twoday

May 2024

twoday Application Modernization Pipeline

by Mantas Urbonas and Eugenijus Medelis

Summary

This paper describes techniques for unsupervised (automated) large scale tech debt cleanup performed by twoday AMP tool. This technique has only become feasible due to the recent advances in generative AI.

The tool guarantees preservation of exact original semantics of the source code before and after the refactoring. This security guarantee, however, puts a limitation on the types of refactoring that the tool is able to do, which are discussed in detail within the paper.

Introduction

A large portion of software currently in use contains a significant level of tech debt. Various surveys give somewhat different statistics but the problem is sizable[1]. Tech debt can be assessed in different ways, and there are multiple specialized tools analyzing code bases – such as Sonar Cube, Cast Software, Code Scene being several examples.

Obviously, the detection (and analysis) of tech debt is not solving the actual problem. Each identified instance of problematic code still needs to be fixed manually, either by editing the code fragment manually or with some help from IDE. Most modern language ecosystems have all kinds of means for manually triggered code improvements (e.g. Built-in IDE features, OpenRewrite for Java, ReSharper for C# etc.). Generative AI tools, such as Github Copilot can offer intelligent code refactoring, however they need a close supervision from human programmers as the Generative AI is still very unreliable (the produced code tends to introduce new bugs or change original semantics).

In this paper we present AMP - our novel Generative-AI powered source code improvement tool, that **eliminates certain types of tech debt in unsupervised (fully automated) manner while preserving the exact original code semantics.** We do not aim to eliminate 100% of tech debt; rather, we address only specific instances that (a) guarantee 100% identical semantics after the transformation; and (b) have high probability of readability improvements.

Such functionality was unattainable before the advent of Generative AI.

[1] https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/demystifying-digital-dark-matter-a-new-standard-to-tame-technical-debt

Motivation

The examples below are simplified and distilled fragments from actual production code, each originally exceeding 500 lines. The code became convoluted for various reasons, such as time pressure, absence of quality control mechanisms, or insufficient developer expertise. The current maintenance team is hesitant to modify this code as it has a Cognitive Complexity index > 70, indicating that it is difficult to fully comprehend.

Example 1:

```
/* BEFORE */

void print(Param param) {
  if (param != Param.one) {
     if (param != Param.two) {
         System.out.println("three");
     } else {
         System.out.println("two");
     }
  }
  else
  if (param != Param.two) {
     if (param != Param.three)
         System.out.println("one");
     else
         System.out.println("three");
  }
  else
  if (param != Param.three) {
     if (param != Param.one)
        System.out.println("two");
     else
        System.out.println("one");
  }
}
```

```
/* AFTER: conditions distilled */

void print(Param param) {
  switch(param){
    case one: printParamOne(param); break;
    case two: printParamTwo(param); break;
    case three: printParamThree(param); break;
    default: printParamDefault(param); break;
  }
}

private void printParamOne(Param param) {
    System.out.println("one");
}

private void printParamTwo(Param param) {
    System.out.println("two");
}

private void printParamThree(Param param) {
    System.out.println("three");
}

private void printParamDefault(Param param) {
    System.out.println("three");
}
```

Example 2:

```
/* BEFORE */

if (a < 0) {
  if (b < 0) {
    if (c >= 0) {
      // actual processing here
      ...
    }
    else {
      throw new RuntimeException(
                 "c must be positive");
    }
  }
  else {
    throw new RuntimeException(
                 "b must be negative");
  }
}
else {
  throw new RuntimeException(
                 "a must be negative");
}
```

```
/* AFTER: early return guards */

if (a >= 0)
    throw new RuntimeException(
                    "a must be negative");
if (b >= 0)
    throw new RuntimeException(
                    "b must be negative");
if (c < 0)
    throw new RuntimeException(
                    "c must be positive");

// actual processing here
...
```

We aim to transform the code into a form that is easier to read and maintain. AMP performs similar transformations across entire codebase.

Note that out-of-the-box generative AI falls short of this goal. Even for a small code fragment, LLM generates code with subtle differences in semantics, which most likely will mean unexpected behavior in production.

Goals and Scope of AMP

AMP aims to improve maintainability of large *legacy code bases*. More specifically, it targets code bases with > 100 K lines of code that contain significant number of methods with cognitive complexity > 50. Typically, such software contains parts that are hard to comprehend, and the current maintainers are reluctant to make changes due to a fear of accidental defect introduction.

For such codebases, AMP improves code readability up to the point where human maintainers can become productive again. In certain cases, code readability improvements allow easy discovery of logical bugs. Best improvement is seen on domain logic codebases, regardless of if its UI, server-side or monoliths.

A fundamental design principle of AMP is *preservation of the exact original code semantics*. Anymore "drastic" code modifications are instead left for human maintainers to take care of.

Another design principle is transformation in atomic, well-defined steps rather than large leaps.

There are certain aspects that are **outside the scope** of AMP:

- Changing of the coding paradigm, i.e. from imperative to declarative or from OOP to functional.
  While some programmers may prefer different coding styles, our *goal is simplifying the original code* rather than a complete re-write in another paradigm.
- Architecture-level refactoring.
  AMP focus is on eliminating code smells at the method- and class level, rather than architectural level fixes.
- Generation of unit tests or code documentation.
  There are other technologies that use Generative AI for such goals.
  Note however that AMP guarantees identical functionality after the transformation, hence we can transform legacy code that either does or does not have unit test coverage.
- Version updates.
  No dependencies, libraries, tools or language versions are changed, as this involves high risks.

Some of the Code Transformations

Extracting *condition trees*

By “condition trees” we mean any non-trivial conditional statement that focus solely on assigning a value to a single variable. It is a specific case of multi-branch IF statement.

A condition tree example might look like this:

```
DateTimeFormatter formatter;

if (pattern == null || pattern.equals(DEFAULT_FORMAT))
   formatter = ISO_FIXED_FORMATTER;
else if (pattern.length()==5) {
   if (isNumeric(pattern))
       formatter = TIMESTAMP_FORMATTER;
   else
       formatter = INVALID_FORMATTER;
}
else if (configSettings.hasFormatter(pattern))
   formatter = configSettings.getFormatter(pattern);
else
   formatter = DEFAULT_FORMATTER;
```

Such condition trees are ideal candidates for method extraction because a complex structure gets replaced by a single call to a low-level method with a clear single responsibility, i.e.

```
DateTimeFormatter formatter = getFormatter(format);
```

Multiple syntactic and semantic preconditions must be met for a condition tree extraction. For example, this block should NOT be extracted because one branch (line 5) is an early return:

```
1    if (condition1)
2       value = ISO_FIXED_FORMATTER;
3    else
4    if (condition2)
5       return something;
6    else
7       value = DEFAULT_FORMATTER;
```

AMP performs multiple validations at abstract syntax tree level ensuring semantic and syntactic correctness before continuing with method extraction.

Modifying condition trees prior to extraction

In certain cases AMP first attempts certain modifications of the original condition tree. Consider a similar example where the IF statement (lines 1 to 7) cannot be directly extracted into a separate method:

```
1   if (pattern == null || pattern.equals(DEFAULT_FORMAT))
2      formatter = ISO_FIXED_FORMATTER;
3   else
4   if (pattern.length()==5)
5      return FIVE_CHARS_FORMATTER.format(input);
6   else
7      formatter = DEFAULT_FORMATTER;
8
9   return formatter.format(input);
```

However, the code above can be safely transformed into this (equivalent) form:

```
1   if (pattern == null || pattern.equals(DEFAULT_FORMAT))
2      return ISO_FIXED_FORMATTER.format(input);
3   else
4   if (format.length()==5)
5      return FIVE_CHARS_FORMATTER.format(input);
6   else
7      return DEFAULT_FORMATTER.format(input);
```

which finally can be extracted into a separate high-level method leaving only

```
1   return formatByPattern(pattern, input);
```

One might argue that an intermediate step is not strictly necessary, as a single-step action could achieve the same result (i.e. extracting the IF statement together with the following RETURN statement, lines 1 to 9). Most likely an experienced human programmer would do this in one step, marking these lines in IDE and invoking the “extract method” refactoring.

However, as a general principle, we prefer to employ multiple simpler transformation steps whenever possible. Moreover, we have observed that generative AI provides much better method naming for a “normalized condition tree” form.

Distilling conditional statements (extracting the State pattern)

Consider the following code fragment:

```
    void print(Param param) {
        if (param != Param.one) {
            if (param != Param.two) {
                System.out.println("three");
            } else {
                System.out.println("two");
            }
        }
        else
        if (param != Param.two) {
            if (param != Param.three)
                System.out.println("one");
            else
                System.out.println("three");
        }
        else
        if (param != Param.three) {
            if (param != Param.one)
                System.out.println("two");
            else
                System.out.println("one");
        }
    }
```

Here multiple condition branches test against the same invariant `param`. Such code form is quite confusing and difficult for humans to understand.

Interestingly, such code is also somewhat confusing for LLMs. Here are the outputs from several top-ranking models[2]:

ChatGPT 4

```
void print(Param param) {
  switch (param) {
     case one:
       System.out.println("one");
       break;
     case two:
       System.out.println("two");
       break;
     case three:
       System.out.println("three");
       break;
     default:
       // Handle unexpected cases
       //  or do nothing
       break;
  }
}
```

CodeLlama 70b-instruct

```
void print(Param param) {
    if (param == Param.one) {
        System.out.println("one");
    } else if (param == Param.two) {
        System.out.println("two");
    } else if (param == Param.three) {
        System.out.println("three");
    }
}
```

[2] prompt: "*Refactor the following Java code, preserving the exact original semantics*"

Although the LLM refactored code seems convincing at a first glance, it actually has a subtle change in original semantics. Here is the AMP-transformed code that preserves the exact entire original behavior:

```
void print(Param param) {
   switch(param){
       case one:  printOne(param); break;
       case two:  printTwo(param); break;
       case three:  printThree(param); break;
       default:  printDefault(param); break;
   }
}

private void printOne(Param param) {
   System.out.println("one");
}

private void printTwo(Param param) {
   System.out.println("two");
}

private void printThree(Param param) {
   System.out.println("three");
}

private void printDefault(Param param) {
   System.out.println("three");
}
```

Here AMP carefully examined the original code fragment and *distilled* the lines for each of the possible variable value. Every distilled case is extracted to a separate method resulting in a much cleaner form.

We believe that such refactoring is a good step toward a better object oriented design: this code form now enables (and encourages) the human programmer to apply a *“Replace Conditional With Polymorphism”* refactoring:

```
interface IParamSpecificAlgorithm {
    void print();
    // not covered by examples above, but following the same pattern:
    void verifyInput(String input);
    void encrypt(String content);
}

void print(Param param) {
    getParamSpecificAlgorithm(param).print();
}

void verifyInput(String input, Param param) {
    getParamSpecificAlgorithm(param).verifyInput(input);
}

void encrypt(String content, Param param) {
    getParamSpecificAlgorithm(param).encrypt(content);
}
```

twoday

Extracting code blocks into methods

We strongly support notion that long methods with multiple responsibilities are difficult to comprehend and maintain:

```
if (object instanceof int[]) {
    // 10 lines of serialization of int[] array
    ...
}
else
if (object instanceof short[]) {
    // 10 lines of serialization of short[] array
    ...
}
else
if (object instanceof long[]) {
    // 11 lines of serialization of long[] array
    ...
}
else
if (object instanceof boolean[]) {
    // 10 lines of serialization of boolean[] array
    ...
}
...
```

In comparison, this form is much more readable and maintainable:

```
if (object instanceof int[])
    serializeIntArray((int[])object);
else
if (object instanceof short[])
    serializeShortArray((short[])object);
else
if (object instanceof long[])
    serializeLongArray((long[])object);
else
if (object instanceof boolean[])
    serializeBooleanArray((boolean[])object);
```

For most programmers, this latter form requires less mental effort to comprehend because it has significantly less code, and that code consists of well named method calls; each of the sub-routines have single responsibility and can be inspected in isolation.

Method extraction has multiple challenges and if done poorly can downgrade the maintainability.

AMP performs multiple heuristics on abstract syntax tree to assess relationships between code elements and identify highly cohesive fragments. Next, we use modern LLMs to reason about boundaries of areas that semantically and logically have the same responsibility. Finally, we ensure that the target method is split into sufficiently large blocks so that high-level and low-level details aren’t mixed in the same method.

One important aspect to consider is that naming and coding styles are subject to personal preference. In the previous example, subroutines in lines 2,5,8 and 11 can be alternatively named “serialize(..)”, “write(..)”, or “persist (...)”. Inconsistent naming and inconsistent coding style is also a tech debt. AMP tries to make a best effort when creating new method names: therefore, we first analyze naming conventions of other methods within the same compilation unit. Also, we always prefer to only make necessary modifications only, so that the original programmers still find the code familiar.

Note that AMP performs automated, unsupervised method extraction refactorings quite late in the overall process. This is because a series of smaller (atomic) improvements may improve the original code, making the responsibility boundaries easier to identify, or sometimes eliminating the need for method extraction at all.

Microimprovements

AMP performs several other localized incremental improvements.

For example, re-ordering the conditional branches and introducing early return guard conditions makes a nominal reduction to cognitive complexity.

```
//  before AMP


if (param1 != null) {
   if (param2 != null) {
      //process parameters
      ...
   } else
       throw new IllegalArgumentException(
                   "param2 cannot be null");
} else
    throw new IllegalArgumentException(
                   "param1 cannot be null");
```

```
//  after AMP


if (param1 == null)
   throw new IllegalArgumentException(
                  "param1 cannot be null");

if (param2 == null)
   throw new IllegalArgumentException(
                  "param2 cannot be null");

//process parameters
...
```

In isolation such an improvement is not very significant. However, applying similar improvements to large complex conditional statements (spanning across dozens and hundreds of lines of code) does make a measurable difference: the cognitive load on a human reader is significantly reduced. Note that codebases having this kind of issue tend to have it across the code as well as occasional deep nesting, so even though change itself is minor, the cumulative impact of such improvements can be significant.

AMP ensures that no accidental error slips in when transforming conditions, making early returns etc.

Consequences to software asset quality

One obvious benefit for a readable code is that defects are much easier to spot. Consider this example of AMP-processed code:

```
1   // ...
2   } else if (text.length() == 23)
3       formatter = determine23charsFormatter(...);
4
5   if (text.length() >= 17)
6       formatter = determineAsianFormatter(...);
7
8   // ...
```

Originally this code fragment was much longer, and the code in lines 3 and 6 were separated by dozens of low-level codes. Now, after all the low-level code was extracted into separate methods, and the high-level method calls ended up next to each other, a human reader can easily spot that line 6 is always executed together with line 3 - which is probably not the intention. This defect was not discovered for several years.

Most often local improvements in code-level hygiene do not immediately manifest in the code design. However, we observed that after running an automated code cleanup the maintenance teams started restructuring the code, relocating methods and moving them into newly discovered classes and packages. We believe that a clean code is a prerequisite for emergent OO design.

Among other benefits, smaller methods and simplified conditionals help developers navigate within the source code and locate areas for bug fixing or feature development, this way directly improves productivity.

Arguably the largest obstacle to legacy software modernization lies in difficulties of decoupling key business logic artifacts from the legacy infrastructure. Code cleanup enables code modularization and separation of concerns, which is essential for business logic code reuse when migrating to new frameworks and platforms.

Empirical statistics

We use cognitive complexity[3] as a proxy to code-level tech debt level in the codebase.

Our tests demonstrate that AMP reduces cognitive complexity rate by up to **23%** on either medium-sized (> 15 000 LoC) and large (> 100 000 LoC) legacy Java codebases. For the most complex cases (methods with CC >100) the improvements are much more significant.

Here is the statistics from one typical medium-sized project:

		Original	**Revised**	**Change %**
Entire codebase	**Total Cognitive Complexity**	16 537	13 344	-19,31%
	Total number of methods	1 996	3 097	+55.16 %
	Total LoC	44 089	44 918	+4,24%
methods with Cognitive Complexity > 20	**Number of methods**	179	121	-32,4%
	LoC	25 103	16 167	-35,60%
methods with Cognitive Complexity > 100	**Number of methods**	26	18	-30,77%
	LoC	8 685	5 564	-35,95%

Of course, not every codebase can be cleaned up. For active, well-maintained projects there is not much room for improvement. Also, some domains are inherently complex (for example, highly optimized ML or parser algorithms) and hence cannot be easily improved.

However, most legacy enterprise software projects do benefit significantly from large scale automated code refactoring powered by artificial intelligence.

[3] https://www.sonarsource.com/docs/CognitiveComplexity.pdf

Confidentiality and IPR considerations

AMP tool uses Large Language Models at certain stages, and as such, the confidentiality and IPR concerns must be clarified.

For performance and cost-effectiveness, AMP leverages online Azure services, which are optimized for OpenAI models and offer the best price-to-value ratio. Microsoft Azure ensures robust data privacy protections, which should satisfy most use cases. However, if these guarantees are not strong enough, AMP can use locally hosted open source LLMs instead, such as Llama 3 and others. This configuration ensures that no data exits the machine hosting AMP, allowing for a fully offline setup on customer premises.

Regarding intellectual property, the primary concern is the potential for LLM-generated output to inadvertently contain copyrighted material from third parties. AMP addresses this by guaranteeing that any code being produced by it is entirely generated via an algorithmic transformation of the original source code, and therefore does not incorporate any material of any third party. AMP employs LLMs in roles akin to analysts or consultants, strictly limiting their use to generating method and variable names without introducing any copyrighted or third-party source code.

Contact Us for More Information

mantas.urbonas@twoday.com

eugenijus.medelis@twoday.com

www.twoday.com

www.twoday.lt