# Understanding software artifact provenance

Michael W. Godfrey*

*David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, CANADA*

**Abstract**

In a well designed software system, units of related functionality are organized into modules and classes, which are in turn arranged into inheritance trees, package hierarchies, components, libraries, frameworks, and services. The trade-offs between simplicity versus flexibility and power are carefully considered, and interfaces are designed that expose the key functional properties of a component while hiding much of the complexity of the implementation details. However, over time the design integrity of a well-engineered system tends to decay as new features are added, as new quality attributes are emphasized, and as old architectural knowledge is lost when experienced development personnel shift to new jobs. Consequently, as developers and as users we often find ourselves looking at a piece of functionality or other design artifact and wondering, "Why is this here?" That is, we would like to examine the *provenance* of an artifact to understand its history and why it is where it is within the current design of the system.

In this brief paper, we sketch some of the dimensions of the broad problem of extracting and reasoning about the provenance of software development artifacts. As a motivating example, we also describe some recent related work that uses hashing to quickly and accurately identify version information of embedded Java libraries.

*Keywords:* Software artifact provenance, software analytics, code cloning

> Paul, it is easy to be impressed by your body of work at a distance. There are so many important and well-cited papers over many years, and the tools and techniques you and your group have contributed have had concrete and demonstrable impact on the research community. But when I spent my sabbatical at CWI, I learned an awful lot from you about running a research group, and getting the best out of talented people. You make it seem so easy and effortless, although I know it is neither. Your management style is low key and encouraging, yet the yield is so high. Perhaps most importantly, I have seen how the young researchers you supervise — many of whom go on to careers in industry — come away with justifiable pride in their efforts, an understanding of exactly how their work fits into the grand scheme of things, and a strong belief in the value of research itself. Surely this is the right way to build an ongoing, vibrant, and collegial community of scientists, engineers, and practitioners.

## 1. Introduction

In art and antiques, the term "provenance" is used to denote a set of evidence as to the origin and history of an artifact. Sometimes, the evidence is direct and clear, consisting of chains of accepted documentation. For example, the origin and history of the Italian renaissance painting *Diana and Actaeon* is well documented historically [16]: it was painted by Titian in the late 1550s for the Spanish ambassador; it subsequently passed through a series of French, Belgian, and British owners until it was purchased in 2009 by a consortium of UK public trusts. Sometimes, however, the evidence is much less certain, and guesses must be made according to imperfect knowledge. For example, the painting *La Bella Principessa* is believed to have been executed by Leonardo da Vinci [19]; this belief is based on a combination of incomplete historical documentation plus "best guess" analysis of the materials, presumed tools and processes, and artistic styles.

*Corresponding author

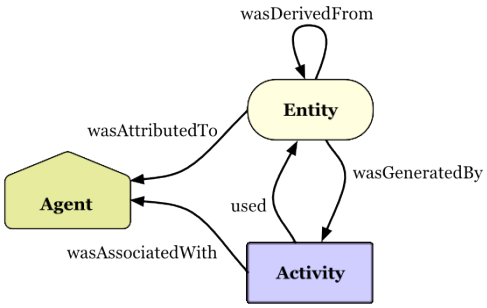


Figure 1: The W3C model for web document provenance (diagram taken from [3]); its *entity-agent-activity* metaphor also works well for other problem domains, including software development.

More recently, the term provenance has been used in new ways. In software security, for example, the provenance of an executable component determines how trusted it is and what its permissions are [8]. Data provenance in the sciences is important for reproducibility of experiments; scientists must be able to tell which parts of which data set were generated at what time, by whom, using what tools, and which settings [12]. And the W3C has recently published a specification for web-based documents that authors can use to mark up their web pages with meta-data about the provenance of web resources (Fig.1).

Increasingly, the term provenance is being used within the context of software development. Developers, managers, QA team members, and other stakeholders often wish to understand how and why a feature, component, chunk of code, test suite, or other development artifact came to be where it is. For example, when a manager finds a section of code that is involved in an unusually large number of problems, he may wish to find out how old the code in question is, which developers have been working on it recently, which change sets it has been involved in, and which features, quality attributes, and bug fixes it implements. When a IP lawyer is concerned about a claim of open source licensing violations within a closed source system, she may wish to be able to track the design history of the code in question [4]. When a software architect is trying to understand the current design of a module within a software system, he may wish to be able to compare the change history against discussions in the developer mailing list archives [5]. Additionally, the burgeoning research areas of Mining Software Repositories [10] and Software Analytics [9] concern developing infrastructure for asking these kinds of questions by analyzing disparate kinds of artifacts, considering how best to model and link the knowledge together, and presenting a practical interface to allow stakeholders to explore the space.

In this brief paper, we attempt to sketch the emerging domain of software development provenance. We discuss some of the typical problems, and what they have in common. We consider a general approach to attacking these problems using the metaphor of Bertillonage, a $19^{th}$ century approach to forensic analysis. And finally, we describe an example of such an approach that was used to solve a software artifact provenance problem.

## 2. The problem of software development provenance

The general problem of software development provenance can be stated simply. That is, given a software development entity — such as a function, feature, test suite, or mailing list discussion topic — we want to ask questions such as:

- What is the history of this entity? How did it come to be how and where it is currently?
- What other entities is it related to, and how?
- What is the evidence? How reliable is it?

Examples of provenance-related problems with software development include:

**Clone detection and IP violation** — Duplication of source code within or across projects is a well-known phenomenon, and may be done for a variety of reasons [6]. When confronted with the presence of code cloning within a software system, a manager will want to understand why the decision was made to do so, and what the perceived short- and long-term trade-offs are. Cloning across different projects is more difficult to manage; of special concern is the possibility of code from an open source project being inserted into a closed source system [4, 2], which may open up the company to legal action, such as was alleged in the recent lawsuit involving Oracle and Google [18].

**Recommender systems** — Recent work on adapting the metaphor of recommender systems for use in aiding software development tasks requires that a tool be able to recognize "similar" developer tasks, and suggest artifacts that may be of use [11]. For example, unlike most IDEs, the Mylyn plugin for Eclipse models development tasks as a multi-dimensional first-order artifacts; this allows the underlying system to recognize the implicit context of what files, APIs, bug reports, mailing list discussions that may be of interest, based on previous experience of "similar" development tasks [7].

**Bug report duplication** — A common headache faced by open source projects is the task of bug triage: taking bug reports from the field, evaluating their legitimacy and importance, and assigning them to an appropriate developer for fixing. Widely used open source systems generate a lot of bug reports every day, yet inevitably many of them will be symptoms of a common underlying problem and need to be marked as such. Unfortunately, in practice the triage process is mostly manual, and requires triagers with a good understanding of the project history. Consequently, researchers have sought ways to automatically suggest which new bug reports might be related to previously reported bugs using a variety of techniques, such as performing textual analysis of the free form text, looking for commonly recurring error messages, noting which API elements may be involved, and using machine learning techniques [13, 14].

What all of these problems have in common is the requirement to consider a software development entity — that may not be modelled as a first class entity by existing tools — and either be able to trace its historical evolution or look for "similar" entities for comparison, according to an appropriate definition of similarity. Common issues that need to be considered include:

**Scoping and defining the entity** — For some entities, their definition and scope is clear enough; for example, classes, functions, and test suites are usually managed as near first-class objects by the IDE and version control system. For other entities, their composition may be implicitly defined by a set of fields, which may have implicit structure of their own; for example, bug reports may consist of a set of discrete values, such as date and priority, together with a set of free-form textual fields that describe the problem. For still others, no objective boundaries may yet be possible; for example, maintenance task histories or mailing-list discussion topics are areas of active investigation and so lack a clear, well-accepted ontology.

**Artifact linkage and ground truth** — Some questions about identity and relatedness are straightforward to answer authoritatively, given enough evidence. For example, evolving source code is usually managed through a version control system; consequently, it is easy to track changes to files, classes, methods and even chunks of code, once an appropriate level of granularity has been decided on. However, in some cases, there may be gaps in our knowledge; for example, mailing list discussions may concern only a subset of the known features of a system, making it risky to draw broad conclusions. Also, the analysis techniques may be approximate and non-authoritative; for example, using a topic mining algorithm to infer what developers are talking about in a mailing list discussion may be useful enough to be practical sometimes, but have poor precision and/or recall in the general case.

**Scaling up matching algorithms to large data sets** — The goal of software analytics is to allow stakeholders to ask high-level questions about software development that depend on the integration of various kinds of analyses within a unified interface. When we are asking these high-level questions of identity and similarity, we are often requiring that complex matching algorithms be run on demand on very large data sets; thus, we must consider strategies for making such queries practical.

The problem of scaling is particularly vexing, yet it is key to practical software provenance investigation: many of the search spaces are very large and naively comparing each entity against every other entity is usually an impractical approach. In the next section, we will consider a general two-phased strategy that is similar to a $19^{th}$ century approach to forensic detection: simple, cheap techniques applied widely followed by more expensive and precise techniques applied narrowly.

## 3. Software Bertillonage

Alphonse Bertillon worked as a clerk for the Paris police in the late 1800s [15]. At the time, the standard procedure for identifying criminals was for the police to laboriously wade through a large library of photographs of known criminals. Lacking a better organizing principle, the library was arranged alphabetically by name; of course, the Parisian underworld soon realized that they could usually beat the system if they gave a false name when being arrested. Unless the policeman was unusually persistent or happened to recognize the suspect from a previous encounter, the huge scale of the library of photographs meant that it was hard to identify suspects reliably.

Bertillon realized that while suspects might be able to lie about their name, they could not lie about their body measurements. His idea was to model identity on what we would now call bio-metrics: on arrest, a suspect was both photographed and his body measured in about 10 places. The photograph library was organized hierarchically by the various measurements; practically, this meant that "lookup" of a new suspect's measurements narrowed the set of photographs that had to be looked at (i.e., that matched the measurements) from hundreds to a small handful. Bertillonage was the prevailing forensic method of identifying criminals for several years; however, it was quickly succeeded by fingerprinting, which turned out to be more reliable, cheaper, and easier to implement.

We use the metaphor of Bertillonage — instead of the more precise fingerprinting or DNA analysis — for several reasons. First, more precise approaches often require a single, positive match; there is a specific candidate we are looking to match against and have a reasonably faithful model of, so we can be as detailed as we like in our queries. In software provenance analysis, on the other hand, we are often looking for similar but non-identical entities, such as cloned code segments that have diverged over time or bug reports that appear to concern the same underlying problem. Thus, in the case of matching Java library versions discussed in the next section, we may gain useful knowledge from matching to a close relative of our target if the target itself is not in our database; performing a stricter matching might mean missing useful loose matches. Second, we may not have enough good quality information in our model to make a close match in the first place; for example, if we are comparing maintenance tasks, it may not be clear which structural elements are most relevant in a given situation. Looser matching might mean better recall, but possibly at the expense of some precision.

The metaphor of Bertillonage for investigating software provenance implies a two-phased approach. First, we seek a simple metric that is relatively cheap to compute on a large data set, is applicable at the level of granularity desired, and has good discriminatory value on candidates (i.e., relatively good precision and recall). Second, we perform a more expensive and precise analysis on the result set from the first phase; for example, an expensive clone detection algorithm might be used that requires deep static analysis of the code, or we may perform a manual analysis of the entities, such as we do in the example in the next section.

## 4. Library version identification: A detailed example

Developing an industrial Java program typically means addressing the issue of how to ensure that all library requirements can be met on the deployment site. The most common approach to this is to statically include recent versions of required libraries in the deployment package that ships as the system. Typically, a Java library archive comprises a single *jar* or *zip* file that contains a set of compiled Java classes; some archives also contain the corresponding source code.

Unfortunately, there is no easy way of authoritatively determining which versions of which libraries have been included by the developers. While it is common practice to use the version identifier as part of the label in the archive manifest, this is by no means universally practised or enforceable, and in practice, we have observed that developers will sometimes strip away the version information. On the other hand, if you are running a version of such a system that uses a library version with known security flaws, it is important to be able to determine just which versions are included in your running system.

To address this problem, we decided to create a master database of well known Java libraries using the Maven2 public repository as a basis [1]; Maven2 contains many versions of most popular libraries from the Java world, stored as archives that include byte code and (sometimes) the source code. The idea was that a developer could present their Java library archive to a query engine which would be able to return the version identifiers of the closest matches found inside Maven2.

We decided to use hashing, specifically SHA1 hashes, to compare candidate archives against the database we were going to build. However, since not all archives in Maven2 also contain the corresponding source code, we could not expect to be able to perform source-to-source matching. At the same time, we could not simply match class binaries against class binaries, as the same sources might have been compiled using different Java compilers and command line options. What we *could* use to match against was the class interface, since this was present in both the source and the byte code; that is, for each binary class in Maven2 — and there were 27 million classes in Maven2 when we did this — we extracted the method, field, and (recursively) inner class signatures, and created an SHA1 hash of the resulting strings.

The atomic level of comparison was the class interface, and two classes either matched exactly — i.e., had the same SHA1 hash — or did not match at all. However, a prepackaged Java archive may contain dozens, hundreds, or even thousands of classes, and we were primarily interested in comparing archives rather than individual classes. So we defined a metric for containers called the *similarity index*, which measures the amount of overlap between two archives.

More formally, given an archive $A$ composed of $n$ classes $A = \{c_1, ..., c_n\}$, we define the signature of an archive as the set of signatures of its contained classes.

$$\vartheta(A) = \{\vartheta(c_1), ..., \vartheta(c_n)\}$$

We further define the *Similarity Index* of two archives $A$ and $B$, denoted as $sim(A, B)$, as the Jaccard coefficient of their signatures [17]:

$$sim(A, B) = \frac{|\vartheta(A) \bigcap \vartheta(B)|}{|\vartheta(A) \bigcup \vartheta(B)|}$$

This approach to matching has the advantage of being fast when comparing a target system against the database of hashes (once the database has been constructed, that is). Additionally, false negatives are impossible, since if two libraries have non-identical interfaces then they cannot be the same version. However, false positives *are* possible when a library's interface remains stable across multiple versions.

Additionally, Maven2 is by no means comprehensive; while it is a convenient "depository" for commonly used versions of Java libraries, it does not aim to contain all versions of all of the libraries. Consequently, there may be gaps in the version history of any given library; that is, we maybe searching for a library for which many versions exist in Maven2, but not the one we are looking for. Finally, since Maven2 is not systematically organized, there is a enormous amount of duplication of the contained archives; the same version of a given library may be present dozens or even hundreds of times. In the first version of our tool we chose not to weed out duplicate responses for the sake of simplicity; consequently, a query may return a large result set as a best match, comprising many instances of the same archive version.

The question was, how well would this naive matching approach work in a realistic setting? Do Java libraries evolve often enough to make it easy to distinguish between successive versions? Would the amount of archive duplication in Maven2 lead to very large result set, requiring subsequent time-consuming and tedious manual analysis? Was Maven2 comprehensive enough to return useful answers most of the time?

Building the database required downloading all of Maven2 and processing the millions of contained class files, but once the database had been built comparisons against target systems were very fast. We performed some test extractions, using 1000 randomly chosen binary jars that we knew to be in Maven2. When we compared our selected binary archives to the Maven2 hash database, we found that for every archive in our sample, our tool returned the correct version identifier within the set of "best matches". We also found that the median number of best matches was 5, meaning it appeared to be a tractable technique most of the time, and that the maximum number of best matches was 487, which occurred when very popular library versions has been added to Maven2 many times. We then used a real-world Java application from the domain of e-commerce as our target; and we found that of the 84 libraries it

included, we were able to get the correct version within the best match set for 81 of them (the remaining 3 libraries versions were found not to be present in Maven2).

In summary, the Bertillonage metaphor of applying a computationally cheap and conceptually simple matching algorithm to a large data set, then applying a more expensive technique — in this case, manual analysis of the best matches — worked well on the problem of matching library versions identifiers to a large space of possible matches taken from a near-comprehensive master repository.

## 5. Analyzing software entity provenance: Moving forward

The question of provenance within the context of software development has been around in various guises for a long time. With the recent advent of new techniques for analyzing and linking different artifact kinds from the fields of Mining Software Repositories and Software Analytics, we can begin to explore how and why a variety of kinds of software entities have evolved to be as they are. In such large and complex domains, we must resist the urge to become too detailed too quickly; instead, we must be willing to trade some precision on our initial queries for recall, and then perform more expensive analyses on the result sets. In so doing, we can then begin to address complex and subtle questions about the identity, history, and relatedness of software development entities.

## References


[1] J. A. Davies, D. M. German, M. W. Godfrey, and A. J. Hindle. Software Bertillonage: Finding the provenance of an entity. In *Proc. of the* $8^{th}$ *IEEE Working Conference on Mining Software Repositories*, MSR '11, May 2011.

[2] M. Di Penta, D. M. German, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the evolution of software licensing. In *Proc. of* $32^{nd}$ *ACM/IEEE Intl. Conference on Software Engineering*, ICSE '10, May 2010.

[3] Gil, Yolanda and Miles, Simon (eds.). PROV Model Primer — W3C Working Group Note. `http://www.w3.org/TR/2013/NOTE-prov-primer-20130430/`.

[4] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *Proc. of* $8^{th}$ *IEEE Working Conference on Mining Software Repositories*, MSR '11, May 2011.

[5] A. J. Hindle, M. W. Godfrey, and R. C. Holt. What's hot and what's not: Windowed developer topic analysis. In *Proc. of IEEE Intl. Conference on Software Maintenance (ICSM-00)*, MSR '11, September 2009.

[6] C. J. Kapser and M. W. Godfrey. 'Cloning considered harmful' considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6), November/December 2008.

[7] M. Kersten and G. Murphy. Using task context to improve programmer productivity. In *Proc. of the* $14^{th}$ *ACM Intl. Symposium on Foundations of Software Engineering*, FSE '06, November 2006.

[8] R. Lu, X. Lin, X. Liang, and X. S. Shen. Secure provenance: The essential of bread and butter of data forensics in cloud computing. In *Proc. of the* $5^{th}$ *ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, April 2010.

[9] T. Menzies and T. Zimmermann. Guest editors' introduction: Software analytics: So what? *IEEE Software*, 30(4), July/August 2013.

[10] N. Nagappan, A. Zeller, and T. Zimmermann. Guest editors' introduction: Mining software archives. *IEEE Software*, 26(1), January 2009.

[11] M. P. Robillard, R. J. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4), July/August 2010.

[12] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3), September 2005.

[13] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proc. of* $32^{nd}$ *ACM/IEEE Intl. Conference on Software Engineering*, ICSE '10, May 2010.

[14] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. of* $30^{th}$ *ACM/IEEE Intl. Conference on Software Engineering*, ICSE '08, May 2008.

[15] Wikipedia. Alphonse Bertillon. `http://en.wikipedia.org/wiki/Alphonse_Bertillon`.

[16] Wikipedia. Diana and Actaeon (Titian). `http://en.wikipedia.org/wiki/Diana_and_Actaeon_(Titian)`.

[17] Wikipedia. Jaccard index. `http://en.wikipedia.org/wiki/Jaccard_index`.

[18] Wikipedia. Oracle v. Google. `http://en.wikipedia.org/wiki/Oracle_America,_Inc._v._Google,_Inc.`

[19] Wikipedia. Portrait of a Young Fiancée. `http://en.wikipedia.org/wiki/La_Bella_Principessa`.