A systematic mapping study of clone visualization

Citation for published version (APA):

Document license:
CC BY

DOI:
10.1016/j.cosrev.2020.100266

Document status and date:
Published: 18/08/2020

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
A systematic mapping study of clone visualization

Muhammad Hammad, Hamid Abdul Basit, Stan Jarzabek, Rainer Koschke

Abstract

Knowing code clones (similar code fragments) is helpful in software maintenance and re-engineering. As clone detectors return huge numbers of clones, visualization techniques have been proposed to make cloning information more comprehensible and useful for programmers. We present a mapping study of clone visualization techniques, classifying visualizations in respect to the user goals to be achieved by means of clone visualizations and relevant clone-related information needs. Our mapping study will aid tool users in selecting clone visualization tools suitable for the task at hand, tool vendors in improving capabilities of their tools, and researchers in identifying open problems in clone visualization research.

© 2020 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Contents

1. Introduction................................................................. 2
2. Related work.............................................................. 3
3. Literature survey process.............................................. 4
   3.1. Definition of research questions............................ 4
   3.2. Conduct of search................................................ 4
      3.2.1. Sources of information................................. 4
      3.2.2. Inclusion and exclusion criteria...................... 5
   3.3. Screening of papers.............................................. 6
   3.4. Keywording and classification................................ 6
   3.5. Mapping of papers onto the classification................ 7
4. User goals and information needs................................ 7
   4.1. Clone management activities.............................. 7
   4.2. Program comprehension...................................... 9
   4.3. Program evolution and test.................................. 10
   4.4. Defect management........................................... 10
   4.5. Quality assessment and improvement.................... 10
   4.6. Merging and compaction..................................... 11
   4.7. Re-use........................................................... 11
   4.8. Code mapping and evolution.............................. 12
   4.9. Origin analysis................................................ 12
   4.10. Malicious Software Detection............................ 13
   4.11. Productivity................................................... 13
   4.12. Summary....................................................... 14
5. Facets of clone visualization....................................... 14

* This study was supported by the grant of WZ/WI-IIT/3/2020 from Białystok University of Technology, Poland and founded from the resources for research by Ministry of Science and Higher Education, Poland.

* Corresponding author.

E-mail addresses: m.hammad@tue.nl (M. Hammad), hbasit@psu.edu.sa (H.A. Basit), s.jarzabek@pb.edu.pl (S. Jarzabek), koschke@uni-bremen.de (R. Koschke).

https://doi.org/10.1016/j.cosrev.2020.100266
1574-0137/© 2020 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).
1. Introduction

Software clones are similar code fragments within or across software systems. According to Roy and Cordy [1], around 5–50% of the code in software systems can be contained in clones. Clones occur in programs for a variety of reasons. At times, programmers introduce clones intentionally to achieve some engineering goal that can be more difficult to achieve using other means [2]; pattern-driven development may lead to clones; some clones may be caused by the limitations of the underlying programming language or may appear by accident. While not all clones are harmful or even avoidable, many clones contribute to the maintenance cost due to the increased program size or the need to consistently propagate changes across duplicated code during program updates [3]. Software maintenance and evolution are known to require a large part of the total costs spent in software development. Any automation of routine tasks can result in substantial improvements of software productivity. Much developer time is spent on trying to comprehend the system and to change code to fix an error or to enhance system functions. Automatically finding, analyzing, and visualizing clones can improve program understanding, reduce the maintenance cost, or help in software re-engineering for better maintainability or re-usability.

Code clones can be detected with the help of different algorithmic techniques, and several clone detection tools have been developed based on those techniques [1]. Clone detection tools typically identify a huge number of clones in large software systems. Even larger numbers of clones are reported when clone detectors are used to identify clones in a family of similar software systems, a frequent prelude to re-engineering such families...
into software product lines for systematic reuse [4]. Furthermore, tools often report false positive clones in the end result. Post-detection analysis of clones is then a must to help developers discard false positives and to zoom into the areas of interest in the bulk of cloning information [5]. Visualization, along with abstraction, clustering, and filtering are the cornerstone techniques for post-detection clone analysis.

Information visualization is an interdisciplinary area that helps humans analyze and understand data through the use of visual representations. It is defined as a “set of technologies that use visual computing to amplify human cognition with abstract information” [6]. The need to understand software has sparked much research on presenting information on software in a comprehensible form, which includes various forms of information visualization [7].

Most clone detectors report clones in textual form, which is hard for the user to comprehend. Clone visualization borrows from general information visualization and proposes new, clone-specific visualizations. Clone visualization techniques have been either integrated into clone detection tools or integrated development environments (IDE) or implemented as an independent clone visualization tool that works with the output of a clone detector. Each visualization technique provides a different view of the gathered clone data for the purpose of providing presumably a useful insight in achieving some specific software engineering tasks, such as detecting candidates for reuse or program understanding.

Extensive surveys have been published on software clones in general [1,8,9], but none of these has covered clone visualization comprehensively. Several surveys on visualizing other aspects of software exist [10–14], yet neither has any of them treated clone visualization. The only very brief summary of existing clone visualization was presented by Zibran [15] who tagged those techniques with respect to clone granularities (code fragments, functions, files, subsystems) and clone relationships (clone pair, clone class, super clone) in a two-page position paper.

Our survey represents the results of a comprehensive systematic mapping study on clone visualization. Systematic mapping studies – also known as scoping studies – give an overview of a research area through classification and counting contributions in relation to the categories of that classification [16–18]. They search the literature to uncover what topics have been covered and where the literature has been published [17]. Mapping studies are similar to systematic literature reviews (e.g., with respect to searching and study selection), yet they differ in terms of goals and thus approaches to data analysis [18]. According to Petersen et al. systematic reviews aim at synthesizing evidence for a given set of research questions, also considering the strength of evidence, whereas systematic mappings are primarily concerned with structuring a research area [18].

This paper presents a comprehensive and consolidated systematic mapping study of the existing scientific literature on clone visualization. We believe our survey will be useful to developers (i.e., developers of clone analysis tools), tool vendors, and researchers in the area of software clones. Tool developers will be able to assess capabilities of various visualizations, select visualizations relevant to their goals, and evaluate tool capabilities in view of their needs. Tool vendors will be able to design a consistent and comprehensive set of visualization techniques for their tools. Clone researchers may find our survey useful as a reference point in their work and in identifying open problems in the area of clone visualizations. Our paper aims at consolidating work on clone visualization done so far, setting a ground for advancing this important area.

In 2015, we published a paper on user goals and their mapping onto information needs related to software clones [19]. Based on this framework, we classified a subset of visualization techniques of software clones [20], providing a partial survey of existing clone visualizations. That preliminary survey, however, covered only a small set of visualizations and has not provided a broader and comprehensive view on those visualizations in terms of visual aspects, inherent strengths and weaknesses, and their overall contributions in relation to information needs and user goals. Furthermore, that paper did not provide a more thorough assessment of the research area, lacking aspects of a systematic mapping study such as current trends, most important publication venues, author contributions as well as a discussion of underdeveloped research areas of the field of clone visualization. In this paper, for completeness, we have refined and integrated material published as our preliminary work [19,20] on the classification framework of visualizations along user goals and information needs. While our earlier publication [20] has described only a smaller set of clone visualizations, we now present much more comprehensive results of a broader systematic search for clone visualizations (both in academic publication forums and in open-source and proprietary tools) and the mapping of those additionally found visualizations onto the classification. We describe the visualizations in much more detail based on both visual aspects as well as content and discuss their inherent strengths and weaknesses. In addition, we look at the publication activity in terms of where and when papers on clone detection are published, who has contributed to the literature, the gap between existing and needed visualizations, and the state of empirical evaluation in this field.

In summary, the added values over our previous publications are as follows: (1) we examine the full extent, range, and nature of research activity in clone visualization, (2) summarize and disseminate all current research findings to serve as a comprehensive inventory for other researchers in this field, and (3) identify research gaps in the existing literature at both broader and deeper scope.

The rest of the paper is structured as follows: Section 2 presents related work. Section 3 describes the survey methodology of gathering user goals, information needs, and visualizations. Section 4 describes user goals and their information needs. In Section 5 we will lay out the frame of reference in which we describe the types of clone visualization we found in the literature. Section 6 then explains the clone visualization techniques, their features, and our assessment in all facets of our frame of references. Section 7 answers our research questions regarding trends in publication, describes the evaluation of clone visualizations on the basis of user goals, summarizes the state of empirical evaluation, and discusses industrial adoption. Section 8, finally, summarizes the conclusions of this paper. To more easily identify the papers on clone visualization among our references, we have two distinct bibliographies at the end: one with general references and one with references to the publications in the scope of our survey. Normal references appear in brackets (e.g., [21]) while the letter P will be added in front of references to papers in the scope of our survey (e.g., [P1]). In addition to that, we have a list of URLs at the end referring to web sites of clone-visualization tools. For these, we put the letter U in front of the reference number (e.g., [U1]).

2. Related work

This section describes related work covering visualizations in other domains, surveys done in the domain of clones from other perspectives, and other papers that discuss clone visualization techniques. Before we come to that we summarize the terminology generally used in the literature relating to clones and used in this paper, too.

We will use the term code fragment for every unit that can be duplicated, which subsumes sequences of tokens, lines of source
code, whole methods, classes, or any other kind of code or model entity, including files, directories, and even whole programs.

Duplicated code fragments come in various types, which are usually distinguished on the basis of syntactic and functional similarity [22]. A type-1 clone is an exact duplication of a code fragment, which allows changes only in layout and comments. A type-2 clone represents syntactically equivalent code with only parametric differences (e.g., in identifiers, types, or literals), layout, and comments. For a type-3 clone, code can be modified in arbitrary ways by adding or deleting statements or parts of them. Finally, a type-4 clone represents semantic resemblance among two or more code fragments, where the syntax may be totally different. Cloned code fragments form clone pairs and clone classes. A clone pair consists of two copied fragments, whereas a clone class is a group of two or more similar code fragments wherein any two of the fragments constitute a clone pair. An element of a clone class is named clone instance.

Lower-level clones can be aggregated into higher-level clones based on clone-graph structures. A clone-graph structure is a graph whose nodes are clone instances of the same clone class and whose edges represent arbitrary relations among these, for instance, call dependencies. A structural clone relation according to Basit and Jarzabek [23] holds between two clone-graph structures S1 and S2 if they are isomorphic, that is, if (and only if):

1. there is a bijective mapping between S1 and S2 for each clone instance contained therein induced by a clone relation of any type (type 1 to 4)
2. there is a bijective mapping between S1 and S2 for each relation contained therein

A special case for a relation of two clone instances c1 and c2 in clone-graph structures is contained-in-same-file(c1, c2), which holds if c1 and c2 are contained in the same file. Rieger et al. [946] call such special structural clones clone class families and Jiang et al. [229] super clones.

Koschke [8,24] has done a thorough survey on the wider area of software clones, focusing on clone categorizations, clone detection, and clone evolution. The discussion of clone visualizations is, however, very brief. Similarly, Roy and Cordy [1] provided a detailed survey of clone detection techniques [25], but again the section on clone visualization is brief.

Software clone visualization is a sub-domain of software visualization. A number of surveys have been done on different aspects of software visualizations in general. Novais et al. [26] present a systematic study of software evolution visualization technologies, discuss the support of maintenance tasks each visualization is supporting, and compare them to each other. Carpendale and Ghanam [10] discuss software architecture visualizations, comparing visualizations on the basis of dimensionality, multiplicity of views, and use of metaphors. They also suggest characteristics of effective visualizations for software architecture. Teyseyre and Campo [11] have surveyed 3D software visualizations in general, focusing on visual representations, development tools, evaluation methods, and interaction issues. Bassi and Keller [27] evaluate software visualization tools in a study involving more than 100 participants. They identify various aspects of these visualizations, such as functional, practical, cognitive, and coding issues. The participants of their study evaluated visualizations according to the usefulness and importance of these aspects, and suggested criteria to rank these visualizations. The authors also propose improvements to current tools. Caserta and Zendra [14] have surveyed 2D and 3D visualization techniques for static analysis and evolution of software systems. They categorize these techniques according to the features that assist in comparing them and finding the most appropriate techniques for a specific problem. Koschke [13] has also surveyed software visualization by focusing on getting the perspectives of researchers on software visualizations. Yet, none of these surveys summarizes work in the area of clone visualization. Zibran [15] discusses the importance of clone analyses and visualizations with respect to inheritance hierarchy and call graphs in dealing with clones for refactoring and very briefly lists 17 clone visualizations proposed in the literature in a table, which classifies those techniques only with respect to clone granularities (code fragments, functions, files, subsystems) and clone relationships (clone pair, clone class, super clone). The paper is just a two-page position paper, which of course does not provide enough space for a comprehensive summary. Overall, we can conclude that a more comprehensive and up-to-date survey of software clone visualization does not exist.

3. Literature survey process

We conducted a systematic mapping study of clone visualization techniques by following existing guidelines [16–18,28,29]. The steps of systematic mapping studies according to Petersen et al. [17] are: (1) definition of research questions, (2) conduct of search, (3) screening of papers, (4) keywording using abstracts, and (5) data extraction and mapping of papers onto the classification. This section describes these steps in detail.

3.1. Definition of research questions

Table 1 summarizes the research questions of our literature review. Beyond presenting aspects generally considered in mapping studies, such as where, when and by whom papers are published (RQ1), the contribution of those papers to the research theme (RQ2), and their classification (RQ3), we also assess the current state of empirical studies in the field (RQ4) and investigate possible gaps between existing and needed visualization (RQ5).

One of the main purposes of this literature survey is to find and categorize clone visualization techniques and to relate them to user goals and information needs (RQ3). Every visualization is an abstraction designed to support human exploration and sense making for different types of human viewers having different goals and is therefore highly contextual. The specifics of a user goal determine the kind of information needed to reach this goal and, as a consequence, what is to be conveyed by a visualization. User goals and their associated information needs help to assess and categorize the types of visualizations and to put them into context. Beyond inherent features, strengths, and weaknesses of different types of visualizations with regards to the general human perception, we gather what kinds of information needs visualizations address and, consequently, for which kinds of user goals related to software clones they may help. For these reasons, we searched the clone literature not only for clone visualizations but also for user goals and information needs.

3.2. Conduct of search

Our protocol for the second step in the above process for systematic mapping studies (“conduct of search”) consisted of determining the literature databases and sources in which to search and then searching and extracting data from these sources. When executing this protocol, we frequently iterated after discussions and cross-checking in the light of the current findings, for instance, when a found paper provided us with additional literature data bases. Thus, in reality we did not follow a waterfall model, but a rather iterative process. Nevertheless, for the sake of better understanding, we will present it as consecutive steps.
Because mapping studies aim at research trends, Kitchenham et al. [30] argue that the search requirements are less stringent compared to systematic reviews. Only for the latter, Kitchenham et al. [30] recommend to aim at finding all studies relevant to a research question. Even so, we tried to reach a coverage of the existing literature as comprehensive as possible in our mapping study.

Similarly, quality assessment of the found papers is more essential in systematic reviews to determine the rigor and relevance of the included studies, because systematic reviews search for evidence for a given set of research questions. In a mapping study instead, it is useful to include papers at an early stage of maturity with respect to empirical evidence. In particular, new trends often start with initial ideas which are not fully explored empirically yet. Hence, mapping studies should include existing papers even if they have no convincing empirical assessment yet. Wieringa et al. [32] propose a classification of research types regarding the different levels of empirical evidence. “Philosophical papers” such as sketch a new way of looking at things and a new conceptual framework, without assessment in real settings. Such premature work would generally not be included in a systematic review, though in systematic mapping studies it is important to spot trends of emerging topics. Hence, we include papers without empirical evaluation, but do assess the state of empirical evidence for clone visualization in Section 7.

3.2.1. Sources of information

As recommended by Petersen et al. [18], we used different strategies for the search to ensure a high coverage of the existing literature. First, we used existing surveys on clones in general [1,8,9,33–36] and the references therein. Another important initial source was Tairas’s quite comprehensive list of clone papers.2 This list consists of more than 360 references to research papers on all aspects of software clones and is arguably complete until 2013. Ten papers in this list refer to visualization in their title. Tairas has systematically browsed and collected the literature until 2013 and also received and integrated references from the clone-research community.

Second, we selected a set of electronic databases to find highly relevant material, following the recommendations of Brereton et al. [28] and Kitchenham and Charters [16]. Table 2 mentions the list of databases used to conduct this survey.

By looking at these, we cover also papers published after 2013—at the time when Tairas stopped to maintain his list of literature. We identified key phrases from the research questions and gathered synonyms and alternate spellings of the key phrases for our queries of the digital literature data bases. Then we joined each key phrase with its possible synonyms and alternate spellings using the Boolean OR operator and combined these with other key phrases using the Boolean AND operator. The resulting query was mentioned in Table 3.

In addition to the digital libraries, we used Google’s search engine to browse the Internet for papers and clone tools (in particular, commercial tools) not listed in those data bases, using the same query as above (Google supports AND (\∧) and OR (\∨) operators and grouping by brackets). In addition to this query, we also searched for the phrases to find additional hints on clone visualization tools mentioned in Table 3, by appending a specific name of a visualization. Hints on clone tools could also be found in the published literature when a tool name was stated or an URL was provided. Mostly commercial tools not mentioned in the published scientific literature were found by Google’s search engine. In case of tools, we looked at the advertised material of the tool vendors on their web sites, considered claims of researchers in their publications, and explored publicly available feature descriptions such as scientific publications, web sites, demo videos, and tutorials, over the Internet. We were not able to run and test all tools because some of them were not publicly available—they were merely mentioned in publications—or were not free. Other tools had been developed a long time ago for meanwhile outdated versions of Windows, Eclipse, or Java. However, for all tools mentioned in our survey, the publicly available material was comprehensive enough to form a firm ground for our analysis.

As a third measure to ensure high coverage, we used snowball sampling in both directions: forward and backward. That is, we looked not only at the papers cited in the papers we found but also searched for papers citing the found papers whenever a digital library allowed. Moreover, we presented preliminary results to experts of the clone community at workshops and conferences [19,20] and through personal contacts. We extended the initial scope and used different strategies to make sure

---

1 Wohlin et al. [31] counter-argued that full coverage may be unrealistic for systematic reviews and a good sample with respect to the characteristics of the population the reviewers are aware of should suffice.

that our survey covers the relevant literature on software clone visualizations as comprehensively as possible.

3.2.2. Inclusion and exclusion criteria

We included publications in peer-reviewed journals, conferences and workshops. Publications in dubious venues without rigorous quality control were excluded. Ph.D. dissertations are also reviewed publications and often summarize other publications that were part of the Ph.D. research and – because they generally offer more space – provide additional details. Some publications start as a Master’s thesis work and are later turned into a peer-reviewed publication. For these reasons, we included also Ph.D. dissertations and Master’s theses if there was at least one other kind of included publication citing them. We also included technical reports but only if they were frequently cited in other included publications and if they were different from every other publication included (some authors publish their accepted publications or earlier versions of these also as technical reports). We also looked at tool websites, tutorials, demo videos, and the like to gather additional information, yet do not count them in the publication statistics.

Clone is a general term also used in other fields such as biology, networks, or mathematics. We included only papers related to software clones, of course, that is, duplication in source code or software models. We excluded papers on duplication in documents other than software that cannot be transferred to software clones, for instance, written student assignments outside programming.

All papers had to have a discernible focus on clone visualization, for instance, by proposing a new clone visualization or by evaluating a visualization. We did not insist that visualization is the only subject of the paper, however. Many papers have multiple focus points, and visualization is only of one them. Furthermore, there are also papers primarily on a technique to manage clones that mention a clone visualization to present a particular aspect just in passing, but spend only very little of its space on this visualization. We included such papers – where clone visualization is just a secondary focus point –, too, for reasons of completeness. Later in Section 7.1, we make a clear distinction between papers where visualization is the only, one out of many, or just a secondary focus point.

3.3. Screening of papers

We identified more than 160 papers based on the search query for finding clone visualization literature, which were narrowed down to 110 papers after applying our inclusion and exclusion criteria. All four authors searched for relevant papers. The relevance of a paper was assessed by its title, abstract, and a brief glance at the content of the paper by a single person initially as follows:

- Read the title and discard any paper, which is obviously not related to any of our research questions.
- Read the abstract and keywords to discard any paper that had a related title, but is not related to our research questions.
- Read the paper and include it in the list, if any of the research questions are addressed.

When there was not a single doubt that the paper did not fulfill our inclusion criteria or fulfilled any of our exclusion criteria, the paper was excluded immediately. If the judging person was not fully convinced that a paper was out of scope, at least one other judge was consulted.

Our survey covers 68 publications we found relevant because they introduce or evaluate clone visualization. These were published in scientific journals, conferences, workshops, books, technical papers, and Ph.D. and Master’s theses. This number excludes ten survey papers related to clones. Out of these, eight also cover the subject of clone visualization, but do not propose or evaluate clone visualization techniques, merely summarize relevant papers. In Section 7, we will return to more detailed statistics on the publications we found.

3.4. Keywording and classification

Often, a classification is created in the course of a systematic mapping study – step (4) above – by labeling through extracting keywords and grouping similarly labeled papers in a bottom-up fashion [17, 18]. While this bottom-up process may assist in organizing and presenting a research area, it may fail in identifying the gap between what actually exists and what should exist as it retrieves the classification only from existing literature. In this paper, we rather start with an existing classification of the goals developers may have in handling clones and for which they may take advantage of visualization. A user goal could be, for instance, the migration of a set of copy-paste variants into an organized software-product line to save maintenance costs. Each user goal has specific clone-related information needs—information that helps developers to achieve that goal. For the product-line migration, for instance, developers would need to know the degree of copying and pasting and the differences between the variants.

A particular type of visualization shows a certain type of information about clones and thus addresses an information need that arises in the context of a goal a user wants to achieve. Mapping visualizations onto information needs and those in turn onto user goals allows one to identify the user goals a visualization can support. Vice versa, it allows to identify the existing types of visualization that may be supportive for a given user goal. Furthermore, user goals and information needs for which no or only a few types of visualization exist today pinpoint a gap that should be addressed by future research. Therefore, we annotated visualization techniques in our study with the information needs they satisfy and in turn the user goals this information supports rather than extracting keywords to form a bottom-up classification. We trust a faceted classification framework for visualization techniques formed in that way will help readers to easily comprehend the results of our survey and more directly apply them in practical scenarios. It enables a reader to better see when and how each visualization can be useful in a real task.

We gathered user goals and their information needs in earlier research [19] based on initially 50 research papers not necessarily related to visualization specifically. Beyond that, we brainstormed additional information needs that were not mentioned by the authors, but would be needed in practice to fully address a given user goal (where our own experience using and developing clone detectors and researching software clones came in handy). Thus, the user goals did not only help us categorize the information needs, they also helped us to identify information needs that were not covered by the descriptions of user goals in the original papers. The brainstorming was first done individually by the authors of this paper and then jointly continued in the group as a whole to reach a consensus. Altogether, we identified 25 user goals and 29 relevant information needs. The results were published at the International Workshop on Software Clones (IWSC) [19] and presented there to international experts of the clone research community. Our previous publication describes the process of our search for user goals and information needs in software cloning and lists all the identified papers [19]. We will summarize its result in Section 4.
We took into account the feedback we received from the experts in the software-clone community, further refined the classification, collected a preliminary set of published papers on clone visualization, and mapped these onto the classification to further assess the suitability of the classification. This preliminary mapping study demonstrated that the classification indeed allows us to organize research papers on clone visualization. The classification turned out to be stable and comprehensive enough to map all identified types of clone visualization. The results of that preliminary mapping study were published at the Working Conference on Software Visualization (VISSOFT) [20], this time presented to international experts on software visualization, where we again collected feedback from relevant experts.

In summary, we presented our preliminary works on the classification that have lead to the mapping study described in this paper to experts of two domains related to the focus of our study, namely, software cloning and software visualization. In doing so, we followed the guidelines for systematic mapping studies recommended by Petersen et al. [18], who suggested to consult with pertinent experts as a measure of quality assurance. We do not claim that our list of user goals and information needs is complete, but at least it should be a good starting point. We invite other researchers to extend this list.

3.5. Mapping of papers onto the classification

The papers meeting our criteria were tagged by the user goals and information needs of the above classification by us using a two-step approach. One judge looked at a paper in full detail and provided an initial tagging. This work was distributed among all of us. The individual tagging was then presented to the team as a whole. The initial judge presented the evidence and the other judges discussed it in a virtual team meeting with all judges until a consensus was reached.

Whenever the selected papers used different terms to describe the same user goal, we selected a term most commonly used. Sometimes papers or tools did not explicitly state information needs, however, goals and needs could be inferred from the description of the results produced by the methods described in the paper. We considered such implied information needs and user goals in our mapping, too.

Developers would also like to know how well a particular type of visualization actually works, that is, they will ask for empirical evidence. Conducting only a mapping study, we cannot ourselves run empirical studies to assess the real contributions of the proposed visualizations—their number is simply too large. Yet, we did annotate the papers as to whether they provide empirical evidence of the actual usefulness of the proposed visualization. The outcome will be presented in Section 7.3.

4. User goals and information needs

Clones are detected, analyzed and visualized to help programmers complete software development tasks—called user goals in our survey—such as refactoring, bug detection or identifying candidates for reusable components. A user goal is the reason for detecting and then visualizing clones. Each user goal has specific requirements for clone-related information—called information needs in this survey—that can help developers achieve that goal. Suppose a programmer changes code to fix a defect. As changing code can have unwanted ripple effects, a programmer is likely to take further investigation into the code areas potentially affected by the change. To this end, a programmer may ask these questions: Is this defect embedded in code that has been copied elsewhere? If so, where is it copied to? How different is the context of the copied code from the context of the original code? Will the same patch work in this new context?

Information needs for bug detection comprise data that help a programmer in answering the above questions. This may include information about types of clones that can be helpful in a given situation, their size, location, a similarity threshold that clones should meet, among many others. Any given clone visualization technique addresses only some of the information needs, with various degrees of relevance and usefulness. The two aspects of user goals and information needs create a useful framework for classifying visualization techniques that helps us comprehend and explain visualization techniques. By positioning each visualization technique in respect to the relevant user goals and information needs, the reader will get a clearer picture of when and how a given visualization technique can be used. This approach is similar to the goal-question-metric approach by Basili and Weiss [37], where first goals for data collections are set, then a list of questions of interests is established related to the goals, and finally the data collection is devised. In our case, the next step is the visualization of the collected data.

In the following sections, we summarize 24 user goals. This summary is a major reorganization of the user goals listed in our earlier publication [19]. We have excluded two miscellaneous user goals, namely, Ontology Alignment [38] and Web-Service Similarity Detection [39], which use clone detection as an enabling technology. The reason for excluding them here is that these two user goals are rather exceptional. Furthermore, we refined the rather generic Program-Code Comprehension user goal into the more concrete new user goals Program-Concept Assignment and Feature Location and added two maintenance user goals Evolution and Testing. In addition to that, we have clustered those user goals in terms of their information needs, which will be elaborated in Section 4.12 in more detail. Furthermore, to simplify the description, we have organized the user goals into cohesive groups related to software engineering tasks such as Re-Use or Defect Management and introduced a layer of clone-management goals that we use as building blocks in the description of more general user goals (see Section 4.1).

The information needs are summarized in Table 4. Later in Section 6, we will delve into the details about how each visualization addresses the information needs. In comparison to our earlier publication [19], we have removed some redundant information needs and rephrased some of those questions to make them clearer. In the following description of the user goals, we will refer to these information needs by labels in the form of $N_{xx}$, where $xx$ is a unique number found in Table 4.

4.1. Clone management activities

Clone management summarizes all process activities that are targeted at detecting, tracing, assessing, avoiding, removing or otherwise handling clones [40]. These activities appear only rarely as an explicit task description for a developer or a work package in a project plan, but rather as an implicit, yet necessary part of a more general goal such as quality improvement or feature development. For instance, a developer typically may have to make a change, then finds cloned code hindering the primary change, and in preparation to the primary change, she or he handles the clones as a secondary goal.

Whether user goals related to clone management are primary goals or not— at any rate, introducing these here helps us to ease the description of the more general user goals that must deal with clones somehow. The user goals presented in this section serve as coherent building blocks for the other more general user goals in the next sections. For this reason, they are chosen so that they do not overlap in terms of the information needs
they address. In many cases, they will interact, however. For instance, all clone-management activities need some sort of clone detection and many goals need to know the differences among clones. Yet, we will keep detection and differencing separate here. The interaction will become visible later in the more general user goals. For instance, the more general goal of finding bugs created by inconsistently changed clones will require both clone detection and differencing.

**Clone Detection:** First of all, clones must be detected, which is addressed in the activity *Clone Detection*. It has different variants depending upon whether all clones should be found and how the scope of the detection is chosen. In many cases, all clone classes must be found (N1), that is, the result must be comprehensive. Yet there are also cases in which only those clones are relevant that resemble a single particular given piece of code—in other words, all instances of a single clone class are needed (N2). An example scenario for the latter is when a developer found a defect in a piece of code and needs to check whether the same defect exists elsewhere in similar pieces of code, too. Regarding to the scope of the search, clones may be searched within a program unit or across units, where a unit can be any kind of code container such as a file, a package, or a whole program. An example for clone detection across units is when copyright violations or cases of plagiarism are suspected. Then only clones between different units are of interest.

The two dimensions—*all* versus *single* and *within* versus *across*—allow four different combinations leading to conceptual variants of clone detection. How these conceptual variants are implemented is irrelevant for our purpose as we are concerned here about the information needs rather than implementation details. For instance, to obtain a single clone class across two units, a general clone detector yielding all clones could be applied to the union of the two units and then its results could be filtered to those for a particular piece of code occurring across units.

**Clone Detection** answers the question where clones exist in program units. The units themselves may be hierarchical, e.g., a package hierarchy in Java, a directory hierarchy in C, or a logical class hierarchy. When the units are hierarchical, the found clones in one level of the hierarchy can be aggregated to higher levels, an aspect that can be relevant to clone analysis and visualization.

**Clone Differencing:** When clones of type 2 or higher are found, a developer needs to know their differences (N9). The difference may not only be in the clones themselves but also in the context in which they are embedded (N12). Clone Differencing is the activity determining these differences.

**Clone Tracking:** Clones are not only searched for in the latest version of a program but also in previous versions, for instance, to fix a duplicated bug in previous versions that are still operational. *Clone Tracking* traces clones across time in terms of versions. It is also necessary to study trends in cloning and addresses the information needs how a clone evolved (N17, N18, N19, N20, N21).

**Clone Awareness:** It is not enough to only detect clones, developers must also be made aware of them. Because there are generally too many clones, chances are that the clones in the scope of the code in which a developer is currently working are overlooked. The activity *Clone Awareness* aims at making developers aware of the presence of clones on demand (N16). If a developer works on a piece of code that is cloned, knowing the locations of all its clone instances in a program is essential for consistent changes [41]. Harder and Tiarks [42], for instance, have observed in an experiment that developers may miss clones requiring bug fixes. Not rarely, developers overlook clones even though they are presented in their IDE [43]. That is, some developers ignored the information that was given to them. It is important for the acceptance by developers to achieve *Clone Awareness* in a non-intrusive way that minimizes interference with their primary tasks, but gains the attention of the developer when it is needed. Software visualization should be at the heart of this clone-management activity. *Clone Awareness* includes also to document the reasons why a clone exists, which may be valuable information when it comes to decide whether a clone should be removed. As mentioned above already, in bug fixing for a particular piece of code, the same bug might also need to be fixed in earlier versions, hence, there is an interplay with *Clone Tracking*. Obviously, that does not only hold for code that has been copied. In a way, versions of code may be viewed as clones in time. Awareness should be raised not only when code is viewed or changed, but also when other related artifacts such as architecture or requirement specifications are consulted. If a plan is made to modify part of the architecture or specification, it is helpful to be aware of copied code in the source code that can be traced back to that part, which is the subject of *Clone Linkage* described below.

**Clone Assessment and Ranking:** Typically, clone detectors find many clones in a system. Not all of them, however, are

---

**Table 4** *Information Needs (Revised from our Previous Paper [20]).*

| N1 | Where are clones in a given program unit located? |
| N2 | Where are instances of a given clone class located? |
| N3 | Which clones exist across programs or program variants? |
| N4 | How many instances are in a clone class? |
| N5 | How large is a cloned fragment? |
| N6 | How much is cloned in a given program unit? |
| N7 | How much code is contained in a clone class (volume: the total sum over the size of all instances of a clone class)? |
| N8 | What types of syntactic structures are contained in a clone? |
| N9 | What are the differences among the instances of a clone class? |
| N10 | How different are the clones? |
| N11 | What is the relevance ranking of clones for a particular task at hand? |
| N12 | What are the differences in the context of clones (e.g., the units containing the clone instances)? |
| N13 | How much code of one program unit is cloned in another program unit? |
| N14 | How much code of one program unit is cloned across a particular given group of program units? |
| N15 | What is the size of the program unit containing clones? |
| N16 | Is there a clone in the current scope of the program? |
| N17 | Which clones are present across versions of the evolving system and which clones disappeared? |
| N18 | How has the location of a clone changed in different system versions? |
| N19 | How has a clone changed from one version to another? |
| N20 | What is the overall trend of cloning across versions? |
| N21 | What is the spread of cloning across versions? |
| N22 | What are the effects of a clone onto quality and maintenance costs? |
| N23 | Who is the owner of original and copied code? |
| N24 | What is the purpose of the clone? |
| N25 | Is the code private? |
| N26 | How much code can be eliminated when clones are removed? |
| N27 | How much code can be eliminated when a clone class is removed? |
| N28 | What is the reduction of maintenance effort? |
| N29 | Which clones can be removed? |
| N30 | How can a given clone be removed? |
| N31 | What are the costs and risks of removing a given clone? |
| N32 | How are the clone classes related to each other (e.g., are their instances contained in the same files)? |
| N33 | How are clones related to formal and non-formal system artifacts other than code? |
| N34 | What is the higher-level abstraction/concept behind the clones or clone classes? |
| N35 | What is the reason for the clones? |
| N36 | Which other instances of a clone class need to be updated? |
| N37 | How to update other instances of a clone class? |
| N38 | Which (inconsistent) type-2 or type-3 clones must be re-synchronized? |
relevant for a particular task. Hence, clones must be assessed and ranked with respect to a particular concern, which is the subject of activity **Clone Assessment and Ranking** [44]. Various metrics can be used to assess and rank clones in order to gauge their relevance and impact (N11). The metrics may relate to the clones themselves (N4, N5, N6, N7, N8) – for instance, the length of a clone – or the program units containing them (N13, N14, N15) – for instance, the length of a file containing a clone. Such metrics could also quantify evolutionary aspects of the clones (for instance, how often they were changed in the version history) or the degree of differences among instances of the same clone class, in which case there would be an interaction with **Clone Differentiating and Clone Tracking**, respectively. Based on those metrics, it may also be possible to measure or estimate a clone’s impact on maintenance costs (N22) and to predict the benefit of removing a clone through **Clone Removal** (N26, N27, N28).

**Clone Removal:** If the decision is made to remove a clone, **Clone Removal** comes into play. Here a developer needs to know which clone instances can be eliminated and how (N29, N30). Impact and risks of removing the clones must be assessed and mitigated (N31). There may be many means to remove a clone [45] [55]; refactorings are just one option; another one is code generation [45]. All these options must be gauged. The assessment of risks and benefits of each option may be valuable information for **Clone Assessment and Ranking**.

**Clone Compensation** aims at limiting the harmful effects of clones that remain in a program [47]. If a change needs to be made, means are required that ensure consistent changes [47]. **Clone Awareness** and **Clone Differentiating** – the latter for inspecting the differences if there are any – are necessary prerequisites. If changes have become inconsistent in previous versions, the missed changes in earlier versions may also need to be re-applied [48], hence, there is also a connection to **Clone Tracking**. The additional information needs beyond what is required for these three other activities is to select the clone instances that need to be updated, given the chance that some divergent changes are actually wanted—and how to adjust those instances truly requiring similar updates (N36, N37). Once that is solved, linked editing may be used to save time and to prevent editing errors by linking code clones that need to stay in sync – the siblings – and replicating modifications in one fragment to all its siblings automatically [55]. Another aspect of **Clone Compensation** is to identify clones that have already become inconsistent undesirably and which require so-called **late propagations** [49] to turn them into consistent clones again (N38). Type-3 clones are inconsistent by definition, but also type-2 may be inconsistent if the parameter substitution is inconsistent. There is a chance that both kinds of inconsistencies represent defects [50–52].

**Clone Ownership:** Copying and pasting code may also have legal issues such as plagiarism or copyright infringement. **Clone Ownership** determines the owner and possibly the license of a clone (N23, N24). Related to license issues is that sometimes code should not be made public in the first place. Occasionally, private code is leaked in organizations developing both open-source as well as proprietary code when a developer accidentally copies private code to a public place. Hence, the question arises whether any code that should have been kept private has been copied to a publicly accessible code repository (N25). Knowing the owner of clone instances may also be relevant for the communication among developers, for instance, if a developer A copies code from another developer B and B makes changes to the original code, A may need to be informed. Thus, **Clone Ownership** may also play a role in **Clone Awareness**. Similarly, if multiple developers change clone instances, chances are that they become inconsistent [53]. Hence, the ownership may also be an aspect relevant to **Clone Assessment and Ranking**.

**Clone Linkage** is an activity that relates clone classes to each other (N32) or to artifacts other than code (N33). For instance, Basit and Jarzabek [54] describe an approach in which they build structural clones (see Section 2) from clone classes related to each other by co-occurring in the same files. Clones – as any other type of code – may be traced back to documentation of design decisions or requirements. Traceability links from these formal or non-formal artifacts to cloned code helps to assess the impact of changes to these (**Clone Assessment and Ranking**), to keep documentation and code in sync (**Clone Compensation**), and to understand reasons for clones (see **Clone Reason and Prevention**).

**Clone Conceptualization** is the attempt to understand the meaning of a clone, that is, the higher-level concept behind a given piece of code that has been copied (N34). Programmers do not copy code arbitrarily; they want to re-use an implemented concept. For instance, two similar pieces of copied nested loops may both implement bubble sort, one in ascending and one in descending order. Thus, the fact that code has been copied is a strong indicator that it has meaning and forms a concept. Conceptualization is a prerequisite for any treatment of clones that requires a deeper semantic understanding. Being able to trace back copied code to the same or similar requirements in **Clone Linkage** eases **Clone Conceptualization**.

**Clone Reason and Prevention:** Whenever possible and reasonable, creation of clones should be prevented [47,55]. To do that, it is necessary to understand the underlying reasons of cloning (N35). Prevention may be achieved by training of developers, organizational changes, reorganization of the architecture, improvement of programming languages, among many other means. **Clone Reason and Prevention** is the activity devoted to these aspects. To understand the reasons for a clone, it is necessary to get a deeper semantic understanding of the higher-level abstraction or concept behind a clone. Similar requirements or particular design decisions may contribute to the willingness of developers to copy code. Thus, effective clone prevention requires a semantic level of cloning and, thus, benefits from **Clone Conceptualization** and **Clone Linkage**.

Now that we have described basic clone-management activities with non-overlapping information needs, we will turn to the more general user goals where the activities described in this section come together as building blocks.

### 4.2. Program comprehension

Program Comprehension is the process of gaining new or regaining lost knowledge about a computer program. Due to the generic nature of program understanding and the fact that some level of program understanding is a prerequisite for virtually every software-development activity, we will further refine it into more specific activities. Program comprehension starts at the level of code (**Program-Code Understanding**) and then attempts to raise the level of abstraction to design (**Design Recovery**) and application domain (**Domain Analysis**). Program-Code Understanding subsumes tasks such as Program-Concept Assignment or Feature Location.

**Program-Concept Assignment:** If a developer needs to understand the behavior and purpose of a given piece of code, he or she needs to determine the underlying programming concept the given code implements, for instance, what kind of sorting algorithm the code implements. Program-Concept Assignment at the level of code is exactly this process: mapping a piece of code onto a programming concept. Knowing clones can help in assigning program concepts: if a developer already understood a piece of code, she or he can detect other similar pieces of that code that perform similar functions [56] (**Clone Conceptualization**). For example, when we have a code fragment containing a search
algorithm, we can infer that all files that contain a copy of this code also perform some kind of search. Knowing the syntactic differences among the found clone instances as well as the differences in the contexts in which they are contained (Clone Differencing) allows one to determine the differences among similar programming-concepts implementations.

**Feature Location**: While Program-Concept Assignment starts at a given piece of code and tries to figure out the feature it implements, Feature Location is the reverse process. Here a developer searches for pieces of code that implement a given feature of interest, for instance, to fix a bug of that feature. If a developer searches for a feature $f$ and already knows that a piece of code $c$ implements a feature $f'$ similar to $f$, she or he can search for code similar to $c$ (Clone Class Detection within Program Unit) and inspect the differences of the cloned content as well as the differences in the respective contexts the two pieces of code are embedded (Clone Differencing). Although Program-Concept Assignment and Feature Location are inverse to each other, the potential contribution of clone detection to these and, hence, the information needs specific to cloning are the same.

**Design Recovery**: Research shows that bigger program structures that have been cloned often mark the presence of higher-level design concepts in a program [54]. Finding such repetitions (All-Clone Detection within Unit) along with their differences in representation and context (Clone Differencing) to form higher-level clones can aid in Design Recovery, which is the process of examining program code to find significant design information or abstractions (Clone Conceptualization). Such higher-level clones may be formed by relations among clone classes (N32 in Clone Linkage). Knowing clones may also help in architecture recovery in software-product lines based on the reflexion method [57] where existing reverse engineered mappings of cloned code onto architecture components may be re-used across variants [58].

**Domain Analysis** often involves analyzing commonalities and variabilities in software systems in a given domain. Clones found across systems signify commonalities in the domain [59]. The relevant information needs for this user goal are to locate all clone classes across systems in a given domain (N3) and identifying how these systems and their clones differ from each other (Clone Differencing). Typically, a Design Recovery process is required in advance to find domain concepts at a higher level of abstraction (Clone Conceptualization), such that all information needs for Design Recovery are relevant for Domain Analysis, too.

### 4.3 Program evolution and test

**Evolution**: When changes need to be made to a piece of code due to changed requirements, similar changes may need to be made to similar pieces of code. Hence, finding other cloned instances of a piece of code to be adjusted (N2, N36, N37) and inspecting their representational and contextual differences may be needed (Clone Differencing). Possibly, while making a change, developers need to be notified proactively of potential clones of the code being changed (Clone Awareness). Associated documentation may need to be consulted or updated (N33). Possibly Program-Concept Assignment, Feature Location, or Design Recovery may be necessary prerequisites, hence, all their information needs may apply here, too.

**Testing**: When code units are similar, chances are that their associated test cases are similar, too. Hence, it may be possible to re-use test cases through manual or automated tranplantation [60]. Moreover, test code as any other type of code is often cloned and needs to be adjusted during program evolution, too [P54], thus all information needs for the user goal Evolution are relevant for Testing, too.

### 4.4 Defect management

Empirical studies indicate that cloning code may introduce bugs. There is a number of reasons for that. First, the original code may already be defective, in which case a defect is replicated. Second, code that works correctly in the original context may introduce subtle unexpected side effects when copied into another context where it does not quite fit [61] and third, the adjustment to the duplicated code may be buggy. For instance, substituting parameters inconsistently is known to introduce problems [62], and clones that evolve inconsistently [63] are also known to introduce potential defects.

**Bug Detection** aims at finding defective code. Clone detection may help here by searching for all clones (All-Clone Detection) that may be defective due to inconsistencies either in the code itself or in the context in which they are embedded (Clone Differencing). For this user goal, the inconsistently changed clones are of primary interest (N38). To better understand the inconsistent changes, it may be worth looking at the point in the version history at which the changes have become inconsistent (N17 and N19 from activity Clone Tracking).

**Duplicated Bugs**: If defective code has been found already, clone detection can be used to locate similar code with similar defects (Clone Class Detection) and help analyze the potential differences both in the clone instances themselves and their contexts (Clone Differencing). The defective code may have been introduced earlier in the version history and different versions still in operation may be affected, too. Hence, in many cases the defective code must not only be looked up in the latest version but in many previous versions, too (N17, N19 from activity Clone Tracking), and possibly also between different program variants or programs in which code has been re-used (N3 in Clone Class Detection). For instance, [64] have developed an approximate search of similar code that allows one to detect similar pieces of code not only in one version of a program, but also in all its earlier versions to check whether a defect must also be fixed in other versions that are still in operation. Li and Ernst [65] report on a study in which they find duplicated defects in various large programs (Linux, Git, PostgreSQL, commercial product line). Jang et al. [66] provide tool support to search for unpatched code in product lines or multiple revisions that are all in use. Unpatched code is similar code where a patch has not yet been applied to.

### 4.5 Quality assessment and improvement

**Quality Assessment**: A large number of clones may be an indicator of low software quality. Quality Assessment in the context of clones is to get an estimate on how clones affect the overall system quality and to find hot spots, i.e., parts of a system that are most heavily cloned. Thus, Clone Assessment and Ranking based on all clones (All-Clone Detection) and their differences (Clone Differencing) is a central part of Quality Assessment.

The potential metrics considered by Clone Assessment and Ranking in the context of Quality Assessment beyond the size, number, and diversity of clones include reduction in maintenance and audit cost, increased quality of internal assessment, or improved quality of the third-party assessment of suppliers, for instance, for software escrow [41]. Juergens et al. [67] have proposed a cost model for software with clones that takes into account the blow-up caused by clones, that is, how much larger the artifact is compared to a hypothetical variant that contains no clones and how this affects maintenance costs. For a deeper assessment of the impact of clones onto quality, it is also helpful to understand the concepts underlying clones (Clone Conceptualization), to know the reasons for their existence (Clone Reason and Prevention) and to link them to higher-level software artifacts (Clone Linkage).
Quality assessment often looks not only at the current state of affair but also on trends, hence, tracking the evolution of clones is often wanted, too (Clone Tracking). Because correctness is one important quality that needs to be assessed, Quality Assessment implies Bug Detection and its associated information needs.

Refactoring aims at improving program maintainability by changing the code structure without changing its external behavior. We have already discussed refactoring as a means to remove clones in the context of Clone Removal. For instance, the refactoring Pull-up Method can be used to remove a cloned method in two classes derived from the same super class. Here, we take also the opposite perspective and ask the question how the knowledge about software clones can assist in refactoring. Because refactoring aims at improving maintainability by removing bad smells and clones are one kind of bad smell, refactoring will benefit from the knowledge where clones exist (All-Clone Detection). Generally, however, refactoring is not a global clean-up activity that searches for all kinds of bad smells, but is driven by a primary reason for a change and in the course of making the change, a developer notices bad smells that represent a hurdle for that change [68]. Then refactoring overcomes the hurdle by removing the bad smells. Hence, clone detection is rather driven by demand, and mostly the clones in the scope of the change are of interest. Consequently, Clone Awareness and Clone Class Detection are relevant. Some IDEs, such as Eclipse, already have a feature for their offered refactoring Extract Method that searches for other code segments that are identical to the one selected and that can be replaced by a call to the very same newly generated method, too. If the clones are not identical, Clone Differencing may help to identify the differences, which may need to be abstracted [45] [554]. For a meaningful extraction of statements into a new coherent method, it is also important to understand the concept behind the extracted method (Clone Conceptualization) and to know the reasons for the existence of clones (Clone Reason and Prevention). Analyzing the relations among clones and architectural design decisions (Clone Linkage) may show that the true reason lies in the architecture rather than code refactorings should be applied. Before a refactoring is actually done, assessing the bad smells (here: clones) with respect to their impact on maintainability, their feasibility, benefits, and risks is needed (Clone Assessment and Ranking). Once the refactoring is done, associated documentation may need to be adjusted (N33 from Clone Linkage). It may also be necessary to inform other developers owning the refactored clones (N23).

Quality Improvement aims at improving various aspects of software quality, among which are correctness, understandability, non-redundancy, and more. Quality Improvement needs to first assess the existing quality and then to select and take measures in order to improve it. In the context of cloning, this means to assess clones and their impact on software quality and then possibly to refactor or at least document the clones in order to avoid unwanted inconsistent changes. Thus, all information needs relevant to Quality Assessment, Clone Removal, Clone Compensation, and Clone Awareness are also required for Quality Improvement. The specific needs for Quality Assessment are those required to rank the clones requiring treatment (Clone Assessment and Ranking) and to make an informed decision on whether and how to handle the clones. For training purposes and to improve communication among developers regarding changes, it may be useful to know the ownership of clones (N23). Almost all information needs are required for Quality Improvement due its rather all-embracing nature. The only information needs left out are those related to license issues and proprietary code that must not be made public and clones across programs: we restrict quality improvement to technical aspects of a single program here. In case of quality improvement for variants in a product line, these issues might become relevant, too.

4.6. Merging and compaction

Merging: Sometimes we need to merge two existing similar program variants, for instance, in parallel development branches or to consolidate copy-and-paste software variants in a software product line [69,70]. Merging can be viewed as a combination of establishing a mapping of the corresponding code between program variants followed by a second step that creates a new variant by combining the shared code and integrating the differences. Software clones are here leveraged to establish the relationship between the variant units to be merged (File Mapping; see Section 4.8). This comparison is only between units (All-Clone Detection and Clone Differencing across units) and clones only within a unit are not relevant in merging. Then code is merged by keeping the common parts and integrating the differing parts, analogously to textual merge tools built into version control systems or dedicated tools such as meld.

For this user goal Merging, we exclude advanced merge strategies through more sophisticated refactorings to keep the notion of user goals as coherent as possible. Simple textual merging could be followed by Refactoring to further improve the result, but we have included the information needs for Refactoring in that user goal separately.

Code Compaction can be viewed as another special kind of refactoring with the goal of reducing the number of code elements to optimize storage of a program’s code [71]. This is usually achieved by eliminating unnecessary redundancy, removing irrelevancy, or using special coding techniques and may be applied to the code generated by the compiler or as an intermediate step to prepare the source code for the compilation. The transformed code is considered an intermediate representation not actually maintained and the transformation is merely a compiler optimization. Hence, questions related to costs, risks, and benefits with regard to maintenance are not relevant. Clone detection techniques aid in code compaction by identifying redundancies in the form of clones [71,72]. To achieve this goal, we need to know the location of all clones All-Clone Detection within a program unit to be compacted and their differences (Clone Differencing) as well as how to eliminate them (N30, N31 from Clone Removal). Ranking clones to identify clone groups with maximum benefit in reduction and other contextual information is also relevant (N4, N5, N7, N8, N11, N26, N27 from Clone Assessment and Ranking). This information will be used by an automated algorithm rather than a human, because one would like to completely automate code compaction as much as any other kind of compiler optimization.

4.7. Re-use

Library Candidates: Sometimes, a clone is present across a wide range of systems and should be put into a library. Use of such libraries may reduce development costs and increase the reliability of software systems [73]. For this kind of Library Candidates, one needs to locate all clones and their differences not only within a system but also across multiple systems (All-Clone Detection or Clone Class Detection, Clone Differencing), assess the risks and benefits of moving the code into a library (Clone Assessment and Ranking), and then select and apply appropriate refactorings (Clone Removal). The library code should have a coherent meaning, hence, Clone Conceptualization is required. Because code may be collected from different systems possibly under different licenses, it may also be necessary to know the ownership and license of the copied code, and possibly whether the library code may be made public at all (Clone Ownership).

Aspect Mining: Other times, a clone may represent a cross-cutting concern and can be re-used as an aspect instead of being moved into a library. Aspect-Oriented Programming (AOP)
intends to increase modularity of programs by allowing the separa-
tion of cross-cutting aspects [74]. Information needs relevant to
this kind of user goal are analogous to Library Candidates except
that – assuming the aspect is not encapsulated across different
systems, but only within a single system – clones are not searched
for across programs (N3) and license and ownership issues (Clone
Ownership) are not relevant.

**PL Migration:** Occasionally in the context of domains where
similar, yet not identical requirements must be fulfilled by a set of
programs, entire systems or at least very large parts are cloned
several times, which calls for PL Migration [75–78]. The main
aim of clone detection for product-line migration is to assess the
overall level of similarity among software components [79] and
then to merge similar components where it makes sense. Thus,
all information needs for Library Candidates are relevant for PL
Migration, just at a larger scale. Because we can assume that a
company has developed all the variants to be migrated into the
product line, information needs regarding license and ownership
issues are likely not relevant. Because whole units (e.g., files) may
be merged, it is also interesting to measure the amount of code
copied between two or more variant units (N13, N14).

4.8. Code mapping and evolution

When two or more different versions or variants of a program
are to be compared, their code entities must be mapped onto each
other to make a connection. If the connection relates clones in
different versions, their evolution can be studied. If the connec-
tion relates variant program units of different programs, merging
those variants may be made possible.

**Evolutionary Mapping** establishes a mapping of the clones
of one program version to another version. It is used in clone
evolution studies as well as for carrying over any information
from ancestors to their descendants (e.g., in software-quality
dashboards). The information needs beyond knowing where the
clones are across versions or variants (All-Clone Detection)
and what their differences are (N9 and N10 from Clone Differencing),
is the mapping between historical descendants, which may relate
etities in different versions, variants, or even systems (N3, Clone
Tracking) [69]. In many cases, the mapping is established by
consulting the changes stored in the version control system [80].

**File Mapping** re-establishes the lost change history when it
is not preserved in a version control system [81]. It attempts to
map files from one version to another version in the presence of
possible modifications, renamings, or re-locations of files within
the directory hierarchy if no version control system was used to
preserve such information. It differs from Evolutionary Mapping
in that it maps files by way of clones. The clones themselves,
however, are of secondary interest here. They are only a means
to establish the mapping from files – or more generally, program
units – of one version onto program units of the other version.

The input to File Mapping are two or more program versions
ordered by time. The output is a mapping of program units
between subsequent versions. Clones are detected across two
versions (All-Clone Detection across multiple program versions)
and the most similar clones are selected (Clone Differencing) and
then aggregated in a similarity measure of the program units
containing those clones. The mapping is not trivial because there
is not necessarily a simple one-to-one mapping of clones and
there may be possible modifications to clones that must be taken
into account. Lavrova et al. [91] propose a nearest-neighbor clone
detection to establish this mapping, where only clones between
two versions are considered.

**Clone Evolution** studies how clones are modified from one
version to another and other evolutionary aspects. Evolutionary
Mapping is a prerequisite (hence, inherits all information needs of
Evolutionary Mapping), but Clone Evolution goes beyond just mapp-
ing clones. It attempts to discover the properties and patterns
displayed by clones during their evolution within a system. These
insights are relevant for software-clone research, but may also
have practical relevance in a software-development organization
that seeks a better understanding of the evolution of their clones,
for instance, to predict future trends.

A systematic review of clone evolution is given by Pate et al.
[82]. Lage et al. [55] have performed clone evolution analysis
to examine the usefulness of assimilating a clone detection tool
in the software development life cycle. Evolution of function
clones across multiple versions of a huge system is examined
inspecting how many clones are added, modified and deleted
in each subsequent version (N17, N18, N19). Clones that have
never been modified during evolution are also identified. Antoniol
et al. [83] have proposed a time-series model for monitoring and
forecasting the evolution of clones (N20). In another study, An-
toniol et al. [84] examined the clones of several versions of the
Linux Kernel regarding the location (N21) and stability of clones
(N19). Multiple researchers, for instance, Kim et al. [85], have
explored clone genealogies, which occur when clone classes are
evolved across several versions of a software system. They have
explored how each instance of a clone class is modified with
respect to the other instances over the different releases of a
system (N19). Li et al. [62] have studied the changing cloning
rates during evolution of Linux and FreeBSD systems (N20). The
evolution of clones can be investigated for all clones of a system
(All-Clone Detection) or just for all instances of a given clone class
(Clone Class Detection). The evolution may be studied for virtually
all kinds of properties of clones such as size, number of instances
of a clone class, the differences among clone instances, and many
more aspects of a clone.

4.9. Origin analysis

Origin analysis creates a mapping between ancestors and de-
cendants of entities in source code [86]. It is an integral part of
evolutionary studies [69] and also of all practical applications in
which it is necessary to carry over information from an ancestor
to its descendants (e.g., software quality dashboards for multiple
program versions) or vice versa. In the perspective of origin
analysis of software clones, the entities of interest are clones. The
mapping can relate clones between system versions or between
different systems.

**Provenance Analysis** refers to the origin or source of software
artifacts. Clone detection provides a starting point for provenance
analysis by tracking clones between applications [87]. Knowing
clones can provide clues regarding parts of code that might have
been copied (possibly illegally) from other systems. Provenance
information is helpful in scenarios such as verifying that free
open-source code is not being used by commercial enterprises
because of legal issues and to know the origin of some suspicious
piece of code [88,89]. Provenance Analysis is a special kind of
Evolutionary Mapping, so they share the same information needs
in regards to locating and tracking clones. An additional very im-
portant information need for Provenance Analysis is the ownership
and licenses of the cloned entities (N23 from Clone Ownership).

**Due Diligence:** If the provenance turns out to be a program
with a license incompatible to the context where the copy resides,
we may have a case of a potential copyright infringement. If the
submitter of a code claims to be the author but the provenance
of the code is actually a different source, we may have a case of
plagiarism. Companies and teachers are committed to exercise
due diligence to detect such cases.

Clone detection can be used for Due Diligence to detect and
measure the code similarity between suspected software systems.
At a more detailed level, clone detection can also report which part of the suspected system code has been copied from which part of the original system [90] [P32]. Detection of plagiarism and copyright infringement are a special case of Provenance Analysis limited to cases where the additional information need is to know the true source of a piece of code and whether the claimed authorship is wrong or whether copyrights are infringed, that is, whether code is copied from a source with incompatible ownership rights (N24 from Clone Ownership).

**Code Leakage** describes the situation where code that should be kept private is copied to a forbidden place, for instance, from an internal repository to a publicly accessible server location. It is a special case of Provenance Analysis where the focus is on the offspring rather than the origin. Here the additional information need is whether the offspring is located in forbidden places [41]. Code Leakage must find those instances of a clone class (Clone Class Detection) across units (N3) that should have been kept private but appear in public code (N25 from Clone Ownership).

### 4.10. Malicious Software Detection

**Malicious Software Detection:** Matching and relating malware by their common parts through clone detection is helpful for identifying fingerprints that can be used to identify programs affected by malicious software [91]. Similarly, self-mutating malware – a specific type of code obfuscation – may sometimes also be identified with clone detection when there is a relatively stable code snippet embedded in the self-mutating code [92]. Relevant information needs include those of All-Clone Detection (for identifying significant fingerprints) and Clone Class Detection (for searching for specific fingerprints) across units and Clone Differencing.

### 4.11. Productivity

**Productivity Measurement:** Cost-estimation models based on lines of code such as COCOMO and productivity measures must be normalized in the presence of clones to become meaningful. Management, hence, needs to know the degree of redundancy (N7) compared to a clone-free version of the program (N27) in order to have unbiased information for making management decisions derived from program size.

### 4.12. Summary

In this section, we have described user goals related to software clones derived from the scientific literature. For each user goal, we identified the information needs. Fig. 1 summarizes which user goal has which information needs as a Dotplot. The information needs are listed in the rows and the user goals in the columns. A colored cell in this matrix means that the user goal in the column requires the information listed as row. The necessity for an information need were explained in the description of the user goals above.

It makes sense to investigate similarities among user goals in terms of overlapping information needs. If user goals share similar information needs, a set of visualizations may be identified providing that kind of information not only for a single, but multiple similar user goals. In order to gather similarity among user goals, we applied hierarchical agglomerative clustering (HAC) to the set of user goals. HAC starts by treating each object as a singleton cluster initially and then merges the two most similar clusters successively until all clusters have been merged into one. The result is a tree-based representation of the objects to be clustered, named dendrogram, which shows the degree of similarity at which clusters were merged by the height of inner nodes, which represent the merge of two or more clusters. The dendrogram for the user goals is drawn above the matrix in Fig. 1.

There are two different kinds of similarities that defines a HAC: (1) similarity between two objects and (2) similarity between two clusters of objects. Many different variants of HAC can be created by choosing those two similarity functions in different combinations. In our application of HAC, we want to cluster user goals with respect to their overlap of information needs, hence, for (1), we use Jaccard’s coefficient. Let \( I(g) \) be a function that yields the set of information needs for a given user goal \( g \). Then Jaccard’s coefficient for two user goals \( g_1 \) and \( g_2 \) is defined as \( \text{sim}(g_1, g_2) = \frac{|I(g_1) \cap I(g_2)|}{|I(g_1) \cup I(g_2)|} \). Because we want clusters in which all elements are highly coupled to each other, we use the unweighted average linkage for (2). Let \( G_1 \) and \( G_2 \) be two clusters of goals to be merged, then the new similarity of the merged cluster \( G_1 \cup G_2 \) to every other cluster \( G \) is defined as...
follows:
\[
\frac{1}{|G_1 \cup G_2| - |G_1|} \sum_{g \in G_1} \sum_{g' \in G_2} \text{sim}(g, g')
\]

Analogously, we cluster also the information needs with respect to the set of user goals they may support using again Jaccard’s coefficient and unweighted average linkage. The resulting dendrogram is shown left from the matrix. The rows and columns are ordered so that more similar information needs and user goals, respectively, are closer together. There are additional heatmaps above and left from the matrix that we will explain later in Section 7.

The dendrogram at the top of Fig. 1 shows two quite distinct user goals, namely, Productivity Measurement and Code Leakage. The former requires only N7 and N26, which are two information needs found in many other user goals, too, hence are subsumed by these. The reason why Productivity Measurement is clustered very late in the dendrogram is the fact these two information needs are the only ones for this user goal and, thus, the symmetric measure for overlap used in Jaccard’s coefficient is small. That means, every set of visualizations providing the information needs N7 and N26 (there are seven other user goals with those needs) will also serve Productivity Measurement, but this set would be unnecessarily large for just Productivity Measurement. Code Leakage is similar in this respect: its two information needs N2 and N3 are required for six other user goals; yet its information need N25 is quite unique. The only other user goal with this information need is Library Candidates, which also requires N2 and N3. Thus, a set of visualization supporting Library Candidates would also support Code Leakage but would be again unnecessarily large.

A larger cluster of similar user goals is formed by Quality Assessment, Quality Improvement, Code Compaction, Refactoring, Aspect Mining, Library Candidates, Pl. Migration where Quality Assessment and Quality Improvement are more similar to each other and somewhat more distinct from the rest of this cluster. They all subsume clone-management activities in which the impact of clones on software maintenance is assessed and refactoring is considered to remove clones. Because so many factors need to be weighed for these goals, all of these constitute major software-engineering tasks requiring a lot of diverse and detailed information. As a consequence on software visualization, we can expect that many diverse kinds of visualization will be required to support these goals and integration of these will be a major challenge.

The next cluster of four goals consists of Evolutionary Mapping, Clone Evolution, Due Diligence, and Provenance Analysis whose specific common core of information needs deal with tracking clones between different versions of the same program or between different programs and their differences. Here, visualizations are needed that focus on the evolution of clones. Evolutionary Mapping and Clone Evolution are actually indistinguishable in terms of their information need because Evolutionary Mapping is a prerequisite for Clone Evolution and Clone Evolution simply analyzes its data.

Not surprisingly, Bug Detection and Duplicated Bugs are grouped together as they both deal with defects. Yet, as also shown by the dendrogram, they are more distinct from each other than one might have thought. The reason for that is that Bug Detection searches for clones that potentially contains defects, while Duplicated Bugs looks out for code similar to a given piece of code already known to be defective—basically searching other instances of a clone class.

Of course, testing and evolving programs are quite different activities, yet Evolution and Testing have identical information needs—with respect to what clone management can offer to them. Because test code must be evolved as productive code, all information needs of Evolution are included for Testing. Creating similar test cases for similar production code is just an instance of clone detection already included in Evolution; hence their needs are identical in regard to cloning.

Merging and File Mapping have identical information needs, too. Both require a mapping between corresponding code units of two or more program variants; that is why they appear as identical with respect to the information needs considered for the dendrogram.

Domain Analysis and Design Recovery are almost alike. The extra information need N3 of Domain Analysis is justified by the fact that Domain Analysis often looks at many programs and their clones to obtain a broad overview of a domain, while Design Recovery is generally limited to the design of one particular program. Both of them share four information needs with Merging and File Mapping, namely, N1, N12, N9, and N10, which call for the locations of clones and their similarities and differences, which represent a common core for most user goals. The reason why Domain Analysis, Design Recovery, Merging and File Mapping form a cluster is that the former two have only two extra information needs not required by the latter two: N32 and N34 which are both more advanced needs looking into the relation among clones and their underlying concepts. Domain Analysis and Design Recovery search for more semantic information on clones.

Malicious Software Detection has relatively few information needs and none of these is specific to it. The level of support clone analysis can offer for this user goal just is to search for shared code fingerprints among programs known to have malware and then to search for those fingerprints in other programs that may be affected by this malware. For this reason, it is clustered very late in the dendrogram.

Program-Concept Assignment and Feature Location are identical in respect to their information needs in the context of software cloning. There are of course conceptual difference in those two user goals as Program-Concept Assignment starts at a given piece of code and tries to figure out the feature it implements, while Feature Location starts with a feature and looks for the code that implements it. Yet, the kind of information that can be gathered by clone analysis for both goals is the same.

The relevance of this clustering is that the same set of visualizations can be very likely re-used for identical or at least very similar user goals in the dendrogram. There are of course conceptual difference in those two user goals as Program-Concept Assignment starts at a given piece of code and tries to figure out the feature it implements, while Feature Location starts with a feature and looks for the code that implements it. Yet, the kind of information that can be gathered by clone analysis for both goals is the same.

The relevance of this clustering is that the same set of visualizations can be very likely re-used for identical or at least very similar user goals in the dendrogram. Before that, however, we need to introduce the visualizations.

5. Facets of clone visualization

Before we describe the clone visualizations, we will first lay out our frame of reference of their description. This section discusses types of clone data collected and describes how we organize the description of the visualizations.

5.1. Clone data

Before we describe the different kinds of visualizations, we will first elaborate on the nature of clone data. This discourse will help us to categorize the different kinds of visualizations later on. The elaboration in this section is actually largely independent from cloning and applies to many different kinds of data in general.

Given a collection of items, let I represent aspects of cloning. The items may be heterogeneous, for instance, source-code lines,
files, classes, packages, or directories—any concept related to cloning. An item may have characteristics that can be measured. A measurement is the assignment of a number to a characteristic of an item, which can be compared with other items. Mathematically, a measurement is a function \( f \) whose domain is \( I \) and whose range is a potentially infinite set of numbers (denoted by \( \mathbb{R} \) in the following). For instance, clone length may be a function that yields the number of tokens for a cloned code fragment.

A measurement may have one of the following scales: nominal, ordinal, interval, or ratio scale describing the nature of information within the values of the measurement, thus, \( \mathbb{R} \). Without knowing the scale of the data, they cannot be interpreted. These scales form a subsumption hierarchy, that is, every ratio scale is also an interval scale, every interval scale is an ordinal scale, and every ordinal scale is also a nominal scale. Interval and ratio scales are often summarized as quantitative scales.

On a nominal scale, the values of the measurement in \( \mathbb{R} \) are simply labels that allow to form categories of equal items. They can only be used to distinguish or lump together items. Mathematically, for two items \( i_1, i_2 \in I \), we can answer the two questions \( f(i_1) = f(i_2) \) or \( f(i_1) \neq f(i_2) \). For instance, \( f \) could be a function that identifies the file that contains a cloned code fragment by a unique number. The number is just a name for the file. Then we could determine whether two cloned fragments are in the same file or in different files.

Ordinal scales have a total order relation \( \leq \) by which items can be sorted \( f(i_1) \leq f(i_2) \). For instance, \( f \) could determine the type of a clone (one of type 1, 2, 3, or 4), and we could determine whether one clone's type is subsumed by another clone's type.

On interval scales, the degree of difference between items can be expressed. While on ordinal scales, we can say that for two items \( f(i_1) \leq f(i_2) \) holds, on interval scale we can even determine by how much they differ; that is, we can subtract them \( f(i_2) - f(i_1) \). For example, if \( f \) measures the length of a cloned fragment in terms of lines of code, we can determine by how many lines one cloned fragment is longer than another cloned fragment.

A ratio scale has a meaningful (unique and non-arbitrary) zero value that allows one to measure the ratio between a magnitude of a continuous quantity and a unit magnitude of the same kind. For a measure \( f \) on the ratio scale, we can meaningfully compute \( f(i_1)/f(i_2) \), thus, measure the percentage by which they differ. For instance, to compute the clone rate of a file, we need to measure the fraction of cloned tokens in a file over the total number of tokens of that file. We can do that because the meaningful zero value of the number of tokens is the length of the empty file, which has no tokens at all. There cannot be any file shorter than that.

Beyond measurable characteristics (functions yielding numbers), each item of \( I \) may also have relations to other items in \( I \). Mathematically, an \( n \)-ary relation over \( I \) is any subset of the Cartesian product of \( I \). Very often, we have binary relations \( I \times I \) \((n = 2)\). The fact that two fragments are clones can be expressed as such a binary relation. A relation itself may also have characteristics that can be measured, and those measurements can be any of the above scales. For instance, if there are two fragments that are mutually clones of each other, we could annotate their clone relation by the clone type on the ordinal scale type 1, 2, 3, or 4. For all clone types above type 1 — that is, clones that are not identical (also known as near-miss clones) —, we could measure the difference in terms of the number of tokens that are contained in only one of the two cloned fragments.

Binary relations can be further refined into tree structures, where each item has exactly one direct descendant except for root items that do not have any parent (and at least one such root must exist), acyclic graphs, where the relation forms a graph that has no loops, and general cyclic graphs. The edges of such graphs may be directed or undirected. The nesting of cloned code fragments into files and these into directories forms a tree, for example. The relation that describes the strict containment or overlap between cloned code fragments forms an acyclic graph because a shorter cloned fragment may be completely or partially be contained in multiple longer overlapping cloned fragments. An example for an undirected relation that forms a cyclic graph is cloning between files because a file can share code with any other file.

The relevance of all these kinds of information structures is that visualizations must match the information structure of the data. For instance, classical bar charts can be used to visualize clone length because length is at an interval scale. Bars of different length are less suited for nominal data because their length invites a human observer to make visual length comparisons although that is meaningless for nominal data.

When we describe the clone visualizations, we will present in greater detail and more concretely what kind of measurements and relations of cloning are generally visualized. Before that we will introduce a classification by which we describe and group the kinds of clone visualization we identified.

### 5.2. Taxonomies of visualization

To describe the found types of clone visualization more coherently, we will group related visualizations together in a classification—or taxonomy. This classification also allows us to identify similarity among visualizations. Different researchers in information visualization in general as well as in software visualization in particular have already introduced such taxonomies, e.g., [93–104]. Some are based on the data shown, some on the tasks to be fulfilled with them, and some on inherent visual properties of the visualizations. All of these are important aspects calling for a multi-faceted taxonomy. We will classify the types of visualization we found for cloning in all these aspects. Similarly to Maletic et al. [101], we will relate the visualizations to the user goals, that is, the primary tasks their beholders intend to solve with them, by first listing the information needs the visualizations support and then by relating user goals to information needs. Moreover, we also follow Card and Mackinlay [98] who base their taxonomy on the other two major aspects: the structure of the data as outlined in the previous section and the aspects of human vision—making a distinction between what is visualized and how it is visualized and perceived by humans. We will use the terms data facets for the former and human-vision facets for the latter aspect.

Concerning the structure of the data — the data facets — their taxonomy considers the nature of measurements and relations discussed in Section 5.1. Regarding visual aspects, Card and Mackinlay [98] base their taxonomy on how human process visual information. According to Shiffrin and Schneider [105], human visual processing involves two levels: automatic and controlled processing. Automatic processing works on visual properties such as position and color and some of these properties are perceived pre-attentively, thus, before they arrive in the conscious mind. Automatic processing is highly parallel, but limited in expressive power. Controlled processing, on the other hand, works on abstract encodings such as text and generally requires the conscious mind. It has powerful operations, but is limited in capacity.

Automatic processing is further refined into mark types, relational properties, and positions in space and time. Mackinlay [106] argues that visualizations are basically made from marks and their graphical properties. There is a limited set of such marks and properties as identified by Bertin [107], expanded by Mackinlay [106], and further expanded by Card and Mackinlay [98]: point,
Fig. 2. Classification of the types of visualization in terms of human-vision facets.

- **Line, surface, area, and volume.** Retinal properties comprise color, size, connection, and enclosure. Positions in space and time span a four-dimensional space where each position is identified by its x, y, z co-ordinates and another co-ordinate t for time. The latter is used by animated visualizations. These human-vision facets are summarized in Fig. 2.

The combination of all these aspects introduces a taxonomy where each visualization can be characterized by a multitude of data and human-vision facets. A simple scatter plot to compare the relation between the size of files and their degree of cloning among two different systems, for instance, represents two variables both at interval data where a point is used as marker, color is used as a retinal property to distinguish the data for the two different systems, and the x- and y-positions are used to present the file size and clone rate, respectively. A tree map showing the directory structure, the size of code files and their clone rate, as another example, is a visualization where the containment relation (tree relation) of files and directories is shown as nested areas (marker: area, retinal property: enclosure), file size (a quantitative scale) is shown as the inner-most rectangles whose area is proportional to the size of the file (marker: area, retinal property: size), and the clone rate as a color range of the file area (retinal property: color range). The x and y co-ordinates are chosen by an algorithm in such a way as to produce a compact nesting of the rectangles that makes use of all available displayed space; otherwise they are meaningless. Below where we present the clone visualization we found, we will discuss the found visualizations along these visual facets.

Whereas Card and Mackinlay [98] provide a very refined taxonomy, where each visualization is characterized by multiple facets, Shneiderman [95] provides a simpler categorization into seven distinct types of visualization and seven tasks to be fulfilled by the visualizations. The tasks are rather abstract and generic and should not be confused with our notion of goals and information needs introduced in Section 4. The latter are specific to the domain of cloning and more refined. The tasks in the taxonomy by Shneiderman [95] are as follows:

- **Overview:** gain an overview of the entire collection
- **Zoom:** zoom in on items of interest
- **Filter:** filter out uninteresting items
- **Details-on-demand:** select an item or group and get details when needed
- **Relate:** view relationships among items
- **History:** keep a history of actions to support undo, replay, and progressive refinement
- **Extract:** allow extraction of sub-collections and of query parameters

The seven types of visualizations are related to the data type presented by the visualization, similar to our distinction discussed in Section 5.1. They are:

- **One-dimensional data:** the collection of items is organized in a linear/sequential manner. An item may have multiple attributes nevertheless, that is, one-dimensional should not be confused with one-factorial where only one variable is measured. The linear order rather relates to the items themselves not necessarily to a single attribute of these. A spatial relation exists for one-dimensional data, where one can express left and right for items.
- **Two-dimensional data:** planar or map data items. Each item in the collection covers some part of the total area and may be rectangular or not. Again, each item may have multiple attributes nevertheless. Beyond the spatial relations of one-dimensional, we also have above and below.
- **Three-dimensional data:** items with volume spatial relations: left/right, above/below, and in front/behind.
- **Multi-dimensional:** items are organized in an n-dimensional space, with n > 3.
- **Temporal data:** the items are organized along a time line. The distinction to one-dimensional data is that items may have a start and finish time and that items may overlap in time, that is, co-exist.
• **Tree**: hierarchies or tree structures are collections of items with each item having a link to one parent item (except the root). Items and links can have multiple attributes. The spatial relation inside/outside exists.

• **Network**: collections of items that have links to an arbitrary number of items. The relation represented is connected-to.

Note that this taxonomy relates to the information structure of the represented data rather than the actual way of drawing the data. For instance, bar charts are drawn in a two-dimensional space, but the underlying information content is one-dimensional expressed by the length of a bar, while the width has no meaning in the visualization. Yet, quite naturally the information structure suggests a particular type of visualization suited for this kind of structure. A tree data structure, for instance, would hardly be visualized in only one dimension, but rather by way of node-link trees, tree maps, circle packing, sunburst, or any other kind of hierarchical visualization.

Fig. 3 shows our classification regarding data facets based on the ideas by Card and Mackinlay [98] as well as Shneiderman [95] that we adopt here to group our clone visualizations. The various n-dimensional categories by Shneiderman roughly correspond to our notion of measurements of items (attribute measures) introduced in Section 5.1, while Shneiderman’s tree class is equal to our corresponding tree relation among items. His notion of network corresponds to our more refined cyclic or acyclic graphs, respectively. Moreover, we added another subcategory bipartite graph of relational data as this was necessary for classifying the **Clone-Class Family Enumeration** described in Section 6.4.1 properly. A bipartite graph is a graph whose nodes can be divided into two disjoint and independent sets V1 and V2 such that for every edge (s, t) the following holds: source s ∈ V1 and target t ∈ V2. Bipartite graphs model assignments between objects of two different kinds.

Shneiderman highlights one important one-dimensional data type: text (also mentioned in the taxonomy by Card and Mackinlay [98] as information requiring controlled cognitive processing). Copied source-code fragments are obviously text,3 hence, textual representations are of primary concern in our domain. For this reason, textual representations deserve an explicit category in our taxonomy, too. We note, however, that we do not consider purely textual representations as visualization in the following, except when enriched by additional visual attributes such as colors. Similarly, we do not consider tables of texts such as tabular views. They are popular and useful means to present, for instance, metrics, and allow one to filter and sort data quickly. Yet, they are just a flexible organization of textual data. They offer very little abstraction for large amounts of data and do not support “visual analytics” well.

Other examples of one-dimensional data are single-factorial quantitative scales, that is, metrics like the amount of copied code. When more than one kind of metric is gathered (such as number of copied code, number of differences, and degree of inconsistency of the differences), we have two- or three- or multi-dimensional data. We group such information content in category attribute measure and will state their dimensionality later in Section 6, where we describe every visualization in detail. Clone analysis also looks into the evolution of clones, that is where temporal data or times series come into play. We adopt Shneiderman’s category of temporal data and group visualizations focusing on evolutionary aspects into category temporal in our taxonomy.

The leaves in Fig. 3 are the clone visualizations we will discuss in Section 6. We point out that there is not always a strict mapping of these onto a single taxonomic entry, because a visualization may show different attributes at the same time. For instance, polymeric views may show the nesting as a tree structure and the size of a clone by the area of a rectangle. For such visualizations, we use the main kind of data structure presented that is the most prominent visualized aspect.

6. **Clone-visualization techniques**

Now that we have explained what kind of facets we will use to describe the clone-visualization techniques we identified in our literature survey, we describe these in detail addressing our research questions RQ2 and partly RQ3. Our classification is based on multiple orthogonal facets, hence, no single strict order exist to present them. We will group them based on data facets as summarized in Fig. 3 here, because a visualization of a particular data structure can take advantage of many different human-vision facets and answer multiple information needs, yet the data have generally one primary structure to be visualized most prominently. In the description of each visualization, however, we will also describe their human-vision facets and explain which information needs they satisfy. In Section 7.2, we will group the visualizations also in terms of the information needs they address.

The description of each visualization follows a uniform structure in the remainder of this section. First it will be introduced in general and its human-vision facets will be explained, then we describe its use in clone visualization and which information needs it addresses, and finally we conclude each description with an assessment of its inherent strengths and weaknesses.

6.1. **Visualization of attribute measures**

In this section, we list visualizations for n-dimensional attribute measures related to cloned items. We will start off with classical data visualizations, namely, **Pie** and **Bar Charts** for one-dimensional data, **Line Charts** for two-dimensional data, and **Parallel-Coordinate Charts** for higher-dimensional data, before we come to more advanced types of clone-data visualization in the next section. Because of lack of space and the fact that several types of charts for n-dimensional data discussed in this section are well known, we will omit screenshots of those.

6.1.1. **Pie Chart (pc)**

A **Pie Chart** is a circle partitioned into typically distinctly colored segments reflecting the proportion of some variables or categories in relation to the whole. The area is a marker for the proportion of distinct categories and color or texture are used as a retinal encoding of the categories. **Pie Charts** are normally drawn in 2D, hence, the z- and t-position is not used. Sometimes they are also shown in 3D but then the z-dimension has no meaning and just makes comparing the areas more difficult. The x and y co-ordinates may be chosen to lump together segments of related categories in proximity (for instance, data on clones of type 1, 2, 3, and 4 in that order). The order of segments may also follow the size of the segments. Other than that, the x and y co-ordinates have no meaning generally.

**VisCad**, for instance, uses **Pie Charts** to relate the clone rates in different subsystems [93]. The Clone Visualizer by Zhang et al. [162] visualizes the proportions of files in a directory together forming so called **file-clone classes** (FCC). An FCC is a group of highly similar files [23], that is, files sharing many recurring patterns of cloned fragments. In general, **Pie Charts** can be used to show the proportion of cloned code in relation to a whole file or directory, the proportion of cloning in a given entity’s parts,
or the distribution of clones for the different types of clones (N6, N7, N13, N14).

A Pie Chart allows a quick visual assessment of proportions, for instance, of all kinds of clone aggregates such as clone pairs, clone classes, and super clones at various levels of granularity such as subsystems, files, and code segments and clone types (1–4). However, Pie Charts with too many segments are difficult to understand. Similarly, if the values represented by the segments are very similar, the Pie Chart will be difficult to decipher because the segments will be very similar in size, too. Generally, it is difficult to visually compare different segments of a given Pie Chart or to compare segments across different Pie Charts, because although humans are good at comparing lengths, they have difficulties in comparing areas or angles [109]. Spence [110] provides a summary of the empirical evidence for and against Pie Charts. Several variants have been proposed such as Doughnut Charts and Exploded Pie Charts [111].

6.1.2. Bar Chart (bc)

A Bar Chart consists of rectangular bars (marker: line) representing an attribute measure at interval scale, where the length of each bar is proportional to the attribute measure. The measure can be shown by either the x or y co-ordinate as a retinal property depending upon whether the bars are horizontally or vertically oriented. The respective other co-ordinate may be used to group together related bars or provide the order of a secondary attribute measure at ordinal scale or simply reflect the order of the bars in terms of their length. The retinal encodings color or texture can be used to represent different categories (ordinal data) on a Bar Chart. A bar itself can be further segmented with various colors to show the relative sizes of different sub-categories of the bar’s category, yielding Stacked Bar Charts. If the attribute measure shown is a proportion, a Bar Chart may represent the same data as a Pie Chart.

Horizontal Bar Charts are used, for instance, in VisCad to show the number of clones, lines of code, lines of cloned code, and other kinds of metrics for each file [P3]. Vertical Stacked Bar Charts are used in Zhang et al.’s Clone Visualizer, where the length of the bar shows the size of the file and the length of the colored segments within a file shows the sizes of the different clones in that file at the level of File Clone Classes (FCC) [P62]. In clone visualization in general, Bar Charts can be used to compare all kinds of single-factorial quantitative attribute measures of clones (both relative and absolute values) [N6, N7, N13, N14]. Stacked Bar Charts could also relate the amount of code cloned in relation to the size of the code unit containing those clones (N15).

A Bar Chart is effective to compare values and see correlations among metrics in a series more quickly than by scanning values in a table of numbers. As the number of bars grows, it is difficult to present an overall picture easily readable. Scroll bars may then be needed, which limit the field of view. Pie Charts instead manage with a given fixed space—although their segments may become too small to be readable, if there are too many.

6.1.3. Line Chart (lc)

A Line Chart displays the relationship between attribute measures of two variables at interval scale plotted along x-axis and y-axis of a grid. The actual marker in a Line Chart is the point for each pair of values of the two variables. Yet, Line Charts connect such subsequent points by straight lines, hence the marker becomes a line suggesting a linear relationship in between two neighboring truly observed data points, which may or may not be true. If the chart shows the relationship of two variables for more than one entity, the line connections at least help to distinguish the entities’ data points from each other more clearly. To further support this distinction, color and shape of the data points and lines may be used as retinal encodings. If one of the two variables is actually representing time, a Line Chart can help to see a trend in data over intervals of time.

A Line Chart is used by Cyclone to show the lifetime of clone genealogies, that is, how many clone genealogies live for a given number of versions, where the x-axis denotes the number of versions – the lifetime – and the y-axis shows the number of genealogies, whose lifetime equals the corresponding x-value. Another Line Chart shows the change frequency, that is, how many genealogies changed a given number of times, where the x-axis denotes the number of changes and the y-axis denotes the number of genealogies that changed exactly this often. This visualization gives a beholder a characteristic line for one single code unit and if multiple code units are combined into one chart, their characteristic lines can be compared to each other, enabling cross-unit comparisons. Likewise, rather than combining different code units, one could as well plot characteristic lines for the same code unit, but at different points in time to look for trends.

All kinds of quantified attributes can be related to each other in a Line Chart (N4, N5, N6, N7, N13, N14, N15). A Line Chart is more expressive than a Bar Chart (Section 6.1.2) because multiple data series with two variables at interval scale can be plotted at a time along two axes, whereas Bar Charts show only different values of the same kind of variables for different entities. If multiple entities are to be visualized, a Line Chart can use multiple lines. It also better shows the trends in data over different time intervals (see Section 6.3.1). A general limitation of Line Charts is that only two variables can be compared at a time. In addition, the lines used to connect measured points suggest a linear interpolation for missing data in between, which may create a misleading illusion.

6.1.4. Parallel coordinates view (pcv)

While Pie and Bar Charts visualize an attribute measure of a single aspect of a collection of items I and Line Charts of two aspects, Parallel Coordinate Views can be used to analyze n-dimensional data with n > 2. The n-dimensional space is drawn by n parallel and evenly spaced vertical axes—one axis for each variable. Each item of I has a dot as a marker on each axis corresponding to the measured value of that aspect. Similarly to Line Charts, a polyline connects the dots on different axes belonging to the same item. One could similarly to Line Charts, each polyline can be colored differently or drawn by different textures (e.g., drawn through, dotted, or dashed) as a retinal encoding of the item to further highlight which data points belong to the same item.

A Parallel Coordinate View is used in CCFinder/Gemini, where it is called the Metric Graph View [P57] [P58], to present the values of various clone-class metrics (N4, N5, N2, N7). Here the different metrics are the axes and the clone classes are the items. Higo et al. [P24] use an extended version of CCFinder/Gemini that presents six metrics in a Parallel Coordinate View that are to help developers in the decision whether refactoring clones is worthwhile (N31). One could also use Parallel Coordinate Views to compare program units based on different metrics (N6, N7, N15).

The Clone-Evolution View by Mondal et al. [P40] [P41] adapts a Parallel Coordinate View to show the number of changes to a clone class over multiple revisions (N19). Each x-monotone line in the view corresponds to a clone genealogy, that is, a sequence of corresponding clone classes across revisions, and the parallel vertical lines represent revisions. The height of a clone class at each revision axis corresponds to the number of changes (added, deleted, and modified lines) the instances of that clone class experienced from one revision to the next one.

The Parallel Coordinate View represents high-dimensional data as a two-dimensional visualization. It works well for data sets
with a moderate number of dimensions and items. When the number of data items increases beyond a certain limit, it becomes difficult to see anything in the chart because of over-plotting. This issue may be sometimes addressed to a certain degree by changing the order of the axes so as to avoid line crossings. If one has $n$ axes, however, $n!$ possible permutations are possible. Hence, it may be difficult to compute the optimal order with the minimum number of crossings. There may also be semantic aspects attached to the order. One would often like to see related or correlated metrics next to each other [112,113].

6.1.5. Relative Similarity Plot View (rsp)

Within a clone class of type 2 or 3, some clone instances can be more similar to each other than others according to a given similarity measure such as Levenshtein’s edit distance. Asaduzzaman et al. have introduced the Relative Similarity Plot in VisCad [P3] to visualize similarity among instances of a clone class. The Relative Similarity Plot—cf. Fig. 4—graphically presents the similarity of clone instances relative to a given instance called the leading node (N10). This leading node (visual marker: point) is determined automatically for every clone class using some similarity criteria (e.g., the centroid). The leading node is drawn at one end of the plot in red as a retinal encoding, while other nodes – shown in blue – are drawn at a vertical distance from the leading node. This distance is computed as the inverse of the given similarity measure. The $y$-position denotes the distance, while the $x$-position is chosen so as to not overlap clone instances with the same dissimilarity to the leading node and has no further meaning.

The Relative Similarity Plot can be used for inspecting similarity of similar, yet not identical cloned entities such as code fragments, methods, files, and directories. It is intended to show only the instances of one clone class. It could as well be extended to all clones (both clone instances as well as clone classes), yet that would quickly raise scalability issues. Moreover, it abstracts from the exact differences. For the latter, a more detailed Code Inspection View could be used (see Section 6.2.5). Hence, the Relative Similarity Plot finds its use primarily in between very detailed and very high-level views of similarity. Another issue may be that the visualization is relative to a selected node; if that node changes, extra effort would be needed to maintain a viewer’s mental map.

6.1.6. Heatmaps (hm)

A Heatmap is a diagram that uses colors or a color gradient as retinal encoding of the values of an attribute that is at least on an ordinal scale. Its roots go back to the 19th century [114]. It is often used in cartography and geography to show a variable of interest (e.g., temperature) on a two-dimensional geographic map.

A Heatmap in clone visualization is used, for instance, in the Near Duplicate Browser [P37,P38] as exemplified in the middle part of Fig. 5. The Heatmap here is a minimap of the whole text, where a color gradient is used to encode the number of copies of text segments.

The Heatmap gives a quick overview of where the clone instances are located (N1) by the $x$-position, how long they are (N5) by the area, and how often they are copied (N4) by the color gradient. It also gives a rough impression of the extent of cloning (N6).

Clones can be nested, that is, a larger clone $A$ may contain several smaller clones. For instance, $A = \alpha BC$, where $\alpha$ is non-cloned code and the subsumed clones $B$ and $C$ may also occur elsewhere outside of $A$ on their own. Then the frequencies of $A$, $B$, and $C$ – and consequentially the color gradients for these – may differ. Suppose $C$ would have the highest frequency, then $A$ would appear as three stripes with increasing color gradient. A user could not visually differentiate that situation from three subsequent non-overlapping clones with different frequencies in a Heatmap. Thus, subsuming clones cannot easily be spotted in Heatmaps. Moreover, one cannot immediately see which clone instances in the Heatmap belong to the same clone class. Other kinds of visualization specific to relational data are better suited for this purpose.

6.2. Textual visualizations

Source code is text and many modeling languages have a textual notation, too. When it comes to the exact details on what is cloned and what changes have been made, it is important to show the copied text. Visualization may help in showing the characteristics, similarities, and differences at that level of detail. Hence, textual visualization is very important. In this section, we describe types of visualization related to textual data. These may be simply decorations such as Bar Maps or more involved visualizations of similarity such as the Triangular Matrix View. Text itself is a controlled processing graphical features, its retinal encodings (e.g., colors) and positioning, however, are automatically processed graphical features.
6.2.1. Bar Map (bm)

A Bar Map is a decoration of text regions in a textual editor or view (see Fig. 6 for an example). The Bar Map is located left or right of the shown text and represents the complete content of the editor. Interesting parts are marked by a line. In clone detection, the lines show where the clones are located in a file. Enclosure of that line is used as a retinal encoding of the relative position and the length (size) of a bar is a retinal encoding of the proportional size of a clone instance. Color may be used as an additional retinal encoding to show other information such as the clone class an instance belongs to. A Bar Map is a kind of minimap of the whole file content (the y-position gives the position of the code marked by the bar relative to the complete file), thus, gives a quick overview and also allows a developer to quickly navigate to the interesting parts by clicking on the bar. A Bar Map is only a decoration of a textual editor or view (e.g., the Code Inspection View with bars in Section 6.2.5), thus, part of a more general visualization. Yet, as it has multiple variant visual features of its own, we describe it as a separate visualization here.

A Bar Map may present relative positions (N1) and sizes (N5) of certain important sections in code views related to clones. It can also be used to establish the connection between a selected clone instance and all other instances of the clone class the selected clone instance belongs to either within or across files (N2, N3). Through further interaction (e.g., by clicking on the bars, a user may navigate to other instances of a clone class) it may, thus, support consistent updates (N36)—although only for the clones within the same file. Additional information on the clone class (N7, N4) can be overlaid possibly by tool tips. Frequently, Bar Maps also show the currently visible part of the file (cf. the red box in Fig. 6) similarly to scroll bars. As the Bar Map is always scaled to the height of the code editor and the currently visible part is proportionally highlighted in the Bar Map, it can also give a rough impression of the size of the file containing a given clone (N15) and how much has been copied from a file (N6). Because, it marks the code that has been copied, it gives also the plain details of what syntactic constructs are contained in a clone (N8). For clone visualization, a Bar Map marks cloned code fragments in the source files and can raise the developers’ awareness of clones (N16). The color of the bars can be used to encode a ranking of clones (N11). For instance, Forbes et al.’s tool Doppel-Code uses a color gradient for Bar Maps showing a so-called impact factor of clones [P17], which is a metric combining the degree of similarity among clone instances of the same clone class along with the number of instances and other weighting factors.

Bar Maps are very popular in clone-visualization tools and most of them use different colors and other visual encodings to provide additional information. For instance, Atomiq represents the repetitions in the same file with red bars, whereas repetitions across multiple files are represented with blue bars. Similarly, Gemini—a visualizer of the clones detected by CCFinder—presents clone instances within a file with yellow vertical bars at the left side of the white bar, while gray vertical bars at the right side of the white bar indicate clone instances across the files. Con-QAT/Teamscale [U15] draws bars in either orange or red to achieve a similar purpose. Orange bars indicate that all clone instances of a clone class reside in the same file, whereas red bars specify that at least one clone instance of the clone class can be found in another file. CloneBoard [P13] highlights clone instances by a thin gray bar. It also warns developers about inconsistently modified clones by showing red bars and highlights clone instances on which a cursor is currently placed with blue bars to help developers recognize the extent of the clone instance they are currently editing. XIAO [P11] defines categories of statements known as block types, which are highlighted with different colors, and numbers are used for block correspondences. CReN [P25] uses differently colored Bar Maps for original code segments and cloned code segments. The tool by Yujiang et al. [P61] marks only suggestions about similar code elements that may require similar updates by using Bar Maps rather than all clones that exist.

A Bar Map helps in tracking clone pairs and clone classes in a Code Inspection View at the level of source code. A lot of information can be shown by Bar Maps either directly or by additional interaction, which may explain its popularity. The level of information, however, is rather fine-grained, always anchored at detailed source code. Hence, it can only complement but not replace higher-level visualizations. Because of its thin line structure, it may be difficult for a user to select a particular clone instance in a Bar Map from a region densely populated with bars. Similarly, it is difficult to show any labels on the Bar Map other than via tool tips, requiring further interactions.

6.2.2. Triangular Matrix View (tmv)

The Triangular Matrix View [P6] identifies code regions with clones and states the frequency of a clone in those regions. It can be viewed as a variant of Bar Maps turned by 90 degrees. It decorates a code sequence with the regions containing clones. Similar to Bar Maps and Seesoft (see Section 6.2.3), the source code is represented as a single sequence P at the top (cf. Fig. 7). While regions of cloned code in Bar Maps are highlighted as bars decorating a representation of the code sequence scaled to a text view, Triangular Matrix Views use triangles to decorate regions with code clones, which are drawn below the sequence. The left point where the triangle meets the source-code sequence is the start, S, of the region about which information is provided by the triangle; the right point where the triangle meets the single source code sequence identifies the end of the region, E, within the code sequence. In the following, let R be the region of a triangle as the sub-sequence of P starting at S and ending at E. Each region contains at least two instances of a found clone C. At the bottom tip of the triangle a labeled and colored node is placed. This node is connected to S and E with two gray lines forming the triangle and represents the region R. Clone information about the clone C in R is visualized at the node as follows: (1) the content of the clone C is shown textually—at least in abbreviated form—as the label of the node, (2) the absolute number of occurrences of C within R (this is named "support") is visualized by the height of the font of the textual representation of C as the node label, (3) the relative frequency of C in R (this is named "confidence") is presented as a color gradient of the background color of the node. The support metric is the number of occurrences of C in R, whereas the confidence value is the support divided by the
maximal number of possible occurrences of \( C \) in that region—in other words, the number of all sub-sequences of length \( |C| \) in \( R \). The terms support and confidence are similar to equally named concepts for association rules in data mining for frequent item sets. The retinal encodings used are color for confidence, font size for support, and connection (gray lines connecting the node at the triangle tip to the start \( S \) and \( E \) of the region \( R \)). The x-position of the code lines in the sequence at the top of the figure is the textual order in the source code. The y-position of a region’s node corresponds to the length of \( R \): the larger the region, the farther the node label is positioned. This way the extent of a region can easily be spotted. Because smaller regions with clones may be nested in larger regions with the same clones, the connecting lines and the node layout may give the impression of nesting of regions with clones. A user can interactively set the thresholds for the clone length, support, and confidence to filter clones.

The Triangular Matrix View shows in which region clones occur (N1, N16), how often they occur (N4), and also the dispersion of clones (N2), that is, how far away instances of the same clone class are separated. Yet, while there is a natural strict order of code lines within a file, there is no such strict order for files. The positioning of files within the single-line sequence may, however, influence the insights that can be gained from this visualization. If two files with clones are placed together, the triangle is minimal. If they are put more remotely from each other, the triangle gets bigger. Scalability may be an issue, too. A very large program can hardly be drawn as a single line. If there are many clones, a very large number of triangles will decrease the visualization’s readability. Hence, this visualization might be useful for showing clones within a single file or among a smaller set of files.

6.2.3. SeeSoft (ss)

The original SeeSoft visualization by Eick et al. [115] was proposed to show line-oriented software metrics, such as test coverage or the recently changed lines. Each line is mapped onto a thin row (marker: line), whose color or color gradient is a visual encoding of the metric of interest. The lines of the same file are framed together (retinal encoding enclosure) and ordered in their textual appearance in the file (y-position). SeeSoft can be imagined by viewing the source code from a very far distance, where one is no longer able to read the content of a line but at least able to recognize the code layout. SeeSoft in general is used in various areas such as source code discovery, project management, quality assurance and testing, source code coverage analysis, and source code optimization [115].

This type of visualization was adopted and adapted by the clone-visualization community. In this adaptation (e.g., Fig. 8), files are shown as vertical bars and code lines as thin horizontal rows within bar segments (stripes) representing clones [P51,P62].

The color of each stripe is an encoding of a clone class or may represent another clone characteristic such as the number of clone instances. This helps in visualizing a large volume of source code on a single screen, where the user can visually find interesting patterns of the code.

The Clone Visualizer [P62], CloViz/CeDAR [P51], and ConQAT/Teamscale [U15] are examples of tools using this kind of SeeSoft visualization. Martinez et al. [P38] have used a very similar kind of visualization to examine similarities and differences among a collection of model variants specified in a textual notation. This visualization is typically enriched with further interaction. For instance, tool tips can display details about a selected clone. By selecting a stripe, the corresponding source code may be shown in a text view. Filters can be used to display only the selected file and the files that have a clone relation with the selected file. The SeeSoft visualization is used to display the position and size of each cloned segment, some metric of interest, and – with further interaction – its relationship to other segments. Stripes with the same color are clones of each other (N1, N2, N4, N5, N6, N7, N13, N14, N15). Multiple bars of corresponding files across versions or variants can be placed next to each other to analyze multiple systems or versions (N3, N13, N14, N15, N17, N18, N20, N21).

SeeSoft is helpful in displaying clones at the level of files. It provides a summarized version of the Code Inspection View (cf. Section 6.2.5) and may help in identifying the interesting parts and cloning patterns in a system. However, it is difficult to identify individual clone classes – if they are encoded by color – when there are too many clone classes, as the human eye can only distinguish between a small numbers of colors. Similarly, it is hard to show code fragments belonging to multiple clone classes because of the difficulty of showing overlapping regions with colors.

6.2.4. Tag Cloud View (tcv)

A Tag Cloud View, also known as Weighted List or Word Cloud, is a way to represent the importance of keywords. The significance of each keyword is represented as a retinal encoding by the font color or size or both. The x- and y-position of each word is often arbitrary, but may also be used to put highly rated words into the middle, analogous to fish-eye views, or to group words co-occurring in the same context together. This visualization helps a user to quickly identify most important keywords [116].

In software cloning, tag clouds [P48] are used to display the set of identifier names such as method, variable, and type names or
words in comments in the cloned source code (cf. Fig. 9) to help a user to more quickly grasp the semantics of a clone (N34). The size and color may denote the frequency of those words. Color may alternatively be used to represent the type of word such as word in a comment or identifier for a method, type, and variable. Rather than visualizing only the words of a clone, Tag Cloud Views can also be used to visualize all words of a file, where color can then be used to distinguish among words occurring in cloned or non-cloned code. For instance, in Fig. 9, red words are a part of cloned source code, whereas black words are never cloned.

Superficial differences of words can be reduced by stemming and very common, yet relatively meaningless words can be eliminated by stop-word lists. Nonetheless, the underlying assumption of Tag Clouds is that the metric determining the size of the words reflects their relevance to a user. Frequency – although simple to measure – size may or may not be a proxy for relevance. Sometimes rare words are most significant. Furthermore, empirical research by Oelke and Gurevych [117] has shown that users of Tag Clouds more easily understand them, if the words are grouped semantically, which calls for ontologies and more advanced linguistic processing. If there are too many words, Tag Clouds will occupy too much space and become too overwhelming.

6.2.5. Code Inspection View (civ)

If a developer needs to assess the differences of similar, yet not identical clones in detail, he or she needs a view that shows similarities as well as differences at the level of code. Inspecting differences is also a frequent task in assessing changes in version histories and merging in the context of version control. Tools developed for these contexts can be re-used for showing differences among clones. The distinction between visualizing differences in cloning and version control is the scope of what is shown. While traditional differencing tools consider whole files—typically two variants of the same file where the order of lines in a file has not been changed much, in clone detection only cloned regions of generally different files are to be compared, where only the cloned segments may correspond between files.

A Code Inspection View is an adaptation of file difference visualization for the specifics of clone analysis to let a user inspect the differences (N9) among instances of a clone class (N2). Most variants of the Code Inspection View show only pairwise differences through two text views placed side by side. Some show the commonalities and differences among all instances of a clone class at once in a unified representation. All Code Inspection Views need to (1) distinguish between cloned and non-cloned code, (2) show the common code of clones, and (3) highlight the code differences, that is, what is added, removed, or modified. Whether code is considered added or removed, is relative to a selection of the origin. The various types of Code Inspection Views differ in how they visualize those aspects. Tools show either only the cloned code [U9] or the cloned code embedded in the complete file [P32,P35,U8,U12,P12]. In the latter case, the cloned code is typically shown with a background color that distinguishes cloned from non-cloned code as a retinal encoding. Added code in type-3 clones is frequently shown in a particular background or foreground color and removed code may be crossed out or highlighted in yet another color as a retinal encoding. Background colors are often preferred, because they do not collide with colored syntax highlighting. Modified code may appear as removed and added code or – if code is shown side by side – highlighted in a color of its own. Fig. 10 shows the Code Inspection View by the commercial vendor Pattern Insight as an example.

Comparisons of more than two fragments require further annotations. Some tools, for instance, CeDAR [P52], use a unified presentation showing the common and different code of all clone instances in one place. The differences are highlighted and can be inspected interactively (cf. Fig. 11). The Near Duplicate Browser [P37,P36] offers a view that highlights the differences between one selected instance of a clone class to every other instance of that clone class (cf. Fig. 12).

The Code Inspection View may not only compare instances of clone classes (N9), but can also be used to compare the same logical fragment across multiple versions in a clone genealogy (N17, N19) to inspect the changes to a clone in the version history [P28]. Besides showing the textual differences among clone instances (N9), Code Inspection Views often offer a list of
Clone classes from which to select the instances to be shown (N1). Because, they show the source code, they give insights into the syntactic constructs contained in the clone (N8), although in a rather raw form without any kind of abstraction or summary. If they show the clones embedded in the wider context of the files containing those clones, they allow a user to inspect the context of the clone—but yet again at a non-abstracted level (N12).

Fish et al. [P16] have proposed a visualization named CloneMap that allows one to detect missing, incomplete, or otherwise incorrect updates to cloned fragments of source code from one version (baseline) to another version (updated version). Both versions to be compared are visualized not only by their textual differences as the other types of Code Inspection Views described in this section so far, but also by their abstract syntax trees reduced to syntax subtrees that have been cloned within the baseline version. The tree nodes can be unfolded interactively. Syntax subtrees with consistent changes are drawn in gray and inconsistencies are highlighted by colors. This visualization allows one to locate the clones in the syntax tree of a program and identifies the syntactic elements of a clone (N8) and the syntactic differences between clones in different versions (N19).

A variant of a Code Inspection View proposed by Tsantalis et al. [P54] [P39] is the Refactorability Inspection View, which annotates the Code Inspection View with additional information regarding refactoring opportunities of clones. An example is shown in Fig. 13. Analogous to an ordinary Code Inspection View, two instances of a clone class are presented next to each other with their inconsistent differences highlighted in red and their consistent differences in green. A consistent difference is one where a statement or expression is substituted consistently between the two clone instances. An inconsistent difference is one that violates the precondition of the refactoring to eliminate the clone. Which clone instance should be considered the baseline for the comparison can be selected in a menu (point 1 in Fig. 13). Syntactic subtrees can be folded and unfolded as it is common in syntactic text editors, which is useful for long clones. Differences can be selected. In case a consistent difference is selected, the substitution is shown (point 2 in Fig. 13). A summary of consistent variable substitutions appears at the bottom of this view (point 5 in Fig. 13). If an inconsistent difference is selected, an explanation is provided in which way the precondition of the refactoring is violated. In some cases, the tool even makes suggestions what could be done to make the clone instances refactorable (N29, N30). Automated refactoring is still a research topic. Even though there is considerable progress and many researchers have proposed solutions for particular cases [118–124] [P54], the problem is still not solved, in particular for type-3 clones where clones can differ. Yet, all current approaches and those to come could use the Refactorability Inspection View to visualize whether and in so how clones could be refactored.

The Code Inspection View is one of the fundamental clone visualization techniques and very popular [U8, U9, U12] [P12, P26, P32, P35, P52]. It provides a view of the differences among clone instances at the lowest level of detail by presenting their source code. Hardly any clone management activity can be performed without this view, because the precise details of the differences among clones are so important. It is even used to support safe cloning. Cottrell et al. use a variant to help a developer in reusing code through copy and paste [P10]. A major limitation of this view is that it can only show a comparison of a limited number of clone fragments at a given time, in particular for type-3 clones with many diverse differences. Furthermore, if a clone is long, only a small part of its content may fit into the visible area. Synchronized scrolling of multiple panes is a useful feature to compare long clones, yet scrolling has always a higher cognitive demand because of limitations of the human short-term memory.

6.3. Temporal data visualization

In the context of evolutionary data on cloning, temporal data visualization comes into play. Here data are organized along a time line and may have a start and finish time overlapping with the lifetime of other data items. Many researchers in software engineering have proposed various kinds of visualization of software evolution in general [115, 125–131]. In software evolution, data of not only one software version are visualized, but of a sequence of versions. In our case, the software data are related to software clones and how they evolve.

6.3.1. Time Series (ts)

A Time Series displays the change of one or more variables of at least ordinal scale over time. Time Series can be considered a particular kind of Line Chart. A Line Chart relates two arbitrary variables along the two axes, while in Time Series one of the axes is determined to present time.

In clone visualization, Time Series are used to show the evolution of a single or multiple clone metrics over time. In Cyclone [P20, U7], for instance, a Time Series is used to show different clone related metrics against the versions of the software (N19, N20, N21). The x-axis represents versions of the system, whereas the y-axis represents the corresponding quantity of the respective clone metric, such as number of clone classes, and cloned lines of code.

Because Time Series can be considered a special case of Line Charts, where one of the axes represents time, all strengths and weaknesses discussed in Section 6.1.3 for Line Charts apply to Time Series, too.

6.3.2. Clone Evolution View (cev)

The Clone Evolution View visualizes the evolution of clones as one particular aspect of software in the version history. It shows clone classes for each version of the software and matches the clone classes across versions to show their evolution as clone genealogies. With this view, a user can examine the appearance of new clone instances (N17, N20) and whether modifications to clones were made consistently (N36, N38).

![Fig. 12. Differences among instances of the same clone class by Near Duplication Browser [P37, P36]. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)](image)
An example of the Clone Evolution View of Cyclone [P20,P18] is shown in Fig. 14. A dot (marker: point) represents a cloned code fragment in one particular version of a program. All instances of a clone class (N4) of one version are nested in a rectangle (retinal encodings: shape and enclosure). The y-axis shows the timeline in the version history and clone fragments corresponding between subsequent versions are connected by lines (retinal encoding: connection). The nodes are positioned so that preferably each clone instance and clone class has its own lane while minimizing the required space. Other than that the x-axis has no further meaning, although it could be used to represent the order of another aspect, for instance, length of the clone instances or time of birth of a clone class. Color highlighting is applied as a retinal encoding of changes in the version history. If the background of a rectangle of a clone class is colored with light gray, the fragments of that clone class have changed consistently from the version where the clone class is colored to the next version. If the rectangle is colored with dark gray in the background, the fragments of the clone class were changed inconsistently. If no changes were made, the rectangle background is white. The frame color of the rectangle indicates the type of clone: black for type 1, dark gray for type 2, and light gray for type 3. The view is interactive and one can select a clone class to more easily follow it along the timeline. The selected clone class uses a color gradient of red to show the same types of changes while highlighting its selection. Hovering over clone classes reveals more details. Cyclone draws on ideas of SoftGUESS [P1], where this view is called the Genealogy Browser. The additions of Cyclone over SoftGUESS are the rectangles enclosing clone instances, the coloring of changes and clone types, and the orientation (SoftGUESS shows the time line from left to right).

The Clone Evolution View shows the lifetime of clone classes at the level of code fragments including transitions and clone types. Transitions occur when a change in a particular clone class took place either consistently or inconsistently. Whether the Clone Evolution View scales to very large numbers of clones in large systems over a very long history is questionable because scrollbars may be needed for the x- and y-axes [P47]. Furthermore, clone instances can overlap or completely subsume other smaller clone instances, which cannot be seen in this visualization, because the clone instances within one version appear as independent dots on the x-axis.

6.4. Bipartite graph visualization

Binary relations can be modeled as graphs where the nodes represent entities and the edges represent their relations. Bipartite graphs are graphs in which the set of nodes can be separated in two disjoint sets, where there is no edge between nodes in the same set. Such graphs occur when there are two different types of concepts in a domain and one wants to model only the relations among those two types of concepts. In the domain of cloning, one concept could be a clone and the other concept a container of clones, such as a file, and one wants to visualize only where clones are contained in. This section describes clone visualization where the data items and their relations form a bipartite graph.

6.4.1. Clone-Class Family Enumeration (ccfe)

The Clone-Class Family Enumeration is a part of Rieger et al.’s polymetric views for clones [P44,P46]. Polymetric views were first introduced by Lanza and Ducasse [132] for visualizing attributed
graphs. They can accommodate five different variables which are mapped on the retinal properties width, height, and color (or color gradient) and the x- and y-position of elements presented as rectangles for the nodes of the graph. One could even use textures or shapes to encode additional aspects. Relations among those elements are represented by connections – the edges of the graph – which may use thickness, coloring, or shape such as straight line, and dotted, to encode additional information.

The Clone-Class Family Enumeration visualizes the bipartite relation among clone-class families and source files (cf. Fig. 15). A clone-class family – or super clone – is a group of all clone classes that have instances in the same set of source files (N1, N2). In this visualization, smaller nodes represent clone-class families, whereas larger nodes represent source files; that is, size is used as a retinal encoding of the entity type a node represents. The placement on the x-axis of clone-class families and source files is dependent upon the lines of copied code (LCC) (N7) and the lines of code (LOC) metrics (N15), respectively. The diagram is split vertically in two levels. Above are all clone-class families where the y-axis in reverse represents the number of source files (NSF) of a clone-class family and in the level below are all files where the y-position is defined by the number of clone-class families of a file. Clone-class families located in the top rows are not much significant as they link to only a small number of source files, while the families below represent a larger set of source files. Similarly, source files located at the bottom of the view are related to only a single clone-class family, whereas files above are related to more clone-class families.

The Clone-Class Family Enumeration abstracts from single clones classes and summarizes these as clone-class families – that is, shows a relation among clone classes – at file granularity. Programmers can easily see the larger clones and files (x-axis) and also the locality of clones (y-axis). The most severe clone-class families can be found in the right lower corner because they contain a large amount of cloned code in a large set of files (N11). Analogously, the most severe files can be found in the right upper corner because they are large and are related to many clone-class families. However, if there are many files and clone-classes, the visualization might not fit on the visible screen and scrollbars might be needed. Edges and nodes may overlap other edges as well as other nodes, which decreases readability. Width and height of the boxes have no further meaning except that they signify the type of node. Similarly, color, texture, and shape are not leveraged.

6.4.2. Clone Cohesion and Coupling Graph (cccg)

The Clone Cohesion And Coupling Graph—part of the set of visualizations by Jiang [P29, P27]—shows the cohesion of a system by displaying cloning within subsystems (N1) and coupling induced by cloning across subsystems (N3, N13, N14) as a node-link diagram. An example can be seen in Fig. 16. The visualization reminds of the entity-relationship diagram by Chen [133] with some visual adjustments.

The nodes (mark: area) represent two different types of items with shape as retinal encoding of their type. Rectangles indicate subsystems whereas diamonds indicate super clones—an aggregation of all clone classes between the same subsystems. An edge connecting a rectangle with a diamond is a retinal encoding of the containment relationship. The rectangle height is a retinal encoding of the number of clone classes present within the subsystem and its width represents the number of cloned lines within the subsystem (N6). Similarly, the height of a diamond indicates the number of clone classes and its width the number of cloned lines involved with its connected subsystems. An increased size of a diamond indicates a large amount of cloning across subsystems (high coupling). Similarly, an increased size of a rectangle indicates a large amount of cloning within a subsystem (high cohesion). A force-directed layout is used to place the nodes. Thus, subsystems having more clones with each other are positioned nearer to each other supporting the law of proximity of the gestalt theory, which states that entities closer together will be perceived as related by a human beholder. The absolute values for the x- and y-positions of the nodes have no further meaning. Different colors for diamond nodes are used to highlight the amount of cloning—based on heat coloring from gray, over green and yellow to red assigned by percentiles of the clone-rate distribution.

The Clone Cohesion And Coupling Graph is a visualization of cloning at the architectural level. It provides a summary about the extent of clone coupling and cohesion in a software system at a higher level of cloning through super clones. It is helpful in highlighting the most significant clones within subsystems and
across subsystems. Heavy cloning between subsystems can be spotted as large red diamonds connecting many rectangles and heavy cloning within subsystems as large rectangles. Subsystems are encoded visually by two visual attributes: shape (rectangle) and color (blue). A color gradient of blue could be used for additional information. Based on a force-directed graph layout highly coupled subsystems appear in close proximity. As the number of nodes and edges grow, it becomes increasingly difficult to "read" the graph. Laying out graphs aesthetically (or readable if you prefer) is a computationally demanding problem and a research area of its own. Generic aesthetic constraints are to minimize edge crossing, avoiding node-edge overlap, showing symmetry where it exists, preferring uniform edge lengths and node distribution, and more. In addition, there are often application-specific semantic criteria or conventions determining an intuitive understanding of a graph layout, in our case for instance, that two subsystems are part of the same architectural layer. Recent progresses in the graph-layout research community, summarized for instance by Gibson et al. [134]—such as dimension reduction, multi-level techniques, or constraint-based layouts have not fully been explored in the area of clone visualization yet.

6.5. Tree data visualization

Hierarchies can be frequently found in modeling software data including clone data. For instance, clone instances may be contained in files, which are contained in directories, which in turn may be contained in other directories. It is quite obvious to visualize hierarchies as a tree. This section summarizes clone visualizations focusing on hierarchical data structures.

6.5.1. Navigation Tree View (ntv)

The Navigation Tree View is a generic and commonly used visualization to display hierarchical data items forming a tree. The hierarchy is shown as a list with increasing indentation at each level of the tree. The items are presented generally as a label (controlled processing graphical feature) but are often decorated by an icon representing the type of the element (retinal encoding: shape). A user can drill down into the sub-items of a specific item recursively, until an item has no children left by successive unfolding (retinal encoding: enclosure). Similarly, an item can be collapsed to hide sub-items. The positions are often meaningless, but the ordering of items at the same level (siblings) could also be chosen to reflect a ranking (N11).

Navigation Tree Views are used in various applications such as project explorers in IDEs, XML viewers, and others. They are popular in clone visualization, too. Tools with clone visualization such as Atomiq [U1], ConQAT/Teamscale [U6], VisCad [U16], CLICS [P33, U2], and SimEclipse [P56] are using it for different purposes. An example of SimEclipse is shown in the lower part of Fig. 17. The leaves are cloned methods identified by their signature and their code range in square brackets. They are nested in a file, which in turn is nested in a directory. If a user selects a cloned method, its source is shown in the text editor above. In general, Navigation Tree Views are used in clone visualization to present a hierarchical view of various clone related elements, such as clone instances, clone classes, source files with clones, and directories to show their locations in the code hierarchy (N1, N2). If an inner node represents a clone class and its children are clone instances, a user can see the cardinality of a clone class (N4). This number as well as the volume of a clone class or the length of a clone instance can also be shown as additional annotations analogously to the code range of Fig. 17. Yet, a user needs to unfold elements to see the entries and the data is shown rather inconspicuously and not truly by some kind of visualization, hence, requires controlled processing in the beholder’s mind.

A Navigation Tree View can present all type of hierarchical data related to cloning such as clone classes and super clones up to the levels of files and subsystems. Developers are familiar with it from other applications and, hence, unlike many other kinds of clone visualizations Navigation Tree Views should not require any kind of training to use them. They provide a similar screen efficiency as that of tables. However, as the hierarchy grows, it becomes difficult to navigate through the tree as there may be many entries and many levels may need to be unfolded. Moreover, it offers very little abstraction.
6.5.2. Hierarchical Tree View (htv)

A Hierarchical Tree View [135,136] shows hierarchical data as a node-link diagram where a link represents containment. Thus, the retinal encoding of containment is by connection, not by enclosure. The shape, size, and color of nodes and links are additional retinal encodings for other kinds of information. The x-/y-positions of nodes depend primarily upon the chosen layout (e.g., top-down, radial, or orthogonal; see also [137]). The order of nodes with the same ancestor, however, is a degree of freedom and may be used to reflect a ranking of siblings (N11). A Hierarchical Tree View may show the same data as a Navigation Tree View, but has additional degrees of freedom by having links that can be decorated and colored – whereas Navigation Tree Views show containment only through nesting – and by sizing and coloring the nodes differently. Although the edges are logically directed, they usually do not have arrowheads to indicate the direction. Instead the direction is implicit and decided by the tree layout algorithm uniformly, for instance, from an upper node to a lower node in a top-down layout.

Many tools were developed to display Hierarchical Tree Views in diverse disciplines such as biology, business organization, and computer science. For clone visualization, a Hierarchical Tree View can be used to show cloning at various levels of granularity as well as the evolution of clones over time and also in different kinds of hierarchies, such as the file-system or type-inheritance hierarchy. Rieger et al. [P46], for instance, use this visualization to display the layout of the system directory structure and its cloning (N1)—named System Model View by the authors (see Fig. 18). Leaf nodes in this tree represent source files, while internal nodes represent directories. They combine the node-link diagram with ideas of the polyomict views proposed by Lanza and Ducasse [132] by mapping attributes of an entity shown as leaf node onto the width and height of the node. The width of a leaf node in their System Model View denotes the number of code copied externally to other files (LEC) and the height represents the number of lines copied internally (ILC) in the file (N6). The tree is overlaid with other undirected edges among leaves connecting files with shared code (retinal encoding: connection). The thickness of those cloning edges indicates the amount of cloning between the two files (N13). While these additional clone edges turn the tree actually into a graph, we still classify this kind of visualization in the category tree of our taxonomy in Fig. 3 because nesting is the primary criterion for ordering the data.

Jiang [P27,P28] has used this visualization to present the spread of cloning in a system superimposed on the directory tree of the system (cf. Fig. 19). Analogous to Rieger et al.’s Hierarchical Tree View, its internal nodes indicate directories and leaves represent files. Whereas Rieger at al.’s System Model View gives only a view of the file system, Jiang’s view presents only inner nodes in different sizes, where the height of the inner directory nodes corresponds to the number of clone instances and the width represents the number of cloned source lines. External cloning of a node is indicated by the thickness of the edges representing the hierarchy. Jiang’s visualization, named Clone System Hierarchical Graph, does not connect files sharing code as Rieger et al.’s System Model View does. While the focus of the System Model View is on the leaves (files), the focus of Jiang’s Clone System Hierarchical Graph is on the internal nodes (directories). Another difference is that Jiang’s ordering of siblings has meaning, reflecting the number of their children.

Jiang [P27] has also used a variant of a Hierarchical Tree View to highlight changes in the number of cloned source lines over a series of versions (cf. Fig. 20). In this variant, the Clone System Hierarchical Graph is used to visualize the nesting of directories and files for the last version, where this time the nodes are of equal size and the edges are all equally thick. The children of an inner node are ordered by the degree of changes in cloning from the highest to the lowest. The degree of changes is calculated by a decay function as follows:

\[ \text{decay} = (\text{new value} - \text{old value}) + \text{decay rate} \times \text{old value} \]

where \( \text{old value} \) refers to the number of internally as well as externally copied lines of the first version of the series of versions to be considered, \( \text{new value} \) is the corresponding value of the last version, and \( \text{decay rate} \) is a factor that can be chosen by the user. If a user is more interested in the more recent changes, the decay rate should be set small and vice versa. The decay value is then discretized into one of four categories and edges with the highest degree of changes are colored in red, followed by yellow, light green and gray for very little changes. This variant of a Hierarchical Tree View is named Clone System Evolution Graph by Jiang and emphasizes changes of cloning in version histories (N20).

In SoftGUESS [P1], a Hierarchical Tree View (named Encapsulation Browser by the authors) is used to present the hierarchical containment of clone instances (leaf nodes) into methods, classes, and finally packages as inner nodes. Colors are used as a retinal encoding of clone classes for the leaf nodes. This way, the spreading of instances of a clone class can be seen (N2). The Encapsulation Browser uses also animation to show how clones have moved in the hierarchy in the version history (N18). To do that, each version is shown as a Hierarchical Tree View and the Encapsulation Browser allows one to move from one version to the next one, where each clone appears at the location of its predecessor in the genealogy and moves to its new location in the hierarchy. To the best of our knowledge, this is the first and only type of clone visualization that uses the position \( t \) (for time; see the classification of human-vision facets presented in Fig. 2).

Another variant of a Hierarchical Tree View can be found in VisCad (cf. Fig. 21), there named Hierarchical Dependency Graph, to present the distribution of clones across subsystems (N1) [P3]. It indicates the dependencies among different subsystems based on the common clones between them. This tree represents the directory hierarchy of the system with directory and file nodes arranged along concentric circles such that the root directory is located in the middle and the containment relationship between directories and files is shown with directed edges. The radial layout is the most distinct factor of the Hierarchical Dependency Graph over other kinds of Hierarchical Tree Views. The node width and height represent the cohesion (clone rate within a node, N6) and coupling values of directories (clone rate with external nodes, N14), respectively, while the node color represents the degree of
external duplication with the selected directory (N13). For leaf nodes, these measures directly related to a node, for inner nodes, the values are aggregated from their descendants. The shape of a node is a retinal encoding of a node's type (file versus directory). The edge width can also be used to indicate the number of clone classes or clone pairs shared among a pair of directories (N13). To simplify this visualization, a directory may be selected (shown as a diamond shaped node) and all nodes which do not contain any cloning relation with the selected directory will be filtered.

The Hierarchical Tree View is a form of a node-link diagram showing where clones are located in a hierarchical structure of a program. The hierarchy is made explicit through links representing the nesting. It offers a large degree of freedom to accommodate other kinds of information through colors, sizes, shapes, textures, and positions. It can be overlaid with other types of edges relating, for instances, nodes sharing code—although there should not be too many of these to avoid edge crossing. When nodes and edges have tags, this view has problems in screen area usage [138]. Hierarchical Tree Views do not use screen space efficiently and do not scale well to large datasets. It requires a great deal of scrolling for large hierarchies. In order to resolve some of these limitations, several variants have been proposed such as Tree Maps [139], Cone Trees [140], the Hyperbolic Browser [141], Elastic Hierarchies [142], and Cactus Trees [143]. To the best of our knowledge, only Tree Maps among those alternatives have been tried in clone visualization so far (see Section 6.5.3). If clone-visualization researchers are interested in visualizations of trees using those alternatives, they should benefit from existing comparative evaluations among these alternatives such as the one by Dang et al. [144], who conducted a user study to compare those alternatives for trees where the nodes may have additional relations to each other expressed by bundled edges.

6.5.3. Tree Map

A Tree Map [145] presents hierarchical data (a tree) as a group of nested rectangles (mark: area, retinal encodings: enclosure). A rectangle captures each branch of the tree, which further contains smaller rectangles to denote sub-branches or leaves. The area size of an innermost rectangle is a retinal encoding of a given property of the leaf element and chosen proportionally to that property. Often the color of a rectangle is an additional retinal encoding of another property. Holten et al. [146] proposed to add textures for yet another property and Van Wijk and Van de Wetering [147] to add shading to increase visual realism and to enable a human viewer to better distinguish neighboring rectangles from each other. How rectangles are divided and ordered is determined by a tiling algorithm. There are different variants of these. One of the earliest and simplest ones is slice-and-dice tiling, which alternates the orientation of rectangles at each level of the hierarchy (vertically vs. horizontally). The popular squarified tiling instead attempts to keep each rectangle as square as possible, which is advantageous because humans have difficulties in comparing areas with different aspect ratios.

Tree Maps are used in several disciplines such as biology, entertainment, financial industry, and computer science. For clone visualization, the Tree Map typically represents the directory or package hierarchy of the system with rectangles representing sub-directories (or sub-packages) as outer rectangles and files (or classes) as the leaves. Fig. 22 gives an example Tree Map of Cyclone [U7]. The size of a rectangle denotes the number of lines of code contained in that element (N15) and the color ranges represent the clone rate in a leaf (N1, N6). Cyclone’s Tree Map is interactive. A user can select a rectangle (shown in blue) and all other rectangles sharing code with the selected one are highlighted in green and connected by edges (retinal encoding: connection). The number in the green rectangles denotes the number of clone instances shared with the selected one (N13). Tree Maps can be found in various clone analysis tools, e.g., VisCad [P3], ConQAT/Teamscale [U5], and SolidSDD [P43].

Rieger et al.’s polymeric views for clones contain a variant of a Tree Map named Duplication Aggregation Tree Map [P46]. An example is shown in Fig. 23. The difference among the two is that the Duplication Aggregation Tree Map maps two quantities onto a rectangle’s shape, namely, onto its width and height, while a classical Tree Map maps only one quantity onto the area of a rectangle. Inner rectangles are laid out so that their widths and heights do not overlap, which causes empty space as opposed to traditional Tree Maps. A leaf rectangle denotes a file, the outer rectangles are directories. The width of a leaf rectangle denotes the number of code copied externally to other files and the height represents the number of lines copied internally (LEC) in the file. The width and height of a rectangle representing a directory are the total LIC and LEC, respectively, of the files and sub-directories contained. The gray level of a rectangle’s area visualizes the overall number of copied lines (LIC + LEC).

Wang et al. [148] introduced an alternative to the Tree Map called Circle Packing, also known as Circular Tree Map or Balloon Tree. Circle Packing is also an enclosure diagram, where circles are used instead of rectangles. The advantage is that nesting is easier to see; the disadvantage is that more space is needed. Murakami et al. [P42] have adopted this type of visualization in their tool ClonePacker to visualize clones (Fig. 24). In ClonePacker, the innermost circles indicate methods, the circles wrapping the innermost circles indicate files, and the circles wrapping file circles represent directories. If a method is a type-1 clone, it is colored in red. Type-2 clones are colored blue and type-3 clones green. The method a user selected to show its source code is colored yellow.

A Tree Map efficiently utilizes the display area [145]. It can display hierarchical structures that show attributes of a code element by size, color coding, and textures. It enables users to compare sizes of rectangles and is especially helpful in revealing patterns [149]. Showing labels for rectangles is a challenge. Unless there are only a few data points, the labels quickly disappear or become unreadable, hence, they can generally be revealed only through additional interaction, e.g., mouse hovering. Similarly, the implicit hierarchical structure in a Tree Map is hard to discern [14] even if shading is used. A Circle Packing better displays...
the hierarchy. However, it is not as space efficient as a Tree Map. For clone visualization, a Tree Map presents a general overview of the entire subsystem and file hierarchy, which eases navigation of large hierarchies. Yet, when visualizing a huge system, the rectangles in the Tree Map can become very fine grained. Tree Maps are not good at visualizing the structure of balanced trees, where each parent has the same number of children and each leaf has the same size [150]. Stasko et al. [151] also report the limitations of Tree Map in terms of presenting structure and hierarchy by comparing it with a sunbursts layout. As a result many approaches have been proposed as alternatives to Tree Maps such as Voronoi Tree Maps [152], Jigsaw Tree Maps [153], and Gosper Maps [154]. To the best of our knowledge, only Voronoi Tree Maps have been tried for visualization of clones (and code smells in general) so far [P49].

6.5.4. EvoStreets (es)  
Utilizing the third dimension, an additional metric in Tree Maps can be depicted by mapping its value to the height of the corresponding area (z-position), yielding three-dimensional blocks [155–157]. Due to the effect of perceiving these blocks as a city, this visualization is known as CodeCity. Tree Maps and CodeCities, on one hand, are well suited to visualize large software systems in limited space but, on the other hand, have the disadvantage of being inconvenient when the underlying structure changes over time [158]. For example, when analyzing the evolution of a system – elements are added, removed, or relocated –, the layout may change radically and thus the mental map of the observer may be lost. Moreover, comprehending the hierarchy is a challenging task due to missing distinct patterns in the city to memorize for navigation purposes.

To overcome these shortcomings, Steinbrückner and Lewerentz [159] introduced a visualization named EvoStreets. Building on the city metaphor, EvoStreets visualize a software’s structures using road junctions rather than subdivided areas (cf. Fig. 24). That is, each level of the hierarchy is depicted by an individual line, representing a street within the city. The lower a level is, the thinner the corresponding street gets. The leaves of the hierarchy, in turn, are depicted using three dimensional blocks as in the CodeCity approach placed next to the streets (marks: surface and volume). The nesting is expressed both by connection (forking streets) and spatial enclosure. Subsequently, the generated layout allocates more space than Tree Maps and CodeCities but, at the same time, is more robust to changes in the underlying structure and may feature more distinct patterns of city districts that can help in navigation thereby maintaining the mental map. Height, breadth, width, texture, and color of a block can be used as retinal encodings of other attributes. The positions on the x- and y-axes are determined by the layout algorithm specific to EvoStreets [160] and have generally no meaning, although the ordering of forking streets and placing buildings within the same level of the hierarchy could express some ranking (N11). All buildings are typically on the same plane, in which case the z-position of a block’s ground area has neither meaning, although Steinbrückner [160] has also introduced hills to express, for instance, the age of an element represented as a block.

Steinbeck et al. used EvoStreets to visualize cloning at the system level in virtual reality (VR) [P50]. The streets depict the subsystem hierarchy in terms of directories and the buildings represent files. The buildings have all the same shape but differ in color and volume. For their particular purpose, color, width, height, and breadth were used as a retinal encoding of the same property, namely, the number of copied lines of code of a file.
so that the same aspect could be seen from every possible angle in the virtual world. If two files share code, they are connected by hierarchically bundled edges [161] (see Section 6.7.2). The visualization shows where the clones are located (N1), how much is cloned (N6), and whether two files share code. Other attributes such as how much has been cloned externally (N13, N14) or the size of the files (N15) could be easily mapped onto its visual attributes, too. Since the original design of EvoStreets envisioned software evolution, many of the questions related to tracking clones across versions could be integrated by animation (position for time) (N17, N20, N21) showing the evolution of cloning as a movie, as already envisioned and implemented by Steinbrückner [160] for the original EvoStreets.

Steinbeck et al. did not intend to introduce a new type of visualization. Their main research focus was rather to compare human cognitive capabilities in orientation in 2D and 3D visualization on a desktop environment versus VR with a head-mounted display in a controlled experiment. Visualizing cloning with EvoStreets was just one use case out of a wide range of potential uses. To the best of our knowledge, however, they were the first ones using EvoStreets for clone visualization.

3D and VR techniques are relatively new in the domain of clone visualization, but have been explored for quite some time in software visualization in general. Knight and Munro [162] gave an overview on VR for software visualization as early as 2000. Since then various other researchers have used VR and 3D for software visualization for static information [163–170] or for dynamic data such as resource bottlenecks or memory leaks [171,172]. Elliott et al. [173] discuss the affordances and challenges of VR in software engineering in general and present ideas on how it can be used for code reviews. There is already a great body of knowledge on VR and 3D visualization outside of computer science [174–176]. Studies in other disciplines have shown that head-mounted displays may enhance the orientation in three-dimensional environments as they allow to rotate and move more naturally as opposed to traditional approaches [177]. Sousa Santos et al. [178] give an overview on virtual environments, discuss several papers that compared VR to desktop environments, and report on an experiment they conducted themselves. In their experiment on navigation in a virtual maze, however, they found that – although users were generally satisfied with the VR system and
found the VR interaction intuitive and natural – most performed actually better in the desktop environment. Similarly, Steinbeck et al. [P50] found in their controlled experiment no significant differences in solving task that required counting nodes and edges (clones) among 2D, 3D, and VR visualization. Yet, they found a significant difference in the way how participants moved between 3D desktop and VR environments. Researchers have already started to visualize EvoStreets in VR [179,180] to investigate whether that assists in solving software engineering tasks by offering more intuitive interactions. It would be worthwhile to explore the advantages and disadvantages of 3D and VR visualization [169] specifically in the area of software clones in more detail, too. This holds not only for EvoStreets, but also for many other types of clone visualization.

6.6. Acyclic graph data visualization

Shorter clones can be part of multiple longer clones. For instance, a sequence α may occur five times in a program; twice in the context of a sequence αβ, twice in a sequence γα, and once independent of β and γ. Then the larger clones including α, namely, αβ and γα are clones, and α is a clone in its own right, too. If the containment of clones in other clones is modeled, a simple tree hierarchy, where each item is contained in at most one other item does not suffice. The data structure to model this kind of containment actually forms an acyclic graph. This section describes clone visualization showing relational data where the transitive closure R+ of the underlying relation R is irreflexive, that is, (n, n) ∉ R+ for every item n; in other words, does not contain cycles.

6.6.1. Hasse Diagram (hd)

A Hasse Diagram—also known as Lattice Diagram—is a visualization of a lattice, which consists of a set of objects, O, that are partially ordered by a given relation ≤. The lattice is drawn as a directed acyclic graph (DAG). The nodes of the DAG are the objects (marks: area) and its edges show the direct partial order between two objects (retinal encoding: connection), where transitive relations are omitted (i.e., if there is an edge (o1, o2) then ¬∃o′ different from o1 and o2 : o1 ≤ o′ ≤ o2). Hasse diagrams were developed in mathematics and are widely used to visualize partial orders. The nodes in Hasse diagrams are laid out so that all edges representing the directed relation ≤ are pointing in the same direction, in which case arrowheads are superfluous. Moreover, automated layouts attempt to minimize edge crossing. Other than that, the x- and y-positions have generally no meaning.

Johnson [P30] has used Hasse Diagrams to represent cloning relationships between clusters of files. Here the nodes represent clusters of files sharing the same code (N1) and are formed as follows according to Johnson [181]: “If one considers a graph whose nodes are files and where arcs have been added whenever a match involves a set of files containing the given pair [cloned fragment; note of the authors of this paper], then one can define clusters of files corresponding to the connected components of this graph”. The partial order F1 ≤ F2 between two clusters of files F1 and F2 holds if F1 is a subset of F2 (N32). Obviously, a file shares code with itself, hence, each file constitutes a leaf node in the DAG. All inner nodes show larger groups of files with common clones (N13). An innovation of Johnson in the visualization by way of Hasse Diagrams is to position the nodes vertically proportionally to the size of the file (N15) when a cluster is a singleton (leaf) or the number of lines of code shared between all files of a cluster (N7) when it is a set with more than one file (inner node). The size measure may be log-scaled if the range of its values is too wide. Then clusters that are larger in absolute or relative terms stand out without totally hiding weaker matches. An example is shown in Fig. 26, where node cm represents the cloned code shared between file c and m. Node mm, on the other hand, represents shared code among the files m and n. The cluster cm has more shared code than cluster mm and file m is longer than file n.

The Hasse Diagram gives insights into the complex relation of code sharing among files. Algorithms and tools exist to automatically draw them [182]. Lattices with many nodes and edges can become quite complex and edge crossing cannot always be avoided, however, so that navigation becomes difficult [183]. For this reason, ideas are being explored in the research community of information visualization in general to simplify Hasse Diagrams [184]. They have not been tried in the area of clone visualization yet. Johnson has not taken advantage of other retinal encodings in Hasse Diagrams such as shape, color, size, to accommodate additional attributes—something that could also be explored further.

6.6.2. Content–Context Diagram (ccd)

Kamiya [P31] suggests a graph-based visualization of clones that depicts both the content and the context of a single given clone class (N2) (cf. Fig. 27). The visualization is somewhat specific to his technique to identify clones, which is based on the call graph. Two functions are considered clones if they call a similar set of other functions directly or indirectly. The transitive closure in the call graph for a given function F identifies the set of functions called by F. The Apriori algorithm (frequent item set mining) identifies similar subsets among those sets for all functions. If two functions have a high overlap in their sets, they are considered clones.

The graph for the visualization is directed and acyclic (DAG) and constructed as follows. Given a clone class C and a call graph CG. The visualized DAG consists of multiple layers, each having different types of nodes (encoded by shape). Layer 1 consists of a single node constituting the root node R of the DAG. The root node R is the immediate shared dominator node in the call graph CG of all functions containing the clone fragments of C. A function d dominates a function f if every path from the main function leading to f contains d. The immediate dominator is the last function with this property, in other words, is dominated by all other dominators of f. Thus, R represents the context of the clone instances of C in the call graph. Such nodes are
graph that is neither a tree nor an acyclic graph. That is, the acyclicity condition of the visualizations described in the previous section is now dropped. A cyclic graph occurs, for instance, when the type-2 clone relation between code fragments is visualized. Because the type-2 clone relation is reflexive, commutative, and transitive, a graph representing three fragments that are all type-2 clones of each other forms a fully connected graph with cycles. Cycles often occur, too, when clone data are enriched by additional static dependency edges, for instance, method calls, which can be cyclic.

This section summarizes types of visualization for arbitrary graphs even if the shown relation does not always form a cyclic graph. We rather list a visualization in this section if it has the potential to show arbitrary graphs.

6.7. Dotplot (dp)

A Dotplot (or Scatterplot) is a two-dimensional matrix visualization technique where each cell of the matrix shows a relationship between two sets of items listed as column and row items. If a relation holds between two items on the two axes at the x- and y-position, respectively, a point is set as a mark where the respective row and column meet. The type of relation – if there are several types – can be encoded by the color, shape, or texture of the cell. If a non-symmetric shape is used for the cell, even its orientation could be used. Dotplots have been used since the 19th century in various disciplines such as mathematics, statistics in particular, and bioinformatics. In the latter field, Dotplots are often used for visualizing similarity among gene sequences, which is very close to their use in clone visualization. Many tools exist to draw them.

Dotplots are the first kind of visualization of software clones we are aware of. They were used as early as 1992 [P4,P8,P5]. Dotplots are used in clone visualization to show clone relations among code entities such as lines, methods, files, and directories, either from the same set (e.g., all methods of a program) or from two disjoint sets (e.g., all methods of either of two programs) put on the different axes [P4,P5,P8,P22,P45,P15,P58,P23,P44,P9,P35] [U8,U16]. For instance, in Fig. 28 the source directories of the Unix derivatives Linux and FreeBSD are compared to each other (a dot is set if two directories contain method clones), while in Fig. 29 the subsystems of FreeBSD only are compared to each other. Fig. 28 encodes the clone density (percentage of cloned methods) as a color gradient of the dot ranging from red for high density over green to blue for low density as in Heatmaps. Fig. 29 uses a Heatmap, too, which represents the clone coverage, defined as the percentage of cloned lines of code between two subsystems [P35]. Unlike Fig. 28, Fig. 29 does not use equally sized dots but areas as a mark, where the height is proportional to the size of the subsystem on the y-axis and the width proportional to the size of the subsystem on the x-axis. If the rows and columns represent elements from the same set of entities, as in Fig. 29, the matrix is symmetric and the diagonal is completely filled with dots, because clone relations are reflexive and generally symmetric. In this case, the Dotplot can show only one half of the plot, either below or above the diagonal line, as the other half will be a mirror image of the first half. The other half can be left blank or be used for a different kind of relation among the entities (e.g., static calls among methods).

In general, Dotplots can show all clones (N1) or only the instances of a selected clone class (N2, N4) within a program or across two (sets of) programs (N3, N13, N14) or versions (N17, N20). The size of the code entity containing clones can be visualized by the width or height of the rectangle as in Fig. 29 (N15). Color, texture, shape, and orientation of the cells can be used to visualize additional information such as the amount of copied code (N6). Instead of code entities, one can also put the

---

6.7. General graph data visualization

This section describes visualizations of relational information where the relation among items can be modeled by a general

---

https://www.graphviz.org/.
clones (at any level of granularity such as tokens, lines, methods, classes, and files) themselves onto the axes when two versions are to be compared. This idea was suggested by Saha et al. \[^{[47]}\] and is illustrated in Fig. 30 for the level of method clones. Here both axes list the same cloned methods but in two different versions. Then one can use retinal encodings (e.g., levels of gray for the dots) to indicate whether a clone changed (N19) and if so whether the change introduced an inconsistency (N36, N38). For a given method \(m\), the cell \((m, m)\) describes how \(m\) itself changed from one version to another (not leveraged in Fig. 30). If one looks further into the row or column, respectively, one can spot how its clones – that is, all clone instances of the same clone class – changed. A similar Dotplot was later proposed by Mondal et al. \[^{[40,41]}\] to visualize the degree of change of clones from one revision to another. The rows and columns of the Dotplot are clone instances of two revisions. Each cell in the Dotplot is colored according to a so-called SPCP score using a color gradient. The SPCP score (Similarity Preserving Change Pattern) is a measure of how similar two fragments are after a change. They call their visualization SPCP-Heatmap View. Because it is very similar to the other Dotplot visualizations presented in this section where the main factor for organizing the data is a matrix and the Heatmap is just a decoration of the matrix, we classify it as a Dotplot.
If a Dotplot shows individual lines or even tokens on its axes, the size of a clone (N5, as the number of dots in a diagonal), differences in abstract form (N9, as gaps or offsets), the number of instances of a clone class (N4, as repeated diagonals in a region of the matrix), and a visual impression of the degree of redundancy (N6, N7, N2) can be presented.

A lot of information can be put into a Dotplot, which explains why it is one of the most common visualization techniques not only in software cloning. Dotplots are also used in diverse other fields such as biology, where it is used for homology—the study of similarity in structure between different parts of the same individual. It also has various applications in software analysis, such as author identification, plagiarism detection, module and version identification, and information retrieval.

Dotplots can show large quantities of data and easily display correlation between variables and clustering effects. They can represent any binary relation on a continuous or discrete scale, but are limited to two sets of elements to be compared to each other. Dotplots can display cloning information at all levels of granularity such as subsystems, files, and code lines, where interesting visual patterns can emerge. Whether a visual pattern actually emerges, however, depends on the chosen order of the elements on the axes. In some cases, there is a natural order, for instance, for code lines within the same file. Then, dense regions with more dots represent parts of the software with a high degree of cloning and long connected sequences of dots indicate longer clones. If there is no natural order (for instance, among methods or files), a cluster analysis might be needed to find a suitable order to bring patterns forward [P58]. Behrisch et al. [185] provide a survey on matrix reordering methods for Dotplot visualizations in general. Clone-visualization researchers have not explored suitable ordering methods yet. Last but not least, Dotplots can become huge images, depending on the number of elements to be compared. Half of the space may even be wasted if the two sets of elements to be compared are identical. If the matrix is sparse, other types of relational visualizations may make better use of the screen.

6.7.2. Wheel View (wv)

A Wheel View or Chord Diagram with bundled edges is a visualization based on the spline model to visualize hierarchical entities and their relationships [161,186]. As such it is a visualization for a graph, whose composite nodes form a tree, on one hand, and whose edges may provide cyclic paths, on the other hand. Because the visualization of the edges is the primary aspect of the data visualized, we classify it within the category cyclic graph of Fig. 3.

An example Wheel View is shown in the left part of Fig. 31 and consists of two parts: the surrounding node area and the inner edge area. The structure of the visualized data is shown in the right part of Fig. 31 as a skeleton for the left part to explain the design of a Wheel View. The skeleton is used here only to illustrate the layout; the true visualization is the one on the left. The red circle in the skeleton on the right separates the node and edge areas. The nodes in the node part are depicted as a tree data structure in the skeleton. This tree (the node decomposition) is visualized in the Wheel View as nested circle segments (mark-area) that are placed in concentric rings representing the hierarchy, where each segment represents an entity—as can be seen on the left. The outermost ring represents the top-level entities of the hierarchy; in the concrete example the five outermost nodes in the skeleton. If an outer node has only one child in the tree, the two nodes are merged together in the same segment to save space. Each tree level with more than one node leads to another set of inner segments. The leaf nodes are in the innermost circle of the node part. They can be equally sized or their size could be proportional to an attribute of the leaves.

Unlike node-link diagrams for trees, the retinal encoding for nesting in Wheel Views is not truly enclosure, which would mean that one area is a part of another area. Instead, the idea of enclosure is created through neighboring and orientation in a beholder’s mind. While orientation is a visual facet mentioned by Card and Mackinlay [98] (cf. Fig. 2), the impression of nesting for this kind of visualization is rather a prediction of the gestalt theory [187], namely the laws of proximity and similarity. Gestalt theory postulates principles by which humans interpret visual sensation. The respective laws state that neighboring (law of proximity) and similar (law of similarity) objects are perceived as a group. The similarly shaped and neighbored circle segments are grouped together in the eye of the beholder and are interpreted as being nested. The principles of gestalt theory play an important role in how humans interpret visualization and must be considered for every kind of visualization, not just Wheel Views.

Because of the concentric layout of the nodes, the visualization leaves space in the middle that is used for edges representing the binary relation (retinal encoding: connection), which prevents edges from crossing nodes. To further increase readability, the edges are bundled hierarchically. Bundling means that edges from and to similar locations attract each other, analogously to a cable tie. Hierarchical bundling means that control points are added that act as the source of attracting forces for edges and the location of these control points is based on the hierarchical structure of the nodes. The control points are located in the edge part at invisible inner circles just above the hierarchical segments. They are made explicit in the skeleton right in Fig. 31 within the red circle marking the edge area. Each node level in the outer node tree introduces one inner circle for hierarchical control points. The outermost circle in the edge area is for the leaves and the innermost circle consists of the five outermost hierarchy nodes. Each control point is placed at the center atop its children. Edges that remain within a subtree of the node hierarchy are attracted only by the control point corresponding to their common ancestor. The bundling strength can be chosen to retain visual bundles while avoiding ambiguity, which would occur if edges could not be distinguished visually. This kind of hierarchical edge bundling creates edge paths with high locality and a more tidy visualization overall.

A Wheel View is found in the commercial tools Atomiq and SolidSDD [P43] and also discussed in the scientific clone-
visualisation literature [P21] [P19,P59,P3]. It shows clone relations between hierarchical code entities – generally above the statement level – as edges (N1, N3) for all clone classes or just the instances of one selected clone class (N2, N4) and relates these to a system’s hierarchy. The size and color gradient of segments representing leaf nodes may be chosen proportionally to the amount of the cloned code in (N6) or the size of the entity represented by that segment (N15), and the thickness of the edges may correspond to the number of lines that are shared between the two connected entities.

Hanjalic [P19] discusses the benefits of the Wheel View as part of SolidSDD for tracking clones as the source code evolves (N17, N18). Elements not present in the user-selected revision range are grayed out. To obtain a stable visualisation across multiple revisions, the union of all revisions’ files, scopes and clones are taken into account for the layout. Alsallah et al. [188] have proposed various improvements for the Wheel View in general (not specific to clone data). An empirical evaluation of the Wheel View outside clone visualisation was conducted by Pohl et al. [189]. Steinbeck [P49] utilises several techniques such as Voronoi Tree Maps, Heatmaps, Hierarchical Edge Bundles, and Bar Charts to enhance the capability of a Wheel View in analyzing and refactoring clones.

The Wheel View is a flexible and generic visualisation which can be used as alternative to existing tree visualisation techniques in cases where additional adjacency edges need to be drawn. By avoiding edges crossing nodes and applying edge bundling it reduces visual clutter for dense relations. The strength of the edge bundling can be controlled by the user, allowing her or him to skim from low-level, straight node-to-node connectivity (no bundling) to more high-level overviews where practically only single adjacency edges between parent nodes can be seen (extreme bundling) [161]. It is useful for quickly getting insights about clone pairs present in hierarchically organized systems. However, Holten [161] considers the bundle overlap in case of layouts with a large number of collinear nodes to be the biggest problem of hierarchical edge bundles. Moreover, it becomes difficult to show segments and curves on a huge data set with variant colors [P21] and the segments are often too small to put in label so that additional interaction (e.g., mouse hovering) is needed to find the name of the entity represented by a segment. Edge picking will be difficult, too, when edges are bundled strongly.

6.7.3. Duplication Web View (dww)

The Duplication Web View is another polymetric view by Rieger et al. [P46,P44] showing clone relations among files as a graph (cf. Fig. 32). The nodes of the graph represent all files in a system and are arranged in a circular layout (the x- and y-positions have otherwise no meaning). The inner part of the circle is used for the edges, which connects two files sharing common code (N1). The node width is a visual encoding of the lines of internally copied code. The height is uniform for all nodes and has no meaning. The thickness of an edge reflects the number of lines of copied code between the two connected source files (N13).

A Duplication Web View is used to show clone pairs at the file level. Its layout resembles the Wheel View, but has neither edge bundling nor a node hierarchy. Not showing the hierarchy has the advantage of being able to vary node widths along the x-axis to visually encode a clone metric. This extra degree of variation and the design decision to anchor an edge always at the center of a node’s width together, however, can lead to edges that are outside of the inner circle as can be observed in Fig. 32. This phenomenon may be considered peculiar, on one hand, but highlights, on the other hand, exceptional cases. A Wheel View could encode such a clone metric by the length of a segment representing a leaf in the innermost circle.

As the system size increases, it will be difficult to indicate the cloning information for all the files. Grouping of files into directories, modules, or subsystems can simplify the visualisation, with possible loss of accuracy because then several entities would be folded into one node. Similarly, the edges in the center of the view decrease readability when there exists a large number of clone relations between the nodes, but at least conveys the information that there exists a large amount of copy-and-paste programming in the system [P44,P46]. For graphs with many edges, edge bundling would be recommended. Overall, the information conveyed by the Duplication Web View might be better visualised by a Wheel View.

6.7.4. Dependency Browser (db)

The Dependency Browser – as part of SoftGUESS [P1] based on the general graph-exploration system GUESS [21] – is a node-link diagram showing relations between cloned code fragments and code entities such as methods, classes, or packages. An example is shown in Fig. 33. Code entities are visually encoded as square nodes and cloned code fragments as circle nodes. Edges represent relations and are drawn as straight lines connecting two nodes. An edge from a code entity e to a cloned code fragment f means that a copy of f is contained in e (N1, N2, N4), thus, a cloned code fragment is a representative for a clone class. An edge from a cloned code fragment f to a code entity e tells that here is a static dependency such as a call, inheritance, or field access contained in f relating to e. Thus, the code f depends upon e statically. Similarly, an edge between two code entities represents a static dependency, too. Only nodes with at least one clone are shown to reduce the size of the graph to the minimum required to investigate cloning. A force-directed layout is used to place the nodes. Incoming and outgoing edges for cloned code fragments are colored differently to better distinguish them visually. Each node can be colored or sized based on different properties (such as length and in-degree) (N5, N6 N13, N15). Through the containment and dependency edges, clone classes can be visually related.
to each other (N32), for instance, whether they occur in the same files or depend upon similar other code entities.

The **Dependency Browser** is structurally and visually similar to the **Clone Cohesion And Coupling Graph** discussed in Section 6.4.2. Both relate code entities to clone classes in a similar manner through a node-link diagram laid out by a force-directed layout. The addition of the **Dependency Browser** is that it overlays the clone information with dependency information, which gives a more complete context of static dependencies beyond just dependencies induced by copy & paste. Because of their similarity, the strengths and weaknesses we have discussed for the **Clone Cohesion And Coupling Graph** apply for the **Dependency Browser**, too. Visualizing graphs with many edges as a node-link diagram is a challenge, where the **Dependency Browser** deals with even larger graphs because of the lower level of granularity (methods, classes, packages versus subsystems) and the extra dependency edges.

### 6.7.5. **Clone Scatterplot (cs)**

The **Clone Scatterplot** (see Fig. 34) from Rieger at al.’s poly-metric clone views visualizes cloning between files and relates file size and clone rate. This visualization – similar to **Duplication Web View** (Section 6.7.3) – consists of nodes and edges indicating source files and clone relations, respectively. All nodes have the same size and shape. Edges represent external duplication between files (N1) and edge thickness indicates the number of lines of code shared between files as for the **Duplication Web View** (N13). Unlike **Duplication Web Views**, the x- and y-positions have meaning. As for **Dotplots** described in Section 6.7.1 the x- and y-axes are used to encode two attributes. The x-axis represents the size of the files in terms of lines of code (LOC), where smaller files are located towards left and large files are located towards right (N15). The y-axis encodes the amount of cloning in terms of lines of copied code (LCC), where nodes with less cloning are placed in the upper part of the view, while nodes containing more duplication are placed at the bottom (N6). If both axes are equally scaled, all nodes will be placed at or above the 45° diagonal, because \( \text{LOC} \geq \text{LCC} \) always holds for a file. Nodes that are closer to the 45° diagonal have a high clone rate. Nodes in the right lower corner are particularly interesting because they are large and their clone rate is high.

A **Clone Scatterplot** shows cloning at the file level. It is helpful in acquiring an overview of the system with respect to cloning and file size. It helps programmers in locating exceptional files with respect to clone rate and size (or any other code metric that is mapped onto the x-axis). It is simple, scalable, and customizable with other metrics. Yet, edges crossing other edges and nodes cause visual clutter and decrease readability. If there are dense clouds of nodes, picking a particular node to obtain more details may become difficult.

### 6.7.6. **Similarity Graph (sg)**

A **Similarity Graph** is a node-link diagram in which the nodes are laid out according to their similarity. The more similar two nodes, the closer they appear in the diagram. Often similarity is a function of the edges, that is, strongly connected nodes attract each other. Generally, force-directed layouts are used to generate those diagrams. There are different variants of **Similarity Graphs**, which differ in what nodes and edges represent and how similarity is defined. **Similarity Graphs** support the law of proximity of the gestalt theory, allowing a human beholder to spot related entities intuitively and quickly.

An example **Similarity Graph** by Yoshimura and Mibe [P60] is shown in Fig. 33. The nodes in their graph are files, where the area of nodes is proportional to their size in terms of lines of code and the edges indicate cloning across files. A force-directed layout is used to show clusters of files coupled through cloning. Files without externally copied code (nodes without edges) appear in gray. Otherwise, connected components (transitive closure of nodes reachable through edges) in this graph are identified as clusters in a preprocessing step. All nodes of the same connected component receive the same color. The same color can be reused for other connected components if they are not close by in the visualization. Because nodes of the same connected component appear together in the same color, edges are not actually drawn, which avoids the edge-crossing problem. In Fig. 35, clusters emerge that appear to be in a circle with a lot of unused space that are not further explained in the original paper. We speculate that these are byproducts of the force-directed layout based on the repulsion of nodes that are not connected.

Alalfi et al. [P2] use a **Similarity Graph** to visualize the clones detected by clone detector SIMONE for clones in models. They call their variant of a **Similarity Graph** SIMGraph. A **SIMGraph** visualizes clone pairs among subsystems in model-based software (see Fig. 36). The underlying tool to create the visualization is Gephi,7 an open-source program for visualizing and exploring graphs.

**SIMONE** detects clones in the modeling language Simulink, which is structured in models, which in turn consist of subsystems. Each subsystem is represented by a circular node in **SIMGraph** (retinal encoding: shape) whose area represents the size of the subsystem (N15). Each model and all its subsystems are represented by the same unique color. Cloning between subsystems is represented by an edge (N1) and its thickness shows the extent of cloning (N13). An edge inherits the color of its source node, although the selection of which end of an edge is its source or target node is somewhat arbitrary if the clone relation is symmetric.

7 https://gephi.org/.
By default, all subsystems of the same model are laid out in the same layer of a grid separate from subsystems of other models as shown in Fig. 36 on the left. A filter can be applied that leaves only those subsystems that contain at least one clone. The grid has no particular order, thus, the x- and y-positions have no meaning. This view shows pairs of subsystems sharing clones and can be used to compare modules and subsystems densely connected internally. Shapes of the nodes represent the clone type (1, 2, 3). Color may be chosen to represent the same file, directory, or community. The File-Community View is analogous to the Clone-Community View where the nodes represent files. An edge may express that two files are located in the same directory or share common code. The files are visualized by circles whose size represents the number of clones they contain. The darker an edge, the higher the frequency that the files were changed together. Nodes can be colored by the directory or community.

For this reason, SIMGraph offers also a force-directed layout by Fruchterman and Reingold [190] that visually clusters subsystems highly coupled by cloning as shown in Fig. 36 on the right. This visualization then becomes a true Similarity Graph, which allows one to spot sets of subsystems that are highly coupled through cloning. It tears subsystems of the same module spatially apart, yet the information to which module a subsystem belongs is still present in the color of the node.

Mondal et al. [P40, P41] proposed three other variants of a Similarity Graph: Clone-Community View, File-Community View, and Clone-Landscape View. The nodes in a Clone-Community View can be clone instances or clone classes as controlled by the user. If they represent clone instances, the edges may represent clones changed together or identify clone instances of the same clone class (N4), again controlled by the user. If they represent clone classes instead, edges connect clone classes having some of their clone instances in the same file. So-called communities are identified automatically and these are laid out by a force-directed layout. A community of a graph (or network) is a subset of nodes densely connected internally. Shapes of the nodes represent the clone type (1, 2, 3). Color may be chosen to represent the same file, directory, or community. The File-Community View is analogous to the Clone-Community View where the nodes represent files. An edge may express that two files are located in the same directory or share common code. The files are visualized by circles whose size represents the number of clones they contain. The darker an edge, the higher the frequency that the files were changed together. Nodes can be colored by the directory or community.

The third variant of a Similarity Graph proposed by Mondal et al. [P40, P41], Clone-Landscape View, is based on a graph of clone fragments and a similarity function for all nodes based on their distance in the file system (cf. Fig. 37): the closer they are in the file system, the more similar they are. A multi-dimensional scaling technique, analogous to force-directed layouts, is used to lay out the nodes. Thus, clone fragments closer together in the program hierarchy appear also closer in the visualization. As in the Similarity Graph variant by Yoshimura and Mibe [P60], the edges are not really drawn. The change rate of a clone fragment is visualized by a color gradient as in Heatmaps. The Clone-Landscape View allows one to spot frequently changed clones easily and also whether these changes occur in neighboring parts of the program hierarchy, although one cannot really decipher the hierarchy precisely.

A Similarity Graph is able to capture only a very limited number of levels of a hierarchy by design: the lower-level entities are represented by the nodes and additional higher levels must be represented by retinal encodings of the nodes such as texture, shape, or color as in SIMGraph. The edges are reserved for clone relations and cannot visualize nesting as in Hierarchical Tree Views. Neither can enclosure be used to express nesting because the x- and y-positions are determined to represent the similarity among the entities.
Using a layout based on similarity that puts more similar nodes together allows one to detect clusters of similar entities (such as clone instances, clone classes, and files) easily and intuitively. Node size, color, or texture as well as edge color, shape, and width can be used as retinal encodings of additional information. For instance, through the node colors in SIMGraph, one can immediately see whether all subsystems of a cluster come from the same module. Yet, if there are too many modules it becomes increasingly difficult for a human to distinguish the colors. In Yoshimura and Mibe's visualization the node color is a visual aspect redundant to the [260] clustering and just lets a beholder more easily identify the nodes of the same cluster. Similarity Graph may suffer from visual clutter through edge crossing. That is why some variants do not actually draw the edges. A Similarity Graph is very similar to the Dependency Browser, except that the latter shows additional types of static dependencies beyond cloning, that is, there are different types of edges.

6.7.7. Exploration Graph (eg)

The Exploration Graph by Liu et al. [P34] allows one to inspect the similarity among method clones. It was proposed specifically for the use case of code search, that is, finding similar methods for re-use, but may as well be used to visualize the results of a clone detector. The graph consists of nodes representing clone classes of type-1 and type-2 method clones. Edges connect similar clone classes. More precisely, given two nodes \( n_1 \) and \( n_3 \), there is an undirected edge \( \{n_1, n_2\} \) if \( n_1 \) contains a clone instance that is a type-3 clone of another clone instance in \( n_2 \). The difference among clone instances of different clone classes is measured based on the longest common subsequence (LCS) of the token sequences of the methods contained in those clone classes. Because all methods within the same clone class are type-1 or type-2 clones of each other, one representative can be randomly chosen for the comparison. This comparison is done for all pairs of clone classes, yielding a fully connected graph. A threshold is used to eliminate edges representing insufficient similarity. Further edges are removed, if they can be considered transitive similarity (details on that can be found in the original paper).

The Exploration Graph shows type-1 and type-2 clone classes and other such clone classes that are type-3 clone classes with respect to them – above a threshold and non-transitively – as a node-link diagram with a force-directed layout. An example is shown in Fig. 38. The visualization shows a center point (node 3 in Fig. 38) that is determined based on the distances among all clone classes. In clustering terminology, this is the centroid, that is, the node that has the overall shortest distance to all other nodes. The distance from one node \( n_1 \) to another node \( n_2 \) is calculated as the minimum sum over all paths from \( n_1 \) to \( n_2 \), where the distance for a single edge is the inverse of the LCS similarity represented by this edge. All nodes will be drawn in equal sizes, except for the center point, which is drawn larger to make it more prominent. A color gradient is used to encode the similarity of every node to the center point. The edges are labeled with the token sequences in which they differ.

The information conveyed by an Exploration Graph is quite different from other clone visualizations. The main focus here are clone classes and how they relate to each other (N32) in terms of differences (N10). Most other types of visualizations focus on the location of clones in a system or how the code entities are related to each other through cloning. The different focus of the Exploration Graph results from the original use case this visualization is supposed to support, namely, search for similar methods for re-use. In the area of maintenance, however, this visualization can be useful to identify clone instances that may need to be updated in a similar way (N36). The center point could then be the clone class containing the method that must be changed initially. All other instances of that clone class are type-1 or type-2 clones of the seed method and are, thus, the primary candidates for inspection. Then one could follow the edges guided by the color gradients. Thus, this visualization could provide a ranking (N11) when changes need to be made to a clone instance. Although a system may have many clones and, as a consequence, the Exploration Graph may have many nodes, one should expect that this graph is actually a large set of small fully connected components (subgraphs whose nodes are all reachable from each other). If only a single connected component is to be drawn, node-link diagrams should likely work well. Moreover, the nodes by themselves reduce the amount of information already by the fact that they summarize all type-1/type-2 clones into one single node.

6.7.8. Nested Graph (ng)

If a user should be able to drill down in a hierarchy of a system enriched by clone information, a Nested Graph may be useful. A Nested Graph is a node-link diagram with composite nodes that can be unfolded on demand. The Shrimp Views by Storey and Muller [191] are one of the first examples of this kind of graph in software engineering to show hierarchical dependency graphs in general. The academic tool CLIPS by Kapser and Godfrey [P33] has a Nested Graph view specifically for clone dependencies in hierarchical systems. Commercial tools such as the Axivion Suite [U2] offer similar visualizations for cloning (see Fig. 39).

A node of a Nested Graph can either be a composite node consisting of other nodes (e.g., subsystem, directory, package, class) or a leaf node (typically a unit of cloning; e.g., code fragment or method). Generally the type of node is visually encoded by shape and/or color. Composite nodes can be unfolded by a user, in which case their immediate children are shown, typically framed in a rectangle (retinal encoding: enclosure). If a composite node is folded again, all its visible descendants disappear. A node is visible only if all its ancestors are unfolded or if it is a root node. Edges represent clone relations among the nodes (retinal encoding: connection). There are two kinds of edges: base edges and lifted edges. Base edges represent the actual clone relation derived by some clone detector (N1). Other types of edges may as well represent other kinds of dependencies derived by a static analysis as for the Dependency Browser, in which case the type of an edge is often visually encoded by color. Lifted edges are

---

8 Given a composite node \( n \). We will use the term child for a node directly nested in \( n \); \( n \) is the parent of all its children. If a node is transatively nested in \( n \), we will use the term descendant; \( n \) is an ancestor of all its descendants.
Fig. 39. Nested Graph by Axivion Suite for Linux [U2]. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

artificially added by the visualization to guide the user in drilling down by summarizing the clone/dependency information of the content of a folded node that is currently hidden. There is a lifted edge \((n_1, n_2)\) if \(n_1\) has a transitive descendant \(n'_1\) and \(n_2\) has a transitive descendant \(n'_2\) and a base edge \((n'_1, n'_2)\) exists and \(n'_1\) or \(n'_2\) is currently hidden. Edges are visible only if both ends are visible. In CLICS [P33], the edges are annotated by the extent of cloning \((N_6, N_{13})\), more precisely, the number of base edges a lifted edge represents.

**Nested Graphs** allow one to drill down along the lines of Shneiderman’s information seeking mantra. One can start with an overview of a hierarchical system where only the top-level nodes are shown (where the initial layout may be decided by a force-directed layout) and then successively unfold nodes. The clone relations among currently hidden nodes are still indicated by lifted edges. As in most node-link diagrams, this kind of visualization may suffer from edge crossing. Bundling edges hierarchically as suggested by Holten [161] for graphs with composite nodes may help to some extent, but edges cannot be fully prevented from crossing nodes or other edges. A problem specific to Nested Graphs is the folding and unfolding of nodes as newly shown children may overlap with other nodes. Animated continuous zooming as suggested by Bartram et al. [192] and implemented by the Shrimp Views, for instance, may help to maintain the mental map of a human beholder [191]. Yet, if many nodes are unfolded in deep hierarchies with a high number of nodes, it is difficult to maintain a compact layout and large areas of unused space may result.

The x- and y-positions are generally chosen by an automated force-directed layout and have no meaning. Yet, the positioning of nodes could also be along two axes showing two different quantities along the lines of Rieger et al.’s Clone Scatterplot. The layout may also be created by a human if the nodes represent architectural components in a design diagram, in which case the positioning would carry semantics (e.g., architectural layering). Such box-and-arrow diagrams are often used to model static architectures. It would be worthwhile to re-use human layouts whenever possible rather than creating a layout fully automatic. A human uses additional semantics that are generally unavailable to an automated layout algorithm and also wants her or his mental map to be preserved.

The idea of node folding and lifted edges can also be – and indeed is – used in other kinds of diagrams for hierarchies with additional dependencies as, for instance, Tree Maps, Circle Packings, or Wheel Views.

6.8. Clone Refactoring Visualization View (crvw)

The Clone Refactoring Visualization View by Chen et al. [P7] visualizes refactoring edits and anomalies in the history of a clone class as a node-link diagram. It is used to highlight missed refactoring for inconsistently modified code. An algorithm determines all refactorings that took place across a given range of versions within methods containing clones. If the same refactoring was applied to all instances of a clone class, those methods have been changed consistently. Otherwise there exists at least one instance of the clone class, where a refactoring has been missed. Another static analysis then checks whether the refactoring could have been applied to that clone instance, that is, whether all preconditions of the refactoring are fulfilled. Both kinds of missed
refactorings are presented to the user by way of a node-link diagram.

Fig. 40 shows an example. The blue root node View represents a Java class. All other nodes represent methods of that class connected by solid edges to the class. The edge label of solid lines is the source line within the file containing the method. The node label for methods states the method name and all parameter types. Because this visualization is intended for Java and the return type cannot be overloaded in Java (it is omitted to save space), those labels are unique. As one can see in Fig. 40, however, there are two node pairs with the same label, namely, processKeyEvent and processKeyEventV2. These represent the same method, respectively, but in different versions, where the same refactorings have been applied. Because the refactorings have been applied to all of them, they are colored in green. The methods in which the refactorings have been missed are colored in red instead. Because that holds true over the whole period of time considered, they occur only once in the diagram as opposed to green nodes. A marker before the node label distinguishes methods with missed refactorings that could have been applied from those where the refactorings are not directly applicable because of violated preconditions.

The visualization depends upon the selected time slot, for which the refactorings are determined. The selection of the time slot determines which refactorings are to be considered as well as the clone instances to be considered. All red and green nodes contained instances of the same clone classes at the beginning of the selected time slot. At the end of the time slot, the methods – even the ones colored in green – are not necessarily clone free. The refactorings may have eliminated the instances of some clone classes but not necessarily all clone classes.

In addition to methods in which the same refactorings have been applied and the methods where those refactorings have been missed, the visualization shows a summary of what the refactorings modified and when. The yellow node is a method that was introduced by the Extract Method refactoring removing some of the clones. The dotted edges is a call relationship, where the label states the revision in which the call has been introduced for the first time. As one can see in Fig. 40, the Extract Method refactoring has been applied first in revision 7075 for method processKeyEvent. Later in revision 7076 other instances of the same clone class in processKeyEventV2 have been replaced by a call to that method, too.

This visualization not only shows the location of instances of a clone class for a given time slot (N2) but may also give a rough idea of how a clone has been changed (N19). Not all details of the changes are given but at least the details like the kind of refactorings that have been applied and whether there are new relations to other (extracted) methods are shown. Neither does this visualization capture all kinds of changes: it is limited to refactorings only. Stating changes in terms of refactorings – whenever possible – is more abstract and meaningful than just highlighting every single token that was modified as in the Code Inspection View. Because it shows refactorings that have removed clones through refactorings and then detects similar code fragments where the same refactorings could be applied too, it also conveys information on which remaining clones could be removed (N29) and how (N30) – yet again limited to code elimination by way of recognizable refactorings. The main focus of this visualization is to highlight which other instances of a clone class should be updated (N36) and how they can be updated (N37).

7. Discussion

In this section, we will provide statistics on publication activities in the research area of clone visualization (RQ1), identify gaps between visualizations, information needs and user goals (RQ5), present the state of empirical evaluations in clone visualization (RQ4), and discuss the status quo and challenges in industrial adoption.

7.1. Trends in publications

First, we provide some statistical information on the 68 papers covered in our survey. This summary answer our research question RQ1 about the publication activity in clone visualization.

Table 5 lists the distribution of publication venues of the papers covered. As Table 5 shows, the International Workshop on Software Clones (IWSC) is by far the most preferred publication forum for clone visualization, which comes as no surprise because it is specifically devoted to software clones. The International Conference on Software Engineering, which is arguably one of the most renowned conferences on general aspects of software engineering, ranks two in this list. This fact shows that the subject is not just a niche research area, which is further evidenced by the fact that the topic is prominently covered by several conferences in the area of software evolution, namely, the International Conference on Software Maintenance and Evolution (ICSME), IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), International Working Conference on Source Code Analysis and Manipulation (SCAM), and IEEE International Conference on Program Comprehension (ICPC).

That is expected as software clones are a maintenance issue and most general papers on software clones are published in those conferences, too. In case of SANER, one could as well add the two publications at the IEEE Working Conference on Reverse Engineering (WCRE) because WCRE is one of the two predecessors of SANER. Even the ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA) ranks high on this list. The subject is covered also in many renowned journals on software engineering, such as the IEEE Transactions on Software Engineering, and four
PhD dissertations and two Master’s theses, although generally as one among other topics on software clones. Included in this list are also conferences and journals specializing in information visualization, which shows that the topic is not only treated in software engineering, but has also arrived in information visualization research in general. The length of the list of publication forums in itself shows how widespread clone visualization is dealt with in computer science.

One must note, though, that most of the papers we included in our survey do not focus on visualization solely, but rather treat the subject in the context of detecting clones or studying the characteristics of clones. The upper chart in Fig. 45 gives a summary. Out of the 68 papers, there are 27 papers that have visualization as their only focus point, that is, their only subject is to introduce or evaluate a clone visualization—often mentioning the term visualization explicitly in their title. There are 13 papers that deal with visualization as one of their primary focus points. That is, these papers present a visualization in great detail and in a substantial part of the presented material or evaluate a visualization—among other things equally well treated. All other papers covered in our survey use a visualization to show data gathered by some kind of clone analysis, but this analysis is their primary concern and they mention the visualization just in passing; neither do they evaluate the visualization in any form. Thus, visualization is only a secondary focus point. For instance, de Wit et al.’s paper introduces CloneBoard that tracks clones and their live changes in the IDE. They briefly mention that clones are marked by Bar Maps, but the paper is almost entirely dedicated to the level of support CloneBoard can offer with respect to consistent changes assessed by a controlled experiment [P14], where the experiment does not specifically evaluate how the location of clones is visualized. In summary, 41% of the papers in scope mention some kind of visualization in passing but do not have visualization among their primary focus points and 40% of the papers have visualization as their primary and only topic. The remaining 19% have multiple focus points and visualization is only one of them.

Fig. 42 displays the number of these papers published per year between 1992 and 2019, including Master’s theses and Ph.D. dissertations (Master’s theses are included only if they have been cited in a publication in a workshop, conference, or journal). First, the figure shows that visualization has been a topic from the very beginning of research on software clones. Brenda Baker [P4] is a pioneer in detecting software clones. In her first paper on clone detection she used Dotplots to visualize clones as early as 1992 [P4]. In 2002, the number of papers starts to grow and reaches a first peak in 2011. Interestingly, after that peak the number of papers declined drastically in 2013, followed by two years with a high number of publications again. In 2016, the number dropped to one paper again and since then the publication rate resembles the period between 2002–2010. We do not have an explanation for the two drops at 2013 and 2016, respectively. In 2011 and 2012, there were multiple papers published at IWSC on clone visualization contributing to the high number, while there was none at IWSC 2013. Possibly, this community felt that they should move to other aspects of cloning. In 2014 again, no visualization paper was published at IWSC. All seven papers published in 2013 appeared at other conferences, many of them as full papers. So the subject spread to other venues and often reached a higher level of maturity (workshop versus conference). This trend continued in 2015 where only one clone-visualization was published at IWSC. The interest cooled down again in 2016 where only one short paper was published at ICSE [P39], which treated the subject not even as one of its primary focus points. As in many areas, we can observe a cyclical fluctuation of interest in the research community regarding clone visualization. At the time of writing this paper (July 2019), already three papers were published on this subject, so quite likely the average number of 3–4 publications per year since 2002 will be continued.

Fig. 41 gives a timeline when the papers appeared in those publication forums. In the initial period between 1992 and 2000, clone visualization was presented at conferences and journals which have not had any other paper on this subject after that. WCRE/SANER and ICSME, both conferences on software evolution, are an exception. After that the subject was covered by other conferences and journals with more sustainable publication rates. In the last decade, these conferences and journals with multiple publications are from the domain of software engineering in general, software evolution in particular, and information visualization (most notably: IWSC, VISSOFT, ICSE, and SANER). ICPC – the conference on program comprehension – has had only two publications on this subject in its entire history, which is surprising as visualization is meant to support comprehension.

Table 5: Distribution of publications (note: SANER is a successor to CSMR and WCRE).

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Venue</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICSE</td>
<td>International Conference on Software Engineering</td>
<td>8</td>
</tr>
<tr>
<td>IWSC</td>
<td>International Workshop on Software Clones</td>
<td>8</td>
</tr>
<tr>
<td>ICSME</td>
<td>IEEE International Conference on Software Maintenance and Evolution</td>
<td>4</td>
</tr>
<tr>
<td>PHD</td>
<td>Ph Dissertation</td>
<td>4</td>
</tr>
<tr>
<td>SANER</td>
<td>IEEE International Conference on Software Analysis</td>
<td>4</td>
</tr>
<tr>
<td>OOPSLA</td>
<td>ACM SIGPLAN Conference on Object-Oriented Programming</td>
<td>3</td>
</tr>
<tr>
<td>APSEC</td>
<td>Asia-Pacific Software Engineering Conference</td>
<td>3</td>
</tr>
<tr>
<td>ICPC</td>
<td>IEEE International Conference on Program Comprehension</td>
<td>2</td>
</tr>
<tr>
<td>MASTER</td>
<td>Master’s Thesis</td>
<td>2</td>
</tr>
<tr>
<td>SCAM</td>
<td>International Working Conference on Source Code Analysis and Manipulation</td>
<td>2</td>
</tr>
<tr>
<td>TSE</td>
<td>IEEE Transactions on Software Engineering</td>
<td>2</td>
</tr>
<tr>
<td>VISSOFT</td>
<td>Working Conference on Software Visualization</td>
<td>2</td>
</tr>
<tr>
<td>WCSE</td>
<td>IEEE Working Conference on Reverse Engineering</td>
<td>2</td>
</tr>
<tr>
<td>ACSAC</td>
<td>Annual Computer Security Applications Conference</td>
<td>1</td>
</tr>
<tr>
<td>CASCON</td>
<td>Conference of the Centre for Advanced Studies on Collaborative Research</td>
<td>1</td>
</tr>
<tr>
<td>COMPASC</td>
<td>Annual Computer and Applications Conference</td>
<td>1</td>
</tr>
<tr>
<td>ECOOP</td>
<td>European Conference on Object-Oriented Programming</td>
<td>1</td>
</tr>
<tr>
<td>EIT</td>
<td>IEEE International Conference on Electro/Information Technology</td>
<td>1</td>
</tr>
<tr>
<td>ESMFA</td>
<td>Modelling Foundations and Applications</td>
<td>1</td>
</tr>
<tr>
<td>ICCS</td>
<td>International Conference on Computer Science and Software Engineering</td>
<td>1</td>
</tr>
<tr>
<td>IPSJ</td>
<td>IPSJ SIG Notes</td>
<td>1</td>
</tr>
<tr>
<td>IV</td>
<td>International Conference on Information Visualisation</td>
<td>1</td>
</tr>
<tr>
<td>JCGS</td>
<td>Journal of Computational and Graphical Statistics</td>
<td>1</td>
</tr>
<tr>
<td>JSEP</td>
<td>Journal of Software: Evolution and Process</td>
<td>1</td>
</tr>
<tr>
<td>JUICS</td>
<td>Journal of Universal Computer Science</td>
<td>1</td>
</tr>
<tr>
<td>SCP</td>
<td>Journal Science of Computer Programming</td>
<td>1</td>
</tr>
<tr>
<td>SEA</td>
<td>IASTED International Conference on Software Engineering and Applications</td>
<td>1</td>
</tr>
<tr>
<td>SESE</td>
<td>International Symposium on Empirical Software Engineering</td>
<td>1</td>
</tr>
<tr>
<td>SMSM</td>
<td>IEEE Symposium on Software Metrics</td>
<td>1</td>
</tr>
<tr>
<td>SYRCOSE</td>
<td>Proceedings of the Spring/Summer Young</td>
<td>1</td>
</tr>
<tr>
<td>TAPOS</td>
<td>Theory and Practice of Object Systems</td>
<td>1</td>
</tr>
<tr>
<td>VI</td>
<td>Visual Informatics</td>
<td>1</td>
</tr>
<tr>
<td>VLHCC</td>
<td>Symposium on Visual Languages and Human-Centric Computing</td>
<td>1</td>
</tr>
<tr>
<td>WISDMP</td>
<td>International Workshop on Innovative Software Development Methodologies and Practices</td>
<td>1</td>
</tr>
</tbody>
</table>
Similarly, VISSOFT – a conference on software visualization – has had only two papers on clone visualization and its last one in 2014. Next we will look into the main contributors of these publications and how their contributions changed over time. As Fig. 43 shows for all authors with at least two publications on clone visualization the years in which they published their papers. The number of dots per line represents the contribution overall. Among the top-most contributors to clone visualization are mostly researchers from the Japanese clone community, where Katsuro Inoue contributed seven papers, Shinji Kusumoto and Yoshiki Higo have published six papers, and Toshihiro Kamiya five papers. All of these are originally from the University of Osaka and all of these have shown a sustainable contribution. Chanchal Roy has six, Kevin Schneider has five, and Matthias Rieger and Robert Tairas have each contributed four papers, where Roy and Schneider are co-authors at the University of Saskatchewan with continuous research in this area since 2011. The Japanese research community and the group at the University of Saskatchewan are also strong contributors to other areas of software clones. Rieger has published his papers in the period of his Ph.D. research but then left to industry. Tairas – although still in academia – published his last paper on clone visualization in 2011. His research interest has shifted to other aspects of cloning. As Fig. 43 shows there are many newcomers on this subjects bringing in new ideas. What seems to be missing, however, are more research groups beyond the groups at Osaka and Saskatchewan that accept clone visualization as one of their major and sustainable research challenge. Moreover, both groups started their research originally on clone detection and only later stepped into visualization. Complementary to that, faster progress could be achieved if research groups from the domain of software visualization or information visualization in general would devote themselves to clone visualization bringing in their rich expertise in visualization. Alexandru Telea, for instance, is well known in software visualization in general and has contributed two papers on clone visualization in particular, but both appeared in 2014 and since then no new contribution followed. If researchers who may have a lot of expertise in clone detection, but little expertise in visualization, are the only ones working on clone visualization, the danger is that existing knowledge is ignored and wheels are reinvented. There are many surveys on different types of visualization, such as the one by Herman et al. [193] on graph visualization and navigation, summarizing the body of knowledge in information visualization on this topic, which are not cited in papers on clone visualization and, hence, appear to be not known, although they are highly relevant.

7.2. Coverage of information needs and user goals

In Section 4, we have presented user goals related to software clones and their information needs. In the previous Section 6, we have summarized different kinds of visualization that were proposed to help users in comprehending software-clone data. There we have discussed their inherent strengths and limitations resulting from their way to encode data visually. These aspects were related to how they present data. In addition, we have listed what kind of information is shown by each type of visualization and, in consequence, what information needs are addressed. Thus, we also assessed those visualizations in terms of what they convey. In this section, we bring everything together. We relate clone visualizations to the user goals they support via the information needs they address. This mapping allows us to assess the extent of available visualization supporting information needs and user goals and to identify information needs of a user goal that are currently not addressed by any of the current visualizations. This section provides answers to our research question RQ5.

Fig. 44 summarizes which visualization supports which information needs. It is presented as a Dotplot analogous to the one presented in Fig. 1 for the relation between user goals and information needs discussed in Section 4.12. The visualizations
are listed as rows and the information needs in columns in abbreviated form. A colored cell indicates that a visualization represented by the row supports an information need represented by the column of the cell. The number after the slash of each row and column label indicates the number of colored cells in the row or column, respectively. As in Fig. 1, we clustered both visualizations and information needs, respectively, by hierarchical agglomerative clustering (HAC) using Jaccard’s coefficient and unweighted average linkage (see Section 4.12 for details). The resulting dendrograms are shown above and left from the matrix.

7.2.1. Information needs without visualization

First of all, we note that the following information needs are not listed in the matrix in Fig. 44: N22, N23, N24, N25, N26, N27, N28, N31, N33, and N35. These are information needs for which we identified no visualization proposed in the literature. For some of these, the knowledge or analyses do not exist to gather the data that could be visualized. N22, to begin with, asks for the effects of clones on quality and maintenance costs. While there are many studies on this subject for particular aspects – some of which contradicting each other – there is not enough empirical data to actually quantify these effects reliably. Although Juergens and Deissenboeck [194] have proposed an analytical cost model for clones, the various parameters of this model have not been determined or calibrated using empirical data. If we had this kind of information, visualizing should be straightforward as costs can be expressed as a quantitative measure for which we have presented various visualizations in our survey. The same holds for the costs and benefits of removing and refactoring clones, N28 and N31. This subject is not well understood—neither in cloning nor in refactoring in general. Yet, the similar, although more code-oriented information needs regarding the amount of code that can be saved if all clones or all instances of a particular clone class are removed, N26 and N27, should be computable as a function of the volume of the redundant code and the overhead of the refactoring used to eliminate the code. For instance, the overhead of Guard Method, would be the method signature for the newly generated method and its calls and the savings would be the lines of code that are replaced by the method call. As these are quantified attributes, classical means of visualization for such kinds of attributes may be used or existing clone visualizations could be extended through suitable retinal encodings such as color gradient or size.

For other information needs missing in Fig. 44, there are already existing analyses to gather the necessary information. Who has checked in or altered a clone can be derived from the version control system (VCS). A VCS reports the changes typically at the level of source lines. Harder [53] has refined that even to the level of cloned tokens. Hence, the ownership of clones, N23, can be derived as long as one has a VCS and one can rely on the metadata data checked in. One of the first use cases of SeeSoft was in fact to show code ownership. Other researchers have proposed alternative kinds of visualizations for code ownership since then, for instance, Seeberger et al. [195] and D’Ambros et al. [196], that have not been used for clones yet. The existing kinds of clone visualization could also be extended by overlaying them with color, heatmaps, texture, or form to encode authorship, but there may also be a need to present more elaborated information regarding ownership (such as degree of ownership, frequency of
changes, and extent of change) where more sophisticated types of visualization in their own right may be required.

Similarly, there are different automated analyses to gather the license data of a file, required for information need N24 [197–200]. The license of a piece of code could be visualized by the same means as code ownership. In fact, visualizing license information may be even simpler, because even though licenses change, too, they change generally less frequently than code contributions. Furthermore, there are typically fewer licenses than developers in large projects and the level of granularity is higher because licenses typically relate to a whole file whereas code ownership to lines or even tokens in many cases.

If one had a visualization for licenses, it could be re-used to visualize also the privacy of code, asked by information need N25. While licenses are multi-valued, privacy is rather binary: code should be kept private or not. While code ownership and licenses can be gathered automatically to a large extent, whether code should be kept private is often not annotated in the code. Yet, if this meta data were provided, e.g., by comments in the code or in some configuration files, it could be processed automatically and then visualized with the same means as code ownership and code privacy.

Information need N33 calls for links between clones and formal and non-formal system artifacts other than code such as architecture documentation or requirement specifications. The mapping among code and these related artifacts is known as traceability links [201–203]. Many researchers in reverse engineering have proposed techniques to recover these links [204–208]. If those links exist, they can be visualized by different means. The underlying structure of these traceability data is a bipartite graph connecting code to higher-level artifacts such as architectural concepts or requirements. So, all clone visualizations that we described for bipartite graphs in Section 6.4 may be candidates for adaptation.

Researchers in the area of traceability links in general have already proposed different types of visualization for the same purpose that could be used, too, and which in fact use very similar visual attributes as those used for visualizing clone data. For instance, Chen et al. [209] proposed to use two connected Tree Maps, one for the code hierarchy and one for a document hierarchy (chapters, sections, paragraphs). If a user selects a rectangle in either of the two Tree Maps, the rectangles connected via traceability links in the other Tree Map are highlighted. They also proposed to use bundled edges to see all traceability links between the two Tree Maps at once. As an alternative to Tree Maps, they proposed to use two variants of Hierarchical Tree Views. One variant represents both hierarchies (code and artifact) as separate trees and connects the leaves again with bundled edges showing the traceability links. The other variant visualizes only one of the two hierarchies as a tree, say T1, and adds the leaves of the other hierarchy, T2, as another layer of nodes to this tree. This way, the leaves of T2 become the new leaves of T1 and the original leaves of T1 become inner nodes. The original leaves of T1 are connected by straight edges to the leaves of T2 representing the traceability links, which may turn the tree into a directed acyclic graph.

Marcus et al. [210] have proposed to use two connected Navigation Tree Views, one for the hierarchy of code elements and one for the hierarchy of artifacts as a plug-in to Eclipse. If a user selects an element in either of those Navigation Tree Views, the corresponding elements in the other are highlighted. Two Eclipse property views give additional details on the selected elements and their traceability links and the browsing history.

In addition to these, a so called link area is presented, which shows the links for a specific source or target (one of them is selected in the Navigation Tree Views). The links are grouped into categories, based on the chosen view, shown as rectangles. In case of artifacts, the categories represent different domains of artifacts such as user manual, developer guide, test specification, or resource data. In case of code elements, the system decomposition into packages, files, and code entities is shown. The hierarchy of those categories are represented as nested rectangles. The innermost rectangle corresponds to the source or target link, respectively, for the selection. The innermost rectangles can be colored based on different link attributes. Merten et al. [211] have proposed to use a Wheel View and Sunbursts to visualize traceability links instead. As one can see in all these examples, types of visualization are used for traceability links that are very similar to the ones we identified in our survey for clone visualization. Hence, it should not be difficult to find a suitable visualization for traceability links from clones onto other artifacts.

Li and Maalej [212] have compared different means to visualize traceability links (matrix, graph, list, hyperlinks) in a user study in which participants were asked for their experiences in using those types of visualization. Their insights could be used in the selection and design of a suitable visualization for traceability between clones and other artifacts. Winkler [213] has listed several user goals and user roles for visualizations of traceability links in general as well as different means to access the information (reporting, searching, browsing) and summarized visualizations that were already proposed. Those aspects define a framework in which a visualization can be classified and assessed, very similar to our approach. He concludes that future visualizations of traceability visualization should focus more on users and tasks. We strongly believe that the analogous conclusion can be drawn for clone visualization.

The final missing information need N35 regarding the reasons for clones is much more problematic. First, the reasons for clones are still an open research question. Furthermore, it is unclear whether we can ever find any reliable indicators for possible reasons in the code or its evolution that we can detect and then visualize. Similarly to design decisions, we can find the final outcome in the code but not the rationale that has led to this outcome because that is hardly expressed in code. It may be searched for in comments and other meta information, but this information may be incomplete so that the original developers may need to be asked. The level of support visualization can provide for the information need N35 is likely limited to computable information needs regarding size, location, authorship, difference, and evolution of clones. These data can be visualized by the visualization and the process of making sense then be left to human interpretation.

7.2.2. Information needs supported by visualization

Now we turn to the information needs that are supported by at least one visualization as listed in Fig. 44. The dendrogram on the left clusters visualizations with regard to the information needs they address and, hence, lets us identify possible redundancies. For instance, Parallel Coordinate View (pcv) completely subsumes all information needs Line Chart (lc) is able to answer. Similarly, the Exploration Graph (eg) answers the information need also addressed by Relative Similarity Plot (rsp) but three additional information needs.

We note, however, that this clustering is based on the information that is conveyed by a visualization, in other words, based on what is shown rather than how it is shown. Even though there may be alternative visualizations showing the same information, one may show this information better than others. For instance, Tree Maps (tm) and Clone Scatterplots (cs) address the identical set of information needs, but, in very different ways. Similarity Graph (sg) and Nested Graph (ng) are similar to those two, too. The information needs for each are subsumed by either one of those.
To judge the suitability of alternative visualizations for a similar set of information needs, empirical comparisons are necessary through controlled experiments. In Section 7.3, we will summarize existing empirical studies assessing clone visualizations. Fig. 44 helps to identify candidates to be compared by future experiments. Until such experiments are conducted, one can at least use our assessment of the inherent strengths and weaknesses of each candidate visualization presented in Section 6.

Fig. 44 shows also how generally a visualization may be used: The more colored cells in a row, the more information needs are satisfied by a visualization. The number after the slash of each label counts the dots of each row or column, respectively. The Tag Cloud View (tcv), for instance, is very specific as it addresses only one information need, namely N34 on the underlying abstraction of a clone. The visualizations that can be used most frequently for various information needs are Dotplot (dp), Seesoft (ss), Bar Map (bm), and Parallel Coordinate View (pcv). Parallel Coordinate Views are a classical and general means to visualize quantitative attributes. These kinds of attributes can also be expressed by Bar Charts (bc), Pie Chart (pc), and Line Chart (lc), which are in fact clustered along with Parallel Coordinate Views in the dendrogram. Because such quantitative attributes occur in many questions, the visualizations in this cluster have a high count. Bar Maps are also frequently helpful because many information needs call for the exact locations of clones in the code, which can be conveniently marked by Bar Maps.

There is also a large overlap of the frequently helpful Dotplot (dp) and Seesoft (ss) as the dendrogram in Fig. 44 highlights. Seesoft is more code and location oriented and, hence, better shows how clones have moved or spread in the code (N18, N21). Because Dotplots offer a direct comparison between two elements put onto the different axes, they are better suited to show their differences (N19, N9). Other than that, they address the same information needs. An empirical comparison of those two would be valuable, not only because of their large overlap, but also because they are also widely applicable. Thus we can generalize, Fig. 44 helps to identify in particular those similar subsets of visualizations to be compared empirically that not only have a large overlap in terms of their addressed information needs, but can be used also for many information needs. In other words, Fig. 44 may be used to set priorities for similar widely helpful visualizations to be compared by controlled experiments.

7.2.3. User goals and their visualizations

Fig. 44 provides a mapping of visualizations onto information needs and Fig. 1 a mapping of information needs onto user goals. The composition of both mappings allows us to assess to which extent user goals are supported by existing visualizations at present. This information is added as heatmaps in Fig. 1 in terms of two metrics: visuals and support.

Let $G$ be the set of user goals, $I$ the set of information needs, and $V$ the set of visualizations. Fig. 1 defines a mapping $I : G \mapsto I$ enumerating all information needs required to reach a given user goal. Similarly, Fig. 44 introduces a mapping $V : I \mapsto V$ listing all visualizations supporting a given information need.

Metric visuals is the absolute number of visualizations that exist for an information need, $i \in I$: $\text{visuals}(i) = |V(i)|$. The metric yields the number of alternative visualizations for a particular information need and is represented as a heatmap in Fig. 1 with four different categories from 0 visualizations in red to information needs with 15 or more visualizations in dark green. As we have already discussed in Section 7.2.1, there are 10 information needs that do not have any visualization; all of them are marked in red. The information needs in the top-most category are N1 (19), N6 (17), and N13 (15), which are all very fundamental information needs. The precise value of visuals for each information need can also be found in the row label after the slash in Fig. 44.

Metric support is the fraction of information needs of a goal, $g \in G$, among all its needs that have at least one visualization:

$$\text{support}(g) = \frac{|\{i | i \in I(g) \land \forall(i) \neq \emptyset\}|}{|I(g)|}$$

This metric helps us to discover gaps needed to be closed by additional development or even research. Again, we use a heatmap of four categories in Fig. 1, this time using a color gradient of blue–green, to visualize the support. The precise numbers can also be found in Table 6. As Table 6 shows, eleven user goals have a complete set of visualizations for their information needs and only five user goals have less than 75% support. The one with the least support is Productivity Measurement, which has, however, only two information needs, one of which is N26 that has no associated visualization yet. We already discussed all information needs currently without visualization in the previous section and argued that most of them could be supported by minor adaptations of existing visualizations. N26, asking for the amount of code that can be eliminated when clones are removed, falls into this category. The more problematic information needs are N22 (effects of clones onto maintenance), N28 (reduction of maintenance costs when clones are removed), N31 (costs and risks of removing clones), and N35 (reasons for a clone). For all of these, there is not enough knowledge to gather this information yet, which affects all user goals where costs and benefits must be weighed to make a decision how to handle and to avoid clones. The answer to N22, N28, and N31 can be expressed by costs, which is a quantitative measure and as such can be visualized rather simply. The reasons for clones, N35, on the other hand, is probably better expressed in words. Hence, the main focus of research to close the gap for all user goals having those four information needs is to better understand the interrelations and to gather empirical evidence for cost models. Visualization is only a secondary emphasis.

7.3. The state of empirical evaluation

In this section, we summarize the state of empirical evaluations of clone visualizations (RQ4). We classify each publication considered for our survey into one of the following categories based on the research method that is applied. This classification

<table>
<thead>
<tr>
<th>User goal</th>
<th>Support in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Productivity Measurement</td>
<td>50</td>
</tr>
<tr>
<td>Code Leakage</td>
<td>67</td>
</tr>
<tr>
<td>Refactoring</td>
<td>67</td>
</tr>
<tr>
<td>Library Candidates</td>
<td>68</td>
</tr>
<tr>
<td>Aspect Mining</td>
<td>74</td>
</tr>
<tr>
<td>Quality Improvement</td>
<td>77</td>
</tr>
<tr>
<td>PL Migration</td>
<td>77</td>
</tr>
<tr>
<td>Quality Assessment</td>
<td>79</td>
</tr>
<tr>
<td>Due Diligence</td>
<td>83</td>
</tr>
<tr>
<td>Code Compaction</td>
<td>85</td>
</tr>
<tr>
<td>Design Recovery</td>
<td>86</td>
</tr>
<tr>
<td>Evolution</td>
<td>91</td>
</tr>
<tr>
<td>Provenance Analysis</td>
<td>91</td>
</tr>
<tr>
<td>Testing</td>
<td>91</td>
</tr>
<tr>
<td>Bug Detection</td>
<td>100</td>
</tr>
<tr>
<td>Clone Evolution</td>
<td>100</td>
</tr>
<tr>
<td>Domain Analysis</td>
<td>100</td>
</tr>
<tr>
<td>Duplicated Bugs</td>
<td>100</td>
</tr>
<tr>
<td>Evolutionary Mapping</td>
<td>100</td>
</tr>
<tr>
<td>Feature Location</td>
<td>100</td>
</tr>
<tr>
<td>File Mapping</td>
<td>100</td>
</tr>
<tr>
<td>Malicious Software Detection</td>
<td>100</td>
</tr>
<tr>
<td>44th revision</td>
<td>100</td>
</tr>
<tr>
<td>Merging</td>
<td>100</td>
</tr>
<tr>
<td>Program-Concept Assignment</td>
<td>100</td>
</tr>
</tbody>
</table>
A certain set of information needs, visualization addresses well-defined problems along the lines of analysis are relatively mature research areas. Papers on clone are nopapers with exploratory research. Visualization and clone tests), solely qualitative (e.g., user studies with questionnaires or controlled experiments with precise measures and statistical we separated empirical papers further into solely quantitative (only usagescenariosoranecdotalevidenceatmost). All other papers provide constructive research with only a limited or even no evaluation. As a matter of fact, 25 papers in our survey do not have any kind of evaluation of the clone visualization. The earliest paper of the seven empirical papers appeared in 2004 [P53], the two similar publications [P14] and [P13] in 2009, three more in 2015 [P42,P56,P55], and the three most recent ones in 2019 [P50,P40] and [P41] (where – once again – [P40] and [P41] report on the same study). Thus, more thorough empirical evaluations in clone visualization were conducted rather recently given the fact that the first paper on clone visualization was published already in 1992. Moreover, there does not seem to be a trend towards more continuously conducted empirical research given the gaps in between those publication dates.

If a paper has some form of evaluation of its visualization, we also state who has actually used the visualization in the evaluation. We distinguish the following four classes representing different levels of representativeness of the intended users:

- **none**: there was no evaluation with humans
- **personal**: the authors themselves used the visualization
- **students**: students used the visualization
- **professionals**: professional developers used the visualization

We want to point out that this classification relates only to the evaluation of the visualization presented. There may be papers with multiple focus points and only one of them is visualization. These papers may provide a thorough evaluation of these other focus points. If they, however, do not have an evaluation of the visualization – which is the research theme of our survey –, we will group them into category none. Furthermore, the classes of study participants above are not disjoint. There are papers in which both students and professionals used a visualization, in which case we tag them with both classes.

The lower chart of Fig. 45 provides the numbers. If there was an evaluation at all, in most cases the visualization was used only by the authors themselves. In 17 cases (more precisely, 15 if we count studies and not publications), the visualization was used by other kinds of users, where the distribution of studies where only students, only professionals, and combinations thereof used the visualization are well balanced.

The references in part Clone-Visualization References in the bibliography listing all papers of our literature survey are annotated by their visualizations, focus point, research method, and participants. Details of each paper can be found there. Here we provide only a summary. Overall, the majority of papers follow constructive research. Among these there are many papers with no evaluation at all. This finding is in line with similar findings by Merino et al. [214], who observed that 62% of 367 full papers published in the SOFTVIS/VISSOFT conferences (two conferences on software visualization) they took a look at lack a strong evaluation (only usage scenarios or anecdotal evidence at most).

The lack of evaluation in the papers of our survey, however, must be relativized somewhat because our survey subsumes all kinds of publication venues including workshops where often early ideas are presented. Moreover, clone-visualization researchers actually do not necessarily need to conduct empirical studies for all types of clone visualization. Researchers in information visualization have already conducted empirical studies.

![Fig. 45. Statistics on Clone-Visualization Evaluations.](image-url)
for visualizations that were also adopted for visualizing clones, e.g., [189,215–220]. If the settings of those studies can be generalized to software clones, we can safely build on their results. It is, however, surprising that those evaluations in information visualization in general are not cited in the clone-visualization community, which may be interpreted that this community is not aware of them. As already discussed above, many authors of clone-visualization papers come originally from the area of clone detection and may not be aware of results obtained in information visualization. These authors may benefit from the many points to the relevant literature in software visualization we give in this paper. They may also be advised to work closer together with visualization experts.

In future work researchers should try to emphasize more empirical evaluations preferably with professional developers for all types of new visualizations as well as visualizations that have been evaluated already; however, in settings that may be distinct from those in software-clone management or for which not only the kind of presentation but also the content matters. Indeed, some of the visualizations described in our survey may not be very innovative with regard to visualization per se, but they are innovative with respect to the information they convey. For instance, the Exploration Graph and Content–Context Diagram are just node-link diagrams. Yet, they stick out by the kind of information they offer. Visualization can hardly be evaluated in the abstract; content and presentation are inevitably intertwined. Hence, whether results obtained in other domains can truly be transferred to software clones must be carefully assessed.

If clone-visualization researchers want to conduct empirical studies they can again learn from other researchers in visualization in general. There is already a large body of ideas and knowledge on how to evaluate visualization techniques [103, 221–233].

Because the evaluation of a visualization always depends upon the kind of tasks it is to support, one must select tasks that are representative. Many diverse tasks have already been suggested in the software visualization and comprehension evaluation literature [234–239]. These tasks, however, are relatively general and need to be adjusted and supplemented for more specific clone-visualization evaluations. Here again, our framework connecting visualization, information needs, and user goals may be helpful—and it has in fact already been used for that purpose [420]. The information needs we listed could be refined by even more concrete tasks related to managing clones along the lines of more general comprehension tasks. More user observation of development practice and discussion in the research community are needed to define a set of meaningful tasks for such evaluations.

The currently used types of empirical evaluations subsume usability studies, case studies, and controlled experiments. They are all important but fall short of evaluating the long-term effects of a visualization. It takes time to learn how to “read” a visualization and to make optimal use of it. It does make a difference whether a developer looks at data of an unknown system in a laboratory setting or at data of his or her own system in the normal working environment over a long time, where a developer’s decision has true consequences. For this reason, we agree with Pleasant [226] and strongly believe that we also need long-term studies to adequately evaluate visualizations. In addition, we fully agree with Buckley [240], too, who argues that we need to carefully elicit the precise requirements from users first, before we start to design any kind of visualization. These requirements must be re-assessed periodically in long-term studies because they will change. This way clone visualizations will evolve over time. Action research is a research approach along these lines [241–243]. It refers to a wide variety of evaluative, investigative, and analytical research methods designed not only to diagnose problems or weaknesses but also to overcome these. Action research in clone visualization would be a participatory inquiry process balancing problem solving by means of visualization implemented in a collaborative context with data-driven collaborative analysis. Researchers and practitioners should work together to adopt, adapt, and design visualizations in real projects and over a longer period of time.

7.4. Challenges in industrial adoption

Clone detection has already moved from research into practice and we begin to see clone visualizations in practice, too. Here we discuss the status quo of industrial adoption and challenges to be overcome.

Above we have presented visualizations not only from academia but also from different commercial tools such as Atomiq [U1], Axivion Bauhaus Suite [U2], Clone Doctor by Semantic Designs [U4], JetBrains’s dupFinder [U10], Microsoft Visual Studio [U11], Pattern Insight [U12], Solid Source [U13], SonarQube [U14], and Teamscale [U15]. Most of these tools, however, use only some of the more basic types of clone visualization. Overall, as yet, the more advanced visualizations have not been widely adopted by the software industry. There can be many reasons for that. A tool vendor must offer both a clone detector and visualizer. Detecting relevant clones is still an ongoing research problem. If a vendor’s clone detector reports incomplete or false positive clones, the visualization can hardly be useful. Thus, industrial adoption is not just a matter of the visualization per se.

Even though most developers are aware of the problems of copying and pasting, handling clones is not among their top priorities. Fixing defects and implementing new features is their main business. Bazrafshan and Koschke [43] have reported cases in which developers have even ignored available information on existing clones although this information was prominently offered to them. On one hand, visualization should not interfere with the normal working mode of developers and should not distract them from their current task, but on the other hand, make them aware of clones if needed. Clone researchers may need to learn from researchers of software recommendation systems to find smarter approaches of information visualization on demand.

Moreover, as already discussed in the previous section, there are very few empirical studies with professional developers. Their requirements and feedback, however, are paramount for making any progress in industrial adoption. Action research may be a research model to better involve industrial developers. It is of course difficult to partner with companies for empirical long-term studies, yet the chances to convince industrial partners should be higher than in many other areas of software engineering, because the problem is highly relevant, there is some level of problem awareness, and tools are even already used in industry. It is an encouraging sign that clone visualization tools have become a part of industrial software development tools despite all of these obstacles. For instance, XIAO, which started as a research prototype [244,245] [P11,P12], has meanwhile become a component of Microsoft’s Visual Studio to allow users to perform clone-management tasks with the help of visualizations. Other attempts to use clone analysis in industry have been discussed by other researchers [41,246,247] [P60,P43]. We hope our classification framework of software clone visualizations paves the ground for more conscious use of clone visualization in industrial settings by inspiring vendors of professional tools.

8. Conclusions

At last, we summarize the answers to our research questions based on the result of our systematic mapping study on
software clone visualization. Since 1992, 68 scientific publications appeared on this subject, where 19% of those papers had visualization as one of their primary focus point and 40% as their only focus point. About 3–4 papers are published annually. They appear in a broad list of publication venues both in information visualization and software engineering, but predominantly in the latter. Most researchers in this area come from the area of clone detection; there are rather few contributors from researchers on visualization in general. The most frequent and continuous contribution comes from only two research groups: Osaka University and University of Saskatchewan. It would further advance the field if more researchers with interest in both software engineering and visualization would bring in their expertise.

We have described and classified clone visualizations in terms of visual attributes (how data is presented) and the information needs that they may support (what data is presented) and discussed their inherent strengths and limitations. Our mapping of visualization onto information needs and those in turn onto user goals allowed us to identify information needs and user goals with multiple alternative visualizations and those with currently insufficient support. For the former, empirical evaluations should be conducted to actually measure the level of support of alternative visualizations. For the latter, we have discussed how one could close the gap. Many of the less supported information needs could be addressed by adapting existing visualizations. For a few others, additional research is still needed to gather the necessary data to be visualized.

Our assessment of the state of empirical evaluations showed that for only a few clone visualizations, empirical evidence is currently available, which is inline with software visualization in general [214]. We gave pointers to the relevant literature on evaluating visualization in general that may be used by clone-visualization researchers to devise empirical studies. We believe that case studies and controlled experiments, although certainly valuable, are not enough because they do not capture long-term effects and involve users of those visualizations too late. More participatory long-term studies as implemented by action research, for instance, should be considered, too.

It is at least a promising sign that we found several commercial tools not only detecting but also visualizing clone data. There seems to be a real need for clone information. We believe that clone visualization should be paired with recommendation systems to create visualizations on demand so that developers are not distracted from their primary tasks and are informed only when the knowledge about clones becomes critical. To better integrate clone visualization into the development process, a better understanding of cognitive and social processes in development activities may be necessary, which calls for observatory studies of developers and organizations dealing with software clones. We argue that suitable visualizations must be developed in close partnership with real users managing clones in long-term studies. The design of useful visualizations must start with a clear vision of information needs pertinent in a particular task.

We hope our survey and classification of visualizations will be useful to developers (as user of clone visualization), tool vendors, and researchers alike. Users will be able to evaluate tool capabilities in view of their needs for clone detection and analysis. The survey may help tool vendors decide which visualizations should be supported by their tools, based on the goals of their envisioned users. Researchers may use our survey as a reference point for their work and to identify open problems worth further investigation. We do not claim that our presented classification framework is complete and, hence, invite others to extend it.

Declaration of competing interest
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References
Clone-Visualization References


