Assignment 2: Solving the Brewery Problem
Erdun E
October 23, 2024
Dr. Lindsay Jamieson

CS 5010 Programming Design Paradigm
Complexity Analysis of Brewery Problem

1 Metrics Analysis

WMC	DIT	NOC	CBO	RFC	LCOM
54	13	0	15	74	Low (except medium in `ProductionController`)

Table 1: Summary of Metrics, almost same as peer review

This complexity analysis thoroughly examines the final version of my Brewery Control System project. I'm comparing this iteration to the feedback I received during the peer review two weeks ago. Overall, I've only made detailed optimizations since the last review and no structural changes, so the total number of metrics hasn't changed. Below, I'll share the six key metrics I used to analyze the system's complexity.

1.1 WMC (Weighted Methods per Class)

WMC measures how many lines of code there are per method and the complexity of those methods. The complexity increases with decision points, like if statements and loops, which adds to the cyclomatic complexity.

- ProductionController: This class is still one of the most complex since it coordinates several key functions like create batches, transfer them, and manage vats. The batch creation and transfer processes involve multiple conditionals, which raises the WMC.
- InventorySystem and RecipeLibrary: These two classes have a relatively low WMC. Their operations mainly focus on manage data like ingredients or recipes, with only a few control flows.
- Other Classes: Simpler classes like Batch, Vat, Pipe, and Ingredient are designed to encapsulate specific data, keeping the WMC low. Their methods mainly involve straightforward logic.

Even though I've reduced the complexity in the ProductionController, some methods are still large because of the number of responsibilities involved.

1.2 DIT (Depth of Inheritance Tree)

DIT measures how deep the class hierarchy goes. In this project, the structure is quite flat.

- DIT for All Classes: Every class in the system exists at the same level, with no subclassing. Classes like Batch, Vat, Pipe, and Recipe don't have any children, which means their inheritance depth is minimal.

While this simplicity makes the code easier to read, it also limits code reuse. Using interfaces or abstract classes could provide more flexibility if I decide to extend functionality across multiple classes like sharing behavior between Vat and Batch.

1.3 NOC (Number of Children)

NOC measures how many child classes inherit from a given class. Since none of the classes in my project use inheritance, the NOC is at 0.

- NOC Across Classes like Pipe, Batch, and Ingredient don't have any children. Each class stands independently, which helps minimize complexity from inheritance. This approach makes maintenance easier and keeps things simple, but it also reduces opportunities for shared behavior.

The simplicity is the key point for this version, but introducing inheritance could be beneficial if I decide to extend the system in the future like using polymorphism for different types of containers.

1.4 CBO (Coupling Between Objects)

CBO measures how dependent different classes are on each other. High coupling can make a system harder to maintain, while low coupling promotes modularity.

- ProductionController and BreweryControlSystem: These two classes have the highest coupling. The ProductionController relies heavily on InventorySystem and RecipeLibrary to coordinate operations.
- Pipe and Vat: There's also some coupling between these classes during batch transfers, which makes sense given their relationship.
- Other Classes: Classes like Ingredient and Batch have low coupling. They operate independently and don't interact with too many external components.

I aimed to keep coupling as low as possible, but some level of coupling is unavoidable due to the coordinating role of the controller. Use interfaces to further reduce these dependencies and improve modularity.

1.5 RFC (Response for a Class)

RFC measures how many methods a class has and how many external methods it calls. A higher RFC suggests a more complex class with multiple responsibilities.

- ProductionController: This class has a high RFC because it interacts with Vat, Batch, InventorySystem, and RecipeLibrary. This reflects its role as the main coordinator of production operations.
- InventorySystem and RecipeLibrary: These classes have a moderate RFC since they manage their specific tasks without calling too many external methods.
- Other Classes: Classes like Vat and Batch keep their RFC low, with simple methods that focus on just a few operations.

While the high RFC in ProductionController is expected due to its coordinating role, breaking this class into smaller, more specialized controllers could help reduce its complexity.

1.6 LCOM (Lack of Cohesion of Methods)

LCOM measures how well the methods within a class work together toward a single purpose. High cohesion is a sign of good class design.

- Batch, Vat, and Recipe Classes: These classes show high cohesion because their methods all focus on specific data and perform related tasks, like managing batch status or storing recipe ingredients.
- ProductionController: This class has medium-level cohesion since it handles various operations, from batch creation to transfer and monitoring.
- BreweryControlSystem: This class demonstrates good cohesion by focusing solely on user interaction and menu navigation.

Raj and I discussed this metrics almost over 90% time in whole discussion, his feedback highlighted some cohesion concerns with the ProductionController. I've simplified some of the logic to address this, but there's still more work to do to improve cohesion across all operations.

2 Summary (Coherent Discussion of the Results of These Metrics)

In analyzing the complexity of the Brewery Control System, several key insights emerge that highlight both strengths and areas for improvement.

- This final complexity analysis reflects the improvements made since the previous review while highlighting areas that still need attention. The ProductionController remains the most complex part of the system, with high WMC, RFC, and moderate LCOM. This complexity emphasizes its role as the main coordinator but also indicates a need for further refactoring
- The DIT and NOC are low, which keeps the structure simple but limits opportunities for code reuse. Coupling is generally acceptable, although some classes, like ProductionController, are inherently more coupled due to their coordinating role.
- On the other hand, data-centric classes like Batch and Vat show good cohesion, while multi-purpose classes like ProductionController have room for improvement. Overall, metrics like WMC and RFC reflect the complexity of the controllers, and splitting large controllers into smaller ones would enhance modularity and make the system easier to maintain.
- While the overall structure is functional, it has its imperfections. The design ensures simplicity where needed but compromises on flexibility and code reuse.