Dark Knowledge and Graph Grammars

How is a derivation (sequence of rewrites according to a graph grammar) an example of dark knowledge? How would you capture it?

Answer

Part 1: Every step of a rewrite sequence was based on some decision an engineer made. The knowledge and reasoning underlying these decisions are in the developers head or somewhere else easy to get lost and to be forgotten. This is dark knowledge. It exists, but it's not visible in the source code.

Part 2:

There are quite a few options for capturing portions of this dark knowledge, each with their own tradeoffs. Here's a non-exhaustive list.

1. Do nothing; expect the reader to reconstruct the reasoning when needed. In one perspective, we do this every time we use a variable name but don't document why the variable name was chosen.
2. Write a comment (or documentation) briefly hinting or fully explaining how the code or design was arrived it.
3. Use any of the teachings in the lecture, where applicable.
4. Make the derivation be the source code. For the example in the video of a parallel bloom filter join, one approach is design the system to explicitly construct then optimize a dataflow graph.

Extra discussion

What's the purpose of this question? Don Batory's stuff isn't practical, right? No. They laid out a cool vision and showed some demos, but never brought the technique to practicality, and Don Batory retired in 2020.

Dark Knowledge and Graph Grammars

My favorite principle for code quality,

Suppose you only refactored this example according to the Don't Repeat Yourself principle or the Single Point of Truth principle. How would that refactoring differ?

Answer

Since this question is about what you would do, we accept any answer to this question.

But we expect students faithfully applying DRY to obtain something like this:

```
  public int getStatAndPrint(Supplier<Integer> countFunction, int
   ↪ oldValue, String metricName) {
    if (lastCachedTime <= lastMidnight()) {
      int result = countFunction.get();
      lastCachedTime = Time.now();
      print(metric+ ": " + numUsers);
      return result;
    } else {
      print(metricName + ": " + oldValue);
      return oldvalue;
    }
}
```

which would then be used

```
public void displayStats() {
  numUsers = getStatAndPrint(() => countUsers(), "Total Users");
  numArticles = getStatAndPrint(() => countArticles(), "Articles
 ↪ written");
  numWords = getStatAndPrint(() => countWords(), "Words written");
}
```

There are many variations of this. Outside of general local changes (e.g.: the choice above to use two separate return statements), factors in variation include: whether the starting point is the very first code block in the post or a subsequent one and how to handle that each if-else statement has multiple outputs.

(Sidenote: DRY was introduced in *The Pragmatic Programmer* with a meaning closer to "single point of truth." But, due to the name, it is commonly interpreted to mean minimizing repeated subexpressions, i.e.: anti-unification. We assume this interpretation in this exercise.)