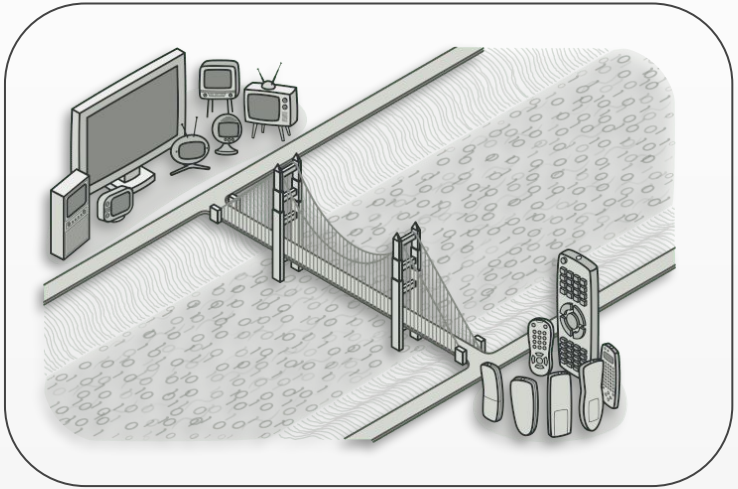
BRIDGE
Design Pattern

Farcas Amalia
Cuconu Maria
Pop David

Bridge Pattern

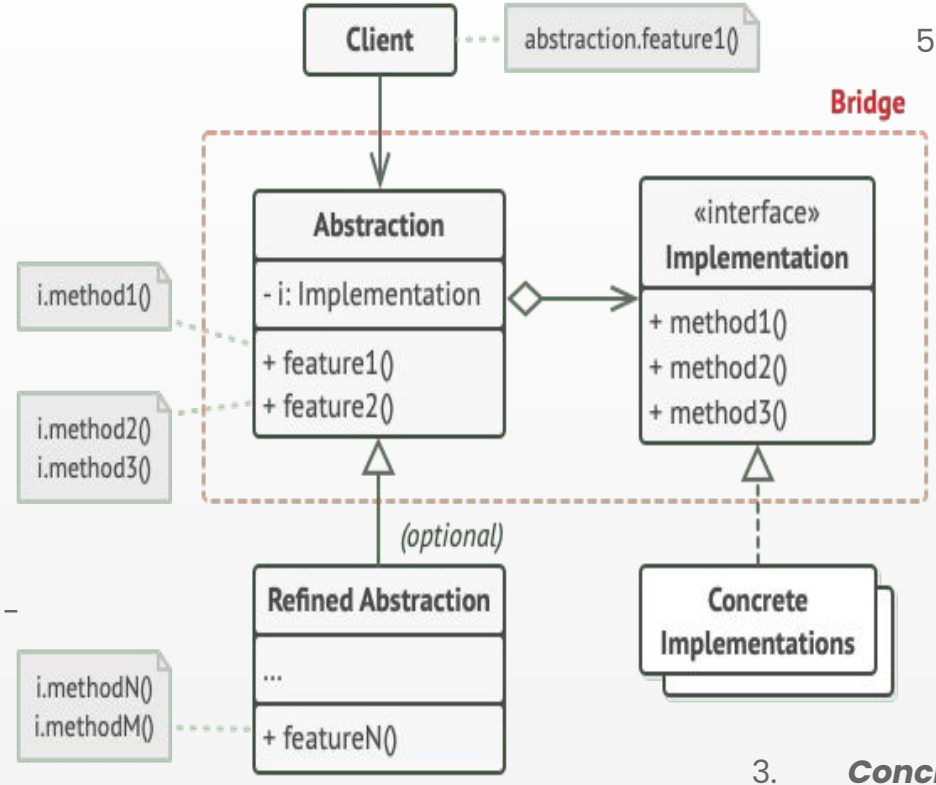
- Structural design pattern
- Lets you split a large class / a set of closely related classes into 2 separate hierarchies: abstraction & implementation. Those hierarchies can then be developed independently of each other
- Bridge can be recognized by a clear distinction between some controlling entity and several different platforms that it relies on



- Usage examples: cross-platform apps, supporting multiple types of database servers, working with several API providers of a certain kind (cloud platforms, social networks, etc.)

UML Diagram of the Bridge Pattern

1. ***Abstraction*** - provides high-level control logic.



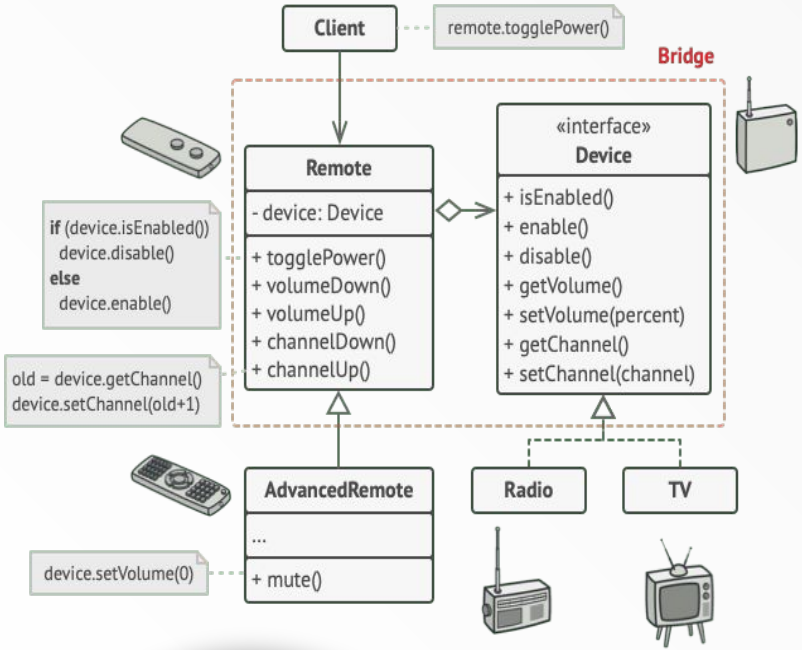
5. ***Client*** - works with the abstraction

2. ***Implementation*** - declares the interface

4. ***Refined Abstraction*** - provides variants of control logic.

3. ***Concrete implementation*** - contains platform-specific code

Bridge Implementation Example



- The Device classes act as the implementation, whereas the Remotes act as the abstraction
- The base remote control class declares a reference field that links it with a device object. All remotes work with the devices via the general device interface, which lets the same remote support multiple device types.
- You can develop the remote control classes independently from the device classes. All that's needed is to create a new remote subclass. For example, a basic remote control might only have two buttons, but you could extend it with additional features, such as an extra battery or a touchscreen.

How to

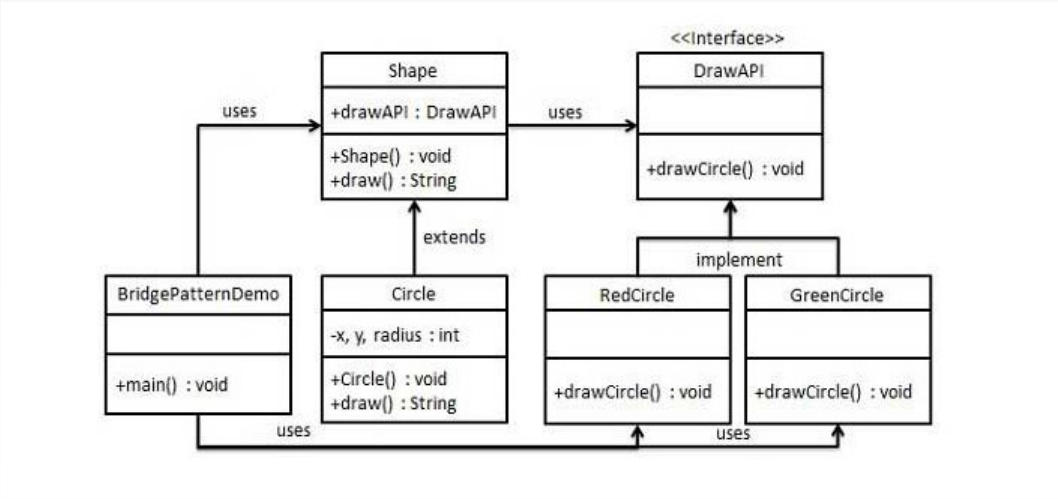
We are demonstrating use of Bridge pattern via following example in which a circle can be drawn in different colors using same abstract class method but different bridge implementer classes.

Implementation

We have a *DrawAPI* interface which is acting as a bridge implementer and concrete classes *RedCircle* and *GreenCircle* implementing the DrawAPI interface.

Shape is an abstract class and will use object of *DrawAPI*.

BridgePatternDemo, our demo class will use Shape class to draw different colored circle.



How to

Step 1 - Create bridge implementer interface

DrawAPI.java

```
public interface DrawAPI {
        public void drawCircle(int radius, int x, int y);
 }
```

Step 2 - Create concrete bridge implementer classes implementing the DrawAPI interface

RedCircle.java

```
public class RedCircle implements DrawAPI {
        @Override public void drawCircle(int radius, int x, int y) {
                System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x + "," + y + "]");
        }
}
```

GreenCircle.java

```
public class GreenCircle implements DrawAPI {
        @Override public void drawCircle(int radius, int x, int y) {
                System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x + "," + y + "]");
        }
}
```

How to

Step 3 - Create an abstract class Shape using the DrawAPI interface

Shape.java

```
public abstract class Shape {
        protected DrawAPI drawAPI;
        protected Shape(DrawAPI drawAPI){
                this.drawAPI = drawAPI;
        }
        public abstract void draw();
}
```

Step 4 - Create concrete class implementing the Shape interface.

Circle.java

```
public class Circle extends Shape {
        private int x, y, radius;
        public Circle(int x, int y, int radius, DrawAPI drawAPI) {
                super(drawAPI);
                this.x = x;
                this.y = y;
                this.radius = radius;
        }
        public void draw() {
                drawAPI.drawCircle(radius,x,y);
        }
}
```

How to

Step 5 - Use the Shape and DrawAPI classes to draw different colored circles

BridgePatternDemo.java

```
public class BridgePatternDemo {
	public static void main(String[] args) {
		Shape redCircle = new Circle(100,100, 10, new RedCircle());
		Shape greenCircle = new Circle(100,100, 10, new GreenCircle());
		redCircle.draw();
		greenCircle.draw();
	}
}
```

Step 6 – Verify output

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[ color: green, radius: 10, x: 100, 100]
```

When should you use the Bridge Pattern?

1. **When you want to divide and organize a monolithic class that has several variants of some functionality**

- *Problem*: Monolithic class complexity
- Solution: Divide into multiple class hierarchies
- Benefits:
  - Easier modifications
  - Reduced error risk

2. **When you need to extend a class in several orthogonal (independent) dimensions.**

- Strategy: Extract & delegate to separate hierarchies
- Outcome:
  - Enhanced flexibility
  - Simplified organization

PROS and CONS

PROS

1. You can create platform-independent classes and apps
2. The client code works with high-level abstractions. It isn't exposed to the platform details
3. *Open/Closed Principle*. You can introduce new abstractions and implementations independently from each other
4. *Single Responsibility Principle*. You can focus on high-level logic in the abstraction and on platform details in the implementation.

CONS

1. You might make the code more complicated by applying the pattern to a highly cohesive class.
2. If you are not adding further features, then the bridge pattern would add more classes.

Example 1

Comprehensive transaction support is among the most compelling reasons to use the Spring Framework. The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

- Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, Java Persistence API (JPA), and Java Data Objects (JDO).
- Support for declarative transaction management.
- Simpler API for programmatic transaction management than complex transaction APIs such as JTA.
- Excellent integration with Spring's data access abstractions.

- **Declarative Support**

The easiest way to use transactions in Spring is with declarative support. Here, we have **a convenience annotation available to be applied at the method or even at the class**. This simply enables global transaction for our code:

```
@PersistenceContext
EntityManager entityManager;

@Autowired
JmsTemplate jmsTemplate;

@Transactional(propagation = Propagation.REQUIRED)
public void process(ENTITY, MESSAGE) {
   entityManager.persist(ENTITY);
   jmsTemplate.convertAndSend(DESTINATION, MESSAGE);
}
```

The simple code above is sufficient to allow a save-operation in the database and a publish-operation in message queue within a JTA transaction.

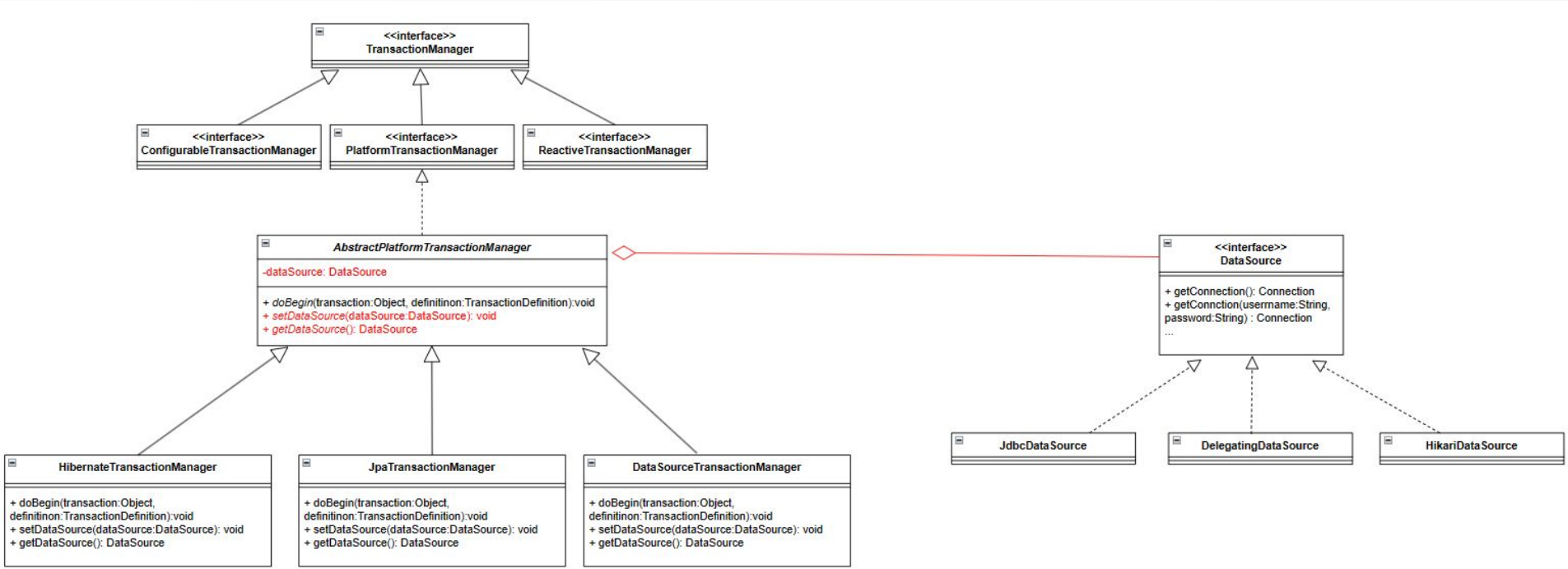
- **Programmatic Support**

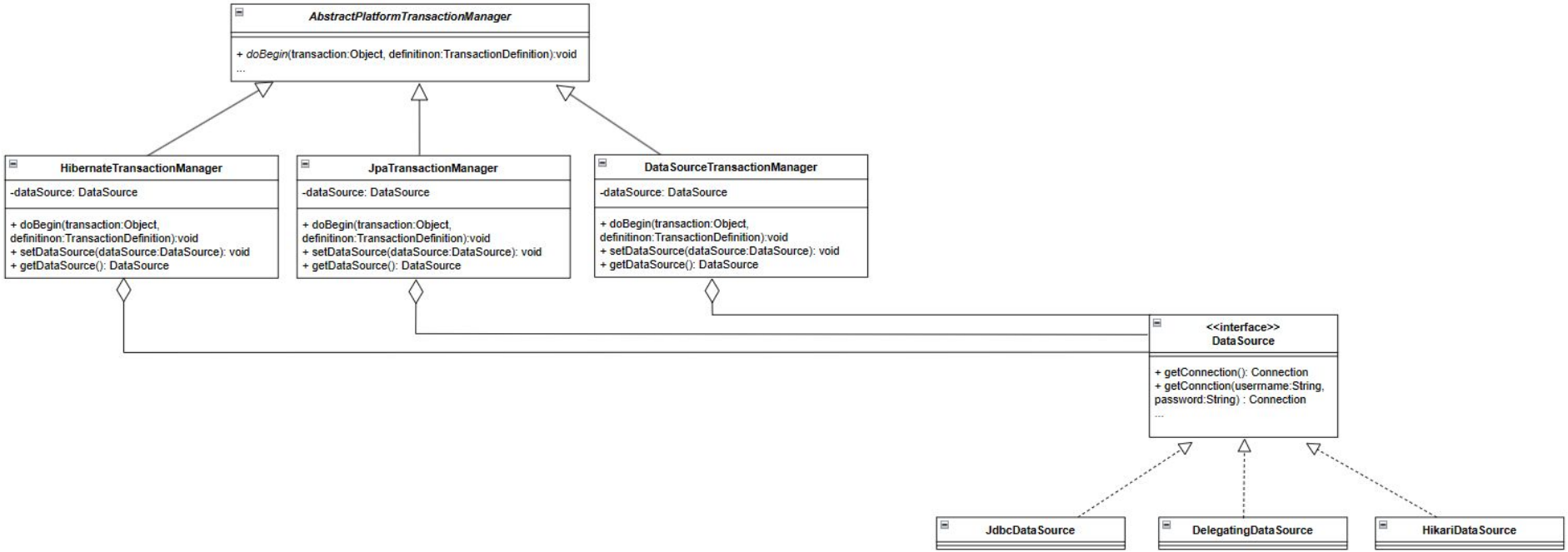
While the declarative support is quite elegant and simple, it does not offer us the **benefit of controlling the transaction boundary more precisely**. Hence, if we do have a certain need to achieve that, Spring offers programmatic support to demarcate transaction boundary:

```
@Autowired
private PlatformTransactionManager transactionManager;

public void process(ENTITY, MESSAGE) {
    TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);
    transactionTemplate.executeWithoutResult(status -> {
        entityManager.persist(ENTITY);
        jmsTemplate.convertAndSend(DESTINATION, MESSAGE);
    });
}
```

So, as we can see, we have to create a *TransactionTemplate* with the available *PlatformTransactionManager*. Then we can use the *TransactionTemplete* to process a bunch of statements within a global transaction.


<<interface>>
TransactionManager
<<interface>>
ConfigurableTransactionManager
<<interface>>
PlatformTransactionManager
<<interface>>
ReactiveTransactionManager
AbstractPlatformTransactionManager
-dataSource: DataSource
+ doBegin(transaction:Object, definitinon:TransactionDefinition):void
+ setDataSource(dataSource:DataSource): void
+ getDataSource(): DataSource
<<interface>>
DataSource
+ getConnection(): Connection
+ getConnction(usermname:String, password:String) : Connection
...
HibernateTransactionManager
+ doBegin(transaction:Object, definitinon:TransactionDefinition):void
+ setDataSource(dataSource:DataSource): void
+ getDataSource(): DataSource
JpaTransactionManager
+ doBegin(transaction:Object, definitinon:TransactionDefinition):void
+ setDataSource(dataSource:DataSource): void
+ getDataSource(): DataSource
DataSourceTransactionManager
+ doBegin(transaction:Object, definitinon:TransactionDefinition):void
+ setDataSource(dataSource:DataSource): void
+ getDataSource(): DataSource
JdbcDataSource
DelegatingDataSource
HikariDataSource


AbstractPlatformTransactionManager
+ doBegin(transaction:Object, definitinon:TransactionDefinition):void
...
HibernateTransactionManager
-dataSource: DataSource
+ doBegin(transaction:Object, definitinon:TransactionDefinition):void
+ setDataSource(dataSource:DataSource): void
+ getDataSource(): DataSource
JpaTransactionManager
-dataSource: DataSource
+ doBegin(transaction:Object, definitinon:TransactionDefinition):void
+ setDataSource(dataSource:DataSource): void
+ getDataSource(): DataSource
DataSourceTransactionManager
-dataSource: DataSource
+ doBegin(transaction:Object, definitinon:TransactionDefinition):void
+ setDataSource(dataSource:DataSource): void
+ getDataSource(): DataSource
<<interface>>
DataSource
+ getConnection(): Connection
+ getConnction(usermame:String, password:String) : Connection
...
JdbcDataSource
DelegatingDataSource
HikariDataSource

```
//HibernateTransactionManager.doBegin()
ConnectionHolder conHolder = new ConnectionHolder(
            () → sessionToUse.getJdbcCoordinator().getLogicalConnection().getPhysicalConnection());
...
TransactionSynchronizationManager.bindResource(getDataSource(), conHolder);


//JpaTransactionManager.doBegin()
ConnectionHandle conHandle = getJpaDialect().getJdbcConnection(em, definition.isReadOnly());
ConnectionHolder conHolder = new ConnectionHolder(conHandle);
...
TransactionSynchronizationManager.bindResource(getDataSource(), conHolder);


//DataSourceTransactionManager.doBegin()
DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
Connection newCon = obtainDataSource().getConnection();
txObject.setConnectionHolder(new ConnectionHolder(newCon), true);
...
TransactionSynchronizationManager.bindResource(obtainDataSource(), txObject.getConnectionHolder());
```

```java
@Configuration
public class TransactionManagerConfig {

    @Bean
    @Primary
    public DataSource dataSourceHikari() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setJdbcUrl("jdbc:hikariYourDatabaseUrl");
        dataSource.setUsername("yourUsername");
        dataSource.setPassword("yourPassword");
        return dataSource;
    }

    @Bean
    public DataSource dataSourceJdbc() {
        JdbcDataSource dataSource = new JdbcDataSource();
        dataSource.setURL("jdbc:h2:mem:testdb");
        dataSource.setUser("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public DataSourceTransactionManager transactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

}
```

Example 2

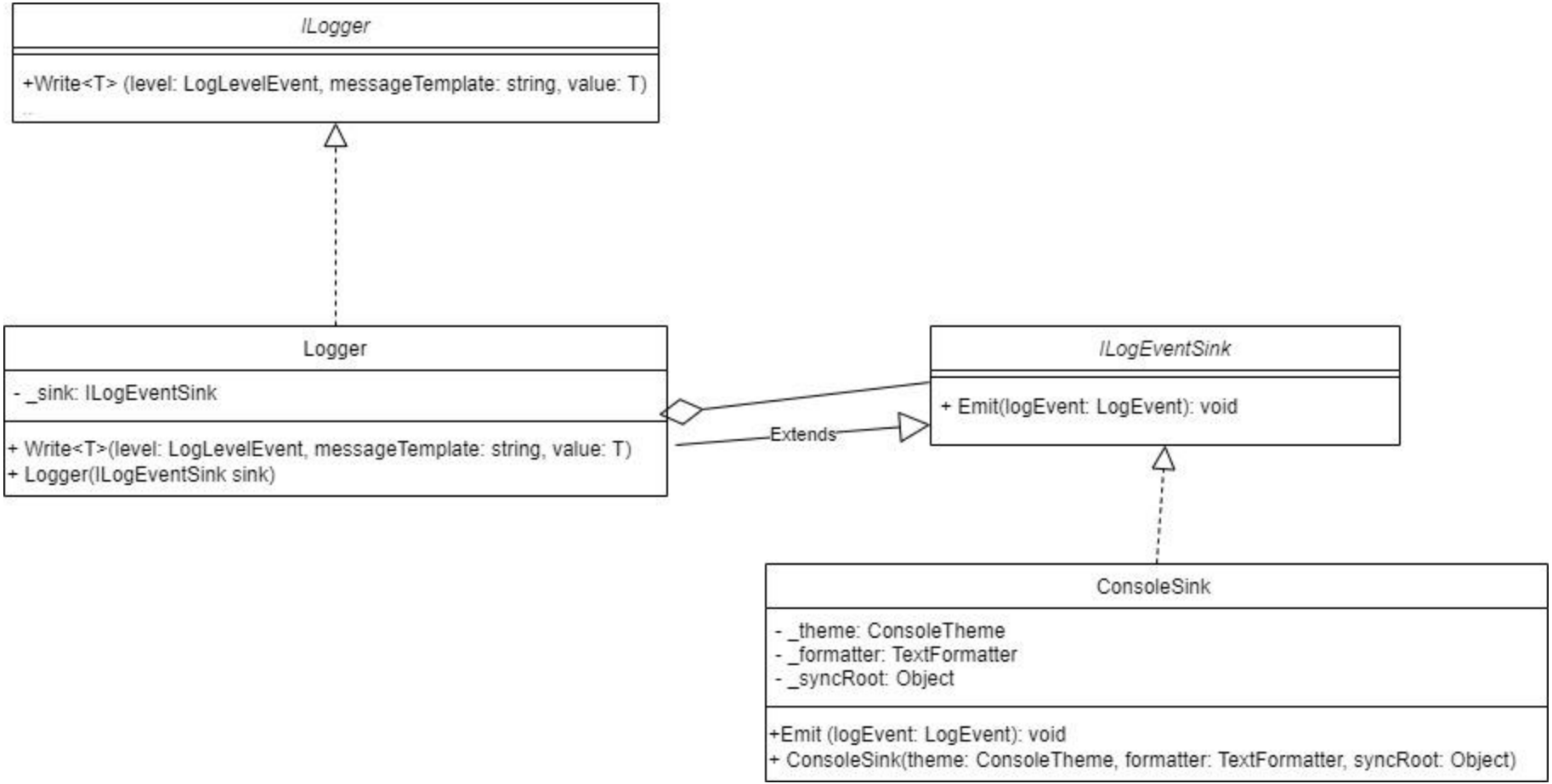
Logging in .NET

Using Serilog

🚀 Logging in .NET using Serilog package

Logging is a way of display log messages on different output environments(known as Sinks).

For the second example we will look at how bridge pattern is implemented in the Serilog package.


ILogger
+Write<T> (level: LogLevelEvent, messageTemplate: string, value: T)
Logger
- _sink: ILogEventSink
+ Write<T>(level: LogLevelEvent, messageTemplate: string, value: T)
+ Logger(ILogEventSink sink)
Extends
ILogEventSink
+ Emit(logEvent: LogEvent): void
ConsoleSink
- _theme: ConsoleTheme
- _formatter: TextFormatter
- _syncRoot: Object
+Emit (logEvent: LogEvent): void
+ ConsoleSink(theme: ConsoleTheme, formatter: TextFormatter, syncRoot: Object)

```
using Serilog;

namespace ConsoleApp2;

class Program
{
    static void Main(string[] args)
    {
        //builder
        Log.Logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
            .WriteTo.Console() // LoggerConfiguration
            .CreateLogger(); // Logger

        Log.Information( messageTemplate: "Hello, Serilog!");
    }
}
```

Thanks!

Do you have any questions?

Resources

Information & Examples

- Refactoring guru [https://refactoring.guru/design-patterns/bridge]
- Bridge Design Pattern - Derek Bananas [https://www.youtube.com/watch?v=9jlgSslfh_8]
- Tutorials Point [https://www.tutorialspoint.com/design_pattern/bridge_pattern.htm]
- Spring.io [https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/transaction.html]
- Baeldung [https://www.baeldung.com/java-transactions]
- Spring framework github [https://github.com/spring-projects/spring-framework]
- Serilog [https://github.com/serilog]

Photos

- Medium [https://medium.com/javarevisited/transactional-annotation-in-spring-framework-d571e91bf6bb]
- Billy Okeyo [https://www.billyokeyo.com/posts/logging-in-serilog/]