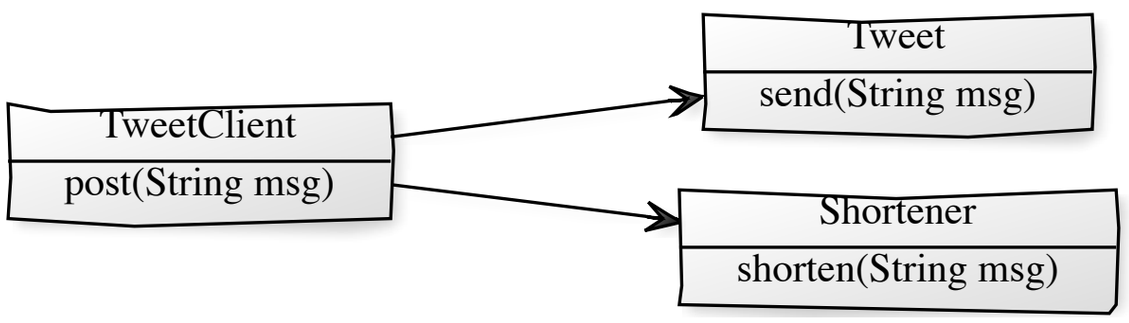
README.md

Tutoriel : Injection de dépendances

Exemple : Messagerie *Twitter*

Dans ce tutoriel nous allons nous baser sur l'exemple d'une messagerie *Twitter*. Cette application est composé d'une classe client *TwitterClient* qui dépends de deux autres fonctionnalités : une pour écourter le message si sa longueur dépasse 140 caractères autorisé par *Twitter*, et l'autre pour envoyer le message. Ces dépendances sont illustrées dans le diagramme de classes ci-dessous.



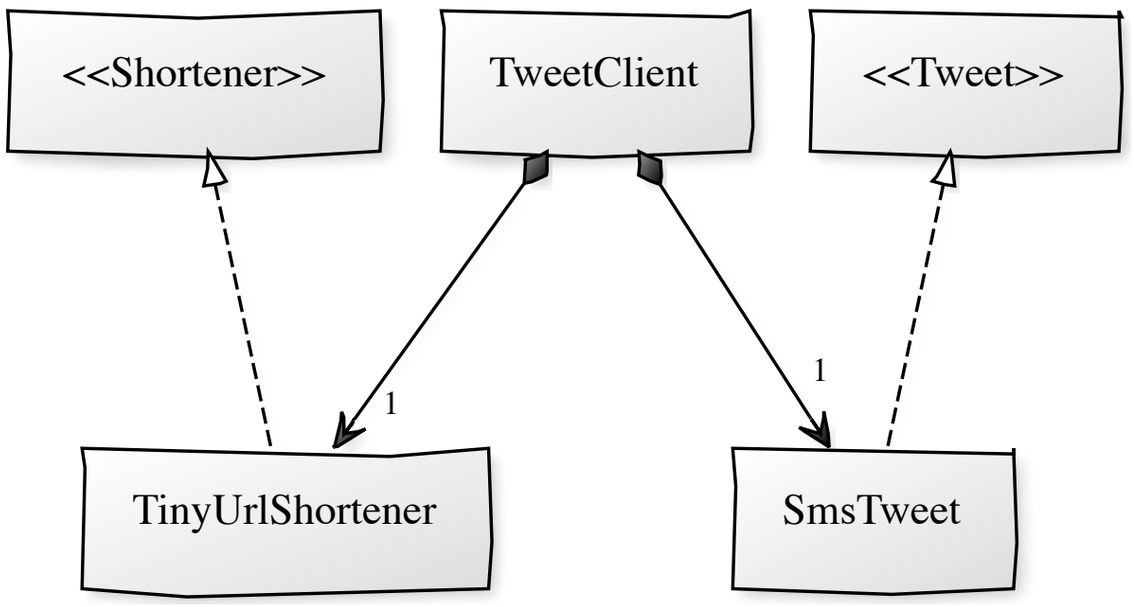

```
public void post(String msg){
   if (msg.length() > 140){
       msg = shortener.shorten(msg);
   }

   if (msg.length() <= 140){
       tweeter.send(msg);
   }
}
```

Gestion de dépendances

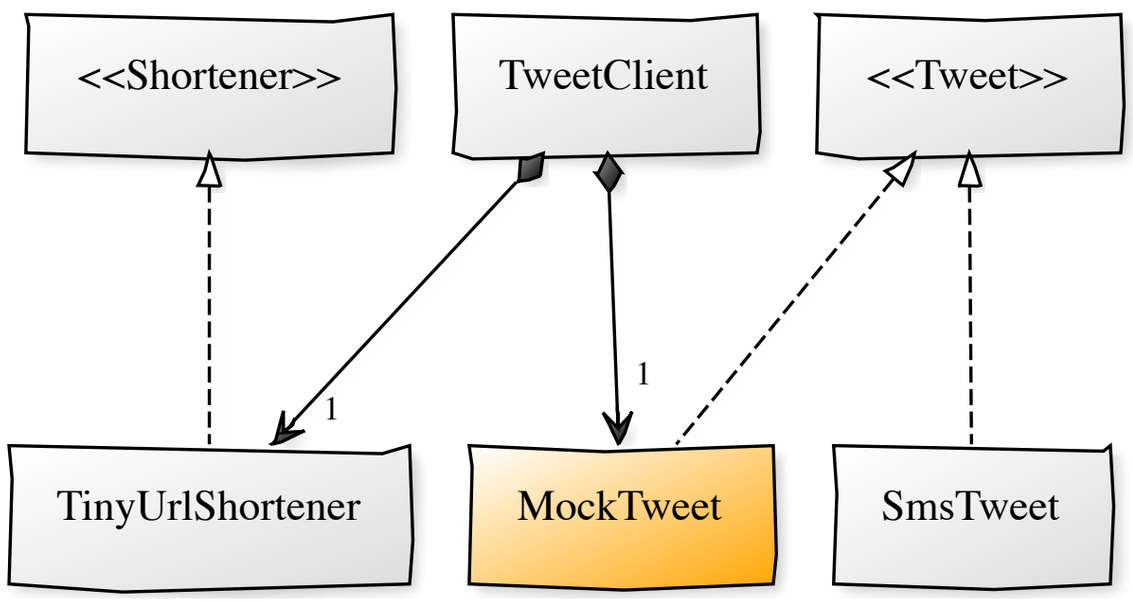
Il y a différentes moyens de gérer les dépendances dans la programmation orienté objet. L'objectif de cette section est de montrer les avantages et inconvénients de quelques unes.

Dépendances dans le constructeur



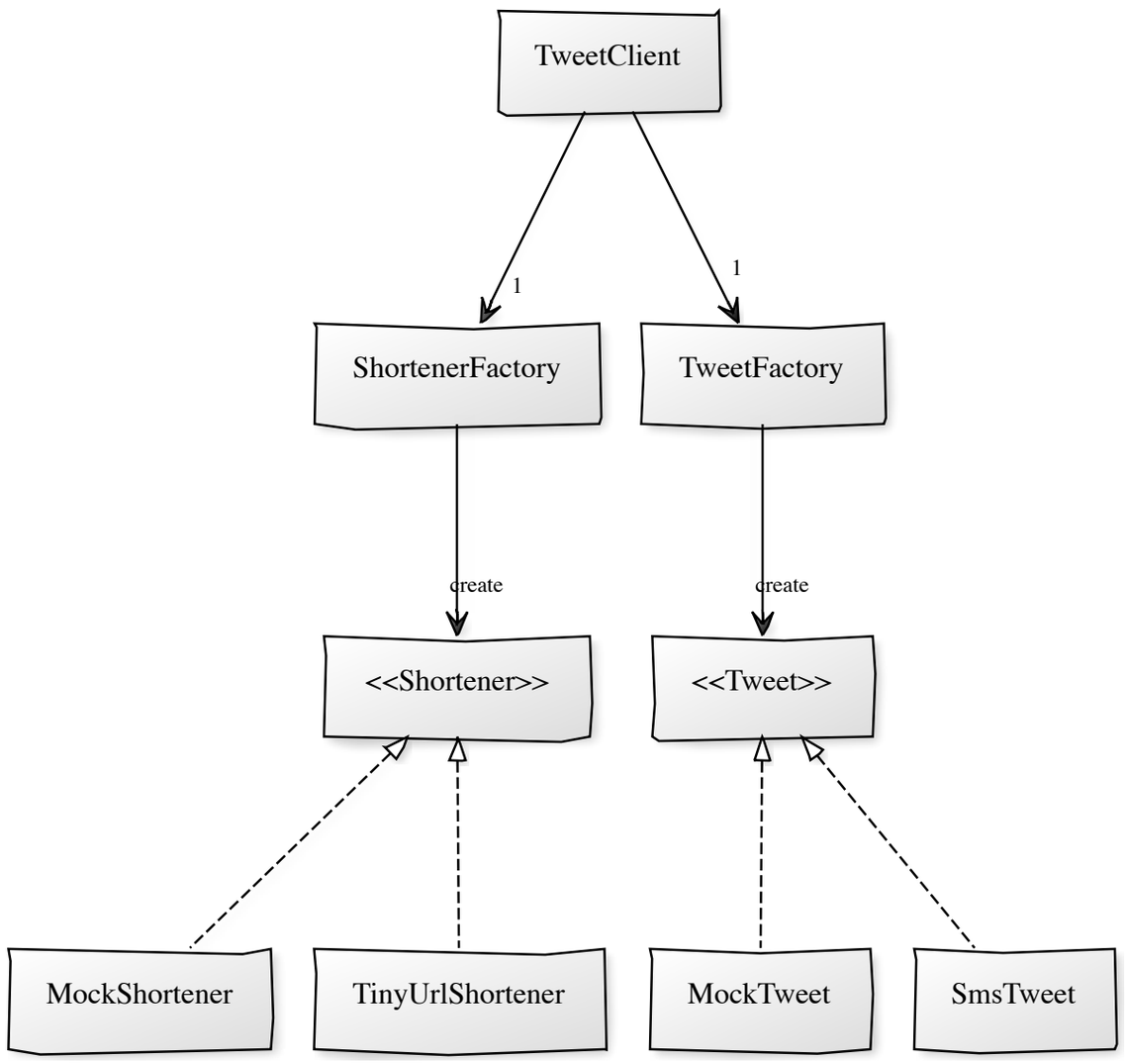
- Implémentez les classes indiquées dans le diagramme ci-dessous
- Implémentez une classe de teste pour tester votre implémentation
- Quels sont les inconvénients de l'approche par composition?

Supposons que nous voulions tester notre application avec une classe `MockTweet` à la place de `SmsTweet` car envoyer des messages SMS pour effectuer des testes coûte de l'argent.



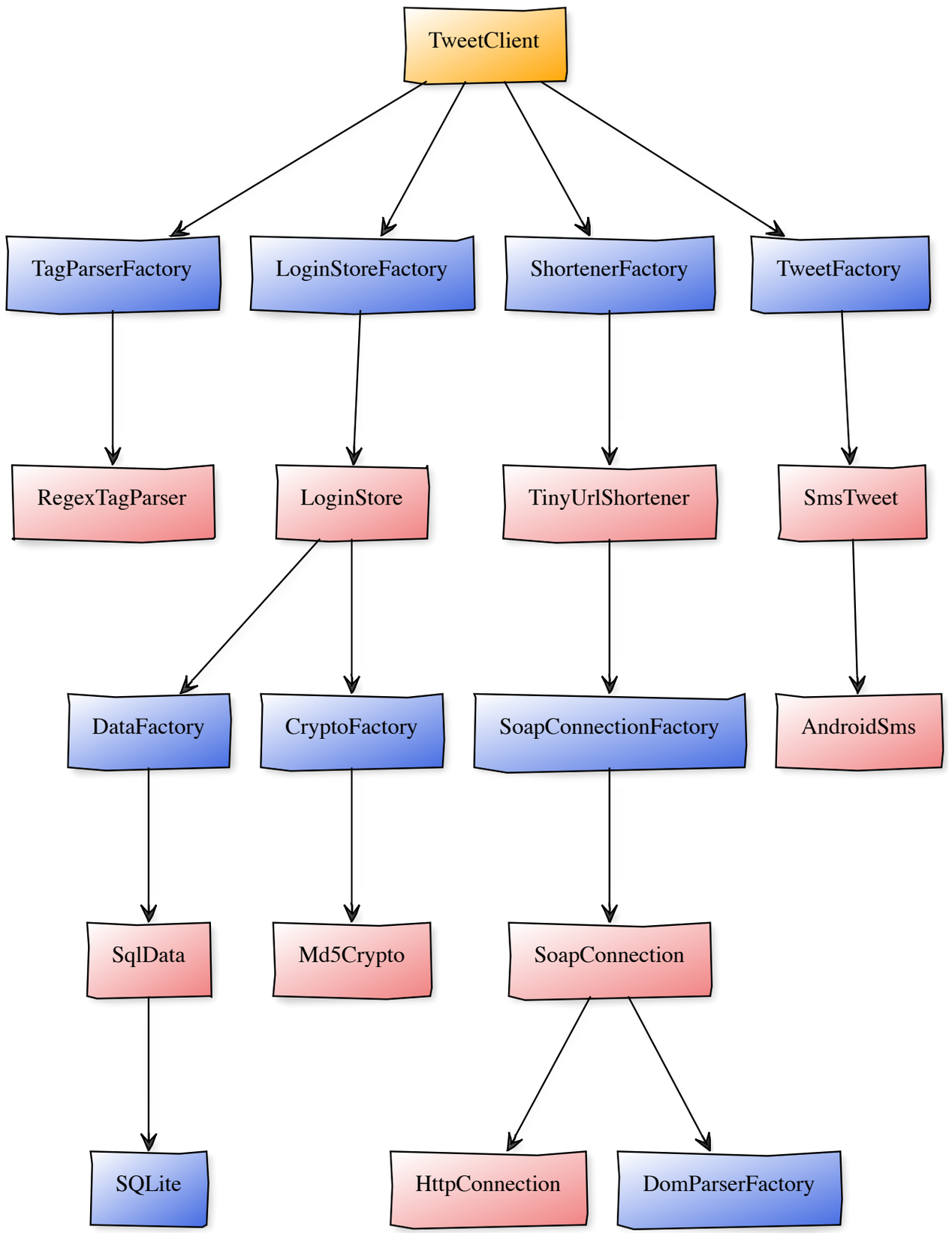
Ce petit changement engendre un changement aussi dans la (les) classe (s) dépendante (s), ce qui montre le **manque de flexibilité** de cette approche.

Dépendance via des fabriques statiques



- Implémentez les classes indiquées dans le diagramme
- Implémentez une classe de teste pour tester votre implémentation
- Quels sont les avantages par rapport à l'approche par composition?
- Quels sont les inconvénients de cette approche?

Il est clair que cette approche augmente le niveau d'abstraction car on accède plus directement les constructeurs des dépendances, mais le problème de dépendances demeure. La seule différence est que maintenant les classes métiers dépendent des fabriques au lieu d'autres classes métiers, ce qui peut être gênant pour les applications plus grandes, comme illustré ci-dessous.

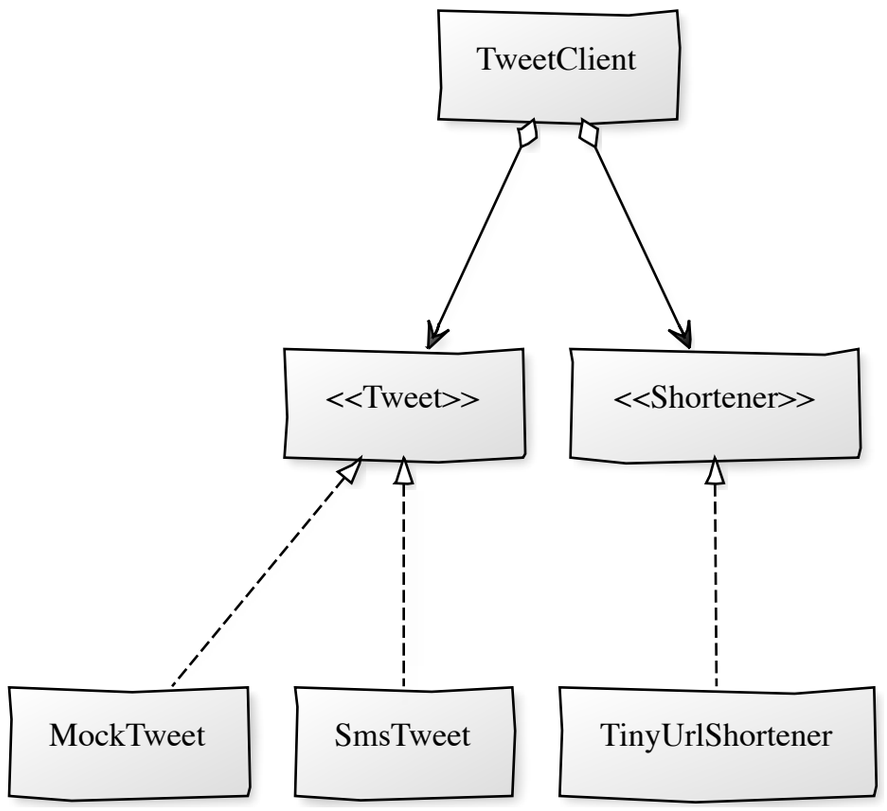


Injection de dépendances

Au lieu de créer les dépendances (ou utiliser les fabriques) à l'intérieur des classes, mieux vaut, du point de vue de la modularité, injecter les dépendances créés auparavant.

Injection de dépendances par agrégation

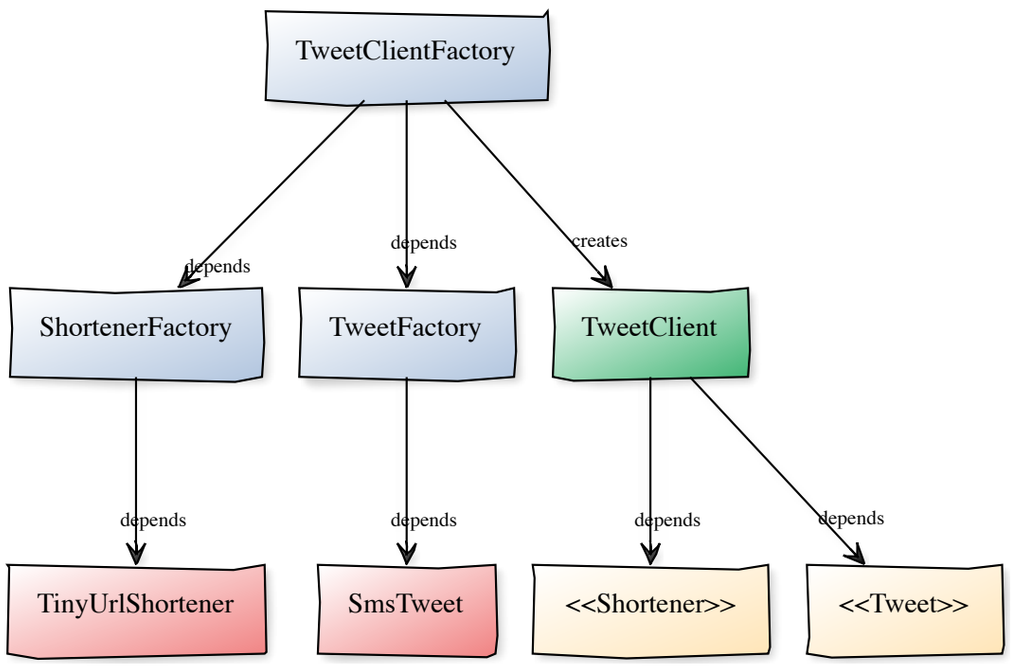
- Implémentez les classes indiquées dans le diagramme ci-dessous
- Implémentez une classe de teste pour tester votre implémentation
- Quels sont les avantages par rapport aux approches précédentes?



Par rapport aux approches précédentes, celle-ci a l'avantage d'isoler les dépendances des classes dépendantes, c'est à dire, les dépendances sont injectées par agrégation (dans cet exemple via les constructeurs). Une changement d'implémentation de la dépendance n'engendre pas un changement dans la classe dépendante (p.e. : *SmsTweet* -> *MockTweet*) tant que l'interface n'est pas modifiée. Cependant, c'est à la classe client (dans notre exemple *TwitterClient*) de créer et gérer les dépendances, ce qui peut devenir vite une tâche très laborieuse.

Injection de dépendances par agrégation + fabriques

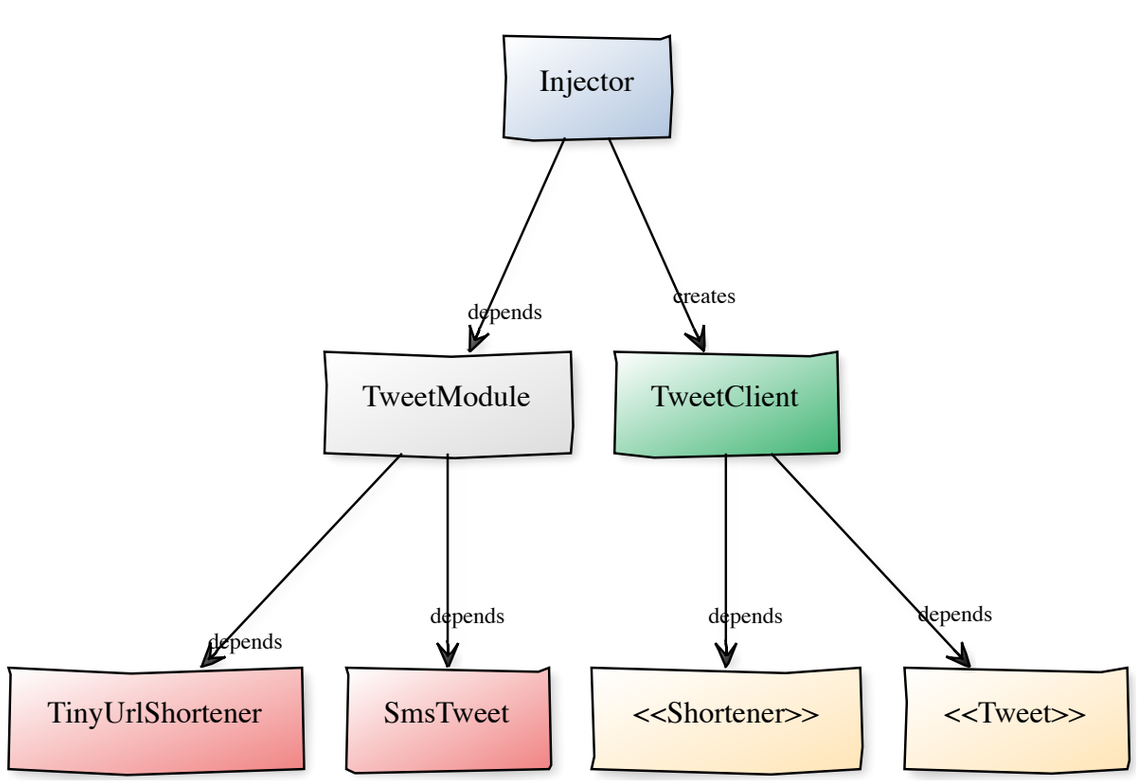
- Implémentez les classes indiquées dans le diagramme
- Implémentez une classe de teste pour tester votre implémentation
- Quels sont les avantages par rapport aux approches précédentes?
- Quels sont les inconvénients de cette approche?



Par rapport aux autres approches, cette approche a l'avantage d'isoler les dépendances de leur création via les fabriques. Par contre, est-ce que nous avons besoin de coder manuellement toutes ces fabriques ?

Injection de dépendances avec Guice

Guice est un *framework* Java qui facilite la mise en œuvre du patron de conception *injection de dépendances*. Il suffit de définir un *mapping* des interfaces (types) des dépendances vers les classes implémentation pour pouvoir bénéficier d'un injecteur générique. Ceci joue le rôle d'une fabrique, c'est à dire que l'injecteur s'occupe de créer et injecter les dépendances.



Module

```
import com.google.inject.AbstractModule;

public class TweetModule extends AbstractModule {
   @Override
   protected void configure() {
    bind(Twitter.class).to(SmsTwitter.class);
    bind(Shortener.class).to(TinyUrlShortener.class);
   }
}
```

Injecteur

```
Injector injector = Guice.createInjector(new TweetModule());
TweetClient client = injector.getInstance(TweetClient.class);
```

Classe dépendante

Du côté de la classe dépendante, on a également besoin de préciser quelles sont les dépendances que l'on veut que soient injectées.

```
import com.google.inject.Inject;

public class TweetClient {
    ...
    @Inject
    public TwitterClient(Shortener shortener, Tweet twitter) {
        this.shortener = shortener;
        this.twitter = twitter;
    }
    ...
}
```