

Visualization of Object-Oriented Variability Implementations as Cities



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

Université Côte d'Azur, CNRS, I3S, France

VISSOFT '21

September 27, 2021

Highly-variable Systems with a Single Code Base



16.000 options managed
in 25M LoC

#ifdef



ANDROID

24.000 different platforms in
2015

Object-orientation



2.000+ options generating variants for
platforms, security levels...

#ifdef / object-orientation

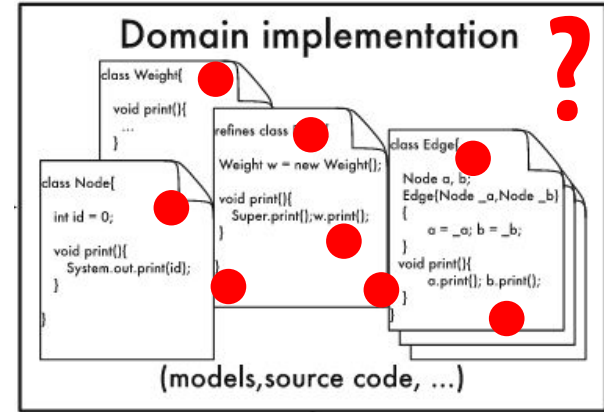
and multiple management techniques...

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

Undocumented
OO variability
implementations



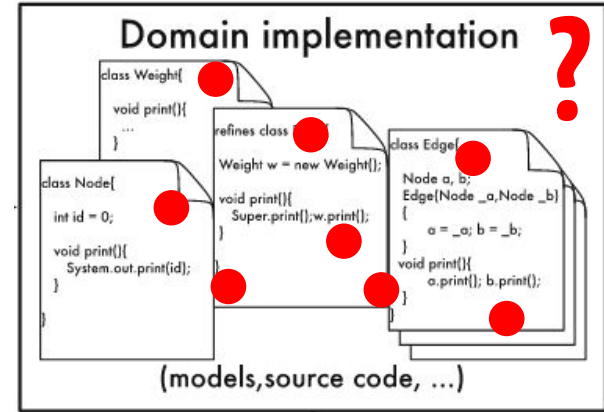
OO codebases use OO mechanisms to implement variability in a single codebase

- 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**



Problem: How to identify and comprehend object-oriented variability implementations?

Variation points and variants

```
1 | public abstract class Shape {
2 |     public abstract double area();
3 |     public abstract double perimeter(); /*...*/
4 | }
5 |
6 | public class Circle extends Shape {
7 |     private final double radius;
8 |     // Constructor omitted
9 |     public double area() {
10 |         return Math.PI * Math.pow(radius, 2);
11 |     }
12 |     public double perimeter() {
13 |         return 2 * Math.PI * radius;
14 |     }
15 | }
```

```
15 | public class Rectangle extends Shape {
16 |     private final double width, length;
17 |     // Constructor omitted
18 |     public double area() {
19 |         return width * length;
20 |     }
21 |     public double perimeter() {
22 |         return 2 * (width + length);
23 |     }
24 |     public void draw(int x, int y) {
25 |         // rectangle at (x, y, width, length)
26 |     }
27 |     public void draw(Point p) {
28 |         // rectangle at (p.x, p.y, width, length)
29 |     }
30 | }
```

Variation points and variants

```
1 | public abstract class Shape {
2 |     public abstract double area();
3 |     public abstract double perimeter(); /*...*/
4 | }
```

vp_Shape

```
5 | public class Circle extends Shape {
6 |     private final double radius;
7 |     // Constructor omitted
8 |     public double area() {
9 |         return Math.PI * Math.pow(radius, 2);
10 |    }
11 |    public double perimeter() {
12 |        return 2 * Math.PI * radius;
13 |    }
14 | }
```

v_Circle

```
15 | public class Rectangle extends Shape {
16 |     private final double width, length;
17 |     // Constructor omitted
18 |     public double area() {
19 |         return width * length;
20 |    }
21 |     public double perimeter() {
22 |         return 2 * (width + length);
23 |    }
24 |     public void draw(int x, int y) {
25 |         // rectangle at (x, y, width, length)
26 |    }
27 |     public void draw(Point p) {
28 |         // rectangle at (p.x, p.y, width, length)
29 |    }
30 | }
```

v_Rectangle

Variation points and variants

```
1 | public abstract class Shape {
2 |     public abstract double area();
3 |     public abstract double perimeter(); /*...*/
4 | }
```

vp_Shape

```
5 | public class Circle extends Shape {
6 |     private final double radius;
7 |     // Constructor omitted
8 |     public double area() {
9 |         return Math.PI * Math.pow(radius, 2);
10 |    }
11 |    public double perimeter() {
12 |        return 2 * Math.PI * radius;
13 |    }
14 | }
```

v_Circle

```
15 | public class Rectangle extends Shape {
16 |     private final double width, length;
17 |     // Constructor omitted
18 |     public double area() {
19 |         return width * length;
20 |     }
21 |     public double perimeter() {
22 |         return 2 * (width + length);
23 |     }
```

v_Rectangle

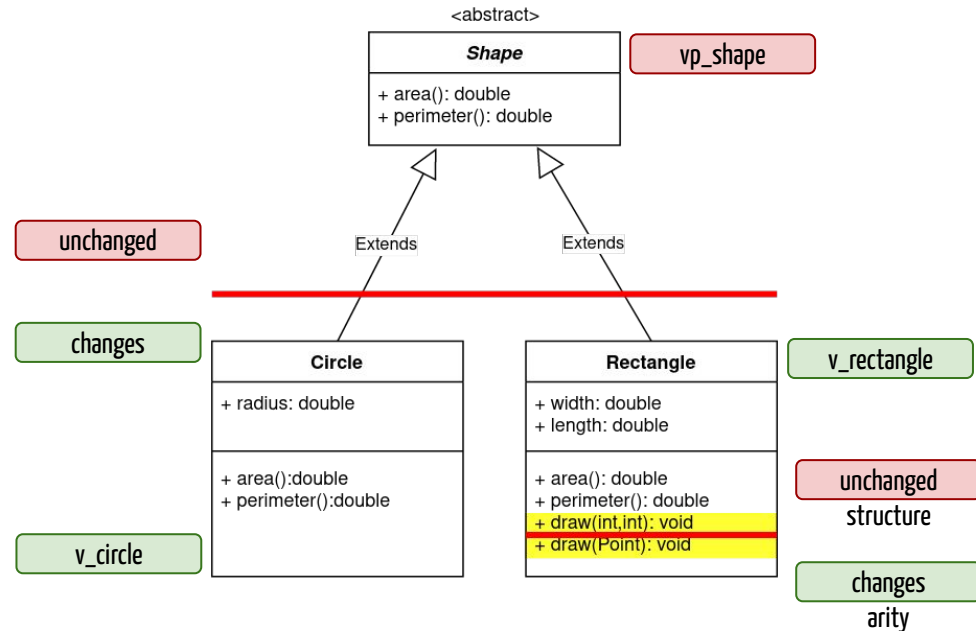
```
24 | public void draw(int x, int y) {
25 |     // rectangle at (x, y, width, length)
26 | }
27 | public void draw(Point p) {
28 |     // rectangle at (p.x, p.y, width, length)
29 | }
30 | }
```

vp_draw

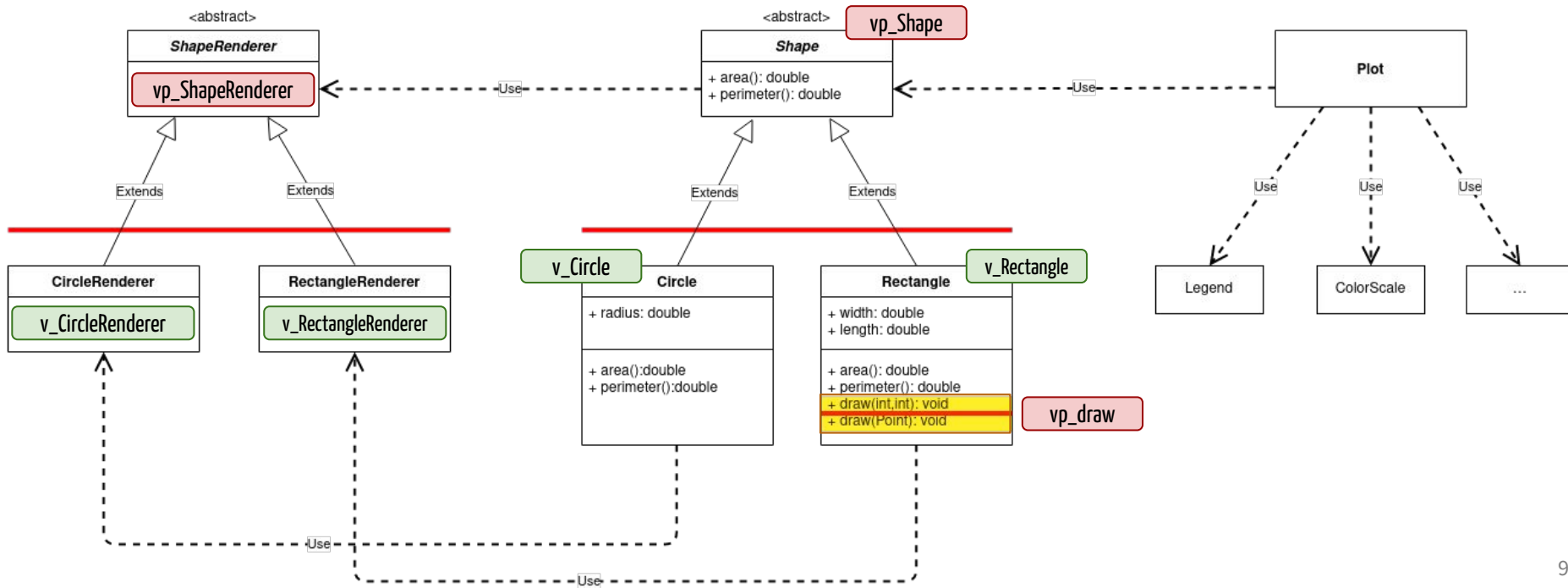
Identifying OO variability implementations with symmetries

- **Symmetries** exist in each OO mechanism (Coplien and Zhao's work)
- Symmetries present in **mechanisms implementing variability**

High density of symmetries
⇒ **high density of variability**

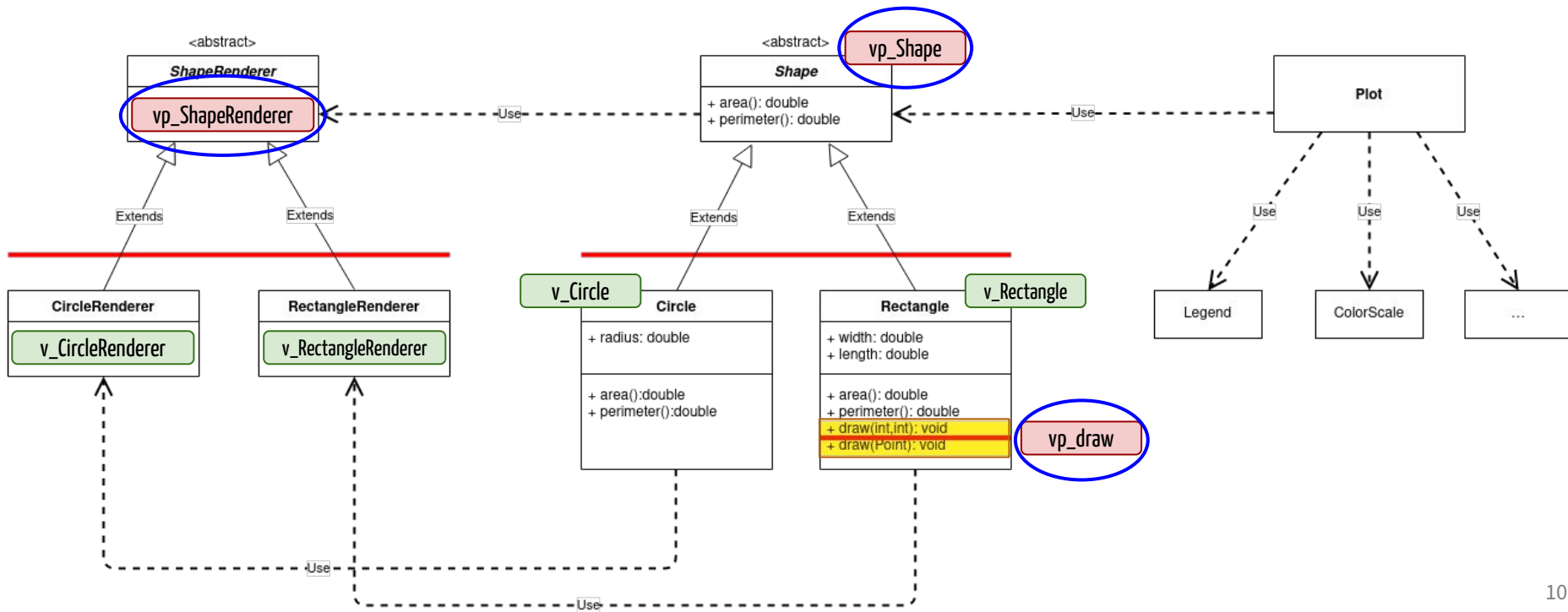


Density of symmetries



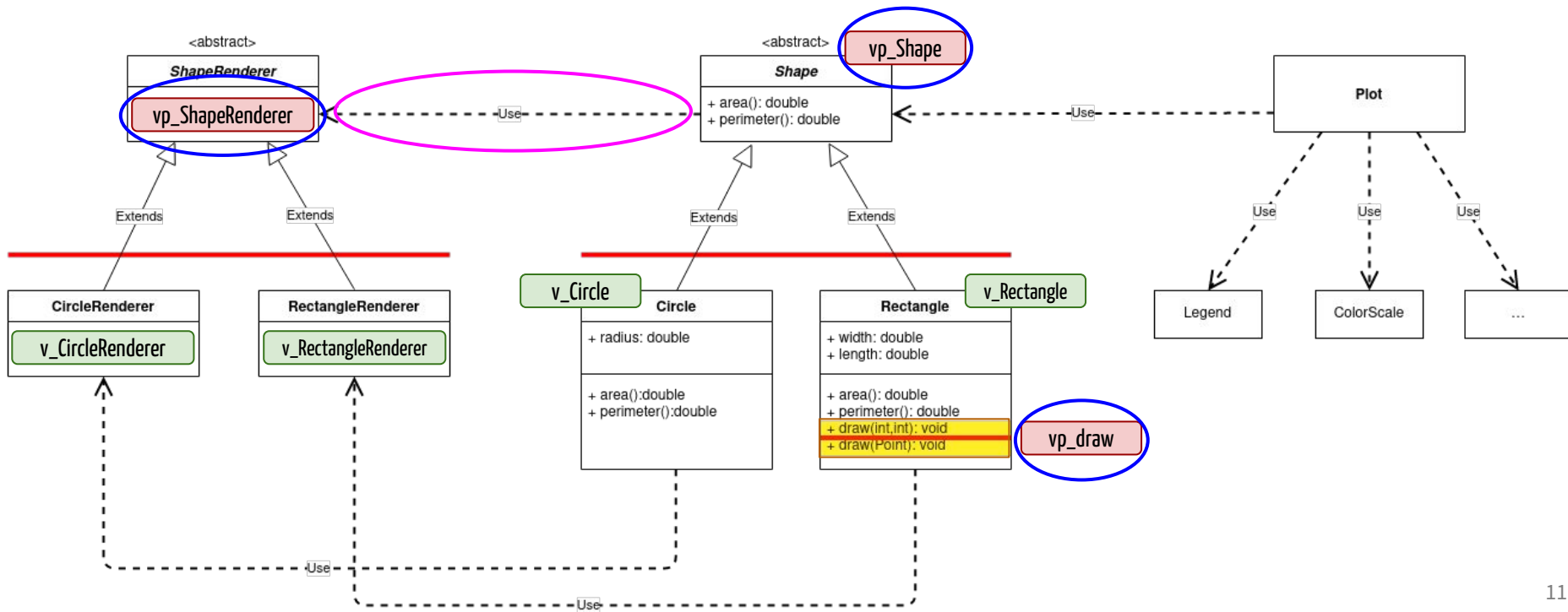
Density of symmetries

vp (class or method level) with important number of variants



Density of symmetries

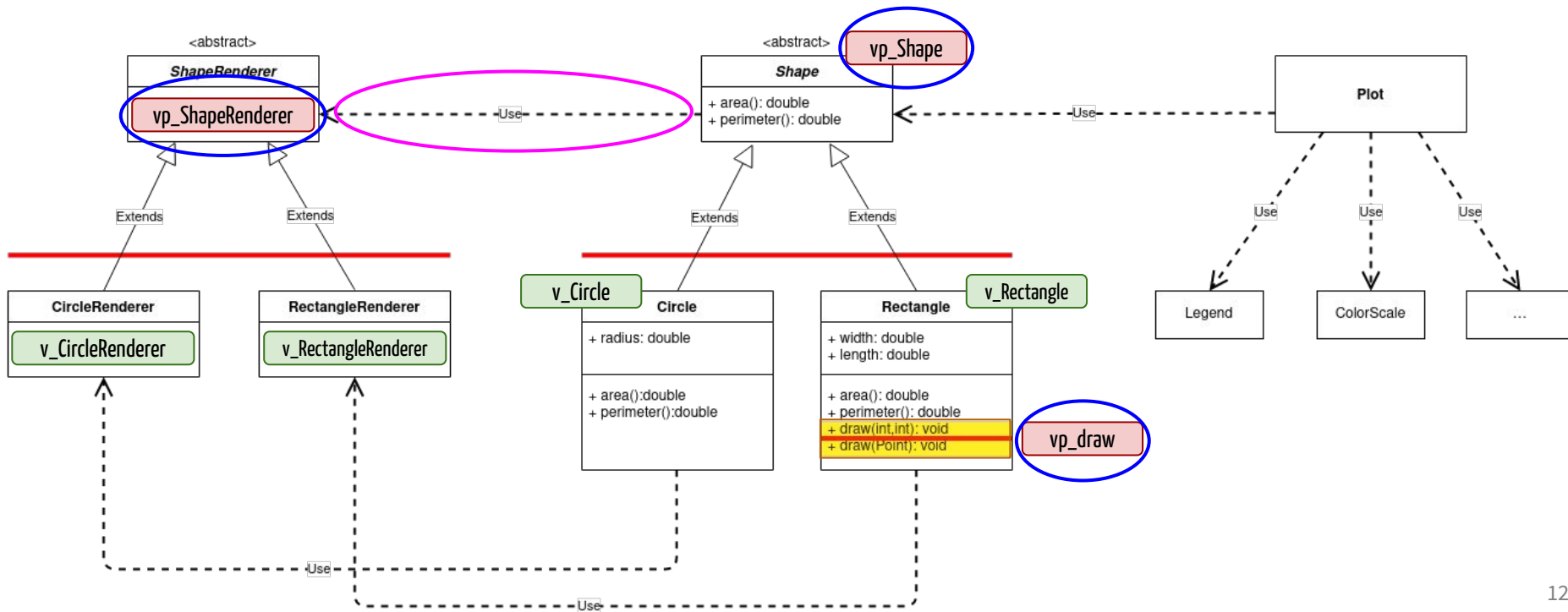
vp (class or method level) with important number of variants
vp-s using each other



Density of symmetries

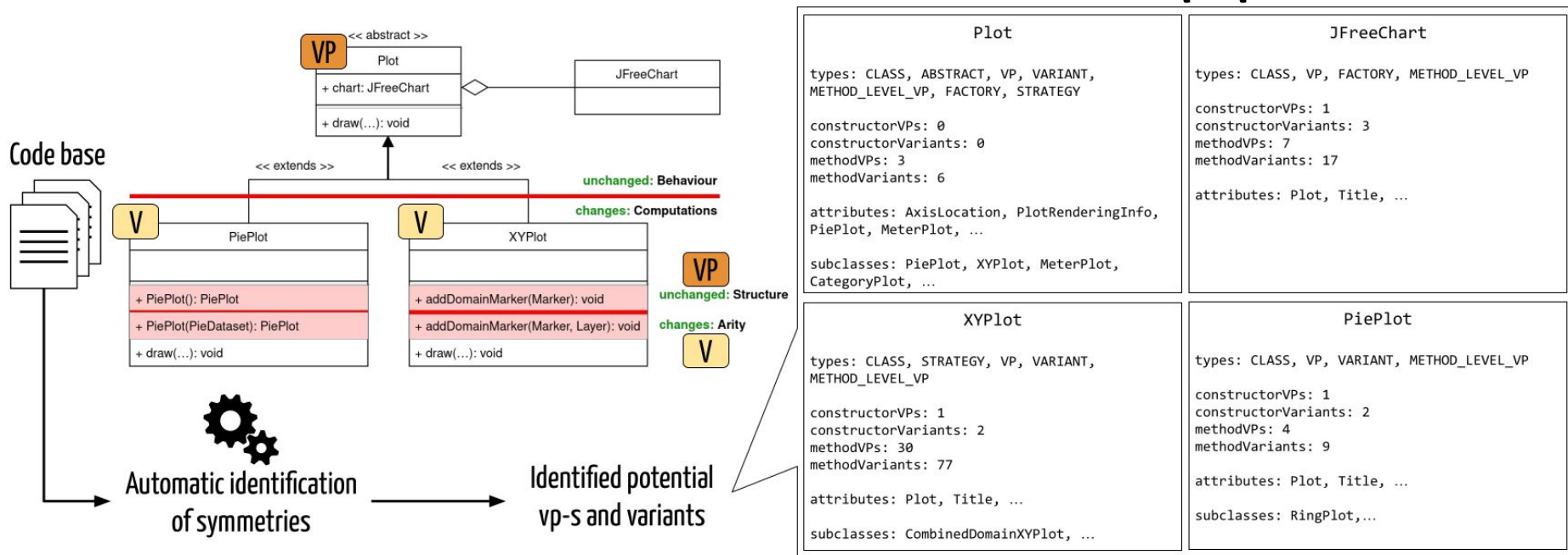
vp (class or method level) with important number of variants
vp-s using each other

HOTSPOTS



Automatic identification of variability implementations in an OO codebase

metrics / properties



Finding an appropriate visualization

Goal: help the comprehension of variability intense zones in a large codebase

Finding an appropriate visualization

Goal: help the comprehension of variability intense zones in a large codebase

Requirement 1

the visualization must **display metrics on classes** and **relationships between them**, exhibiting the density of variability implementations

Finding an appropriate visualization

Goal: help the comprehension of variability intense zones in a large codebase

Requirement 1

the visualization must **display metrics on classes** and **relationships between them**, exhibiting the density of variability implementations

Requirement 2

the visualization must **scale on large systems**

Finding an appropriate visualization

Goal: help the comprehension of variability intense zones in a large codebase

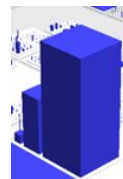
Requirement 1

the visualization must **display metrics on classes** and **relationships between them**, exhibiting the density of variability implementations

Requirement 2

the visualization must **scale on large systems**

The city metaphor



Finding an appropriate visualization

Goal: help the comprehension of variability intense zones in a large codebase

Requirement 1

the visualization must **display metrics on classes** and **relationships between them**, exhibiting the density of **variability implementations**

Requirement 2

the visualization must **scale on large systems**

The city metaphor

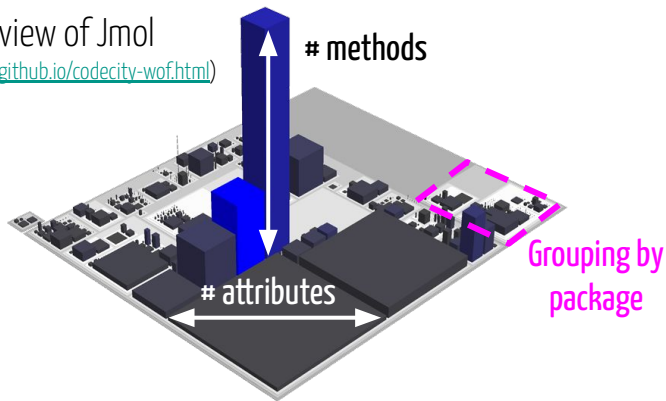


adapted for variability implementations

From CodeCity and Evo-Streets to VariCity

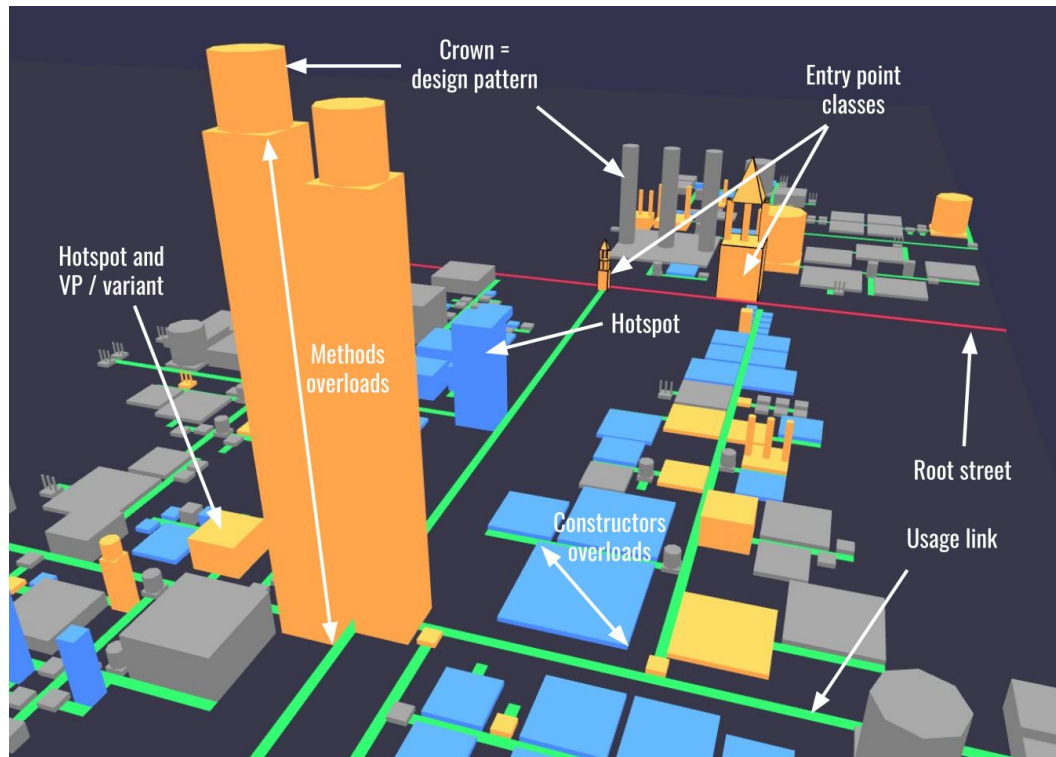
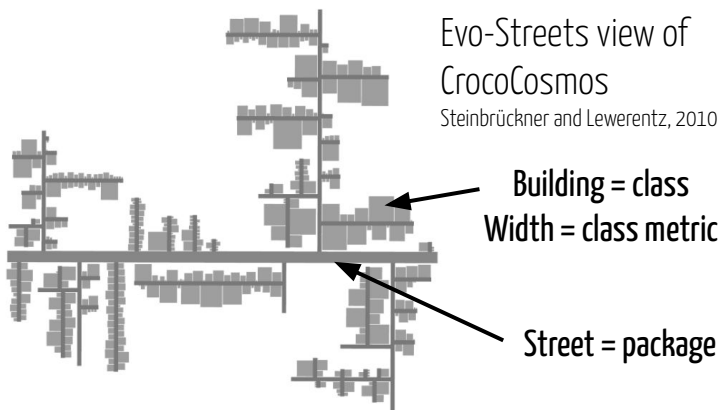
CodeCity view of Jmol

(<https://wettel.github.io/codecity-wof.html>)



Evo-Streets view of
CrocoCosmos

Steinbrückner and Lewerentz, 2010

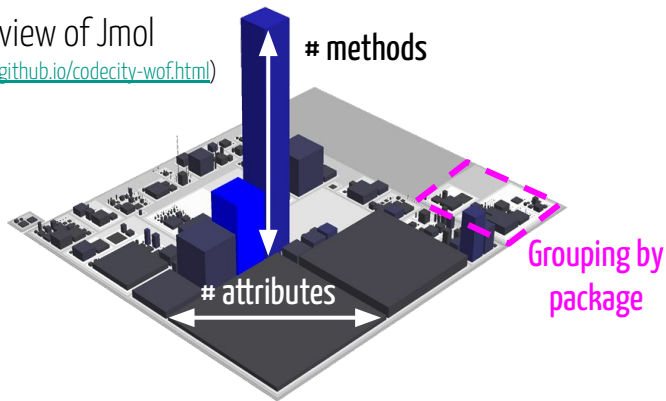


VariCity view of JFreeChart

From CodeCity and Evo-Streets to VariCity

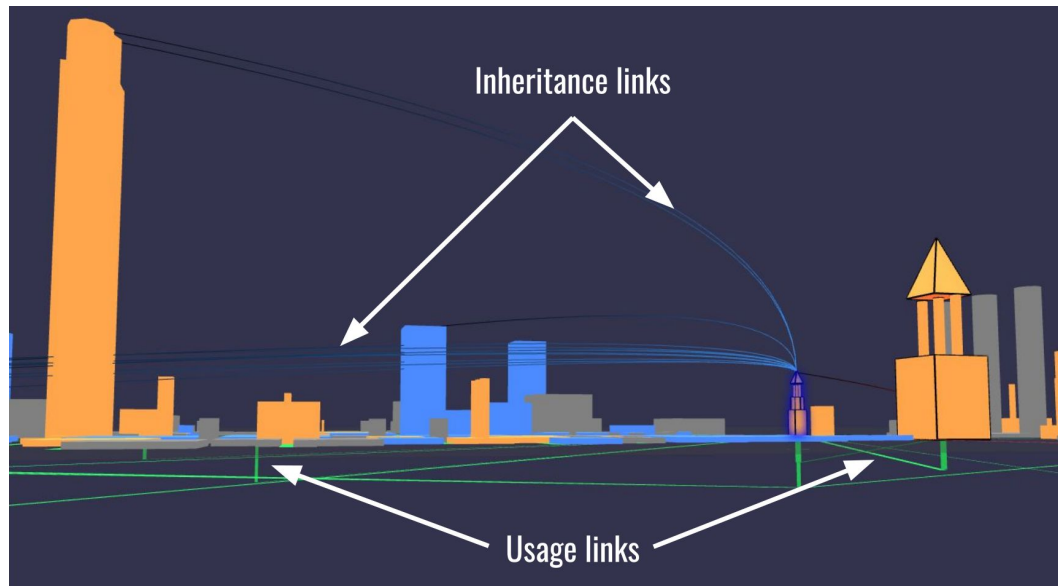
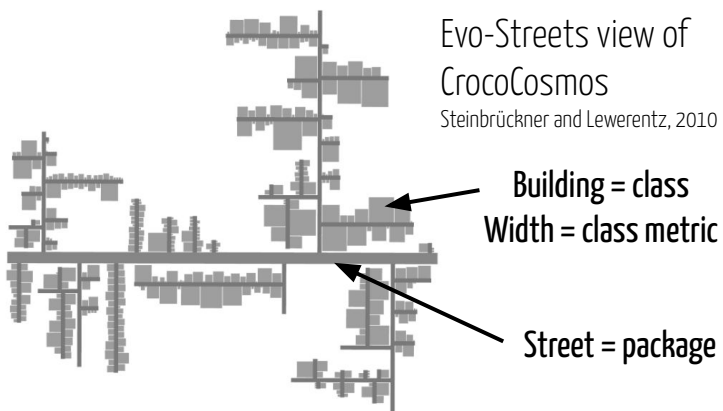
CodeCity view of Jmol

(<https://wettel.github.io/codecity-wof.html>)



Evo-Streets view of
CrocoCosmos

Steinbrückner and Lewerentz, 2010



Additional links for inheritance

Interaction capabilities

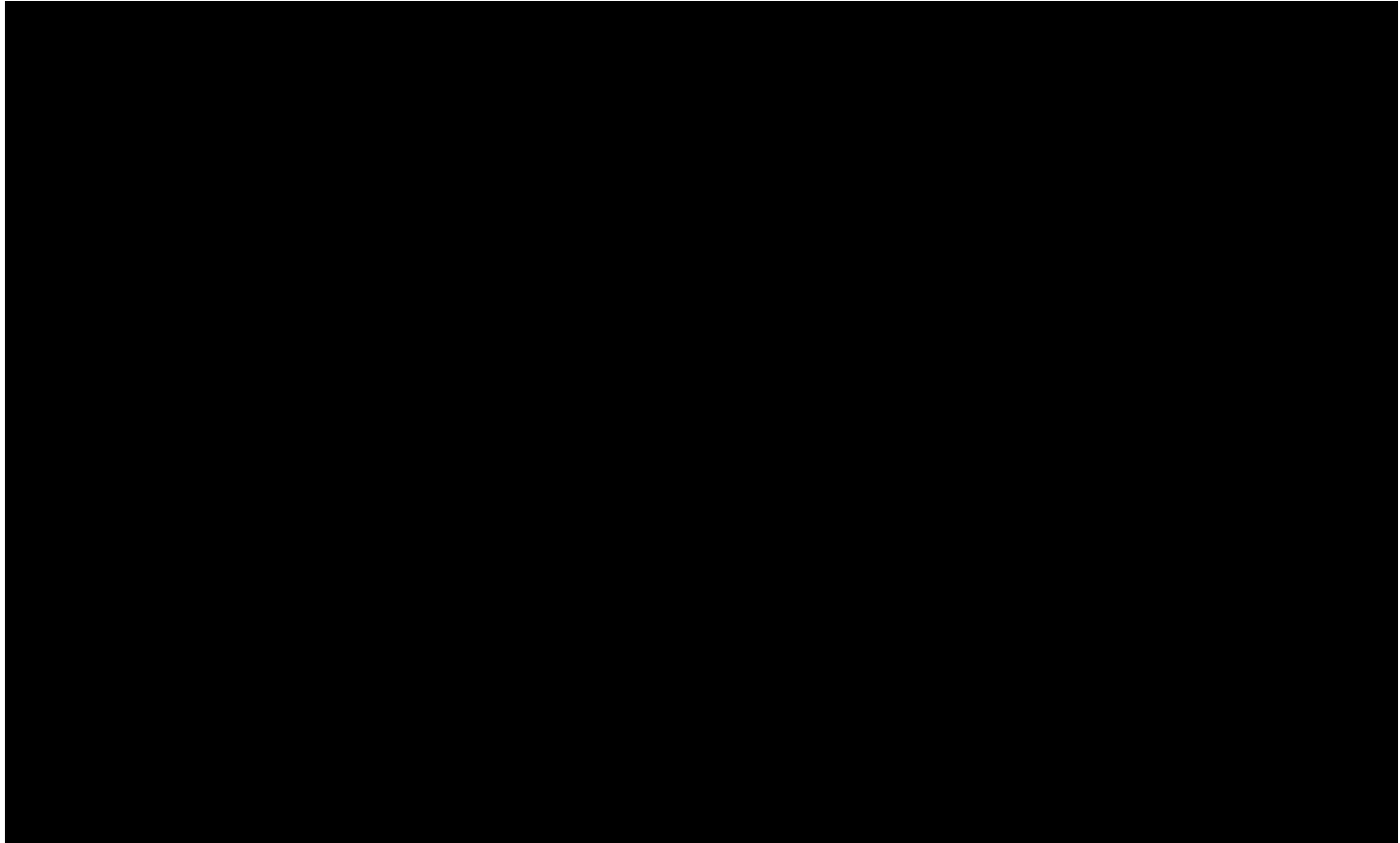
Zooming, spanning

Hovering buildings to
display additional links

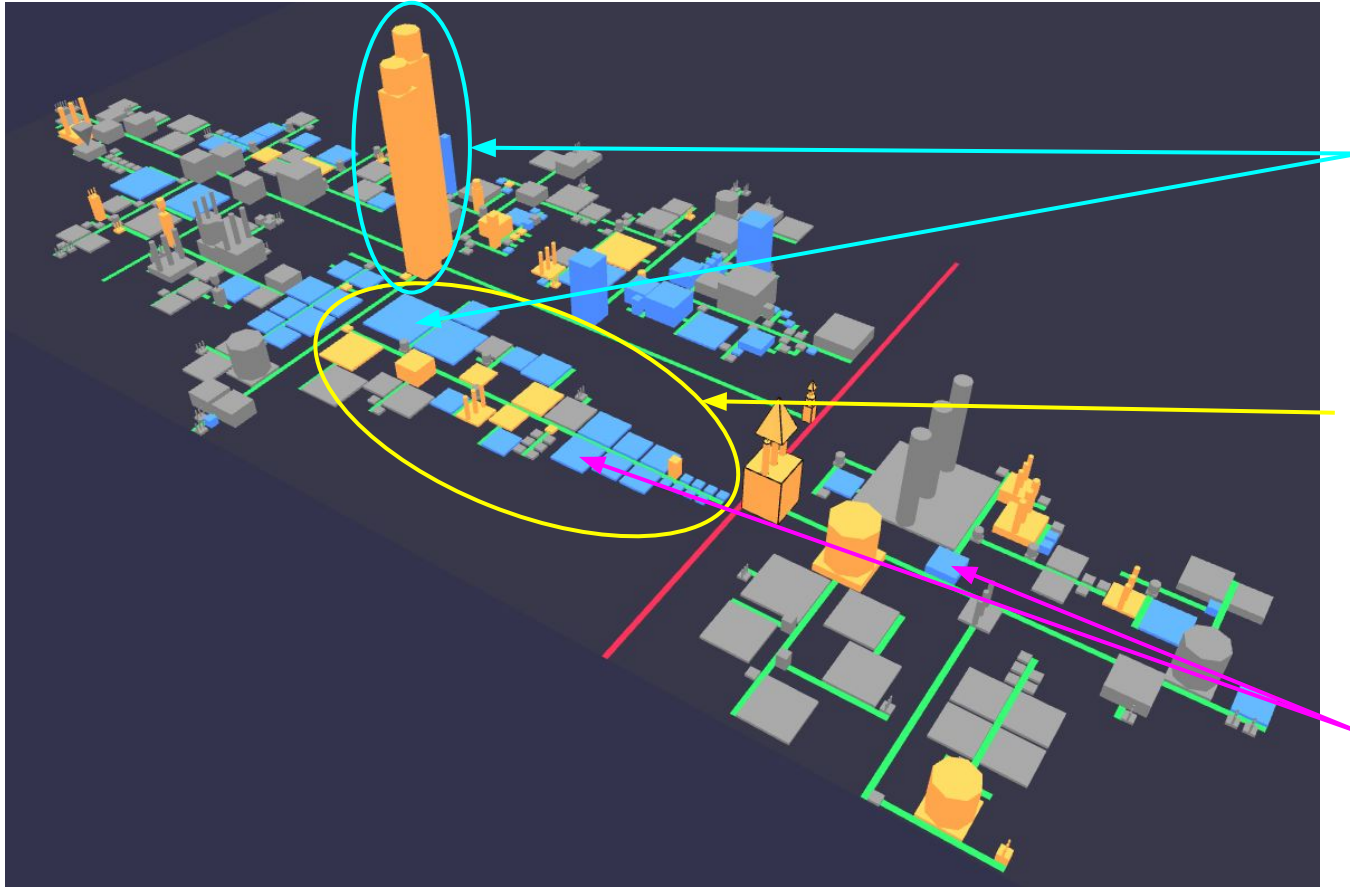
Increasing / decreasing
the usage level

Orientation of the usage
relationships

Adapting the entrypoints



Exhibiting density of variability implementations



**Tall and / or large buildings =
density at method level**

**Large neighbourhoods =
density by usage relationships**

**Hotspots maximize both
density types**

Evaluation

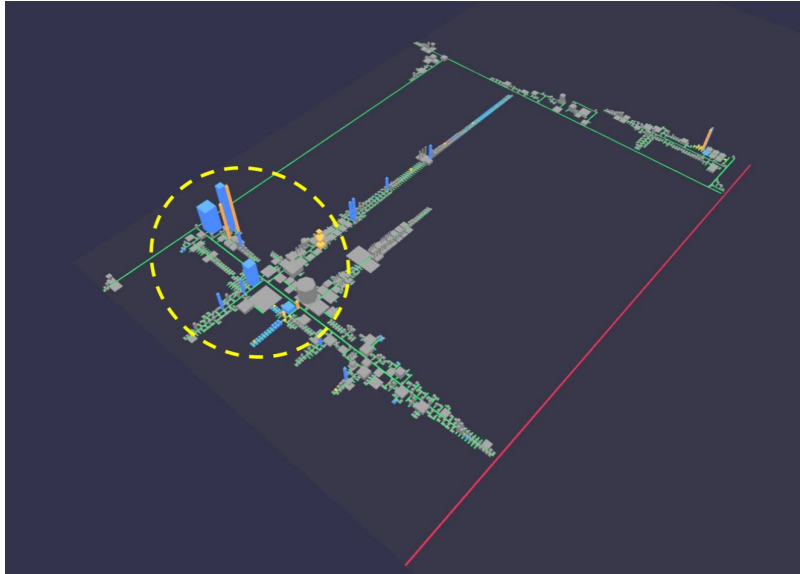
Variability implementations are **complex zones** in the code that **newcomers onboarding on a project seek to understand** [1]

Two types of users are part of onboarding scenarios:

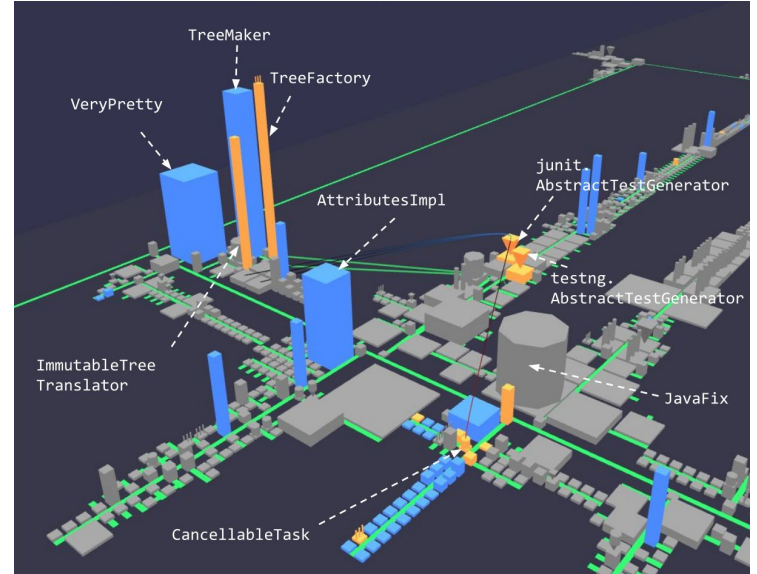
- a **newcomer** is **onboarded on a project** and has to grasp its important parts
- an **expert** has a **deep knowledge of the codebase**, and helps the newcomer to discover it

[1] R. Yates, N. Power, and J. Buckley, “Characterizing the transfer of program comprehension in onboarding: an information-push perspective,” *Empirical Software Engineering*, vol. 25, no. 1, pp. 940–995, 2020.

Scenario 1: An expert wants to **facilitate the exploration** of the codebase by giving a pre-configured visualization to the newcomer.

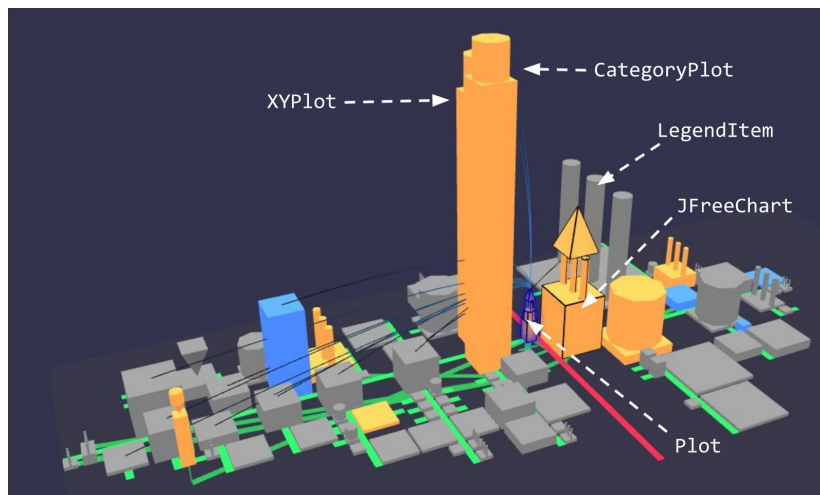


Preconfigured view of NetBeans, neighbourhood of tall and blue buildings detaches

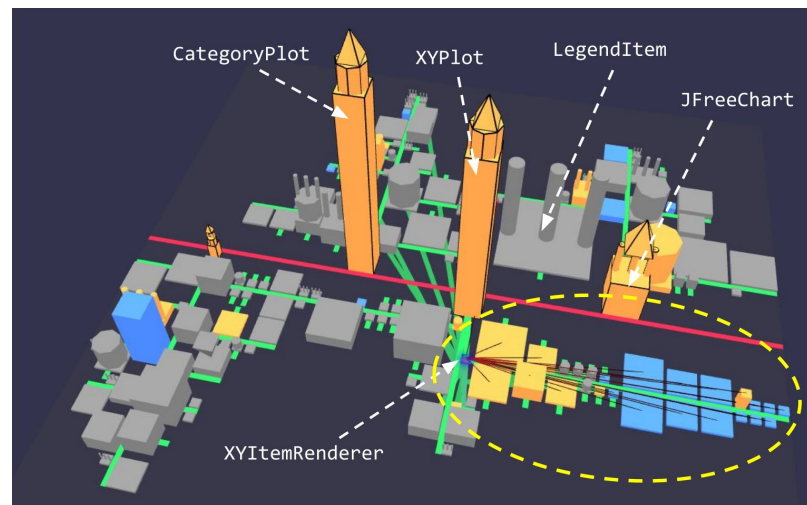


Zooming and spanning allow to explore at finer-grain the city

Scenario 2: The expert wants the newcomer to **comprehend a subpart** of the codebase for the newcomer to be able to reuse it.



Preconfigured view of JFreeChart with Plot as entrypoint. Displaying links of Plot reveals that XYPlot and CategoryPlot are subclasses.



Adding XYPlot and CategoryPlot as entrypoints allows to display other buildings forming a variability intense neighbourhood.

Future work

Real experts evaluation

Integration in an IDE

Add other metrics of code quality



⇒ gain new insights on how to better facilitate the identification of variability implementations

Visualization of Object-Oriented Variability Implementations as Cities

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

OO variability implementations are
complex to identify and comprehend

VariCity provides a **visualization**
relying on the **city metaphor** of OO
variability implementations

Visualization **exhibits zones of**
high density of variability, in
classes and between classes

Reproduction package:

<https://doi.org/10.5281/zenodo.5034199>

Obtained reproducibility badges

Open Research Objects

Research Objects Reviewed



Get the paper:

<https://hal.archives-ouvertes.fr/hal-03312487>

VariCity website:

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

See you at the tool demo tomorrow!