PA2577 All at Once Article Summary

Samuel Jonsson

January 2, 2024

1 Part 1

1.1 Article 1

Article 1 is *Software performance antipatterns* by Smith and Williams (2000). The authors discuss the concept of software design patterns and antipatterns, focusing on their impact on performance. They describe the difference between design patterns and anti patterns; established solutions to common software design problems vs. recurring design mistakes that lead to negative consequences.

The article introduces four performance-related antipatterns:

- **The God Class**
- **Excessive Dynamic Allocation**
- **Circuitous Treasure Hunt**
- **One Lane Bridge**

The article emphasizes the importance of considering performance consequences when using design patterns and antipatterns. It suggests that performance-related antipatterns help developers recognize and avoid common performance pitfalls, enhancing software architecture and design.

Measurements

The main measurements discussed in the article are related to the performance impact of the various software design antipatterns they discuss, evaluating the negative consequences of these antipatterns on software performance. The main measurements are:

- **Message Traffic**: The article discusses how the "*God Class*" antipattern can lead to excessive message traffic between classes, negatively affecting performance. It emphasizes the increase in the number of messages required to perform operations and its impact.
- **Dynamic Allocation Overhead**: In the "*Excessive Dynamic Allocation*" antipattern, the article measures the overhead caused by frequent creation and destruction of objects. It highlights the performance impact of dynamic allocation, especially in scenarios with high object turnover.

- **Database Access Patterns**: For the "*Circuitous Treasure Hunt*" antipattern, the article discusses the performance consequences of database access patterns. It measures the inefficiency of retrieving data in a way that requires a large number of database calls, particularly in distributed systems.

- **Service Time**: The article quantifies the impact of reducing service time, particularly in the context of the "*One Lane Bridge*" antipattern. It discusses how minimizing the time required for specific operations can improve overall responsiveness.

Main findings

The main findings of the article are that design patterns and antipatterns are valuable tools in software development for capturing expert knowledge and best practices, as well as identifying common design mistakes and their solutions. The authors highlight the need for both design patterns and antipatterns that explicitly address performance issues, as well as the importance of building performance intuition among software developers.

Overall, the article underscores the significance of integrating performance considerations into the design and development process, alongside the use of design patterns and antipatterns, to create more efficient and responsive software systems.

1.2 Article 2

Article 2 is *New software performance antipatterns: More ways to shoot yourself in the foot* by Smith and Williams (2002). The article is a continuation to Article 1 and introduces four new performance antipatterns:

- **Unbalanced Processing**
- **Unnecessary Processing**
- **The Ramp**
- **More is Less**

The article emphasizes the importance of identifying these antipatterns early in the software development process, using models or measurements to assess their impact, and providing solutions that adhere to performance principles. Performance antipatterns help build developers' performance intuition and complement traditional design and architectural patterns.

Measurements

Article 2 does not contain specific measurements or quantitative data. Instead, it provides a conceptual understanding of the antipatterns and emphasizes the importance of identifying and addressing them in software development.

Main findings

Like article 1, the article emphasizes the importance of identifying antipatterns early in the software development process to avoid scalability and performance problems. It also highlights the role of models and measurements in identifying and mitigating these issues.

1.3 Article 3

Article 3 is *More new software performance antipatterns: Even more ways to shoot yourself in the foot* by Smith and Williams (2003). The article aims to expand the documentation of software antipatterns that started in article 1 and continued in article 2. The article introduces three additional antipatterns:

- **Falling Dominoes**
- **Tower of Babel**
- **Empty Semi Trucks**

Measurements

Like article 2, article 3 does not contain any quantitative measurements

Main findings

Like article 1 and 2, article 3 emphasizes the importance of identifying antipatterns early in the software development process to avoid scalability and performance problems. It also highlights the role of models and measurements in identifying and mitigating these issues. At the end, the authors list the currently documented antipatterns (14 in total).

1.4 What can you learn?

Since My answer two all three *What can you learn?* questions will be marginally different, I'll just summarize it in one.

Due to me having done the Software Architecture course I allready knew some of what was discussed in the articles. However, it was interesting to see quantitative data on the effects of some of the antipatterns, and I had not heard of a few of them, or at least not the names used, duch as the *Circutious Treasure Hunt* or *Empty Semi Trucks* patterns.

2 Part 2: Discussion

2.1 Final state

Metric	Value
Completed after	04:05:26:776
Files Count	163,589
Chunks Count	18,211,863
Candidates Count	584,748
Clones Count	24,961
Average File Processing Time	0.0686 s
Average Chunk Processing Time	2.6145 s
Average Clone Expansion Time	0.4594 s
Average Clone Size	108.7991 lines
Average number of chunks per file	159.8541 chunks

Table 1: The final state after running the complete Qualitas Corpus

2.2 Are you able to process the entire Qualitas Corpus? If not, what are the main issues (think in terms of data processing and storage) that causes the cljDetector to fail? How may you modify the application to avoid these issues?

I am able to process the entirety of the Qulitas Corpus, even on my bad laptop, which took 8 hours when not running it with my monitortool service or other programs at the same time.

Some edits I think you could do to save storage would be to what is done in the clone identification step, i.e. remove files as they're chunkified and chunks as they've been turned into candidates, or filtered out; $18 * 10^6$ chunks is a lot to store, and so are 163 thousand files. In the end, the purpose of the program is to identify clones, and not to store the files after all.

Another option could be to utilise another form of database. While NoSQL databases such as MongoDB are more horizontally scalable than relational databases, the type wof application we have will not be distributed and run on multiple instances (at least not with the purpose we have been using it in this course), nor do we really need the flexibility of NoSQL as the database schema seems fairly rigid to me. Beacuse of this, a vertically scalable database, such as SQL, might be a superior option compared to a NoSQL one. I have also read that the way you can use `JOIN`, and similar queries, in relational databases make sorting out any clone candidates much easier and efficient, compared to the options available in MongoDB. An example would be the `identify-candidates` function in from `storage.clj`:

```
(defn identify-candidates! []
  (let [conn (mg/connect {:host hostname})
        db (mg/get-db conn dbname)
        collname "chunks"]
     (mc/aggregate db collname
        [{$group {:_id {:chunkHash "$chunkHash"}
                :numberOfInstances {$count {}}
                :instances {$push {:fileName "
                  $fileName"
                                   :startLine "
                                     $startLine"
                                   :endLine "$endLine
                                     "}}}}
          {$match {:numberOfInstances {$gt 1}}}
          {"$out" "candidates"} ])))
```

which can be done in SQL like this:

```
SELECT chunkHash,
    COUNT(*) AS numberOfInstances,
    ARRAY_AGG(STRUCT(fileName, startLine, endLine)) AS
        instances
FROM chunks
GROUP BY chunkHash
HAVING COUNT(*) > 1;
```

2.3 Is the time to generate chunks constant or does it vary with the number of already generated chunks?

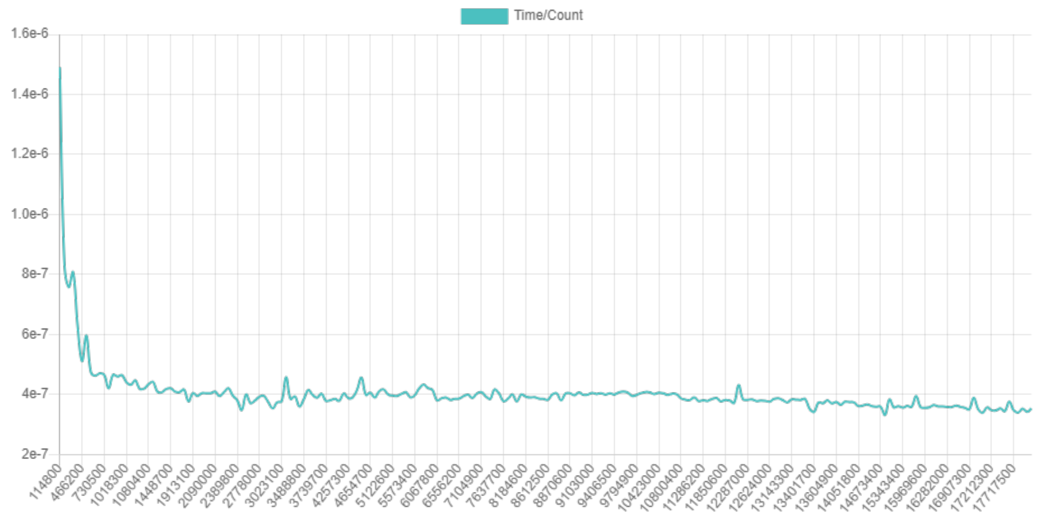

Figure 1: Graph over average processing time/chunk count. Average processing times are mapped on the y-axis and chunk counts are mapped on the x-axis[2].

The from what I can see in the charts, the file storing and chunk generation seems to be somewhat constant. Ignoring the "high" initial time, which could just be caused by how I wrote my monitor function, the line is pretty much straight horizontal, ignoring the small fluctuations.

2.4 Is the time to generate clone candidates constant or does it vary with the number of already generated candidates?

Due to the code utilising the aggregate function form the `mc` library for Clojure, the code doesn't actually add the clones to the database until it is completed. This means I don't have any means of measuring this from the original code without making large changes to the it. However, if I investigate the documentation for the function, it doesn't seem to function in a way that it would be affected by the number of already identified candidates and works similarly to the `STRING_AGG` function in SQL.

[2]Due to issues with the plotting package I'm using for the website, I couldn't get the axis labels to work properly

2.5 Is the time to expand clone candidates constant or does it vary with the number of already expanded clones or the number of remaining candidates?

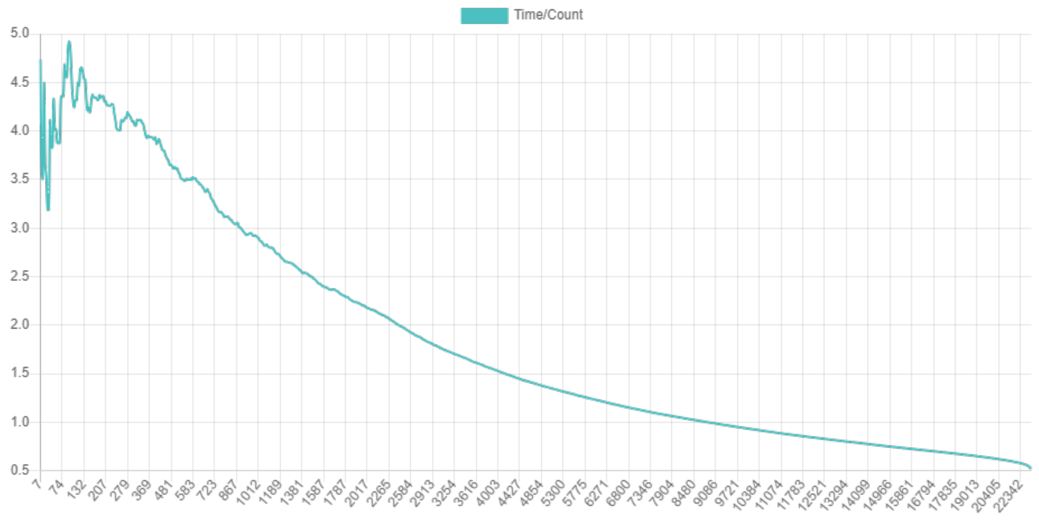

Figure 2: Graph over expansion times/clone count. Average expansion times are mapped on the y-axis and clone counts are mapped on the x-axis.

As can be seen in Figure 2, the time it takes to expand the clones becomes lower as more clones are identified, moving towards zero (the fact that the last value is zero likely has to do with monitortool, not the data.). I suspect this is because we remove candidates as they're expanded, leading to fewer candidates to look through for each iteation.

2.6 What is the average clone size? How can this be used to predict progress during the expansion phase?

The average clone size is 108.7991 lines. This can be used to assume the speed the expansion times will decrease, as the more lines a clone have, the more chunks it will consist off, which in turn will decrease the number of candidates in the database faster, and fewer candidates to look through means faster expansion time decrease rate.

2.7 What is the average number of chunks per file? How can this be used to predict progress during the read/chunkify/identify candidates stages?

With `CHUNK_SIZE` set to 20, I got 159.8541 chunks on average per file (calculated as num_chunks/num_files). While not directly affecting the file reading step, more chunks implies longer files, which will take more time to read. It will also imply that more chunks are created, meaning that the line that the average reading and chunkification times fluctuate around will be higher, but not likely fluctuate more.

On the identify candidates section, however, more chunks will mean that each chunk will be smaller. This means that the likelihood that a clone candidate is identified, e.g. if we have a piece of code A:

```
for i in range(5):
    print("Iteration:", i)
    print("Different line 1")
    print("Different line 2")
    print("Different line 3")
    print("Different line 4")
```

and another piece of code B:

```
for i in range(5):
    print("Iteration:", i)
    print("Another different line 1")
    print("Another different line 2")
    print("Another different line 3")
    print("Another different line 4")
```

if we set a chunk size of two lines, this piece:

```
for i in range(5):
    print("Iteration:", i)
```

would become a clone candidate, however, if we instead have a chunk size of four, there would be no clone candidates in the pieces of code.

In short, the average number of chunks per file have the potential to either describe the average length of a file, which would affect the time it would store all files, as well as how finely chunkified each file is, which would potentially affect how many clone candidates are identified.

References

Smith, C. U., & Williams, L. G. (2000). Software performance antipatterns. In *Proceedings of the 2nd international workshop on software and performance* (pp. 127–136).

Smith, C. U., & Williams, L. G. (2002). New software performance antipatterns: More ways to shoot yourself in the foot. In *Int. cmg conference* (pp. 667–674).

Smith, C. U., & Williams, L. G. (2003). More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer measurement group conference* (pp. 717–725).