

# Project 6 - MultiSet Design

---

## Learning Objectives

By completing this project, you will:

- Demonstrate understanding of object-oriented design principles: abstraction, encapsulation, and composition
  - Compare and justify alternative underlying internal representation
  - Produce a professional design document that communicates your reasoning, trade-offs, and challenges
  - Reflect on how design choices support extensibility, clarity, and maintainability
- 

## Overview

For this project, you will design a **MultiSet** data abstraction. A **MultiSet** is similar to a **Set** in that it represents a collection of elements. In a standard **Set**, each element appears at most once, whereas in a **MultiSet** an element may occur multiple times.

*This project focuses entirely on the **design phase**. As such, you will not implement or submit any source code.*

## Context

You are designing the core container behind a game system where elements can appear multiple times. For simplicity, assume each element has type **string**. Your design will describe how the **MultiSet** serves as a core data structure for a game system, such as:

- A player inventory of items (e.g. potions x 3, arrows x 12)
- An enemy loot table (which items drop and in what quantities)
- A crafting or resource system (materials required and consumed)
- A spawn table for monsters in a region
- Any other system where elements can appear multiple times

You will use one of your previous data structures as the internal representation:

- **Sequence** (Dynamic List) ( `<string>` )
- **HashTable** (HashMap) ( `<string, unsigned int>` )
- **AVLTree** (TreeMap) ( `<string, unsigned int>` )

Your design should demonstrate sound reasoning, clear communication, and thoughtful discussion of trade-offs, not code.

---

## Design Document Specification

Submit a 5–6 page design document (*no implementation*) that includes the following sections:

### 1. Introduction

Describe your chosen context and summarize what your multiset is designed to do in that world.

For example: “My design models a player inventory that stores string item names and counts, built atop a `HashTable<string, unsigned int>`.”

## 2. Design Philosophy

Explain the qualities you prioritized in your design such as efficiency, simplicity, extensibility, and readability.

State who the **client** and **user** of your **MultiSet** class would be (for example, another developer, a game system, or another part of your codebase).

## 3. Core Operations

Define four to five operations that any **MultiSet** should support.

For each operation, describe:

- What it does conceptually in your chosen game scenario
- Its expected time complexity
- Possible edge cases or exceptional situations
- How your underlying data structure supports or constrains the operation

Avoid using C++ syntax. Focus on describing **behavior** and **design intent** at a conceptual level rather than writing full pseudocode for every operation.

## 4. Set Operations

Design one or two set-like operations that are meaningful within your game world (for example, combining inventories or finding common items between two players).

Each operation below represents a possible **conceptual relationship** between game inventories:

Operation	Game Analogy
<code>union_with()</code>	Combine two inventories or loot tables (sum item counts)
<code>intersection_with()</code>	Find items shared between two inventories
<code>difference_with()</code>	Items a player has that another doesn't
<code>symmetric_difference_with()</code>	Items unique to each inventory

You do **not** need to design all four in full detail — choose one or two and explore those thoroughly.

For each operation, explain:

- What it accomplishes in gameplay
- How it manipulates your data structure
- Its conceptual complexity and any relevant edge cases

## 5. Extension Feature

Invent one new capability or behavior for your **MultiSet**.

Examples:

- `top_k(size_t k)` — returns the **k** most frequent items
- `remove_n(size_t n)` — removes and returns **n** arbitrary items
- `serialize()` / `deserialize()` — saves or loads the multiset to or from a file
- `craftRecipe()` — consumes ingredients to create a new item

Describe what new data or methods you would add, and explain **why this feature adds value** within your chosen game scenario.

Once you've established the core and extended functionality of your MultiSet, you are ready to illustrate your complete design.

## 6. UML Diagram / Abstraction Boundary

Now that you have defined your core operations, set behaviors, and extension feature, summarize your design using a UML diagram.

Your diagram should show the **public interface** (methods available to users) and **private internal members** (data and helper methods used internally).

Label each section clearly, and ensure your diagram reflects all the elements you described in the previous sections.

Use standard UML diagram conventions to illustrate your design.

You can find many examples online, such as this [UML Object Diagram Example](#).

Example:

"PhaseDiscriminator"
- matrix: PositronicMatrix - type: DiscriminatorType - beam: AnnularConfinementBeam
+ synchronize(): SyncStatus + setWarpFactor(factor: Factor): ChronitonCount + setBeam(beam: AnnularConfinementBeam, status: SyncStatus): void - invertTachyons(inverter: TachyonInverter): ChronitonFlux

Your UML diagram should clearly distinguish between **public interface** and **private members**.

Label each section accordingly, and explain why certain data or operations are hidden.

## 7. Trade-off Analysis

Compare your chosen base data structure (e.g., **Sequence**, **HashTable**, or **AVLTree**) with at least one alternative.

Write a short paragraph explaining **why you did not choose** the alternative structure, highlighting the trade-offs in performance, complexity, and design clarity.

Include a table summarizing your analysis — list the advantages, disadvantages, and key operation complexities for each data structure.

## 8. Alternative Design Sketch

Building on your trade-off analysis, briefly outline how your design would differ if you had chosen one of the other two base data structures.

This demonstrates your understanding that multiple valid designs can meet the same requirements in different ways.

You do not need to redesign the entire class — a short paragraph or a side-by-side comparison is sufficient.

## 9. Evaluation Plan

Describe how you would test your design if it were implemented.

What kinds of tests, inputs, or measurements would demonstrate that it functions correctly?

Also consider how you would evaluate the **extensibility** and **maintainability** of your design once implemented. What indicators would show that it is easy to modify or adapt for new features?

## 10. Conclusion / Reflection

Summarize what makes your design strong and effective.

Discuss the trade-offs you accepted and what you would improve with more time or iteration.

Finally, reflect on how your design demonstrates core object-oriented principles — **abstraction**, **encapsulation**, and **composition** — and how those principles shaped your decisions throughout the project.

---

## Required Components

You must include:

- **At least one diagram** (UML, block, or text-based)
- **At least one table** (complexity or trade-off comparison)
- **1-2 short pseudocode blocks or illustrative code snippets**
- **At least two cited sources**

You must include **two citations**: one must be a from a

- **scholarly or peer-reviewed source (journal or conference article found through Google Scholar)**
- **a credible technical reference** (such as <https://cppreference.com/>, documentation, or a standards page).

Citations should support your design choices, definitions, or analysis.

You may use any standard citation style (APA, IEEE, etc.) and include a short “References” section at the end of your document.

---

Format	Citation Resources
Microsoft Word	<a href="#">Add citations in Word (Microsoft Support)</a>
Google Docs	<a href="#">Add citations and a bibliography in Google Docs (Google Support)</a>
Markdown	<a href="#">Markdown citation basics (Markdown Guide)</a>
LaTeX	<a href="#">Bibliography management with BibTeX (Overleaf)</a>
Typst	<a href="#">Citations in Typst (Official Docs)</a>

---

## Document Format (+5 Bonus Points)

Many technical documents and papers in science and engineering are written using **plain text files**. These rely on **typesetting** or **document formatting languages** that transform raw text into a properly formatted, professional, and visually polished document such as a PDF.

Using these languages typically involves more effort than a traditional word processor like Microsoft Word or Google Docs. However, one major advantage of keeping your document text-based is **version control**. Yes – you can version-control your writing assignments.

Files created by word processors are more like binary data: they encode both text and formatting in ways that make it difficult for tools such as Git to track changes effectively. In contrast, plain text files make every edit visible and traceable, exactly like the commits you make to your code.

To introduce you to these document-creation approaches, you may optionally author your design document in one of several text-based formats. The three text-based formats below are listed roughly in order of **increasing learning curve** – and also **increasing control** over the final document’s appearance. Most formatting languages require a “compiler” to build the final PDF, so you will need to install the appropriate software for the chosen language.

Format	File Extension	Resource Links	Getting Started
Markdown	<code>.md</code>	<a href="#">GitHub Markdown Guide</a> , <a href="#">Markdown Cheat Sheet</a>	<a href="#">Markdown and Visual Studio Code</a>
Typst	<code>.typ</code>	<a href="#">Typst Documentation</a> , <a href="#">Typst Playground</a>	<a href="#">Install Typst</a> , <a href="#">Tinymist Typst VS Code Extension</a>
LaTeX	<code>.tex</code>	<a href="#">LaTeX Wikibook</a> , <a href="#">Overleaf Tutorial</a>	<a href="#">The LaTeX Project (recommend MikTeX for Windows)</a> , <a href="#">Visual Studio Code LaTeX Workshop Extension</a>

Markdown is simple and fast to learn and widely used for **readme files, software documentation, and lightweight technical writing** because of its simplicity and compatibility with platforms like GitHub.

Typst is a newer, modern typesetting language that offers a balance of structure and flexibility. It combines **Markdown-like readability with LaTeX-level control and precision**, making it well suited for reports, research documents, and design specifications. In fact, this project description PDF was written using **Typst**!

LaTeX (pronounced “lah-tekh” or “lay-tekh”) provides the highest degree of control and professional typesetting at the cost of greater complexity. Most journals, conference papers, theses, dissertations, and other professional documents in science and engineering are written using LaTeX.

Submitting your document in one of these text-based formats demonstrates technical professionalism and earns up to +5 bonus points.

## Turn in and Grading

You will submit your work through **GitHub Classroom**, following the same process used in previous projects.

Your repository must contain **all necessary files to reproduce your final PDF** and the **final exported PDF** itself.

Your submission must include:

- Your design document in its **source format** (`.md`, `.typ`, `.tex` or another plain-text format).
- The **final exported PDF** version of your document.
- If you used a word processor such as Microsoft Word or Google Docs, include the `.docx` (or equivalent) file *and* a PDF export.
- Any additional files required to build or display the document, such as:

- 
- Images or diagrams you reference
  - Custom LaTeX or Typst style files, packages, or macros
  - Any auxiliary data or configuration files used in your design document

Your repository should be complete enough that your instructor can regenerate your PDF from the provided files without additional setup.

***Commit your progress regularly, and ensure your final submission reflects your most recent and complete work***

Category	Description	Points
Context & Problem Understanding	Clear scenario, realistic goals	5
Abstraction Boundary	Well-defined interface vs. internals	7
Core Operation Designs	Correct, consistent, and well-reasoned	7
Trade-off Analysis	Compares alternatives with justification	7
Set Operations	Contextually meaningful and well-reasoned	5
Extension Feature	Creative and plausible	5
Diagram/Table Quality	Clear, accurate, and readable	5
Evaluation Plan	Shows forethought about testing/validation	5
Sources and Citation	At least 2 relevant and properly attributed citations	2
Overall Clarity & Professionalism	Well-written, structured, and formatted	2
Document Format (bonus)	Use of text-based document formatting	5
<b>Total</b>		<b>50</b>

---