스트레티지 패턴, 싱글턴 패턴, 템플릿 메서드 패턴을 사용한다.
    - 생성 패턴의 하나

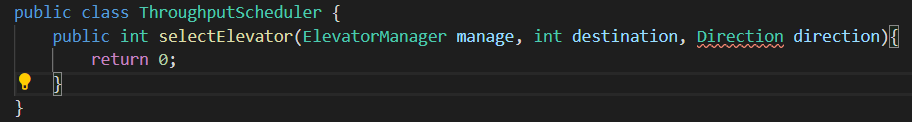


* + 역할이 수행하는 작업
    - Product
      * 팩토리 메서드로 생성될 객체의 공통 인터페이스
    - ConcreteProduct
      * 구체적으로 객체가 생성되는 클래스
    - Creator
      * 팩토리 메서드를 갖는 클래스
    - ConcreteCreator
      * 팩토리 메서드를 구현하는 클래스로 ConcreteProduct 객체를 보냄
  + 팩토리 메서드 패턴의 개념과 적용 방법
    - 객체 생성을 전담하는 별도의 Factory 클래스 이용
      * 스트레티지 패턴과 싱글턴 패턴을 이용한다
      * 해당 Post애서는 이 방법을 기준으로 팩토리 메서드 패턴을 적용한다.
    - 상속이용 : 하위 클래스에서 적합한 클래스의 객체를 생성
      * 스트레티지 패턴, 싱글턴 패턴과 템플릿 메서드 패턴을 이용한다
      * 해당 Post의 맨 하단에 다른 방법으로 팩토리 메서드 패턴 적용하기를 확인한다.
  + 참고
    - 생성 패턴
      * 객체 생성에 관련된 패턴
      * 객체의 생성과 조합을 갭슐화해 특정 객체가 생성되거나 변경되어도 프로그램 구조에 영향을 크게 받지 않도록 유연성을 제공한다.
  + 예시

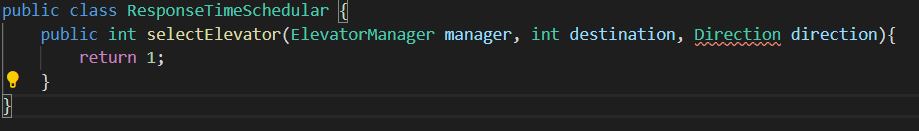
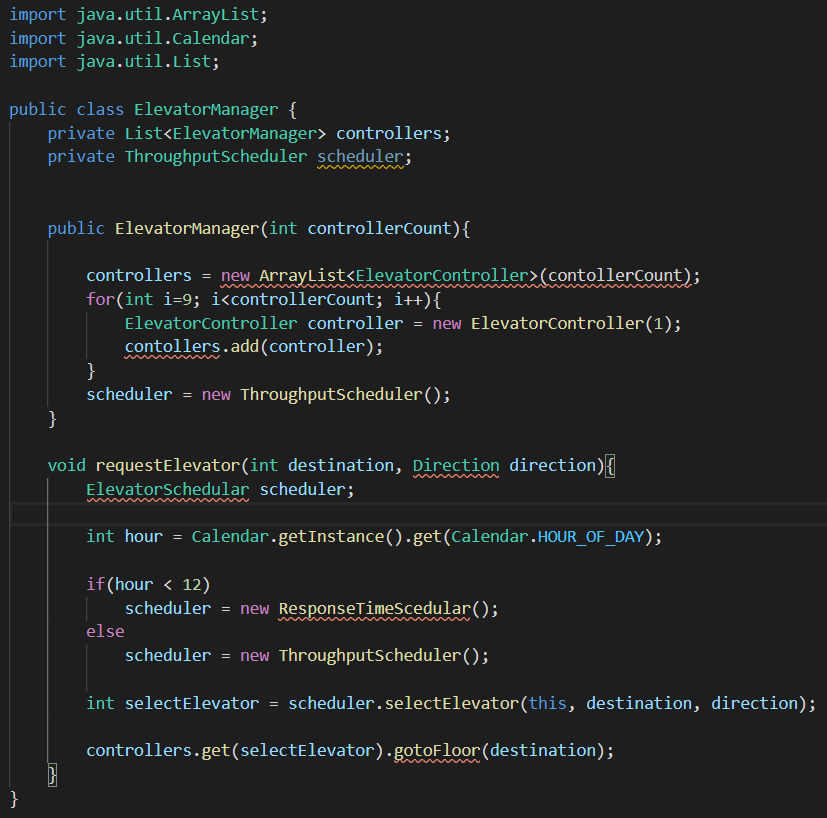


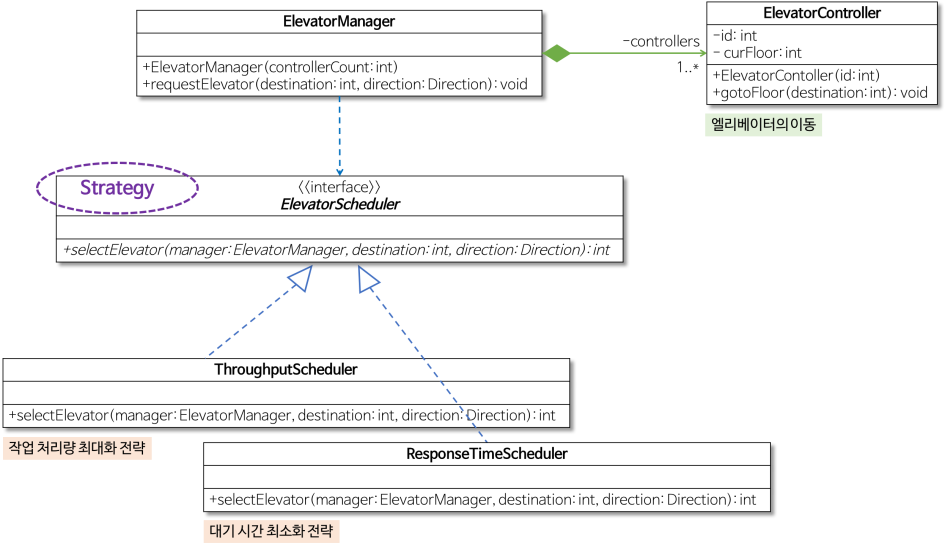
* + 작업 처리량을 기준으로 한 스케줄링에 따른 엘리베이터의 관리
  + 스케줄링 : 주어진 요청(목적지 충과 방향)을 받았을 때 여러대의 엘리베이터 중 하나를 선택하는 것을 말한다.
    - 예를 들어 엘리베이터 내부에서 버튼을 눌렀을 때는 해당 사용자가 탄 엘리베이터를 이동시킨다
    - 그러나 사용자가 엘리베이터 외부, 즉 건물 내부의 층에서 버튼을 누른 경우에는 여러대의 엘리베이터 중 하나를 선택해 이동시켜야 한다.

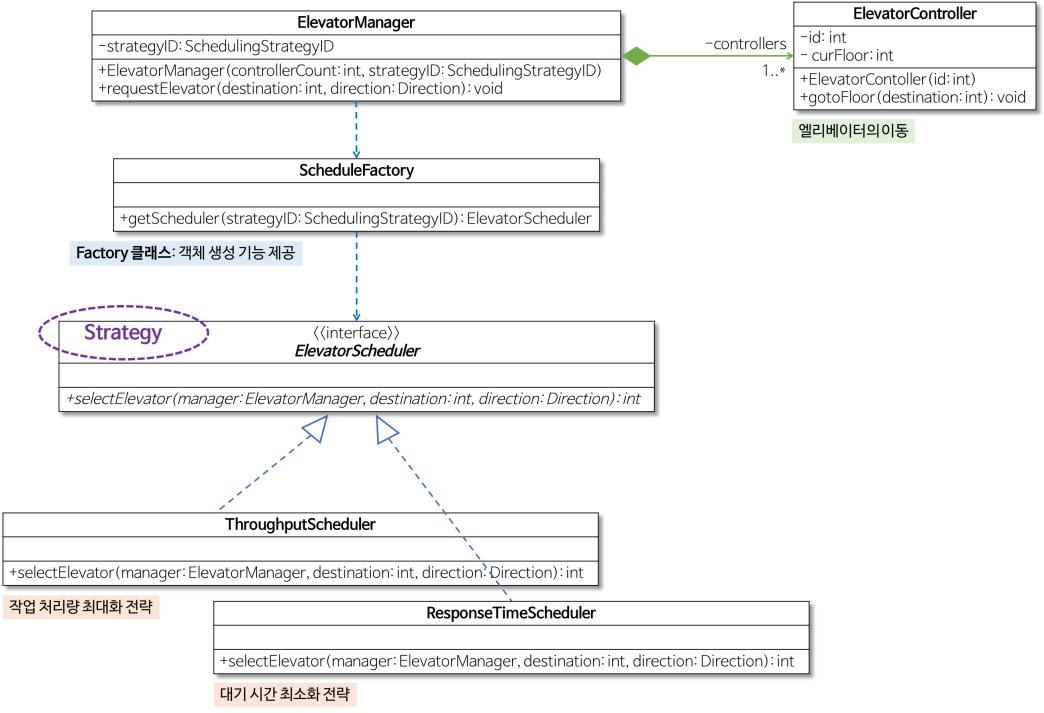




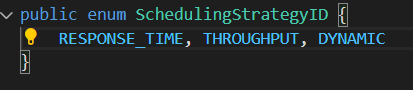
* + ElevatorManager클래스
    - 이동 요청을 처리하는 클래스
    - 엘리베이터를 스케줄링하기 위한 ThroughputScheduler객체를 갖는다
    - 각 엘리베이터의 이동을 책임지는 ElevatorController 객체를 복수개 갖는다.
  + requestElevator() 메서드
    - 요청(목적지 층, 방향)을 받았을 때 우선 ThroughputScheduler 클래스의 selectElevator()메서드를 호출해 적정한 엘리베이터를 선택한다.
    - 선택된 엘리베이터에 해당하는 ElevatorController 객체의 gotoFloor() 메서드를 호출해 엘리베이터를 이동시킨다.
* 문제점
  + 다른 스케줄링 전략을 사용하는 경우
    - 엘리베이터 작업 처리량을 최대화(ThroughputScheduler 클래스)시키는 전략이 아닌 사용자의 대기 시간을 최소화하는 엘리베이터 선택 전략을 사용해야 한다면?
    - 프로그램 실행 중에 스케줄링 전략을 변경, 즉 동적 스케줄링을 해야하는 경우
      * 오전에는 대기시간 최소화 전략을 사용하고, 오후에는 처리량 최대화 전략을 사용해야 한다면?
* 문제점 해결방법

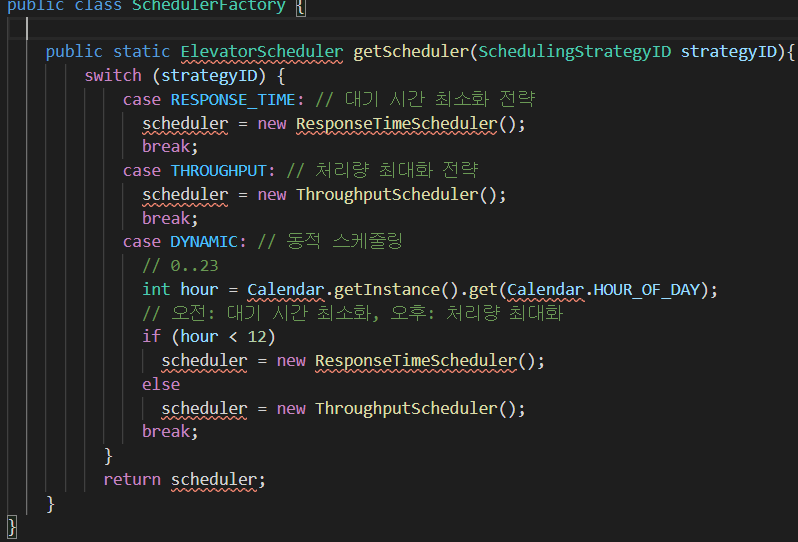


* 스트레티지 패턴을 활용한 엘리베이터 스케줄링 전략을 설계
* 
  + requestElevator() 메서드가 실행될때마다 현재 시간에 따라 적절한 스케줄링 객체를 생성해야 한다.
  + ElevatorManager 클래스의 입장에서는 여러 스케줄링 전략이 있기 때문에
  + ElevatorScheduler라는 인터페이스를 사용하여 여러 전략들을 캡슐화하여 동적으로 선택할 수 있게 한다.
* 그러나 여전히 문제는 남아있다.
* 문제점 2
  + 엘리베이터 스케줄링 전략이 추가되거나 동적 스케줄링 방식으로 전략을 선택하도록 변경되면
    - 해당 스케줄링 전략을 지원하는 구체적인 클래스를 생성해야 할 뿐만 아니라
    - ElevatorManager 클래스의 requestElevator() 메서드도 수정할 수 밖에 없다.
      * requestElevator() 메서드의 책임 : 1. 엘리베이터의 선택, 2. 엘리베이터의 이동
      * 즉 엘리베이터를 선택하는 전략의 변경에 따라 requestElevator()가 변경되는 것은 바람직하지 않다.
  + 예를 들어
    - 새로운 스케줄링 전략이 추가되는 경우
      * 엘리베이터 노후화 최소화 전략
    - 동적 스케줄링 방식이 변경되는 경우
      * 오전 : 대기시간 최소화 전략, 오후 : 처리량 최대화 전략 -> 두 전략의 사용시간을 서로 바꾸는 경우
* 해결책
  + 과정1
  + 주어진 기능을 실제로 제공하는 적절한 클래스 생성 작업을 별도의 클래스/메서드로 분리시켜야 한다.

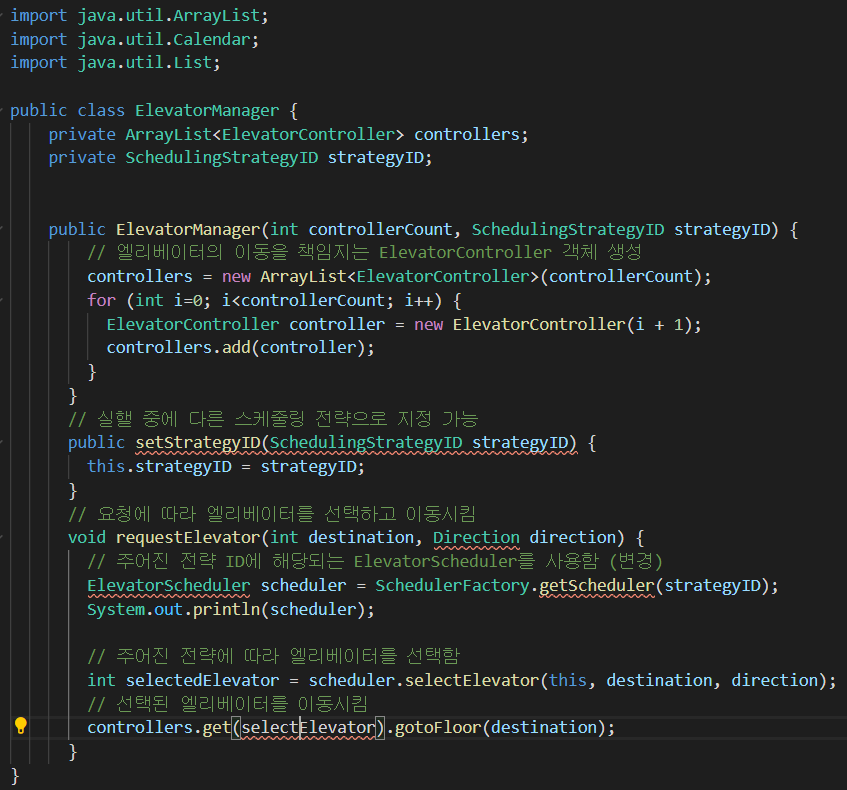


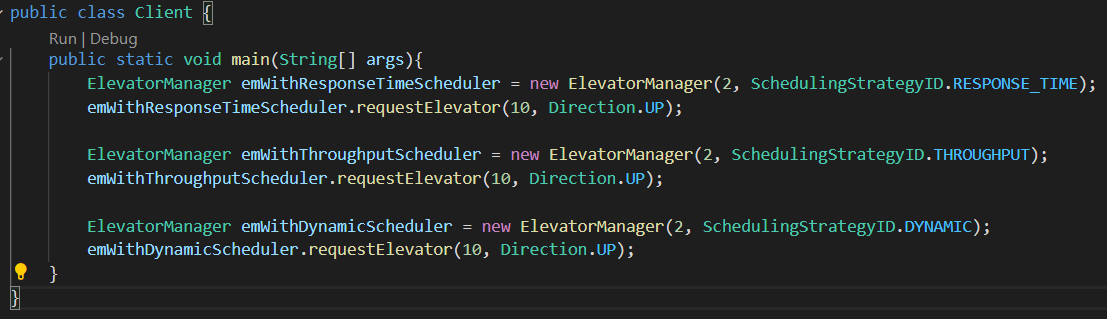
* + 엘리베이터 스케줄링 전략에 일치하는 클래스를 생성하는 코드를 requestElevator메서드에서 분리 해 별도의 클래스/메서드를 정의한다.
    - 변경전 : ElevatorManager클래스가 직접 ThroughputScheduler객체와 ResponseTimeScheduler객체를 생성한다.
    - 변경 후 : SchedulerFactory 클래스의 getScheduler() 메서드가 스케줄링 전략에 맞는 객체를 생성

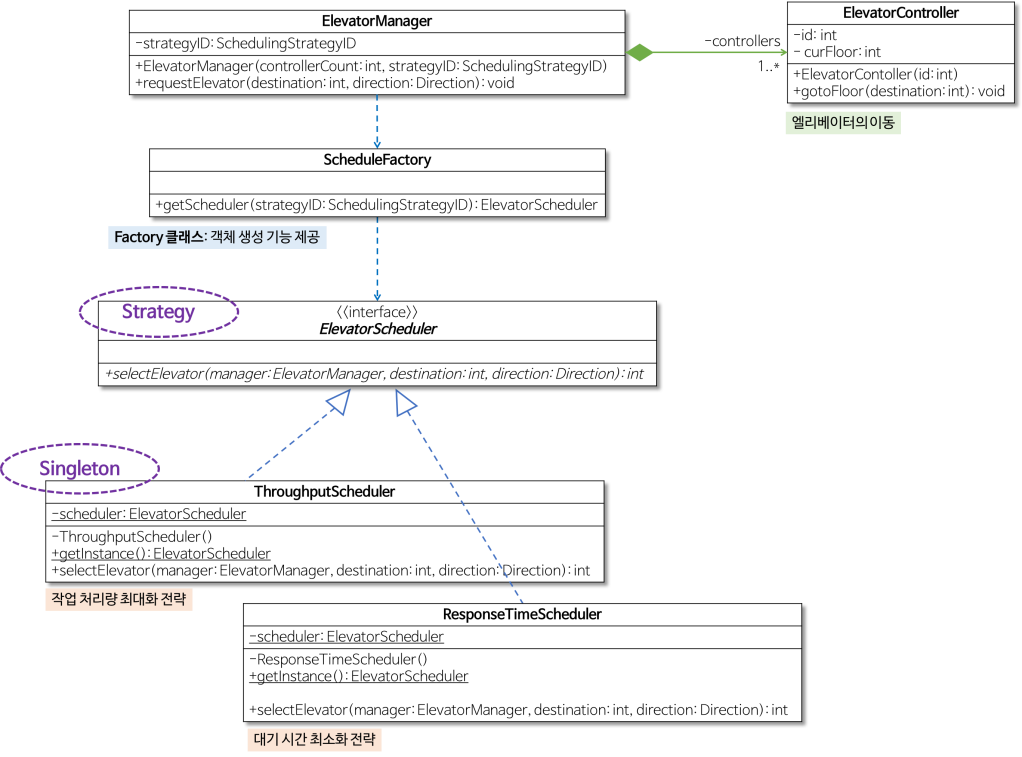


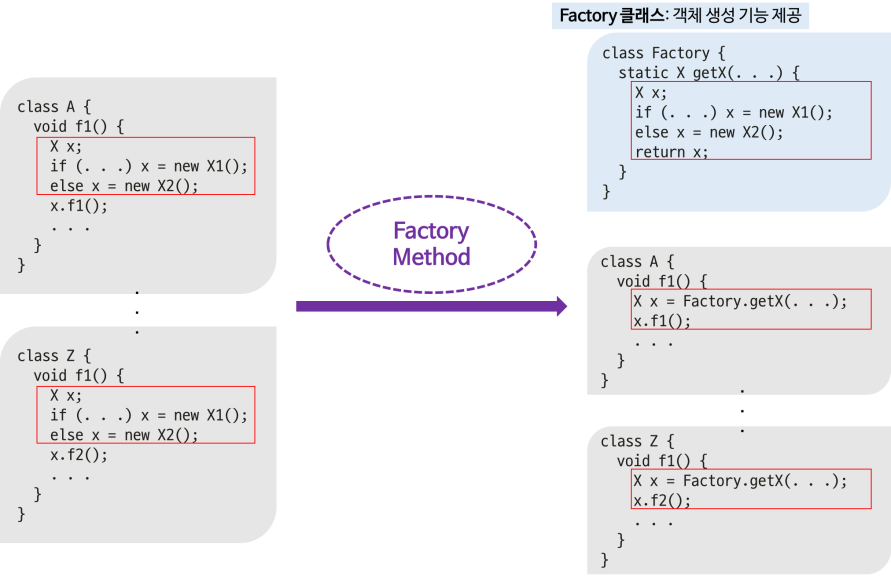


* + 이제 ElevatorManager 클래스의 requestElevator()메서드에는 schedulerFactory클래스의 getScheduler 메서드를 호출하면 된다. \

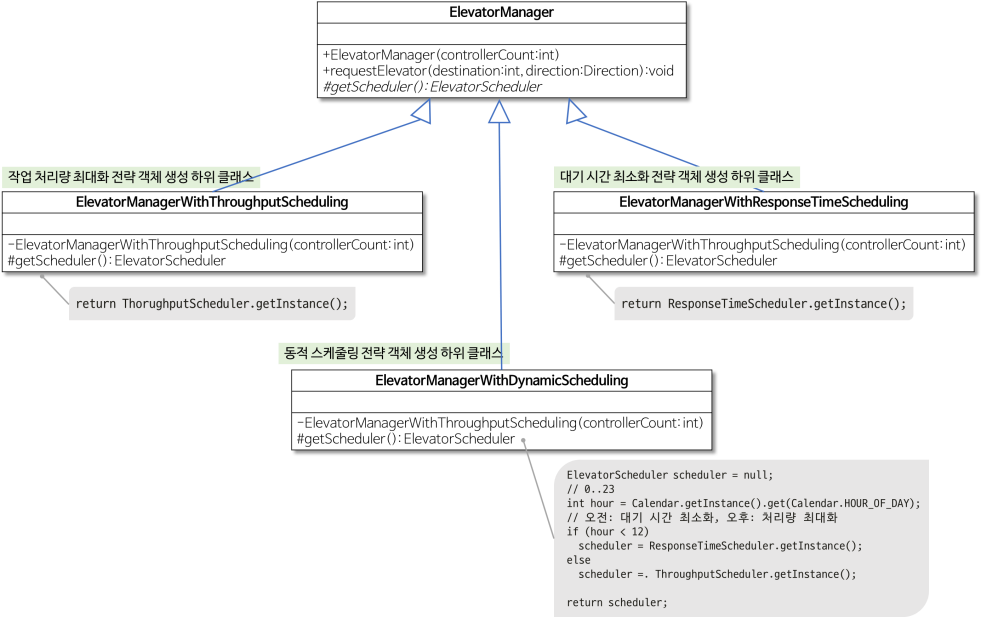




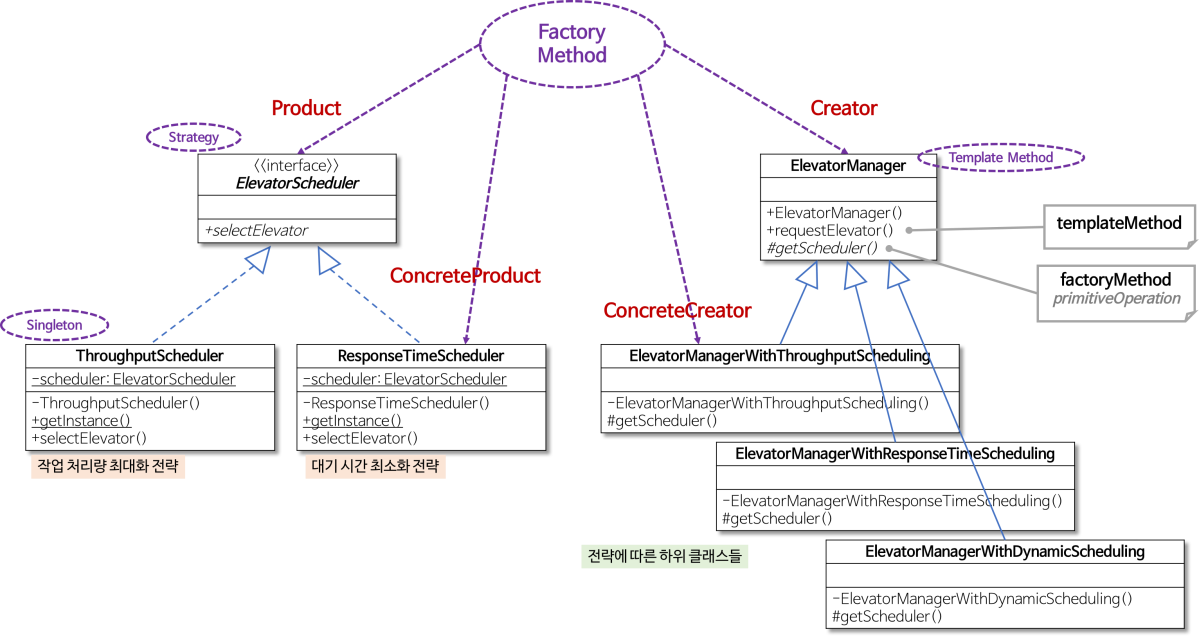
* + Client클래스에는 총 3개의 ElevatorManager객체를 사용하는데, 세 객체 모두 10층으로 이동 요청을 하지만 서로 다른 엘리베이터가 선택할 수 있다.
* 과정 2
  + 동적 스케줄링 방식(DynamicScheduler) 이라고 하면, 여러 번 스케줄링 객체를 생성하지 않고 한번 생성한 것을 계속해서 사용하는 것이 바람직할 수 있다.
    - 싱글턴 패턴을 활용한 엘리베이터 스케줄링 전략을 설계
    - 
  + 스케줄링 기능을 제공하는 ResponseTimeScheduler클래스와 ThroughputScheduler클래스는 오직 하나의 객체만 생성해서 사용하도록 한다.
  + 즉, 생성자를 통해 직접 객체를 생성하는 것이 허용되지 않아야 한다.
    - 이를 위해 각 생성자를 private으로 정의한다
    - 대신 getInstance() 라는 정적 메서드로 객체 생성을 구현한다.
  + 이제 단 한 개의 ThroughputScheduler와 ResponseTimeScheduler객체를 사용할 수 있다.



* + - 다음과 같이 객체 생성을 전담하는 별도의 Factory클래스를 분리하여 객체 생성의 변화를 대비할 수 있다.
    - 이 방법은 스트래티지 패턴과 싱글턴 패턴을 이용하여 팩토리 메서드 패턴을 적용한다.
* 다른 방법으로 팩토리 메서드 패턴 적용하기
  + Factory 클래스 이용
    - schedulerFactory 클래스에서 3가지 방식(최대처리량, 최소대기시간, 동적선택)에 맞춰 ThroughputScheduler 객체나 ResponseTimeScheduler객체를 생성
  + 상속 이용
    - 해당 스케줄링 전략에 따라 엘리베이터를 선택하는 클래스를 elevatorManager클래스의 하위 클래스로 정의
* 상속을 이용
* 하위 클래스에서 적합한 클래스의 객체를 생성하여 객체의 생성 코드를 분리한다.
  + 이 방법은 스트레티지 패턴, 싱글턴 패턴, 템플릿 메서드 패턴을 이용하여 팩토리 메서드 패턴을 적용한다.



* + 팩토리 메서드
    - elevatorManager 클래스의 getScheduler()메서드
    - 스케줄링 전략 객체를 생성하는 기능 제공(즉, 객체 생성을 분리)
    - 참고) 템플릿 메서드 패턴의 개념에 따르면, 하위 클래스에서 오버라이드될 필요가 있는 메서드는 primitive 또는 hook메서드라고도 부른다
  + 템플릿 메서드
    - ElevatorManager 클래스의 requestElevator()메서드
    - 공통기능(스케줄링 전략 객체 생성, 엘리베이터 선택, 엘리베이터 이동)의 일반 로직 제공
    - 하위 클래스에서 구체적으로 정의할 필요가 있는 ‘스케줄링 전략 객체 생성’ 부분은 하위 클래스에서 오버라이드
    - 참고) 템플릿 메서드 패턴을 이용하면 전체적으로는 동일하면서 부분적으로는 다른 구문으로 구성된 메서드의 코드 중복을 최소화시킬 수 있다.
  + 즉 팩토리 메서드를 호출하는 상위 클래스의 메서드는 템플릿 메서드가 된다.
* 상속을 이용한 팩토리 메서드 패턴 적용



* Product : elevatorScheduler인터페이스
* ConcreteProduct : ThroughputScheduler 클래스와 ResponseTimeScheduler 클래스
* Creator : ElevatorManage클래스
* ConcreteCreator : ElevatorManagerWithThroughputScheduling 클래스

ElevatorManagerWithResponseTimeScheduling 클래스

ElevatorManagerWithDynamicScheduling 클래스