29 Behavioral: Template Method Pattern — Report-Generation Assignment

Create a mini framework that **standardizes** the steps to build a report while letting each format plug‐in its own details.  
 Every class must have clear JavaDoc. End with a short **Reflection** on upsides & trade-offs.

.

├── analysis

│ └── template\_need.md ← why template > copy-pasted workflows

├── src/main/java

│ └── reports

│ ├── template

│ │ └── ReportGenerator.java

│ ├── pdf

│ │ └── PDFReportGenerator.java

│ ├── csv

│ │ └── CSVReportGenerator.java

│ └── TemplateDemo.java

├── src/test/java/reports

│ ├── PDFFlowTest.java

│ ├── CSVFlowTest.java

│ └── CustomHookTest.java

├── reflection.md

└── README.md

#### **1 ReportGenerator.java (abstract template)**

package reports.template;

/\*\*

\* <p>Defines the <em>skeleton</em> of report generation.</p>

\* Steps are fixed by {@link #generateReport()} while hooks are left

\* abstract for subclasses to customise.

\*/

public abstract class ReportGenerator {

/\*\* The <b>template method</b> – cannot be overridden. \*/

public final void generateReport() {

fetchData();

processData();

formatReport();

saveReport(); // shared default

}

/\* ===== hooks to override ===== \*/

protected abstract void fetchData();

protected abstract void processData();

protected abstract void formatReport();

/\*\* Shared step – may be overridden as a (protected) hook if needed. \*/

protected void saveReport() {

System.out.println("Saving the report to disk.");

}

}

#### **2 Concrete subclasses**

package reports.pdf;

import reports.template.ReportGenerator;

/\*\* Generates a PDF version of the report. \*/

public class PDFReportGenerator extends ReportGenerator {

@Override protected void fetchData() { System.out.println("Fetching data for PDF…"); }

@Override protected void processData() { System.out.println("Processing data for PDF…"); }

@Override protected void formatReport(){ System.out.println("Formatting as PDF."); }

}

package reports.csv;

import reports.template.ReportGenerator;

/\*\* Generates a CSV version of the report. \*/

public class CSVReportGenerator extends ReportGenerator {

@Override protected void fetchData() { System.out.println("Fetching data for CSV…"); }

@Override protected void processData() { System.out.println("Processing data for CSV…"); }

@Override protected void formatReport(){ System.out.println("Formatting as CSV."); }

}

#### **3 Demo client**

package reports;

import reports.pdf.PDFReportGenerator;

import reports.csv.CSVReportGenerator;

import reports.template.ReportGenerator;

/\*\* Demonstrates template reuse across two formats. \*/

public class TemplateDemo {

public static void main(String[] args){

ReportGenerator pdf = new PDFReportGenerator();

System.out.println("=== PDF ===");

pdf.generateReport();

System.out.println("\n=== CSV ===");

ReportGenerator csv = new CSVReportGenerator();

csv.generateReport();

}

}

Console

=== PDF ===

Fetching data for PDF…

Processing data for PDF…

Formatting as PDF.

Saving the report to disk.

=== CSV ===

Fetching data for CSV…

Processing data for CSV…

Formatting as CSV.

Saving the report to disk.

#### **4 Test ideas (JUnit outline)**

/\* PDFFlowTest.java \*/

StringWriter out = new StringWriter();

System.setOut(new PrintStream(new WriterOutputStream(out), true));

new PDFReportGenerator().generateReport();

assertTrue(out.toString().contains("Formatting as PDF."));

/\* CustomHookTest.java \*/

class TestReport extends ReportGenerator {

@Override protected void fetchData(){} @Override protected void processData(){}

@Override protected void formatReport(){ System.out.println("fmt"); }

@Override protected void saveReport(){ System.out.println("custom save"); }

}

## **reflection.md**

Template Method puts **shared sequence** in one place, cutting duplication and ensuring every report runs fetch → process → format → save.

| **Benefit** | **Note** |
| --- | --- |
| DRY | Common workflow centralised. |
| Safe extension | Format authors can’t break step order. |
| Consistency | Every report persists via the same save step (unless overridden). |

*Trade-offs*

* **Subclass explosion** – each variation → new class.
* **Rigidity** – changing step order affects all subclasses.
* **Inheritance coupling** – cannot swap behaviour at runtime (contrast Strategy).

For pipelines that always follow the same backbone but differ in a handful of stages—reports, ETL jobs, game loops—Template Method delivers clarity and reuse with minimal code.