i1-Welcome

Welcome to this fine-grained code patterns study. This survey is being conducted by Juan Manuel Florez and Andrian Marcus from the University of Texas at Dallas. We expect this survey to take **no more than 15 minutes** to complete.

You can email any questions or concerns to *jflorez@utdallas.edu*

i2-identifying-info

Please enter your four-digit participant number as it appears in the invitation email

What is the length of your experience with the Java programming language in a **professional** setting?

less than 1 year

Between 1 and 2 years

Between 2 and 3 years

Between 3 and 4 years

Between 4 and 5 years

5 or more years

Summary

Expected Time Allocation

The survey is divided in three parts. These are the estimated times for each:

1. **Background information:** approx. 5 minute reading.
2. **Sample question:** approx. 1 minute reading
3. **Survey questions:** 7 questions approx. 1 minute per question.

i3-intro-types

Background Information

Please read the following information carefully before proceeding with the rest of the survey

We are studying the implementations of *data constraints* in Java. We define a data constraint as *"any restriction of the possible values of a variable in the software domain".*

The following are some examples of data constraint descriptions as they would be found in software documents. The constraints are rewritten in a simplified language in the right column:

Constraint Description	Simplified Constraint
If **package *weight is greater than 20 kg***...	package weight > 20
The options for ***marital status*** *are:* ***single, married, or, divorced***...	marital status must be one of {single, married, divorced}
If the ***agent status is set to available*** at that time...	agent status must be "available"
The ***minimum frequency*** *value is set to* ***20*** by default...	minimum frequency = 20

We have identified a set of constraint implementation patterns (CIPs) that may be used to implement data constraints in Java code. The relevant definitions will appear alongside each question, so **you do not need to memorize them**. We present two definitions here for illustration.

> *In all the definitions, when we refer to a value, we mean any construct that returns a value, specifically a variable access, method call, or field access.*

Pattern name: null-check.
Pattern description: A value is checked for nullity using the == or != operators. The value may be the first or second operator.
Pattern generic form:

```
nullableValue == null

null != obj.nullableValue

obj.getNullableValue() == null
```

Pattern example: The portion of the following code highlighted in yellow is an example of the **null-check** pattern in real code:

```
1    // Read WIN configuration file
2    if (configFile != null) {
3      FileReader fileReader = new FileReader(configFile);
4      BufferedReader reader = new BufferedReader(fileReader);
5      timeZone = reader.readLine();
6      channelInfo.clear();
7      ...
```

Pattern name: null-empty-check
Pattern description: 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.
Pattern generic form:

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

obj.stringValue == null || obj.stringValue.equals("")

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

Pattern example: The portion of the following code highlighted in yellow is an example of the **null-empty-check** pattern in real code:

```
1    private static boolean nullOrEmpty(Object value) {
2        return value == null || "".equals(value);
3    }
```

Note that the patterns that are defined on expressions can appear in any statement type where expressions are grammatically correct in Java, for example:

```
if( value != null ) // null-check in if statement

return value != null; // null-check in return statement

while( value != null ) // null-check in while statement

boolean res = value == null || "".equals(value); // null-empty-check in assignmer
```

i4-instructions

Sample question

In the next section, you will be shown 7 constraints and their implementations, one by one.

You will be asked to select the pattern that the implementation matches from a list of options, or "None of the above" if it matches none of the options.

For example:

Consider the bold text in the following paragraph:

*If **configuration file is not available** or readable it will default to 'UTC'.*

Which contains the constraint:

configuration file is not available

And is implemented in the highlighted portion of this code:

```
1    // Read WIN configuration file
2    if (configFile != null) {
3      FileReader fileReader = new FileReader(configFile);
4      BufferedReader reader = new BufferedReader(fileReader);
5      timeZone = reader.readLine();
6      channelInfo.clear();
7      ...
```

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

Given the following pattern definitions:

null-check: A value is checked for nullity using the == or != operators. The value may be the first or second operator.

```
nullableValue == null

null != obj.nullableValue
```

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.

```
obj.stringValue == null || obj.stringValue.equals("")

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

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

- null-empty-check
- null-check
- None of the above

The correct answer in this example is *null-check*.

jedit-28

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

> *The Max number of backups setting determines the number of backups to save. Setting this to zero disables the backup feature. Settings this to more than one adds numbered suffixes to file names.* ***By default only one backup is saved****.*

Which contains the constraint:

> *saved backups = 1*

And is implemented in the highlighted portion of this code:

```
1    # Backup on every save
2    backupEverySave=false
3
4    # Number of backups to make, 0=no backups
5    backups=1
6
7    # Backup directory suggestion: $JEDIT_SETTINGS/backups
8    backup.directory=
```

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

Given the following pattern definitions:

properties-file: The default value for a variable is stored in a properties file.

```
value=100

value=true

value=default string
```

if-chain: 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.

```
if (value == 1) {
    ...
} else if (value == 2) {
    ...
} else if (value == 3) {
    ...
```

assign-constant: A literal value is assigned to a variable or field.

```
int val = 5000

public static final String STRING_VAL = "value"
```

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

properties-file

if-chain

assign-constant

None of the above

ant-35

(Constraint 2/7) Consider the bold text in the following paragraph:

> *If the **build.sysclasspath property** has not been set, **it defaults to ignore** in this case.*

Which contains the constraint:

> *build.sysclasspath = ignore*

And is implemented in the highlighted portion of this code:

```
1    if (bootclasspath != null) {
2        bcp.append(bootclasspath);
3    }
4    bcp = bcp.concatSystemBootClasspath("ignore");
5    if (bcp.size() > 0) {
6        toExecute.createArgument().setValue("-bootclasspath");
7        toExecute.createArgument().setPath(bcp);
8    }
```

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

Given the following pattern definitions:

assign-constant: A literal value is assigned to a variable or field.

```
int val = 5000

public static final String STRING_VAL = "value"
```

binary-comparison: Two variables are compared using one of the relational operators (==, !=, >, <, >=, <=). Use of the equals method is considered an operator in this case. Neither of the operands may be the literal 'null'.

```
obj1.valueA <= obj2.valueB

obj.getValueA().equals(valueB)
```

constant-argument: A literal value (string, integer, boolean) is passed as a parameter to a method call. The call can have other non-literal parameters.

```
obj.method("value")

obj.method(1, arg2, obj.arg3)
```

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

assign-constant

binary-comparison

constant-argument

None of the above

ant-19

(Constraint 3/7) Consider the bold text in the following paragraph:

> *If **any of the sources has been modified** more recently than any of the target files, all of the target files are removed.*

Which contains the constraint:

> *source is modified*

And is implemented in the highlighted portion of this code:

```
1         Resource newestSource = getNewest(sources);
2         logWithModificationTime(newestSource, "newest source");
3         return oldestTarget.getLastModified() >= newestSource.getLastModified()
4     }
```

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

Given the following pattern definitions:

if-chain: 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.

```
if (value == 1) {
    ...
} else if (value == 2) {
```

```
    ...
} else if (value == 3) {
    ...
```

binary-comparison: Two variables are compared using one of the relational operators (==, !=, >, <, >=, <=). Use of the equals method is considered an operator in this case. Neither of the operands may be the literal 'null'.

```
obj1.valueA <= obj2.valueB


obj.getValueA().equals(valueB)
```

assign-constant: A literal value is assigned to a variable or field.

```
int val = 5000

public static final String STRING_VAL = "value"
```

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

if-chain

binary-comparison

assign-constant

None of the above

jedit-17-false

(Constraint 4/7) Consider the bold text in the following paragraph:

> ***Files that you do not have write access to*** *are opened in read-only mode, where editing is not permitted.*

Which contains the constraint:

> *file is not writable*

And is implemented in the highlighted portion of this code:

```
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)

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())
```

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
```

binary-comparison: Two variables are compared using one of the relational operators (==, !=, >, <, >=, <=). Use of the equals method is considered an operator in this case. Neither of the operands may be the literal 'null'.

```
obj1.valueA <= obj2.valueB

obj.getValueA().equals(valueB)
```

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

null-empty-check

binary-flag-check

binary-comparison

None of the above

argouml-77-false

(Constraint 5/7) Consider the bold text in the following paragraph:

> ***Multiplicity**: Editable drop down selector with checkmark. The **default value (1)** is that there is one instance of this attribute for each instance of the class, i.e. it is a scalar.*

Which contains the constraint:

> *multiplicity = 1*

And is implemented in the highlighted portion of this code:

```
1    if (comboText == null) {
2                Model.getCoreHelper().setMultiplicity(getTarget(), "1");
3    } else {
4                Model.getCoreHelper().setMultiplicity(getTarget(), comboText);
5    }
```

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

Given the following pattern definitions:

binary-comparison: Two variables are compared using one of the relational operators (==, !=, >, <, >=, <=). Use of the equals method is considered an operator in this case. Neither of the operands may be the literal 'null'.

```
obj1.valueA <= obj2.valueB

obj.getValueA().equals(valueB)
```

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")
```

null-check: A value is checked for nullity using the == or != operators.

```
value == null

null != obj.value
```

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

binary-comparison

equals-or-chain

null-check

None of the above

ant-36

(Constraint 6/7) Consider the bold text in the following table excerpt:

*The values of **build.sysclasspath** and their meanings are:*

value	*meaning*
only	Only the system classpath is used...
ignore	The system classpath is ignored...
last	The classpath is concatenated to any specified classpaths at the end...
first	Any specified classpaths are concatenated to the system classpath...

Which contains the constraint:

build.sysclasspath must be one of {only, ignore, last, first}

And is implemented in the highlighted portion of this code:

```
 1    String o = getProject() != null
 2        ? getProject().getProperty(MagicNames.BUILD_SYSCLASSPATH)
 3        : System.getProperty(MagicNames.BUILD_SYSCLASSPATH);
 4    if (o != null) {
 5        order = o;
 6    }
 7    if ("only".equals(order)) {
 8        // only: the developer knows what (s)he is doing
 9        result.addExisting(p, true);
10
11    } else if ("first".equals(order)) {
12        // first: developer could use a little help
13        result.addExisting(p, true);
14        result.addExisting(this);
15
16    } else if ("ignore".equals(order)) {
17        // ignore: don't trust anyone
18        result.addExisting(this);
19
20    } else {
21        // last: don't trust the developer
22        if (!"last".equals(order)) {
23            log("invalid value for " + MagicNames.BUILD_SYSCLASSPATH
24                + ": " + order,
25                Project.MSG_WARN);
26        }
27        result.addExisting(this);
28        result.addExisting(p, true);
29    }
```

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

Given the following pattern definitions:

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
```

assign-constant: A literal value is assigned to a variable or field.

```
int val = 5000

public static final String STRING_VAL = "value"
```

if-chain: 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.

```
if (value == 1) {
    ...
} else if (value == 2) {
    ...
} else if (value == 3) {
    ...
```

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

binary-flag-check

assign-constant

if-chain

None of the above

swarm-31

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

> *Wave Options*
> *...*
> ***Remove bias** will remove the mean value from the wave if on. It is **enabled by default**."*

Which contains the constraint:

> *remove bias = enabled*

And is implemented in the highlighted portion of this code:

```
1    // view option
2    DEFAULT_WAVE_VIEW_SETTINGS.viewType = ViewType.WAVE;
```

```
3    // wave options
4    DEFAULT_WAVE_VIEW_SETTINGS.removeBias = true;
5    DEFAULT_WAVE_VIEW_SETTINGS.autoScaleAmp = true;
6    DEFAULT_WAVE_VIEW_SETTINGS.autoScaleAmpMemory = true;
7    DEFAULT_WAVE_VIEW_SETTINGS.waveMaxAmp = 1000;
```

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

Given the following pattern definitions:

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
```

assign-constant: A literal value is assigned to a variable or field.

```
int val = 5000

public static final String STRING_VAL = "value"
```

if-chain: 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.

```
if (value == 1) {
    ...
} else if (value == 2) {
    ...
} else if (value == 3) {
    ...
```

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

binary-flag-check

assign-constant

if-chain

None of the above