목차

1. 객체의 세계
2. 객체 지향적으로 설계 잘하는 법(예시: VLLO 자막 달기 기능)
3. SOLID 법칙을 적용하여 코드 작성

객체의 세계

- 인간의 세계에선 역할, 책임, 협력이 있다→ 객체의 세계에서도 마찬가지
- **협력**이란 특정 요구사항을 해결하기 위해 객체들이 참여하고 도움을 요청하고 응답하며 요구사항을 해결하는 것
- **책임**이란 특정 객체가 특정 요청에 대해 적절한 행동을 할 의무이다.
- **역할**은 관련된 책임의 집합을 의미한다.

그렇다면 객체는 책임을 어떻게 수행할까?

- 객체가 혼자 모든 책임을 수행하는 건 힘들다. 다른 객체의 도움을 받아야한다.
- 객체의 세계에서는 객체가 다른 객체에게 주어진 책임을 수행하도록 요청을 보내는 **메시지**란 의사소통 수단이 존재한다

객체 지향 프로그래밍의 장점은?

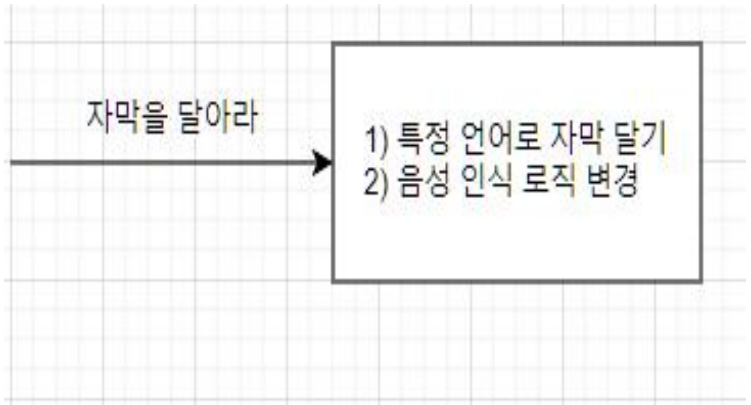
- 유지보수하기 좋고 재사용이 용이한 시스템을 구축할 수 있다.

객체지향적으로 설계를 잘하는 방법

- 송신하는 객체가 수신하는 객체가 무엇인지에 대해서 **알지 못하게 하여라**
- 객체가 메시지를 선택하는 것이 아닌 **메시지가 객체를 선택**하도록 하라
- 객체가 **자율성**을 갖도록 하라.

송신하는 객체가 수신하는 객체에 대하여 몰라도 된다

- 객체의 **캡슐화**와 관련이 있다.
- 송신하는 객체가 수신하는 객체가 **어떻게** 책임을 수행하는지에 대해서 알지 못 한채 메시지만 전송한다면, 수신자가 어떤 객체가 되었든 상관이 없다.
- 송신 객체에게 아무런 영향을 미치지 않고도 새로운 유형의 자막(손글씨를 인식하여 자막 달기 등등)을 다는 기능을 추가할 수 있다.
- 즉 자막을 다는 수신 객체를 캡슐화함으로 **다형성**을 높이며 다형성이 높아짐에 따라 **재사용성**이 높아질 수 있습니다.



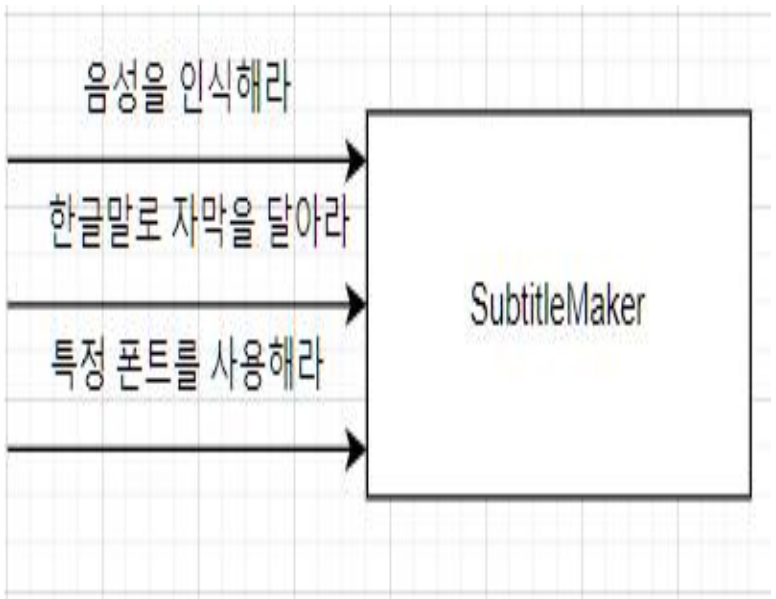
메시지가 객체를 선택한다.

- 객체가 메시지를 선택하도록 하면 데이터 중심의 설계를 하기 쉽다. **데이터 중심 설계**는 객체 자신이 포함해야 하는 데이터를 설계하고 이후 그에 따른 행동을 정의한다.
- Ex) 자막을 달아주는 객체가 있다. 자막을 달아주는 객체는 자막의 배경 색깔들 정보와 배경 모양들 정보도 있을 것이다. -> 이후 배경을 만들어주는 객체를 만들어야겠다고 판단 -> 자막을 달아주는 객체의 배경 색깔과 모양들의 데이터를 요구하는 상황이 올 수 있다.

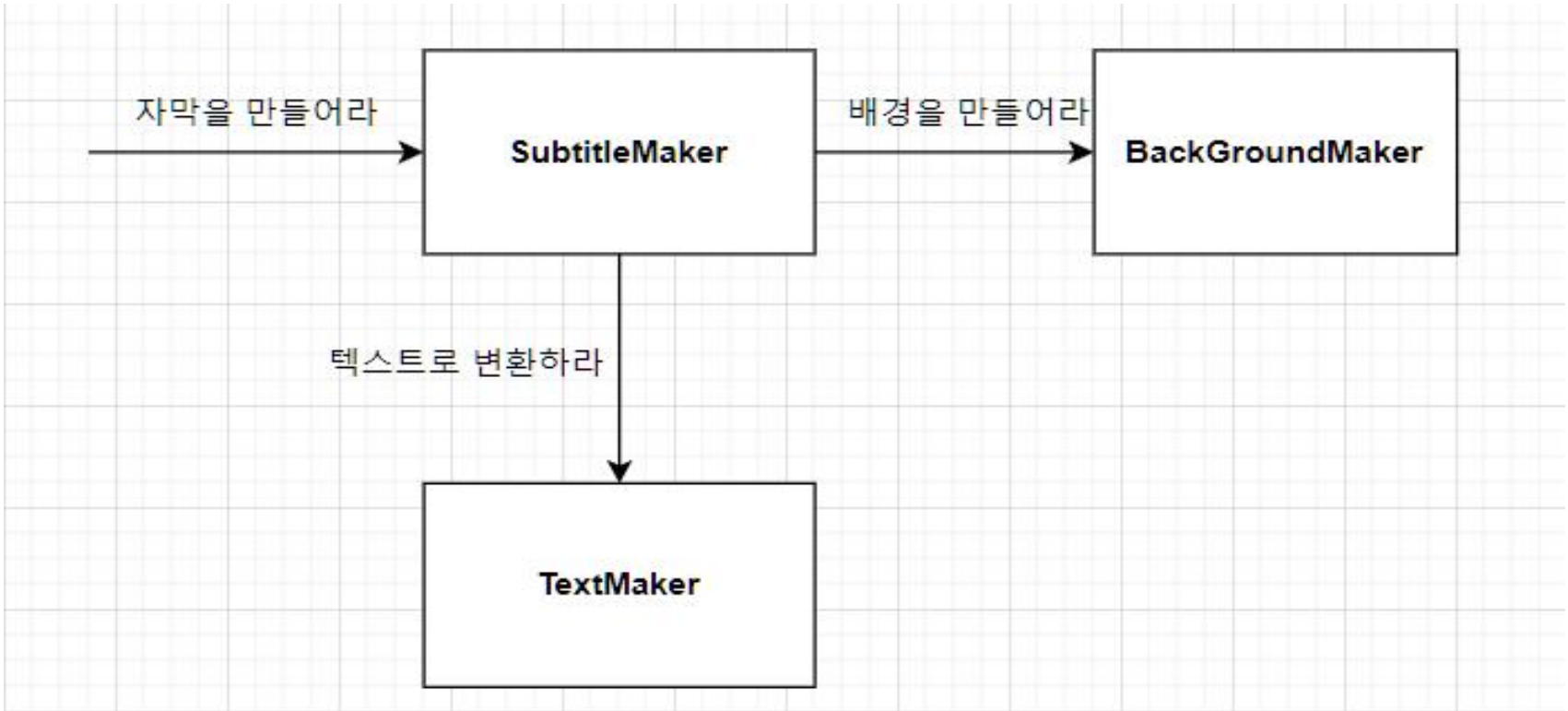
- 데이터부터 생각한다면 자연스레 불필요하고 많은 데이터가 생기고 **캡슐화**가 깨지기 쉽다 -> 메시지에 맞춰 그때 그때 필요한 최소한의 데이터만을 만들어야한다.

객체가 자율성을 갖도록 하라

- 너무 구체적으로 메시지를 보내는 것은 객체의 자율성을 저하
- 이후에 영어로 자막을 달아야하는 기능이 추가된다면, 책임을 요청하는 측이 추가로 영어로 자막을 달으라고 요청해야한다. -> 요지보수성 저하
- "자막을 달아라" 라는 책임만 완수할 수 있다면 어떠한 방법으로 자막을 다는지에 관해서는 신경을 쓰지 않아도 된다.
- 충분한 자율성을 가지고 "자막을 달아라" 라는 책임을 수행할 수 있는 객체는 또한 음성을 인식해라, 한글(영어)로 자막을 달아라는 책임에 모두 대응을 할 수 있음 -> **재사용성** 높아짐



SOLID 법칙을 적용하여 코드 작성



SRP: 하나의 클래스는 하나의 책임만 갖도록한다

```
class TextMaker{
    String text;
    public void translate(){}; // 번역
    public void detect(){}; // 인식
}
```

```
class TextMaker{
    private Translator translator;
    private Detector detector;
    public void makeText() ...
}

interface Translator{
    public String translate(String text) ...
}

interface Detector{
    public String detect() ...
}
```

OCP: 소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다.

```
class TextMaker{
    private KoreanTranslator translator;
    private EnglishTranslator englishTranslator;
    private Detector detector;
    private String text;
    public void makeText(){};
}
```

```
class TextMaker{
    private Translator translator;
    private Detector detector;
    private String text;
    public void makeText(){};
}
```

```
interface Translator{
    String translate(String text);
}

class EnglishTranslator implements Translator{

    @Override
    public String translate(String text) ...
}
class KoreanTranslator implements Translator{
    @Override
    public String translate(String text) ...
}
```

LSP: 프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다

```
class Translator{
    public void applyHonorifics() ... //존댓말 적용
    public String translate(String text);
}
class EnglishTranslator extends  Translator{
    public void applyHonorifics(){
        throw new Exception("영어는 존댓말을 적용할 수 없음");
    }
    public String translate(String text);
}
```

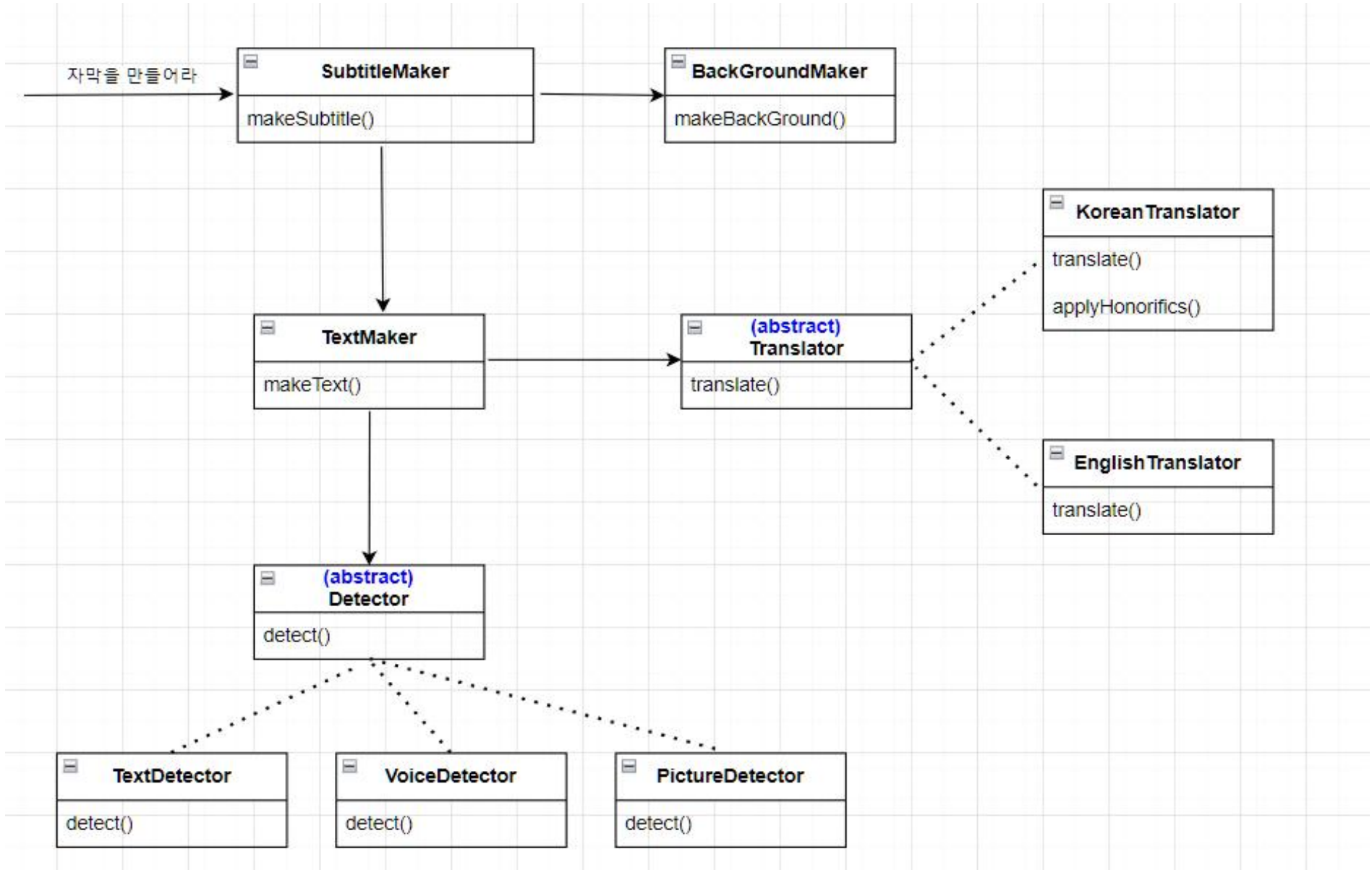
ISP 인터페이스를 사용에 맞게 끔 각기 분리해야한다

```
class Translator{
    public String translate(String text);
}
class KoreanTranslator extends  Translator{
    public void applyHonorifics() ... //존댓말 적용
    public String translate(String text)...
}
```

DIP :변하기 쉬운 것에 의존하기 보다, 잘 변하지 않는 것에 의존해야 한다

```
class TextMaker{
    private final Translator translator;
    public TextMaker(Translator translator) {
        this.translator = translator;
    }

    private String text;
    public void detect(){}; // 인식
}
```


자막을 만들어라
SubtitleMaker
makeSubtitle()
BackGroundMaker
makeBackGround()
TextMaker
makeText()
(abstract)
Translator
translate()
KoreanTranslator
translate()
applyHonorifics()
EnglishTranslator
translate()
(abstract)
Detector
detect()
TextDetector
detect()
VoiceDetector
detect()
PictureDetector
detect()

감사합니다.