Inheritance in API Implementation

How does the superclass and subclass structure, along with variable naming conventions, affect the time spent and the number of errors made in Java API implementation?

Roshan Sood[1] **and Juan Yin**[1]

[1] *University of California, San Diego*

Abstract

In the realm of software development, particularly in Java, the structuring of APIs presents unique challenges and complexities for the developer. This study delves into the intricate world of Java's inheritance system, exploring how the use of superclass and subclass structures, alongside variable naming conventions, impacts the efficiency and effectiveness of API implementation for developers with intermediate Java experience(1-5 years). We conducted one pilot study and four separate studies with 4 people from different backgrounds, with 4 of them providing interview data. We found that the use of inheritance along with long variable names in Java made it harder to implement APIs. Keywords: Inheritance, API, Java.

Keywords: Inheritance. API. Java. Variable Names.

1 Introduction

Application Programming Interfaces (APIs) play a crucial role in modern software development, enabling diverse applications to communicate and interact seamlessly. Among programming languages, Java is widely recognized for its robustness and versatility, particularly in the context of backend and API development. However, the complexity inherent in Java's syntax and its object-oriented archetype, especially the inheritance structure involving superclasses and subclasses, can pose significant challenges for developers, particularly those with intermediate experience.

Inheritance, a fundamental concept in Java and all object oriented programming languages, allows developers to create a new class based on an existing class, thus enabling code reuse and reducing redundancy. This concept, while powerful, brings its own set of complexities in API implementation, particularly in terms of error handling and debugging practices. This study suggests that the use of inheritance in Java can simplify the design and implementation of APIs for developers with 1-5 years of experience. It examines how a well-structured approach to object modeling through inheritance could lead to fewer design issues, timing challenges, and a decrease in error frequency, thereby streamlining the debugging process.

Java's verbose syntax, though comprehensive, often extends the time required to debug and develop solutions, therefore impacting development efficiency. This research aims to understand whether utilizing superclass structures could make code more transparent, thus reducing errors, and how variable naming conventions might influence error rates. Longer variable names, while potentially more descriptive, can lead to increased complexity in code readability and maintenance. This complexity might inadvertently extend the debugging process, as developers may spend more time deciphering and tracking variables across the code base. Additionally, the study investigates the role of superclass in maintaining the invariance property, a crucial aspect for ensuring consistent API behavior. This paper seeks to answer the primary research question:

RQ How does the superclass and subclass structure, along with variable naming conventions, affect the time spent and the number of errors made in Java API implementation?

Based on this research question, we are specifically interested in the following subquestions:

RQ1 Why is it hard for people to implement an API using Java?

RQ2 How could superclasses and subclasses affect the invariance property for API implementation?

RQ3 How does inheritance simplify or complicate the design and implementation of APIs?

RQ4 What are the specific design and timing challenges that developers face when not using inheritance extensively?

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

We designed and conducted a pilot study and four separate studies with four participants. Section 2 will introduce the method of the experiment. Section 3 will present the results from the pilot experiments. Section 4 will mention the limitations and threats to validity of our experiments. Section 5 will discuss and interpret the results in detail.

2 Method

In this section, we describe how we design our experiments and how we conduct them. This chapter includes the preliminary preparations for the experiment the mid-term polite study, the modification and improvement of the task and instruction content and the data collection process for all subsequent experiments with our participants.

2.1 Recruitment

In light of time constraints, the formal recruitment process for experimenters was curtailed. Despite this, the study's participants were exclusively sourced from the University of California, San Diego (UCSD), owing to regional constraints. While this limitation is acknowledged, it is crucial to emphasize that the participants selected conform to specific criteria, ensuring a foundation of proficiency and familiarity essential to the study's objectives.

- All participants possess a foundational level of programming proficiency, establishing a common baseline for engaging with the study.
- A prerequisite for participation is a minimum of one year of practical experience in Java programming.
- Each participant demonstrates a nuanced understanding of the API relevant to the study.
- Participants are required to possess the capability to comprehend and navigate API documentation proficiently.

2.2 Study Procedure

Task design With the Pilot study, we improved and added:

1. Some guidance and tips
2. Provide a Java library for reference to the equations that need to be called
3. Provide more explanatory comments in the starter code
4. Provide the page of rapid API for reference output

We also finalized a flow chart suitable for experimenters to follow:

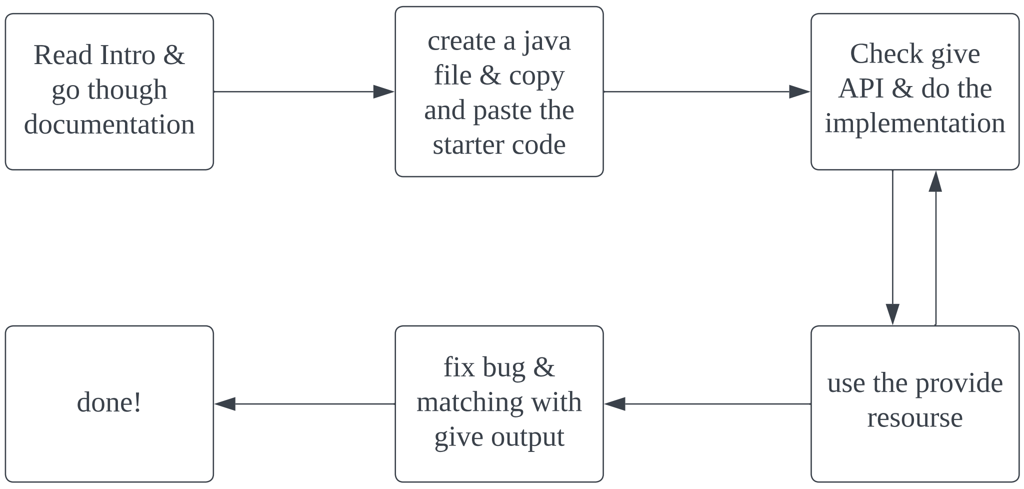

Read Intro & go though documentation
create a java file & copy and paste the starter code
Check give API & do the implementation
use the provide resourse
fix bug & matching with give output
done!

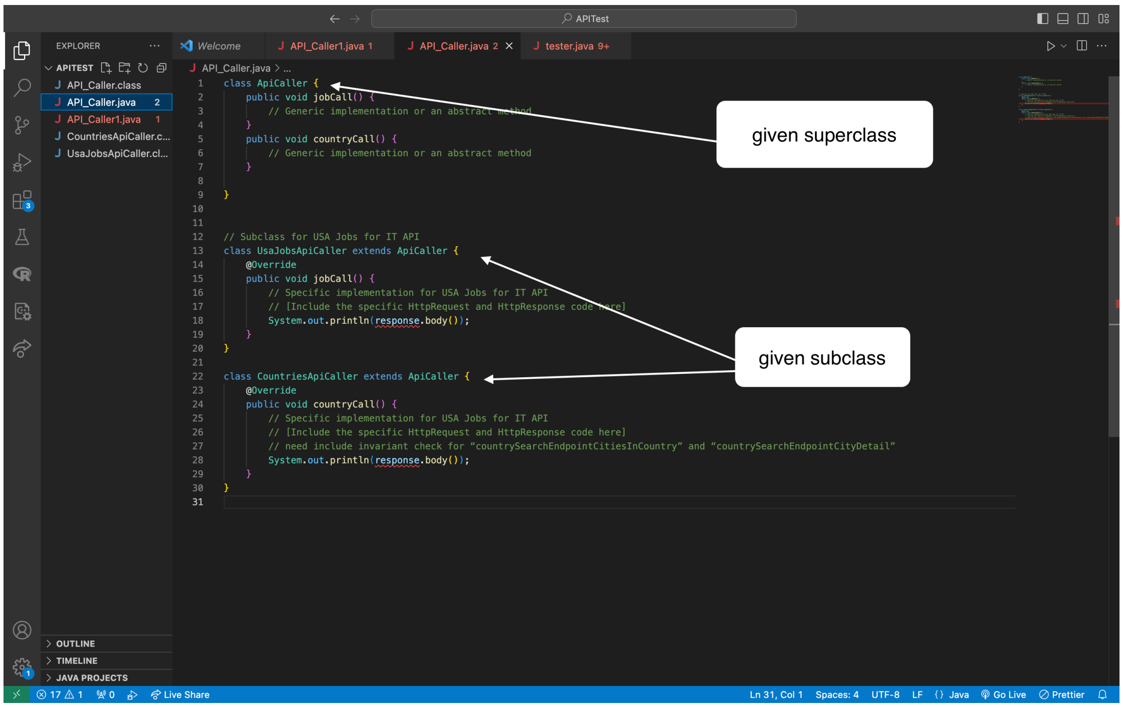
Figure 1. Procedure of Study

Starter code In order to explore whether detailed variable naming will affect the efficiency of people using the API, we also set endpoints to variables and used some more detailed variable names. There are six endpoints in total, and participants are asked to use the name of the variable rather than the endpoint itself:

Listing 1. Initialize endpoints variables

```
String jobsSearchEndpointFullstack
    = "https://usa—jobs—for—it.p.rapidapi.com/FullStack";
String jobsSearchEndpointDataEngineer
    = "https://usa—jobs—for—it.p.rapidapi.com/DataEngineer";
String jobsSearchEndpointBusinessIntelligence
    = "https://usa—jobs—forit.p.rapidapi.com/BusinessIntelligence";
String countrySearchEndpointCitiesInCountry
    = "https://countries—cities.p.rapidapi.com/location/country/US/city/list?
String countrySearchEndpointCityDetail
    = "https://countries—cities.p.rapidapi.com/location/city/5128580";
```

Below is the starter code we generated for participants(they could modify anything other than the superclass - subclass structure):



```java
class ApiCaller {
    public void jobCall() {
        // Generic implementation or an abstract method
    }
    public void countryCall() {
        // Generic implementation or an abstract method
    }

}


// Subclass for USA Jobs for IT API
class UsaJobsApiCaller extends ApiCaller {
    @Override
    public void jobCall() {
        // Specific implementation for USA Jobs for IT API
        // [Include the specific HttpRequest and HttpResponse code here]
        System.out.println(response.body());
    }
}

class CountriesApiCaller extends ApiCaller {
    @Override
    public void countryCall() {
        // Specific implementation for USA Jobs for IT API
        // [Include the specific HttpRequest and HttpResponse code here]
        // need include invariant check for “countrySearchEndpointCitiesInCountry” and “countrySearchEndpointCityDetail”
        System.out.println(response.body());
    }
}
```

Figure 2. Starter Code Interface Template

2.3 Research Method

Think aloud Method We have chosen the think-aloud method as our primary data collection technique to meticulously uncover participants' cognitive processes during API implementation. Unlike traditional methods such as data mining, think-aloud allows us to delve deep into the intricate details of participants' decision-making, revealing correct or incorrect operations.

For Research Question 1, where we aim to understand why it is challenging for people to implement an API using Java, we contend that the degree of difficulty cannot be adequately captured through passive data collection alone. By employing the think-aloud method, we intend to gather participants' real-time opinions on the level of complexity they perceive. This includes insights into their assessments of code completion, the time invested in implementation, and their experiences with periodic bugs that manifest during the coding process.

The think-aloud approach provides a dynamic lens into the participants' mental models as they navigate through the challenges of API implementation. This methodological choice allows us to extract richer, context-specific data that goes beyond the mere correctness of the code, enabling a comprehensive analysis of the challenges inherent in API development with Java.

Post-Task Structured Interview After the task, we conducted a structured interview, posing two fixed questions:

Q1: What do you think of the provided starter code and variable definitions?

Q2: If you need to implement three or more APIs, will you still use the same coding structure as now?

The detailed answers will be introduced in the results section later.

2.4 Data Collection

Coding Part We found a total of five participants and conducted the experiment by asking them to watch a unified instruction. We analyzed six aspects in total:

1. Whether participants get the correct output
2. Whether participants use the inheritance structure and how they feel about using this structure
3. How participants allocate and use variables
4. Number of error messages
5. Types of errors
6. Time to participate in the test

To collect data, we created a form:

Participants	Right output	Inheritance	Variable
Participant 1	right	Did not use inheritance	did not use the name
Participant 2	wrong	Did not use inheritance	N/A
Participant 3	right	Did not use inheritance	used variable name
Participant 4	right	Did not use inheritance	used variable name
Participant 5	right	used inheritance	did not use the name

number of error	Type of error	Time used
4	Java syntax errors	40min
3	Java runtime error(using try), Java syntax errors	40min
4	Java syntax errors Subitem Y Subitem Z	20min
3	Java runtime error(using try)	30min
3	inheritance error, Java wrong output	40min

Table 1. Table with 6 Aspects we collected after test

Interview Part Regarding structured-interview, For Q1, one participant thought "I don't think superclass plays a role in the code, it is very redundant", and the other three participants all thought "had an easier time implementing without inheritance", and One participant cannot use inheritance to get the correct return value. Regarding endpoints and their name definitions, one participant pointed out that "long variable names made it more confusing". Another participant said "I can accept these long names, but I won't define them that way if I have to define them myself". On this basis, I asked a follow-up question: "What if your coding needs to be shown to others?" The participant hesitated for a moment and then said, "Then I may also define an explanatory name." Although additional questions are not allowed in structured interviews, given that we only had five participants, I included this data because I think it is valid. Regarding Q2, when it comes to multiple APIs, most participants believed that inheritance would be helpful for multiple API implementations.

2.5 Data Analysis

Qualitative: The majority of qualitative data in our experiment comes from the recorded data during each interview. We also used the video recording when we found any unusual events. To analyze our data, we used Java and excel to plot and record our data.

Quantitative: The majority of quantitative data came from analysis during each interview. We used metrics such as time tracking and error rate measurement to measure difficulties within the experiment. We also used the video recording when we found any unusual events. We tracked the time it took to complete each task when implementing the API and recorded the number of errors each task yielded. We then categorized and summarized all of the data. Due to time constraints, we had to manually time each task it took and manually record the number of errors that occurred instead of using any special software to record data.

3 Results

Due to our relatively small number of participants, we will focus on qualitative analysis. We will also analyze the answers obtained from the coding tasks and interviews conducted through think-aloud. Combined with the results we obtained, we will analyze the research questions we raised again.

3.1 Participants

Here's all the information we have about the participants: Participant 1, Participant 2, Participant 4, and Participant 5: UCSD undergraduate students who have systematically studied Java courses, but may lack practical hands-on experience. Participant 3: UCSD graduate student, has rich experience in using Java, and has participated in relatively large projects using Java.

3.2 results

Syntax Errors and Java Documentation: Through the analysis of error types in Table 1, it became evident that Java syntax errors are prevalent, even among experienced participants. Notably, observations revealed specific challenges faced by participants:

Insufficient Information in Java Documentation: The first participant encountered difficulties finding comprehensive information in the Java Documentation and resorted to web searches.

Importance of IDE Familiarity: The third participant efficiently used the "Quick Fix" function provided by the IDE, showcasing the significance of familiarity with the development environment.

From these observations, two key conclusions emerged:

1. Java Documentation Limitations: The comprehensiveness of Java Documentation is deemed inadequate, potentially contributing to the reluctance of developers to use Java for API implementation.

2. IDE Proficiency: A higher level of familiarity with the IDE correlates with quicker bug resolution, emphasizing the importance of user familiarity with development tools.

Proposed Solution: To address these challenges, we propose the development of an IDE tool with more intuitive and representative prompt options than the existing "Quick Fix" feature, aiming to enhance user experience and reduce the likelihood of users overlooking critical information.

Inheritance Usage in Java Projects: Analyzing data from Table 1, it was observed that only 20 present of participants utilized inheritance in their Java projects. The one participant who used inheritance stated having an easier time without it, while another participant expressed a preference for encapsulating code into functions in the absence of starter code.

Limited Applicability of Inheritance: The low usage of inheritance suggests that, for small Java projects, it may not be considered beneficial. Need for Further Research: The impact of inheritance on large Java projects remains inconclusive, warranting additional research to determine its effectiveness in different project scales.

Variable-Related Data: Examining variable-related data in Table 1 revealed that participants exhibited greater tolerance for longer variable names, possibly due to familiarity with this naming pattern. However, further research is necessary to understand optimal lengths and patterns that prevent error messages.

We need More for Research Question 4: For Research Question 4, the available data appears insufficient to draw conclusive insights. Addressing this issue necessitates a broader participant pool and a set of comparative experiments. The goal is to explore the suitability of the inheritance structure for specific project types and investigate potential correlations between inheritance usage and code length.

4 Limitations

Participant Background • Except for a small sample size, we identified several other threats to internal validity of the study. We wished to recruit programmers with similar experience in Java. However, we couldn't find such an array of participants. Thus, the results might be biased due to participants with lower familiarity with programming and Java.

Task Selection We selected relatively easy snippets of code to implement APIs in Java. Thus, our result cannot be generalized to more complex problems.

Environment Setup Our environment setup is different from those used in the interview or modern-day programming jobs. Programmers often use different editors and tools to implement and test API endpoints, however we provided vscode and its built in terminal.

Variety of Bugs We gave participants a code snippet to implement given API(s). It is unpredictable what kind of bugs they will encounter, even though we designed our code snippets to yield specific errors. The participant might waste their time on logical errors unrelated to implementing an API. In a future study, we might be able to reduce this threat by having the participant debug a piece of code as the correct answers can be controlled.

Time Constraints The time allocated for completing tasks might not accurately reflect real-world development timelines, which can vary greatly based on the complexity of the API. Participants only had one hour to complete a full API implementation in several different tasks, a duration that may not be sufficient to fully encapsulate the intricacies and challenges of actual development scenarios. Therefore, time constraint might have an undue pressure on the participants.

5 Discussion

The results of our study, while insightful, are constrained by the limitations of our small pilot study, thus lacking statistical significance. Despite these constraints, we draw tentative inferences about Java API implementation using inheritance.

We originally assumed that inheritance would make it easier to implement an APi as using superclasses would make the code structure clear, so participants will make less mistakes when implementing code. However, our experiment refuted our claim as every participant took less time implementing APIs without using inheritance, while also experiencing less error messages as well. All participants encountered bugs when using the subclass to implement the API call, but were able to fix their mistakes through time. We might conclude that the use of inheritance does not make the implementation of APIs in Java easier. These results were surprising as we assumed inheritance would organize the code and make it easier to read, but the results directly opposed that.

The interview results confirmed the results from the qualitative data, emphasizing that inheritance takes longer and makes it harder to implement APIs in Java.

6 Future Work

We found through the interview answers that most participants agreed that "it is necessary to use inheritance when implementing many APIs." However, I did not add additional questions such as "Why do you think it is necessary? Is it from the user's perspective?" A programmer's perspective on compiling programs? I think this issue needs further study."

We also found that when one of the users used Copy/Past to code different APIs, Java would not report an error even if the function was changed. I think this is a potential type of error. If this error is ignored by people, they may take longer to debug because it is difficult to find the location of the error without error information.

Before delving further into the field of Java API implementation using inheritance, it is crucial to assess the actual worth of such studies. While understanding the complexities of Java's inheritance structure and variable naming conventions is important, it is also essential to evaluate the real-world benefits of optimizing these aspects. We might need to perform long-term experiments to fully determine the effect of inheritance on implementing APIs in Java so that participants have more experience with the tools and environments they are provided.

However, a long term project will yield higher costs and should be accounted carefully. Moving forward, refining this experiment by addressing the limitations outlined previously is a priority. Identifying ways to mitigate the identified threats will enable application of the study to a broader range of scenarios and contexts within Java development.

While our study concentrated on the use of superclasses and subclasses in Java for API implementation, it remains unclear how these practices compare with other programming languages used for similar purposes. Therefore, conducting comparative studies across different programming languages and their respective approaches to API development would be beneficial. Such research could provide a more holistic understanding of the best practices in API design and implementation across various programming environments.

Considering the rapid evolution of programming tools and environments, it is important to continuously evaluate and integrate emerging technologies and methodologies into Java API development practices. This underscores the need for adaptive and forward-looking research in the field.

Although we studied inheritance in Java with APIs, it is not clear if all developers should use Java to implement APIs. Additional studies on the topic of "programming language choice for API implementation" can be done before we explore this project further.

7 Conclusion

In this paper, we designed an experiment to verify the research question

- How does the superclass and subclass structure, along with variable naming conventions, affect the time spent and the number of errors made in Java API implementation?

We piloted our experiment with four participants and found that

- Inheritance makes it harder for participants to implement an API
- Participants do not prefer implementing APIs with inheritance
- Inheritance yields more errors when implementing an API
- It takes longer to implement an API when using inheritance
- Java documentation is old and outdated

Lastly, we have acknowledged the limitations of our current study and proposed several areas of improvement for future research, particularly focusing on the use of tools and programming language strategies in the context of Java API development.