Différence entre les Design Pattern

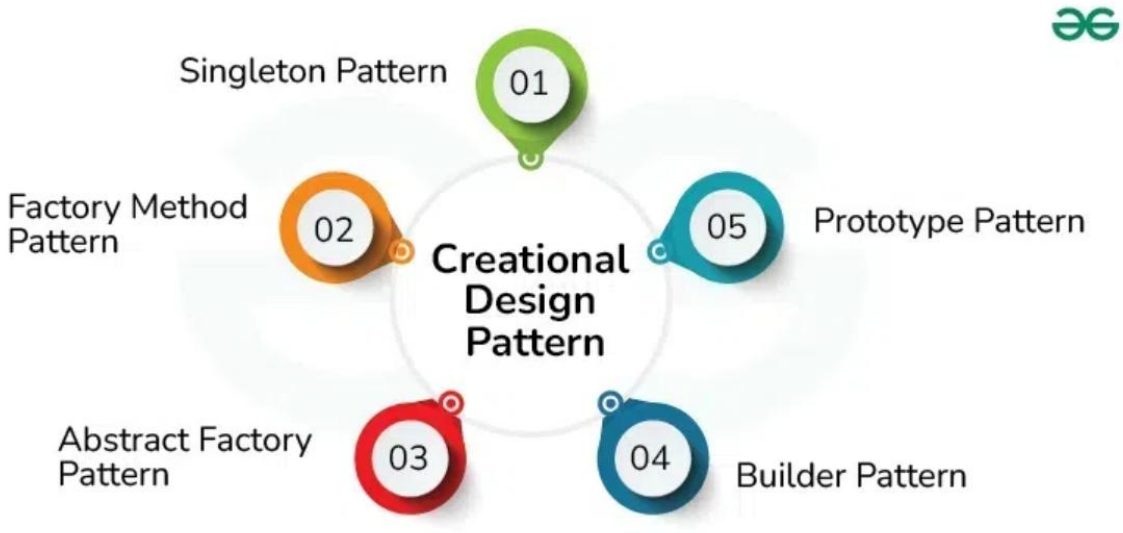
Introduction

Les design patterns sont des solutions éprouvées aux problèmes courants de conception logicielle. Ils permettent de standardiser la conception, d'améliorer la maintenabilité et de faciliter la collaboration entre développeurs.

On distingue trois grandes catégories de design patterns : les patterns **créationnels**, **structurels** et **comportementaux**. Ce rapport explore chaque catégorie et met en évidence les relations entre les différents patterns les plus utilisés.

1. Les Design Patterns Créationnels

Les patterns créationnels concernent la manière dont les objets sont créés. Ils permettent d'encapsuler la logique de création pour éviter les instanciations directes et faciliter l'extensibilité du code.



Principaux Patterns Créationnels

Singleton : Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global.

Exemple en Spring Boot : Utilisation d'un bean Singleton avec `@Service` ou `@Component`.

```
@Service
public class MySingletonService {
    private static final MySingletonService instance = new
MySingletonService();
    private MySingletonService() {}
    public static MySingletonService getInstance() {
        return instance;
    }
}
```

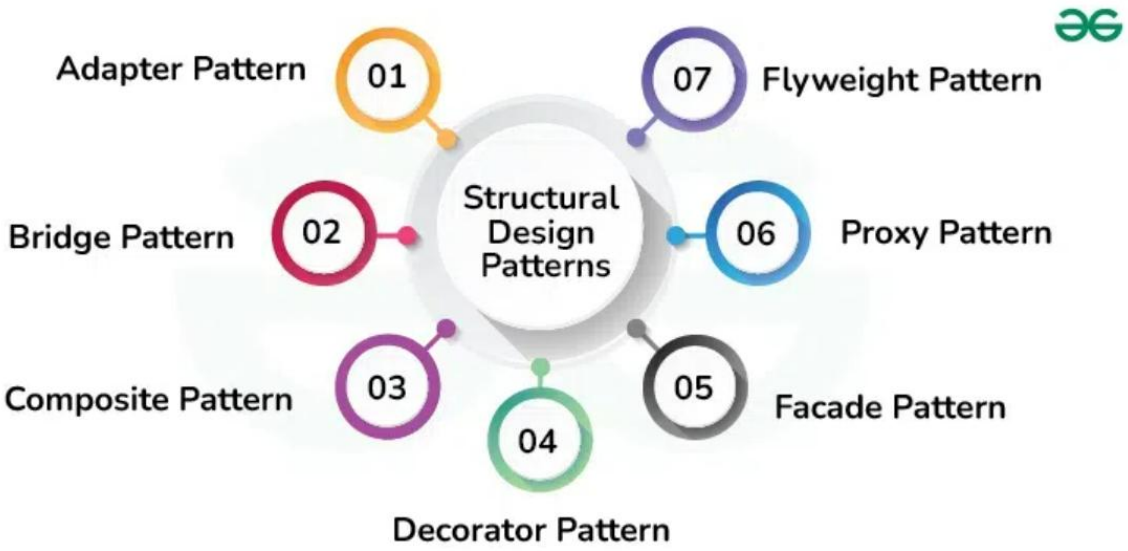
Factory Method : Définit une interface pour créer des objets dans une sous-classe, ce qui permet d'adapter dynamiquement la création d'objets.

Exemple en Spring Boot : Utilisation de `@Bean` pour instancier un objet spécifique.

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

2. Les Design Patterns Structurels

Les patterns structurels s'intéressent à la manière dont les classes et objets sont composés afin d'assurer une structure modulaire et flexible.



Principaux Patterns Structurels

Adapter : Permet d'adapter une interface existante pour la rendre compatible avec une autre interface.

Exemple en Spring Boot : Utilisation d'un adaptateur pour convertir un objet en un autre format.

```
public class UserDtoAdapter {
    public static UserDto convert(User user) {
        return new UserDto(user.getId(), user.getName());
    }
}
```

Facade : Fournit une interface simplifiée pour masquer la complexité d'un ensemble de sous-systèmes.

Exemple en Spring Boot : Création d'un service unique pour encapsuler plusieurs opérations métier.

```
@Service
public class UserFacade {
    private final UserService userService;
    private final EmailService emailService;
```

```
    public UserFacade(UserService userService, EmailService emailService)
{
        this.userService = userService;
        this.emailService = emailService;
    }

    public void registerUser(UserDto userDto) {
        userService.saveUser(userDto);
        emailService.sendWelcomeEmail(userDto.getEmail());
    }
}
```

3. Les Design Patterns Comportementaux

Les patterns comportementaux se concentrent sur la manière dont les objets interagissent et communiquent entre eux.

Principaux Patterns Comportementaux

Observer : Définit une relation de dépendance entre objets afin qu'un changement dans un objet notifie automatiquement ses abonnés.

Exemple en Spring Boot : Utilisation de `ApplicationListener` pour écouter les événements.

```
@Component
public class UserEventListener implements
ApplicationListener<UserCreatedEvent> {
    @Override
    public void onApplicationEvent(UserCreatedEvent event) {
        System.out.println("User created: " + event.getUser().getName());
    }
}
```

Strategy : Permet de sélectionner dynamiquement un algorithme parmi plusieurs.

Exemple en Spring Boot : Implémentation de différentes stratégies de paiement.

```
public interface PaymentStrategy {
    void pay(double amount);
}

@Component
public class CreditCardPayment implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Paid with credit card: " + amount);
    }
}
```

Conclusion

Les design patterns offrent des solutions optimisées pour les problèmes récurrents en conception logicielle. En combinant judicieusement ces patterns, il est possible d'obtenir un code plus flexible, réutilisable et maintenable. La connaissance approfondie de ces patterns permet aux développeurs de concevoir des architectures robustes et évolutives.