Assignment Report

Project 1.0 | Pizza Config App

Ahmed Mohamed Ahmed Elshazly Mohamed

Feb, 2025

Project 1.0: The Pizza Config App (Detailed-ish Report)

1. Intro & Background

So, um, in project 1.0 we kinda set out to whip up a simple pizza configuration app – ya know, just to show off some basic OOP design and encapsulation (plus a bit of serialization for fun). The whole idea was to make a proof-of-concept that's simple, but flexible enough so later projects (like 1.1 and 1.2) can build on it. Basically, we wanted a starting point where you can mix and match pizza options without too much hassle.

2. What We Wanted & Needed

Our main goals were roughly:

- Build a core model that sorta "represents" a pizza config.
- Create a few classes – like **PizzaConfig, OptionSet**, and even an inner class called Option (inside **OptionSet**, since they're so related).
- Use a partially filled array (yep, the assignment said so) to store our option sets in PizzaConfig.
- Show off basic CRUD stuff – that's create, read, update, and delete – on the pizza configuration.
- Get serialization working so we can save (write) and later load (read) the pizza config.

Key things we needed:

- Plain ol' Java classes (think POJO style) with proper encapsulation.
- PizzaConfig must handle an array of configuration options.
- OptionSet should come with an inner class Option for each option.

- Stick to good OOP practices so it can be easily extended later.
- And of course, code needs to be packaged nicely, commented well, and tested.

3. How We Figured It Out

A. Requirements & Brainstorming

Before diving into code, we kinda thought about what a pizza configuration involves:

- A pizza should have a base price and a bunch of options (like size, toppings, etc.).
- Instead of making separate classes for every option (like PizzaSize or MeatChoices), we went for a general OptionSet. Each set holds several Option objects (each with its own name and price tweak).

B. Why a Partially Filled Array?

The assignment literally said "use a partially filled array", so we had to:

- Allocate a fixed-size array and only fill part of it.
- Pros: It shows we understand arrays and memory stuff, and forces us to handle empty spots manually.
- Cons: It's more error-prone than using something like an ArrayList, and you have to be super careful with indices.

C. Our Design Strategy

- **Encapsulation:** We made sure each class (like PizzaConfig, OptionSet, and Option) hides its data and only exposes what's needed.
- **Inner Class Usage**: Option lives inside OptionSet because, well, it just makes sense – options belong to their set.

- **Serialization:** Our classes implement Serializable so we can write them to a file and then read 'em back.
- **Keeping It Modular:** The design is kept super simple so it can be easily upgraded later. Down the road, you could swap out the array for something more dynamic or add new specialized classes.

4. The Nitty-Gritty Design & Architecture

A. Class Rundown

4.1 PizzaConfig

- **What it does:** It's the big picture – represents an entire pizza config.
- **Stuff inside:**
  - name: the name of the configuration (like "Deluxe Pizza").
  - basePrice: the basic price for the pizza.
  - optionSets: a partially filled array holding our OptionSet objects.
- **Main methods:**
  - Constructors to get things started.
  - Getters and setters for the properties.
  - CRUD functions to add, find, update, or remove an option set.
  - A print() method to show the config details.

4.2 OptionSet

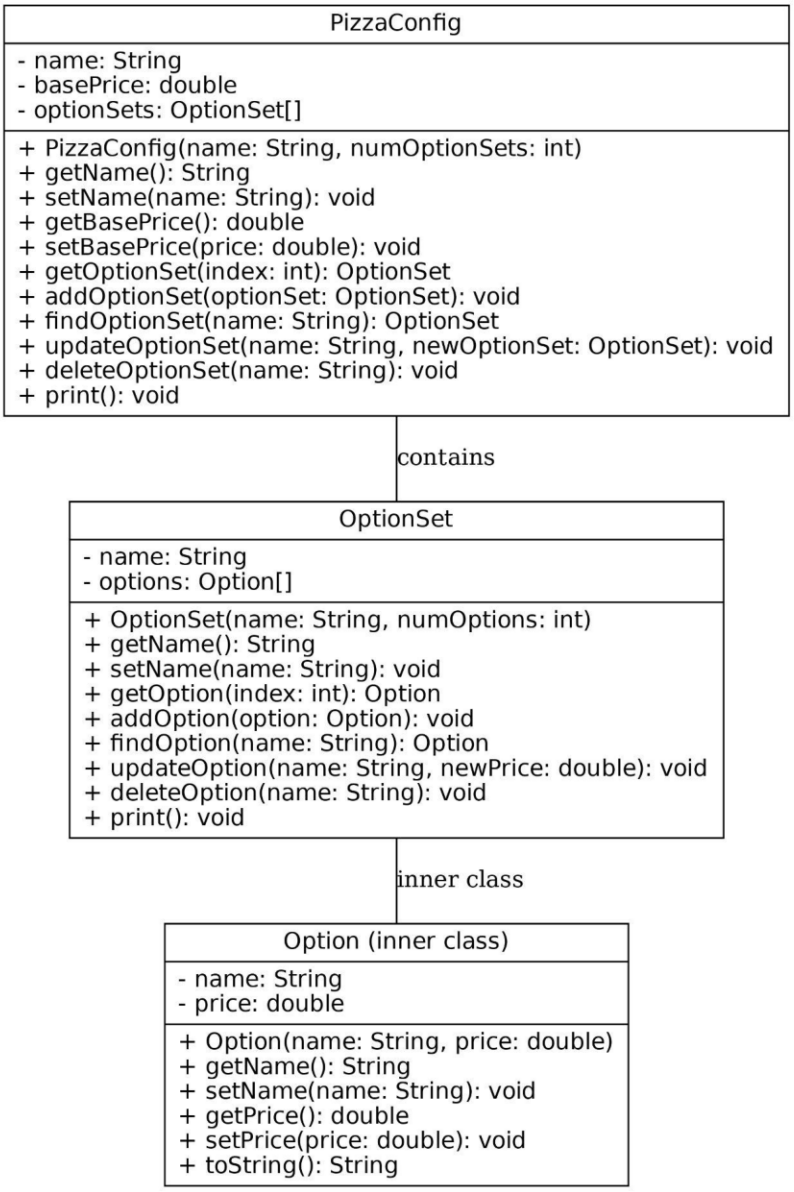
- **What it does:** Groups related options (like sizes or toppings).
- **Stuff inside:**

    - name: a name for the option set (e.g., “Size”).
    - options: an array (or partially filled array) of Option objects.
- **Main methods:**
    - Constructors, getters, and setters.
    - Functions to add, find, update, and remove options.
    - A print() method to display its options.

4.3 Option (Inner Class)

- **What it does:** Represents one single option.
- **Stuff inside:**
    - **name:** e.g., “Medium” for a pizza size.
    - **price:** the extra cost for that option (like 200 for Medium).
- **Main methods:**
    - Constructors, getters, setters, and a toString() to spit out its info.

B. A Rough UML (Text Version)



(This is a rough sketch of our class layout – not super fancy, but gets the point across.)

C. Why This Way?

- **Simplicity & Flexibility:** Using a generic model lets us add new options later without a total overhaul.
- **Manual Array Handling:** Although kinda tricky, using a partially filled array helped us learn the hard way about memory and indices.
- **Good Encapsulation:** Keeping our data private and grouping related stuff (like using inner classes) helps avoid a big ol' mess.
- **Serialization:** This paves the way for saving and loading configurations later on.

5. How We Wrote the Code

A. Key Decisions in the Code

- **Constructors:** We made constructors that kick off key properties and allocate our arrays.
- **CRUD Operations:** Each class got methods to create, read, update, and delete stuff – with error checking (like, "what if the OptionSet isn't found?").
- **Array Management**: Since we're using a partially filled array, we had to track element counts and indices manually (lots of comments in the code explain this).
- **Serialization:** All classes implement Serializable to let us persist the object state.
- **Testing:** We wrote a bunch of test cases and a test driver to cover every scenario – using polymorphism and even testing boundary cases.

B. Why Did We Do It Like That?

- **Generic Model:** Keeps things flexible – if new options come up, they just slot right in.
- **Inner Classes:** They make sure that options stay tied to their set, preventing misuse elsewhere.
- **Arrays vs. Collections:** Even though collections like ArrayList are nicer, we had to use arrays 'cause that's what the assignment said.
- **Serialization:** Lets us archive and later retrieve pizza configs, making the app more robust.

6. Testing & What We Found

A. How We Tested

- **Unit Tests:** We set up tests for every CRUD operation on both PizzaConfig and OptionSet – making sure things get added, updated, and removed correctly.
- **Serialization Tests:** We saved and then reloaded PizzaConfig objects to double-check that all the info sticks.
- **Manual Testing**: A test driver simulated real use – checking that printed outputs match what we expect.

B. Test Results (in a Nutshell)

- Console logs showed that the pizza config printed correctly – with the right base price and all option sets and options.
- All CRUD functions behaved as they should.
- Serialization/deserialization worked fine, keeping all our data intact.

7. What We Learned Along the Way

- **Handling Arrays Manually:** Juggling partially filled arrays was tricky – sometimes a dynamic collection would've been easier, but it was all part of the learning curve.
- **Encapsulation is Super Important:** Keeping our data private and grouping related items (like inner classes) helped maintain a clear structure.
- **Testing is a Must:** Solid unit and integration tests caught many edge cases, like trying to update an option that didn't exist.
- **Document Everything:** Explaining our decisions (and even our mistakes) in code comments and documentation really paid off.
- **Future-Proofing:** Even though this project is basic, it sets the stage for using design patterns (like Factory or Proxy) in future projects.

8. Wrapping It Up (Conclusion)

In a nutshell, Project 1.0 shows that we can build a simple Pizza Configuration App using old-school OOP design. We managed to:

- Create a core model using PizzaConfig, OptionSet, and Option.
- Use a partially filled array to handle our data (even if it wasn't the prettiest solution).
- Implement basic CRUD and serialization.
- Test thoroughly to ensure everything worked as expected.

This whole thing lays a pretty solid foundation for future upgrades – like using dynamic data structures or even throwing multithreading into the mix.

9. References & Stuff We Consulted

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Oracle Java Documentation: Java SE Documentation
- *Effective Java (3rd Edition)* by Joshua Bloch – for all those nifty Java best practices.