Dark Programming and The Quantification of Rationality and Understanding in Software

Isaac Griffith and Stephani Schielke

1 What is Dark Programming?

1. Traditional programming involves a subjective method (or process) designed to solve a problem and which satisfies some rationale.[1] This rationale is typically managed by the programmer in the form of a set of guiding sound engineering principles.[1]
2. A dark program is the product of some technique involving emergent behavior and so it does not depend on or involve any subjective method and use of rationales, it is not constrained to obey sound engineering principles. [1]
3. A dark program is either a proper program, we just haven't satisfied ourselves that it can be rationally explained yet, or it is not a program, because it cannot be rationally explained by the justification criterion. [1]
   (a) Where the justification criterion is:
      i. the structure of the program can be explained by the way it contributes to achieving some goal [1]
4. Sound engineering principles are a set of desireable properties including
   (a) Understandability
      i. Predictability
      ii. Testability
   (b) Maintainability
      i. Modifiability
      ii. Amenability to:
         A. location and correction of errors and performance deficiencies
         B. verification and validation efforts
      iii. Enabling of the division of work
   (c) Reusability - Reuse of solutions in future designes

2 Why Dark Programming

1. Problem: As software engineering has evolved, there is becoming an ever more increasing desire in the industry to use "sound engineering" principles to design software. Even though this trend is needed and should begin producing excellent software solutions, there still exist older software artifacts which are in need of reengineering or refactoring to today's best practices. In essence, due to the poor engineering of older software there is an inherent disconnect between those who have the task of attempting to reengineer the code and those who originally engineered it in the first place. Given that such a disconnect exists, this implies that the original intention of the code, its rationale, has become lost, leading to a darkening of the software. In order to successfully "lighten" such software one needs to invest a considerable amount of time reverse engineering the code, understanding its purpose and underlying design considerations (which may not be in keeping with current engineering principles) and then reengineer the software.

3 The Underlying Process

1. Darkness of a program, in the context of the above problem, is then the quantification of our ability to understand both the execution of the program, the underlying process embedded in the program, and the design consideration motivating the creation of this program.

2. If we then assume that we can immediately propose a means by which we can adjust the structure of the program, not changing its functionality, in such a way that it conforms to current sound engineering principles, then we have begun the arduous process of "lightening" the program. That is, using software engineering metrics which measure the desireable properties mentioned above, we can begin to use techniques, refactoring, to adjust the programs structure. Where, refactoring is a method specifically designed to alter only the structure and not the function, we can then ease the ability to understand the software. Given that these processes are themselve assumed to be in keeping with sound engineering principles, the underlying design choices are immediately understood and all that is truly left is understanding the operation of the program. Therefore, since the understanding of the program execution is itself not what makes the program dark, but instead it is the lack of existing knowledge of the rationality of the program, and given that this process infuse rationale into the program, we have in effect lightened the program.

3. Given that we can design this process and apply this method, we can then provide a piece of software which can automatically apply code refactoring techniques to an existing bank of code. This can allow developers, when provided with an older piece of software which they do not understand, to refactor and reshape the code into something they can quickly understand and with complementary documentation, which also assists in the ability to understand a project. As more and more software is being built to serve various customers we need such tools to ensure not only that code remains within the scope of best practices as defined by software engineering, but we also need such a tool to maximize the developer's efficiency and productive time.

The basic process can be outlined as follows:

1. Initially read in the source code and parse all files into a single graph structure.
2. This graph structure then represents (simultaneously) both the underlying UML of the code as well as the code structure.
3. We then randomly generate a set of refactorings to be applied, where each set is a parent allele in the Evolutionary Algorithm. Using this information, we randomly select from these alleles two parents and cross them, and produce the next generation.
4. The code graph is then cloned and the refactorings are applied.
5. From the newly modified graph we measure the metrics we are interested in and based on whether each metric is a min or max function we determine the value of that specific set and the ordering of refactorings is compared to the overall concepts of understandability, re-usability, and maintainability.
6. This process is continued until the evolutionary algorithm converges and we have an optimal set of refactorings (or the process has stop after N iterations).
7. At this point we can take the code graph and using the Eclipse Modeling Framework, generate both the code, UML, and potentially other types of documentation.

As for the current project there are a things that are needed in order to ensure that this process can happen.

1. A Complete set of metrics which not only measure across an entire code base but which specifically target the underlying issues.
2. A means by which we can divide dark programming up into its core notions and then determine the metrics which can be used to quantify those notions.
3. Originally the program worked with a very limited subset of refactoring techniques. We need to expand that set, which means that we need to analyze the set of refactorings that exist today and determine which apply and how to use them (via graph theory).
4. Although the Eclipse Modeling Framework (EMF) provides a complete set of tools that allows us to pragmatically generate both code and models, we need to fully investigate and design the connection between the current ERRS framework and the EMF.
5. Determine projects we can test this method on, and also define a means by which we can compare this methodology to, in order to provide unbiased results.
   (a) There is another method by which we can attempt this same notion of automatic refactoring
   (b) This second method would involve the use of what is known as "Code Smells." Code Smells are themselves qualitative heuristics by which one can determine locations within a code base which need to be refactored.

   (c) If we are to also apply this approach using a Strategy Pattern or Template Pattern and only changing out the Evolution Algorithm for one which uses the Code Smells heuristics, then we would also need to do the following:
      i. Determine a way in which we can quantify the "Code Smells"
      ii. Determine a means by which we can connect the metrics required for code smells to the metrics required for the overall project
      iii. Implement the code
   (d) Although there is no means by which we can provide a control group for this experiment (in the literature there are other algorithms which represent a means of partially automated refactoring, but none perform completely automated refactoring).

6. Finally we also need a means by which we can compare the input X output relation for the original code to the input X output relation of the generated code.
   (a) Perhaps some notion of Integration/Functional Testing will be needed
   (b) This idea need only be explored in deeper depth

4 Metrics

1. Program Darkness - Can be broken into the following measures
   (a) Understandability
   (b) Resusability
   (c) Maintainability
   (d) Modularity
   (e) Darkness is then the combination of the measures (a) - (d)
2. Code Smells
   (a) Qualitative heuristics to measure the location of poor spots in the design of the software
   (b) We will attempt to quantify this in order to provide a secondary approach to compare against the evolutionary algorithms

5 Significance

- Math
  - Quantification of both rationality and understanding
  - Graph Theory work: There exists a set of refactorings (which do not require human interaction) for which we can define graph manipulations to adjust a graph representing the code
  - Connecting metrics to the Dark Programming Concept
  - Potentially quantifying the underlying notion of code smells allowing this software to have multiple avenues of research, but also to perhaps provide better insight into the concept of Code Smells for both future research and other engineers as well.
- Software Engineering
  - Providing a tool to convert older code to today's standards and best practices as well as regenerating portions of project documentation
  - This then allows the code to be read more easily, to be maintained more easily, and to be reused more easily. All of this is essential to current Software Engineering principles, but what is lacking in older software and in software that is still maintained but maintained by programmers who do not necessarily understand either the code itself or good engineering principles.
- Philosophy

- This project presents work in a new paradigm of computing - A combination of the Technocratic and Scientific for CS
- Dark Programming: Rationality and the logic to understand programs (which is a major question of PCS)
- PCS: What do Computer Scientists Do: This research attempts to provide an answer (at least partially) to the question of what is a rational program and how can we re-infuse the rationality lost over time. That is how can we make programs more understandable after the absence of the original developers combined with the situation of the code not being maintained for some time.

References

[1] Janlert, Lars-Erik. "Dark Programming and the Case for the Rationality of Programs." *Journal of Applied Logic* 6 (2008): 545-52. Print.