Gilded Rose Practices

1. Problems with the code

- Too many nested if-else statements - makes it hard to follow what's happening.
- Same checks repeated over and over again - item.name != “Sulfuras”.This is a problem because if we ever need to change how we handle Sulfuras (like changing its name or adding a condition), we'd have to find and update every single place where it's checked. It's easy to miss one and create bugs.
- Adding a new item is not easy - if we want to add a new item I would have to go through all the code and add new lines making it even more worse.
- Everything is crammed in one method - All the different item rules are mixed together in one big method, so you can't work on one item type without seeing all the others.
- Numbers like 50, 11, and 6 appear with no explanation of what they mean. By reading the requirements document, I found that 50 is the max quality, 10 is when backstage passes start increasing by 2, and 5 is when they increase by 3. In the new design, I'll define these as named constants like `MAX_QUALITY = 50`, `BACKSTAGE_TIER_1 = 10`, and `BACKSTAGE_TIER_2 = 5` so it's clear what they represent.

2. My Solution

I picked the Template Method Pattern because all items follow the same basic update process, but each type handles the details differently. The template method defines the overall algorithm (the steps), and each subclass fills in the specific rules for their item type.

How the Template Method Works

The **ItemUpdater** base class has an `update()` method that acts as the "template" - it defines the steps every item follows:

1. Calculate how much quality should change
2. Calculate how much sell_in should change
3. Apply the changes
4. Handle what happens if the item is expired
5. Make sure quality stays within valid bounds (0-50)

Each item type (Normal, Aged Brie, Sulfuras, Backstage Pass) inherits from ItemUpdater and overrides just the parts that are different for them.

How It Would be Organized

Item class - Unchanged, holds `name`, `sell_in`, `quality`

ItemUpdater (abstract base class) - Has the `update()` template method that defines the update sequence. Subclasses override `calculate_quality_change()`, `calculate_sell_in_change()`, and `handle_expired_item()` to customize behavior.

Concrete Updaters:

- **NormalItemUpdater** - Quality decreases by 1 (or 2 after expiration)
- **AgedBrieUpdater** - Quality increases by 1 (or 2 after expiration)
- **SulfurasUpdater** - Never changes
- **BackstagePassUpdater** - Quality increases by 1/2/3 based on days remaining, drops to 0 after concert

GildedRose class - Loops through items, gets the right updater for each, calls update()

3. How this Might Fix the Problems

- **No more crazy nesting** - The template method has a simple sequence of steps. Each updater class has straightforward logic with minimal nesting.
- **No repeated checks** - Each item type is handled by its own class. The item name is checked once when choosing the updater, not throughout the code.
- **Easy to add new items** - Just create a new updater class (like ConjuredItemUpdater), override the three methods with the new rules, and add it to the get_updater method. No need to touch existing code.
- **Better organized** - Each item type's rules are isolated in their own class. Bug with Aged Brie? Look in AgedBrieUpdater only.
- **Template provides structure** - The update() method in ItemUpdater shows the exact sequence of steps every item follows. This makes the logic clear and prevents mistakes.