Towards Contextual and On-Demand Code Clone Management by Continuous Monitoring

Gang Zhang*, Xin Peng*, Zhenchang Xing†, Shihai Jiang*, Hai Wang* and Wenyun Zhao*
*School of Computer Science, Fudan University, Shanghai, China
† School of Computer Engineering, Nanyang Technological University, Singapore
{gangz, pengxin, 12212010036, 09302010040, wyzhao}@fudan.edu.cn
zcxing@ntu.edu.sg

***Abstract*—Effective clone management is essential for developers to recognize the introduction and evolution of code clones, to judge their impact on software quality, and to take appropriate measures if required. Our previous study shows that cloning practice is not simply a technical issue. It must be interpreted and considered in a larger context from technical, personal, and organizational perspectives. In this paper, we propose a contextual and on-demand code clone management approach called *CCEvents* (Code Cloning Events). Our approach provides timely notification about relevant code cloning events for different stakeholders through continuous monitoring of code repositories. It supports on-demand customization of clone monitoring strategies in specific technical, personal, and organizational contexts using a domain-specific language. We implemented the proposed approach and conducted an empirical study with an industrial project. The results confirm the requirements for contextual and on-demand code clone management and show the effectiveness of *CCEvents* in providing timely code cloning notifications and in helping to achieve effective clone management.**

I. Introduction

Code clones (i.e., similar or identical code fragments) are usually introduced by cope-paste-modify development practices. Code cloning is generally considered to be harmful to the quality of software systems as it may cause additional maintenance effort and incur inconsistent changes to multiple cloned code fragments [1], [2], [3]. However, it is also argued that in some situations code cloning is a reasonable or even beneficial design option [4].

To alleviate the problems caused by code clones, researchers have proposed techniques to detect code clones [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], visualize and filter code clones [15], [16], eliminate code clones by refactoring [17], [18], [19], and analyze the evolution of code clones [3]. Some researchers further propose approaches to improve the management of code clones by ensuring consistent changes to cloned code fragments [20] and notifying developers of changes of code clones [21].

An effective clone management mechanism must allow developers to recognize the introduction and evolution of code clones, to judge their impact on software quality, and to take appropriate measures if required. Our recent industrial study [22] indicates that the reasons why a code clone is introduced or evolved and the harmfulness of a code clone must be interpreted and considered in a larger context from technical, personal, and organizational perspectives.

For example, in a software system with a layered architecture, the architect is sensitive to code clones across layers, as it may bring undesired inter-layer coupling. As another example, if a self-disciplined developer who rarely introduces code clones in the main branch of a system code base, introduces a clone, it may imply the need for improvement in management and technical strategies. Furthermore, stakeholders of different roles may have different concerns about clone management. For example, in a large development team consisting of several groups of developers, quality assurance (QA) personnel may be concerned with code cloning across multiple groups. Legal staff may be concerned with code cloning that involves illegal or unintentional use of copyright protected source code [23].

Existing clone detection tools often report a large number of clones even for a medium-sized system. Without proper filtering mechanisms, the user can be swamped in a mass of clone information. There is a need for contextual and on-demand code clone management. With this support, stakeholders of different roles can define their own clone monitoring strategies according to their concerns and with the consideration of specific technical, personal, and organizational factors.

In this paper, we propose a contextual and on-demand code clone management approach called *CCEvents* (Code Cloning Events). *CCEvents* provides timely notifications about relevant code cloning events for different stakeholders through continuous monitoring of code repositories. It supports on-demand customization of clone monitoring strategies in specific technical, personal, and organizational contexts.

To this end, we propose an event-based code cloning domain model, which captures *What* (code clone itself) and various contexts of code clones including *When* (time when a code clone is introduced or modified), *Where* (code entities in which a code clone resides), and *Who* (developers who introduce or modify a code clone). We define a domain-specific language (DSL) called *CCEML* (Code Cloning Event Monitoring Language), which allows users to define their on-demand clone monitoring strategies based on the domain model. And we propose an event-driven framework to support contextual and on-demand management of code clones.

Based on the proposed approach, we implemented a code clone monitoring system and conducted an empirical study with an industrial software project. In the study, we elicited and analyzed the requirements for code clone management of

978-1-4799-0215-6/13 © 2013 IEEE

ASE 2013, Palo Alto, USA
Experience Track

stakeholders of different roles. We then evaluated the capability of *CCEvents* to satisfy various clone management requirements and the helpfulness of *CCEvents* for effective clone management. The results of our study confirm the requirements for contextual and on-demand code clone management in industrial development. The results also show that *CCEvents* can support a wide range of clone management requirements and provide timely code cloning notifications that are helpful for effective clone management.

The remainder of this paper is organized as follows. Section II motivates our research with some examples. Section III presents the proposed approach for contextual and on-demand code clone management. Section IV describes our implementation of the proposed approach. Section V reports the settings and results of our empirical study. Section VI discusses some issues about contextual and on-demand code clone management. Section VII reviews some related work. Section VIII makes our conclusion with outlines of future work.

II. Motivation

The motivation of our research on contextual and on-demand code clone management is illustrated by the following examples.

Case 1: A telecom product family has several variant products which use chips from different vendors. These chips provide similar functions, and each of them has a driver module provided by the vendor. To ensure that the core modules of the product family are independent of vendor-specific hardware, the development team defines an abstract layer on the top of vendor-specific driver modules and develops an adaptor for each driver module. As a result, the architect does not care about code clones among different adaptor modules but is sensitive to code cloning from vendor-specific adaptor modules to core modules, as the latter may undermine the independence between core modules and vendor-specific hardware. On the other hand, the copyright of driver modules belongs to the vendors whilst the copyright of adaptor modules belongs to the owner of the telecom product family. Therefore, the legal staff in charge of this product family cares about any code cloning from driver modules to adaptor modules.

Case 2: In the survey of our previous industrial study [22], a developer claimed that he never cloned code except to test the feasibility of an idea during development. This claim implies that he would only introduce code clones in tentative branches. However, our clone analysis identified some identical copies of a common algorithm introduced by him in the main branch of the system code base. In our interview, he said that he had to copy the algorithm into different modules because there was no common module to put common algorithms. He further explained that, as there were a lot of managerial process overheads, he chose not to create such a common module. This indicates that code clones introduced by a self-disciplined developer who seldom introduces code clones may imply the need for improvements in management and technical strategies.

Case 3: A big project includes an outsourcing group from an external subcontractor and several other groups from the prime contractor. The outsourcing group gets paid according to their development effort and performance. The capability and experience of the developers in the outsourcing group are thought to be not as good as the developers in the other groups. On the other hand, the calculation of their development effort is based on the number of lines of code. Therefore, out of consideration for software quality and management strategies, the project manager is much more sensitive to code clones introduced by the outsourcing group.

The above cases underscore the importance of contextual and on-demand code clone management. First, the ways in which stakeholders interpret and perceive code clones highly depend on the technical (e.g., architectural constraints), personal (e.g., personal characteristics of the developer who makes a code clone), and organizational (e.g., organization structures and management strategies of a development team) contexts. Second, stakeholders of different roles (e.g., architect, programmer, QA personnel, legal staff) have different concerns about clone management.

III. Approach

In this section, we first describe the event-based code cloning domain model and domain-specific monitoring language. We then present the event-driven monitoring framework and provide guidelines for defining context-sensitive and on-demand clone monitoring strategies.

A. Domain Model

To provide timely notifications about relevant code cloning events, *CCEvents* captures the introduction and evolution of code clones through continuous monitoring of code repositories. All the captured code cloning events are represented and structured based on the event-based code cloning domain model shown in Figure 1.

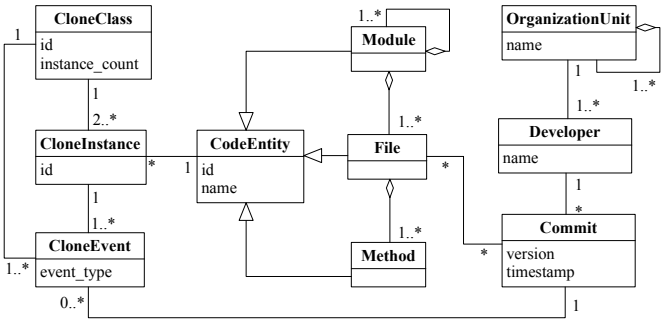

Fig. 1. Event-Based Code Cloning Domain Model

The domain model involves both clone instances (code fragments that are similar to or identical with others) and clone classes (a set of clone instances that are clones of each other). Based on revisions committed to a code repository, the domain model captures clone introduction and evolution events at both the clone instance level and clone class level as shown in Table I.

TABLE I
CODE CLONING EVENT TYPES

Entity	Event	Note
Clone Class	CloneClassCreated	A new clone class is introduced.
	CloneClassUpdated	An instance of a clone class is added, modified or removed.
	CloneClassRemoved	All the instances of a clone class are removed.
Clone Instance	CloneInstanceCreated	A new instance of a clone class is created.
	CloneInstanceUpdated	An existing instance of a clone class is modified.
	CloneInstanceRemoved	An existing instance of a clone class is removed.

To support the definition of contextual and on-demand clone monitoring strategies, the domain model captures additional contexts for code clones from three elementary aspects of time (*When*), location (*Where*), and person (*Who*).

1) ***When:*** A code cloning event is associated to a commit to the repository. A commit has a version number and a timestamp. The time when a cloning event occurs thus can be derived from the timestamp of the associated commit.
2) ***Where:*** A clone instance resides in a code entity (method, file, or directory). A simple clone instance involving only one code fragment resides in a method. A method- or file-level structural clone [24] resides in a module or a file. Code entities of different levels constitute a hierarchical structure, i.e., a file includes multiple methods, a module includes multiple files, and a composite module includes multiple sub-modules.
3) ***Who:*** A commit is made by a developer. The developer who creates/updates/removes a clone thus can be derived from the developer who makes the associated commit. A developer belongs to an organization unit (e.g., a group or team) and an organization unit can contain multiple smaller organization units.

Based on these elementary contextual aspects, more complex technical, personal, and organizational contexts can be specified by combing different elementary aspects. For example, technical contexts usually can be specified by combining the aspects of *When* and *Where*; personal contexts usually can be specified by combining the aspects of *When* and *Who*; organizational contexts usually can be specified by combining the aspects of *When* and *Who* (including the organization units).

To facilitate the definition of clone monitoring strategies, we derive from the code cloning domain model an equivalent attribute-value representation for code cloning events. The attribute-value representation of a code cloning event can be derived by traversing the model from a cloning event object through relationship links. For example, a *CloneInstanceCreated* event can be represented as follows.

```
EventType          = CloneInstanceCreated
CloneInstance.id   = 1003
Method.name        = sayHello
File.name          = Hello.java
Developer.name     = Mary
                ...
```

B. Code Cloning Event Monitoring Language

In our approach, clone monitoring strategies are defined as cloning event monitoring rules based on the domain model in Figure 1. To facilitate the definition of cloning event monitoring rules, we define a monitoring language *CCEML*. The syntax of *CCEML* is defined in BNF (Backus-Naur Form) as follows. Note that *type_name* and *attribute* refer to the types and attributes defined in the domain model (see Figure 1).

```
<rule> ::= EVENT <event_type> WITH <condition> ;
<event_type> ::=
      CloneClassCreated
    | CloneClassUpdated
    | CloneClassRemoved
    | CloneInstanceCreated
    | CloneInstanceUpdated
    | CloneInstanceRemoved ;
<condition> ::=
      <value> <value_logical_op> <value>
    | <type_name> IN "(" <collection> ")"
    | NOT <condition>
    | <condition> AND <condition>
    | <condition> OR <condition>
    | "(" <condition> ")"
    | TRUE |FALSE;
<value> ::=
      [this "."] [<type_name> "."] <attribute>
    | <string> | <number> | <time>
    | <value> <value_arithmetic_op> <value>;
<value_logical_op> ::=
     "==" | ">=" | "<=" | ">" | "<";
<value_arithmetic_op> ::=
     "+" | "-" | "*" | "/";
<collection> ::=
      <type_name> SATISFY "(" <condition> ")"
    | <type_name> EXIST  "(" <condition> ")"
    | <type_name> ALL  "(" <condition> ")"
    | DISTINCT "(" <collection> "," <attribute> ")"
    | <collection> UNION <collection>
    | <collection> INTERSECT <collection> ;
<number> ::= <num_val>
    | <number> <num_op> <number>
    | COUNT "(" <collection> ")" ;
<time> ::= <time_val> | <time_duration>
    | <time> <time_op> <time> ;
```

A clone monitoring rule specifies a cloning event of a specific type and meeting specific conditions. A basic condition can be an attribute comparison condition (e.g., "`Developer.name == "Tom"`") or a membership condition of a defined collection (e.g., "`CloneClass IN (CloneClass EXIST (Module.name == "DriverX"))`"). For a given cloning event object *event*, an attribute comparison condition can qualify an attribute of *event* itself or an attribute of another object that is of the type *type_name* and can be navigated from *event*. Note that only one object of each type in the domain model can be navigated from a cloning event object. A membership condition specifies that the object that is of the type *type_name* and can be navigated from *event* is a member of a defined collection. More complex conditions can be defined as the logical combinations of multiple condition expressions.

A collection can be defined as a set of objects of a specific type and meeting specific conditions. For an object *obj* of specific type, the condition can be one of the follows.

- `SATISFY` condition: the attributes of *obj* meet the condition;
- `EXIST` condition: there exists an object of another type that can be navigated from *obj* and meets the condition;
- `ALL` condition: all the objects of another type that can be navigated from *obj* meet the condition.

The `DISTINCT` construct can be used to define a collection consisting of distinct attribute values of another collection. A collection can also be defined as the `UNION` or `INTERSECT` of another two collections.

A value in a condition expression can be a constant string, number, or time. It can also be a reference to an attribute of an object. The keyword `this` refers to the current cloning event object, i.e., the last cloning event that is detected.

C. Event-Driven Framework

To provide timely notifications for clone management stakeholders, *CCEvents* adopts an event-based clone management framework based on continuous monitoring of code cloning events (see Figure 2). Developers commit changes to the code repository and thus produce code cloning events. Clone management stakeholders of various roles (including developers themselves) define their clone monitoring strategies and obtain timely notifications about relevant code cloning events.

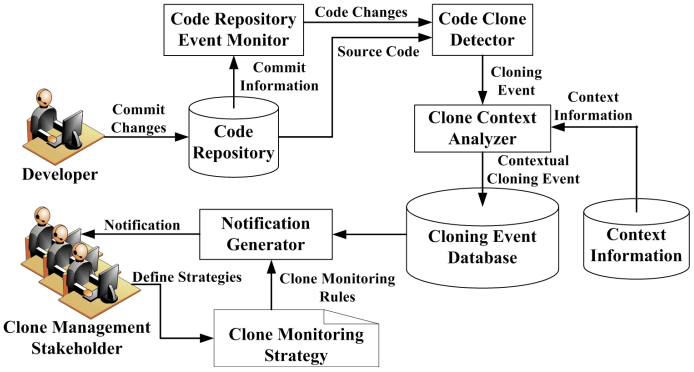

Fig. 2. Event-Driven Clone Management Framework

The code repository event monitor continuously monitors commit events to the code repository and provides code changes for the code clone detector. Based on the code changes, the code clone detector detects whether the code changes lead to one or several code cloning events, i.e., clone instances and clone classes are created, updated, or removed. To achieve the efficiency required by continuous monitoring, an incremental clone detection algorithm should be adopted to involve only code clones that are related to the code changes. If some cloning events are identified from the code changes, the clone context analyzer associates the detected cloning events with additional context information such as organization structures. Then the cloning events with context information are represented based on the domain model shown in Figure 1 and added to the cloning event database.

Clone management stakeholders of various roles define their clone monitoring strategies as *CCEML* rules. Based on the defined clone monitoring rules, the notification generator checks for every rule whether the monitoring conditions are satisfied. If the conditions of a monitoring rule is satisfied, the current code cloning event and related context information are notified as a warning or reminder to the stakeholder who defines the rule, for example, by email or SMS (Short Message Service).

D. Strategy Definition Guideline

To achieve their clone management requirements, stakeholders need to map their clone monitoring strategies to cloning event monitoring rules that can be expressed by *CCEML*.

1) Technical Perspective: Clone monitoring strategies from the technical perspective usually involve monitoring of cloning events that may violate design constraints (e.g., the constraints of layered architecture), degrade the design quality (e.g., extensibility), or indicate opportunities of improvement (e.g., design refactoring). Therefore, monitoring rules from this perspective usually need to combine constructs from the *Where* aspect to specify conditions about the location of code clones and constructs from the *When* aspect to specify conditions about the sequence of related events of clone introduction and evolution.

Example 1: In the first case in Section II, the architect wants to keep core modules independent of vendor-specific hardware. For his concern about code cloning from vendor-specific adaptor modules to core modules, a monitoring rule can be defined as follows to specify the introduction of a new clone instance to the module *Core* and the clone class has at least one instance in the module *AdapterX*.

```
EVENT CloneInstanceCreated WITH
  Module.name == "Core" AND CloneClass IN
  (CloneClass EXIST (Module.name == "AdapterX"))
```

Example 2: Some developers want to be notified about possible opportunities for clone-related refactoring. Our previous industrial study [22] reveals that a critical point of time for the elimination of code clones is the time when the third clone instance is introduced. We found that the more times a piece of code is cloned, the more likely it will be cloned in the future, and the less likely it will be removed. In addition, for a developer leading the development of a specific module *ModuleX*, he is only concerned with the clone classes for which all the instances which reside in *ModuleX*. For this concern, we can define a monitoring rule as follows.

```
EVENT CloneInstanceCreated WITH
  CloneClass.instance_count==3 AND
  Module.name =="ModuleX" AND CloneClass IN
  (CloneClass ALL (Module.name == "ModuleX"))
```

Example 3: For a clone class that is under refactoring, the developers are usually not allowed to introduce more instances of the clone class. The following rule reflects this concern by filtering clone instance creation events of the clone classes for

which developers conducted clone elimination recently (e.g., within the last seven days).

```
EVENT CloneInstanceCreated WITH
  CloneClass IN (CloneClass EXIST
  (CloneEvent.event_type ==
  "CloneInstanceRemoved" AND
  Commit.timestamp > this.Commit.timestamp-7D))
```

2) Personal Perspective: Clone monitoring strategies from the personal perspective consider personal characteristics such as experience, capability, and habits. Therefore, monitoring rules from this perspective use developer information to filter relevant code cloning events.

Example 4: As a novice developer, Tom is thought to lack of good judgment of design quality. Therefore, the manager pays special attention to new clone classes introduced by Tom, but does not care if Tom makes new instances of existing clone classes. This strategy can be specified as the following rule.

```
EVENT CloneClassCreated
  WITH (Developer.name == "Tom")
```

Example 5: In the second case in Section II, the manager cares about code clones introduced by a self-disciplined developer who seldom makes code clones. For this strategy, we can use the number of clones that a developer has made in the past period of time (e.g., 90 days) to characterize a self-disciplined developer. This strategy can be specified as the following rule.

```
EVENT CloneInstanceCreated WITH
  COUNT(CloneEvent EXIST(
  CloneEvent.event_type ==
      "CloneInstanceCreated"
  AND Developer.name == this.Developer.name
  AND Commit.timestamp >
         this.Commit.timestamp-90D)) < 3
```

3) Organizational Perspective: Clone monitoring strategies from the organizational perspective usually concern code cloning events that are related the organizational policies on the organizational structures, team management, and software process. Therefore, monitoring rules from this perspective usually need to combine conditions about the developers, organization units and the time. In some cases, the location of code clones is also involved if the organizational policy deals with, for example, the module structure.

Example 6: In the first case in Section II, the legal staff cares about code clones between driver modules and adaptor modules. This concern can be specified by the following monitoring rule.

```
EVENT CloneInstanceCreated WITH
  Module.name == "AdapterX" AND CloneClass IN
  (CloneClass EXIST (Module.name == "DriverX"))
```

Example 7: In the third case in Section II, the project manager is concerned with code clones introduced by the outsourcing team. This concern can be reflected by the following monitoring rule.

```
EVENT CloneInstanceCreated WITH
  (OrganizationUnit.name == "subcontractorX")
```

IV. Implementation

We have implemented *CCEvents* in a code clone monitoring system. The structure of the implemented monitoring system is shown in Figure 3. The system has a layered architecture consisting of a series of subsystems, including ① user interface (UI) subsystem, ② clone detection subsystem, ③ context analysis subsystem, ④ cloning event subsystem, ⑤ code repository subsystem. The *CCEvent* database (using *MySQL* in our current implementation) stores updated code cloning events with context information.

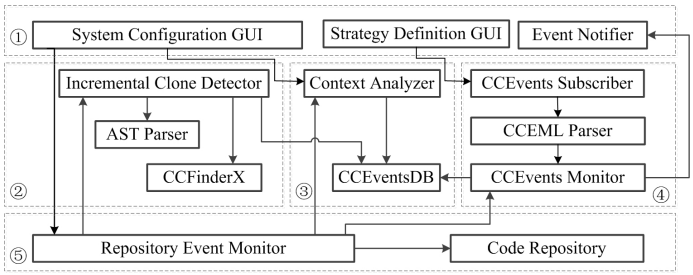

Fig. 3. *CCEvents* Implementation Structure

The UI subsystem provides web-based graphical user interfaces (GUIs) for the administrator to configure the whole system and for clone management stakeholders to define their strategies. The administrator can use the system configuration GUI to configure access parameters of the code repository (e.g., the IP address) and context information such as organization structures. Stakeholders of various roles can use the strategy definition GUI to define their own clone monitoring strategies using *CCEML*. The event notifier generates notifications about relevant cloning events for stakeholders. In our current implementation, the event notifier can provide timely notifications for stakeholders by sending emails containing the captured code cloning events and related context information.

The code repository subsystem monitors commit events in a target code repository. Our current implementation supports *subversion (SVN)* and *mercurial*. We designed a set of abstract repository interfaces, which allow us to support more kinds of repositories (e.g., *git*) by providing repository-specific implementation for the interfaces.

The clone detection subsystem identifies code cloning events using an incremental clone detection algorithm based on code changes. Our incremental clone detector is implemented based on CCFinderX, which is an efficient open-source clone detector re-designed from the previous tool CCFinder [8]. The incremental clone detection process can be described as follows: divide all the files into two groups, i.e., new or changed files (G_1), unchanged files (G_2); remove all the clone instances in G_1 from the existing clone results; detect code clone pairs between G_1 and G_2; detect code clone pairs within G_1; merge the detected new clone instances with the existing clone results. After an incremental clone detection process, the clone detector generates various kinds of cloning events based on the changes of the clone results and stores them in the *CCEvent* database. Moreover, it uses an abstract syntax tree (AST) parser to

identify the changes of code structures and updates the code entities in the database.

The context analysis subsystem analyzes context information from different sources and associates it with detected cloning events. The context analyzer obtains the *When* (the commit time and version) and *Who* (the committer) information of a cloning event from the code repository based on the related commit event. It gets the *Where* (the code entities where related clone instances reside) information from the code structures of the system. Additionally, it associates individual developers with the organization units maintained by the administrator.

The cloning event subsystem implements the on-demand cloning event monitoring and notification. It reads clone monitoring strategies defined by different stakeholders and translates them into the internal representation of event monitoring rules. For each captured code cloning event, it evaluates all the event monitoring rules and sends matched events with related context information to the event notifier.

V. Empirical Study

To validate the requirements for contextual and on-demand code clone management and evaluate the effectiveness of our approach, we conducted an empirical study with an industrial project. The objective of this study is to investigate the following three research questions:

1) Are there really requirements for contextual and on-demand code clone management in industrial projects?
2) Can the elicited clone management requirements be well supported by the domain model and cloning event monitoring language of *CCEvents*?
3) How do the practitioners evaluate the monitoring results generated by *CCEvents* in terms of their helpfulness for clone management?

A. Study Design

The subject project of our empirical study is a medium-scale telecom product of about 629,347 lines of code developed by a large company. The product is developed in C/C++ and some of the source code was reused from another similar product. The project has a team of more than 40 developers and most of them are experienced developers with more than five years of software development experience. It has a development history of more than 20 months and is still in a period of active development. This ensures that: we can identify plenty of code clones of various types for our study; we can find enough representative stakeholders of different roles for interview and survey. Moreover, the project has deployed a continuous integration system with a *mercurial* code repository from the very beginning and has complete records of historical commits.

Our empirical study followed a three-stage process. The objective of the first stage is to investigate the requirements for contextual and on-demand clone management. In this stage, we interviewed 17 stakeholders of the team, including one project manager, two architects, four group leaders, and 10 programmers. Two of the 10 programmers are also module owners, who are developers but also in charge of the integration and quality assurance of some modules. In the interview with each stakeholder, we first asked him whether he noticed the phenomena of duplicated code fragments in his development. As code clones are so popular in source code, all of the stakeholders confirmed that they knew the problem of code clones. We then asked the stakeholder whether he thought code clones had impact on software development from his point of view. Among all the 17 stakeholders, 10 of them agreed that code clones had great impact on their development, six of them evaluated the impact to be medium, and only one of them evaluated the impact to be little. Based on these two questions, we had a preliminary understanding of the stakeholder's attitude and experience about code clones and began to elicit his requirements on code clone management. The elicited requirements were iteratively refined in an interactive process. And in the process we tried to elicit not only the clone management requirements of stakeholders but also the rationale behind their requirements.

The objective of the second stage is to evaluate *CCEvents* in terms of its ability of supporting clone management requirements elicited in the first stage and generate clone monitoring results for the stakeholder feedback in the third stage. We analyzed each of the elicited requirements and defined a cloning event monitoring rule for it, if possible. We then simulated the continuous monitoring of *CCEvents* on the subject project with the historical data of its code repository in four months. There were totally 1,655 commits (13.8 commits per day on average) from 23 developers during the period and each commit was made by one developer. Among all the 1,655 commits, 694 commits from 16 developers have at least one cloning event detected. In total, *CCEvents* identified 2,029 code cloning events and generated 1,958 monitoring results (notifications).

The objective of the third stage is to get stakeholders' feedback on the helpfulness of the monitoring results generated by *CCEvents*. In this stage, we invited all the stakeholders involved in the requirement elicitation process to participate in an email survey for their feedback on the generated clone monitoring results. For each participant, we selected five representative results from all the monitoring results generated for him with the following criteria: cover different clone management requirements; cover code clones in different modules and files. The reason why we only selected five results for each participant lies in the fact that it requires much time and effort for a participant to recall the situation reflected by each result and evaluate its implication. Each monitoring result is presented with a HTML file describing the involved code clone and its context information. For each monitoring result, the participant was asked to give a score for its helpfulness for clone management (1 being *not helpful at all*, 3 being *medium*, 5 being *very helpful*) and explain his evaluation. We received responses from nine participants in one week, including one architect, two group leaders and six programmers (two of them are also module owners).

TABLE II
ELICITED CLONE MANAGEMENT REQUIREMENTS

ID	Requirement	Rational	Context	Role
R1	Notify me about clone introduction and evolution in newly developed modules.	The reused legacy modules are hard to be kept in a good quality as they have accumulated a lot of design problems.	technical	architect (2), group leader (3),
R2	Notify me about clone introduction and evolution in methods or files of high complexity.	These modules have higher value of refactoring to reduce the maintenance effort.	technical	architect (2)
R3	Notify me when someone else copies code from my modules.	This may imply opportunities of refactoring.	technical	module owner (2)
R4	Notify me when new code clones are introduced in my modules.	This may degrade the design quality of the modules.	technical	module owner (2)
R5	Don't notify me about cloning events of some specific clone classes.	These code clones reflect known design problems that cannot be solved currently or are thought to be harmless.	technical	architect (2), module owner (2), programmer (5)
R6	Notify me when a clone relation is introduced between two different modules that have no clone relations before.	This may introduce undesired dependencies and coupling between these two modules.	technical	architect (2)
R7	Notify me when someone makes code clones among modules of different layers.	This may violate the architectural constraints about inter-layer dependencies.	technical	architect (2)
R8	Notify me when the instance number of a clone class in my modules has reached or exceeded three and all the instances of the clone class are in my modules.	This indicates a critical point of time for the elimination of the clone and the module owner has full authority of the clone class.	technical	module owner (1)
R9	Notify me when a new instance is introduced to a clone class that is currently under refactoring.	This may break the refactoring process.	technical	architect (2), module owner (1)
R10	Notify me when a specific novice developer introduces a new clone class.	The novice developer may lack good judgment of design quality.	personal	project manager (1), group leader (1)
R11	Notify me about code clones introduced by a self-disciplined developer who seldom makes code clones.	This may imply possible improvements in the management and technical strategies.	personal	project manager (1), group leader (1)
R12	Notify me about code clones introduced by myself.	Some programmers care about only code clones introduced by themselves.	personal	programmer (8)
R13	Notify me when someone copies code from copyright protected modules (e.g., open-source modules or modules from other vendors) to other modules.	This may bring legal risks of violating source code licences.	organizational	project manager (1)
R14	Notify me when the members of my group introduce code clones or modify existing clones.	These clone-related activities can reflect the technical maturity of group members.	organizational	group leader (2)

B. RQ 1: Requirements Investigation

After the investigation in the first stage, we elicited 43 clone management requirements from all the 17 stakeholders. We found a large part of the requirements were similar or even identical ones raised by different stakeholders. So we merged similar and identical requirements together and finally obtained 14 clone management requirements. These requirements, the rationales behind them, the contextual perspectives, and the roles of the stakeholders who were concerned with the requirements are presented in Table II. The numbers in the last column indicate how many stakeholders of different roles mentioned the requirement in the investigation.

It can be seen that a large part of the requirements (9 out of 14) are considerations from the technical perspective. These requirements reflect technical concerns at different levels. Architects usually are concerned with code clones that may bring risks and opportunities at the architecture level such as inter-layer clones and the first clone pair between two modules. Module owners are concerned with code clones that are related to their modules. Some of the requirements (R1, R3, R4, R5) only involve basic information of code clones such as number of instances and modules where the clones reside. Others (R2, R6, R7, R8, R9) are based on more extensive context information such as architectural strategies and distribution of clone instances.

The three requirements from the personal perspective concern code clones introduced by specific developers such as a novice developer (R10), a self-disciplined developer (R11), or the stakeholder himself (R12). The characteristics of the targeted developers cause their cloning actions get more attention from related stakeholders. For example, it is thought to be not so significant if a novice developer makes a new clone by following an existing clone class, but it is noteworthy if he introduces a new clone class.

The two requirements from the organizational perspective reflect considerations about the organization structures and management strategies. R13 concerns code clones that may violate the intellectual property strategies of the organization. R14 concerns code clones introduced or modified by the members of a specific group.

Although Table II is not a complete list of all the clone management requirements in industrial projects, it reflects the fact that stakeholders of different roles in an industrial project have quite different viewpoints and concerns about code clone management. It can also be seen that many of the requirements reflect considerations about specific technical, personal, and organizational factors. Therefore, our answer to the first

research question is positive, i.e., **there are requirements for contextual and on-demand code clone management in industrial projects**.

C. RQ 2: Clone Monitoring Strategy Definition

In the second stage, we tried to define clone monitoring rules with *CCEvents* for each of the elicited requirements in Table II and analyzed the compliance of the defined monitoring rules with the requirements.

Among all the 14 elicited requirements, we found 11 of them (R1, R3, R4, R5, R6, R7, R8, R10, R11, R12, R14) are fully supported by *CCEvents*. R1, R3, and R4 can be implemented by qualifying the scope of relevant modules and R5 can be easily supported by eliminating some irrelevant clone classes. R6 can be supported by checking both the clone relations between two modules before and after the current code cloning event. R7 can also be supported by qualifying specific modules, if the layered architecture is reflected in the modular structure, i.e., each layer corresponds to a composite module including some sub-modules. R8 can be supported by combining the conditions about the instance number of a clone class and the distribution of its clone instances among modules. R10, R12 and R14 can be implemented by quantifying the developer who makes a clone or the organization which he belongs to. R11 is based on the quantification of a self-disciplined developer, which can be characterized by his frequency of making clones in the past.

Two of the elicited requirements (R9, R13) are partially supported by *CCEvents*, R9 is based on the technical context that a clone class is under refactoring. As refactoring is a high-level conceptual activity, it is hard to fully characterize it with the automatically collected context information. Currently, *CCEvents* can only roughly characterize a clone class under refactoring as that some of its instances were removed recently. The difficulty for R13 is that the team may only have the binary code of third-party modules. Therefore, R13 can only be fully supported if the protected modules have all their source code in the targeted code repository.

One of the elicited requirements (R2) is not supported by *CCEvents* currently. It requires additional context information about the code metrics of files and methods such as lines of code and McCabe's Cyclomatic complexity, which are currently not supported by the domain model and monitoring language of *CCEvents*. This implies the need for the incorporation of more extensive context information by integrating more repositories and tools such as defect tracking systems and code metrics tools.

Based on our analysis, it can be seen that most of the elicited clone management requirements can be fully (78.6%) or partially (14.3%) supported by *CCEvents* and only one of them cannot be supported. Therefore, our answer to the second research question is positive, i.e., **most of the elicited clone management requirements can be well supported by *CCEvents***.

D. RQ 3: Feedback for Clone Monitoring Notifications

We received scores for 45 clone monitoring results from nine participants. The evaluation score for the generated clone monitoring results is 3.47 on average. And the number of results of each score is presented by roles in Figure 4.

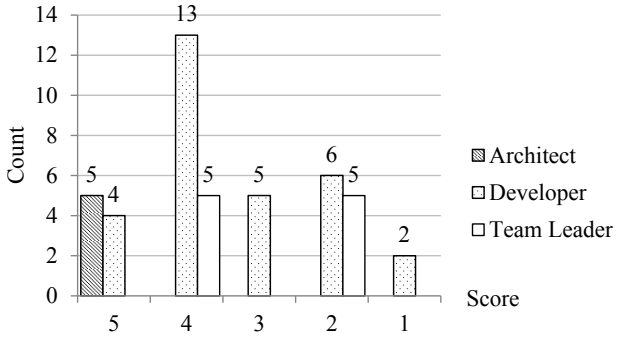

Fig. 4. Evaluation Scores of the Clone Monitoring Results

We investigated the reasons why participants of different roles evaluated a clone monitoring result to be helpful or not based on their explanations.

We found that all the five scores given by the architect are 5 (i.e., *very helpful*). All these monitoring results reflect architecture-level concerns (three for R6 and two for R7). For example, one of the generated monitoring results for R6 revealed that an inter-module clone was first introduced from an authentication module to an http module in an early commit. This early clone had caused other developers to follow, making totally 11 clone instances. The architect told us that the cloned code is a utility function *HEX2INT* which should have been put in a utility module. He further commented that this would have been avoided if he had been notified about the first clone between the two modules. A monitoring result for R7 revealed a cross-layer clone from a driver module to an application module. The cloned code is a function that translates an error code to an error message. The architect told us that this clone is very harmful as it brought implicit coupling between two layers.

We found that a programmer rated all the monitoring results for him to 2 (i.e., *fairly unhelpful*). He commented that all the reported clones were third-party code that was not written by him but just committed by him. In an actual application of *CCEvents*, he can just adjust his clone monitoring strategies to eliminate those clones in third-party modules.

We also found that a programmer rated a monitoring result for him to 1 (i.e., *not helpful at all*) He told us that the detected code clone was a commented out code fragment using the construct "`#ifdef 0 ... #endif`". And we further found that a series of code clones of this kind were introduced by him. We discussed this with his group leader. He said that this way of commenting out code fragments was not encouraged and the generated monitoring result implied a potential opportunity for improvement. The other *not useful at all* rating was given by another programmer. He commented that the two detected clone instances implemented completely different functions.

After analyzing the code, we found these two clone instances were introduced not by copy-paste but by applying similar patterns.

Based on the above analysis, we draw a conclusion for the third research question that **most of the generated monitoring results, especially those for architectural concerns, are thought to be helpful for clone management**.

E. Threats to validity

A major threat to the internal validity of our study is the insufficient requirements elicitation in the first stage. The stakeholders involved in the requirements investigation may not cover all the stakeholders with different concerns on clone management. And it may be hard for the involved stakeholders to fully express their clone management requirements in a short time. Even with such an insufficient requirements elicitation process, we have identified a set of representative clone management requirements involving considerations from technical, personal, and organizational perspectives and reflecting the concerns of different roles. We believe that as more clone management requirements are elicited, our conclusions about contextual and on-demand clone management requirements can be further strengthened.

The other major threat to the internal validity of our study lies in the way we generated clone monitoring results for the feedback survey of stakeholders. The results were not generated by a continuous monitoring process in a real application of *CCEvents*, but obtained by simulating the continuous monitoring with the historical data of the code repository of the subject project. Although the obtained monitoring results actually reflected code cloning events occurring before, they embodied technical, personal, and organizational contexts existing some time in the past. And we told the participants of the feedback survey in the third stage to consider and evaluate the helpfulness of the generated monitoring results by putting themselves in a specific point of time in the past. The participants thus could give similar evaluation to the generated monitoring results as they received instant notifications about their relevant cloning events.

A major threat to the external validity of our study lies in the fact that we only conducted one empirical study with only one project in one specific organization. When more empirical studies are conducted in more projects and organizations, we think more clone management requirements reflecting different project- or organization-specific contexts can be elicited. At the same time, the expressiveness of the domain model and monitoring language of *CCEvents* may be questioned as more clone management requirements are elicited. For this threat, we think on the one hand, *CCEvents* has provided a set of elementary constructs that can be flexibly combined for different purposes. On the other hand, we have seen a need of incorporating more context information by integrating more extensive repositories and tools such as defect tracking systems and code metrics tools.

VI. Discussion

Whether code clones are harmful or not is still an open issue [1]. Our answer to this question is that, the impact of a code clone on software quality needs to be considered in specific contexts. The philosophy behind *CCEvents* is to keep stakeholders of different roles aware of relevant code cloning events based on customized monitoring rules reflecting specific technical, personal, and organizational contexts and their roles in a project. Based on the notified code cloning events and related context information, the stakeholders can identify potential risks and opportunities and take appropriate measures to alleviate the risks and take advantage of the opportunities.

Clone management can play a more active role in software development processes. One proposal of the 2012 Dagstuhl seminar on software clones [25] was to integrate clone detection with code repositories and be part of quality assurance in continuous integration. Our approach is based on continuous monitoring of code repositories. We think a good way to apply *CCEvents* is by integrating it with a continuous integration system. In a continuous integration process, *CCEvents* can detect cloning events based on all the committed changes and generate clone-related warnings and reminders as a part of the continuous integration feedback.

In the empirical study, we found that some developers lack the knowledge required to discover the risks and opportunities contained in the detected clone introduction and evolution events. Therefore, there is a need to further enhance the awareness of and spread the knowledge about the risks and opportunities brought by code clones. Based on the current monitoring and notification framework of *CCEvents*, we think it is beneficial to allow developers to share and exchange their knowledge, for example, through a website. The clone-related warnings and reminders generated for different developers can be posted on the website and the subsequent treatments (e.g., ignored or the measures taken by them) to these warnings and reminders can be tracked on the website. The developers thus can learn from others how to identify potential risks and opportunities contained in detected code clones. They can also comment on the posted warnings and reminders and the subsequent treatments. With this kind of knowledge sharing and exchanging, the knowledge and experience about clone-related risks and opportunities of the developers in an organization or project can be improved.

As discussed in Section V-C, the context information captured in the domain model is still limited. Therefore, it could be helpful to integrate more extensive repositories and tools such as defect tracking systems and code metrics tools. This integration would allows stakeholders to define more sophisticated clone monitoring strategies, for example, by associating code clones with defect distribution and complexity metrics.

Another limitation of *CCEvents* lies in the fact that it may be hard for a stakeholder to map his concerns about code clones to rules defined by the monitoring language of

CCEvents. And with the changes of the technical, personal, and organizational contexts, the clone management requirements of stakeholders may also evolve. Therefore, it is helpful to mine clone monitoring rules that can reflect the changing requirements for specific stakeholders of different roles and recommend potentially relevant monitoring results to them.

VII. Related Work

Clone detection is the basis of clone management. Researchers have proposed clone detection techniques that can provide automated assistance in identifying code clones by analyzing abstract syntax tree (AST) [7], [12], [14], text and string [10], [11], token sequences [8], program dependency graph (PDG) [9], [13], or code metrics [5], [6]. Based on clone detection techniques, some researchers further analyzed the evolution of code clones. Kim et al. [3] developed a clone genealogy tool that automatically extracts the history of code clones and conducted an empirical study of code clone genealogies.

To help developers to identify useful clones, some techniques were proposed to help developers to visualize and understand code clones. Ueda et al. [15] presented a maintenance support environment called Gemini, which can visualize the code clone information produced by clone detection tools. Zhang et. al [16] proposed a technique for filtering and visualization of cloning information generated by clone detection tools.

Some researches [17], [18], [19], [26] focused on approaches and tools that can assist developers to eliminate code clones by refactoring. Tairas and Gray [19] presented an Eclipse plug-in that can bridge the gap between clone detection and refactoring by forwarding clone detection results to the refactoring engine in Eclipse. Lee et al. [26] proposed an approach that can detect code clones that are suitable for refactoring and generate an appropriate schedule to maximize quality improvement through refactoring.

It is thought that inconsistent changes among clone instances of the same clone class may cause additional maintenance effort and even bugs. Therefore, some research has focused on detecting inconsistent clone changes and finding possible bugs caused by inconsistent clone changes. Lucia et al. [27] proposed an approach that can increase the rate of true positives found when a developer analyzes clone anomaly reports by incorporating incremental user feedback to continually refine the anomaly reports. Wang et al. [28] proposed an approach that can automatically predict the harmfulness of a code cloning operation at the point of performing copy-and-paste.

In recent years, there have been growing interests in clone management approaches and tools, which can help developers to be aware of the presence of code clones and make consistent changes. Toomim et al. [29] proposed an editor-based interaction technique for managing code clones. With the technique, a programming environment can keep track of code clones and provide enhanced visualization and editing facilities for developers to understand and modify multiple code clones as one. Duala-Ekoko and Robillard [30] presented a clone tracking system, which can produce clone regions within methods from the output of a clone detection tool, notify developers of modifications to clone regions, and support the simultaneous editing of clone regions. Nguyen et al. [20] presented a novel clone management tool called JSync, which provides supports for developers in being aware of the clone relation among code fragments and in making consistent changes as they create or modify cloned code. Yamanaka et al. [21] proposed a clone change management system, which can categorize code clones based on their evolution patterns and report changed information through a web-based UI and e-mail notification.

CCEvents provides timely notification about relevant code cloning events for stakeholders through continuous monitoring of code repositories. The differences between *CCEvents* and other clone management approaches lie in the following aspects. First, *CCEvents* captures additional contextual information of code clones such as module structure, organization structure, commit time. Second, it allows stakeholders of different roles to define on-demand clone monitoring strategies from technical, personal, and organizational perspectives.

VIII. Conclusion

In this paper, we have proposed a clone management approach called *CCEvents*, which supports contextual and on-demand code clone management through continuous monitoring of code repositories. We have proposed an event-based code cloning domain model capturing elementary context aspects for code clones and defined a domain-specific language called *CCEML* for stakeholders to specify on-demand clone monitoring strategies from technical, personal, and organizational perspectives. Then based on an event-driven framework, *CCEvents* can provide timely notifications about code cloning events for stakeholders according to their on-demand clone management requirements.

We have implemented *CCEvents* in a code clone monitoring system and conducted an empirical study with an industrial project. The results show that clone management stakeholders of different roles do have the requirements for contextual and on-demand code clone management with timely notifications about relevant code cloning events. The results also confirm that *CCEvents* can support a wide range of contextual and on-demand clone management requirements and can provide timely and helpful notifications for clone management.

In future work, we will try to extend the code cloning domain model and *CCEML* and integrate more context information such as code metrics and defect data with *CCEvents*. On the other hand, we will further evaluate the effectiveness of *CCEvents* by applying it in the development process of an industrial project for a long period of time.

Acknowledgment

This work is supported by National High Technology Development 863 Program of China under Grant No.2013AA01A605.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," School of Computing, Queen's University, Tech. Rep. 2007-541, 2007.

[2] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger, "Relation of code clones and change couplings," in *Fundamental Approaches to Software Engineering*, 2006, pp. 411–425.

[3] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 187–196.

[4] C. Kapser and M. W. Godfrey, "'Cloning Considered Harmful' considered harmful," in *Working Conference on Reverse Engineering*, 2006, pp. 19–28.

[5] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1, pp. 77–108, 1996.

[6] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *International Conference on Software Maintenance*, 1996, pp. 244–253.

[7] I. D. Baxter, A. Yahin, L. M. D. Moura, M. Sant'anna, and L. Bier, "Clone detection using abstract syntax trees," in *International Conference on Software Maintenance*, 1998, pp. 368–377.

[8] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[9] J. Krinke, "Identifying similar code with program dependence graphs," in *Working Conference on Reverse Engineering*, 2001, pp. 301–309.

[10] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 1, pp. 37–58, 2006.

[11] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *International Conference on Program Comprehension*, 2008, pp. 172–181.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *International Conference on Software Engineering*, 2007, pp. 96–105.

[13] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *International Conference on Software Engineering*, 2008, pp. 321–330.

[14] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Cleman: Comprehensive clone group evolution management," in *IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 451–454.

[15] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Maintenance support environment based on code clone analysis," in *IEEE Symposium on Software Metrics*, 2002, pp. 67–76.

[16] Y. Zhang, H. A. Basit, S. Jarzabek, D. Anh, and M. Low, "Query-based filtering and graphical view generation for clone analysis," in *International Conference on Software Maintenance*, 2008, pp. 376–385.

[17] R. Fanta and V. Rajlich, "Removing clones from the code," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 11, pp. 223–243, 1999.

[18] F. V. Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in *IEEE/ACM International Conference on Automated Software Engineering*, 2004, pp. 336–339.

[19] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Information and Software Technology*, vol. 54, no. 12, pp. 1297–1307, 2012.

[20] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008–1026, 2012.

[21] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Industrial application of clone change management system," in *International Workshop on Software Clones*, 2012, pp. 67–71.

[22] G. Zhang, X. Peng, Z. Xing, and W. Zhao, "Cloning practices: Why developers clone and what can be changed," in *International Conference on Software Maintenance*, 2012, pp. 285–294.

[23] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder," in *International Conference on Software Engineering*, 2007, pp. 106–115.

[24] H. A. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 497–514, 2009.

[25] R. Koschke, I. D. Baxter, M. Conradt, and J. R. Cordy, "Software clone management towards industrial application (dagstuhl seminar 12071)," *Dagstuhl Reports*, vol. 2, no. 2, pp. 21–57, 2012.

[26] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, "Automated scheduling for clone-based refactoring using a competent GA," *Software: Practice and Experience*, vol. 41, no. 5, pp. 521–550, 2011.

[27] Lucia, D. Lo, L. Jiang, and A. Budi, "Active refinement of clone anomaly reports," in *International Conference on Software Engineering*, 2012, pp. 397–407.

[28] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can I clone this piece of code here?" in *IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 170–179.

[29] M. Toomim, A. Begel, and S. L. Graham, "Managing duplicated code with linked editing," in *IEEE Symposium on Visual Languages and Human Centric Computing*, 2004, pp. 173–180.

[30] E. Duala-Ekoko and M. P. Robillard, "Clone region descriptors: Representing and tracking duplication in source code," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 1, pp. 1–31, 2010.