Structural Pattern: Decorator

Kevin Dockx
Architect

@KevinDockx https://www.kevindockx.com

Coming Up

Describing the decorator pattern

Implementation:

- Collecting statistics & storing emails

Structure of the decorator pattern

Coming Up

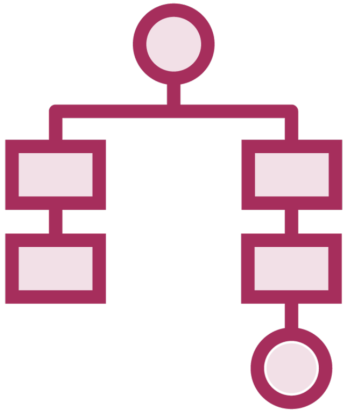
Use cases for this pattern

Pattern consequences

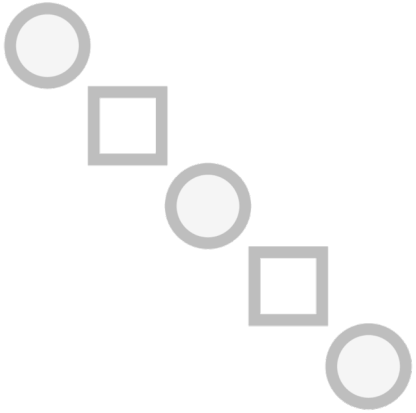
Related patterns

Describing the Decorator Pattern

Creational



Structural



Behavioral

Decorator

The intent of this pattern is to attach additional responsibilities to an object dynamically. A decorator thus provides a flexible alternative to subclassing for extending functionality.

Describing the Decorator Pattern

Adding responsibility to a class can be done by adding an additional method to the class

- We don't want to add that responsibility to a class, we want to add it to an ***instance of a class*** at runtime: an ***object***

```
public class CloudMailService {
    public void SendMail() { … } }

public class OnPremiseMailService {
    public void SendMail() { … } }
```

Describing the Decorator Pattern

```
public interface IMailService {
    void SendMail(); }

public class CloudMailService : IMailService {
    public void SendMail() { … } }

public class OnPremiseMailService : IMailService {
    public void SendMail() { … } }
```

Describing the Decorator Pattern

```
public interface IMailService {
    void SendMail(); }

public class CloudMailService: IMailService {
    public void SendMail() { // additional functionality }}

public class OnPremiseMailService : IMailService {
    public void SendMail() { // additional functionality }}
```

Describing the Decorator Pattern

Classes can be extended with additional functionality

- Leads to violations of the single responsibility principle

```
public interface IMailService {
    void SendMail(); }

public class CloudMailService: IMailService {
    public void SendMail() { // additional functionality }
    // even more additional functionality }

public class OnPremiseMailService : IMailService {
    public void SendMail() { // additional functionality }
    // even more additional functionality }
```

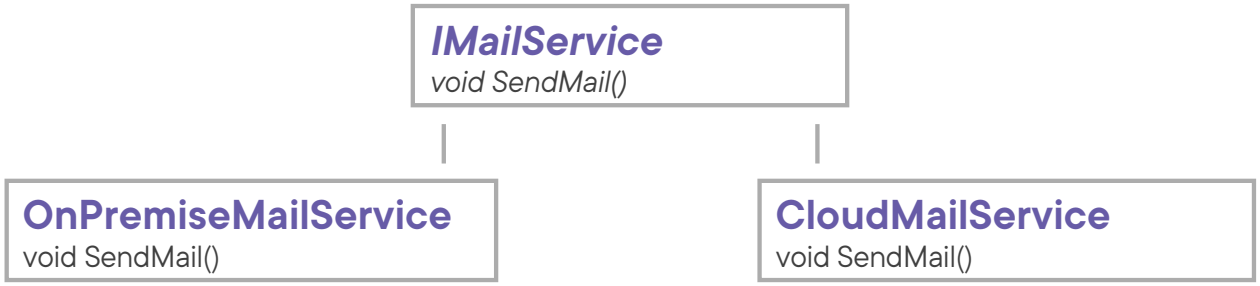
Describing the Decorator Pattern

Classes can be extended with additional functionality

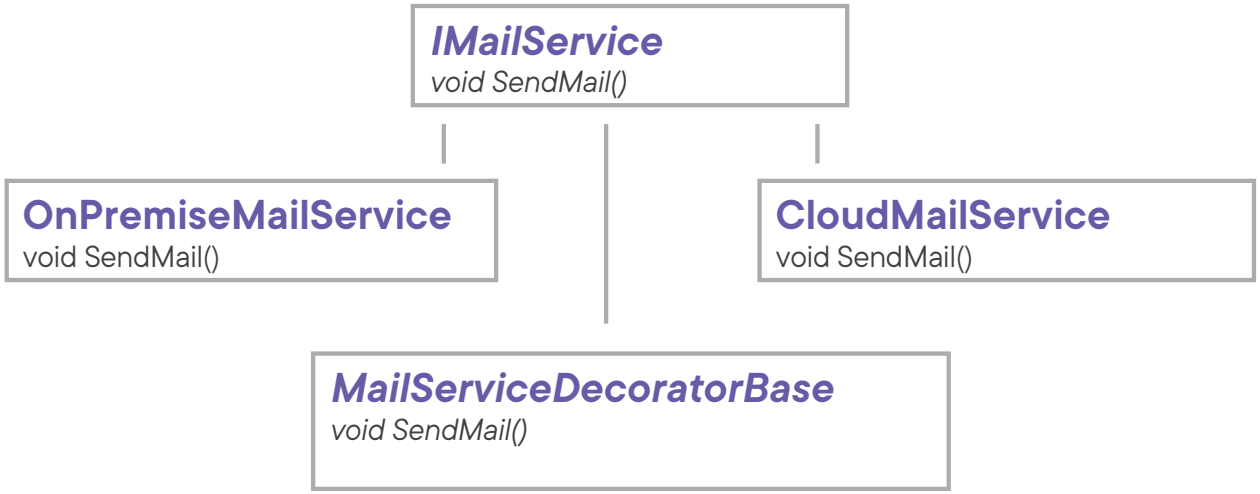
- Leads to violations of the single responsibility principle
- Classes get littered with code that doesn't belong there

Describing the Decorator Pattern

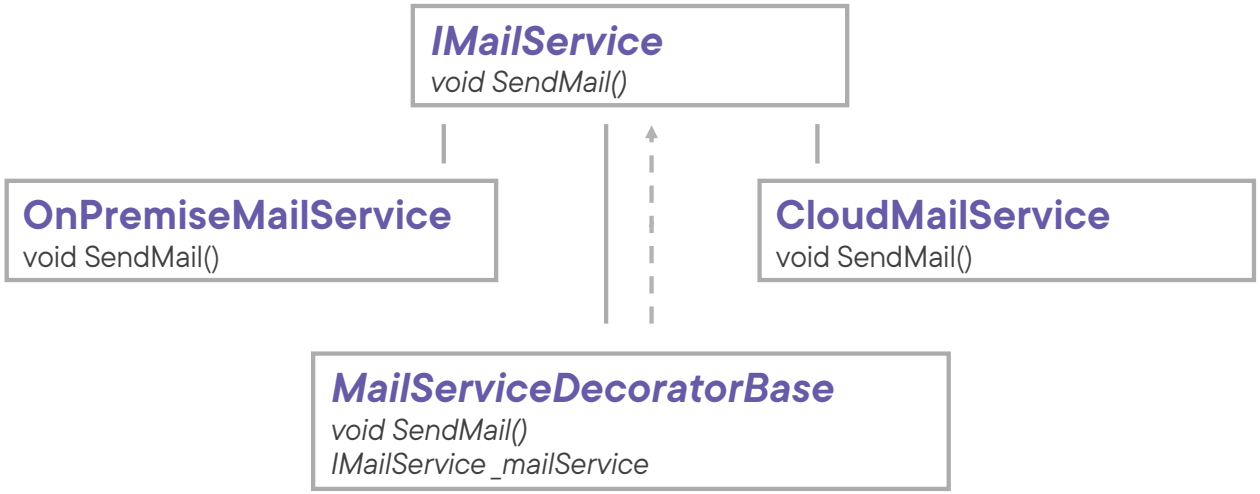
Describing the Decorator Pattern



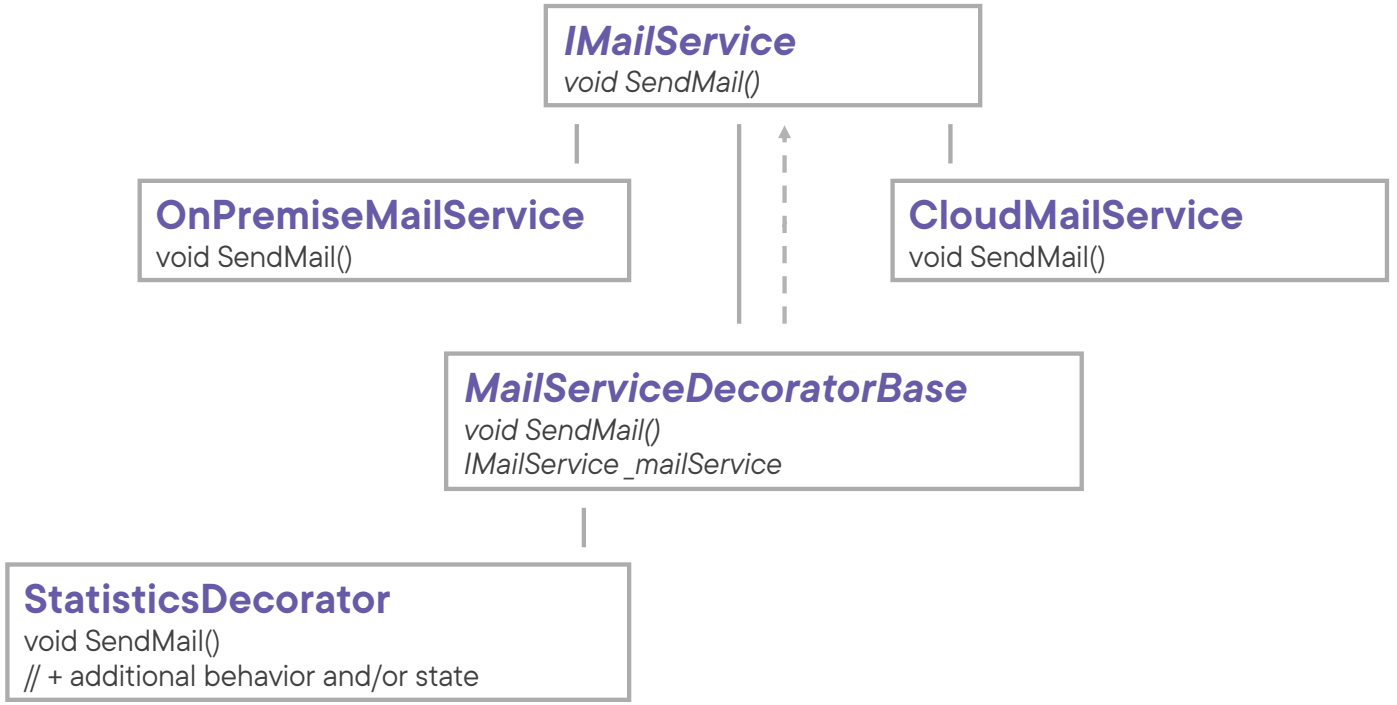
Describing the Decorator Pattern



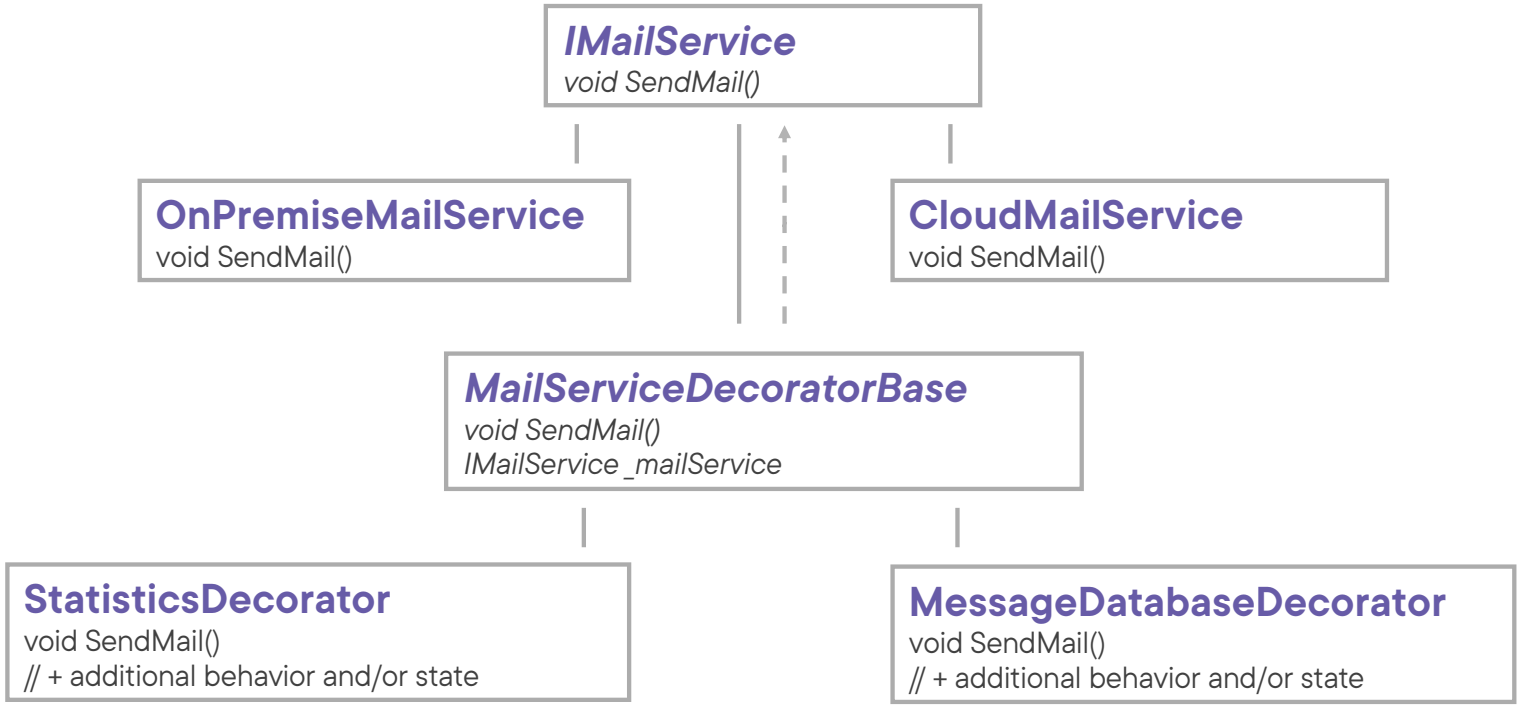
Describing the Decorator Pattern



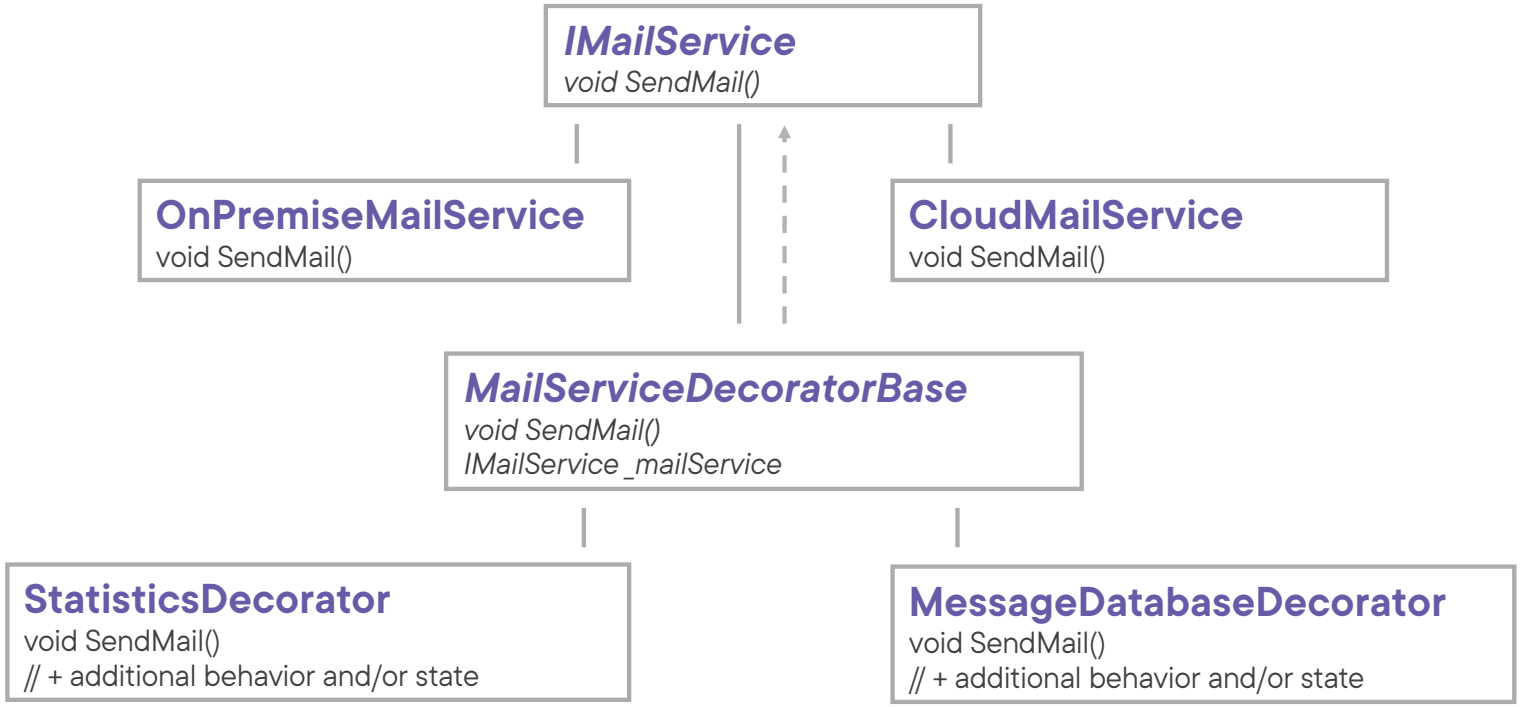
Describing the Decorator Pattern



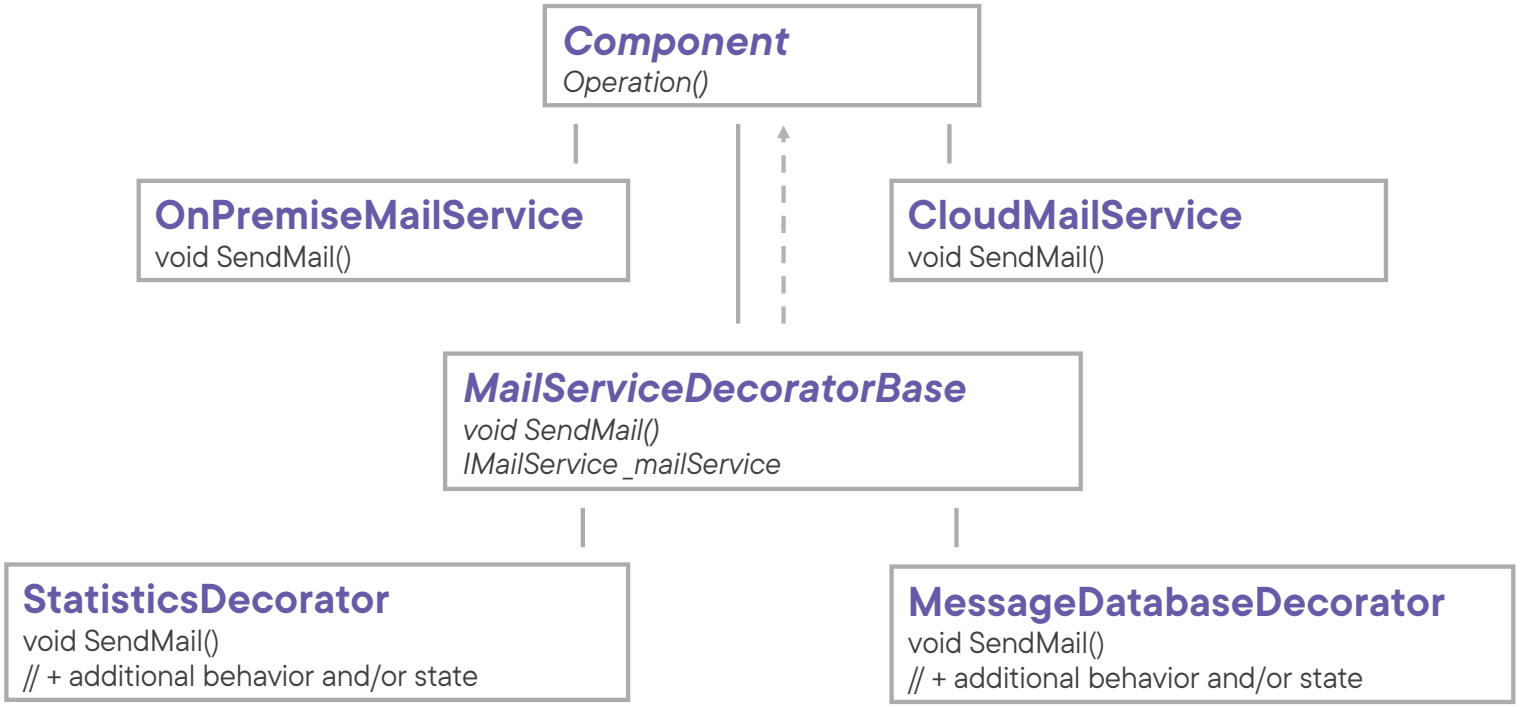
Describing the Decorator Pattern



Structure of the Decorator Pattern

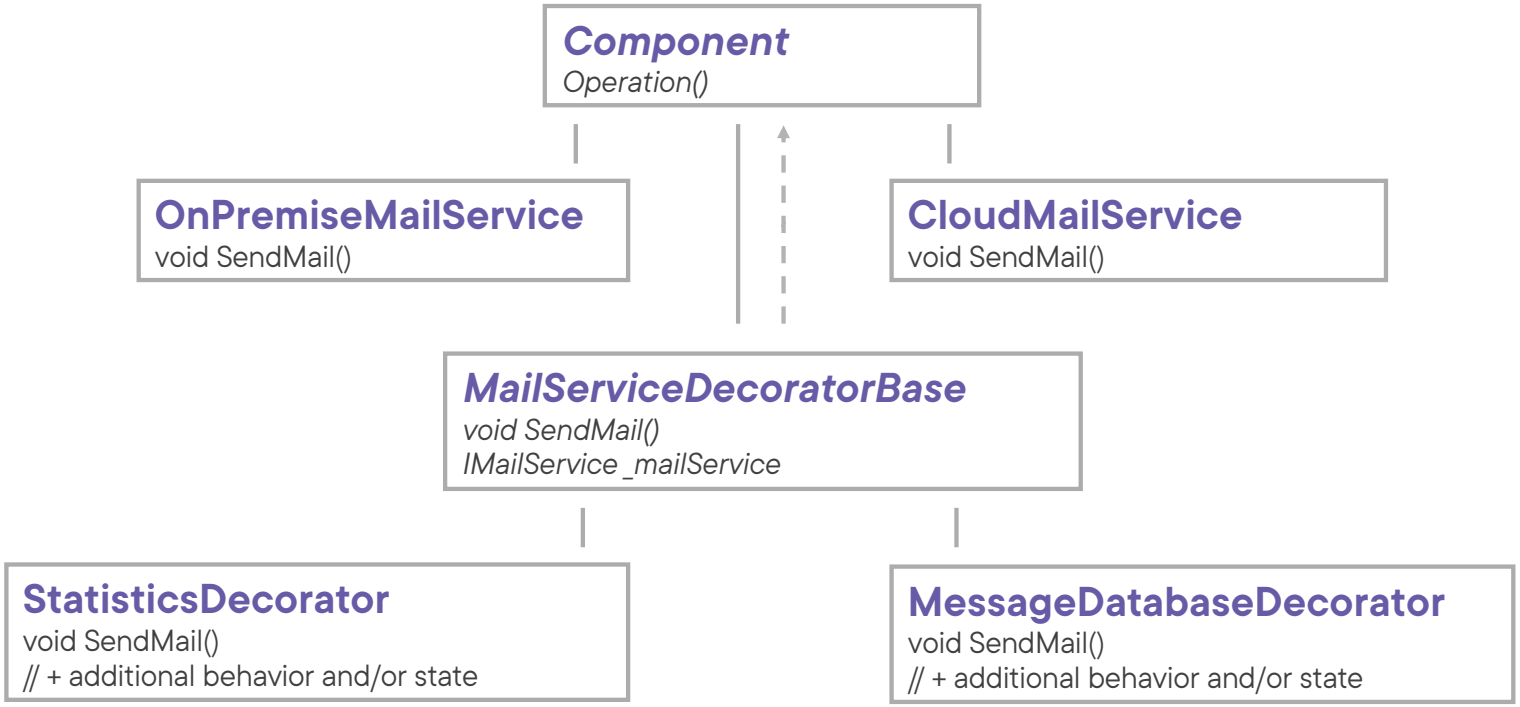


Structure of the Decorator Pattern

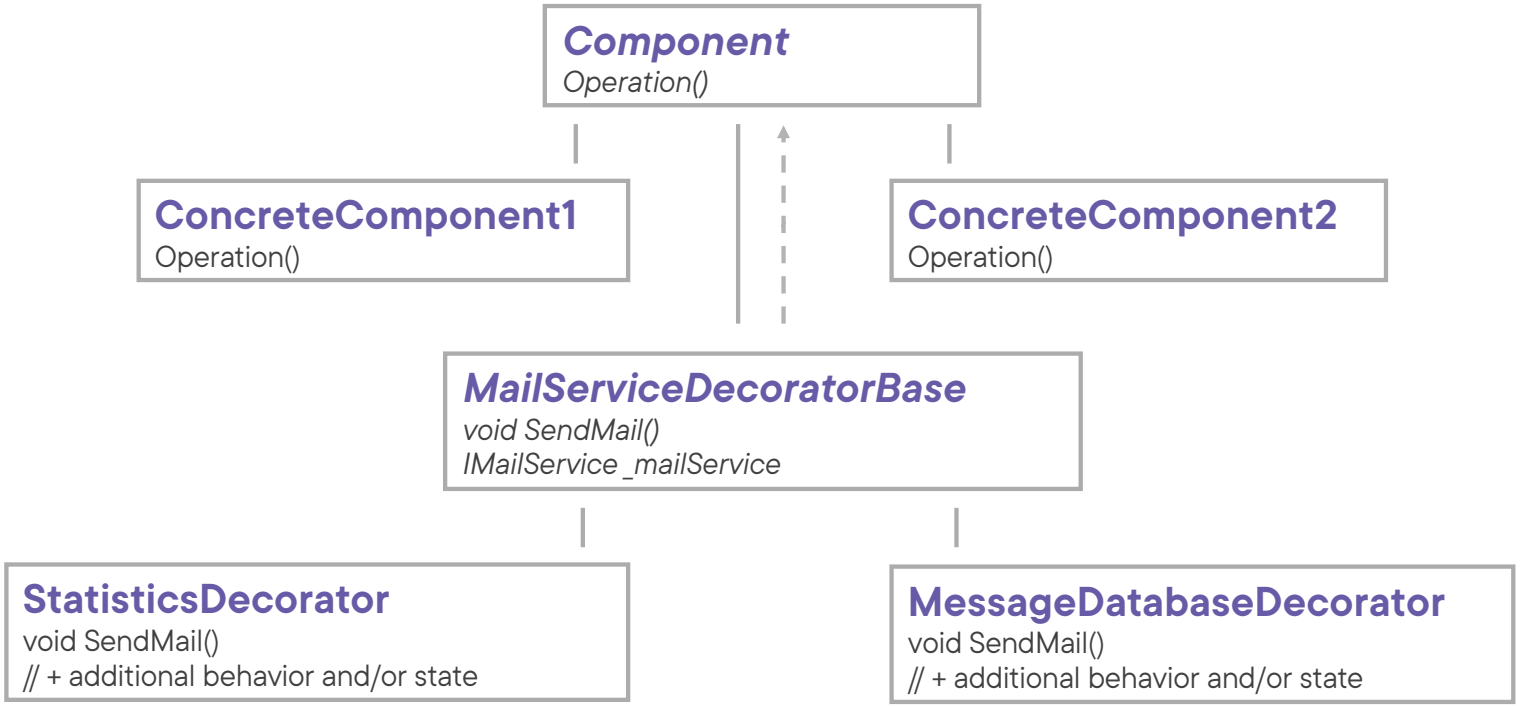


Component defines the interface for objects that can have responsibilities added to them dynamically

Structure of the Decorator Pattern

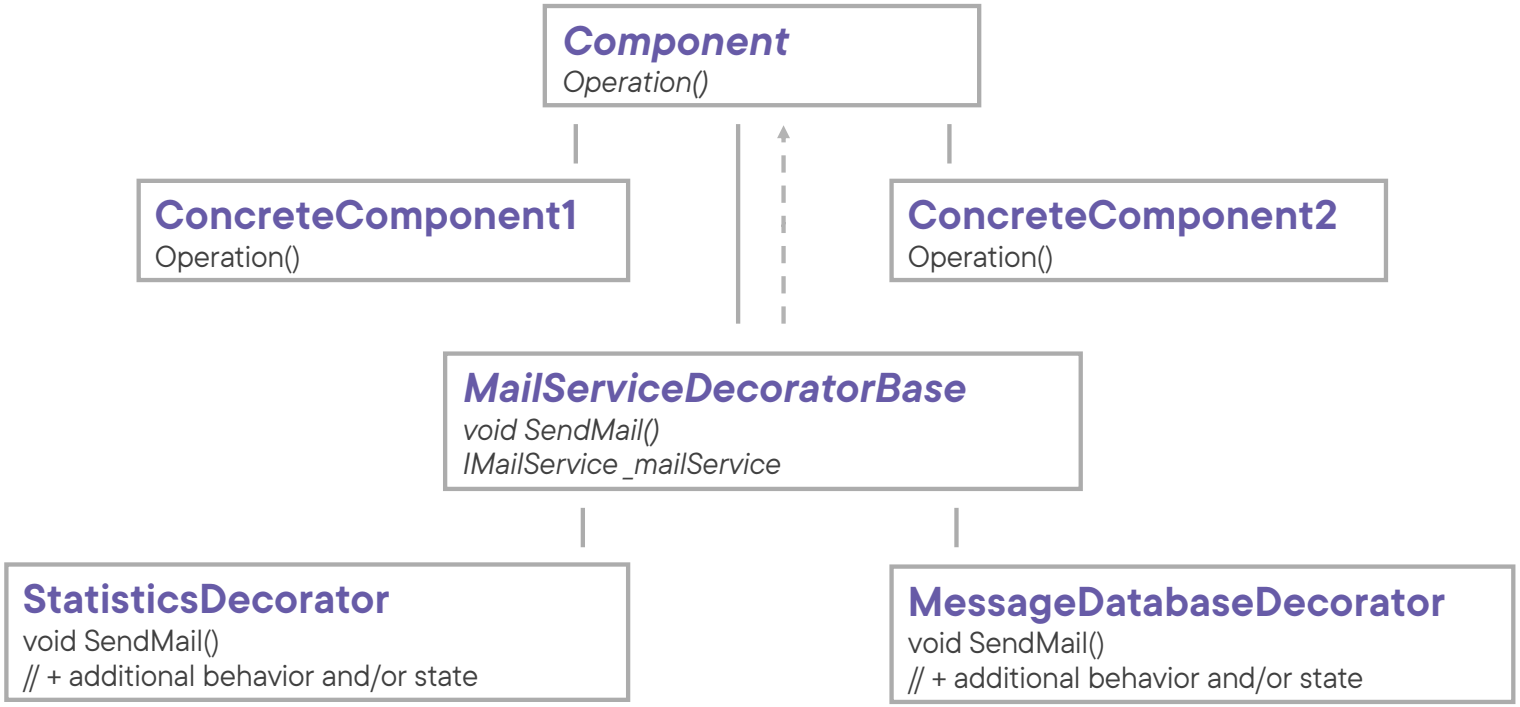


Structure of the Decorator Pattern

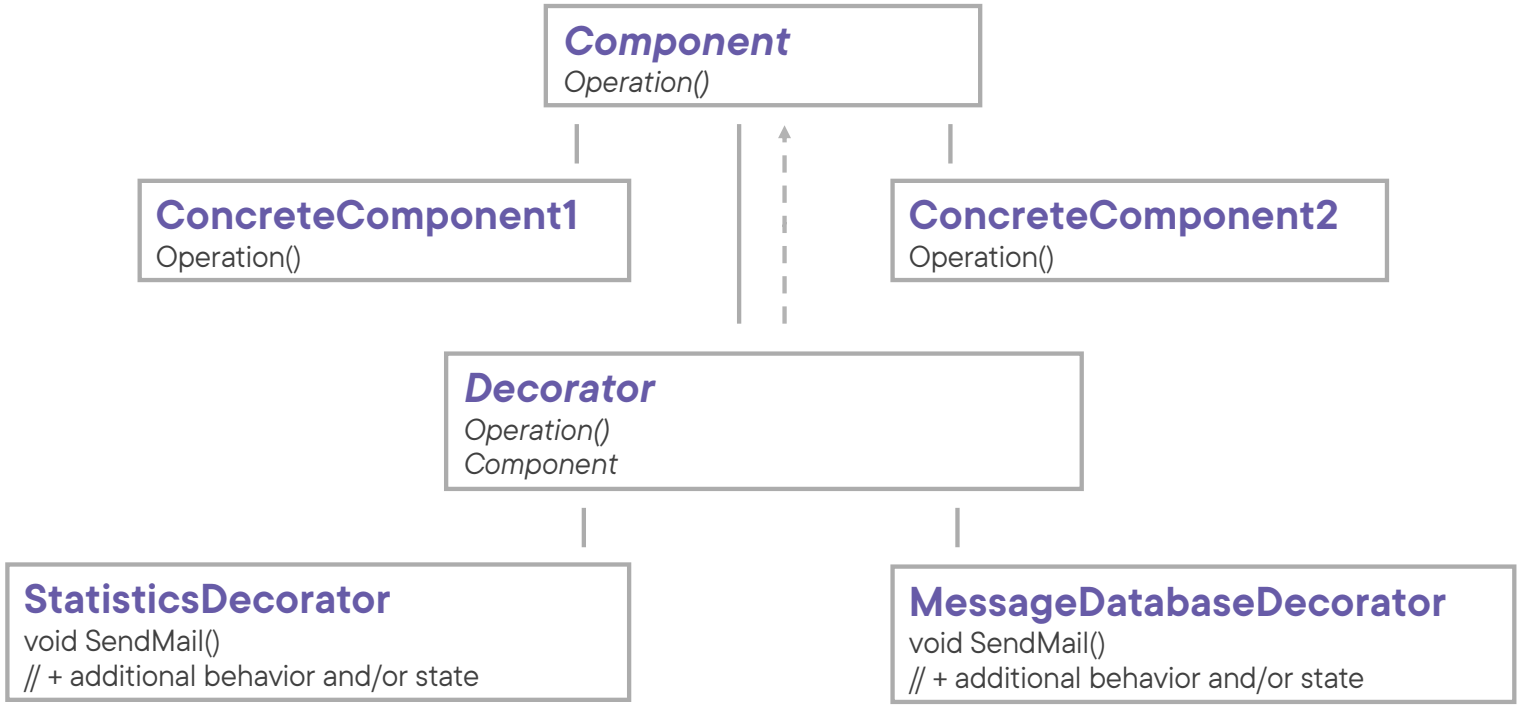


ConcreteComponent defines an object to which additional responsibilities can be attached

Structure of the Decorator Pattern

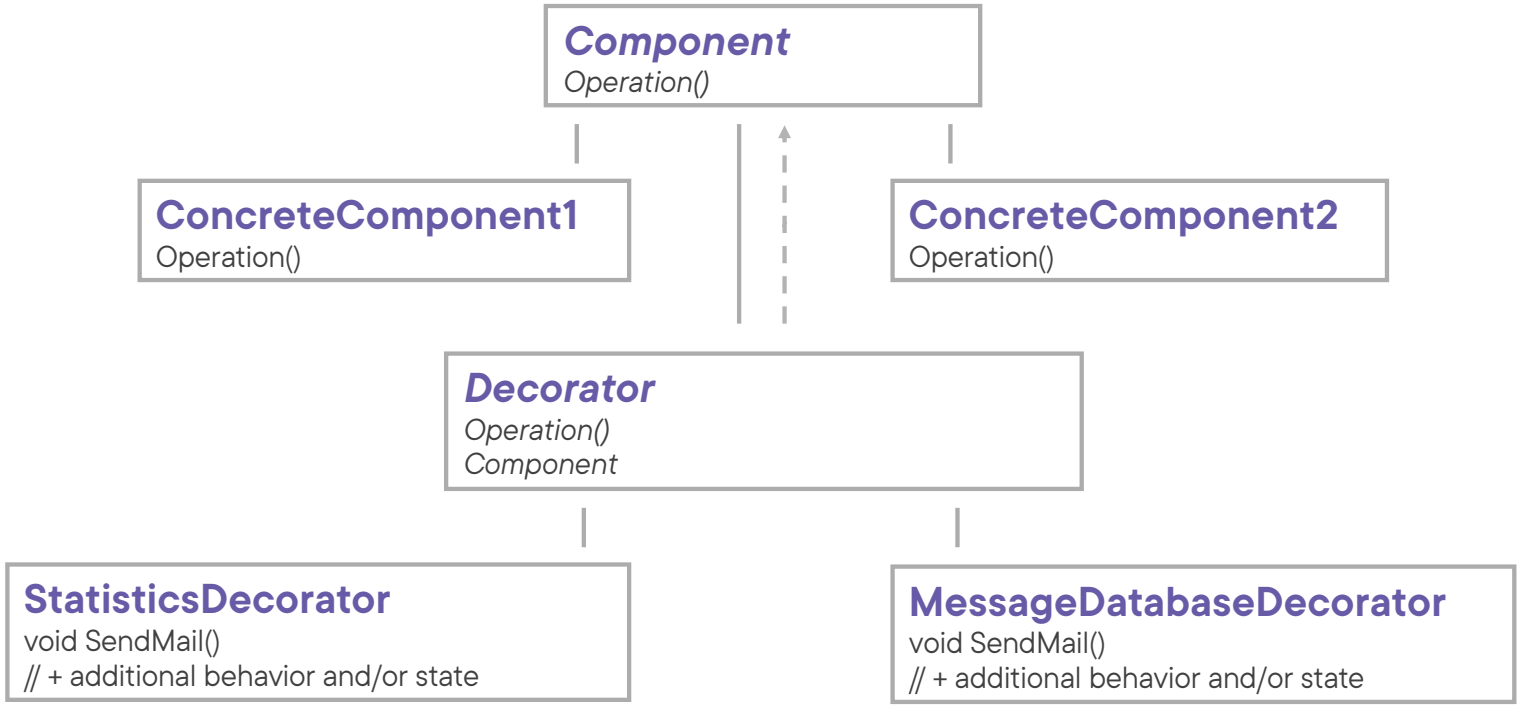


Structure of the Decorator Pattern

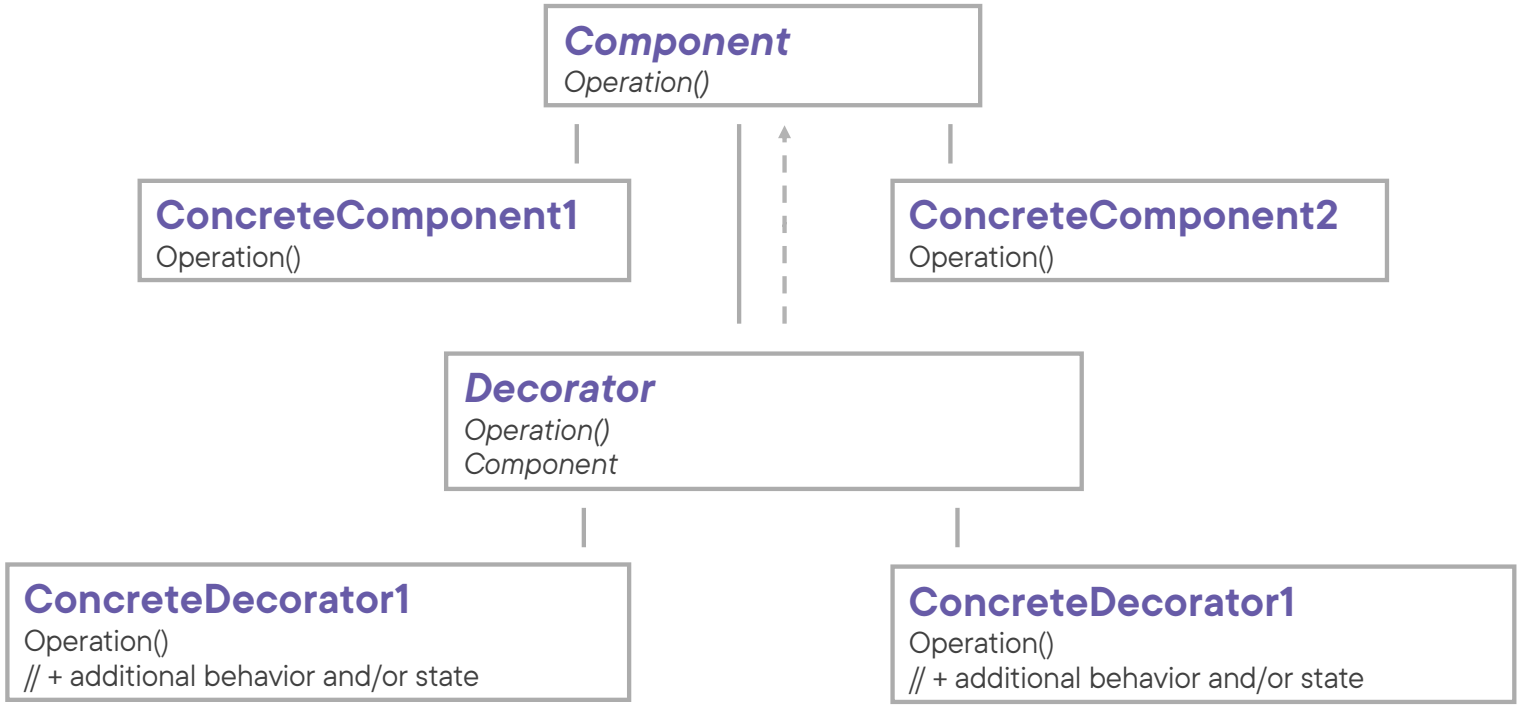


Decorator maintains a reference to a **Component** object, and defines an interface that conforms to **Component**'s interface

Structure of the Decorator Pattern

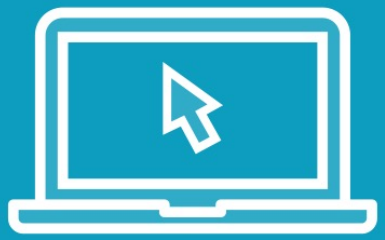


Structure of the Decorator Pattern



ConcreteDecorator adds responsibilities to the component

Demo



Implementing the decorator pattern

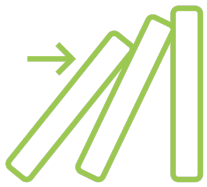
Use Cases for the Decorator Pattern

When you have a need to add responsibilities to individual objects dynamically (at runtime) without affecting other objects

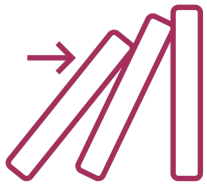
When you need to be able to withdraw responsibilities you attached to an object

When extension by subclassing is impractical or impossible

Pattern Consequences



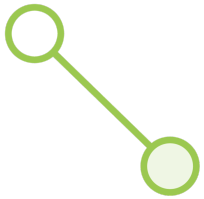
More flexible than using static inheritance via subclassing: responsibilities can be added and removed at runtime ad hoc



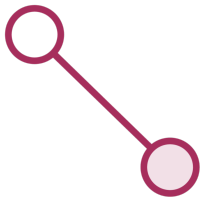
You can use the pattern to split feature-loaded classes until there's just one responsibility left per class: single responsibility principle

Increased effort is required to learn the system due to the amount of small, simple classes

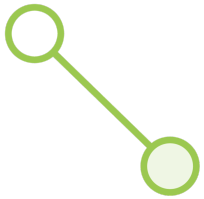
Related Patterns



Adapter
Adapter gives a new interface to an object, decorator only changes its responsibilities



Composite
Adapter can be seen as a composite with only one component



Strategy
Decorator lets you change the skin of an object, strategy lets you change its inner workings

Summary

Intent of the bridge pattern:

- Attach additional responsibilities to an object dynamically

More flexible than static inheritance through subclassing

Up Next:
Structural Pattern: Composite