

IDE-assisted visualization of indebted OO variability implementations

Johann Mortara – Philippe Collet – Anne-Marie Dery-Pinna

Université Côte d'Azur, CNRS, I3S, France

SPLC '22 – Graz, Austria

September 15, 2022

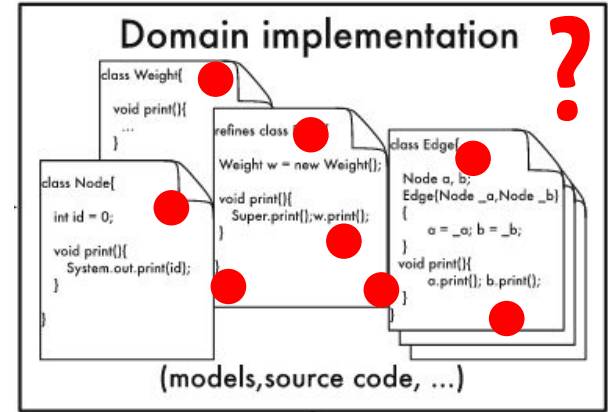
OO codebases use OO mechanisms to implement variability in a single codebase

- inheritance
- overloading of methods and constructors
- design patterns

Creation of **complex zones** in the system

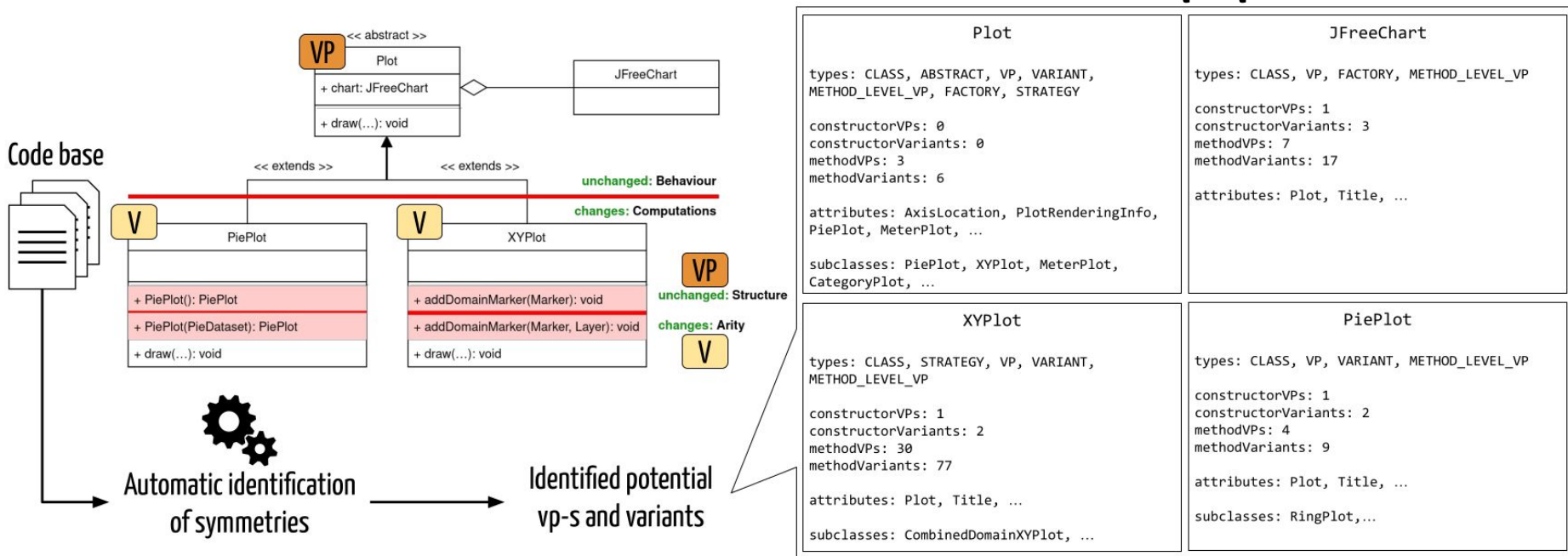
⇒ **understanding them is crucial** to comprehend the codebase variability

**Undocumented
OO variability
implementations**



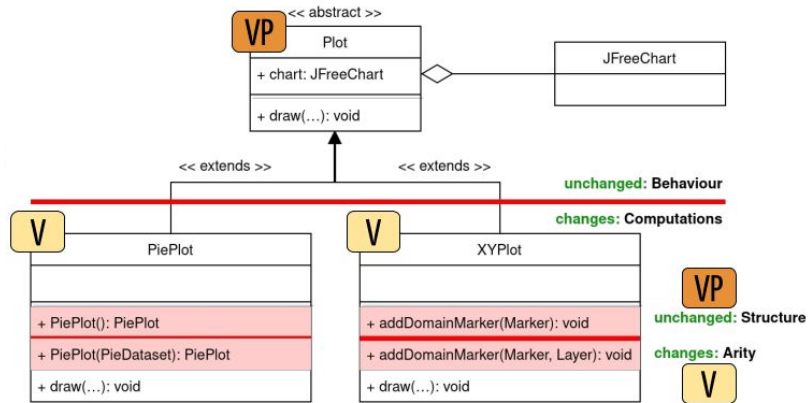
Automatic identification of variability implementations in an OO codebase

metrics / properties



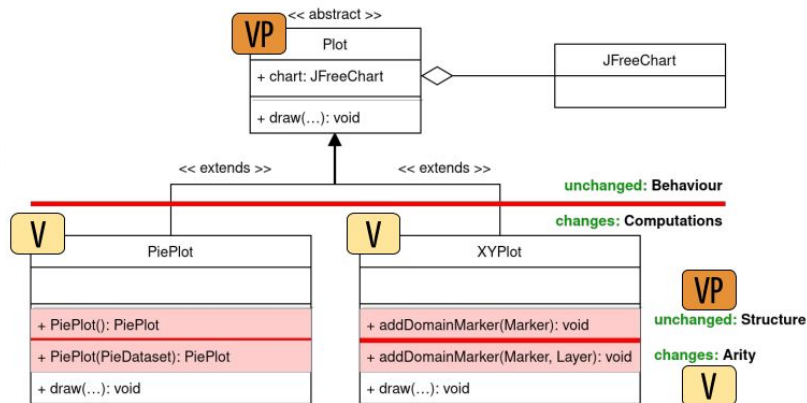
The reality of OO variability implementation

Variability implemented using mechanisms



The reality of OO variability implementation

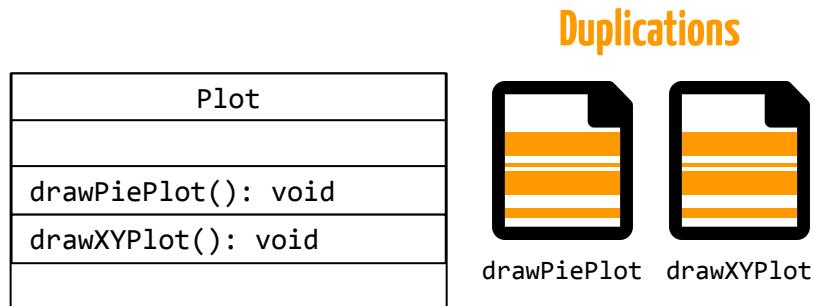
Variability implemented using mechanisms



Duplicated blocks: 3
Code coverage: 80%

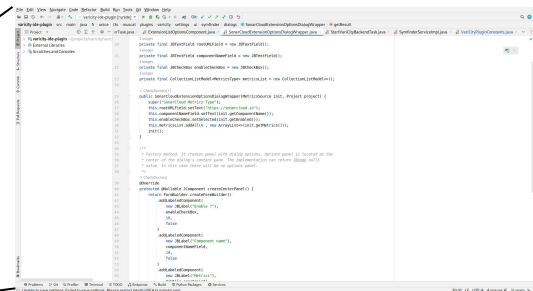
Duplicated blocks: 2
Code coverage: 75%

Variability implemented without using mechanisms

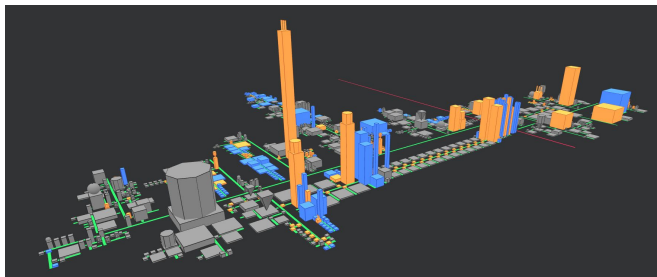


Duplicated blocks: 25
Code coverage: 55%

⇒ technical debt



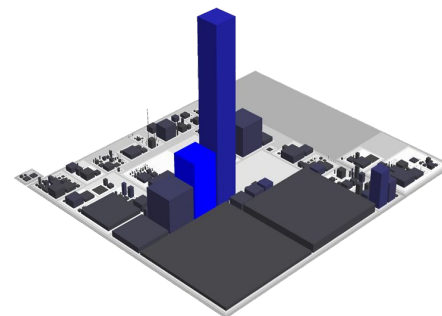
IDE



VariCity

Displays variability implementations

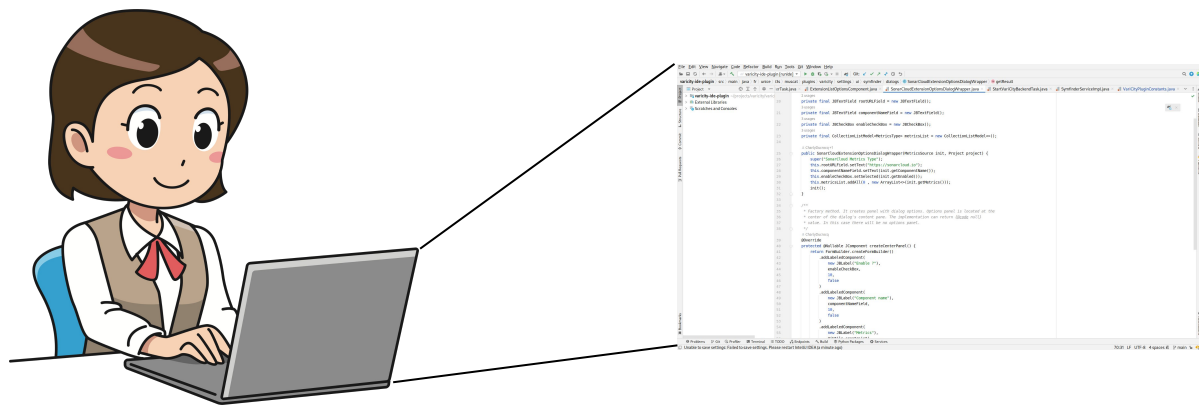
Displays quality metrics



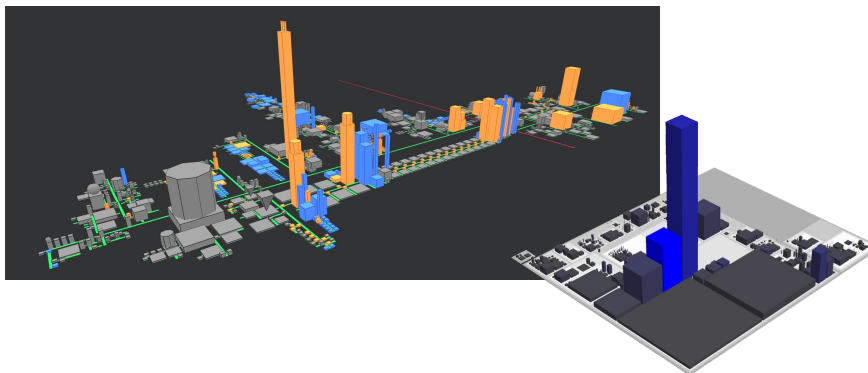
CodeCity

Displays variability implementations

Displays quality metrics



IDE



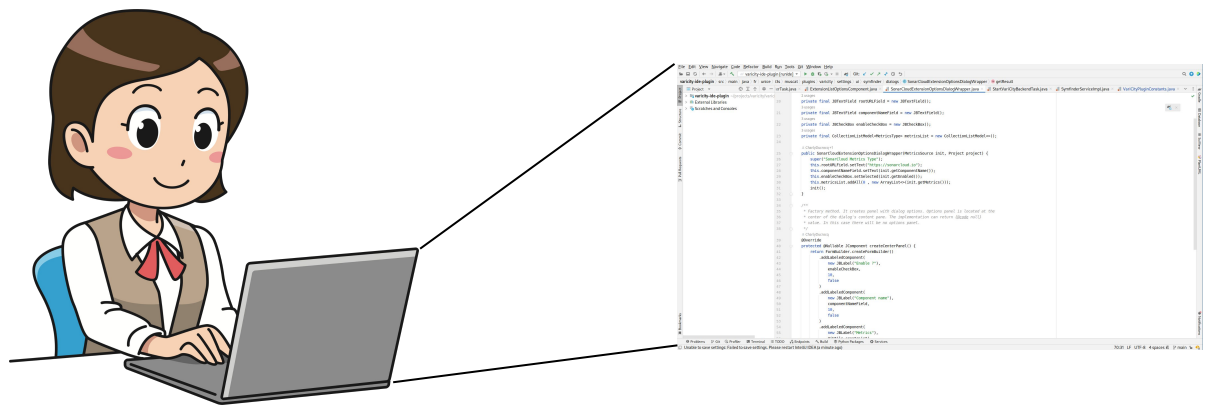
VariCity

Displays variability implementations ✓

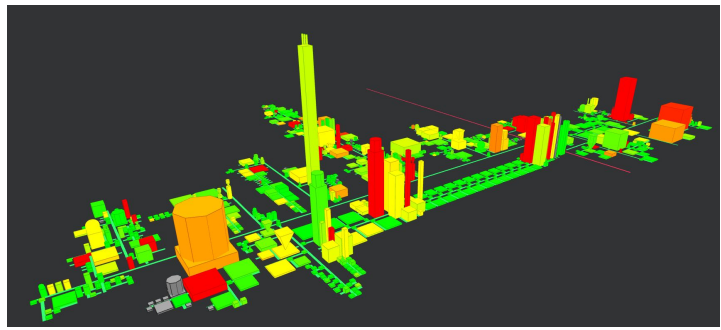
+ **CodeCity**

Displays quality metrics ✓

Same metaphor but different representations → cognitive load



IDE



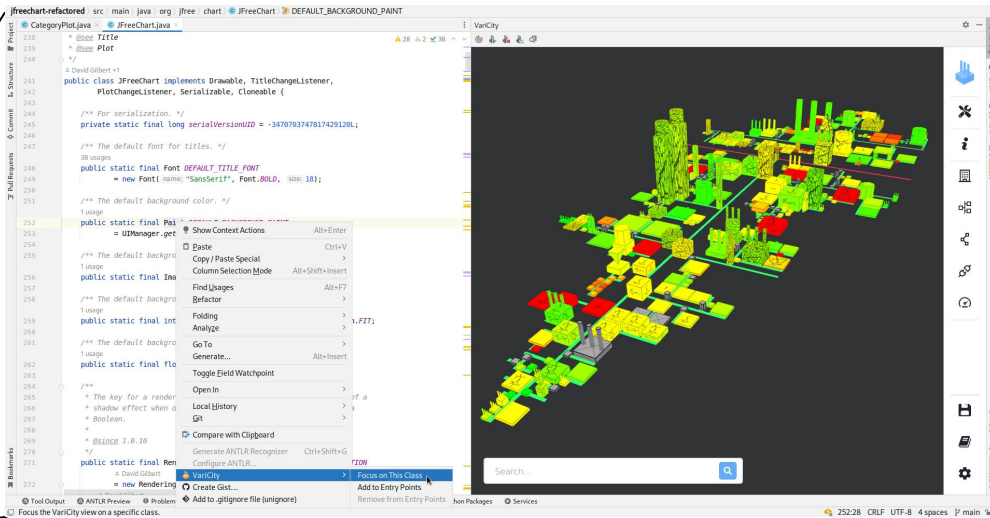
VariMetrics

Displays variability implementations ✓

Displays quality metrics ✓

Repeated switches between multiple environments

VariMetrics-IDE



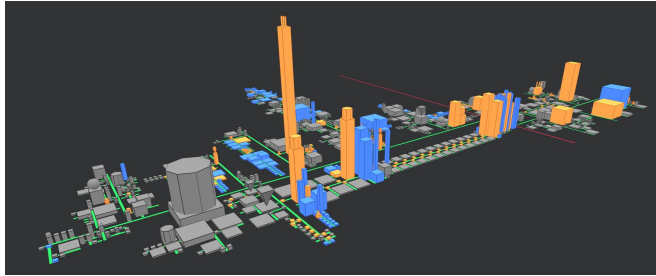
Displays variability implementations ✓

Displays quality metrics ✓

Integrated in the developer's environment

What can we use?

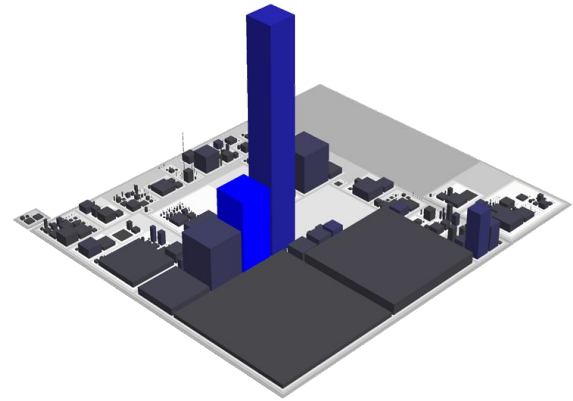
VariCity



Displays variability implementations

Displays quality metrics

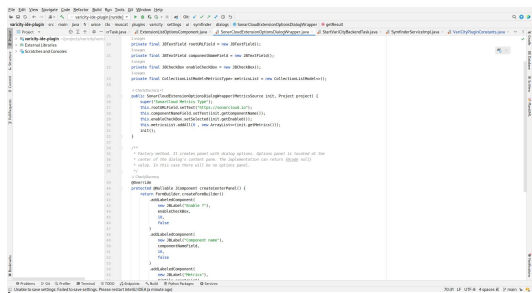
CodeCity



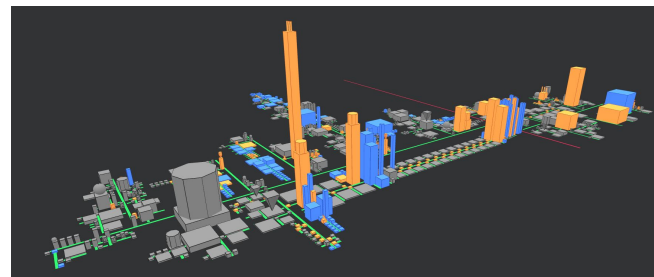
Displays variability implementations

Displays quality metrics

What can we use?



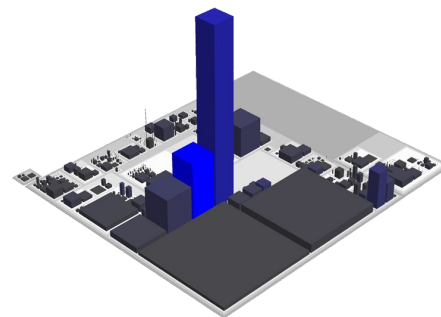
IDE



VariCity

Displays variability implementations ✓

Displays quality metrics ✗



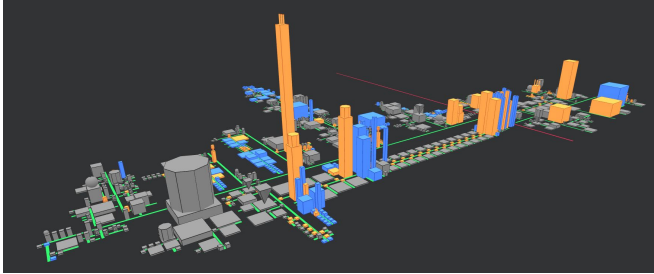
CodeCity

Displays variability implementations ✗

Displays quality metrics ✓

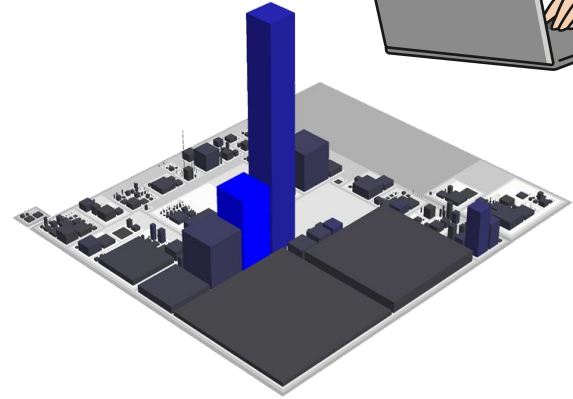
What can we use?

VariCity



Displays variability implementations
Displays quality metrics

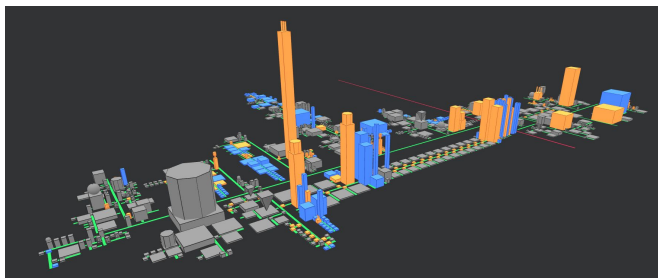
CodeCity



Displays variability implementations
Displays quality metrics

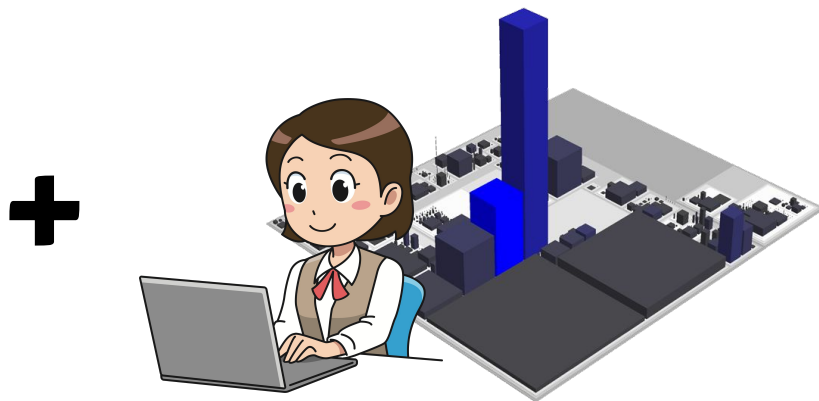
What can we use?

VariCity



AND

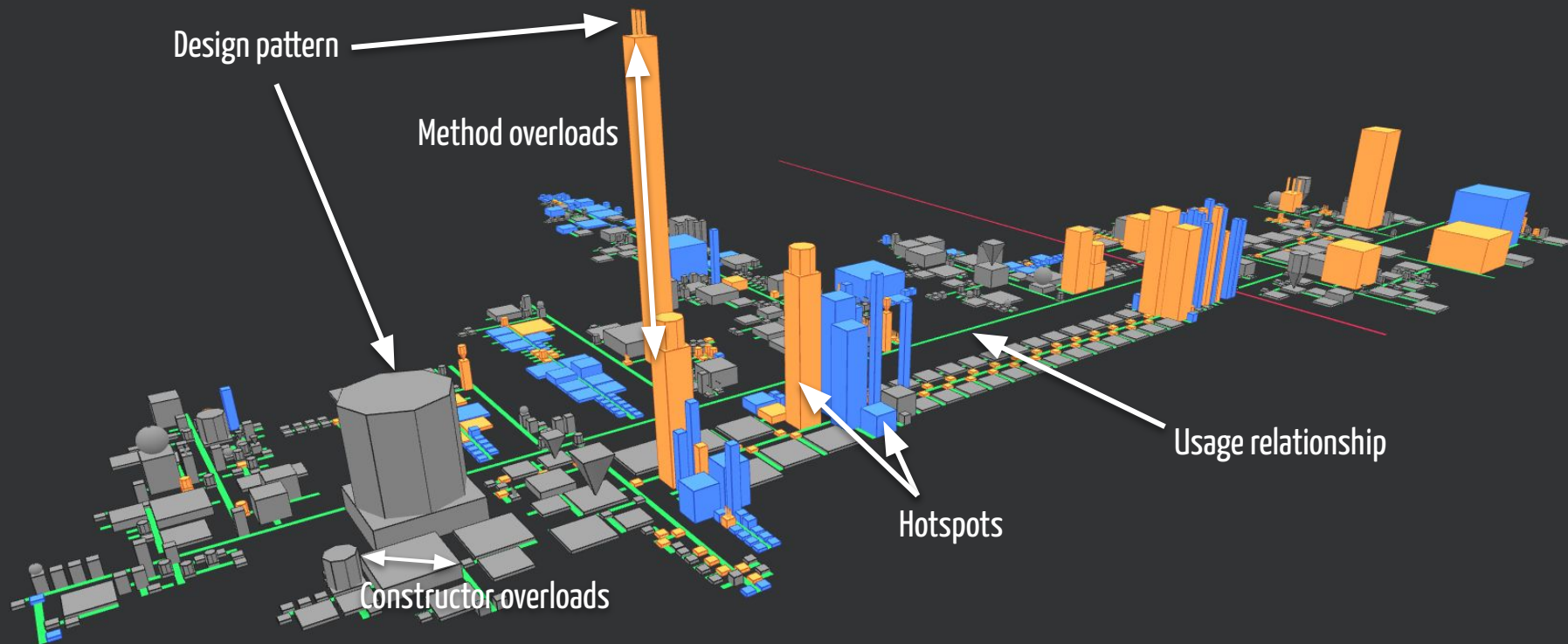
CodeCity



Cumbersome to do!

- Same metaphor but different representations → cognitive load
- Switch between multiple environments is hard for developers

VariCity: a visualization for OO variability implementations



Limitations of VariCity

Variability debt

”Technical debt **caused by defects and sub-optimal solutions in the implementation of variability** management in software systems. [...] Variability debt **leads to maintenance and evolution difficulties** to manage families of systems or highly configurable systems.”

Potential identified causes:

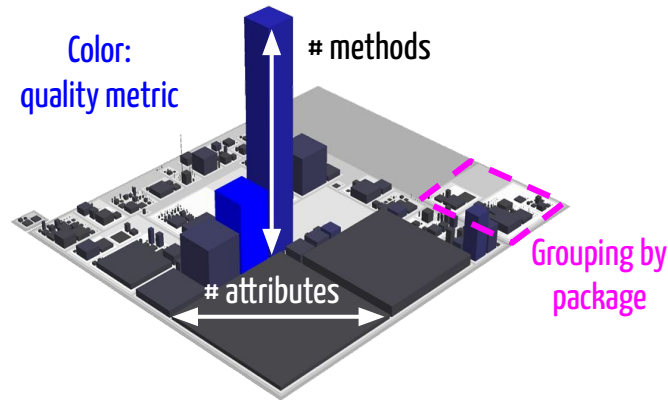
- lack of knowledge of the implemented variability
- absence of traceability
- no known variability implementation mechanisms \Rightarrow artifact duplication + \uparrow code complexity

OO variability implementations are prone to variability debt

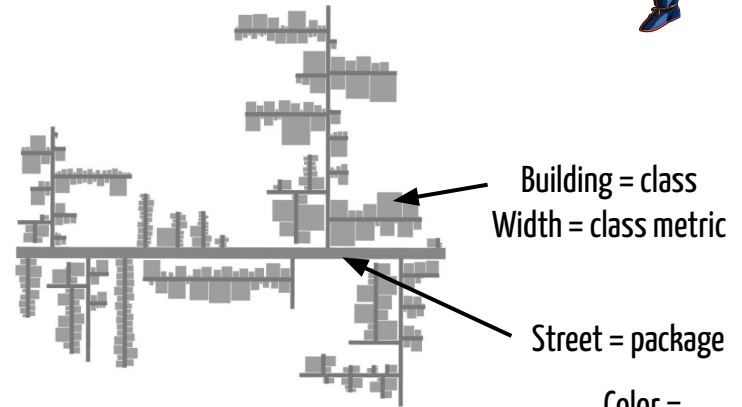
Need for identification and visualization

Identifying and visualizing technical debt in OO variability implementations

OO code quality metrics visualized in the form of a city



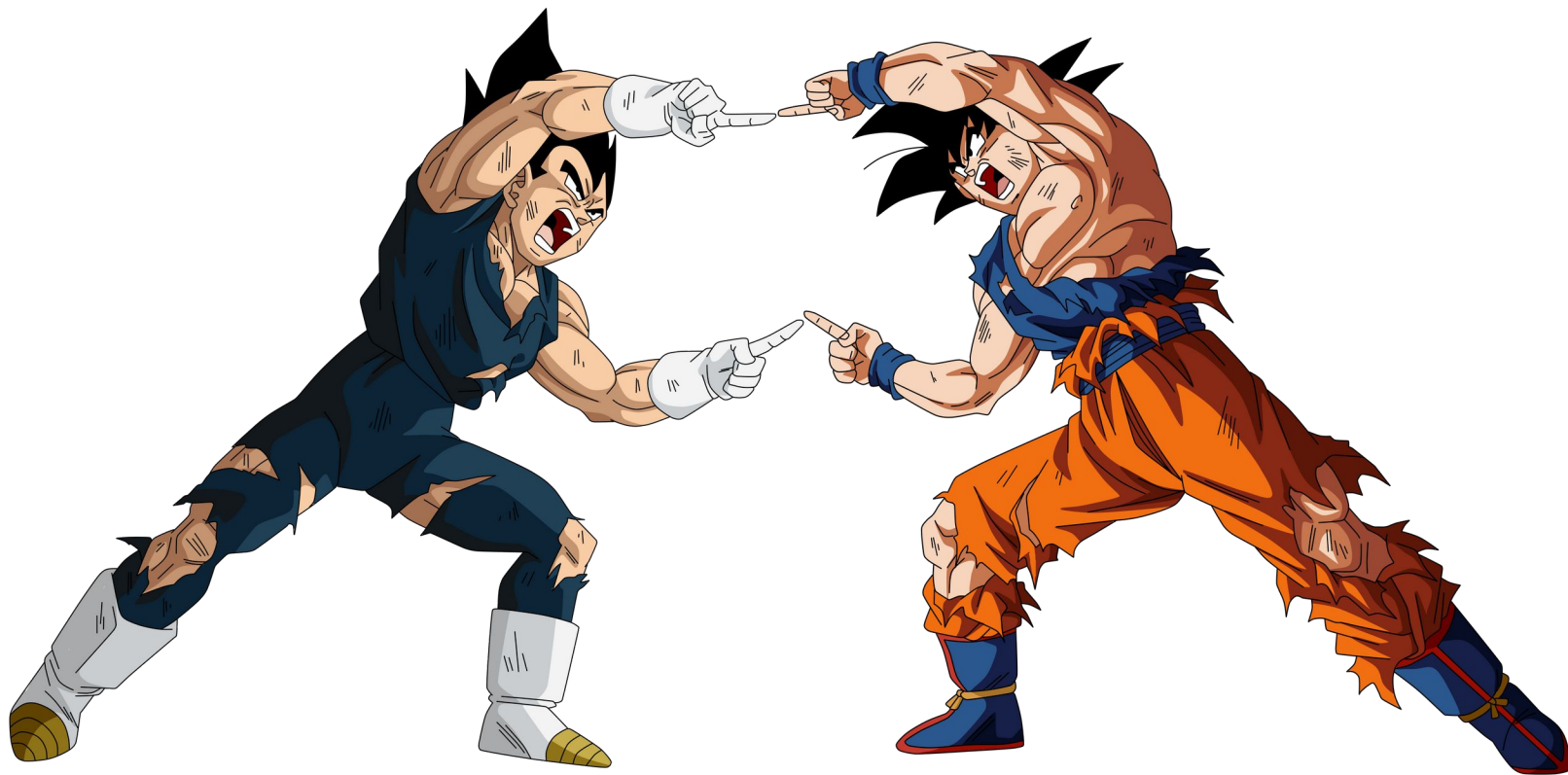
CodeCity



Evo-Streets



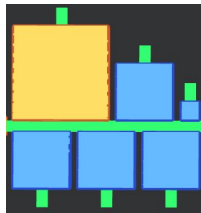
FUUUUUUUUUSION!



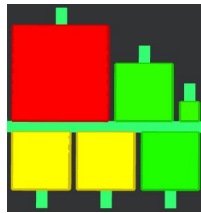
VariMetrics

Configurable visualization of OO quality metrics on the variability implementations

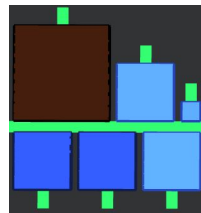
- Additional visual axes that can be combined to display multiple metrics simultaneously
- Ranges of values for metrics are also configurable



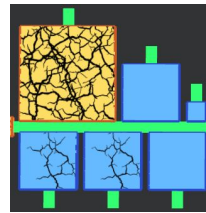
VariCity



Red-green



Saturation



Cracks

Metrics

Variables

width	nbConstructorVariants	▼
height	nbMethodVariants	▼
intensity	-- None --	▼
fade	coverage	▼
crack	duplicated_blocks	▼

Metrics

complexity

Min 0 Max 25 Lower is Better Higher is Better

cognitive_complexity

Min 0 Max 150 Lower is Better Higher is Better

duplicated_lines

Min 0 Max 100 Lower is Better Higher is Better

coverage

Min 0 Max 100 Lower is Better Higher is Better

nbMethodVariants

Min 0 Max 100 Lower is Better Higher is Better

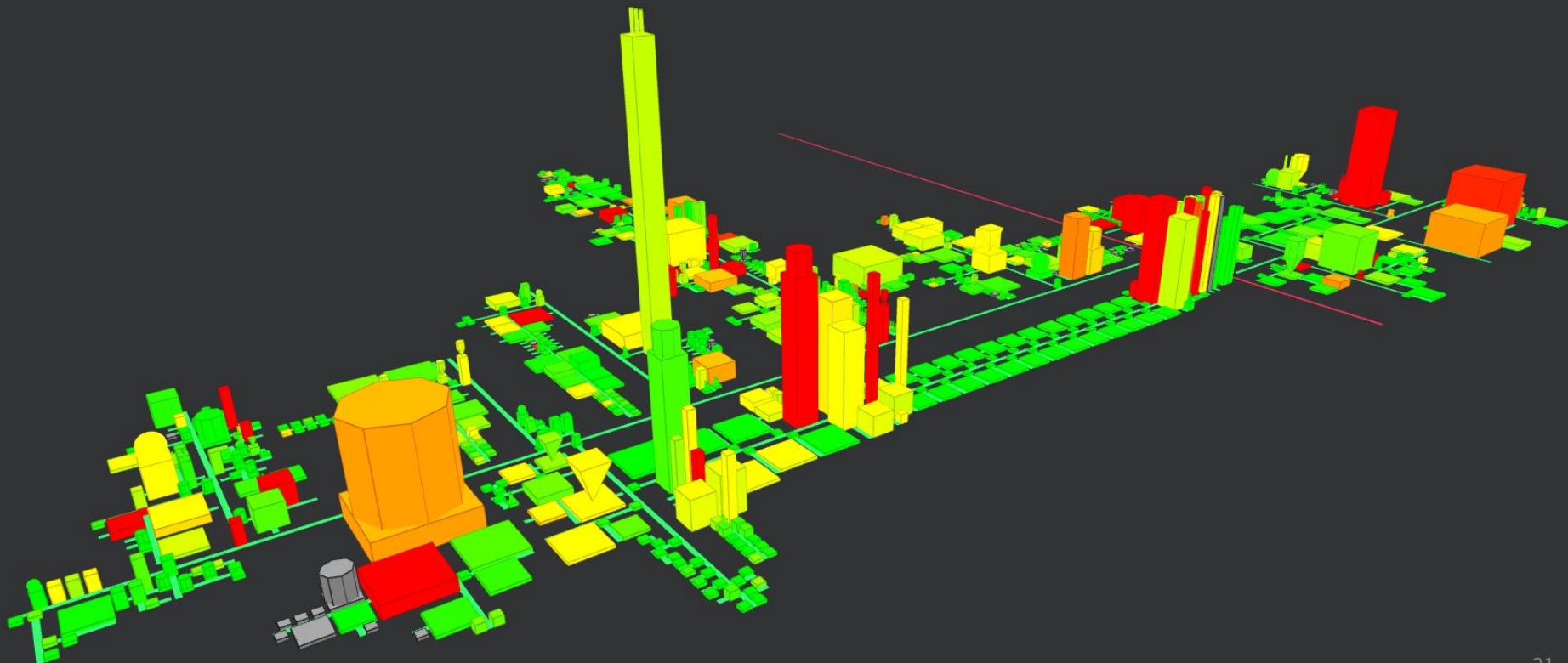
nbAttributes

Min 0 Max 100 Lower is Better Higher is Better

Example of VariMetrics view

Project: **GeoTools**

Red-to-green color scale: **cognitive complexity**



Quantitative evaluation

Does VariMetrics allow to visualize indebted zones of variability implementations?

Visual observation on **7 medium to large open source variable systems written in Java**

Determination of **relevant** VariMetrics visualizations:

- w.r.t. variability → some classes exhibit concentration of variability implementation mechanisms
- w.r.t. quality → some classes have quality issues

Quantitative evaluation

Does VariMetrics allow to visualize indebted zones of variability implementations?

Visual observation on **7 medium to large open source variable systems written in Java**

Determination of **relevant** VariMetrics visualizations:

- w.r.t. variability → some classes exhibit concentration of variability implementation mechanisms
- w.r.t. quality → some classes have quality issues **related to which metrics?**

Determining relevant quality metrics

Different types of variability debt

- System-level structure quality issues
- Code Duplication
- Lack of tests
- Out-of-date or incomplete documentation
- Architectural antipatterns
- Expensive tests
- Multi-version support
- Old technology in use
- Duplicate documentation
- Poor test of feature interactions



Determining relevant quality metrics

Different types of variability debt **applicable to OO codebases**

- System-level structure quality issues
in the implementation
- Code Duplication
- Lack of tests
- Expensive tests
- Multi-version support
- Old technology in use
- Out-of-date or incomplete documentation
- Duplicate documentation
- Architectural antipatterns
- Poor test of feature interactions

Determining relevant quality metrics

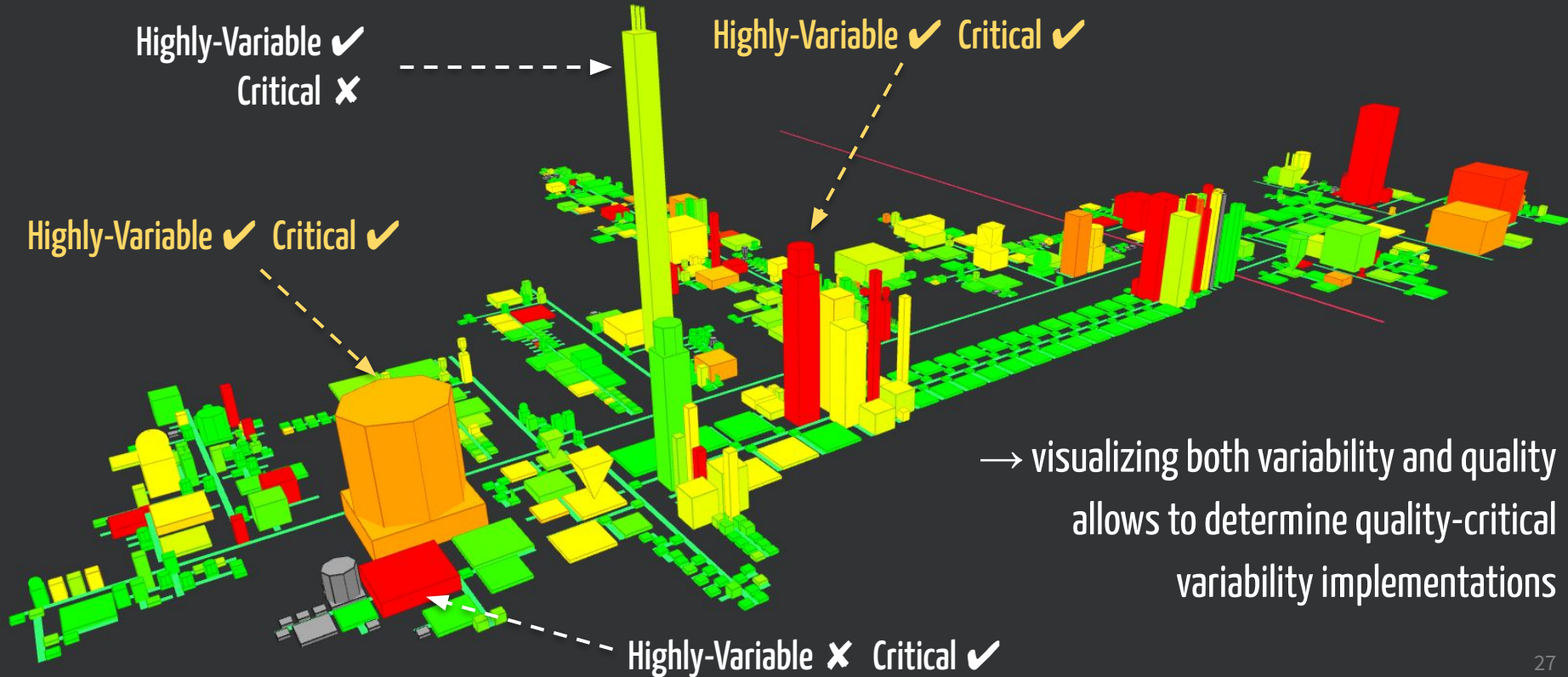
Different types of variability debt **applicable to OO codebases**

- System-level structure quality issues 
in the implementation
- Code Duplication 
- Lack of tests 

Chosen OO metrics

- Cognitive complexity
- Duplicated code blocks
- Unit tests coverage

No variability / quality correlation

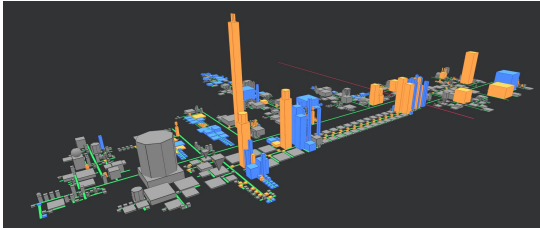


Quantifying the noticeable classes on VariMetrics

Protocol:

Quantifying the noticeable classes on VariMetrics

Protocol:

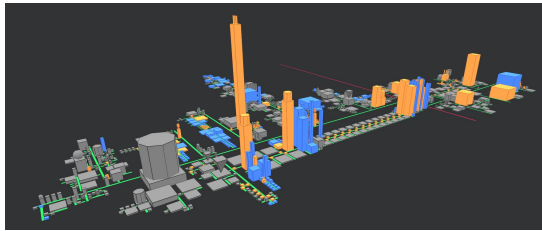


VariCity view

Visual identification of
noticeable classes
w.r.t. **variability**

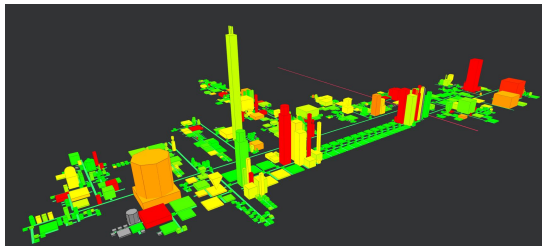
Quantifying the noticeable classes on VariMetrics

Protocol:



VariCity view

Visual identification of noticeable classes w.r.t. **variability**

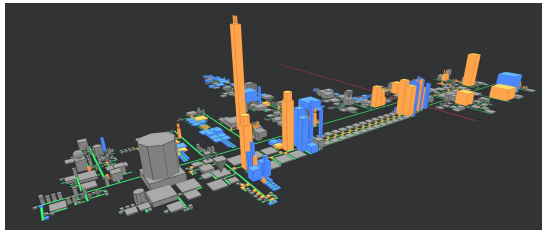


VariMetrics view

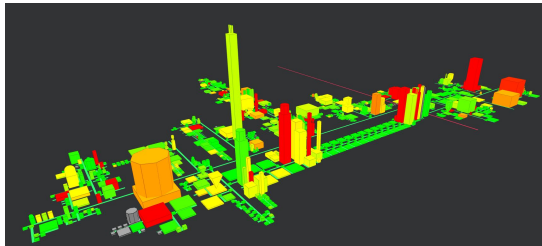
Visual identification of noticeable classes w.r.t. **criticality**

Quantifying the noticeable classes on VariMetrics

Protocol:



VariCity view



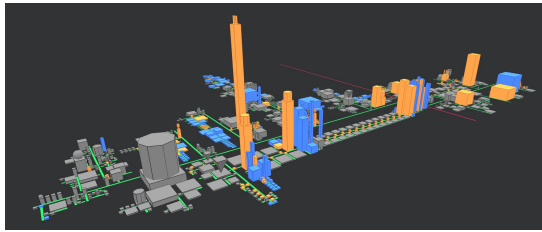
VariMetrics view

Visual identification of noticeable classes w.r.t. **variability**

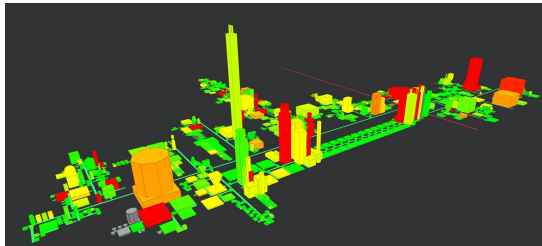
Visual identification of noticeable classes w.r.t. **criticality**

Quantifying the noticeable classes on VariMetrics

Protocol:



VariCity view



VariMetrics view

Visual identification of noticeable classes w.r.t. **variability**

Visual identification of noticeable classes w.r.t. **criticality**

Visible classes w.r.t. **variability and criticality**

Quantitative evaluation

Less relevant classes with VariMetrics than with VariCity

Results depend mainly on:

- **codebase size**
↑ codebase size ⇒ ↑ identified variability
intense zones
 - **global quality**
↓ quality ⇒ ↑ noticeable classes
- explains mildly encouraging results on JKube

System	Visible classes w.r.t.			% reduction VariCity → VariMetrics
	variability	criticality	both	
Azureus	74	32	12	84 %
GeoTools	104	27	18	83 %
JDK	84	17	13	85 %
JFreeChart	35	31	10	71 %
JKube	28	115	14	50 %
OpenAPI Generator	77	51	21	72 %
Spring framework	57	13	6	91 %

Qualitative evaluation on JFreeChart

Are the shown indebted zones of variability implementations relevant?

Qualitative evaluation on JFreeChart

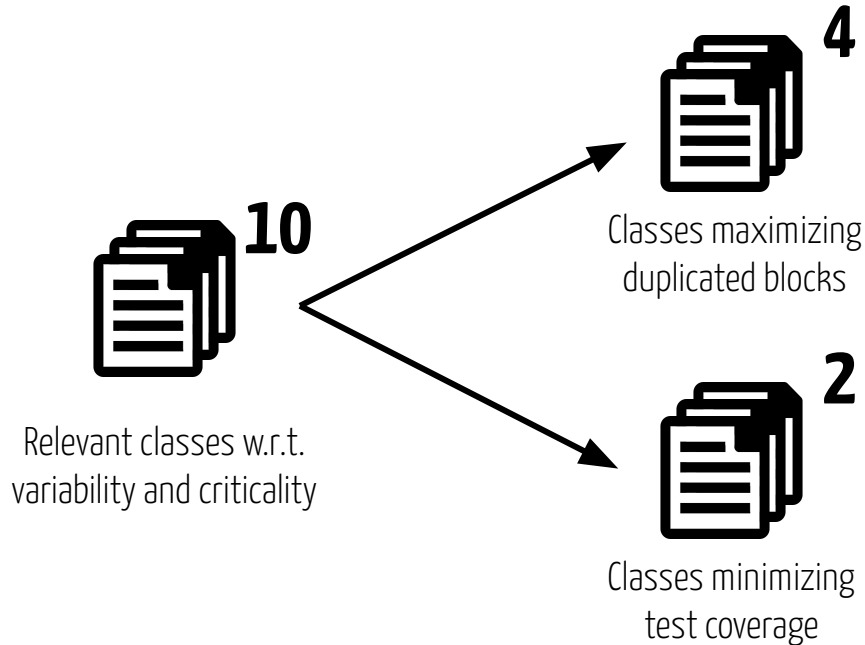
Are the shown indebted zones of variability implementations relevant?



Relevant classes w.r.t.
variability and criticality

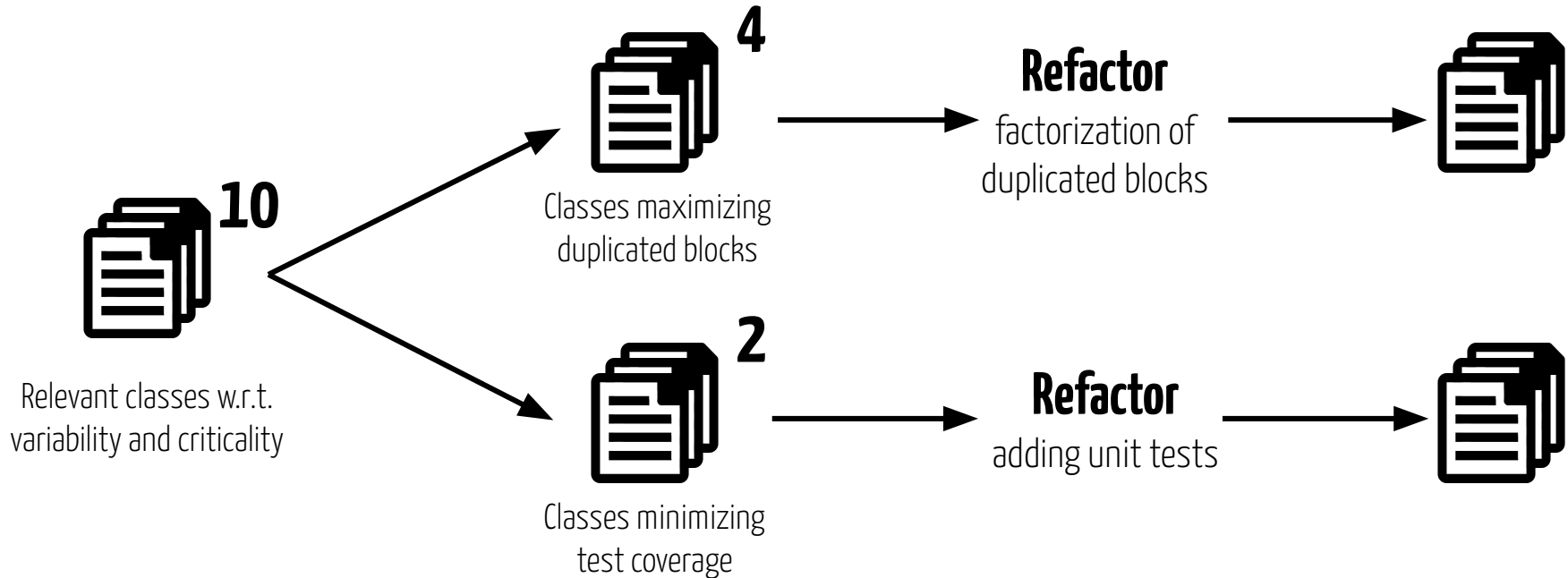
Qualitative evaluation on JFreeChart

Are the shown indebted zones of variability implementations relevant?



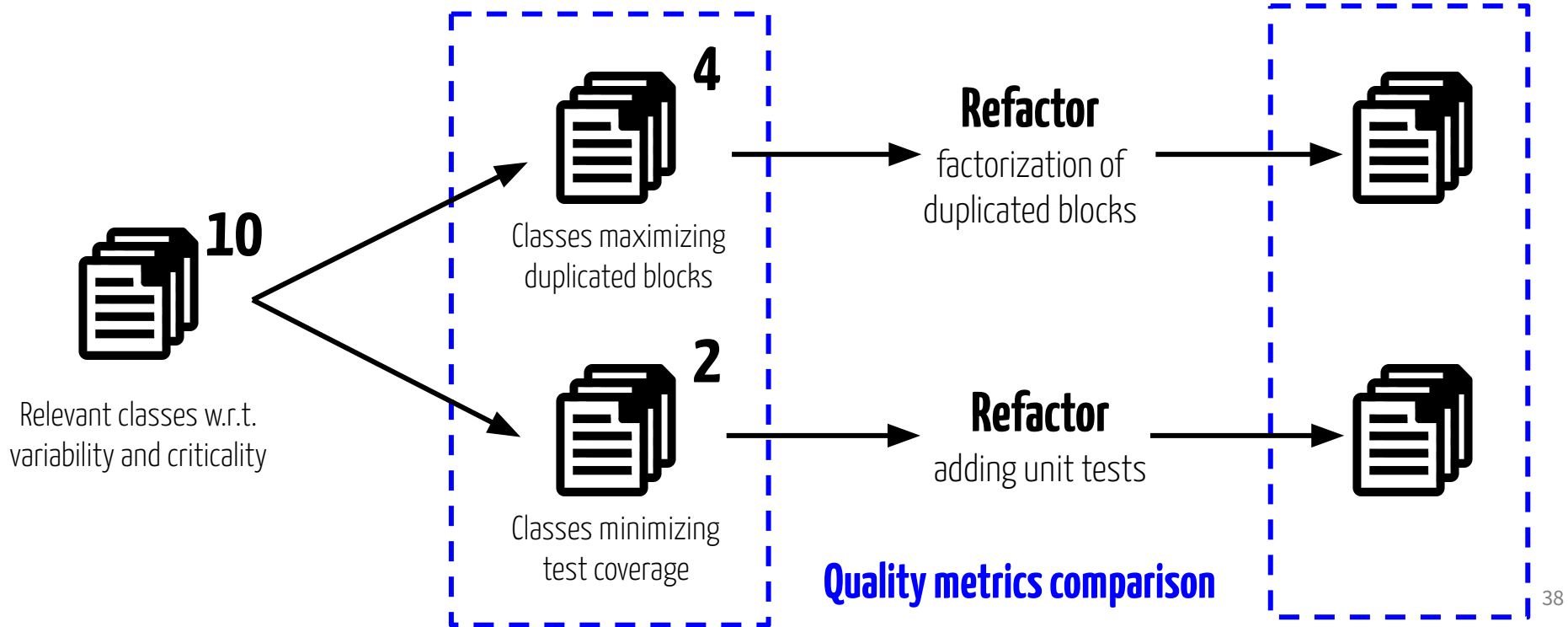
Qualitative evaluation on JFreeChart

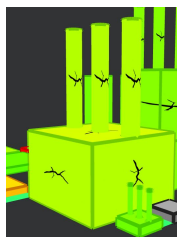
Are the shown indebted zones of variability implementations relevant?



Qualitative evaluation on JFreeChart

Are the shown indebted zones of variability implementations relevant?





Findings

Duplications can be pure technical debt in classes concentrating variability implementations, but can also be improperly managed variability implementations

```
if (isVerticalTickLabels()) {
    anchor = TextAnchor.CENTER_RIGHT;
    rotationAnchor = TextAnchor.CENTER_RIGHT;
    if (edge == RectangleEdge.TOP) {
        angle = Math.PI / 2.0;
    }
    else {
        angle = -Math.PI / 2.0;
    }
}
else {
    if (edge == RectangleEdge.TOP) {
        anchor = TextAnchor.BOTTOM_CENTER;
        rotationAnchor = TextAnchor.BOTTOM_CENTER;
    }
    else {
        anchor = TextAnchor.TOP_CENTER;
        rotationAnchor = TextAnchor.TOP_CENTER;
    }
}
```

refreshTicksHorizontal

```
if (isVerticalTickLabels()) {
    anchor = TextAnchor.BOTTOM_CENTER;
    rotationAnchor = TextAnchor.BOTTOM_CENTER;
    if (edge == RectangleEdge.LEFT) {
        angle = -Math.PI / 2.0;
    }
    else {
        angle = Math.PI / 2.0;
    }
}
else {
    if (edge == RectangleEdge.LEFT) {
        anchor = TextAnchor.CENTER_RIGHT;
        rotationAnchor = TextAnchor.CENTER_RIGHT;
    }
    else {
        anchor = TextAnchor.CENTER_LEFT;
        rotationAnchor = TextAnchor.CENTER_LEFT;
    }
}
```

refreshTicksVertical

Common part

Variable part

Common part

```
protected List refreshTicksHorizontal(org.jfree.chart.axis.DateAxis dateAxis, RectangleEdge edge) {
    List result = new java.util.ArrayList();
    Font tickLabelFont = getTickLabelFont();
    g2.setFont(tickLabelFont);
    if (isAutoTickUnitSelection()) {
        selectAutoTickUnit(g2, dateAxis, edge);
    }
    Date tickUnit = getTickUnit();
    Date tickDate = calculateNextVisibleTickValue(unit);
    Date upperDate = getMaxVisibleDate();
    boolean hasHole = false;
    while (tickDate.before(upperDate)) {
        // could add a flag to make the following correction optional...
        if (!hasHole) {
            tickDate = correctTickDateForPosition(tickDate, unit, this.tickBarPosition);
        }
        long lowestTickTime = tickDate.getTime();
        long distance = unit.addDate(tickDate, this.timeZone).getTime() - lowestTickTime;
        int minorTickSpaces = getMinorTickCount();
        if (minorTickSpaces == 0) {
            minorTickSpaces = unit.getMinorTickCount();
        }
        for (int minorTick = 1; minorTick <= minorTickSpaces; minorTick++) {
            long minorTickTime = lowestTickTime - distance + minorTick * minorTickSpaces;
            if (minorTickTime >= 0 && getRange().contains(minorTickTime) && (!isHiddenValue(minorTickTime))) {
                result.add(new DateTick(tickType_MINOR, new Date(minorTickTime), "", TextAnchor.TOP_CENTER, TextAnchor.CENTER, 0.0));
            }
        }
        if (!isHiddenValue(tickDate.getTime())) {
            // work out the value, label and position
            String tickLabel;
            DateFormatter formatter = getDateFormatOverride();
            if (formatter != null) {
                tickLabel = formatter.format(tickDate);
            }
            else {
                tickLabel = this.tickUnit.dateToString(tickDate);
            }
            TextAnchor anchor, rotationAnchor;
            double angle = 0.0;
            if (isVerticalTickLabels()) {
                anchor = TextAnchor.CENTER_RIGHT;
                rotationAnchor = TextAnchor.CENTER_RIGHT;
            }
            else {
                anchor = TextAnchor.TOP_CENTER;
                rotationAnchor = TextAnchor.TOP_CENTER;
            }
            Tick tick = new DateTick(tickDate, tickLabel, anchor, rotationAnchor, angle);
            result.add(tick);
            hasHole = false;
            long currentTickTime = tickDate.getTime();
            tickDate = unit.addDate(tickDate, this.timeZone);
            long nextTickTime = tickDate.getTime();
            for (int minorTick = 1; minorTick <= minorTickSpaces; minorTick++) {
                long minorTickTime = currentTickTime + (nextTickTime - currentTickTime) * minorTick / minorTickSpaces;
                if (getRange().contains(minorTickTime) && (!isHiddenValue(minorTickTime))) {
                    result.add(new DateTick(tickType_MINOR, new Date(minorTickTime), "", TextAnchor.TOP_CENTER, TextAnchor.CENTER, 0.0));
                }
            }
        }
        else {
            tickDate = unit.rollDate(tickDate, this.timeZone);
            hasHole = true;
            continue;
        }
    }
    return result;
}
```

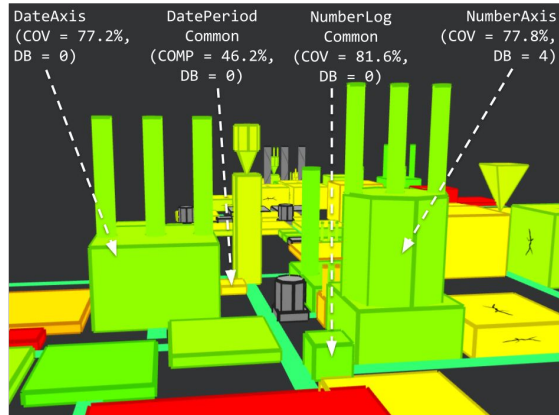
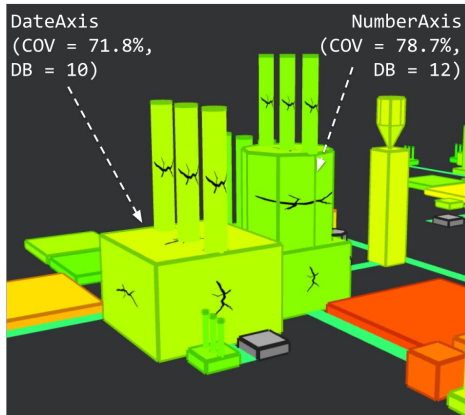
```
protected List refreshTicksVertical(org.jfree.chart.axis.DateAxis dateAxis, RectangleEdge edge) {
    List result = new java.util.ArrayList();
    Font tickLabelFont = getTickLabelFont();
    g2.setFont(tickLabelFont);
    if (isAutoTickUnitSelection()) {
        selectAutoTickUnit(g2, dateAxis, edge);
    }
    Date tickUnit = getTickUnit();
    Date tickDate = calculateNextVisibleTickValue(unit);
    Date upperDate = getMaxVisibleDate();
    boolean hasHole = false;
    while (tickDate.before(upperDate)) {
        // could add a flag to make the following correction optional...
        if (!hasHole) {
            tickDate = correctTickDateForPosition(tickDate, unit, this.tickBarPosition);
        }
        long lowestTickTime = tickDate.getTime();
        long distance = unit.addDate(tickDate, this.timeZone).getTime() - lowestTickTime;
        int minorTickSpaces = getMinorTickCount();
        if (minorTickSpaces == 0) {
            minorTickSpaces = unit.getMinorTickCount();
        }
        for (int minorTick = 1; minorTick <= minorTickSpaces; minorTick++) {
            long minorTickTime = lowestTickTime - distance + minorTick * minorTickSpaces;
            if (minorTickTime >= 0 && getRange().contains(minorTickTime) && (!isHiddenValue(minorTickTime))) {
                result.add(new DateTick(tickType_MINOR, new Date(minorTickTime), "", TextAnchor.TOP_CENTER, TextAnchor.CENTER, 0.0));
            }
        }
        if (!isHiddenValue(tickDate.getTime())) {
            // work out the value, label and position
            String tickLabel;
            DateFormatter formatter = getDateFormatOverride();
            if (formatter != null) {
                tickLabel = formatter.format(tickDate);
            }
            else {
                tickLabel = this.tickUnit.dateToString(tickDate);
            }
            TextAnchor anchor, rotationAnchor;
            double angle = 0.0;
            if (isVerticalTickLabels()) {
                anchor = TextAnchor.BOTTOM_CENTER;
                rotationAnchor = TextAnchor.BOTTOM_CENTER;
            }
            else {
                anchor = TextAnchor.LEFT;
                rotationAnchor = TextAnchor.CENTER_LEFT;
            }
            Tick tick = new DateTick(tickDate, tickLabel, anchor, rotationAnchor, angle);
            result.add(tick);
            hasHole = false;
            long currentTickTime = tickDate.getTime();
            tickDate = unit.addDate(tickDate, this.timeZone);
            long nextTickTime = tickDate.getTime();
            for (int minorTick = 1; minorTick <= minorTickSpaces; minorTick++) {
                long minorTickTime = currentTickTime + (nextTickTime - currentTickTime) * minorTick / minorTickSpaces;
                if (getRange().contains(minorTickTime) && (!isHiddenValue(minorTickTime))) {
                    result.add(new DateTick(tickType_MINOR, new Date(minorTickTime), "", TextAnchor.TOP_CENTER, TextAnchor.CENTER, 0.0));
                }
            }
        }
        else {
            tickDate = unit.rollDate(tickDate, this.timeZone);
            hasHole = true;
            continue;
        }
    }
    return result;
}
```

Impact of fixing on the visualization

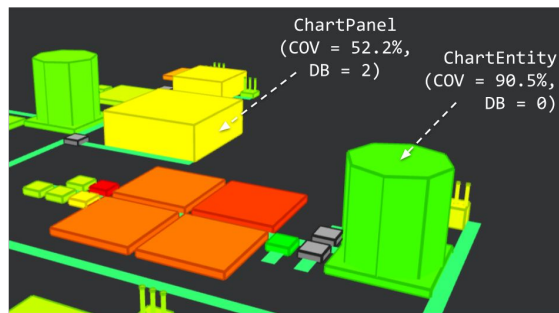
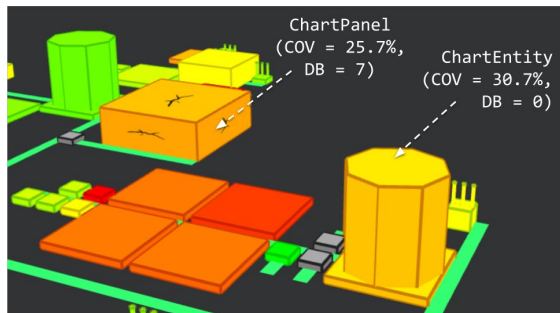
Before

After

Classes with duplicated blocks



Classes lacking tests



Impact of fixing on the classes metrics

Improvements of the quality metrics

- increased coverage
- decreased duplicated blocks

Class (in the org.jfree.chart package)		Coverage	Complexity
entity. ChartEntity	before	30.7 %	26
	AFTER	90.5 %	26
ChartPanel	before	25.7 %	322
	AFTER	52.2 %	295

Class (in the org.jfree.chart.axis package)		# duplicated blocks	Complexity
DateAxis	before	10	201
	AFTER	0	139
PeriodAxis	before	2	112
	AFTER	1	104
DatePeriodCommon	AFTER	0	8

Impact of fixing on the classes metrics

Improvements of the quality metrics

- increased coverage
- decreased duplicated blocks

Modifications also led to a **cognitive complexity improvement**

Class (in the org.jfree.chart package)		Coverage	Complexity
entity. ChartEntity	before	30.7 %	26
	AFTER	90.5 %	26
ChartPanel	before	25.7 %	322
	AFTER	52.2 %	295

Class (in the org.jfree.chart.axis package)		# duplicated blocks	Complexity
DateAxis	before	10	201
	AFTER	0	139
PeriodAxis	before	2	112
	AFTER	1	104
DatePeriodCommon	AFTER	0	8

Impact of fixing on the classes metrics

Improvements of the quality metrics

- increased coverage
- decreased duplicated blocks

Modifications also led to a **cognitive complexity improvement**

Added classes are not critical

Class (in the org.jfree.chart package)		Coverage	Complexity
entity. ChartEntity	before	30.7 %	26
	AFTER	90.5 %	26
ChartPanel	before	25.7 %	322
	AFTER	52.2 %	295

Class (in the org.jfree.chart.axis package)		# duplicated blocks	Complexity
DateAxis	before	10	201
	AFTER	0	139
PeriodAxis	before	2	112
	AFTER	1	104
DatePeriodCommon	AFTER	0	8

Future work

- Conducting an **empirical evaluation with real users** would help us in **validating our evaluation** and **identifying understandability limitations** of our visualization
- Explore deeper the relation between code smells and variability implementations (e.g. code duplication)



⇒ **better understand how VariMetrics could be extended to match the industry's needs and expectations**

Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations

Johann Mortara — Philippe Collet — Anne-Marie Dery-Pinna

OO variability implementations induce **technical debt** that **needs to be identified** as it **hampers the system's quality**

VariMetrics displays **OO quality metrics on VariCity**, a city visualization for OO variability implementations

The visualization exhibits **indebted zones of variability implementations**

Reproduction package:

<https://doi.org/10.5281/zenodo.6644633>

Obtained reproducibility badges

Artifacts Available



Functional & Reusable



Get the paper:

<https://hal.archives-ouvertes.fr/hal-03717858/>

VariMetrics website:

<https://deathstar3.github.io/varimetrics-demo/>

See you at the tool demo at 5:20 PM!