BU BOSTON UNIVERSITY

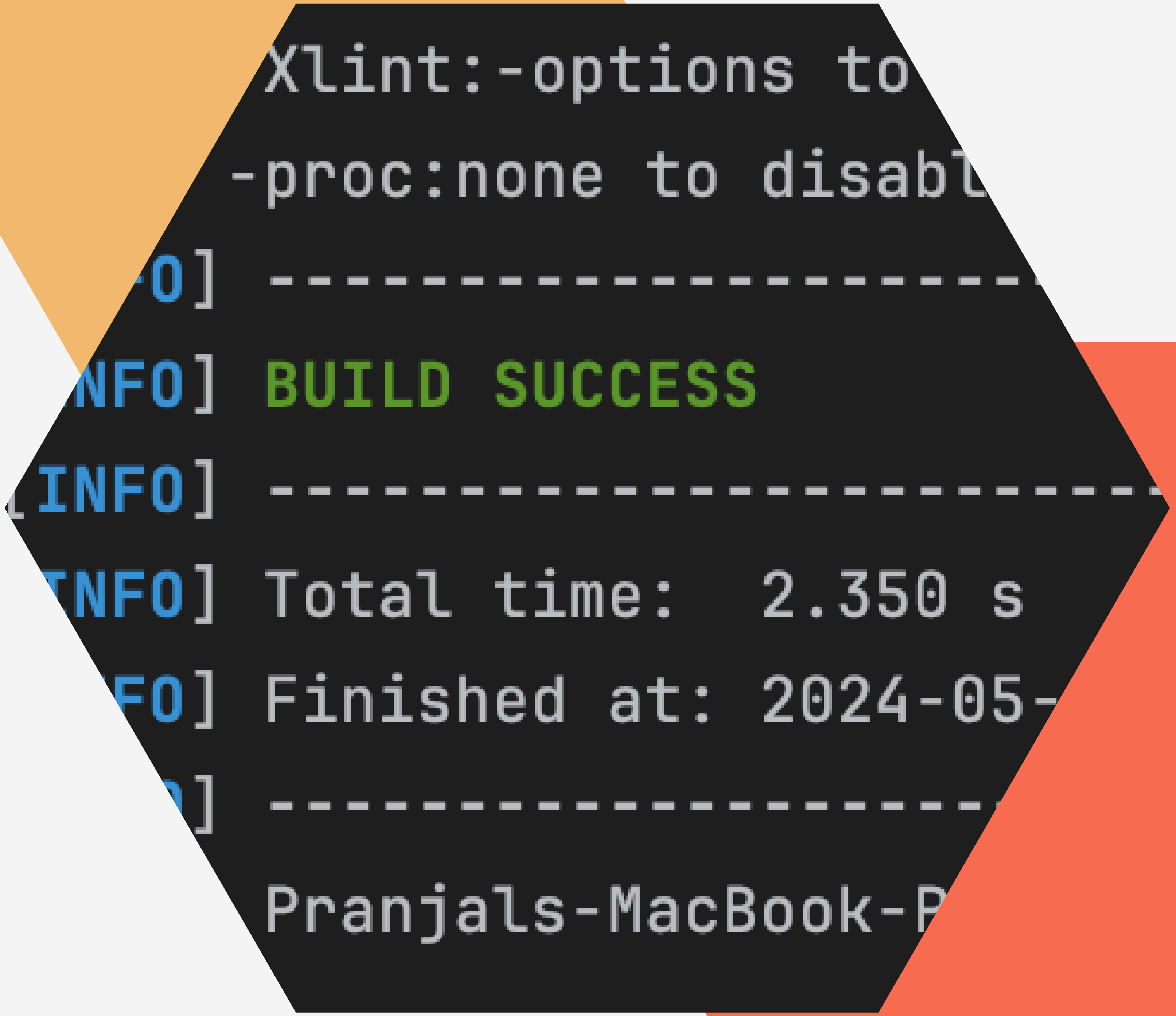
Multi-Format Report Generator Using the Builder Pattern

Pranjal Ekhande (BUID: U68954134)

BU BOSTON UNIVERSITY

Introduction

- A tool to generate reports from data sources in multiple formats (PDF, HTML, Excel)
- Allow users to specify parts of the report like charts, tables, and text sections
- Generate reports based on user preferences and requirements



```
 Xlint:-options to
-proc:none to disabl
O] ------------------------
NFO] BUILD SUCCESS
[INFO] --------------------------
INFO] Total time:  2.350 s
FO] Finished at: 2024-05-
] ------------------------
 Pranjals-MacBook-P
```

Design Patterns

Builder Pattern

- Construct complex reports step-by-step
- Specify parts of the report (charts, tables, text sections)

Decorator Pattern

- Add dynamic formatting and styling options to reports
- Wrap base reports with decorators for font styles, colors, etc.

Singleton Pattern

- Ensures global access to configuration settings for report generation

Project Structure

Builder Module

ReportBuilder interface, ReportBuilderImpl

Models Module

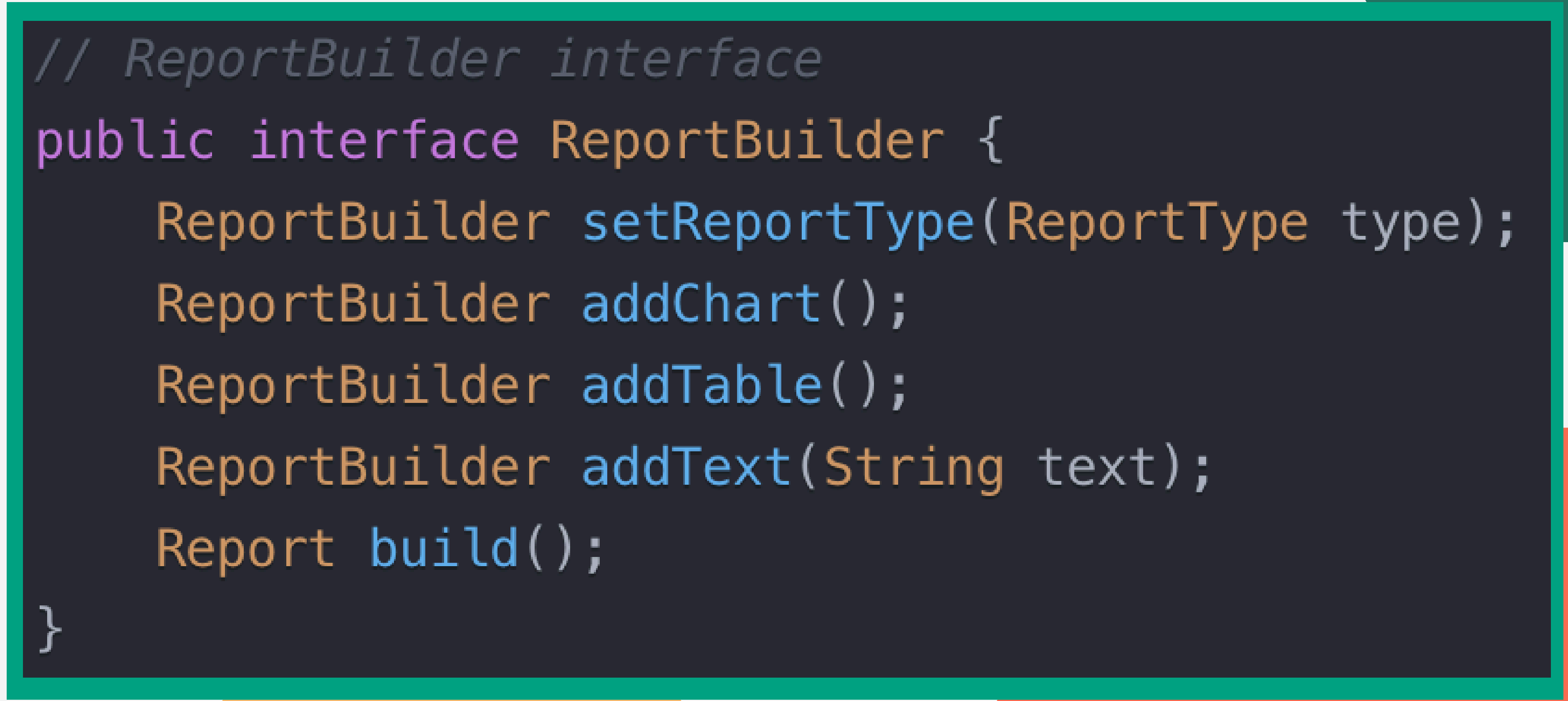
Report class, ReportDecorator (abstract), FontDecoratorReport, ColorDecoratorReport

Services Module

ReportGenerator class

Builder Pattern Implementation

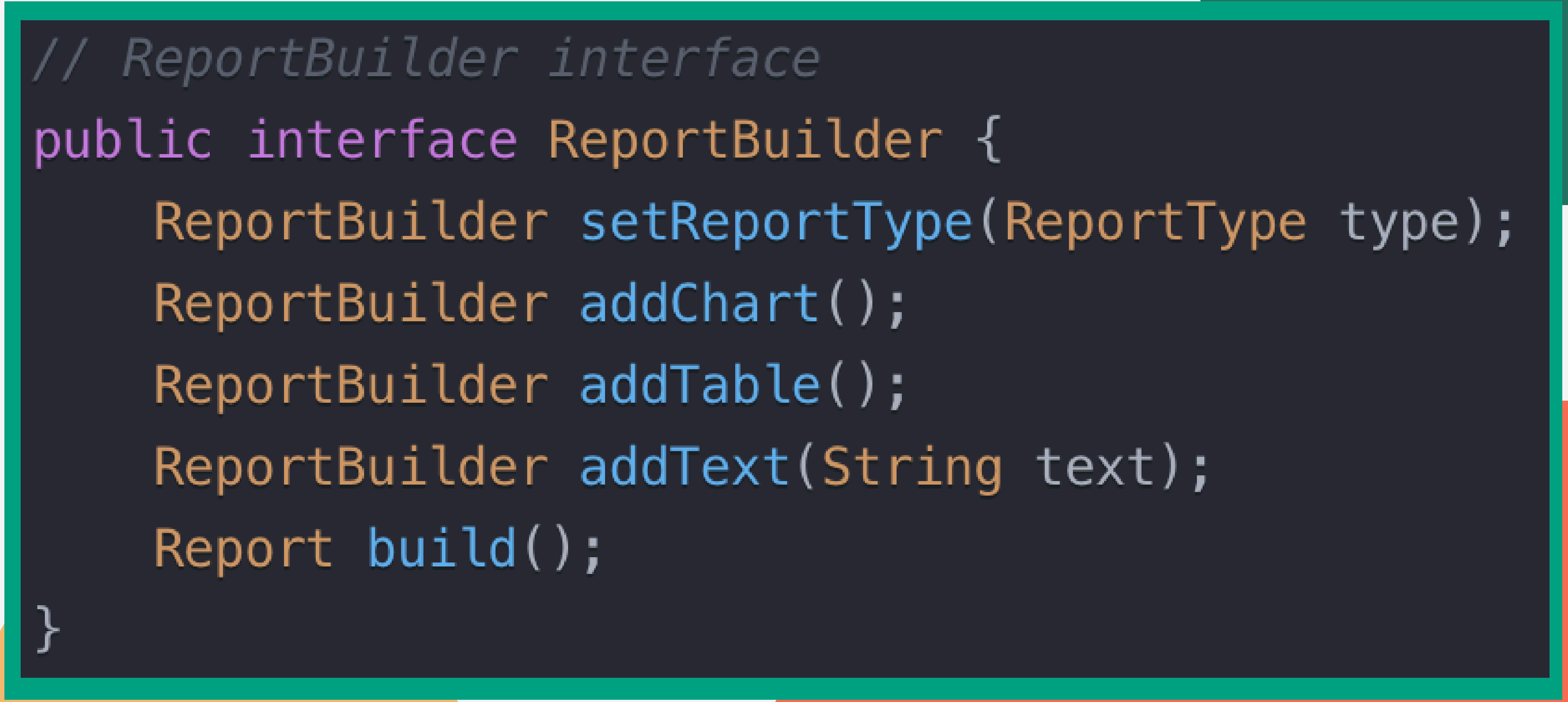
- Builder Pattern uses interface method
- ReportBuilder has its implementation technique ReportBuilderImpl



```java
// ReportBuilder interface
public interface ReportBuilder {
    ReportBuilder setReportType(ReportType type);
    ReportBuilder addChart();
    ReportBuilder addTable();
    ReportBuilder addText(String text);
    Report build();
}
```

Builder Pattern Significance

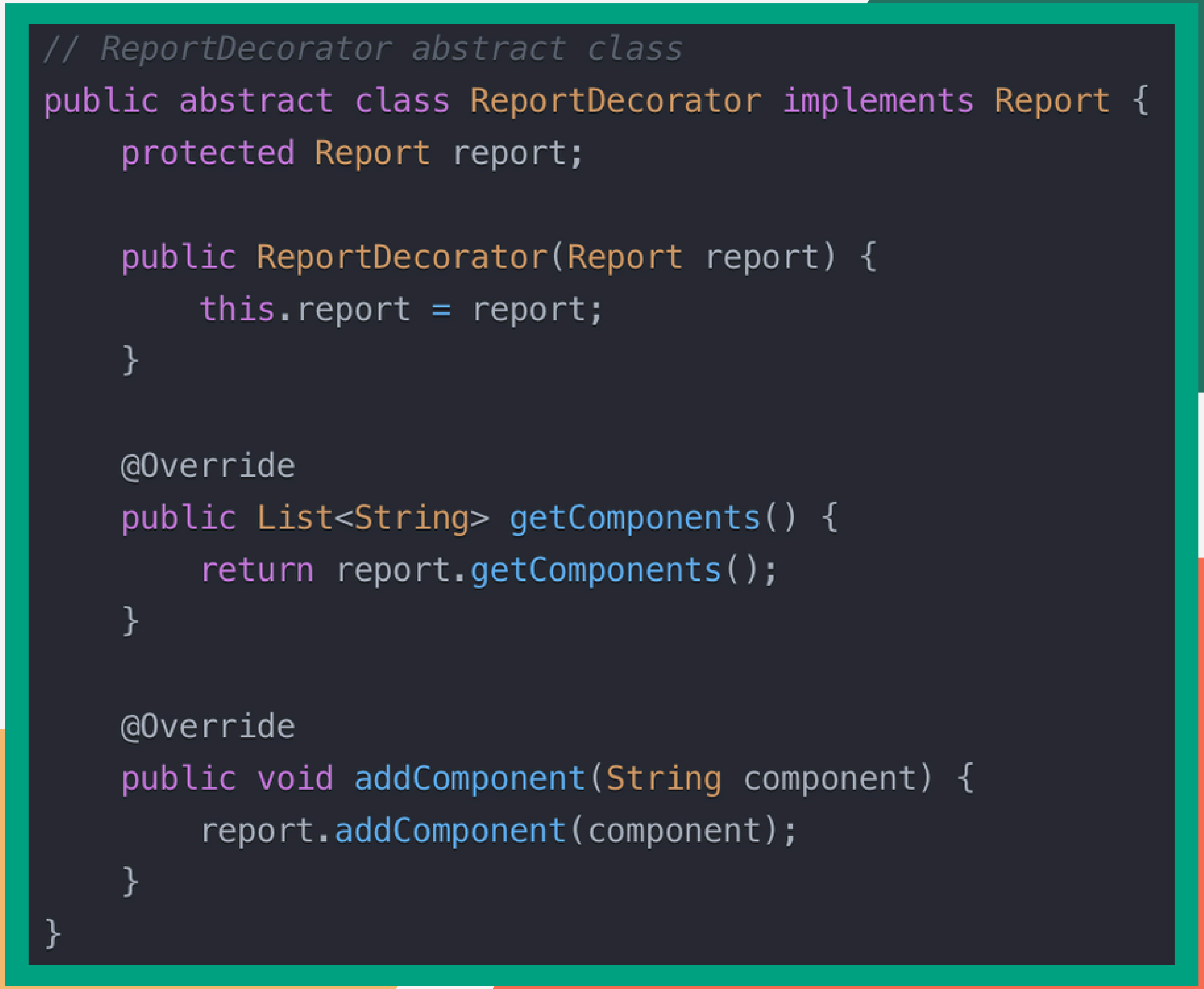
- Separates the construction process from the representation
- Allows for different representations (PDF, HTML, Excel) using the same construction process
- Provides flexibility and extensibility for adding new report components
- Promotes code reusability and maintainability



```java
// ReportBuilder interface
public interface ReportBuilder {
    ReportBuilder setReportType(ReportType type);
    ReportBuilder addChart();
    ReportBuilder addTable();
    ReportBuilder addText(String text);
    Report build();
}
```

Decorator Pattern Implementation

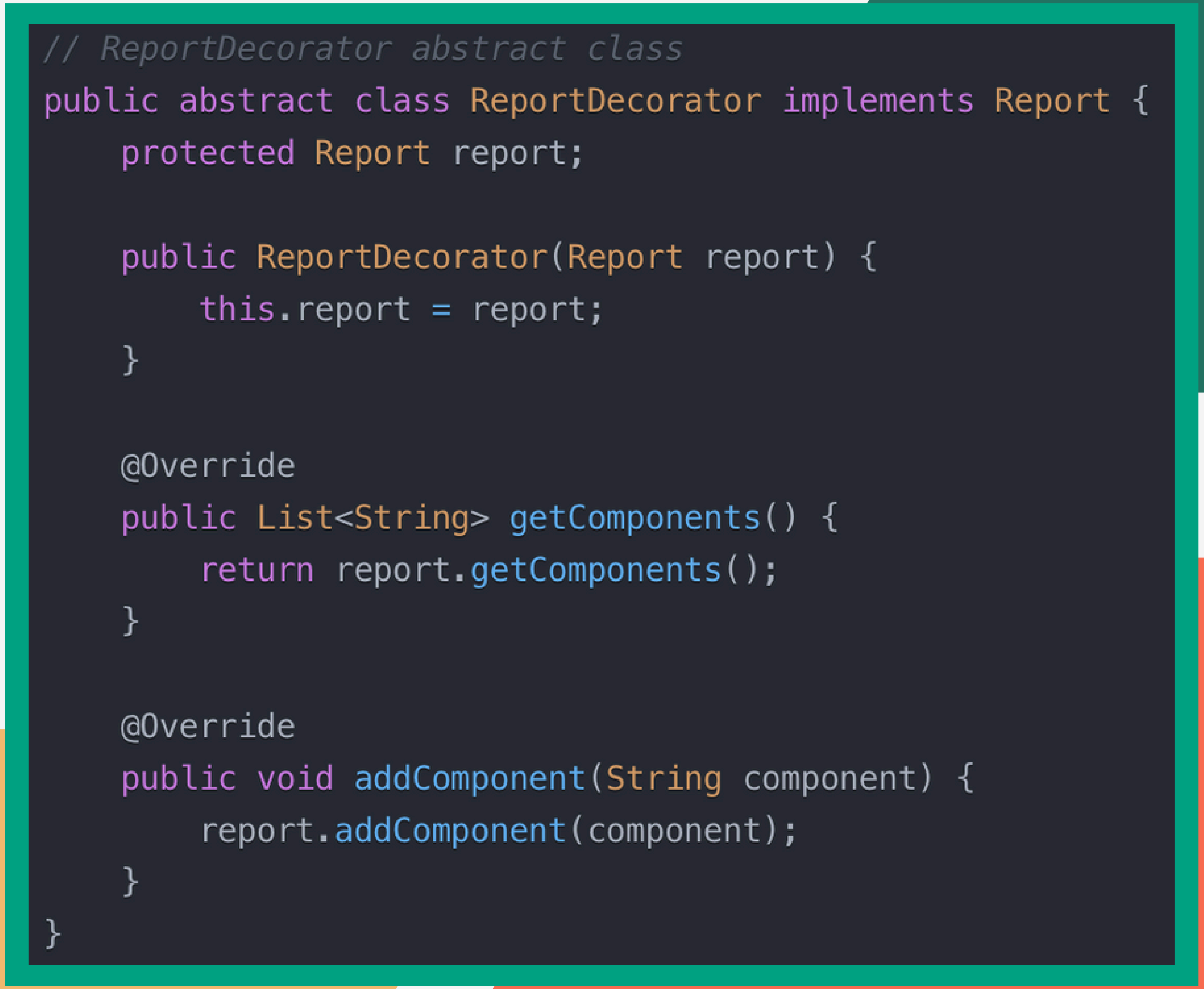
- Decorator Pattern uses abstract class ReportDecorator method
- FontDecoratorReport, and ColorDecoratorReport These two methods are implemented under Decorator Pattern



```java
// ReportDecorator abstract class
public abstract class ReportDecorator implements Report {
    protected Report report;

    public ReportDecorator(Report report) {
        this.report = report;
    }

    @Override
    public List<String> getComponents() {
        return report.getComponents();
    }

    @Override
    public void addComponent(String component) {
        report.addComponent(component);
    }
}
```

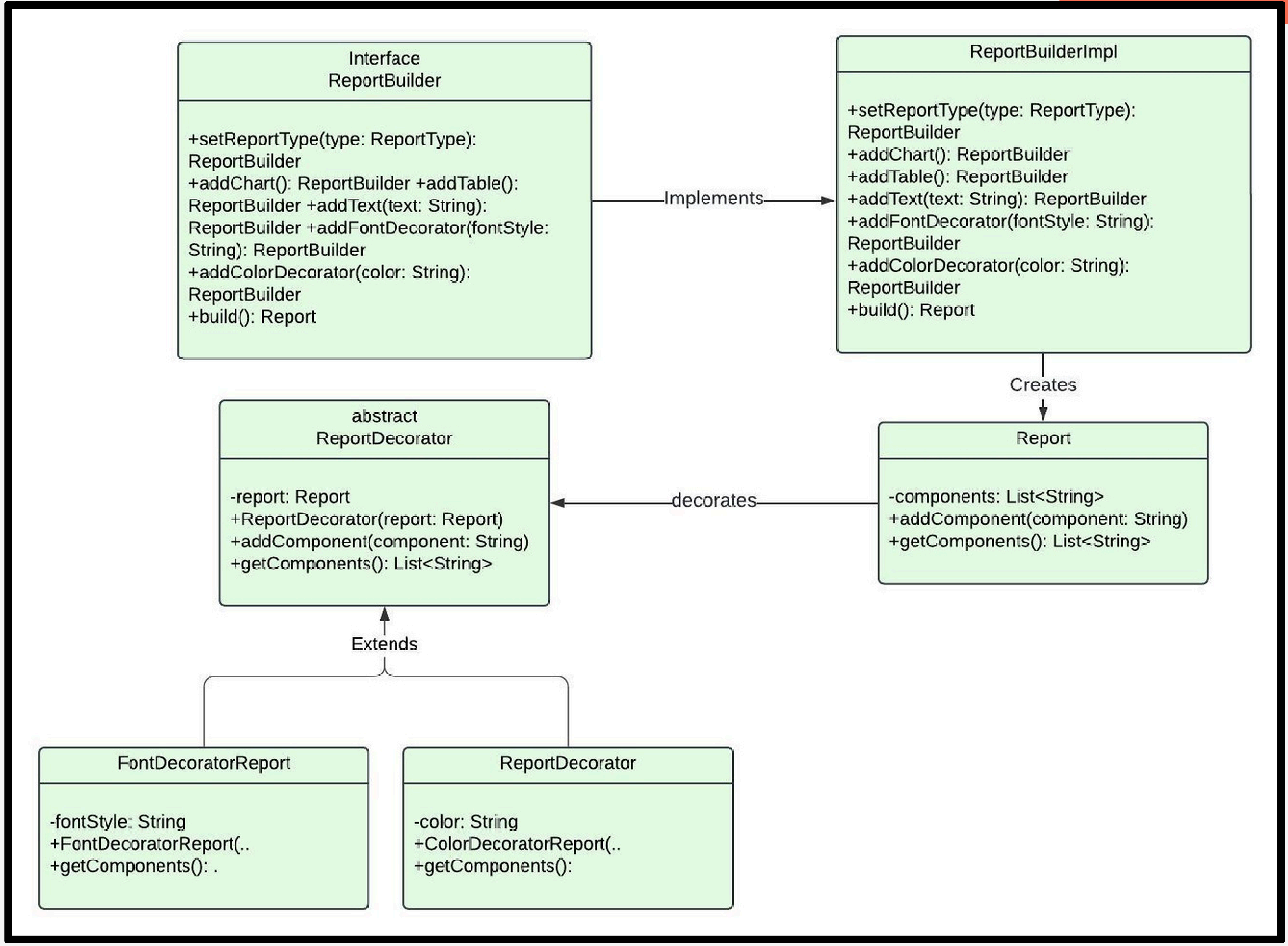
Decorator Pattern Significance

- Follows the Open-Closed Principle (open for extension, closed for modification)
- Allows adding new formatting and styling capabilities without modifying the core Report class
- Provides flexibility and extensibility for adding new decorators
- Promotes code reusability and maintainability
- Follows the Single Responsibility Principle (each decorator has a single responsibility)



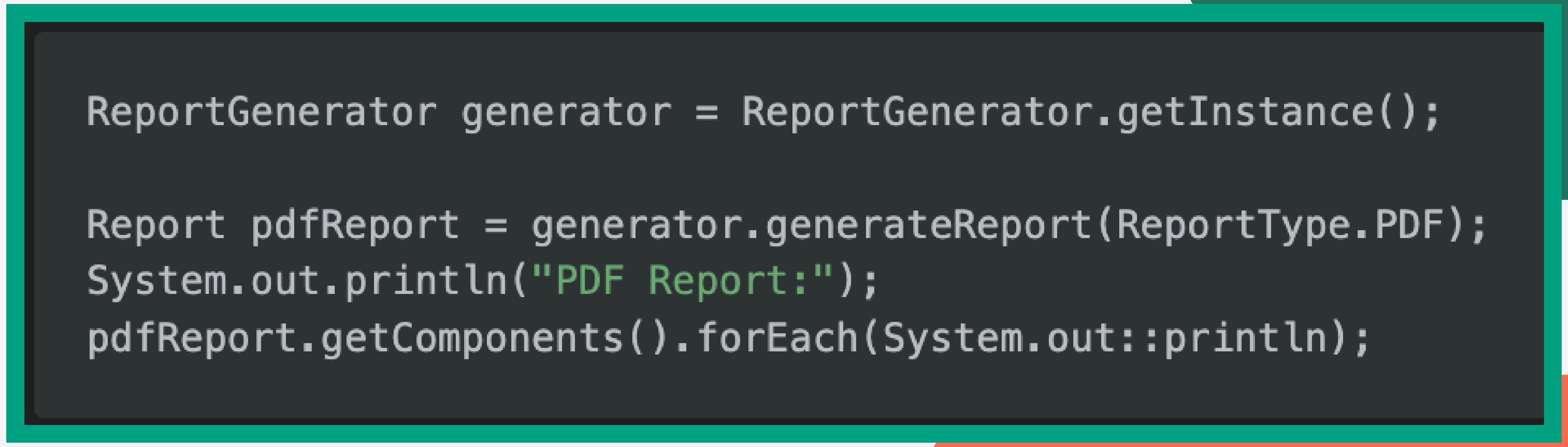
```java
// ReportDecorator abstract class
public abstract class ReportDecorator implements Report {
    protected Report report;

    public ReportDecorator(Report report) {
        this.report = report;
    }

    @Override
    public List<String> getComponents() {
        return report.getComponents();
    }

    @Override
    public void addComponent(String component) {
        report.addComponent(component);
    }
}
```

UML Digram


Interface
ReportBuilder
+setReportType(type: ReportType): ReportBuilder
+addChart(): ReportBuilder +addTable(): ReportBuilder +addText(text: String): ReportBuilder +addFontDecorator(fontStyle: String): ReportBuilder
+addColorDecorator(color: String): ReportBuilder
+build(): Report
Implements
ReportBuilderImpl
+setReportType(type: ReportType): ReportBuilder
+addChart(): ReportBuilder
+addTable(): ReportBuilder
+addText(text: String): ReportBuilder
+addFontDecorator(fontStyle: String): ReportBuilder
+addColorDecorator(color: String): ReportBuilder
+build(): Report
Creates
Report
-components: List<String>
+addComponent(component: String)
+getComponents(): List<String>
decorates
abstract
ReportDecorator
-report: Report
+ReportDecorator(report: Report)
+addComponent(component: String)
+getComponents(): List<String>
Extends
FontDecoratorReport
-fontStyle: String
+FontDecoratorReport(..
+getComponents(): .
ReportDecorator
-color: String
+ColorDecoratorReport(..
+getComponents():

Usage Example

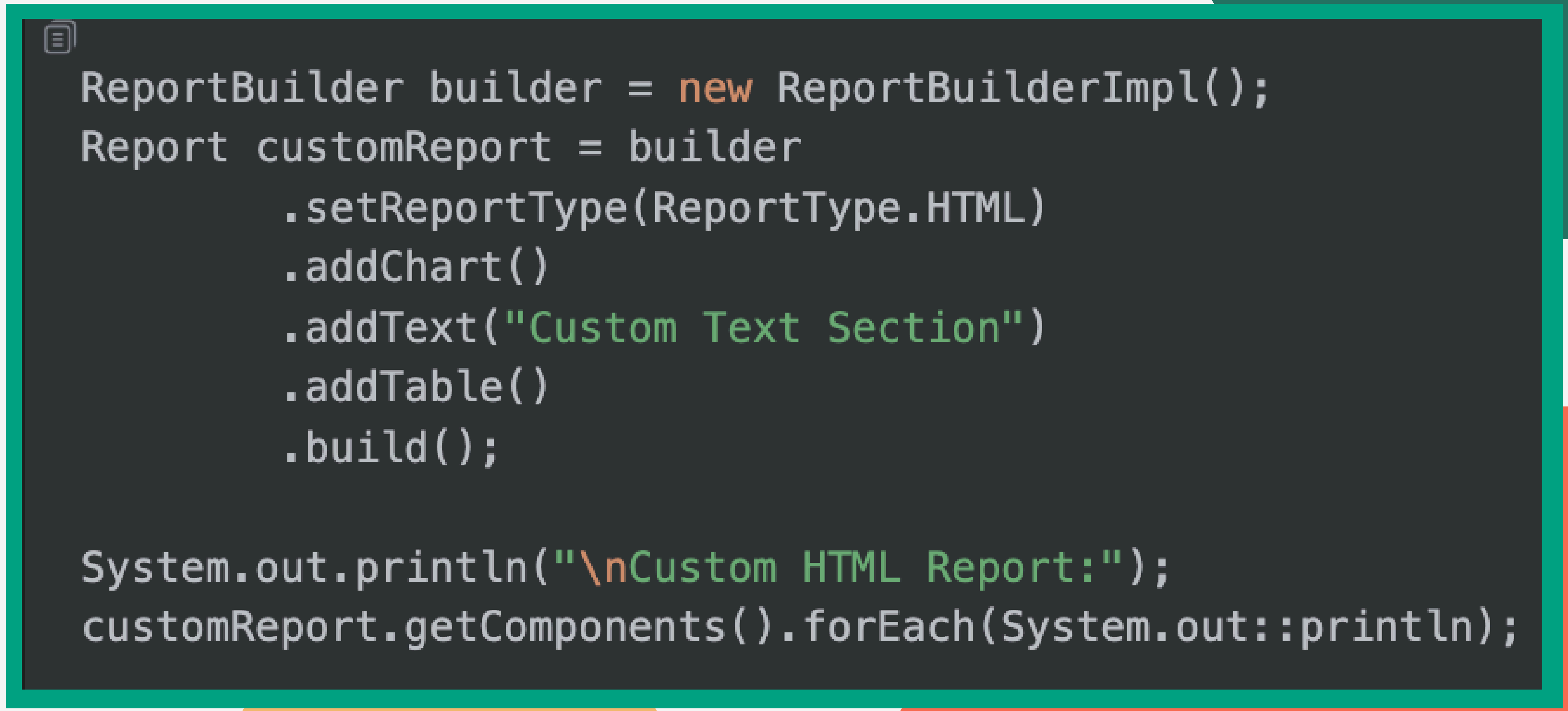
- Using ReportGenerator to generate a PDF report



```java
ReportGenerator generator = ReportGenerator.getInstance();

Report pdfReport = generator.generateReport(ReportType.PDF);
System.out.println("PDF Report:");
pdfReport.getComponents().forEach(System.out::println);
```

Usage Example

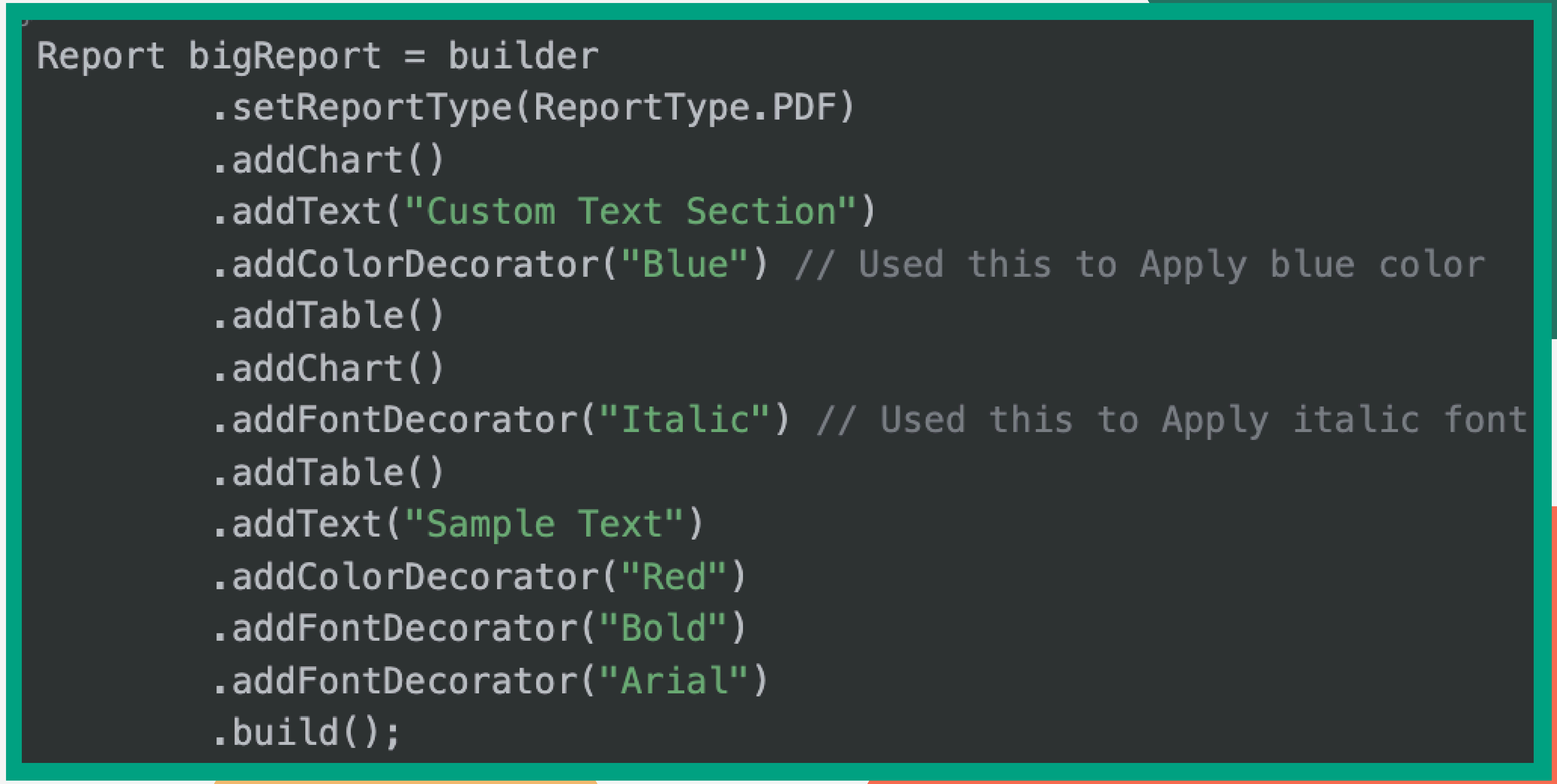
- Using ReportBuilder directly to build a custom HTML report



```java
ReportBuilder builder = new ReportBuilderImpl();
Report customReport = builder
        .setReportType(ReportType.HTML)
        .addChart()
        .addText("Custom Text Section")
        .addTable()
        .build();

System.out.println("\nCustom HTML Report:");
customReport.getComponents().forEach(System.out::println);
```

Usage Example

- Using ReportBuilder to add font and color decorators



```
Report bigReport = builder
        .setReportType(ReportType.PDF)
        .addChart()
        .addText("Custom Text Section")
        .addColorDecorator("Blue") // Used this to Apply blue color
        .addTable()
        .addChart()
        .addFontDecorator("Italic") // Used this to Apply italic font
        .addTable()
        .addText("Sample Text")
        .addColorDecorator("Red")
        .addFontDecorator("Bold")
        .addFontDecorator("Arial")
        .build();
```

Thank You!