Empirical Software Engineering (2022) 27: 119
https://doi.org/10.1007/s10664-022-10175-w

An empirical study of data constraint implementations in Java

Juan Manuel Florez[1] · Laura Moreno[2] · Zenong Zhang[1] · Shiyi Wei[1] · Andrian Marcus[1]

Accepted: 10 May 2022 / Published online: 16 June 2022

Abstract

Software systems are designed according to guidelines and constraints defined by business rules. Some of these constraints define the allowable or required values for data handled by the systems. These *data constraints* usually originate from the problem domain (e.g., regulations), and developers must write code that enforces them. Understanding how data constraints are implemented is essential for testing, debugging, and software change. Unfortunately, there are no widely-accepted guidelines or best practices on how to implement data constraints. This paper presents an empirical study that investigates how data constraints are implemented in Java. We study the implementation of 187 data constraints extracted from the documentation of eight real-world Java software systems. First, we perform a qualitative analysis of the textual description of data constraints and identify four data constraint types. Second, we manually identify the implementations of these data constraints and reveal that they can be grouped into 31 *implementation patterns*. The analysis of these implementation patterns indicates that developers prefer a handful of patterns when implementing data constraints. We also found evidence suggesting that deviations from these patterns are associated with unusual implementation decisions or code smells. Third, we develop a tool-assisted protocol that allows us to identify 256 additional trace links for the data constraints implemented using the 13 most common patterns. We find that almost half of these data constraints have multiple enforcing statements, which are code clones of different types. Finally, a study with 16 professional developers indicates that the patterns we describe can be easily and accurately recognized in Java code.

Keywords Business rule · Data constraint · Empirical study · Code pattern · Discourse analysis

Communicated by: Alexandre Bergel

✉ Juan Manuel Florez
jflorez@utdallas.edu

Extended author information available on the last page of the article.

1 Introduction

Most software systems are designed to automate processes that are described by business rules. Business rules are therefore fundamental to the development process, as they encapsulate the knowledge that is necessary to formulate the requirements of software systems. Eliciting and explicitly referencing business rules helps ensure that the finished software fulfills its goals (Witt 2012). Indeed, business rules have even been called "first-class citizen[s] of the requirements world" (Business Rules Group 2003). Business rules may originate from multiple sources, and in most cases are formulated in response to external factors (e.g., policies, regulations, and industry standards) (Witt 2012). Not only is it important to correctly implement these business rules to comply with applicable regulations (Rempel et al. 2014), but the traces between the business rules and their implementations should also be made explicit to facilitate maintenance in the inevitable case that these rules change (Cemus et al. 2015; Wiegers and Beatty 2013; Cerny and Donahoo 2011).

Unfortunately, business rules are rarely documented and traced thoroughly enough (Witt 2012; Wiegers and Beatty 2013). Even when that is the case, external documentation or traces often become out of sync with other artifacts. This is a known open problem in software engineering and traceability (Rahimi et al. 2016; Dömges and Pohl 1998; Cleland-Huang et al. 2014b), and makes it so that source code is the only artifact that can be reliably assumed to contain this knowledge. Consequently, a significant body of work has sought to reverse-engineer business rules from existing systems (Hatano et al. 2016; Cosentino et al. 2012, 2013; Huang et al. 1996; Sneed and Erdös 1996; Wang et al. 2004; Sneed 2001; Chaparro et al. 2012). Such approaches depend on developer involvement (e.g., finding the relevant variables) and assumptions about how the rules are implemented. For example, a common assumption is that rules are always implemented as conditional branches in the source code (Hatano et al. 2016). However, these assumptions are not based on empirical evidence. We argue that studying how developers implement business rules, identifying patterns and good practices, is important not only for advancing reverse engineering, but also for improving the process of implementing business rules in the first place.

Software engineering textbooks and research papers describe many software design and programming best practices (or anti-patterns), which are usually geared towards high-level issues (e.g., system decomposition or naming conventions) or towards control and data flow organization for specific types of operations (e.g., design patterns) (Gamma et al. 1995; Larman 2005; Fowler 2018). In addition, companies and open source communities have their own coding standards, informed by their experiences. However, there are no such prescribed solutions or best practices when it comes to implementing business rules. Existing literature offers guidance on how to formulate these rules, but not on how to implement them (Wiegers and Beatty 2013; Witt 2012).

In this paper, we focus on analyzing the implementations of one particular type of business rules, that is, *data constraints*. A data constraint is a restriction on the possible values that an attribute may take (Wiegers and Beatty 2013; Witt 2012). While all data used in a software system are subject to constraints, we focus on the constraints stemming from the business rules of the problem domain that a software system occupies. For example, "*[the maximum frequency] is greater than the Nyquist frequency of the wave*" (Swarm 2021) is a constraint on seismic waveform data, while "*the patient is three calendar years of age or older*" (iTrust 2021b) is a constraint on healthcare data. For simplicity, in the remainder of the paper, when we refer to *constraints*, we imply *data constraints*.

The study of data constraints is important because they are described in many business rules taxonomies found in the literature (Wan-Kadir and Loucopoulos 2004; Wiegers and Beatty 2013; Hay and Healy 2000). Moreover, data constraints are common in the specifications of safety-critical systems (Mäder et al. 2013). These constraints are subject to change as business rules and regulations change. Hence, it is essential that developers can easily (or even automatically) change, test, and verify the code implementing the constraints.

In theory, there are countless ways in which one can implement a given data constraint in a given programming language. However, we posit that developers, guided by their experience, are likely to converge towards an identifiable set of patterns when implementing data constraints. The existence of such patterns would allow for the definition of best practices (in the vein of design patterns), and would support reverse engineering, traceability link recovery, testing, debugging, and code reviews, among other applications.

This paper presents an empirical study of data constraint implementations in Java. We extracted 187 constraints from eight real-world open-source systems and used open coding (Miles et al. 2014) to categorize them into four constraint types (Section 4). Then, we manually traced each constraint to its implementation and categorized them into 30 *data constraint implementation patterns* (Section 5). We found that 15 patterns are used frequently and account for the implementation of most constraints in our data set, while the remaining patterns appear rarely in our data. The data also indicate that certain patterns are more likely to implement certain types of constraints. In addition, we found evidence that deviations from these "most likely" patterns are signs of code smells or unusual implementation decisions.

Going further, we developed a tool-assisted protocol to identify additional statements that enforce the manually traced constraints. This protocol is applicable to constraints that are implemented with 13 of the most frequently used implementation patterns, which cover 163 (87%) of the constraints in our data set. Using this tool-assisted protocol, we recovered 256 additional statements that enforce 71 of the 163 constraints. The analysis of the new links shows that 44% of the 163 constraints are enforced in at least two different locations in the source code. In most of these cases (93%), the different statements use the same pattern, which indicates that type 1 and type 2 code clones exist in these implementations (Section 6). A recall assessment of the tool-assisted protocol resulted in the discovery of 7 additional enforcing statements (for five constraints) and one additional implementation pattern.

Finally, we conducted a study with 16 professional Java developers, where we asked them to identify the patterns used in the implementations of a set of constraints. The developers were presented with 22 constraints implemented with 10 patterns. They correctly identified the pattern of the implementations with an accuracy of 91.1%, on average. The results indicate that the patterns are well defined and identifiable in the code with high accuracy.

The main contributions of the paper are:

1. A catalog and analysis of 31 data constraint implementation patterns in Java.
2. A set of 450 curated line-of-code-level traceability links from 187 data constraints definitions to their implementations, in eight real-world Java systems. These links were generated partly manually and, in part, by using a novel tool-assisted protocol.

The implementation patterns catalog and the data used to construct it is publicly available for future development (Florez et al. 2022).

As the first study to investigate the implementation of data constraints, we expect that our results will enable new avenues of research, as well as advancing the practice of

software engineering. The constraint types and implementation patterns we defined will allow for a more focused approaches to automated test generation, when testing the enforcement of data constraints. Information on the implementation of the data constraints will help during code reviews when such constraints are changed. The ability to formally describe the implementation of data constraints will help in defining new approaches for automated traceability link recovery. The implementation patterns catalog will enable the creation of best practices for data constraint implementation, just as other types of patterns in software engineering did. Finally, we anticipate that our study protocol will be used as a template to study other types of constraints or business rule implementations.

The remainder of the paper is organized as follows. Section 2 introduces a motivating example, which shows and discusses the Java implementation of a particular data constraint. Section 3 presents the three specific research questions we address in this empirical study. Sections 4, 5, 6, and 7 describe the data, protocols and analyses we performed to answer each research question, respectively. They also present the results and provide answers to each research question. Section 8 discusses the threats to validity and limitations of the study, while Section 9 presents the related work. Finally, conclusions and future work are in Section 10. The paper includes the catalog of the 31 constraint implementation patterns as an Appendix. A subset of the most frequent ones, which fits on one page, is also included in the paper as a table, to ease reading and understanding.

2 Motivating Example

We present the implementation of one data constraint extracted from a use case of iTrust, a healthcare system widely used in traceability research (Zogaan et al. 2017). This use case evaluates whether a patient is at risk of suffering from type 2 diabetes according to multiple risk factors, one of them being: *"Age over 45"* (iTrust 2021a). The data constraint expressed in this excerpt is $age > 45$.

Listing 1 contains the code relevant to the implementation of this constraint. In the `Type2DiabetesRisks` class, the `getDiseaseRiskFactors()` method defines and adds four risk factors in lines 4 to 7, among which we find the relevant line based on the keyword *age* and the constant 45 in line 4. The constructor of the `AgeFactor` class assigns the constant `45` to its field called `age`. Examining the usages of the `getDiseaseRiskFactors()` method, we see that after being initialized, the `hasRiskFactor()` method is called on each risk factor (line 18). This method delegates the constraint checking to the `hasFactor()` method. Finally, line 31 checks the constraint, which appears in the `hasFactor()` method of the `AgeFactor` class.

While lines 4, 18 , and 31 in Listing 1 are all part of the implementation of the constraint, we consider that the statement that actually enforces the constraint is the last one. We call such a statement the *constraint enforcing statement*. For simplicity, in the remainder of the paper, when we refer to *enforcing statement*, we imply *constraint enforcing statement*. We provide relevant definitions in Section 5.1.

This example shows that it is possible to identify a single enforcing statement for a data constraint which consists of a single expression in the code (`patient.getAge() > age`). However, the data relevant to the constraint are defined in code locations different from where the constraint is being enforced. Specifically, `age` is a field of class `Patient`, and the constant `45` is a parameter to the constructor call of the AgeFactor `AgeFactor` class. This means that the enforcing statement alone is not sufficient to describe the implementation

```
// Class Type2DiabetesRisks
protected List<PatientRiskFactor> getDiseaseRiskFactors() {
  List<PatientRiskFactor> factors = new ArrayList<>();
  factors.add(new AgeFactor(patient, 45)); // <<
  factors.add(new WeightFactor(currentHealthRecord, 25));
  factors.add(new HypertensionFactor(currentHealthRecord));
  factors.add(new CholesterolFactor(currentHealthRecord));
  return factors;
}

...

// Class RiskChecker
public boolean isAtRisk() {
  int numRisks = 0;
  List<PatientRiskFactor> factors = getDiseaseRiskFactors();
  for (PatientRiskFactor factor : factors) {
    if (factor.hasRiskFactor()) // <<
      numRisks++;
    if (numRisks >= RISK_THRESHOLD)
      return true;
  }

  return false;
}

...

// Class AgeFactor
public boolean hasFactor() {
  return patient.getAge() > age; // <<
}
```

Listing 1 Code implementing the constraint in the motivating example

of a constraint. In this case, the implementation consists of (at least) the statement that enforces the constraint (`patient.getAge() > age`), and the definitions of `Person.age` and of the constant `45`.

We can further note that a given enforcing statement may correspond to multiple constraints, i.e., any other uses of the `AgeFactor` class would correspond to different constraints but use the same code for enforcing them. For example, `AgeFactor` initialized with the value `30` would check a different constraint (i.e., *age* > 30) but would use the same code to do so. This is a situation when *multiple constraints use the same enforcing statement*.

Finally, the constraint may need to be enforced in other features of the system. For example, the same risk factor is also used in determining whether a patient is at risk of suffering heart disease, and the `HeartDiseaseRisks` class contains a check for *"Age over 45"*. The implementation in this case is identical, i.e., `AgeFactor` is initialized with the constant 45. This is a situation when *a constraint has multiple enforcing statements or uses*.

This example illustrates that, even though a constraint implementation can be traced to a single enforcing statement and corresponding data definitions, understanding how data constraints are implemented is further complicated by *the need to disambiguate different constraints that use the same code*, and *locating different enforcing statements of the*

same constraint. We seek to build an understanding of data constraint implementations by identifying patterns both in their textual description and their implementation.

3 Research Questions

Based on our collective experience, we posit that many unrelated constraints are implemented in similar ways. This also implies that there should be a relatively small number of forms that constraint implementations normally take. However, little is known about the space of data constraint implementations, which is the motivation for conducting this empirical study.

The main goal of our study is understanding how data constraints are implemented, and we formulate three specific research questions (RQ), addressing three distinct aspects of data constraints and their implementations:

RQ1: *What types of data constraints can be found in textual artifacts of software projects?* For answering RQ1, we perform a qualitative analysis of the textual description of data constraints and identify the kinds of restrictions they specify (Section 4).
RQ2: *What patterns do developers follow when implementing data constraints in Java?* For answering RQ2, we manually identify the implementations of the data constraints. Then, we perform a qualitative analysis for identifying commonalities and differences between them (Section 5).
RQ3: *What are the differences between multiple enforcing statements of the same constraint?* For answering RQ3, we implement a tool that allows us to semi-automatically identify enforcing statements additional to those identified manually before. Then, we analyze the multiple enforcing statements of the same constraint, when they exist, to understand their rationales (Section 6).

In order to validate our constraint implementation pattern catalog we conduct a study with professional developers for answering the following research question:
RQ4: *How accurately can developers identify constraint implementation patterns?* For answering RQ4, we asked 16 Java developers to identify the patterns used in the implementation of 22 constraints from our data. Then, we analyze the accuracy of the developers' answers (Section 7).

4 Types of Data Constraints (RQ1)

In this section, we present the data and analyses we used to answer RQ1: *What types of data constraints can be found in textual artifacts of software projects?* We then describe the results and provide the answer to the research question.

4.1 Software Systems

The targets of our empirical study are eight open-source real-world Java systems (Table 1).

The selection criteria for the target systems were: (1) we required the systems to be real-world open-source Java software; (2) the systems needed to be used at least once in previous traceability studies; (3) the systems had to provide documentation describing the functionality or design of the system, such as user manuals, tutorials, or specifications, which included descriptions of data constraints; (4) we required the systems to be under

Table 1 Software systems used in the empirical study

System	Short name	Domain	Size (KLoC)	Textual artifacts
apache-ant-1.10.6	Ant	Build manager	282	User Manual
argouml-0.35.4	Argo	UML Modeler	154	User Manual
guava-28.0	Guava	Programming utilities	346	User Manual
httpcomponents-client-4.5.9				
httpcomponents-core-4.4.11	HTTPC	HTTP Client/Server library	28	Tutorial, Specifications
jedit-5.6pre0	jEdit	Text editor	196	User Manual
joda-time-2.10.3	Joda	Date/time library	146	User Manual
rhino-1.6R5	Rhino	JavaScript interpreter	77	Specifications
swarm-2.8.11				
volcano-core-1.4.14				
winston-1.3.4				
wwsclient-1.3.7	Swarm	Seismic wave visualizer	101	User Manual

500 KLOC in size, to make manual tracing feasible. To select the systems, we used the data provided by a recent survey that collected all data sets used in traceability studies for the past two decades (Zogaan et al. 2017). We selected all the systems from said survey that fit our criteria. We consider this to be a representative sample of software projects because it consists all the available and usable (for our purposes) research data in the field of traceability, which our study is framed in. Due to the difficulty of procuring requirements documents for open source software (Alspaugh and Scacchi 2013), we selected the textual artifacts that were available for each system, a practice common in traceability research (Eaddy et al. 2008b; Ali et al. 2011, 2012, 2013). These artifacts contain descriptions of the systems features and business rules.

4.2 Exploratory Study

The qualitative methodologies used to answer RQ1 and RQ2 (each explained in their own sections below) require data to build their conceptual foundation. We chose the iTrust system (Zogaan et al. 2017) as the source of these data, due to the fact that its requirements (in the form of use cases) are thoroughly specified and it is one of the most studied systems in traceability research.

The process started with a discussion between the authors in which it was informally agreed what would be considered a data constraint. After this discussion, an author extracted 110 constraints from the use cases of iTrust, which were then approved by the rest of the authors.

We derived three research artifacts from this preliminary study: (1) an actionable definition for a data constraint; (2) a set of four data constraint types (both presented in Section 4.3); and (3) the initial set of constraint implementation patterns for answering RQ2 (expanded in Section 5.2.1).

These data were used to develop a conceptual framework for our empirical study. Due to the fact that iTrust is not a real-world system (it is a student project), and thus does not satisfy our inclusion criteria, these data are not included in the analyses to answer our research questions.

4.3 Constraint Extraction and Categorization

One author extracted the data constraints from the textual artifacts of the target systems (see column 5 in Table 1). A total of 198 constraints were extracted, out of which 11 were discarded after a discussion during the categorization (see below), leaving 187 constraints to be used for the study.

Using the data from the exploratory study, we defined a protocol that describes the characteristics of the constraints that we intend to study. The full protocol is included in our replication package (Florez et al. 2022), but we present a summary here.

We define a **data constraint** as a restriction on the set of possible values of an abstract variable. We define an **abstract variable** as a value in the software domain that corresponds to either a piece of data in the real world or a configuration property of the system. Since these variables exist in the domain, they only have a name and a value (i.e., they can be of any type) and are independent of any underlying implementation.

We look for sentences that explicitly restrict the set of allowable values for an abstract variable. The ways in which these restrictions may appear in the textual artifacts used for the study include:

- directly specifying the value (e.g., "*default value of X is Y*");
- specifying an exhaustive set or range of values (e.g., "*X must be < Y", "A must be one of X, Y, or Z*");
- Implied boolean values (e.g., "*X is enabled/X is disabled*", "*X is set/X is unset*", "*X was found/X was not found*");

This list was not intended to be exhaustive, and the author extracting the constraints was instructed to identify other ways in which data constraints are formulated in the textual artifacts. However, no other ways were identified.

We exclude sentences that may be confused with data constraints but are not data constraints, such as:

- concrete examples or hypothetical scenarios ("*If the weight is 5, for example*");
- non-exhaustive sets of values ("*If i is an integer like 1, 2, 5, etc.*");
- actions or decisions outside of the system's control ("*If the patient displayed is not the one that the user intended, the user will go back to the search screen*");
- user intentions or possibilities ("*The system can be configured to exit if an input is invalid*");
- required values: saying that a value is NOT required does not constitute a constraint ("*The name field is not required in this form*").

The number of constraints we identified varies across systems (see column 6 in Table 3), because some of the artifacts define fewer data constraints than others.

The four constraint types (see next section) were derived from our exploratory study on the iTrust system. These were derived using open coding (Miles et al. 2014), which is an iterative process. This process was conducted by two authors (coders). Both coders shared a codebook which contained the agreed-upon codes (i.e., constraint types) at any given point, and was initially empty. The constraints were organized into categories according to the number of operands (i.e., abstract variables) involved and the type of restriction that the constraint imposes on them.

For example, one of the first iTrust constraints coded in the exploratory study was "*the security question/answer has been set (it is not null)*", which was assigned the code

`attribute-not-empty`, meaning that constraints assigned this code require the attribute to have a value. Later, when the constraint "*the patient has never stored a security question/answer*" was assigned the code `attribute-empty`, meaning that the constraint requires the attribute to not have a value assigned.

Open coding requires data to be systematically re-evaluated after a new code is introduced. After adding this second code, both coders went back over the already coded constraints and judged whether the newly introduced code fit any of them. This lead to the two previous codes being merged into the `DualValue Comparison` constraint type, as both deal with conditions that can take one of two values (e.g., true, false).

The coders then used the resulting codebook to categorize each of the 198 constraints. The constraints were split evenly and coded independently by a single coder, while each coder verified each other's work, and disagreements were solved through a discussion. This is an adaptation of gold-standard coding, in which two coders evenly split the data set, with additional *reliability coders* verifying the work. In our case, each coder acted as each other's reliability coder (Syed and Nelson 2015).

The coders had disagreements on 11 of the labeled sentences, and after a discussion, it was decided that these were not valid data constraints. For example, the Guava manual contains the sentence *"If your cache should not grow beyond a certain size"* (Google 2021a), which is not a constraint according to our protocol (describes a user intention or possibility).

4.4 Results

We identified four data constraint types: `Value Comparison`, `Dual Value Comparison`, `Categorical Value`, and `Concrete Value`. Table 2 defines each type and provides examples.

We encountered constraints of different types in each system, yet not all constraint types appeared in the artifacts of all systems. The distribution across types and systems of the 187 constraints is presented in Table 3.

The most common constraint type in our data set is `Value Comparison`, in which two values are compared using an operator. `Dual Value Comparison` is a subtype of `Value Comparison`, where the operator is equality, and the property can only take one of two mutually-exclusive values. This subtype is important, because often only one of the two mutually-exclusive values is explicit in the constraint description, but it is easy to infer the missing one. The same inference is not possible for constraints of the more generic type, `Value Comparison`. `Concrete Value` directly states the value that an attribute should have. Finally, `Categorical Value` does not specify or compare a specific value. Instead, it restricts the value of an attribute to a finite set of items. Note that this last type only ensures that the value is an item of the given set. A constraint requiring that a value is equal to *one* of the items in the set would instead be of the `Value Comparison` type.

RQ1 answer

We identified four data constraint types: `Value Comparison`, `Dual Value Comparison`, `Categorical Value`, and `Concrete Value`. They differ from one another by the number of operands they include and the type of operations applied to them.

Table 2 Data constraint types

Name: `Value Comparison`.

Definition: The value of an attribute X is constrained by the value of another attribute Y (or constant C). Equality or relational operators are used to determine if X is greater than, less than, equal to or not equal to Y (or C). While equality operators apply to values of all types, relational operators do not apply to all types of values (e.g., binary or categorical).

Example constraint text: "*While SWARM will allow the maximum frequency to be set to any positive value greater than the minimum frequency, this value will adjust automatically if it is greater than the Nyquist frequency of the wave being manipulated.*" (Swarm 2021)

Example constraint(s):

$max\,frequency > 0$

$max\,frequency > min\,frequency$

$max\,frequency > wave\,Nyquist\,frequency$

Name: `Dual Value Comparison`.

Definition: An attribute has only two possible, mutually-exclusive values (e.g., true/false, on/off, null/not-null). Equivalent to `Value Comparison` if the operator is equality and there are only 2 possible values.

Example constraint text: "*If configuration file is not available or readable it will default to 'UTC'.*" (Swarm 2021)

Example constraint(s):

$file\,available == \texttt{false}$

$file\,readable == \texttt{false}$

Name: `Categorical Value`.

Definition: The value of a categorical attribute is constrained to a finite set of values.

Example constraint text: "*A target has the following attributes: [...] onMissingExtensionPoint: What to do if this target tries to extend a missing extension-point. (fail, warn, ignore)*" (Apache Ant 2021)

Example constraint(s):

$onMissingExtensionPoint \in \{fail, warn, ignore\}$

Name: `Concrete Value`.

Definition: The constraint directly dictates what value the property should have.

Example constraint text: "*The GregorianJulian calendar is a combination of two separate calendar systems, the Gregorian and the Julian. The switch from one to the other occurs at a configurable date. The default date is 1582-10-15, as defined by Pope Gregory XIII.*" (Joda-Time 2021)

Example constraint(s):

$switch\,date$ is *1582-10-15*

5 Constraint Implementation Patterns (RQ2)

In this section, we describe the data, protocols, and analyses we used for answering RQ2: *What patterns do developers follow when implementing data constraints in Java?* We then describe the results and provide the answer to the research question.

Table 3 Distribution of constraint types by system

System	Categorical Value	Concrete Value	Dual Value Comparison	Value Comparison	Total
Ant	6	7	18	1	32
Argo	3	14	11	·	28
Guava	·	·	8	3	11
HTTPC	1	3	12	5	21
jEdit	1	3	18	11	33
Joda	2	5	·	6	13
Rhino	3	1	6	12	22
Swarm	2	5	12	8	27
Total	18	38	85	46	187

5.1 Manual Tracing Protocol

Answering this research question requires identifying the implementation of the 187 constraints that we extracted in the previous section. We borrow terminology from software traceability research and call this activity *tracing*. This type of tracing is common in requirements-to-code traceability link recovery and feature location work, among others (De Lucia et al. 2012; Razzaq et al. 2018). Consequently, a *trace* is a link between the description of a constraint (i.e., its source) and the code that implements it (i.e., its target). Tracing was performed by six Computer Science graduate students: five M.S. students, with at least two years of industry experience each, and one Ph.D. student. We refer to the six students as *tracers* from this point forward.

Each tracer received one hour of training from one of the authors, and was compensated with $15 per hour for the time spent in training and tracing. The tracers worked at home, using an online spreadsheet to record their traces. Each trace consisted of its source constraint (i.e., the constraint description) and its target code statements (i.e., the enforcing statements and data definitions), whose identification protocol is described further in this section.

The tracers did not communicate with one another. Each constraint was traced by two tracers independently, and tracing proceeded one system at a time. The tracers were instructed to ignore all test code.

For each system, the tracers received the following data:

1. The source code of the system.
2. A document with details about the system design and architecture, such as a list of the most important classes and their responsibilities. This was assembled by one author according to the documentation of the system and a code inspection.
3. The list of data constraints to be traced.

In addition, for each constraint, we provided the tracers with:

1. The section of the textual artifact where the constraint is described, e.g., a section of the user manual or specification.
2. The text that describes the constraint, e.g., "*Any Content-Length greater than or equal to zero is a valid value*" (HTTP Working Group 2021).

3. A simplified version of the constraint, e.g., "*Content-Length* ≥ 0". This was created by one of the authors, who rephrased the textual description of the constraint in a simpler language, using mathematical notation where possible. This information was provided to ease understanding of the constraint and avoid confusion or ambiguity.
4. A scenario to be used for tracing. The scenario corresponds to a feature of the system that relies on the constraint. It was extracted by one of the authors based on the constraint's context, e.g., *"Validating an HTTP request"* for the example above. This information ensures that the relevant implementation is found, as a single constraint may have multiple enforcing statements, corresponding to different features or scenarios (see Section 2). In the case of a tracer identifying multiple implementations of a constraint, they were instructed to choose only the one corresponding to this scenario.

The tracers were allowed to use any tool or information source to perform the tracing, although the use of an IDE was recommended.

5.1.1 Structure and Granularity of Constraint Implementations

When tracing domain level concepts to their implementation in the code, one important aspect to establish is the granularity of the links (i.e., the source and the target). As mentioned above, our source corresponds to the textual description of a single data constraint, typically expressed in a sentence. We discuss here the structure and granularity of the target (i.e., source code elements) of the traces.

Existing work on traceability link recovery and feature location usually links sources to functions, methods, classes, files, *etc.* (De Lucia et al. 2012). In other words, they use the granularity provided by the file system or the decomposition mechanism of the program language. For our study, such a granularity is not suitable. Recall that our goal is to study *how* data constraints are implemented in Java. For example, determining that the constraint *"Age over 45"* is implemented in class `AgeFactor` will tell us *where* it is implemented but not *how*. We need finer-grained traces (i.e., to line-of-code level) to analyze and understand *how* the constraints are implemented. Conversely, as evidenced by the motivating example (Section 2), tracing a constraint to a single enforcing statement or expression can be ambiguous (in the case that the same code is used to enforce multiple constraints). We aimed to identify the minimum number of statements that unambiguously correspond to a given constraint in its context (i.e., the associated feature).

For this reason, we instructed the tracers to locate both the *constraint enforcing statement* and the *data definition statements* for each constraint, as the tracing targets. *Constraint enforcing statements* check the constraint (e.g., `patient.getAge() > age`) or ensure that it is enforced (e.g., by directly defining the value), while *data definition statements* define the data used therein (e.g., `Person.age`, `45`). We provided additional instructions for helping the tracers identify these statements.

5.1.2 Identifying the Constraint Enforcing Statements

As we saw in our motivating example from Section 2, several statements may be used for implementing a constraint. Among them, we consider the enforcing statement the one that is at the lowest granularity level, that is, it cannot be decomposed any further (e.g., tracing into a method invoked from the statement). Specifically, the tracers applied the following procedure for identifying the enforcing statements from those that are involved in implementing the constraint. Let s be a candidate enforcing statement for a given constraint:

1. If s contains no method invocation, then s is a constraint enforcing statement. Otherwise, investigate the method M invoked from s.
2. If a candidate enforcing statement s' exists in M, then repeat step (1) with s'. Otherwise, s is a constraint enforcing statement.

For example, when tracing the constraint *"[Spectrogram maximum frequency] is greater than the Nyquist frequency of the wave"* (Swarm 2021), the tracer finds the call to `processSettings()`. Inside this method, there exists the statement `if(settings.spectrogramMaxFreq > wave.getNyquist())`. The call to `processSettings()` is not the enforcing statement, because there is another candidate enforcing statement inside the method. Since no statement in the method `getNyquist()` checks that the max frequency is greater than the Nyquist frequency, `if(settings.spectrogramMaxFreq > wave.getNyquist())` is the enforcing statement for the constraint in this example.

The enforcing statements should be in the code of the target system, rather than in the code of third-party libraries or the Java standard library. If a constraint was enforced outside the system code, tracers were instructed to trace to the statement(s) that referred to the external enforcement (i.e., an invocation to the library method). For example, the constraint "*the Iterable is empty*" (Google 2021b) is implemented in the statement `E minSoFar = iterator.next();`. The checking is done inside the `next()` method, which is implemented in the Java standard library.

5.1.3 Identifying the Data Definition Statements

Using the enforcing statements, the tracers found the data definition statements by identifying the operands relevant to the constraint. Following the previous example, if the enforcing statement is `if(settings.spectrogramMaxFreq > wave.getNyquist())`, then the operands relevant to this constraint are the return value of the method `getNyquist()` and the field includegraphicss10664-022-10175-wfmbbj.eps. The tracers were asked to trace the data definition statements according to the following rules:

- If the data is accessed from a field directly (`obj.field`) or a getter method that returns the value unchanged (`obj.getField()`), then the *field declaration* is traced.
- If the data is computed in a method, then the *method declaration* is traced. For example, if the operand is the return value of the method `obj.calculateValue()` and `value` does not exist as a field in the class of `obj`, then the data definition statement is the declaration of the `calculateValue()` method.
- If the data comes from a library class, then the *method call* is traced. For example, if the operand is the local variable `value` which is defined as `intvalue = request.getValue()`, where `request` is an instance of a library class, the data definition statement is the statement where `request.getValue()` is called.
- If the data corresponds to a literal defined in a method, then the *assignment* is traced. For example, if the operand is the variable `value` defined as `intvalue = 100`, this statement is traced. If the literal `100` is used directly, the data definition is the value `100`.
- If the constraint directly refers to a method parameter (e.g., a library entry point), then the *method parameter definition* is traced. For example, the constraint "*the value is not null*" (Google 2021b) refers specifically to the parameter of the `checkNotNull()` method.

This also applies when it is not possible to directly determine the caller of the method where the enforcing statement is located, e.g., when the method implements a listener interface called by the *Swing* library.

The data definition statements are later used for answering RQ3 in Section 6.

5.1.4 Trace Validation

The final traces were decided jointly by two authors. The authors applied their knowledge of the systems and the definitions presented in this section to judge whether or not the semantics of the statements marked by the tracers correspond to the constraint implementation. This kind of approach has been applied in traceability studies when it is not possible to seek the guidance of the developers of the system (Eaddy et al. 2008a, b).

If the two tracers produced overlapping statements for a constraint, the trace was defined by the overlapping statements, once confirmed by one of the authors. We chose this approach as opposed to selecting the union of all statements, as we are interested in identifying atomic implementations for each constraint. As an example, the enforcing statement of the ArgoUML constraint *"[Show multiplicities] is selected" is shown in* Listing 2. The first tracer selected lines 1, 2, and 3 as the trace, while the second tracer selected only line 3. Setting the enforcing statement trace to only line 3 results in an atomic trace, as the other lines do not enforce the given constraint, despite being part of the same statement.

Disagreements (i.e., the tracers identified disjoint sets of statements) were resolved by one author, with the trace being later verified and validated by another author. Overall, only 13 (7%) traces resulted in disagreements that were resolved through the discussion between one author and the tracers. Some of the disagreements were caused by the misunderstandings of the code semantics (e.g., the variable being checked is related, but it is not the one that the constraint refers to). In other cases, the enforcing statement was correct but did not match the scenario outlined in the documentation.

In the end, each of the 187 constraints was traced to one enforcing statement and the corresponding data definition statements. As we discussed in Section 2, some constraints may be involved in several features of the system, which may lead to multiple traces. Here, we focused on providing a single trace per constraint, so we produced 187 traces.

5.2 Identifying Patterns in Constraint Implementations

While the 187 constraint implementations we traced are different from one another, they share structural properties. We grouped them into categories according to structural properties they share. We used open coding (Miles et al. 2014) to define these categories, based exclusively on the data (i.e., a descriptive approach as opposed to a prescriptive one).

```
1 if ((multiplicity != null)
2         && (multiplicity.length() > 0)
3         && showMultiplicity) {
4     sb.append("[").append(multiplicity).append("]").append("␣");
5 }
```

Listing 2 Example of atomic trace. The highlighted code is the trace by tracer one, and the underlined code is the trace by tracer two

5.2.1 Coding Protocol

Open coding results in the creation of a set of *codes*, which we denominate *constraint implementation patterns* (CIPs). From here on, we refer to them as *patterns* or CIPs.

In order to determine an initial set of codes (i.e., patterns), we used the 110 iTrust constraints from our exploratory study (Section 4.2). One of the authors traced these constraints to their implementations, in a similar manner to the protocol described above. Upon analysis of the traces, the author identified 27 patterns, which served as the initial set for the coding. Note that the iTrust traces are not included in this study.

The CIPs have the following components: *name*, *description*, *statement type*, *parts*, and *example*. The *name* and *description* of a pattern support identifying and understanding the meaning of each pattern. The pattern's *statement type* describes the type of the enforcing statement (e.g., expression, method call, or variable definition), and each pattern is defined only on a particular *statement type*. The pattern's *parts* are structural programming elements from the enforcing statement. These *parts* describe number and types of the operands and operators and differentiate one pattern from another.

The *statement type* and *parts* derived from an enforcing statement determine how to label it (i.e., which patterns it follows). For example, the enforcing statement `maxFrequency > nyquistFrequency` is implemented using the `binary comparison` pattern, because it is an expression and `maxFrequency`, `>`, and `nyquistFrequency` match the *parts* $\{variable1, op \in \{>, \geq, <, \leq, =, \neq\}, variable2\}$ respectively (see Table 4). Finally, the examples provide an illustration of each pattern.

Two authors coded the 187 traces from Section 5.1. Each enforcing statement was categorized according to (1) what type of statement it is, and (2) the number and types of operands and operators involved in it. If no existing pattern matched the type of the enforcing statement or the amount of operands and operators, a new pattern was created. As the coding progressed, patterns were renamed and/or merged, and the previously coded data were re-checked against the new CIPs.

Each trace was coded by one author. Each coded trace was verified by the other coder, discussing any disagreements with the original coder. This is also an adaptation of *gold-standard coding* in the same way as explained in Section 4.3. In 16 (9%) cases, there were disagreements that were resolved through discussions. It is not possible to report standard agreement measures (e.g., Cohen's Kappa (Cohen 1960), Krippendorff's Alpha (Krippendorff 2004)) because these require knowing in advance the set of all possible codes, with the purpose of estimating the percentage of agreement that happened by chance. Since open coding allows for the creation of new codes to fit new data, and the process is iterative, it is not possible to determine what the set of all possibilities would be at any given point. Instead, we increase confidence in the reliability of our catalog by both using the methodology of gold-standard coding (Syed and Nelson 2015), and having the rest of the authors approve the final catalog.

5.3 Results and Analysis

The open coding resulted in the definition of 30 CIPs. Although we discovered an additional implementation pattern while answering RQ3 (see Section 6.3.2), our analysis in this section is limited to the 30 CIPs discovered while answering RQ2.

Table 4 Seven most frequently used constraint implementation patterns form our CIP catalog. The remaining CIPs are available in an Appendix and the replication package (Florez et al. 2022)

CIP name: `boolean property`.

Description: A variable of type Boolean is checked in a Boolean expression.

Statement type: Boolean expression.

Parts: {*variable*}

Example:

Instance: `if(buffer.isModified)`

Parts: {`isModified`}.

CIP name: `binary comparison`.

Description: Two variables are compared using one of the relational operators(*equals*, *does not equal*, *greater than*, *less than*, *greater than or equal to*, *less than or equal to*). Use of the `equals` method is considered an operator in this case. The types of the operands may be of any type for which these operations are allowed (e.g., *greater than* cannot be applied to Boolean values).

Statement type: Relational expression.

Parts: {*variable1*, *op* $\in \{>, \geq, <, \leq, =, \neq\}$, *variable2*}

Example:

Instance: `if(maxFreq > wave.getNyquist())`

Parts: {`maxFreq`, >, `wave.getNyquist()`}.

CIP name: `constant argument`.

Description: A literal value is passed as a parameter to a method call.

Statement type: Method call.

Parts: {*method*, *constant*}

Example:

Instance: `settings.setShowVisibilities(false);`

Parts: {`setShowVisibilities`, `false`}.

CIP name: `null check`.

Description: A nullable variable is tested for nullity using the `==` or `!=` operators. The `null` keyword is not considered a part of the pattern because it will always appear in instances of it.

Statement type: Relational expression.

Parts: {*variable*}

Example:

Instance: `if(name == null)`

Parts: {`name`}.

CIP name: `assign constant`.

Description: A literal value is assigned to a variable.

Statement type: Assignment.

Parts: {*variable*, *constant*}

Example:

Instance: `refreshInterval = 15;`

Parts: {`refreshInterval`, `15`}.

Table 4 (continued)

CIP name: `binary flag check`.

Description: An integer variable is used as a bit field and checked with a bitwise operator against a constant integer.

Statement type: Relational expression.

Parts: {*variable*, *constant*}

Example:

Instance: `flag & NEW_FILE == NEW_FILE`

Parts: {`flag, NEW_FILE`}.

CIP name: `if chain`.

Description: A chain of ifs is used like a switch on a variable, checking against its possible values. Each `if` clause uses the == operator or `equals` method.

Statement type: If statement.

Parts: {*variable*}

Example:

Instance: `if(onset == EMERGENT) {...} else if(onset == IMPULSIVE) {...} else if ...`

Parts: {`onset`}.

5.3.1 CIP Catalog

Table 4 shows part of the catalog, containing the 7 most commonly used CIPs. The complete CIP catalog, including all identified CIPs, is included as an Appendix and also in our replication package (Florez et al. 2022). Four of the 187 (2%) constraint implementations rely on external libraries. While we traced these constraints to the relevant library method call (as explained above), they were not taken into account when defining the CIPs. For this reason, we limit the following analysis to 183 constraints (i.e., excluding the 4 enforced externally).

Table 5 shows the distribution of pattern instances across systems for the 30 CIPs. Out of the 30 patterns, 15 are used to implement 168 of the 183 constraints in our data, and we consider them *frequent patterns.* 15 of the patterns have only one instance in our data and we consider them *rare patterns*. The two most common patterns (i.e., `binary comparison` and `boolean property`) appear in nearly every system, and they alone account for 50% of all constraint implementations in our data. We consider these *very frequent patterns*.

5.3.2 Catalog Analysis

Rare Patterns One author examined all instances of the 20 patterns that are either rare or appear only in one system (see Table 5) and categorized them according to the reason why the implementation used this pattern as opposed to a more common one. These explanations were derived based on the author's understanding of the enforcing statement and the system architecture. Note that while we can attempt to explain why a pattern exists in one particular system, we cannot explain why a pattern would be absent from a given system. The explanations are:

Table 5 Distribution of pattern instances by system

Pattern	Ant	Argo	Guava	HTTPC	jEdit	Joda	Rhino	Swarm	Total
`boolean property`*	12	8	3	3	10	·	2	11	49
`binary comparison`*	2	·	3	8	12	6	8	7	46
`constant argument`*	2	9	·	1	·	5	1	2	20
`null check`*	3	·	2	5	1	·	3	1	15
`assign constant`*	3	1	·	1	1	·	·	3	9
`if chain`*	1	3	·	·	·	·	·	1	5
`binary flag check`*	·	·	·	·	2	1	2	·	5
`equals or chain`*	3	·	·	·	·	·	1	·	4
`properties file`	·	1	·	·	2	·	·	·	3
`switch-len char`*r	·	·	·	·	·	·	2	·	2
`self comparison`*r	·	·	·	·	·	·	2	·	2
`return constant`*r	·	·	·	2	·	·	·	·	2
`polymorphic method`r	·	·	2	·	·	·	·	·	2
`null-zero check`*r	·	·	·	·	2	·	·	·	2
`null-empty check`*	1	1	·	·	·	·	·	·	2
`switch case`r	·	·	·	·	1	·	·	·	1
`str starts`r	·	·	·	·	1	·	·	·	1
`str ends`r	·	1	·	·	·	·	·	·	1
`setter`r	1	·	·	·	·	·	·	·	1
`override value set`r	·	1	·	·	·	·	·	·	1
`null-boolean check`r	·	1	·	·	·	·	·	·	1
`mod op`r	·	·	·	·	·	1	·	·	1
`iterate-and-check literal`r	1	·	·	·	·	·	·	·	1
`index loop find`r	1	·	·	·	·	·	·	·	1
`if-return chain`r	1	·	·	·	·	·	·	·	1
`enum valueOf`r	·	·	·	·	1	·	·	·	1
`delta check`r	·	·	·	1	·	·	·	·	1
`constructor assign`r	·	1	·	·	·	·	·	·	1
`cast self-comparison`r	·	·	·	·	·	·	1	·	1
`assign class call`r	1	·	·	·	·	·	·	·	1
external	·	1	1	·	·	·	·	2	4
Total	32	28	11	21	33	13	22	27	187

*Pattern has a detector for the tool-assisted study in Section 6

r Pattern has only one instance or appears in only one system

1. *Architectural constraint* (10 patterns). The architecture of the system makes it more natural or only feasible to implement the constraint using this pattern. This is the case with the `switch-len char` pattern (only used in Rhino), which is the result of an optimization specific to the Rhino system, arising from a need to make the code more efficient. According to the developers of the system: "It is used in every native Rhino class that needs to look up a property name from a string, and does so more efficiently than a long sequence of individual string comparisons".[1] They further explain that the pattern was first devised in older versions of Java where it made a significant difference in performance, though it is not clear if that still is the case.
2. *Uncommon constraint* (4 patterns). The constraint fits one of the 4 types, but their specific semantics lends itself to one of these patterns. We would expect to see more instances of these patterns in a larger data set with constraints with similar semantics. One example is the `enum valueOf` pattern, which is an intuitive way of implementing a `Categorical Value` constraint. We hypothesize that the reason we did not observe more instances of this pattern is the relative scarcity of this constraint type in our data set.
3. *Constraint requiring specific implementation* (3 patterns). Using this pattern is the only feasible way of implementing the constraint due to its semantics or characteristics of the programming language. One example is the implementation of the constraint "*[Call to ToNumber] is NaN*" (Rhino 2021), which is simplified as "*ResultOfToNumber == NaN*". An intuitive way of implementing this constraint would be `d == Double.NaN` (i.e., an instance of the `binary comparison` pattern). However, in Java, the value of `NaN` is not equal to itself, which leads to the implementation `d != d`. We call this pattern `self comparison`.
4. *Implemented using Object-Oriented constructs* (2 patterns): this is the case for the `polymorphic method` and `override value set` patterns. While the other patterns are defined inside of code structures that span one or a few lines inside of a method, these are defined over multiple methods using object-oriented programming principles. For example, the Guava constraint *"this Optional contains a non-null instance"* is implemented in the `Optional` class and its sublclasses: `Present` and `Absent`. The `isPresent()` method is abstract in the `Optional` class and it is implemented in its subclasses, which means the constraint is checked at runtime depending on the concrete type of the `Optional` object.
5. *Interchangeable idiom* (1 pattern). The pattern `null-zero check` can be replaced with `null-empty check`, as comparing the length of the string to zero has the same semantics as comparing it to the empty string.

According to our analysis, for 11 patterns, the constraint could be implemented with one of the frequent patterns (i.e., architectural constraint, interchangeable idiom). For 7 patterns (i.e., uncommon constraint, specific implementation) we would expect these patterns to become frequent with a larger data set.

Most interesting are the two cases where the constraint was implemented using OO constructs. The rarity of this phenomenon in our data set would suggest this is a rare occurrence, which stands to reason as data constraints are conceptually simple, and as such one would not expect them to have a complex implementation spanning multiple classes. However, more research into the subject is necessary to confirm this observation.

[1] https://groups.google.com/g/mozilla-rhino/c/bdEX2Wa3pSQ/m/QXizSSdGEwAJ

Relationship Between Constraint Types and Implementation Patterns The data in Table 6 indicate that there is a correlation between certain constraint types and patterns. For example, that the two most common patterns, `boolean property` and `binary comparison` implement mostly constraints of types `Dual Value Comparison` and `Value Comparison`, respectively. This indicates that such constraints are implemented in rather predictable ways. It can be argued that `boolean property` (i.e., checking a Boolean) is an intuitive way of checking that a variable can only take two values. Likewise, `binary comparison` (i.e., comparing two variables) is an intuitive way of implementing a comparison of two values. This has clear implications for problems such as traceability link recovery, as it means that a simple heuristic-based approach could be used to retrieve highly-likely line-of-code candidates if the constraint type is known.

Another example is the `Categorical Value` type, which checks whether a value belongs to a finite set of options, and it is most frequently implemented with the `if chain` pattern, as a series of `if` statements. One can argue for using a `switch` statement instead. However, none of the implementations we examined checked constraints of the `Categorical Value` type in this way. Of course, semantically a `switch` statement is equivalent to a chain of `if` statements, but they differ structurally. It would be expected to see this constraint being enforced more often using `switch` statements or `Enums`, however it is not possible to make assumptions that are not supported by our data. This could further imply that developers do not always use the most natural programming constructs to implement a concept, evidencing the need for best practice guidelines for constraints. In turn, this observation could also be

Table 6 Distribution of pattern instances by constraint type for frequent patterns

Pattern	`Categorical Value`	`Concrete Value`	`Dual Value Comparison`	`Value Comparison`
`boolean property`	.	.	48	1
`binary comparison`	1	.	6	39
`constant argument`	.	20	.	.
`null check`	.	.	14	1
`assign constant`	.	8	1	.
`if chain`	5	.	.	.
`binary flag check`	1	.	4	.
`equals or chain`	3	.	1	.
`properties file`	.	3	.	.
`switch-len char`	2	.	.	.
`self comparison`	.	.	.	2
`return constant`	.	2	.	.
`polymorphic method`	.	.	2	.
`null-zero check`	.	.	2	.
`null-empty check`	.	.	2	.
Total	12	33	81	43

explained by the relatively small number of constraints of this type that were documented in the target systems.

In the case of the `Dual Value Comparison` type constraints, we find a second common implementation using the `null check` pattern. In this case, instead of checking a boolean variable, a nullable variable is checked for presence, with present/absent being the two possible values.

A different situation arises with the `Concrete Value` type constraints. The `constant argument` pattern is the most frequently used to implement this type of constraint. In our data, this pattern appears when the concrete value appears directly in the call to a setter or constructor. It is important to note that instances of this pattern suffer from the "magic number" code smell (Fowler 2018), which suggests that the use of this pattern is prone to introducing code smells. An alternative is the less common `assign constant` pattern, which does not introduce this code smell.

Additionally, we find some uncommon patterns that apply to some particular situations and could possibly be adapted into the more common ones. One example is the `binary flag check` pattern. In this pattern, an integer is used as a bit field, which effectively turns it into an array of boolean values (a set of binary flags). To check if one of the values is enabled, a *mask* consisting of an integer constant with only the corresponding bit turned to 1 and the bitwise *and* operation is applied with the value of the variable. This kind of pattern is commonly found in languages such as C (Oualline 1997). The pattern could be converted into the `binary comparison` pattern by turning each flag into its own boolean field. The same idea is applicable to the `null-zero check` and `null-emptycheck` patterns. These patterns exist because the `String` class has two possible empty values: empty string and `null`. Hence, they could be turned into `nullcheck` by ensuring that all empty strings are instead turned to `null` at creation.

Finally, we discuss the cases in which a constraint is implemented with a pattern that is the common implementation of a different constraint type. For example, the constraint "*If m is less than zero*" from Rhino (Rhino 2021), of `Value Comparison` type, is implemented with the `boolean property` pattern, when one would instead expect a `binary comparison` pattern. The enforcing statement is `if (sign[0])`, where `sign` is a boolean array of size 1. This construction exists because this array is passed to another method that sets it according to the value of `m`, which is reminiscent of passing a parameter by reference in the C language (Oualline 1997). We argue that this is quite an unusual construction in Java, as passing by reference is not supported for primitive types such as boolean. This unusual construction could be transformed into the more expected `binary comparison` by having the method return the data instead of modifying it in-place.

Some constraints of `Dual Value Comparison` type are implemented with the `binary comparison` pattern, and we attribute the implementation rationale to discrepancies between the language of the constraint and concrete implementation decisions. For instance, the Ant constraint "*unless either the -verbose or -debug option is used*" (Apache Ant 2021) contains the constraint "*the -verbose option is used*", which can be either true or false. However, this is implemented by iterating over the arguments and checking each against the text of each option successively, shown in Listing 3 (constraint implementation is line 6). Such a long chain of `if-else` statements (more than 20 in this case) is a code

Listing 3: Checking for verbose argument in Ant.

```
for (int i = 0; i < args.length; i++) {
  final String arg = args[i];
  [...]
  } else if (arg.equals("-quiet") || arg.equals("-q")) {
    msgOutputLevel = Project.MSG_WARN;
  } else if (arg.equals("-verbose") || arg.equals("-v")) {
    msgOutputLevel = Project.MSG_VERBOSE;
  } else if (arg.equals("-debug") || arg.equals("-d")) {
  [...]
```

Listing 3 Checking for verbose argument in Ant

smell (Fard and Mesbah 2013). Note that the contents of the `args` array could be cached into an object, which could later be queried on whether it contains the *verbose* option, both getting rid of the code smell and applying the more common `boolean property` pattern.

RQ2 answer

We identified and defined 30 constraint implementation patterns. The `binary comparison` and `boolean property` occur very frequently, while 13 other patterns are utilized frequently and the rest are rare, in our data. This indicates that developers tend to use a rather small number patterns when implementing data constraints. Additionally, our data suggest that certain patterns are commonly used to implement certain constraint types. There is also evidence that implementations of constraints that do not follow the most common patterns are due to unusual implementation decisions or exhibit code smells.

6 Multiple Enforcements of a Constraint (RQ3)

As shown in the motivating example from Section 2, constraint implementations may have multiple enforcing statements in different code locations. We refer to them as being enforced in multiple distinct locations in the code or as having multiple enforcements. Hence, such constraints have several trace links to the code (i.e., one set of data definition statements and multiple distinct sets of enforcing statements). Intuitively, one constraint should be enforced in one place in the code. We study how many constraints are enforced in multiple locations for answering RQ3: *What are the differences between multiple enforcing statements of the same constraint?*

The study for answering RQ2 relied on manually identifying only *one* enforcing statement per constraint (and the corresponding data definitions). For answering RQ3, we need to identify multiple enforcing statements for a given constraint, where they exist. Unfortunately, it is prohibitively expensive to manually identify all enforcing statements of a constraint in large projects. Hence, tool support for collecting additional traces is essential. We leverage the constraint implementation patterns discovered in Section 5 and use static analysis techniques to automatically find candidate enforcing statements, based on the data definitions that were manually identified.

6.1 Detectors for Tool-Assisted Tracing

We implemented 13 static analysis-based detectors to assist the identification of multiple enforcing statements for a given constraint. Each detector is designed to detect the instances of one frequent CIP (used in at least two instances in our data). There are 15 frequent patterns in Table 5. We did not build detectors for two frequent patterns, `properties file` (3 instances) and `polymorphic method` (2 instances). The instances of the `properties file` pattern do not appear in Java code, but in text files. Those of the `polymorphic method` pattern make use of dynamic dispatch, hence static analysis may be insufficient for accurate detection.

Each detector uses the data definitions of a constraint as input and returns candidate enforcing statements. The number of inputs that a detector accepts is the same as the number of "*parts*" of the pattern it implements, as defined in Table 4. Hence, a detector may have 1 to 3 inputs depending on the pattern it implements. For example, for detecting the `boolean property` pattern the detector takes a single operand as input. For detecting the `binary flag check` the detector takes two operands as input, while for detecting the `binary comparison` it uses two operands and one operator as input.

We use syntax analysis (for identifying the pattern) and dataflow analysis (for finding all instances of a pattern corresponding to a particular data definition) to automatically detect instances of our CIPs. Syntax analysis at the Abstract Syntax Tree (AST) level is suitable for analyzing the source code structures, while the dataflow analysis is able to trace data dependence in an intermediate representation (IR). Specifically, we implemented the detectors using a combination of JavaParser (2021)—a parser with AST analysis capabilities, and WALA (2021)—a static analysis framework for Java. For the AST analysis, we parse every Java file in the system's source code and record the lines where every instance of each pattern appears. The instances are identified by matching code structures with *statement type* and *part* defined in Table 4. For the WALA analysis, we first build a *call graph* and a *system dependence graph*, which is the program representation commonly used for program slicing (Tip 1994). For each of the CIPs, there exists data dependence between the data definitions and the enforcing statements. We perform forward program slicing on the system dependence graph to track such data dependence. In general, each detector performs slicing from the input data definitions and then matches any occurrences of the *statement type* and *part* defined in Table 4 on the IR along the slice (or intersection of the slices, in case there are two operands). It later confirms the match by checking that the source-code pattern exists in that location using the syntax analysis, as IR does not perfectly keep the code structures.

As an example, the constraint "*If [the buffer] has unsaved changes*" (jEdit 2021) is enforced by the statement `if(buffer.isDirty())` with the `boolean property` pattern. The data definition statement in this case is the definition of the `Buffer.dirty` field. Passing this input to the `boolean property` detector returns a list of lines in files EditPane.java, View.java, BufferAutosaveRequest.java, among others, where the value of the field is used and the pattern appears.

6.2 Tool-Assisted Tracing Protocol

Our goal is to retrieve trace links in addition to the ones identified manually in Section 5.1. If a constraint is implemented using multiple enforcing statements, we create a separate trace link to each enforcing statement (and the associated data definitions). Because it is possible

that multiple enforcing statements for the same constraint may follow different patterns, we use several detectors for each constraint.

Given a constraint, we execute all detectors that take the same number of inputs as the manually-traced pattern from Section 5.2. Recall that the number of data definitions depends on the size of "*parts*" in the CIPs. For example, if the manually-traced pattern of a constraint is `null check` pattern (which has a single part), we used its data definition to run all one-input detectors. Therefore, these detectors would potentially find candidate enforcing statements that follow all patterns with a single part (i.e., `boolean property` and `null-empty check`).

We used our detectors to retrieve candidate links for 163 constraints from all eight systems, i.e., those implemented using one of the 13 patterns. Two authors independently examined a subset of the candidate links for each constraint, classifying each link as true positive or false positive. The authors followed the same protocol that was used to verify the traces in Section 5.1.4. When the detectors returned more than 25 candidate links for a constraint, 25 of these were randomly sampled for classification. In total, the authors inspected 1,362 candidate enforcing statements out of the total 7,272 results. On average, the detectors retrieved 44 (median 4) candidate links per constraint.

6.3 Results and Analysis

We present a summary of the tool-assisted tracing results and analyze the cases where one constraint is enforced in multiple places, in Section 6.3.1. We also perform an assessment of the recall achieved by the tool-assisted protocol in Section 6.3.2.

6.3.1 Analysis of the Multiple Enforcing Statements

After classifying the 1,362 candidate enforcing statements, our tool-assisted tracing identified 256 new enforcing statements (i.e., true positives) for 71 constraints (44%) out of the 163 used in the tool-assisted study. We further studied the enforcing statements of these 71 constraints, which have more than one enforcing statement. Figure 1 shows the distribution of the CIPs implementing these constraints. We observed that in most cases (66 out of 71) the same pattern is used for all the enforcing statements of the same constraint (i.e., the corresponding bar in Fig. 1 has a single texture and color). We call them *consistent* implementations. In five cases, the constraint has more than one enforcing statement and they

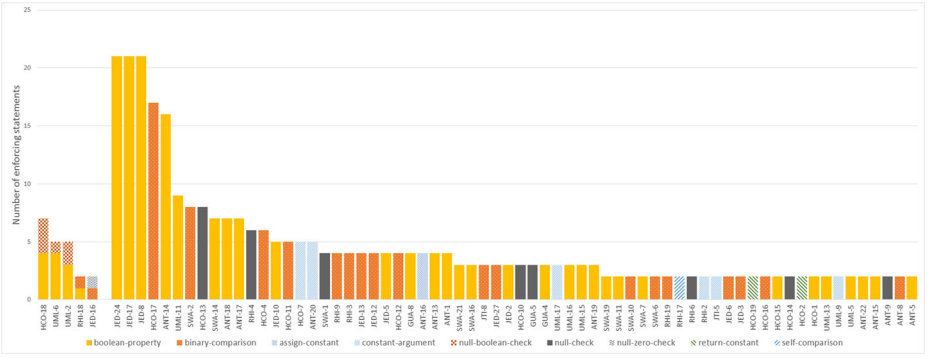

Fig. 1 Distribution of patterns for constraints with multiple enforcing statements. The five bars on the left correspond to the inconsistent constraints

follow multiple patterns (i.e., the corresponding bars in Fig. 1 show multiple textures and colors). We call them *inconsistent* implementations.

While these multiple enforcing statements are not inherently problematic, they are essentially instances of code cloning (Baker 1995). The cases of consistent implementations result in *type 1* or *type 2* clones, i.e., they are either exact copies, or the only changes occur in identifiers and literals (Bellon et al. 2007). However, inconsistent implementations lead to *type 4* clones, in which the code is syntactically different, but the semantics are the same (Roy et al. 2009).

We identified two types of inconsistent implementations:

1. *Related patterns.* The *"supplied entity is already repeatable"* (HCO-18 in the chart) constraint for the HTTPComponents system (HTTP Working Group 2021) is checked using the method invocation `RequestEntityProxy .isRepeatable(request)` in four different code locations. This repeated enforcing statement is an instance of the `boolean property` pattern. The constraint is additionally enforced in three different code locations using the enforcing statement: `entity != null && entity.isRepeatable()`. This enforcing statement corresponds to the `null-boolean check` pattern. These two patterns are similar, as they both check a boolean value, but the second one additionally accounts for a `null` value. This is the case for the constraints HCO-18, UML-2, UML-6, and JED-16.
2. *Unrelated patterns.* The Rhino constraint *"[Result of toNumber] is [...] $+\infty$"* (Rhino 2021) is implemented by an instance of `binary comparison` pattern (`d == Double.POSITIVE_INFINITY`) and an instance of `boolean property` pattern using a standard library utility method (`Double.isInfinite(d)`). While, these two patterns are used to implement the same constraint, they have different structures. This situation occurs in the implementation of constraint RHI-18.

It is easy to argue that inconsistent implementations are detrimental to code maintainability, as *type 4* clones are challenging to detect automatically (Komondoor and Horwitz 2001; Gabel et al. 2008). Additionally, they pose challenges when their rationale is not well documented (see example of related patterns above, the code does not specify why some cases require the `null` check while others do not).

Consistent implementations also pose potential problems. Although existing research suggests that developers often evolve duplicated code consistently (Thummalapenta et al. 2010), handling a large number of duplicates (over 20 in some cases in our data) can lead to a more demanding and error-prone change process when these constraints need to be modified. We argue that most of these enforcing statements can be refactored. For example, the constraint SWA-1 "*configuration file is not available*" (Swarm 2021) is implemented as `if (WinDataFile.configFile == null)`. Refactoring this enforcing statement so that the null checking happens in a method of the `WinDataFile` would encapsulate the logic and make the semantics of the constraint clearer, which would make eventual changes easier.

The presence of duplicated code corresponding to business rules can also indicate the presence of duplicated business processes, which are challenging to identify in textual artifacts (Guo and Zou 2008).

Even though the literature is divided on whether code clones are detrimental, evidenced by the extensive research on clone detection (Ain et al. 2019; Roy et al. 2009), or a necessary part of development (Kapser and Godfrey 2006), we argue that it is counter-intuitive for data constraint implementations to exhibit a large amount of clones.

RQ3 answer

Nearly half (44%) of the constraints we studied are implemented with more than one enforcing statement. These multiple enforcing statements result in code clones: 108 type 1, 138 type 2, and 10 type 4. We attribute these implementations to design decisions, rather than to the intrinsic properties of the constraints. We argue that most constraints with multiple enforcing statements would benefit from refactoring into simpler patterns, *i.e.,* `boolean property`.

6.3.2 Tool-Assisted Tracing Recall and Precision Assessment

While it is not possible to provide accurate precision and recall values for the tool-assisted protocol (due to its very nature), we provide estimates below and discuss their implications for our conclusions. These metrics are not provided as an evaluation of our tool, but instead as statistics for better understanding of our data.

We first verified that our detectors retrieved the manually-traced link for each constraint. Retrieving a large number manually-traced links would indicate that the detectors have an acceptable level of recall. We expect and accept that some manually-traced links are not detected, as the consequence of the trade-off between performance, soundness, and precision in static analysis (Livshits et al. 2015). The manually-traced links were retrieved for 153 out of 163 constraints (94%). The detectors retrieved candidate links for 159 out of 163 constraints, meaning that for four constraints, the detectors did not retrieve any candidate links.

Of the 1,362 candidate enforcing statements which we manually analyzed, our tool-assisted tracing identified 256 new enforcing statements (i.e., true positives). This means that, in total, 415 (30%) of the 1,362 manually examined detector results are true positives. This is an estimate of the precision, however, note that it is possible that the detectors retrieved additional links which we did not classify.

We performed a more thorough recall assessment, which consisted of exhaustively tracing all the enforcing statements for the 22 constraints of the ArgoUML system that were part of the tool-assisted study. We chose to perform this assessment only on the constraints of a single system to make the task feasible, as exhaustively tracing a constraint is a labor-intensive process, prohibitive for all the systems.

The tracing protocol was similar to the one used for the tool-assisted study. The tracing was performed by two authors, with the constraints being evenly split between them. The final traces were set after being verified by both tracers. The main difference is that the tracers performed the slicing manually as opposed to assisted by our tool. Starting at each data definition, the tracers obtained a list of uses with the help of an IDE. From each one of these uses, the tracers propagated the slice forward, following the data flow through method call arguments and assignments, but stopping when the value was modified.

This process located the ground truths for the 22 constraints, in addition to the 25 true positives that were found using the tool-assisted protocol. The exhaustive manual tracing unveiled seven additional true positives for five constraints, three of which did not have any true positives (other than the manually-located ground truth) found using the tool-assisted protocol. We analyzed these 7 new true positives and found that:

– Five enforcing statements, from four constraints, were not found in the result list of our tool. A manual examination of these enforcing statements indicated that they do not exhibit any inherent properties that would make their localization impossible using our tool (e.g., they are each implemented with a pattern that exists in our catalog). Instead, they could not be retrieved due to limitations in our particular implementation, which are required to make the analysis feasible for these systems. For example, in one case, the corresponding data definition statement is used in a method which is an override of a library method (`getPopUpActions`). It was necessary to exclude libraries from the analysis to make the runtime feasible, and our analysis framework cannot include the overridden method in the call graph without having access to the class that originally defined it. This means that the slice cannot propagate to the enforcing statement in the method `FigPackage.buildShowPopUp`, since this requires slicing from the usage in `getPopUpActions`.
– The two remaining enforcing statements, both from the same constraint, are implemented using a pattern not encountered while answering RQ2. We included this new pattern (`enum instanceOf`) in the full catalog, listed in the Appendix.

For ArgoUML, our tool-assisted protocol successfully retrieved 25 out of the 32 true-positive enforcing statements, i.e., a 78% recall. While we cannot generalize this recall performance to the entire dataset, we do not believe that the recall values for the other systems would be substantially different, as ArgoUML has no special properties that would indicate that enforcing statements are easier to retrieve than in the other studied systems. Three out of the 22 constraints (14%) were deemed as having only one implementation, when in fact they had multiple, meaning that our estimate for the proportion of constraints with multiple implementations (44%) is indeed a lower bound.

7 Catalog Validation (RQ4)

In order to validate our constraint implementation pattern catalog, we conducted a study with professional developers for answering the following research question: **RQ4:** *How accurately can developers identify constraint implementation patterns?* If the developers can accurately identify the patterns used in the implementation of data constraints, then we can infer that the patterns are well defined, in as much as the developers do not disagree with their definitions and they can recognize them in the code. Conversely, if the developers cannot identify the patterns accurately, then it means that the patterns are not well defined.

7.1 Subjects

We used convenience sampling to recruit the developers for participating in the study. Specifically, we asked developers that we know directly and also asked collaborators to reach out to developers they know. We aimed at having developers with various degree of Java development experience. We did not account for other attributes (e.g., age, gender, and current employment), since we consider them orthogonal to the task. Sixteen developers answered to our request and participated in the study. The colors and texture in Fig. 2 indicate the professional experience as Java developers of the 16 subjects: two have less than

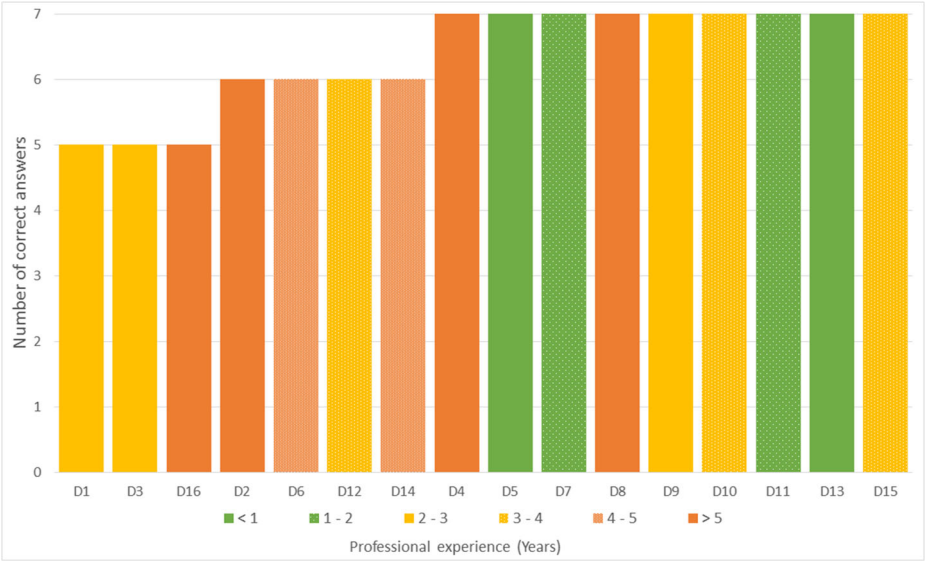

Fig. 2 Results of developer study. The column height shows the number of correct answers given by the developers (out of 7). The color and texture indicate the experience of the developers

one year experience, two have between 1–2 years experience, three between 2–3 years, three between 3–4 years, two between 4–5 years, and four have more than 5 years experience.

7.2 Objects

We sampled 22 constraints for which the developers had to identify the patterns used in their implementation: 2 randomly selected constraints from each of the 10 CIPs that are frequent and also appear in more than one system (20 constraints) plus an additional 2 (randomly selected from all remaining constraints) to use as control questions.

7.3 Questionnaires

The study was conducted as a questionnaire in which the developers are asked to identify the correct CIP for an enforcing statement from our data. The questionnaire starts by briefly introducing the developer to the concept of a data constraints and a CIP, and then presents 7 questions, each corresponding to a constraint. Each question consists of the constraint text as found in its corresponding system's textual artifacts, its simplified form (the same one used for the tracing), and a snippet of code where the constraint is implemented, with the enforcing statement highlighted. Under the code snippet, two links to see the whole method and the whole class are included, in case the developer needs more context to understand the code. For each constraint, the developer was also presented with three CIP definitions: the one that corresponds to the enforcing statement, and two others selected at random, in random order. Finally, four options are presented: the names of the three CIPs whose definitions are presented (randomly shuffled), and "None of the above". The developer is asked to select the CIP that is used in the constraint implementation, or "None of the above", if appropriate. A sample question is presented in Fig. 3.

(Constraint 1/7) Consider the bold text in the following paragraph:

> *The current filename. If* ***no filename for the project is set*** *yet, then the titlebar shows "Unititled"*

Which contains the constraint:

> *filename is not set*

And is implemented in the highlighted portion of this code:

```
1    if (projectFileName == null || "".equals(projectFileName)) {
2        if (ProjectManager.getManager().getCurrentProject() != null) {
3            projectFileName = ProjectManager.getManager()
4                .getCurrentProject().getName();
5        }
6    }
```

(If you need to, you can see the full method, or see the full class)

Given the following pattern definitions:

null-empty-check: A string value is checked for nullity using the == or != operators and immediately after compared to the empty string using the equals method. The two operations are combined using the && or || operators. The operands in each equality may be in any order.

```
value != null && !value.equals("")

null != obj.getValue() && !"".equals(obj.getValue())
```

equals-or-chain: Equality expressions (using the == operator) or equals method calls are chained by “or” operators in an expression checking possible values of a variable.

```
value == 1 || value == 2 || value == 3

value.equals("val1") || value.equals("val2") || value.equals("val3")
```

binary-flag-check: An integer value is operated with a bitwise AND operator (&) against an integer variable, and then the result is compared with == or != against another integer value (literal or variable).

```
value & FLAG == 0

obj.value & FLAG == res
```

Which pattern does the highlighted portion of the code above exhibit (if any)?

- null-empty-check
- equals-or-chain
- binary-flag-check
- None of the above

Fig. 3 Sample developer study question

We designed 4 questionnaires, which can be found in our replication package. This was done because we judged that a questionnaire with more than 20 questions and 10 CIPs that are new to the developers would be too demanding. Each questionnaire aims to test 5 of the 10 selected CIPs by using one constraint of each of these types as a question. We call these 5 "validation questions". The same two control questions were added to each questionnaire, for a total of 7 questions per survey. These are constructed the same way as the rest of the questions, except that the correct CIP is not in the options, and the correct answer is "None of the above". Each validation question is answered by 4 developers, while each control question is answered by all 16 developers.

7.4 Results and Analysis

We measure the *accuracy* of the answers, which is the percentage of correct answers of the total number of answers. We define accuracy per developer and also per constraint. For example, if developer D3 answered correctly 5 out of 7 questions, then D3's accuracy is 71.4%. Likewise, if for constraint ANT-22, four developers answered correctly and one did not, the constraint's accuracy is 75.0%. Note that when we average the accuracy per developer and the accuracy per constraint, the numbers are slightly different, as four developers answered each validation question and all 16 answered each control question.

The results show that the developers could identify the correct CIP for each enforcing statement with high accuracy (Fig. 2). Of the seven questions presented to each developer, three developers answered 5 questions correctly, four developers answered 6 questions correctly, while nine developers answered all 7 questions correctly. The average accuracy was 91.1% and the median was 100% (minimum 71.4%).

An overview of the answers to the 20 validation questions is presented in Table 7. The average accuracy per constraint was 92.9% and the median was 100% (minimum 75.0%).

We analyzed the constraint with the most incorrect answers. Two of the four developers who answered the question of constraint RHI-3 did not select the correct answer. This constraint states "*Because a single-line comment can contain any character except a LineTerminator character [...]*", which contains the constraint *character != LineTerminator*. The code snippet that was presented to the developers is in Fig. 4. The correct answer was `binary comparison`, while one developer chose "None of the above", and another chose `propertiesfile`.

Table 8 shows the distribution of answers for the control questions. Four out of 16 developers selected the wrong answer for the question corresponding to constraint JED-5. The constraint text states "*Files that you do not have write access to are opened in read-only mode, where editing is not permitted*" which we simplify as "*file is not accessible*". The snippet presented in the survey is in Fig. 5. All four developers selected `binary flag check` instead of the correct "None of the above". We attribute this to the presence of an instance of that pattern in the snippet, although outside of the highlighted enforcing statement. This suggests that these developers were able to identify the pattern even though they were not asked about it. We still count these answers as incorrect.

We argue that these results evidence two properties of our catalog of patterns: (1) the CIPs are well defined, i.e., they refer to code constructs that have a useful meaning; and (2) the catalog is easy to understand, as a short introduction and a brief description of each pattern were enough for the developers to classify a set of enforcing statements with high accuracy.

Table 7 Distribution of answers for validation questions. A one indicates a correct answer; a zero indicates a wrong answer

Q. ID	D1	D2	D3	D4	Acc.
UML-1	1	1	1	1	100.0%
ANT-25	1	1	1	1	100.0%
RHI-1	1	1	1	1	100.0%
JED-26	1	1	1	1	100.0%
ANT-22	1	1	0	1	75.0%
RHI-16	1	1	1	1	100.0%
ANT-24	1	1	1	1	100.0%
ANT-27	1	1	1	1	100.0%
RHI-5	1	1	1	1	100.0%
ANT-9	1	1	1	1	100.0%
JED-30	1	1	1	1	100.0%
ANT-20	1	1	1	1	100.0%
ANT-8	1	1	1	1	100.0%
ANT-21	1	1	1	1	100.0%
SWA-5	1	1	1	0	75.0%
ANT-23	1	1	1	1	100.0%
UML-26	1	1	1	0	75.0%
SWA-3	1	1	1	1	100.0%
RHI-3	1	0	1	0	50.0%
SWA-17	1	1	1	1	100.0%
JED-5	.	.	.	.	75.0%
UML-28	.	.	.	.	93.8%
				Avg.	*92.9%*
				Med.	*100.0%*

RQ4 answer

The developers were able to identify the use of constraint implementation patterns with high accuracy, *i.e.*, 91.1% average accuracy (with 100% median and 71.4% minimum) .

```
1    private void skipLine() throws IOException
2    {
3        // skip to end of line
4        int c;
5        while ((c = getChar()) != EOF_CHAR && c != '\n') { }
6        ungetChar(c);
7    }
```

(If you need to, you can see the full class)

Fig. 4 Implementation of RHI-3

Table 8 Distribution of answers to the control questions. A one indicates a correct answer; a zero indicates a wrong answer

Dev	JED-5	UML-28
1	0	0
2	0	1
3	0	1
4	1	1
5	1	1
6	0	1
7	1	1
8	1	1
9	1	1
10	1	1
11	1	1
12	1	1
13	1	1
14	1	1
15	1	1
16	1	1
Average	*75.0%*	*93.8%*

8 Threats to Validity and Limitations

A major contribution of this paper is the discovery and definition of the constraint types and constraint implementation patterns. These definitions are data-driven, created and agreed upon by the authors of this paper. It is possible that different coders would produce a different set of definitions. To mitigate this threat, we defined a clear coding framework based on qualitative data analysis methods (Miles et al. 2014), presented in Sections 4.3 and 5.2. This process revealed 16 (9%) cases of disagreement, which can be considered as a small proportion. Furthermore, our confidence in the significance of the produced catalog of patterns is increased by the results of the developer study.

Due to our experimental design, we are unable to study constraints that are not documented in the textual artifacts of the target systems. We consider this a reasonable tradeoff, as attempting a more complete study would require access to developers intimately familiarized with the systems, as well as a large time investment. We do not claim to study every kind of data constraint, but rather those that are documented in the target systems.

```
1    VFS vfs = VFSManager.getVFSForPath(getPath());
2    if (((vfs.getCapabilities() & VFS.WRITE_CAP) == 0) ||
3        !vfs.isMarkersFileSupported())
4    {
5     VFSManager.error(view, path, "vfs.not-supported.save",
6      new String[] { "markers file" });
7     return false;
8    }
```

(If you need to, you can see the full method, or see the full class)

Fig. 5 Implementation of JED-5

Our answer to RQ2 also depends on the accuracy of traces we produce in Section 5.1. It is possible that some constraints could have been traced to the wrong statements in the code. To make our tracing as reliable as possible, we employed two tracers for each constraint and had two authors decide the final trace through a discussion. This protocol is in line with previous work on traceability (Eaddy et al. 2008b; Ali et al. 2011, 2012, 2013).

Whether our CIP catalog reflects the space of constraint implementations in all Java systems depends on our choice of target systems. Since the systems in our data set are real-world open-source systems from a variety of domains, we expect the constraint implementations we identified to also exist in other similar systems. Further research is needed to establish whether the CIPs and their distributions would be different in other type of software (e.g., different domain and proprietary). As reported in Section 6.3.2, we observed that data constraints can be implemented with patterns outside the ones in our catalog even in the studied systems. Nonetheless, we posit that the distribution we observed should be largely the same if expanded to a broader set of systems. In other words, our data suggests that developers gravitate toward a set of very common patterns.

We make our data and our pattern catalog openly available, such that future research can enrich the catalog with new distribution information or with new patterns. Note that the current catalog applies only to the Java programming language. Future work is required to evaluate the prevalence of these patterns in other programming languages.

Our tool-assisted tracing protocol in Section 6 relied on the inputs and patterns that were derived from the manually defined traces, and it may not find all enforcing statements of each constraint. Our recall and precision assessment in Section 6.3.2 indicates that our estimation of the number of constraints with multiple implementations is reasonably accurate and we present this figure as a lower bound of the real number. Indeed, a larger proportion of constraints than reported here having multiple implementations would indicate that these implementations are exemplars of an even larger phenomenon of interest for future research, which does not invalidate our conclusions, but further motivates the importance of RQ3.

Our answers to the RQs are also dependent on our choice to focus only on the data definition and enforcing statements. As we discussed in Section 2, one can argue that there are other statements relevant to the implementation of a constraint, which should also be traced and analyzed. For example, definitions of variables used by the enforcing statements. We consider that the data definition and enforcing statements pair we trace to is a *minimal subset* of a constraint implementation, that is, eliminating any of them would no longer produce non-ambiguous traces. We argue that including additional constructs in the traces will not alter or invalidate the current catalog of patterns. Instead, it will likely result in the refinement of the existing patterns, based on the properties of these additional statements. Expanding the study of constraint implementations to include additional code constructs/statements is subject of future work.

9 Related Work

The concept of *constraint* has been used in multiple contexts and with different definitions in the software engineering literature. We found two instances to be particularly related to our work. In the context of business rules, Wiegers and Beatty (2013) define a constraint as "a statement that restricts the actions that the system or its users are allowed to perform". This definition covers data constraints, because it can be said that the system *is only allowed* to accept/produce valid data. Breaux and Antón (2008) define a constraint as a statement

that narrows the possible interpretations of a concept based on its properties. For instance, "*patient who receives healthcare services*" restricts the set of all patients to only those who receive healthcare services. This definition is similar to ours in that data constraints conceptually narrow the set of possible entities, e.g., *"request to HEAD method"* is a subset of all requests. It is important to note that neither of these works explore how the constraints are enforced in source code.

Previous studies have leveraged constraints in textual artifacts to extract usage patterns. Xiao et al. (2012) use sentence patterns such as "*[noun] is allowed to [action] [resource]*" to automatically extract security policies. The extracted sentences correspond to constraints, though they constraint access control permissions, not data. The work of Pandita et al. (2012) can infer data constraints (e.g., "*path must not be null*"') from the documentation of a method (e.g., the sentence "*If path is null*"). Similarly, other works classify method parameters according to whether a null value is allowed (Tan et al. 2012), has to belong to a specific type (Zhou et al. 2017), or its numeric value has to be in a certain range (Saied et al. 2015). Note that even though these techniques can infer the existence of a data constraint enforcement in the method, they do not study their implementations and depend on the accuracy of the documentation.

More similar to the study presented in this paper is the work of Yang et al. (2020). It examined three types of data constraints specific to web applications implemented in Ruby on Rails: front-end constraints expressed as regular expressions; application constraints on data fields in model classes, specified in validation functions (which check the length of a text-field, content uniqueness, and content presence); and database constraints specified in the applications' migration files, through Rails Migration APIs. The research found that these type of constraints are often checked inconsistently between the three architectural layers and developed a tool to identify such inconsistencies. In contrast, we identified the constraints in textual documents and then manually traced them to their enforcing statements in Java system, regardless of their architecture or how they are implemented. Our goal is identifying and analyzing all data constraint types and their implementations found in this systems, as opposed to restricting the set to a known type of implementation.

Research on automated business rule extraction proposed methods similar to those we used in the design of our detectors. In particular, backward or forward slicing is done from a previously identified variable to detect the conditional statements that affect its value, hence related to a business rule (Hatano et al. 2016; Cosentino et al. 2012, 2013; Huang et al. 1996; Sneed and Erdös 1996; Wang et al. 2004; Sneed 2001; Chaparro et al. 2012). However, their goals were not to analyze implementation patterns of data constraints.

As part of our study, we performed manual and tool-assisted requirements-to-code traceability link recovery (RCTLR) (Antoniol et al. 2002; Borg et al. 2014), as we consider the data constraints as part of the system requirements. We developed our own detectors for recovering candidate links, because existing approaches are not appropriate for this use. This is because the current state of the art does not achieve retrieval of such links at statement level (Cleland-Huang et al. 2014a). Most existing techniques are based on text retrieval (De Lucia et al. 2012; Borg et al. 2014), while approaches based on machine learning (Guo et al. 2017; Mirakhorli and Cleland-Huang 2016) and AI (Sultanov et al. 2011; Blasco et al. 2020) have also been explored. Closer to our work are the approaches leveraging structural features of the software (Eaddy et al. 2008a; McMillan et al. 2009; Kuang et al. 2017). However, these features often focus on class or method relationships, and do not describe implementation patterns. Recently, Blasco et al. (2020) proposed a statement-level RCTLR approach that uses LSI and genetic algorithms. It works by selecting a set of seed statements based on textual similarity, which is randomly mutated using the crossover and mutation operators

until it results in a set of candidate links. In contrast, our study found the statement-level candidate links by exploiting the implementation patterns that we identified.

The design and use of our detectors is related to research on the automated detection of design patterns. Three characteristics have mainly been used to identify patterns: structural (Guéhéneuc and Antoniol 2008; Tsantalis et al. 2006; Kaczor et al. 2006; Guéhéneuc et al. 2004), behavioral (Shi and Olsson 2006; Park et al. 2004), and semantic (Dong and Zhao 2007). Our work is similar to those that employ structural characteristics, as we use static analysis to pinpoint the location where a certain constraint is enforced. However, our patterns span only statements, while design patterns span multiple classes, focusing on more generic computational solutions for recurring problems.

10 Conclusions and Implications

While the importance of business rules is widely recognized in software engineering and the field of automated business rule extraction provides a wealth of techniques, there still is a lack of understanding of how business rules are implemented in source code. This is not surprising, given the vast diversity in possible rules and implementation decisions. This study is a first step towards better understanding how developers implement business rules.

We focused on understanding data constraints and their implementations through an empirical study. Studying 187 constraints from eight Java systems, we learned that:

- The documentation of studied systems describe four types of data constraints.
- The implementations of the 183 studied data constraints (those that are not enforced externally) can be categorized into 31 constraint implementation patterns (CIPs). 15 of these patterns implement 168 of the constraints, with the two most common (`boolean property` and `binary comparison`) accounting for half of all the implementations. This suggests that developers employ a small number of CIPs to implement most constraints.
- Certain patterns are preferred when implementing constraints of certain types and deviations from these trends are associated with unusual implementation decisions and code smells.
- 44% of the studied constraints are implemented with more than one enforcing statement in multiple code locations. While 93% of them use the same pattern for all of the enforcing statements, they are the result of code cloning (i.e., type 1, type 2, and type 4 clones).

10.1 Implications

We expect that our findings will impact several aspects of software engineering research and practice. Our atomic patterns enable fine-grained reasoning about the source code, which can result in novel approaches for improving various software engineering tasks. Additionally, our catalogs and protocols can serve as templates for future studies related to software requirements and business rules.

10.1.1 Traceability Link Recovery

The ability to describe the implementation of data constraints will help in defining new approaches for automated traceability link recovery. RQ1 and RQ2 suggest that, given the type of a data constraint, we can estimate the probability of which CIP is used for its implementation (Table 6). Our detectors indicate that CIPs can be identified using static analysis

(Section 6.1). Based on these findings, heuristics could be defined to trace data constraints in software at line-of-code granularity. After identifying all CIP instances in the source code, they could be ranked both by the likelihood that the pattern implements the constraint, as well as the textual similarity between the constraint and the pattern instance.

Having access to constraint-level traces can also improve the performance of traceability link recovery techniques with coarser granularity. A technique leveraging this knowledge would first identify the constraints in a requirement, and then assign a larger score to the code elements implementing these constraints, as these finer-grained elements are likely to be related to the implementation of the larger requirement.

10.1.2 Testing

Data constraints are business rules and should, consequently, be thoroughly tested. Knowing the line-of-code implementation of a constraint would facilitate determining whether it is being properly tested, as a coverage report with line-of-code granularity could be used. Tool support may then be developed to ensure that the lines of code that enforce the constraint are covered by each of the the tests.

We posit that constraints implemented with the CIPs that we describe will also be tested in predictable ways. This paves the way for the development of techniques that automatically generate test cases for data constraints.

10.1.3 Code Review

The CIP catalog also has implications on code reviews. The presence of CIPs could automatically be determined in bug-inducing commits, which would result in an assessment of which CIPs are most likely to introduce bugs. Tool support could then be implemented to highlight these patterns during code review, along with an explanation of why they are likely to introduce bugs.

10.1.4 Guidelines for Constraint Implementation

Current best practice software development guidelines do not address data constraints. Our CIP catalog could be used to define guidelines that address the implementation of data constraints, which would avoid some of the code smells and unusual decisions associated with the use of unexpected patterns.

10.1.5 Studying Business Rule Implementations

Our catalogs of constraint types and constraint implementation patterns, as well as the protocols employed to derive them, are just a first step in studying the implementation of business rules. The catalogs are meant to be extended and refined via future research and we anticipate our protocols will be used as templates for future studies on other kinds of constraints and business rules.

Appendix: Constraint Implementation Patterns Catalog

Tables 9, 10, and 11 contain 23 descriptions of the constraint implementation patterns from our catalog. The 7 most common patterns can be found on Table 4.

Table 9 CIP catalog, part 2

CIP name: `equals or chain`.

Description: Equality expressions (or `equals` method calls) are chained by "or" operators in an expression checking possible values of a variable.

Statement type: Boolean expression.

Parts: {*variable*}

Example:

Instance: `option.equals("true") || option.equals("on") || option.equals("yes")`

Parts: `option`.

CIP name: `properties file`.

Description: The value for a variable is stored in a file.

Statement type: File line.

Parts: {*constant*}

Example:

Instance: `backups=1`

Parts: `backups`.

CIP name: `polymorphic method`.

Description: Conditional branching is achieved by calling a method in a superclass that is overridden in a subclass.

Statement type: Method call.

Parts: {*method*}

Example:

Instance: `scriptable.getDefaultValue()` (`getDefaultValue` is an abstract method)

Parts: `Scriptable.getDefaultValue()`.

CIP name: `null-empty check`.

Description: A string value is checked for nullity using the `==` or `!=` operators and then compared to empty string using the equals method. The first expression is a `null check` pattern, but for `null-empty check` to apply, both expressions must be present.

Statement type: Boolean expression.

Parts: {*variable*}

Example:

Instance: `string == null || string.equals("")`

Parts: `string`.

CIP name: `null-zero check`.

Description: A value is checked for nullity using the `==` or `!=` operators and then its length or other numeric property is compared to zero. The first expression is a `null check` pattern, but for `null-zero check` to apply, both expressions must be present.

Statement type: Boolean expression.

Parts: {*variable*}

Example:

Instance: `string != null && string.length() > 0`

Table 9 (continued)

Parts: `string`.

CIP name: `return constant`.

Description: Return a literal value.

Statement type: Return statement.

Parts: {*constant*}

Example:

Instance: `return 80`

Parts: `80`.

CIP name: `switch-len char`.

Description: A switch is done first on the length of a string and then on specific characters to determine which of the options corresponds to the input string.

Statement type: Switch statement.

Parts: {*variable*}

Example:

Instance: `switch(token.length()) {case 1: c=s.charAt(1); if (c=='f') { ... } ...`

Parts: `token`.

CIP name: `self comparison`.

Description: A variable is compared to itself.

Statement type: Relational expression.

Parts: {*variable*}

Example:

Instance: `d != d`

Parts: `d`.

Table 10 CIP catalog, part 3

CIP name: `str starts`.

Description: The `startsWith` method is called on a string variable.

Statement type: Method call.

Parts: {*variable*}

Example:

Instance: `arg.startsWith("-background")`

Parts: `arg`.

CIP name: `null-boolean check`.

Description: A variable is checked for nullity using the == or != operators and then a boolean property of the variable is checked. The first expression is a `null check` pattern, and the second is a `boolean property`, but for `null-boolean check` to apply, both expressions must be present.

Statement type: Boolean expression.

Parts: {*variable*}

Example:

Instance: `saveAction != null && saveAction.isEnabled()`

Parts: `saveAction`.

CIP name: `setter`.

Description: A setter method is used to assign a value to a field.

Statement type: Method call.

Parts: {*method*, *variable*}

Example:

Instance: `project.setBasedir(helperImpl.buildFileParent.getAbsolutePath())`

Parts: {`project.setBasedir`, `helperImpl.buildFileParent.getAbsolutePath()`}.

CIP name: `constructor assign`.

Description: A field is initialized in a constructor or builder method, but not using any of the parameters.

Statement type: Assignment.

Parts: {*field*}

Example:

Instance: `authorname = Configuration.getString(Argo.KEY_USER_FULLNAME)`

Parts: `authorname`.

CIP name: `delta check`.

Description: Two variables are subtracted and their difference is compared to zero.

Statement type: Arithmetic expression, Boolean expression.

Parts: {*variable*, *variable*}

Example:

Instance: `int delta = getMajor() - that.getMajor(); if (delta == 0);`

Parts: {`getMajor`, `getMajor`}.

Table 10 (continued)

CIP name: `enum valueOf`.

Description: The method valueOf of an enum is used to ensure that a string variable represents a valid member of the enum.

Statement type: Method call.

Parts: {*variable*}

Example:

Instance: `BufferSet.Scope.valueOf(jEdit.getProperty("bufferset.scope", "global"))`

Parts: {`jEdit.getProperty("bufferset.scope", "global")`}.

CIP name: `iterate-and-check literal`.

Description: The value of the variable is checked by iterating over a collection of possible values and checking equality for each one. The value of this collection comes from a literal.

Statement type: Loop statement.

Parts: {*variable*, *collection*}

Example:

Instance: `for (ExtensionType value : values) {if (name.equals(value.name())) ...`

Parts: {`name`, `values`}.

CIP name: `mod op`.

Description: Restricts the values that a variable can take to the possible remainders of a division.

Statement type: Arithmetic expression.

Parts: {*variable*}

Example:

Instance: `daysSince19700101 % 7`

Parts: {`daysSince19700101`}.

Table 11 CIP catalog, part 4

CIP name: `str ends`.
Description: The endsWith method is called on a string variable.
Statement type: Method call.
Parts: {*variable*}
Example:
Instance: `name.toLowerCase().endsWith("." + defaultFilter.getSuffix())`
Parts: `name`.

CIP name: `switch case`.
Description: One of the cases of the switch checks the value (switch variable is of type enum).
Statement type: Switch statement.
Parts: {*variable*}
Example:
Instance: `switch(state) {case Buffer.FILE_CHANGED: . . .`
Parts: {`state`}].

CIP name: `override value set`.
Description: Each allowable value for a set is defined as the return value of the override of an abstract method.
Statement type: Method definition.
Parts: {*method*}
Example:
Instance: `public abstract String getExtension();`
Parts: {`getExtension`}.

CIP name: `cast self-comparison`.
Description: A numeric variable is cast to another type and then compared to the original variable.
Statement type: Assignment, Boolean expression.
Parts: {*variable*}
Example:
Instance: `int id = (int)d; if (id == d)`
Parts: {`d`}.

CIP name: `index loop find`.
Description: Iterate over collection of possible values. If the variable matches at some point, return the index. Otherwise return -1 at the end.
Statement type: Loop statement, Return statement.
Parts: {*collection*, *variable*}
Example:
Instance: `for (int i = 0; i < values.length; i++) {if (value.equals(values[i])) {return i;}} return -1;`
Parts: {`values, value`}.

CIP name: `assign class call`.
Description: Assigns a value derived from a method call on the result of a `.class` construct.

Table 11 (continued)

Statement type: Assignment.

Parts: {*variable*}

Example:

Instance: `classname = DefaultExecutor.class.getName();`

Parts: {`classname`}.

CIP name: `if-return chain`.

Description: A chain of ifs is used like a switch on a field, checking against the possible values of the variable. There are no else blocks and the body of each `if` is a return statement.

Statement type: If statement.

Parts: {*variable*}

Example:

Instance: `if ("jikes".equalsIgnoreCase(compilerType)) {return new Jikes();} if ("extjavac".equalsIgnoreCase(compilerType)) {return new JavacExternal();} . . .`

Parts: {`compilerType`}.

CIP name: `enum instanceOf`.

Description: The `instanceOf` operator is used to check whether a variable is an instance of an Enum type.

Statement type: Boolean expression.

Parts: {*variable*, *class*}

Example:

Instance: `visibility instanceof VisibilityKind`

Parts: {`visibility, VisibilityKind`}.

Funding This research was supported in part by grants from the National Science Foundation: CCF-1848608, CCF-1910976, CCF-1955837.

Availability of Data and Material We make available the data set used in our empirical study, as well as the data derived from it. Our replication package includes: software documents corresponding to eight software systems, constraints extracted from the documents, constraint-to-code traces, training and coding protocol material, and our catalog of constraint implementation patterns (Florez et al. 2022).

Code Availability The source code of our enforcing statement identification tool is also included in our replication package.

Declarations

Conflict of Interests There are no conflicts of interest or competing interests to disclose.

References

Ain QU, Butt WH, Anwar MW, Azam F, Maqbool B (2019) A systematic review on code clone detection. IEEE Access 7:86121–86144. https://doi.org/10.1109/ACCESS.2019.2918202

Ali N, Guéhéneuc YG, Antoniol G (2011) Trust-based requirements traceability. In: Proceedings of the 19th IEEE international conference on program comprehension (ICPC), pp 111–120. https://doi.org/10.1109/ICPC.2011.42

Ali N, Sharafi Z, Guéhéneuc YG, Antoniol G (2012) An empirical study on requirements traceability using eye-tracking. In: Proceedings of the 28th international conference on software maintenance (ICSM), pp 191–200. https://doi.org/10.1109/ICSM.2012.6405271

Ali N, Guéhéneuc YG, Antoniol G (2013) Trustrace: mining software repositories to improve the accuracy of requirement traceability links. IEEE Trans Softw Eng 39(5):725–741. https://doi.org/10.1109/TSE.2012.71

Alspaugh TA, Scacchi W (2013) Ongoing software development without classical requirements. In: Proceedings of the 21st IEEE international requirements engineering conference (RE), pp 165–174. https://doi.org/10.1109/RE.2013.6636716

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28(10):970–983. https://doi.org/10.1109/TSE.2002.1041053

Apache Ant (2021) Targets. https://archive.apache.org/dist/ant/manual/apache-ant-1.10.6-manual.zip

Baker BS (1995) On finding duplication and near-duplication in large software systems. In: Proceedings of 2nd working conference on reverse engineering, pp 86–95. https://doi.org/10.1109/WCRE.1995.514697

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33(9):577–591. https://doi.org/10.1109/TSE.2007.70725

Blasco D, Cetina C, Pastor Ó (2020) A fine-grained requirement traceability evolutionary algorithm: Kromaia, a commercial video game case study. Inf Softw Technol 119:106235. https://doi.org/10.1016/j.infsof.2019.106235

Borg M, Runeson P, Ardö A (2014) Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. Empir Softw Eng 19(6):1565–1616. https://doi.org/10.1007/s10664-013-9255-y

Breaux T, Antón A (2008) Analyzing regulatory rules for privacy and security requirements. IEEE Trans Softw Eng 34(1):5–20. https://doi.org/10.1109/TSE.2007.70746

Business Rules Group (2003) The Business Rules Manifesto. https://www.businessrulesgroup.org/brmanifesto.htm

Cemus K, Cerny T, Donahoo MJ (2015) Evaluation of approaches to business rules maintenance in enterprise information systems. In: Proceedings of the 2015 conference on research in adaptive and convergent systems, RACS. Association for Computing Machinery, New York, pp 324–329. https://doi.org/10.1145/2811411.2811476

Cerny T, Donahoo MJ (2011) How to reduce costs of business logic maintenance. In: 2011 IEEE International conference on computer science and automation engineering, vol 1, pp 77–82. https://doi.org/10.1109/CSAE.2011.5953174

Chaparro O, Aponte J, Ortega F, Marcus A (2012) Towards the automatic extraction of structural business rules from legacy databases. In: 2012 19th Working conference on reverse engineering, pp 479–488. https://doi.org/10.1109/WCRE.2012.57

Cleland-Huang J, Gotel OCZ, Huffman Hayes J, Mäder P, Zisman A (2014a) Software traceability: trends and future directions. In: Proceedings of the on future of software engineering (FOSE 2014), FOSE 2014. ACM, New York, pp 55–69. https://doi.org/10.1145/2593882.2593891

Cleland-Huang J, Rahimi M, Mäder P (2014b) Achieving lightweight trustworthy traceability. In: Proceedings of the 22nd ACM SIGSOFT International symposium on foundations of software engineering, FSE 2014. Association for Computing Machinery, New York, pp 849–852. https://doi.org/10.1145/2635868.2666612

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Meas 20(1):37–46

Cosentino V, Cabot J, Albert P, Bauquel P, Perronnet J (2012) A model driven reverse engineering framework for extracting business rules out of a java application. In: Bikakis A, Giurca A (eds) Rules on the web: research and applications. Lecture Notes in Computer Science. Springer, Berlin, pp 17–31

Cosentino V, Cabot J, Albert P, Bauquel P, Perronnet J (2013) Extracting business rules from COBOL: a model-based framework. In: Proceedings of the 20th working conference on reverse engineering (WCRE), pp 409–416. https://doi.org/10.1109/WCRE.2013.6671316

De Lucia A, Marcus A, Oliveto R, Poshyvanyk D (2012) Information retrieval methods for automated traceability recovery. In: Cleland-Huang J, Gotel O, Zisman A (eds) Software and systems traceability. Springer, London, pp 71–98. https://doi.org/10.1007/978-1-4471-2239-5_4

Dömges R, Pohl K (1998) Adapting traceability environments to project-specific needs. Commun ACM 41(12):54–62. https://doi.org/10.1145/290133.290149

Dong J, Zhao Y (2007) Experiments on design pattern discovery. In: Proceedings of the 3rd international workshop on predictor models in software engineering (PROMISE). IEEE Computer Society, Washington, DC, pp 12–12. https://doi.org/10.1109/PROMISE.2007.6

Eaddy M, Aho AV, Antoniol G, Guéhéneuc yg (2008a) cerberus: tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: Proceedings of the 16th IEEE international conference on program comprehension, pp 53–62. https://doi.org/10.1109/ICPC.2008.39

Eaddy M, Zimmermann T, Sherwood KD, Garg V, Murphy GC, Nagappan N, Aho AV (2008b) Do crosscutting concerns cause defects? IEEE Trans Softw Eng 34(4):497–515. https://doi.org/10.1109/TSE.2008.36

Fard AM, Mesbah A (2013) JSNOSE: detecting JavaScript code smells. In: 2013 IEEE 13th International working conference on source code analysis and manipulation (SCAM), pp 116–125. https://doi.org/10.1109/SCAM.2013.6648192

Florez JM, Moreno L, Zhang Z, Wei S, Marcus A (2022) An empirical study of data constraint implementations in Java (Replication Package). https://doi.org/10.5281/zenodo.6624695

Fowler M (2018) Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston

Gabel M, Jiang L, Su Z (2008) Scalable detection of semantic clones. In: Proceedings of the 30th international conference on software engineering, ICSE '08. Association for Computing Machinery, New York, pp 321–330. https://doi.org/10.1145/1368088.1368132

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co. Inc., Boston

Google (2021a) Guava: Google core libraries for java. https://github.com/google/guava

Google (2021b) Guava: preconditions. https://github.com/google/guava/wiki/PreconditionsExplained#preconditions

Guéhéneuc YG, Antoniol G (2008) DeMIMA: a multilayered approach for design pattern identification. IEEE Trans Softw Eng 34(5):667–684. https://doi.org/10.1109/TSE.2008.48

Guéhéneuc YG, Sahraoui HA, Zaidi F (2004) Fingerprinting design patterns. In: Proceedings of the 11th working conference on reverse engineering (WCRE), pp 172–181. https://doi.org/10.1109/WCRE.2004.21

Guo J, Zou Y (2008) Detecting clones in business applications. In: 2008 15th Working conference on reverse engineering, pp 91–100. https://doi.org/10.1109/WCRE.2008.12

Guo J, Cheng J, Cleland-Huang J (2017) Semantically enhanced software traceability using deep learning techniques. In: Proceedings of the 39th IEEE/ACM international conference on software engineering (ICSE), pp 3–14. https://doi.org/10.1109/ICSE.2017.9

Hatano T, Ishio T, Okada J, Sakata Y, Inoue K (2016) Dependency-based extraction of conditional statements for understanding business rules. IEICE Trans Inf Syst E99.D(4):1117–1126. https://doi.org/10.1587/transinf.2015EDP7202

Hay D, Healy KA (2000) Defining business rules ~ what are they really?, rev 1.3 edn. Business Rule Group

HTTP Working Group (2021) Hypertext transfer protocol—HTTP/1.0. https://www.w3.org/Protocols/HTTP/1.0/draft-ietf-http-spec.html

Huang H, Tsai WT, Bhattacharya S, Chen X, Wang Y, Sun J (1996) Business rule extraction from legacy code. In: Proceedings of the 20th international computer software and applications conference (COMPSAC), pp 162–167. https://doi.org/10.1109/CMPSAC.1996.544158

iTrust (2021a) Chronic disease risks. See replication package

iTrust (2021b) UC51 enter/edit basic health metrics. See replication package

JavaParser (2021) JavaParser. https://javaparser.org/

jEdit (2021) Closing and exiting. http://www.jedit.org/users-guide/closing-exiting.html

Joda-Time (2021) GregorianJulian (GJ) calendar system. https://www.joda.org/joda-time/calgj.html

Kaczor O, Guéhéneuc YG, Hamel S (2006) Efficient identification of design patterns with bit-vector algorithm. In: Proceedings of the 10th European conference on software maintenance and reengineering (CSMR), pp 10–184. https://doi.org/10.1109/CSMR.2006.25

Kapser C, Godfrey MW (2006) "Cloning considered harmful" considered harmful. In: Proceedings of the 13th working conference on reverse engineering (WCRE), pp 19–28. https://doi.org/10.1109/WCRE.2006.1

Komondoor R, Horwitz S (2001) Using slicing to identify duplication in source code. In: Cousot P (ed) Static analysis. Lecture Notes in Computer Science. Springer, Berlin, pp 40–56. https://doi.org/10.1007/3-540-47764-0_3

Krippendorff K (2004) Content analysis: an introduction to its methodology. Sage, Thousand Oaks

Kuang H, Nie J, Hu H, Rempel P, Lü J, Egyed A, Mäder P (2017) Analyzing closeness of code dependencies for improving IR-based traceability recovery. In: Proceedings of the 24th IEEE

international conference on software analysis, evolution and reengineering (SANER), pp 68–78. https://doi.org/10.1109/SANER.2017.7884610
Larman C (2005) Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development. In: Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3rd edn. PTR, 2005. Prentice Hall, Upper Saddle River
Livshits B, Sridharan M, Smaragdakis Y, Lhoták O, Amaral JN, Chang BYE, Guyer SZ, Khedker UP, Møller A, Vardoulakis D (2015) In defense of soundiness: a manifesto. Commun ACM 58(2):44–46. https://doi.org/10.1145/2644805
Mäder P, Jones PL, Zhang Y, Cleland-Huang J (2013) Strategic traceability for safety-critical projects. IEEE Softw 30(3):58–66. https://doi.org/10.1109/MS.2013.60
McMillan C, Poshyvanyk D, Revelle M (2009) Combining textual and structural analysis of software artifacts for traceability link recovery. In: Proceedings of the 5th ICSE workshop on traceability in emerging forms of software engineering (TEFSE). IEEE Computer Society, Washington, DC, pp 41–48. https://doi.org/10.1109/TEFSE.2009.5069582
Miles MB, Huberman AM, Saldaña J (2014) Qualitative data analysis: a methods sourcebook, 3rd edn. SAGE Publications, Inc, Thousand Oaks
Mirakhorli M, Cleland-Huang J (2016) Detecting, tracing, and monitoring architectural tactics in code. IEEE Trans Softw Eng 42(3):205–220. https://doi.org/10.1109/TSE.2015.2479217
Oualline S (1997) Practical C programming, 3rd edn. Nutshell Handbook. O'Reilly, Sebastopol
Pandita R, Xiao X, Zhong H, Xie T, Oney S, Paradkar A (2012) Inferring method specifications from natural language API descriptions. In: 2012 34th International conference on software engineering (ICSE), pp 815–825. https://doi.org/10.1109/ICSE.2012.6227137
Park C, Kang Y, Wu C, Yi K (2004) A static reference flow analysis to understand design pattern behavior. In: Proceedings of the 11th working conference on reverse engineering (WCRE), pp 300–301. https://doi.org/10.1109/WCRE.2004.9
Rahimi M, Goss W, Cleland-Huang J (2016) Evolving requirements-to-code trace links across versions of a software system. In: 2016 IEEE International conference on software maintenance and evolution (ICSME), pp 99–109. https://doi.org/10.1109/ICSME.2016.57
Razzaq A, Wasala A, Exton C, Buckley J (2018) The state of empirical evaluation in static feature location. Trans Softw Eng Methodol 28(1):2:1–2:58. https://doi.org/10.1145/3280988
Rempel P, Mäder P, Kuschke T, Cleland-Huang J (2014) Mind the gap: assessing the conformance of software traceability to relevant guidelines. In: Proceedings of the 36th IEEE/ACM international conference on software engineering (ICSE), ICSE 2014. Association for Computing Machinery, Hyderabad, pp 943–954. https://doi.org/10.1145/2568225.2568290
Rhino (2021) ECMAScript language specification. https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%203rd%20edition,%20December%201999.pdf
Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci Comput Program 74(7):470–495
Saied MA, Sahraoui H, Dufour B (2015) An observational study on API usage constraints and their documentation. In: 2015 IEEE 22nd International conference on software analysis, evolution, and reengineering (SANER), pp 33–42. https://doi.org/10.1109/SANER.2015.7081813
Shi N, Olsson RA (2006) Reverse engineering of design patterns from Java source code. In: Proceedings of the 21st IEEE/ACM international conference on automated software engineering (ASE), pp 123–134. https://doi.org/10.1109/ASE.2006.57
Sneed HM (2001) Extracting business logic from existing COBOL programs as a basis for redevelopment. In: Proceedings of the 9th IEEE workshop on program comprehension (IWPC), pp 167–175. https://doi.org/10.1109/WPC.2001.921728
Sneed HM, Erdös K (1996) Extracting business rules from source code. In: Proceedings of the 4th IEEE workshop on program comprehension (WPC), Berlin, pp 240–247. https://doi.org/10.1109/WPC.1996.501138
Sultanov H, Hayes JH, Kong WK (2011) Application of swarm techniques to requirements tracing. Requir Eng 16(3):209–226. https://doi.org/10.1007/s00766-011-0121-4
Swarm (2021) Seismic wave analysis and real-time monitor: user manual and reference guide. Version 2.8.10. https://github.com/usgs/swarm/blob/97f8b2f26830c764b816ca0a74270d5c0db35d06/docs/swarm_v2.pdf
Syed M, Nelson SC (2015) Guidelines for establishing reliability when coding narrative data. Emerging Adulthood 3(6):375–387. https://doi.org/10.1177/2167696815587648
Tan SH, Marinov D, Tan L, Leavens GT (2012) @tComment: testing Javadoc comments to detect comment-code inconsistencies. In: Verification and validation 2012 IEEE fifth international conference on software testing, pp 260–269. https://doi.org/10.1109/ICST.2012.106

Thummalapenta S, Cerulo L, Aversano L, Di Penta M (2010) An empirical study on the maintenance of source code clones. Empir Softw Eng 15(1):1–34. https://doi.org/10.1007/s10664-009-9108-x
Tip F (1994) A survey of program slicing techniques. Tech. rep. CWI, Centre for Mathematics and Computer Science, NLD
Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST (2006) Design pattern detection using similarity scoring. IEEE Trans Softw Eng 32(11):896–909. https://doi.org/10.1109/TSE.2006.112
WALA (2021) WALA: T.J. Watson Libraries for analysis. https://github.com/wala/WALA
Wan-Kadir WMN, Loucopoulos P (2004) Relating evolving business rules to software design. J Syst Archit 50(7):367–382. https://doi.org/10.1016/j.sysarc.2003.09.006
Wang X, Sun J, Yang X, He Z, Maddineni S (2004) Business rules extraction from large legacy systems. In: Proceedings of the 8th European conference on software maintenance and reengineering (CSMR), pp 249–258. https://doi.org/10.1109/CSMR.2004.1281426
Wiegers KE, Beatty J (2013) Software requirements, 3rd edn. Microsoft Press, Redmond
Witt GC (2012) Writing effective business rules: a practical method. Morgan Kaufmann, Waltham
Xiao X, Paradkar A, Thummalapenta S, Xie T (2012) Automated extraction of security policies from natural-language software documents. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, FSE '12. Association for Computing Machinery, New York, pp 1–11. https://doi.org/10.1145/2393596.2393608
Yang J, Sethi U, Yan C, Cheung A, Lu S (2020) Managing data constraints in database-backed web applications. In: Proceedings of the 42nd ACM/IEEE international conference on software engineering (ICSE), ICSE '20. Association for Computing Machinery, New York, pp 1098–1109. https://doi.org/10.1145/3377811.3380375
Zhou Y, Gu R, Chen T, Huang Z, Panichella S, Gall H (2017) Analyzing APIs documentation and code to detect directive defects. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 27–37. https://doi.org/10.1109/ICSE.2017.11
Zogaan W, Sharma P, Mirahkorli M, Arnaoudova V (2017) Datasets from fifteen years of automated requirements traceability research: current state, characteristics, and quality. In: Proceedings of the 25th IEEE international requirements engineering conference (RE), pp 110–121. https://doi.org/10.1109/RE.2017.80

Affiliations

Juan Manuel Florez[1] · Laura Moreno[2] · Zenong Zhang[1] · Shiyi Wei[1] · Andrian Marcus[1]

Laura Moreno
moreno@cqse-america.com

Zenong Zhang
zenong@utdallas.edu

Shiyi Wei
swei@utdallas.edu

Andrian Marcus
amarcus@utdallas.edu

[1] Department of Computer Science, The University of Texas at Dallas, Richardson, TX, USA

[2] CQSE America, Sunnyvale, CA, USA