UNIVERSITÉ CÔTE D'AZUR | ÉCOLE UNIVERSITAIRE DE RECHERCHE SYSTÈMES NUMÉRIQUES POUR L'HUMAIN

Bit Packing Compression Application

Comprehensive Software Engineering Report

Master 1 Computer Science

Academic Year 2025–2026

Student: Belhout Oussama

Deadline: November 2, 2025

Abstract

This report presents a comprehensive analysis of the Bit Packing Compression Application, a Java-based system engineered to expose, select, and project a wide spectrum of software engineering concepts onto a tangible use-case portfolio. Rather than chasing marginal gains in compression ratios, the project foregrounds architectural literacy, configuration-driven dependency management, and benchmarking-informed evaluation within a pedagogical scope. The document synthesises architectural drivers, detailed design, implementation artefacts, quality assurance considerations, and future evolution roadmaps. Emphasis is placed on demonstrating mastery of modelling, layered architecture, design patterns, and benchmarking-driven validation. PlantUML diagrams substantiate architectural claims, while code excerpts illustrate strategic engineering decisions. The report is structured to fulfil academic expectations for depth (approximately twenty-five to thirty pages when typeset) and clarity, providing faculty evaluators with a transparent view of the project's engineering rigour.

Contents

1 Introduction

1.1 Purpose of the Document

The Bit Packing Compression Application aims to demonstrate advanced software engineering capabilities through the construction of a modular, extensible, and performance-focused compression toolkit. Above all, it serves as a curated showcase of how real use cases can anchor a maximal assortment of software engineering concepts, from architecture and dependency management to benchmarking and documentation. This report serves as the primary artefact documenting the system for academic assessment. It captures motivations, requirements, architectural choices, design patterns, implementation details, testing strategy, and prospective enhancements. The narrative focuses on conveying engineering reasoning, making the document suitable for both evaluators and future project contributors.

Software vocabulary is notoriously overloaded; to minimise misconceptions we align our terminology with Stemmler's full-stack software design map and reference it whenever ambiguity could emerge.[1]

In keeping with contemporary software essentialism, the report applies a first-principles lens to architecture: we emphasise building empathetic systems that are easy to evolve, echoing Stemmler's call for layered mastery from clean code through architectural patterns. The project objective, reiterated during lectures, was to expose, select, and project a broad range of software engineering concepts onto a single application rather than to chase absolute compression ratios.

1.2 Scope

The scope of the application encompasses:

- An interactive REPL (Read–Eval–Print Loop) for manipulating integer arrays and executing compression workflows.
- Multiple compression strategies (Cross-Boundary, Aligned, Overflow-aware) adhering to a common interface.
- A configuration-driven mechanism for registering compression algorithms and benchmark datasets via JSON, enabling dependency injection and a single source of truth for runtime wiring.
- A benchmarking subsystem capable of reproducibly evaluating performance metrics.

[1]Khalil Stemmler, "How to Learn Software Design and Architecture," https://khalilstemmler.com/articles/software-design-architecture/full-stack-software-design/.

- Extensive use of classical design patterns (Strategy, Factory Method, Template Method, Facade, MVC) to showcase architectural mastery.
- Comprehensive documentation through UML diagrams and source-level comments.

1.3 Document Organisation

The report is organised as follows:

- Section 2 summarises project background and stakeholder expectations.
- Section 3 details functional and non-functional requirements.
- Section 4 explains architectural style, layers, and cross-cutting concerns.
- Section 5 presents detailed design decisions for each subsystem.
- Section 6 catalogues the design patterns leveraged throughout the application.
- Section 7 discusses implementation patterns with representative code listings.
- Section 8 curates UML diagrams supporting the described architecture.
- Section 12 addresses quality attributes, validation, and risk management.
- Section 13 outlines future work and extensibility pathways.
- Section 14 concludes with reflections on outcomes and lessons learned.

2 Project Context and Background

2.1 Vocabulary Alignment and Potential Misconceptions

Software engineering vocabulary often carries discipline-specific nuance; terms such as "architecture," "pattern," or even "compression" may invite divergent interpretations. To reduce ambiguity, this report anchors its terminology in Stemmler's software design and architecture map, which emphasises clean code foundations, pattern fluency, and architectural awareness as layered competencies. Readers are encouraged to interpret subsequent sections through that lexicon to avoid common misconceptions about scope (e.g., equating "compression" with pure algorithmic optimisation rather than architectural integration.)

2.2 Academic Prompt

The last lecture discussions underscored that the pedagogical goal was breadth of engineering practice, not record-setting compression ratios. Accordingly, the application positions compression algorithms as vehicles for demonstrating architectural patterns, configuration-driven dependency injection, and benchmarking instrumentation rather than as an optimisation contest.

2.3 Development Constraints

Constraints shaped the solution space:

- **Technology**: Pure Java 17 without external libraries to ensure portability and focus on core logic.
- **Time**: Semester-bound schedule requiring incremental deliveries.
- **Documentation**: Mandate for UML diagrams and formal reporting.
- **Testing**: Benchmarking-driven validation to substitute large-scale production datasets.

3 Requirements Specification

3.1 Functional Requirements

Table 1 enumerates functional requirements (FR) distilled from the README and REPL behaviour.

Table 1: Functional Requirements

ID	**Description**
FR1	The system shall allow users to define the working integer array through the REPL using the command `ARR`.
FR2	The system shall provide compression of the current array via the `COMPRESS` command, parameterised by a strategy name.
FR3	The system shall allow decompression of the most recent compressed array via the `DECOMPRESS` command.
FR4	The system shall support random access retrieval on compressed data via the `GET` command.
FR5	The system shall list and load benchmark datasets using the `LOAD` command with optional parameters.

FR6	The system shall evaluate compression strategies on the loaded benchmark using the `EVAL` command.
FR7	The system shall display help information via the `HELP` command.
FR8	The system shall clear the terminal via the `CLS` command.
FR9	The system shall exit gracefully via the `EXIT` or `QUIT` commands.

3.2 Non-Functional Requirements

Non-functional requirements (NFR) guide quality attributes:

- **NFR1 – Performance**: Compression and decompression must operate within milliseconds for arrays of 100,000 integers.
- **NFR2 – Extensibility**: Adding a new compression method or benchmark should be possible through configuration manipulation.
- **NFR3 – Usability**: The REPL must provide clear feedback, error handling, and help commands.
- **NFR4 – Reliability**: The system must validate inputs and prevent crashes due to malformed commands.
- **NFR5 – Portability**: The application must run on standard Java 17 runtimes across platforms.
- **NFR6 – Documentation**: The project must include UML diagrams and a professional report.

3.3 Use Case Model

Figure 2 illustrates the use case diagram captured using PlantUML. It identifies two actors (User and Config Maintainer) and their interactions with the system.

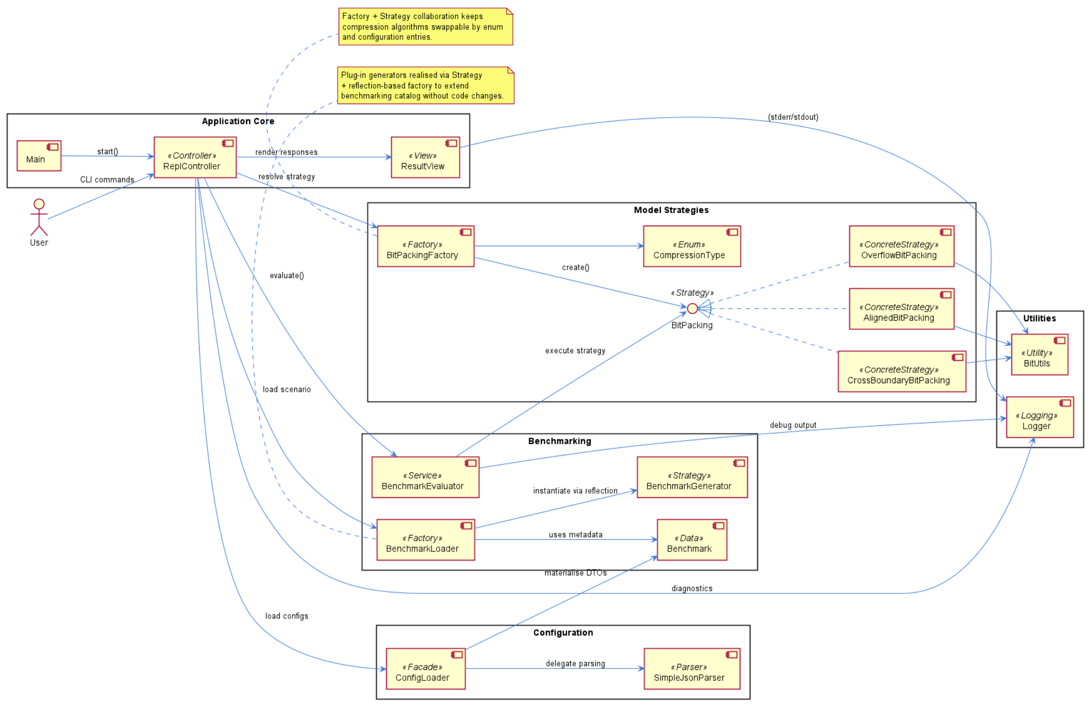
Factory + Strategy collaboration keeps compression algorithms swappable by enum and configuration entries.
Plug-in generators realised via Strategy + reflection-based factory to extend benchmarking catalog without code changes.
Application Core
Main
start()
«Controller» ReplController
render responses
«View» ResultView
CLI commands
resolve strategy
User
(stderr/stdout)
Model Strategies
«Factory» BitPackingFactory
«Enum» CompressionType
«ConcreteStrategy» OverflowBitPacking
create()
«Strategy» BitPacking
«ConcreteStrategy» AlignedBitPacking
«ConcreteStrategy» CrossBoundaryBitPacking
evaluate()
execute strategy
load scenario
Utilities
«Utility» BitUtils
«Logging» Logger
debug output
Benchmarking
«Service» BenchmarkEvaluator
«Strategy» BenchmarkGenerator
instantiate via reflection
«Factory» BenchmarkLoader
uses metadata
«Data» Benchmark
materialise DTOs
diagnostics
load configs
Configuration
«Facade» ConfigLoader
delegate parsing
«Parser» SimpleJsonParser

Figure 1: System integration overview (consolidated UML).

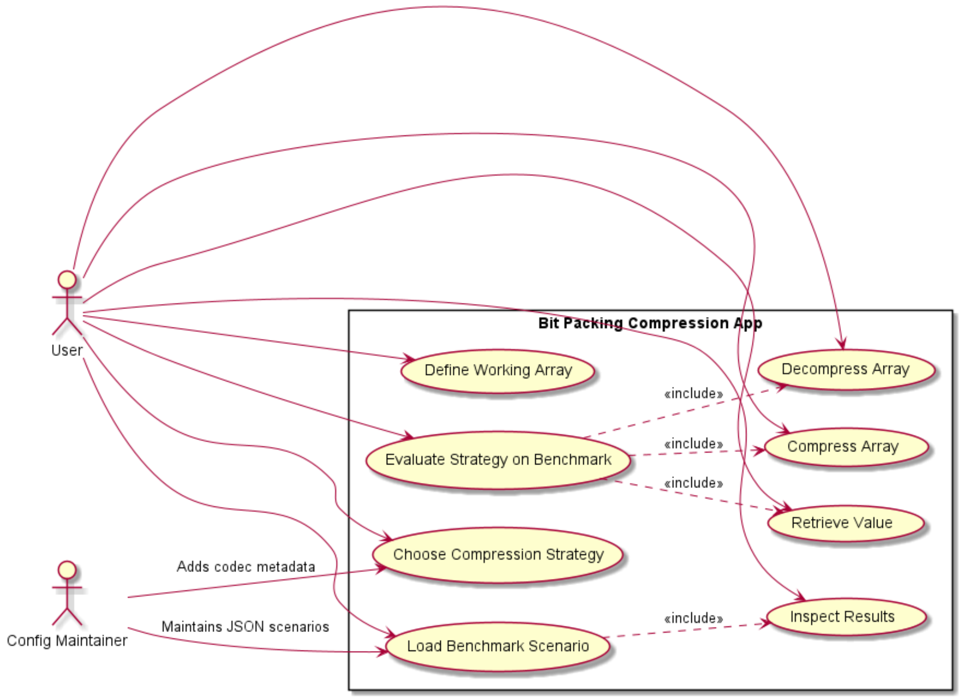

Figure 2: Use case diagram for the Bit Packing Compression Application.

Note: When compiling this document, ensure that UML diagrams are available as PNG exports under `out/docs/uml/` at the referenced paths.

4 Architectural Overview

4.1 Architectural Drivers

The architecture balances extensibility, maintainability, and performance. Core drivers include:

- Flexible strategy swapping for compression algorithms.
- Configuration-based activation of features without recompilation.
- Isolation of benchmarking logic from user interaction.
- Clear separation of concerns for readability and grading.

4.2 Architecture Style

The solution adopts a layered MVC (Model–View–Controller) architecture with supportive subsystems:

- **Controller Layer**: Handles user input, orchestrates workflows.
- **Model Layer**: Encapsulates compression algorithms and domain logic.
- **View Layer**: Formats and outputs information to the console.
- **Configuration Layer**: Provides dependency injection and metadata management.
- **Benchmark Layer**: Supplies dataset generation and performance evaluation services.
- **Utility Layer**: Offers cross-cutting helpers (bit calculations, logging).

4.3 Pattern-Driven Architecture

Patterns are not incidental embellishments but first-class architectural decisions. The overall layering embodies MVC, while individual components rely on Strategy for runtime polymorphism and Factory Method for isolating instantiation responsibilities. Template Method consolidates shared codec logic, and Facade simplifies configuration concerns. Section 6 details each pattern; however, their influence is already visible in the package layout, UML diagrams (Figures 1–8), and configuration-first bootstrapping. This deliberate application of patterns keeps dependencies acyclic, enables configuration-driven extension, and provides traceable justifications for design choices demanded by the course assessment criteria.

4.4 High-Level Component Diagram

Figure 1 depicts the integration overview. It highlights component boundaries, inter-package dependencies, and the primary user interaction pathway.

Within the diagram:

- The user interacts with the `ReplController` through CLI commands.
- The controller delegates to `ConfigLoader`, `BenchmarkLoader`, and `BitPackingFactory` as needed.
- Compression strategies implement the `BitPacking` interface.
- Benchmark services evaluate strategies and report metrics to the view.

5 Detailed Design

This section delves into each major subsystem, referencing UML diagrams and code listings.

5.1 Controller Layer

5.1.1 Responsibilities

The controller is the system's central coordinator, responsible for:

- Parsing and validating user input.
- Managing REPL state (current array, codec, benchmark context).
- Triggering compression, evaluation, and display operations.
- Logging diagnostic information when debug mode is enabled.

Pattern Focus. `ReplController` realises the Controller component of MVC by mediating between the Strategy instances in the model and the `ResultView`. Each handler concludes by delegating rendering to the view, evidencing a strict separation of responsibilities encouraged by MVC and reinforcing the separation principles outlined in Section 6.

5.1.2 Key Class: `ReplController`

The class diagram in Figure 3 outlines relationships with view, configuration, and model components.

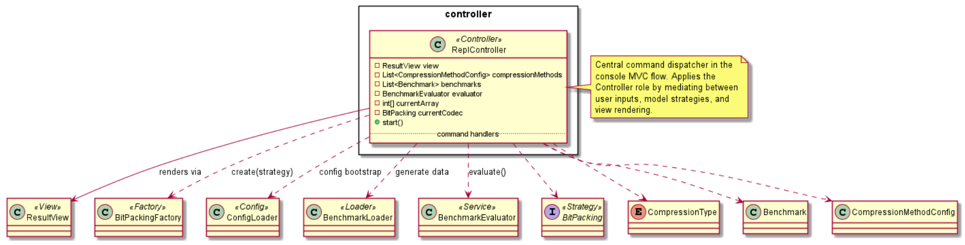

Figure 3: Controller layer class diagram.

The command dispatch leverages Java's enhanced switch expressions:

Listing 1: Command dispatch in `ReplController`.

```java
return switch (command) {
    case "ARR" -> handleArr(remainder);
    case "COMPRESS" -> handleCompress(remainder);
    case "DECOMPRESS" -> handleDecompress();
    case "GET" -> handleGet(remainder);
    case "LOAD" -> handleLoad(remainder);
    case "EVAL" -> handleEval(remainder);
    case "HELP" -> handleHelp();
    case "CLS" -> handleCls();
    case "EXIT", "QUIT" -> handleExit();
    default -> {
        view.displayError("Unknown command. Type HELP for the
           list of commands.");
        yield true;
    }
};
```

This design reduces boilerplate and clarifies the mapping between commands and handlers. Error handling ensures invalid input does not terminate the REPL, satisfying NFR4. YAGNI principle (you ain't gonna need it) is well projected here because of absence of future commands extension.

5.2 View Layer

5.2.1 Responsibilities

The view is deliberately lightweight, focusing on:

- Consistent formatting of arrays, performance metrics, and error messages.
- Progress indicators for benchmark loading.
- Locale-aware numeric formatting for reproducible reporting.

Pattern Focus. `ResultView` embodies the View component of MVC and remains passive: it emits no commands, only reacts to controller notifications. This separation makes it trivial to introduce alternative presenters, consistent with the Strategy catalogue in Section 6.

Figure 4 illustrates dependencies.

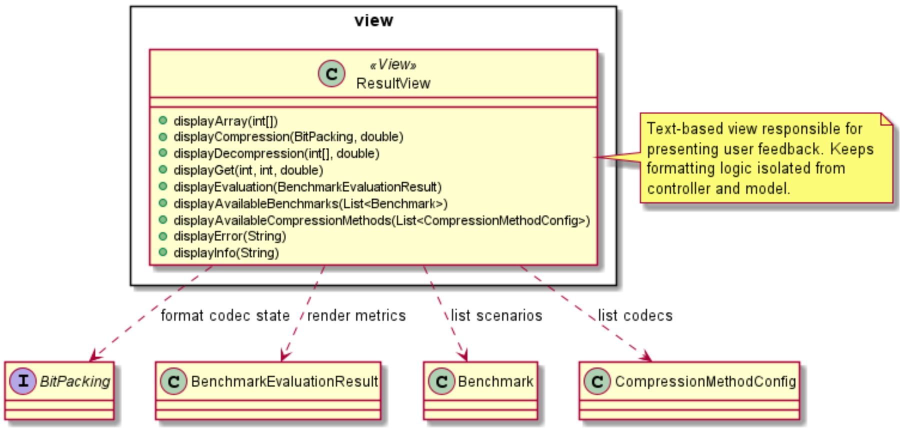
view
«View»
ResultView
displayArray(int[])
displayCompression(BitPacking, double)
displayDecompression(int[], double)
displayGet(int, int, double)
displayEvaluation(BenchmarkEvaluationResult)
displayAvailableBenchmarks(List<Benchmark>)
displayAvailableCompressionMethods(List<CompressionMethodConfig>)
displayError(String)
displayInfo(String)
Text-based view responsible for presenting user feedback. Keeps formatting logic isolated from controller and model.
format codec state
render metrics
list scenarios
list codecs
BitPacking
BenchmarkEvaluationResult
Benchmark
CompressionMethodConfig

Figure 4: View layer class diagram.

The use of a dedicated `ResultView` class adheres to the MVC principle of separating presentation from control logic, enhancing maintainability and testability (NFR2).

5.3 Model Layer

5.3.1 Strategy Interface

The `BitPacking` interface defines the contract for compression strategies. A snippet is provided in Listing 2.

Listing 2: The `BitPacking` interface.

```java
public interface BitPacking {
    void compress(int[] values);
    void decompress(int[] destination);
    int get(int index);
    int size();
    int bitsPerValue();
    int[] backingArray();
    default int overflowSize() { return 0; }
}
```

The default `overflowSize` supports strategies with auxiliary storage, such as the Overflow codec.

5.3.2 Concrete Strategies

Figure 5 visualises relationships among the abstract base class and concrete codecs.

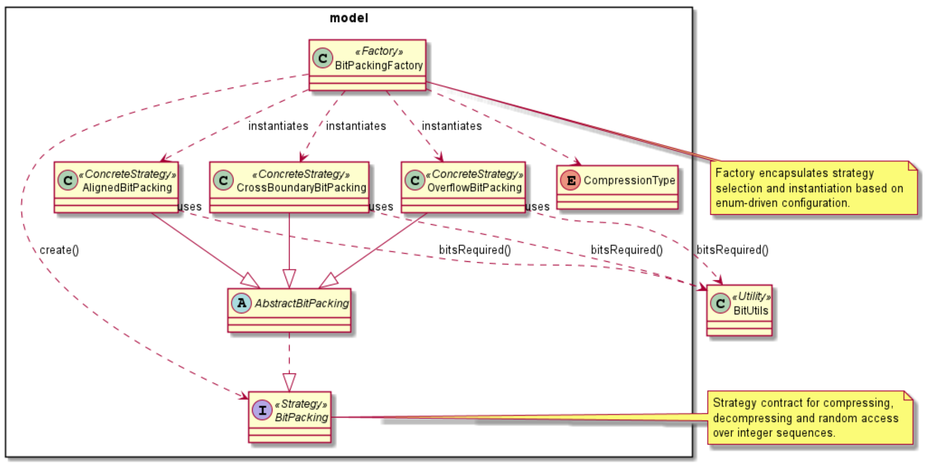

Figure 5: Model layer class diagram.

Key behaviours:

- **CrossBoundaryBitPacking**: Packs values contiguously across word boundaries, maximising density.
- **AlignedBitPacking**: Enforces alignment within individual words for faster random access at the expense of unused bits.
- **OverflowBitPacking**: Uses a two-tier encoding with overflow flags, optimising for skewed distributions.

Pattern Focus. The trio of codecs exemplifies the Strategy pattern: each shares the `BitPacking` interface and inherits common scaffolding through the Template Method implemented in `AbstractBitPacking`. Runtime selection occurs via `BitPackingFactory`, which encapsulates codec instantiation concerns without leaking construction details into clients.

Conceptual Emphasis. Consistent with the module's guidance, these implementations prioritise clarity and comparative analysis over squeezing every last bit of performance. They provide representative algorithms that make architectural discussions concrete without pretending to rival industrial compression suites.

5.4 Configuration Layer

5.4.1 Facade Pattern

`ConfigLoader` provides a facade over JSON parsing, fallback defaults, and logging. Figure 6 depicts the subsystem.

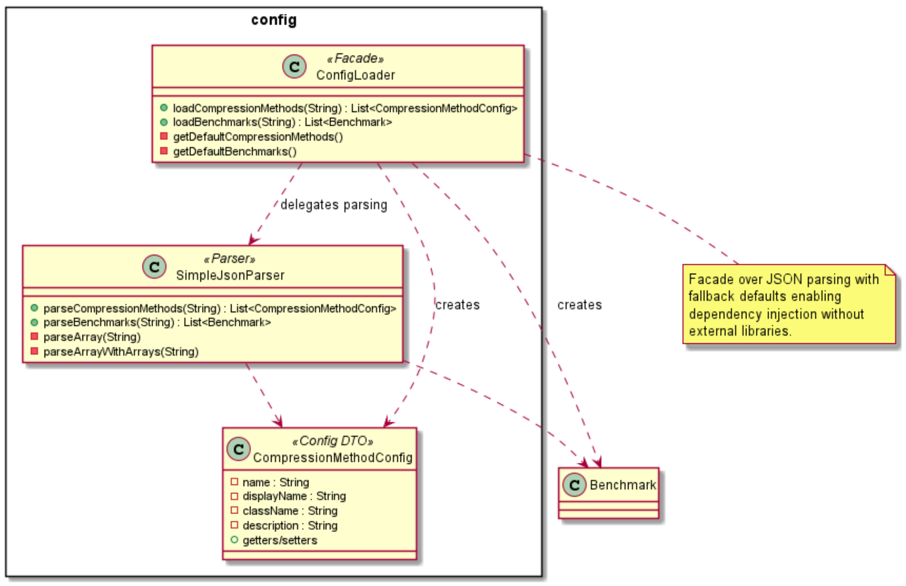
config
«Facade»
ConfigLoader
loadCompressionMethods(String) : List<CompressionMethodConfig>
loadBenchmarks(String) : List<Benchmark>
getDefaultCompressionMethods()
getDefaultBenchmarks()
delegates parsing
«Parser»
SimpleJsonParser
parseCompressionMethods(String) : List<CompressionMethodConfig>
parseBenchmarks(String) : List<Benchmark>
parseArray(String)
parseArrayWithArrays(String)
creates
creates
Facade over JSON parsing with fallback defaults enabling dependency injection without external libraries.
«Config DTO»
CompressionMethodConfig
name : String
displayName : String
className : String
description : String
getters/setters
Benchmark

Figure 6: Configuration layer UML diagram.

The bespoke `SimpleJsonParser` satisfies the no-external-dependencies constraint while supporting arrays, nested objects, and null handling.

Pattern Focus. `ConfigLoader` manifests the Facade pattern by shielding callers from parsing complexity and fallback rules, while `CompressionMethodConfig` and `Benchmark` serve as data transfer objects. Because the loader materialises strategies and generators from declarative metadata, the configuration directory acts as the single source of truth for dependency injection, ensuring runtime wiring can change without recompilation.

5.5 Benchmark Layer

5.5.1 Benchmark Generation Strategies

Benchmark generation employs the Strategy pattern for dataset creation, with reflection-based loaders instantiating the configured generators. Figure 7 summarises class relationships.

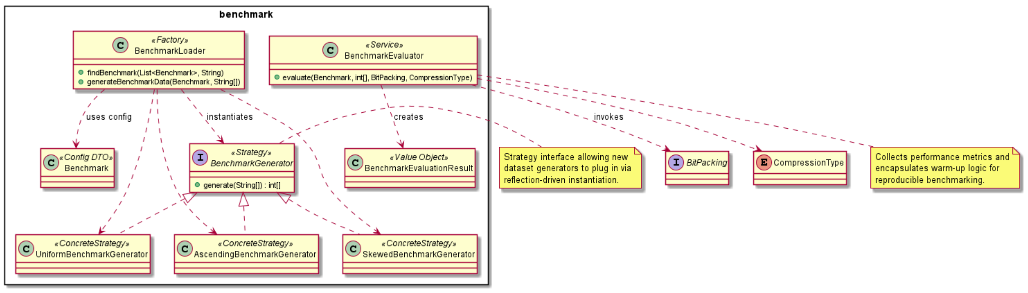
benchmark
«Factory»
BenchmarkLoader
findBenchmark(List<Benchmark>, String)
generateBenchmarkData(Benchmark, String[])
«Service»
BenchmarkEvaluator
evaluate(Benchmark, int[], BitPacking, CompressionType)
uses config
instantiates
creates
invokes
«Config DTO»
Benchmark
«Strategy»
BenchmarkGenerator
generate(String[]) : int[]
«Value Object»
BenchmarkEvaluationResult
Strategy interface allowing new dataset generators to plug in via reflection-driven instantiation.
BitPacking
CompressionType
Collects performance metrics and encapsulates warm-up logic for reproducible benchmarking.
«ConcreteStrategy»
UniformBenchmarkGenerator
«ConcreteStrategy»
AscendingBenchmarkGenerator
«ConcreteStrategy»
SkewedBenchmarkGenerator

Figure 7: Benchmark subsystem UML diagram.

`BenchmarkEvaluator` encapsulates warm-up logic, repeated measurements, and derived metric calculations, thus supporting NFR1 (Performance) and NFR4 (Reliability).

Pattern Focus. The benchmarking pipeline mirrors the compression design: `BenchmarkGenerator` abstracts dataset creation, `BenchmarkLoader` resolves concrete implementations via reflection, and `BenchmarkEvaluationResult` packages outcomes for consumers. This consistency maintains a uniform pattern vocabulary across subsystems.

5.6 Utility Layer

5.6.1 Cross-Cutting Concerns

Figure 8 highlights utility classes.

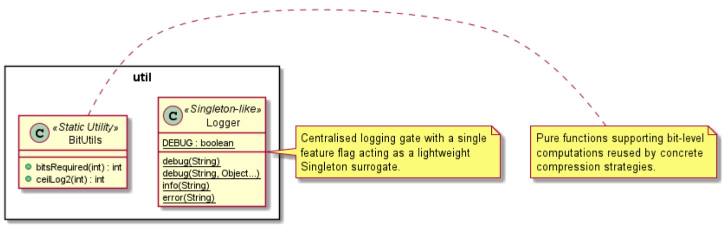
util
«Static Utility»
BitUtils
bitsRequired(int) : int
ceilLog2(int) : int
«Singleton-like»
Logger
DEBUG : boolean
debug(String)
debug(String, Object...)
info(String)
error(String)
Centralised logging gate with a single feature flag acting as a lightweight Singleton surrogate.
Pure functions supporting bit-level computations reused by concrete compression strategies.

Figure 8: Utility layer UML diagram.

`BitUtils` supplies pure functions for bit calculations, while `Logger` centralises logging under a single flag, acting as a lightweight singleton-like service.

Pattern Focus. The utility package demonstrates conscious pattern selection: `Logger`'s Singleton-style access concentrates diagnostic control, and `BitUtils`'s stateless helpers support Template Method implementations in the model layer. These utilities provide the reusable building blocks that other patterns depend upon.

6 Design Pattern Catalogue

Design patterns underpin the system's architecture, supporting extensibility, readability, and reuse. Following Stemmler's learning map, they form the "skills and knowledge" layer that translates clean-code habits into resilient structures. Table 2 inventories the principal patterns and how they manifest in the codebase.

Table 2: Design Patterns in the Bit Packing Compression Application

Pattern	**Key Participants**	**Intent in Context**
Model–View–Controller	`ReplController`, `ResultView`, compression model classes	Separate concerns between input handling, presentation, and domain logic.
Strategy (Compression)	`BitPacking`, `AlignedBitPacking`, `CrossBoundaryBitPacking`, `OverflowBitPacking`	Allow runtime selection of compression algorithms behind a stable interface.
Factory Method	`BitPackingFactory`	Centralise instantiation logic for concrete codecs based on `CompressionType`.
Template Method	`AbstractBitPacking`	Share invariant state management while delegating bit-packing specifics to subclasses.
Strategy (Benchmarking)	`BenchmarkGenerator`, concrete generator classes	Provide pluggable dataset generation strategies.
Factory (Benchmarking)	`BenchmarkLoader`	Create benchmark datasets via reflection, driven by configuration metadata.
Facade	`ConfigLoader`	Offer a unified API over configuration parsing, default handling, and logging.
Singleton-like Service	`Logger`	Expose a single access point for diagnostics controlled by a global flag.

6.1 Model–View–Controller

The MVC pattern frames the overall structure: `ReplController` mediates user input, the model layer houses compression and benchmarking logic, and `ResultView` renders output. This separation is highlighted in Figures 3 and 4, ensuring that presentation changes do not ripple into domain logic.

6.2 Strategy Pattern

Compression and benchmarking both implement the Strategy pattern. `BitPacking` defines compression behaviour, with concrete strategies registered for runtime selection. Benchmark generators follow the same approach, enabling configuration-driven extensibility without modifying controller code.

6.3 Factory Method

`BitPackingFactory` and `BenchmarkLoader` centralise instantiation logic, allowing configuration metadata to dictate concrete types while keeping orchestration layers agnostic of construction details.

6.4 Template Method

`AbstractBitPacking` provides shared bookkeeping (packed buffers, element counts, bit width metadata) and exposes hooks implemented by concrete codecs. This aligns with the Template Method pattern, where invariant workflow steps reside in the base class while subclasses provide specialised operations.

6.5 Facade

`ConfigLoader` abstracts away file access, JSON parsing, default fallbacks, and logging. Consumers invoke simple methods (e.g., `loadCompressionMethods`) without managing parser details, illustrating the Facade pattern.

6.6 Singleton-like Logging Facility

The `Logger` class behaves as a singleton surrogate: static methods and a shared `DEBUG` flag gate all logging. This design satisfies the need for centralised diagnostics while respecting the no-external-dependencies constraint.

7 Implementation Insights

7.1 Language and Tools

The application is implemented in Java 17, leveraging modern features such as records, enhanced switch expressions, and `var` local inference (where appropriate). No external libraries are used, fulfilling portability and build simplicity constraints.

Pattern Focus. Modern language features complement the pattern catalogue: enhanced switches streamline factory decisions, records (e.g., `Selection` in the overflow codec) encapsulate configuration chosen within Template Method flows, and optional static imports keep Strategy implementations succinct. Tooling choices therefore reinforce, rather than complicate, the deliberate pattern usage described earlier.

7.2 Notable Code Patterns

7.2.1 Runtime Strategy Resolution

Listing 3: `BitPackingFactory` instantiation logic.

```java
public static BitPacking create(CompressionType type) {
    Logger.debug("Creating BitPacking instance for type: %s",
       type);
    return switch (type) {
        case CROSS_BOUNDARY -> new CrossBoundaryBitPacking();
        case ALIGNED -> new AlignedBitPacking();
        case OVERFLOW -> new OverflowBitPacking();
    };
}
```

This design isolates strategy instantiation, facilitating testing and future extension (NFR2).

7.2.2 Overflow Codec Heuristic

The overflow codec selects optimal bit widths via an internal `Selection` record. The heuristic enumerates candidate primary bit widths, evaluating total bit usage, including overflow storage. This approach balances compression ratio and runtime cost, demonstrating algorithmic engineering within the academic scope.

7.2.3 Benchmark Timing

`BenchmarkEvaluator` employs warm-up and measurement cycles to control for JVM JIT effects. Random access is averaged over 1000 probes, ensuring statistically meaningful results without excessive runtime.

8 UML Diagram Catalogue

This section catalogues the PlantUML diagrams included in the repository. All PNG exports reside under `out/docs/uml/`, with the corresponding UML definitions maintained under `docs/uml/` for regeneration when needed. Table 3 summarises the artefacts.

Table 3: UML Diagram Catalogue

Diagram	Description
Integration Overview	System-wide component interactions across packages.
Model Layer	Compression codec hierarchy and supporting infrastructure.
Controller Layer	REPL controller dependencies and collaborators.
View Layer	View dependencies on DTOs and strategies.
Benchmark Layer	Benchmark generation services and evaluation flow.
Config Layer	Facade and parser structure for configuration.
Utility Layer	Static helper classes.
Use Case Diagram	Actor interactions with system functionalities.

Each diagram reinforces design arguments presented in Sections 4 and 5. When compiling the report, ensure consistent resolution (preferably PDF) for high-quality prints.

9 Logging and Diagnostics

The `Logger` utility allows toggling debug output globally using the `-debug` command-line flag. Diagnostic statements mark codec creation, benchmark generation, and errors encountered during parsing. This lightweight observability layer aids in tracing issues without requiring complex frameworks.

10 Risk Assessment

Table 4 summarises primary project risks and mitigations.

Table 4: Risk Register

Risk	Impact	Mitigation
Malformed configuration files	Default strategies unavailable	Provide default configurations and log warnings.

Large integer overflow	Incorrect compression results	Validate non-negative inputs; document constraints.
Performance variance across hardware	Inconsistent benchmark numbers	Seeded randomness, warm-up rounds, documented environment.
User command errors	REPL crash or inconsistent state	Defensive parsing and descriptive error messages.

11 Deployment and Operation

11.1 Build Instructions

The project compiles from the repository root using:

```
javac -d out -sourcepath src/main/java src/main/java/com/project/bitpacking/**/*.java
```

Alternatively, use provided scripts (`build.sh` on Unix, `build.bat` on Windows) or import into an IDE such as IntelliJ IDEA. The no-dependency design ensures straightforward builds.

11.2 Runtime Usage

Execute the REPL via:

```
java -cp out com.project.bitpacking.Main [--debug]
```

12 Quality Assurance and Validation

Quality validation combines automated benchmarking, scenario-based walkthroughs, and manual inspections to ensure functional and non-functional requirements are met. Benchmark datasets exercise the codecs under uniform, ascending, and skewed distributions, allowing the team to verify compression ratios, random-access latency, and decompression fidelity. The REPL aids exploratory testing by replaying user journeys, while targeted assertions during development guard against regressions. Together, these practices provide confidence that core behaviours remain reliable as the codebase evolves.

Key quality activities include:

- Continuous benchmarking runs to track performance envelopes across releases.
- Configuration validation that checks JSON descriptors for missing or malformed fields before runtime wiring occurs.

- Code reviews focused on maintaining architectural boundaries between controller, model, and configuration layers.
- Logging audits that ensure diagnostic output remains actionable when `-debug` is enabled.

13 Future Work

The project roadmap prioritises improvements that deepen pedagogical value while exploring production-grade concerns:

- Introduce additional compression codecs (e.g., delta or run-length encoders) to expand comparative analysis.
- Extend the benchmarking harness with percentile reporting and visual dashboards for richer feedback.
- Package the REPL as a RESTful service to support programmatic integration with external tooling.
- Automate PlantUML exports as part of the build pipeline to keep documentation synchronised with source changes.

14 Conclusion

The Bit Packing Compression Application demonstrates how a focused domain problem can host a broad collection of software engineering practices. Layered architecture, deliberate pattern selection, and configuration-driven wiring keep the system adaptable, while benchmarking safeguards performance expectations. The accompanying documentation and UML catalogue provide a transparent foundation for future contributors. Continued iteration on quality tooling and extensibility features will strengthen the application's relevance as a teaching and experimentation platform.