Characterization of Logging Usage: An Application of Discovering Infrequent Patterns via Anti-unification

- Determining the detailed structural similarities and differences between entities of the source code of a software system, or between the source code of different software systems, is a potentially complex problem
- Being able to do so has various actual or potential applications:
  - code clone detection [Bulychev et al. , 2008]
  - semi-automating source code reuse [Cottrell et al. ,2008]
  - recommending the replacements for an API between various versions of a software library [Cossette et al., 2014]
  - collating API usage patterns
  - automating the merge operation of various branches in a version control system
- As a specific application, our focus is on the study of where logging is used in the source code
- [NZ: I want to make it clear that I am only working on characterizing the location of logging calls since I think the study of logging contains the study of two subjects: what logging calls are about (the context or log messages) and where they are used (the location)] **[RW: OK.]**
- Logging is a conventional programming practice to record an application's state and/or actions during the program's execution
- Logging is a pervasive practice during software development [Yuan et al., 2012]
- Researches have often considered logging as a trivial task (AOP/AOSD literature)
- However, several complex frameworks are available to help developers to log (log4J, SLF4J), which suggests that it is an important and not a straightforward task to perform in practice
- Moreover, developers spend a significant amount of effort to modify logging calls as after-thoughts since they are not satisfied with the log quality in their first attempt [Yuan et al., 2012], indicating it is not an easy task to perform high quality logging
- The importance of logging has been identified by its various applications in:
  - problem diagnosis
  - system behavioural understanding
  - quick debugging
  - performance diagnosis
  - easy software maintenance

  - troubleshooting

- Developers have to make decisions about where and what to log
- Logging should be done in an appropriate manner to be effective
  - Excessive logging:
    * can generate a lot of redundant information, masking significant ones for system log analysis
    * requires extra time and effort to write, debug, and maintain the logging code
    * can cause system resource overhead
  - Bad usage of logging can affect the application's performance
  - Insufficient logging may result in losing some necessary run-time information for software analysis
- So far, little study has been conducted on characterizing the usage of logging in real-world systems
- PROBLEM: In this research, we would like to understand where developers log in practice, in a detailed way
- The location of log statements has a great impact on the quality of logging since it helps developers to trace the code execution path to identify the root causes of errors in log system analysis
- SOLUTION: Develop an automated approach to detect the detailed structural similarities and differences in the usage of logging within a system and between systems
- [NZ: I am a little confused about how to find commonalities and differences of logging usage between systems? do you mean that I should compare the results taken from clustering of all Java classes in one system to another one?] **[RW: The process of locating commonalities and differences can be applied within a single version of one system, across multiple versions of one system, across single versions of multiple systems, or across multiple versions of multiple systems. It would be useful to understand the differences and similarities between different systems. Is there some reason that this would be harder to achieve?]**
- **Section 1.1 Broad thesis overview**
- We aim to provide a concise description of where logging calls are used in the source code through creating generalizations that represents the detailed structural similarities and differences between logged Java classes
- Our approach:

  - applies a hierarchical clustering algorithm to classify logged Java classes into groups using a measure of similarity
  - uses an anti-unification algorithm to construct a structural generalization representing the similarities and differences of all logged Java classes in each group

- Our anti-unification approach:
  - uses the Jigsaw framework to determine all potential correspondences between a pair of logged Java classes
  - applies some constraints to avoid anti-unifying logged Java classes with non-logged Java classes
  - determines correspondences between structures containing logging calls by greedily applying a similarity measure to find the most similar sub-structures
  - develops a measure of structural similarity between logged Java classes

- Our approach has been implemented as an Eclipse plug-in, which is evaluated by conducting an empirical study on 10 sample logged Java classes

- Our tool has been applied on the source code of three open-source software systems that make use of logging

- Our tool extracts all logged Java classes from these systems to construct the structural generalizations

- Our evaluation shows ...

- **Section 1.2 Overview of related work**

- Yuan et al. [2012] provides a quantitative characteristic study of log messages on four open-source software system, however, it does not study the location of logging calls in the source code

- So far, anti-unification has been used for various applications:
  - to construct a generalized correspondence view of two source code fragments [Cottrell et al., 2007]
  - to help developers to perform small-scale reuse tasks semi-automatically[Cottrell et al., 2008]
  - Software clone detection [Bulychev and Minea, 2008]

- This study makes the first attempt to characterize where logging calls occur in the source code through finding the detailed structural similarities and differences using anti-unification

- **Section 1.3 Thesis statement**

- The thesis of this work is to determine the detailed structural similarities and differences between entities of the source code that make use of logging to provide a concise description of where logging do occur in real systems
- **Section 1.4 Thesis Organization**
- Chapter 2: motivates the problem of understanding where to use logging calls in the source code through an example
- Chapter 3: provides background information on:
  - abstract syntax trees (ASTs), which are the basic structure we will use for describing software source code
  - how ASTs are realized in the Eclipse integrated development environment, the industrial tool we will build atop
  - anti-unification and its limitations to solve our problem context
  - higher-order anti-unification modulo theories (HOAUMT) that can be applied on an extended form of the AST structure to address our problem
  - the Jigsaw framework, an existing tool for a subset of HOAUMT, which we extend to address our problem
- Chapter 4: describes our proposed approach and its implementation as an Eclipse plug-in
- Chapter 5: presents an empirical study conducted to evaluate our approach and its application to characterize logging usage
- Chapter 6: describes related work to our research problem and how it does not adequately address the problem
- Chapter 7: discusses the results and findings of my work, threats to its validity, and the remaining issues.
- Chapter 8: concludes the dissertation and presents the contributions of this study and future work
- **Chapter 2. Motivational Scenario**
- Logging is a systematic way of recording the software runtime information
- A typical logging call is composed of a log function and its parameters including a text message and verbosity level
  - A log text message consists of static text to describe the logged event and some optional variables related to the event
  - The verbosity level is intended to classify the severity of the logged event (Fatal, error, warn, info, and debug)

- Consider a developer is given the task of logging the source code of a Java class
- She decides to use log4j framework for logging
- She has to make several decisions about
  - what events need to be logged?
  - where to use logging calls?
  - how to decide on the text message and verbosity level of each logging call?
- It is recommended to simply log at the start and end of every method
- However, for example, it is useless to log at the start and end of the method ..., producing redundant information
- She needs more information to perform logging appropriately
- Having a characterization of how usually developers use logging calls in the source code in similar situations would assist her in making decisions
- For example, knowing that developers use logging calls inside of if statements to log a potential error if a variable contains an incorrect value, she adds an if statement to log an error if the value of variable ... is null
- For example, knowing that developers use logging calls inside catch blocks to record an exception, she creates a try/catch blog to capture the potential error thrown by function .. and then use a logging call in its corresponding catch block
- Using a concise characterization she would be able to make informed decisions about where to use logging calls more easily and quickly
- With taking appropriate decisions about where to use logging calls, she can spend more time and energy to write the context of log functions and even other development and maintenance tasks
- **Chapter 3. Background**
- **Section 3.1. Abstract Syntax Tree**
- The Eclipse Java Development Tools (JDT) framework provides APIs to access and manipulate Java source code via Abstract Syntax Tree (AST)
- AST maps Java source code in a tree structure form
- Every Java source code can be represented as tree of AST nodes
- Every AST node represents an element of the Java Programming Language

- Using the AST developers
  - can modify and analyze the Java program in a more convenient way than text-bases source code
  - do not need to parse the source code
  - can determine the bindings between name and type references
  - can obtain specific information of each Java element
- Structural properties of an AST node hold specific information for the Java element it represents
- Structural properties are stored in a map data structure that associates each property to its value
- The structural properties have three different types:
  - *Simple Property:* where the value is an AST constant (simple value)
  - *Child Property:* where the value is a single AST node
  - *Child list property:* where the value is a list of AST nodes
- AST nodes are subtrees of the tree structure and simple values are the leaves
- **Section 3.2. Anti-unification**
- A *term* is a set of
  - *Function symbols*:
    - that can take an unlimited number of arguments
    - are represented by identifiers starting with a lowercase letter (e.g., *f(a,b)*)
  - *Constants*: function symbols with no arguments (e.g., *a*, *b*)
  - *Variables*: are represented by identifiers starting with an uppercase letter (e.g., *X*, *Y*)
- the following are terms:
  - *Y*
  - *a*
  - *f(X, c)*
  - *f(g(X, b),Y, g(a, Z))*
- *Ground term*: a term that does not contain any variables (e.g., *f(a,b)*)
- A *Substitution* is defined on a term to map its variables to ground terms
- *Applying Substitutions*: replacing all occurrences of a variable with a proper term

  - For example, an application of a substitution $\ominus = \ X \longrightarrow a\ $ on a term *f(X,b)* is defined by replacing all occurrences of variable X by term a and thus *f(X,b)* $\xrightarrow{\ominus}$ *f(a,b)*

- *Instance & Anti-Instance* : a is an instance of X and X is an anti-instance of a, if there is a substitution $\ominus$ such that the application of $\ominus$ on X results in a ($X \xrightarrow{\ominus} a$)
- *Unifier*: A common instance of two given terms
- *Most General Unifier (MGU)*: U is MGU of two structures such that for all unifiers $U\prime$ there exist a substitution $\ominus$ such that $U \xrightarrow{\ominus} U\prime$
- *Generalization*: X is a generalization for a and b, where X is an anti-instance for a and b under substitutions $\ominus_1$ and $\ominus_2$, respectively. ($X \xrightarrow{\ominus_1} a$ and $Y \xrightarrow{\ominus_2} b$)
- *Anti-unifier*: A common generalization of two given terms
- *Most Specific Anti-unifier (MSA):* A is MSA of two structures where there exist no anti-unifier $A\prime$ such that $A \xrightarrow{\ominus} A\prime$
- MSA should preserve as much of structure of both original structures as possible
- *Anti-unification*: The process of finding the MSA of two given terms
- *First-order anti-unification*: restricts substitutions to replace only first-order variables by terms
- PROBLEM: First-order anti-unification fails to capture complex structural commonalities
  - When the terms differ in function symbols, First-order anti-unification fails to preserve common details of structures, e.g., anti-unifier of structures *f(a,b)* and *h(a,b)* is X
- SOLUTION: Higher-order anti-unification allow the creation of MSA by extending the set of possible substitutions such that variables can be replaced by not only constants but also functional symbols
  - e.g., anti-unifier of structures *f(a,b)* and *h(a,b)* is *X(a,b)*
- An instance of an AST structure can be mapped to our recursive definition of a term, where AST nodes and simple values may be viewed as function-symbols and constants, respectively
- A set of equivalence equations is defined to incorporate semantic knowledge of structural equivalences supported by the Java language specification

- These equational theories are applied on higher-order extended structures to allow the anti-unification of AST structures that are not identical but are semantically equivalent, e.g., for-loops and while-loops
- The NIL structure is defined to create a structure out of nothing that would allow the anti-unification of two structures when a substructure exist in one but not in the other one
- Defining complex substitutions results in losing the uniqueness of MSA
- The complexity of finding an MSA is undecidable in general since an infinite number of substitutions can be applied on every variable in a structure
- Our goal is to find an MSA that is an approximation of the best fit to our application
- **Section 3.3. Jigsaw**
- The Jigsaw tool is developed by Cottrell et al. [2008] to determine the structural correspondences between two Java source code fragments through anti-unification such that one fragment can be integrated to the other one for small scale code reuse
- The Jigsaw framework could help us to construct an anti-unifier form logged Java classes as it:
  - applies higher-order anti-unification modulo theories to address the limitations of first-order syntactical anti-unification
  - creates an augmented form of AST, called CAST (Correspondence AST), in which each node holds a list of candidate correspondence connections between the two structures, each representing an anti-unifier
  - develops a measure of similarity to indicate how similar the nodes involved in one correspondence connection are
- The Jigsaw similarity function:
  - returns a value between 0 and 1 where indicate zero and total matching, respectively
  - uses several semantical heuristics to improve the accuracy of similarity measurement
- The Jigsaw tool does not sufficiently solve our problem since:
  - it determines all potential correspondences between AST nodes, each representing an anti-unifier; however, we need to determine one single anti-unifier that is approximately the best fit to our problem
  - it does not construct an anti-unifier of two given AST structures with a focus on logging calls, which is required to our application

- **Chapter 4. Anti-unification of Logged Java Classes**
- GOAL: Construct structural generalizations representing the detailed similarities and differences of Java classes that use logging calls in the source code
- APPROACH: Develop an algorithm that:
  - classifies the logged Java classes into groups using a measure of similarity such that entities in each group has maximum similarity with each other and minimum similarity to other ones
  - abstracts structural correspondences of logged Java classes of each group into a structural generalization representing where logging calls do occur
- PROBLEM 1: How to construct a generalization from two given logged Java classes with a special attention to logging calls?
- SOLUTION: Develop an algorithm that:
  - maps source code of the two logged Java classes to AST structures via the Eclipse JDT framework
  - applies the higher-order anti-unification modulo theories to construct the generalized structure
    - SUBPROBLEM: the AST structure does not support the use of variables, which is required to construct an anti-unifier
    - SOLUTION: create an extension of AST structure, called AUAST, to allow the insertion of variables in place of any node in the tree structure, including both subtrees and leaves
  - Determines potential candidate structural correspondences between AUAST nodes using the Jigsaw framework
  - Applies some constraints to prevent the anti-unification of logging calls with anything else, which are:
    1. A logging call should either be anti-unified with another logging call or should be anti-unified with "nothing".
    2. A structure containing a logging call should be anti-unified with a corresponding structure containing another logging call or should be anti-unified with "nothing".
  - Determines the best structural correspondences between AUAST nodes
    - SUBPROBLEM: Despite having multiple potential anti-unifiers, we need to construct one single anti-unifier from the two AUASTs that is an approximation of the best fit to our problem
    - SOLUTION: Develop a greedy selection algorithm to approximate the best anti-unifier by determining the best correspondence (correspondence with the highest similarity) for each node
  - Develops a measure of similarity between the two AUASTs that

    - refers to the number of identical simple values over the total number of simple values of their anti-unifier
    - calculates the similarity between simple values via the Jigsaw similarity function

- PROBLEM 2: How to classify a set of AUASTs of logged Java classes?
- SOLUTION: Develop a modified version of a hierarchical agglomerative clustering algorithm as described below:
    1. Start with singleton clusters, where each cluster contains one AUAST
    2. Compute the similarity between clusters in a pairwise manner
    3. Find the closest clusters (a pair of clusters with maximum similarity)
    4. Merge the closest cluster pair and replace them with a new cluster containing anti-unifier of AUASTs of the two clusters
    5. Compute the similarity between the new cluster and all remaining clusters

    - Repeat Steps 3,4, and 5 until the similarity between closest clusters becomes below a predetermined threshold value
    - The similarity between a pair of clusters is defined as the similarity between their AUASTs
    - Determine the similarity threshold value through informal experimentation

- **Chapter 5. Evaluation**
- Two empirical studies were conducted
- **Section 5.1 Experiment 1**
- [NZ: I am so confused about how to represent the results and which parameters should be evaluated? I wrote what seems reasonable to me, but I would be grateful if you could guide me on how to write it in a way that makes sense] **[RW: OK, but let's worry about that later. I would really like to get to the stage that everything is clear before that.]**
- First experiment is conducted to evaluate the accuracy of our approach and tool
- It addresses the following research questions:
    - RQ1: can our tool determine the structural similarities and differences between logged Java classes correctly?
    - RQ2: can our tool compute the similarity between logged Java classes correctly?

- To do so, 10 logged Java classes were selected randomly from jEdit v4.2 pre 15 (2004), as our test set
- We apply our tool on the test set to create generalizations and to compute the similarity value between logged Java classes in a pairwise manner
- To address the first research question we compute the following measurements for each test case:
  - the number of correspondences that our tool detects correctly
  - the total number of correspondences
- To determine the correct correspondences we performed a manual investigation
- To address the second research question we compute:
  - the number of similarity values between logged Java classes that are computed correctly by our tool
  - the total number of comparisons
- The correct similarity value for each comparison is calculated manually by
  - counting common simple values based on the selection of corresponding AUAST nodes
  - counting total number of simple values of AUAST nodes
- The results taken form our tool are compared with the results taken by manual investigation using the JUnit testing framework
- **Section 5.2 Experiment 2**
- Second experiment is conducted to address the following research questions:
  - RQ3: what structural similarities and differences do logged Java classes have?
  - RQ4: Is it possible to find common patterns in where logging calls do occur?
- To do so, we applied our tool on the source code of three open-source full systems that make use of logging to determine the patterns on a per-system class-granularity basis analysis
- These systems are different from the system that the test set is selected from
- **Section 5.2.1 Results**
- I will describe the results taken form the second experiment
- **Section 5.2.2 Lessons learned**
- I will describe our findings

- **Chapter 6. Discussion**
- **Chapter 6.1 Threats to validity**
- Our goal is to recognize the limitations and pitfalls of our approach and its developed tool support
- The first potential thread to validity of our characterization study is the degree to which our sample set of software systems is a good representation of all real-world logging practices. To address this issue we selected software systems that:
  - are different in terms of application
  - are among the most popular applications in their own product category
  - has long history in software development
- Secondly, our manual investigation to find the correct correspondences might be biased due to human errors. To limit the bias
  - other people can be involved to double check the accuracy of manual work in the future work
- However, these results are still promising
- **Chapter 6.2 Our tool output**
- We investigated the cases where our tool fails and we found that the failures are due to:
  - the assumption taken in developing the algorithms
  - the fundamental limitations and complexities in determining the detailed structural similarities and differences
- The are some issues that our tool is not able to handle perfectly:
  - Our tool does not guarantee to maintain the correct ordering of statements inside a method body
  - Our tool does not guarantee the correctness of determining the best correspondences due to
    - the various conflicts that happen
    - limited typing information to determine correspondences by Jigsaw
  - Structural generalizations constructed by our tool are not in the form of executable code
- **Chapter 6.3 Theoretical foundation**
- Anti-unification and its extensions has several theoretical and practical applications:

  - analogy making [Schmidt, 2010]
  - determining lemma generation in equational inductive proofs [Burghardt, 2005]
  - detecting the construction laws for a sequence of structures [Burghardt, 2005]

- Using higher-order anti-unification modulo theories in our application, which is undecidable in general, leads us to take approximations suitable to our context

- The set of equational theories should be developed particularly for the structure used in each problem context

- **Chapter 7. Related Work**

- **Section 7.1 Logging**

- Logging is a systematic way of recording the software runtime information [Yuan et al., 2012]

- Several studies have been conducted to investigate the application of logging for various software development and maintenance tasks:
  - Jiang et al. [2009a] study the effectiveness of logging in problem diagnosis
  - Fu et al. [2013] present an approach for understanding system behaviour through contextual log analysis
  - Yaghmour et al. [2000] provide an efficient kernel-level event logging infrastructure to record and analyze system behaviour
  - Nagaraj et al., [2012] develop an automated tool to assist developer in diagnosis and correction of performance issues in distributed systems through analyzing system behaviours extracted from log data
  - Bishop [1989] proposes a formal model of systems security monitoring using logging and auditing
  - Elnozahy et al. [2002] present a survey of log-based rollback-recovery protocols
  - Jiang et al. [2009b] present an approach to automatically detect problems of load tests by mining the application execution logs

- Yuan et al. [2012] provides a quantitative characteristic study of log messages on four open-source software systems. Their study shows:
  - Logging is a pervasive practice during software development
  - Developers spend a significant amount of effort to modify logging calls as after-thoughts, since they are not mostly satisfied with the log quality in their first attempt

  - Several findings on where developers spend most of their time in modifying the log messages

- However, the focus of our study is on characterizing where developers log (not the log messages)

- **Section 7.2 Anti-unification**

- Anti-unification is the problem of finding the most specific generalization of two terms

- First-order syntactical anti-unification was introduced by Plotkin [1970] and Reynolds [1970] independently

- Burghardt and Heinz [1996] extend the notion of anti-unification to E-anti-unification

- Burghardt and Heinz [1996] introduced anti-unification modulo equational theories to incorporate background knowledge to syntactical anti-unification, which is required for some applications

- Higher-order anti-unification modulo theories is formally undecidable [Burghardt, 2005]

- Anti-unification has various applications in program analysis:

  - Bulychev and Minea [2008] suggest an anti-unification algorithm to detect software clones, however, the algorithm does not suffice to our problem context since it:
    - does not construct a generalization to represent the deatiled structural similarities and differences
    - does not require to take constraints needed to our problem
  - Cottrell et al. [2007] propose Breakaway to automatically determine structural correspondences between a pair of abstract syntax trees (ASTs) to create a generalized correspondence view, however its algorithm does not suffice to our problem context since it:
    - does not meet the requirements needed for our application to determine the correspondences
  - Cottrell et al. [2008] develop Jigsaw to help developers integrate small-scale reused source code into their own code via structural correspondence through the application of higher-order antiunification modulo theories, but the algorithm does not suffice to our problem context since it:
    - does not require to construct one single generalization
    - does not require to take constraints
  - Sadi [2011] proposed an anti-unification algorithm to characterize the location of logging usages in the source code, however,

    - he has not applied anti-unification appropriately!!! [**RW: You would need to explain what that means**]

- **Section 7.3 Correspondence**

- Several approaches have been used to find correspondences between code fragments
  - Baxter et al. [1998] develops an algorithm to detect code clones in source code that uses hash functions to partition subtrees of ASTs of a program source code and then find common subtrees in the same partition through a tree comparison algorithm
  - Apiwattanapong et al. [2004] present a top-down approach to detect differences and correspondences between two versions of a Java program, through comparison of the control flow graphs created from the source code
  - Strathcona [Holmes et al., 2006] recommends relevant code snippet examples from a source code repository for the sake of helping developers to find examples of how to use an API by heuristically matching the structure of the code under development with the source code in the repository
  - Coogle [Sager et al., 2006] is developed to detect similar Java classes through converting ASTs to a normalized format and then comparing them through tree similarity algorithms
  - However, these approaches do not determine the detailed structural similarities and differences needed in our context
  - Umami [Bradley et al., 2014] presents a new approach, called Matching via Structural generalization (MSG), to recommend replacements for API migration
    - He used the Jigsaw tool to find structural correspondences, but the algorithm does not suffice to our problem context since it:
      - does not construct a generalization to represent structural similarities and differences
      - does not require to take constraints needed to our problem

- **Section 7.4 API usages patterns**

- Various data mining approaches has been used to extract API usages patterns:
  - Unordered pattern mining
    - Association rule mining: e.g., CodeWeb [Michail, 2000] uses data mining association rules to identify reuse patterns between a source code under development and a specific library
    - Itemset mining: e.g., PR-Miner [Li and Zhou, 2005] uses frequent itemset mining to extract implicit programming rules from source code and detect violations

  - Sequential pattern mining: e.g., MAPO [Xie and Pei, 2006] combines frequent subsequence mining with clustering to extract API usage patterns from the source code

- However, none of these approaches suffice to our context since they:
  - do not to determine the detailed structural similarities and differences needed in our context

- **Section 7.5 Clustering**

- Clustering is an unsupervised machine mining technique that aims to organize a collection of data into clusters, such that intra-cluster similarity is maximized and the inter-cluster similarity is minimized [Karypis, 1999] [Grira et al., 2004]

- Clustering methods:
  - K-means [Hartigan et al., 1979]
    - Not a good fit to our problem since it requires to define a fixed number of clusters
  - Hierarchical clustering [Rasmussen, 1992]
    - The cut-off threshold value indicates the number of clusters we will come up with

- Agglomerative hierarchical clustering is one of the main stream clustering methods [Day, 1984] and has applications in:
  - document retrieval [Voorhees, 1986]
  - information retrieval from a search engine query log [Beeferman et al., 2000]

- There are various techniques to measure the distance between clusters [Rasmussen, 1992]:
  - Single linkage
  - Complete linkage
  - Average linkage
  - Centroids
  - Ward's method

- However, in our application, the distance between clusters is defined as the distance between their AUAST nodes

- **Chapter 8. Conclusion**

- Determining the detailed structural similarities and differences between source code fragments is a complex task

- It can be applied to solve several source code analysis problems, for example, characterizing logging practices
- logging is a pervasive practice and has various applications in software development and maintenance
- However, it is a challenging task for developers to understand how to use logging calls in the source code
- We have presented an approach to characterize where logging calls happen in the source code by means of structural generalization
- We have developed a prototype tool that:
  - detects potential structural correspondences using anti-unification
  - uses several constraint to remove the correspondences that are not suited to our application
  - determines the best correspondences with the highest similarity
  - constructs structural generalization using anti-unification
  - classifies the entities using a measure of similarity
- An experiment is conducted to evaluate our approach and tool
- Our experiment found that ...
- An experiment is conducted to characterize logging usage in three software systems
- In summary, our study makes the following contributions:
  -
  -
- **Section 8.1 Future Work**
- Data flow analysis techniques could be applied to resolve the problem of inaccurate statement ordering
- Further analysis can be done to detect and resolve all the conflicts happen in deciding the best correspondences
- However, the complexity of further analysis should be limited to keep the solution practical
- To further validate our findings from the source code analysis:
  - a survey can be conducted to ask developers on the factors they consider when they want to decide on where to log
- Characterizing logging usage could be a huge step towards

- improving logging practices by providing some guidelines that might help developers in making decisions about where to log.
- developing recommendation support tools:
  - to save developers'time and effort
  - to improve the quality of logging practices