On the Impact of Feature Dependencies when Maintaining Preprocessor-based Software Product Lines

Márcio Ribeiro Felipe Queiroz Paulo Borba

Társis Tolêdo Claus Brabrand Sérgio Soares

{mmr3, fbq, phmb, twt, scbs}@cin.ufpe.br

brabrand@itu.dk

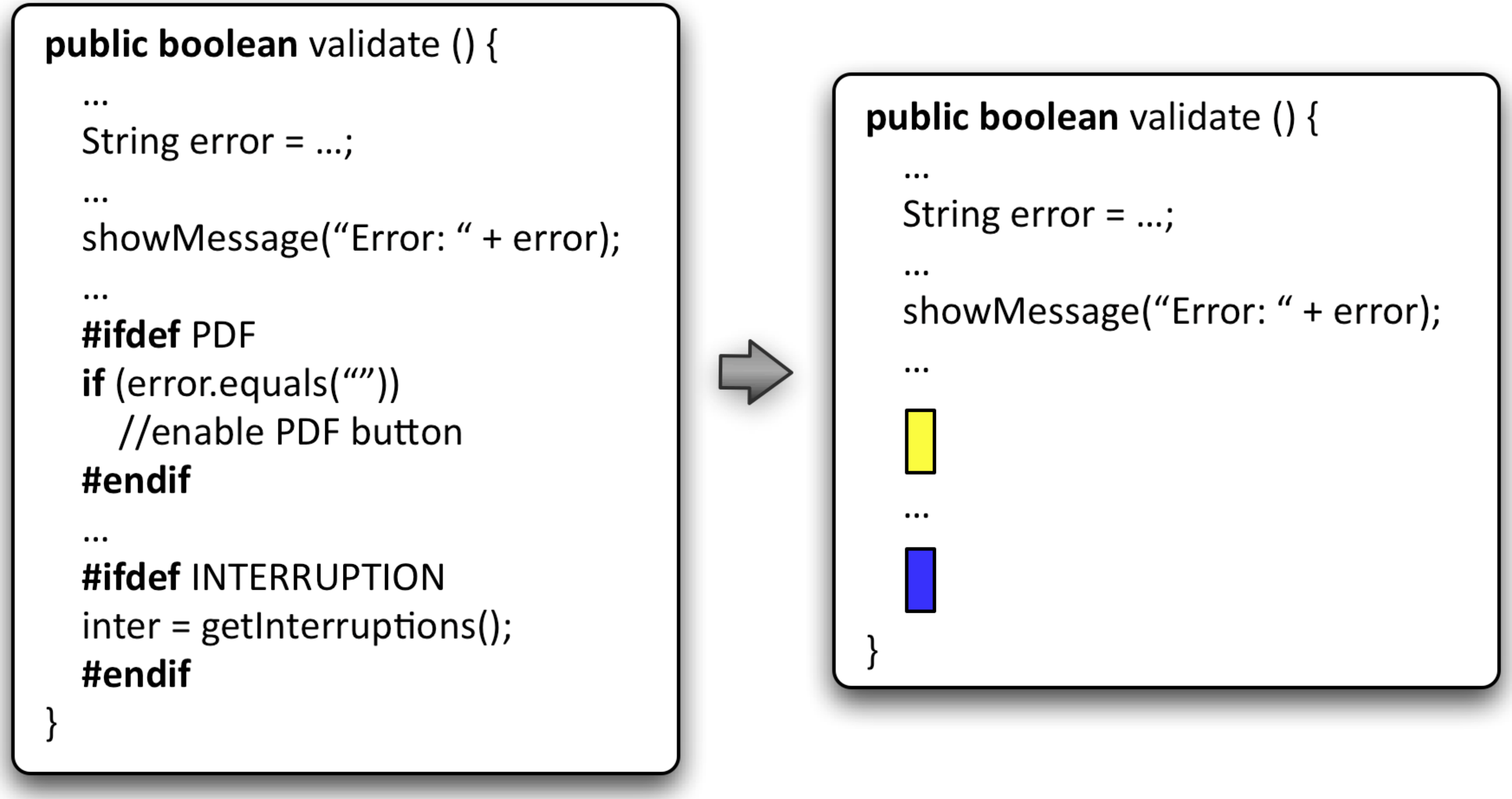
Centro de Informática U·F·P·E

IT University of Copenhagen

SPG SOFTWARE·PRODUCTIVITY·GROUP

ines

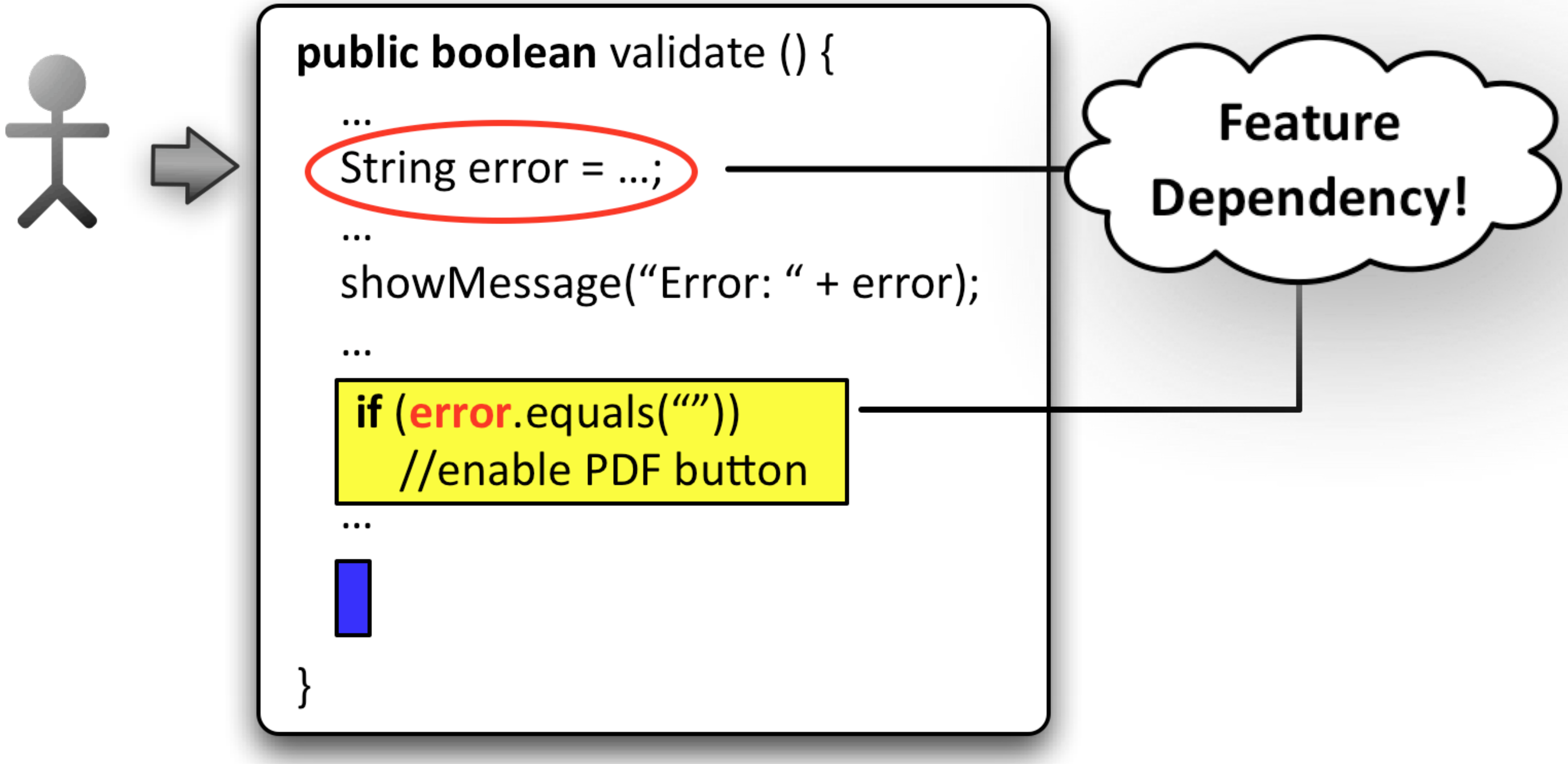
Virtual Separation of Concerns (VSoC)



```
public boolean validate () {
   …
   String error = …;
   …
   showMessage(“Error: “ + error);
   …
   #ifdef PDF
   if (error.equals(“”))
      //enable PDF button
   #endif
   …
   #ifdef INTERRUPTION
   inter = getInterruptions();
   #endif
}
```

```
public boolean validate () {
   …
   String error = …;
   …
   showMessage(“Error: “ + error);
   …
   🟨
   …
   🟦
}
```

But not enough for Feature Modularity…

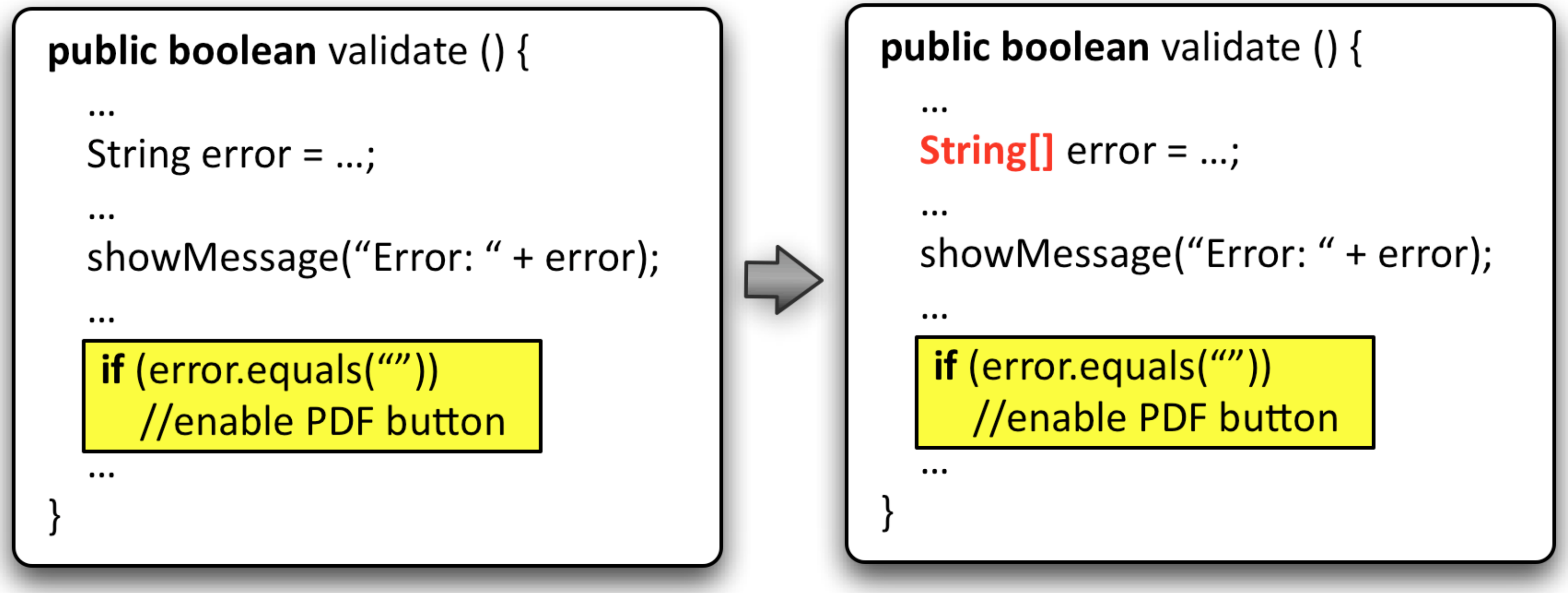


```
public boolean validate () {
    …
    String error = …;
    …
    showMessage("Error: " + error);
    …
    if (error.equals(""))
        //enable PDF button
    …

}
```

Problem

- Developers can introduce compilation and behavioral **errors** to features due to feature dependencies…



```
public boolean validate () {
    …
    String error = …;
    …
    showMessage("Error: " + error);
    …
    if (error.equals(""))
        //enable PDF button
    …
}
```

```
public boolean validate () {
    …
    String[] error = …;
    …
    showMessage("Error: " + error);
    …
    if (error.equals(""))
        //enable PDF button
    …
}
```

… leading to lower productivity

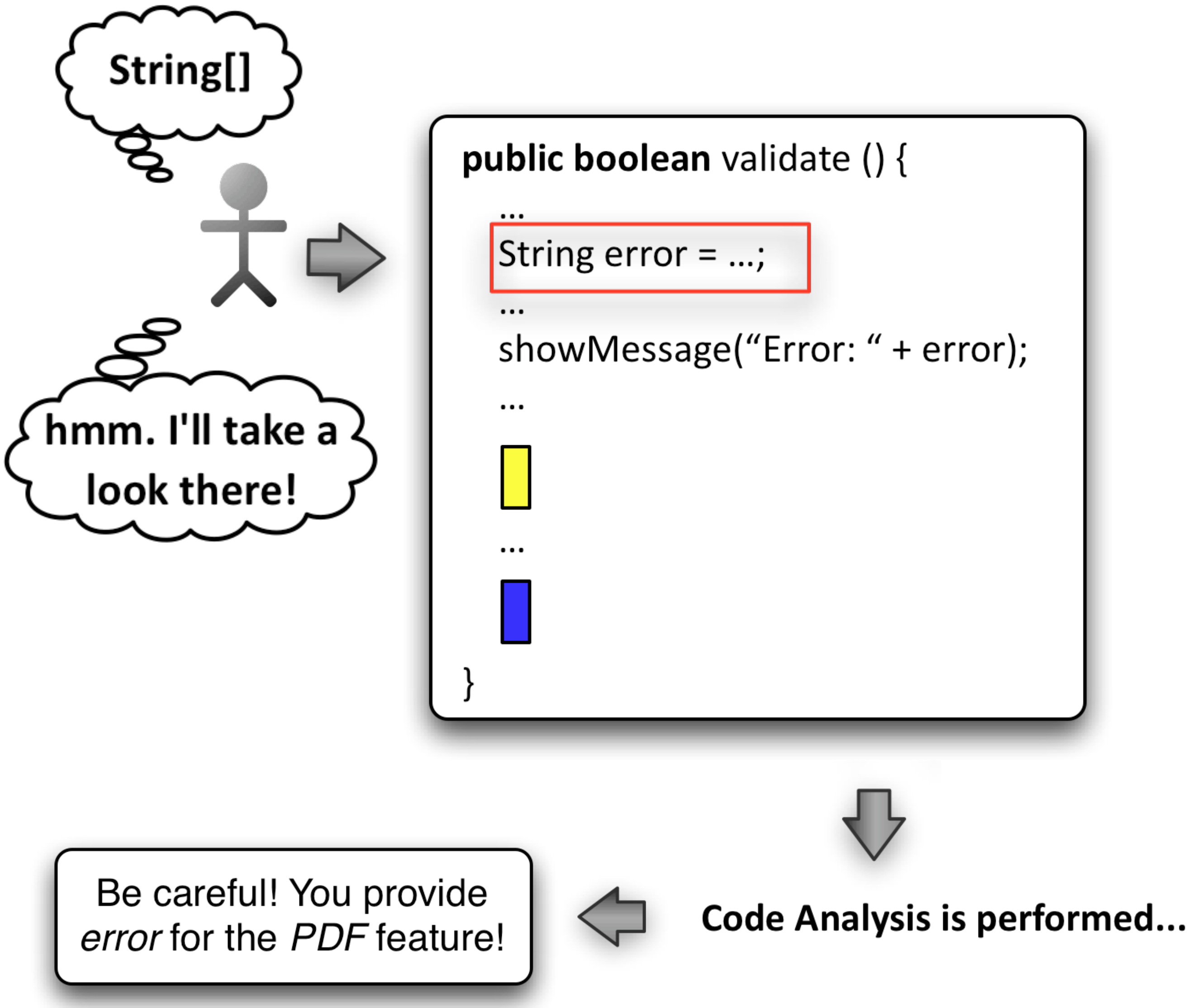
- **Late error detection**
  - Developers discover the problem only when compiling and executing the problematic product
- **Difficult navigation**
  - Developers do not know where are feature dependencies
  - Even worse when using VSoC: features are **hidden**!

```
public boolean validate () {
   ...
   String error = ...;
   showMessage("Error: " + error);
   ...
   ...
   ...
   ...
   ...
}
```

To minimize these problems:

Emergent Interfaces

Emergent Interfaces in a nutshell

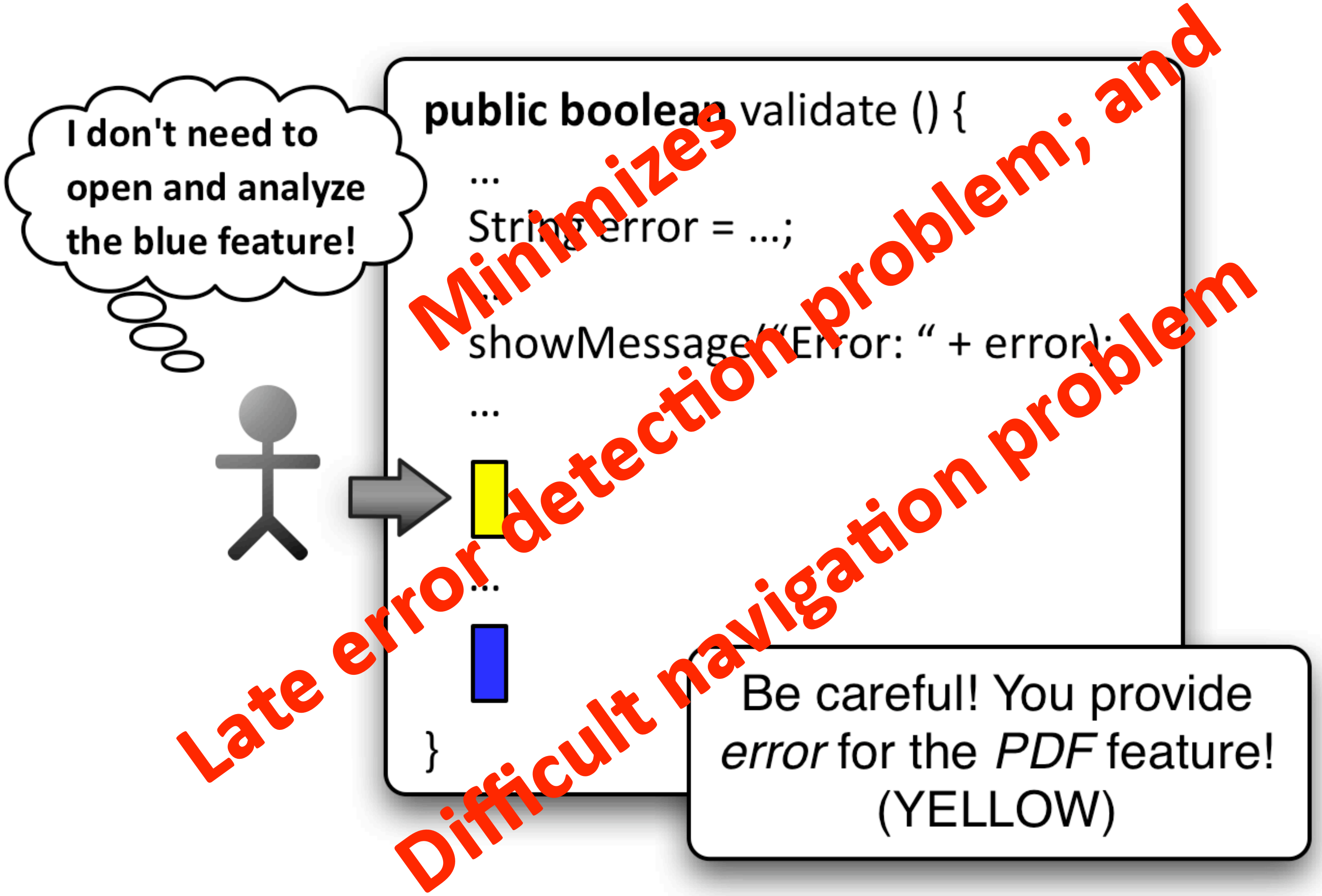


```
public boolean validate () {
   …
   String error = …;
   …
   showMessage("Error: " + error);
   …
   🟨
   …
   🟦
}
```

Code Analysis is performed…

Be careful! You provide *error* for the *PDF* feature!

Dependencies captured! Developers can focus on the impacted features…



```
public boolean validate () {
  ...
  String error = ...;
  ...
  showMessage(“Error: “ + error);
  ...
  🟨
  ...
  🟦
}
```

Be careful! You provide *error* for the *PDF* feature! (YELLOW)

Minimizes problem; and

Late error detection problem

Difficult navigation problem

Now, we know that feature dependencies can cause problems

Also, we know that Emergent Interfaces complement VSoC

Agenda

Question 1: how often methods with preprocessors directives contain feature dependencies?

Why is this question important?
To assess to what extent dependencies is a problem in practice

Question 2: how feature dependencies impact maintenance effort when using VSoC and Emergent Interfaces?

Why is this question important?
To better understand to what extent emergent interfaces complement VSoC

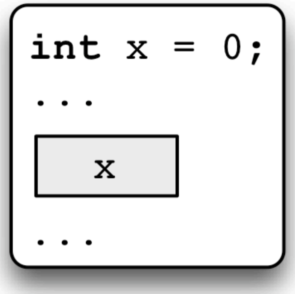
Study settings

Study settings

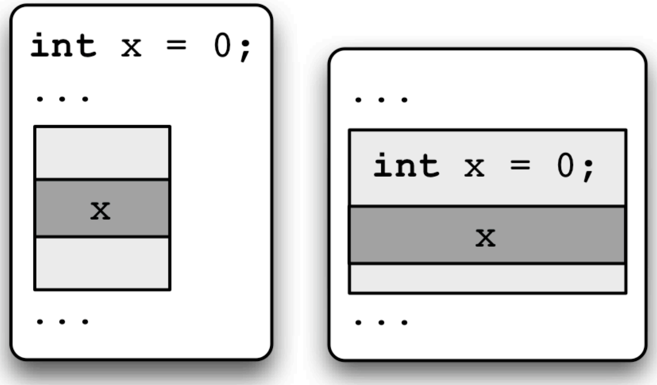
- 43 Preprocessor-based Software Product lines
  - Java and C
  - Different sizes and domains

- Script tool for computing two metrics:
  - *MDi*: number of methods with preprocessor directives
  - *MDe*: number of methods with feature dependencies

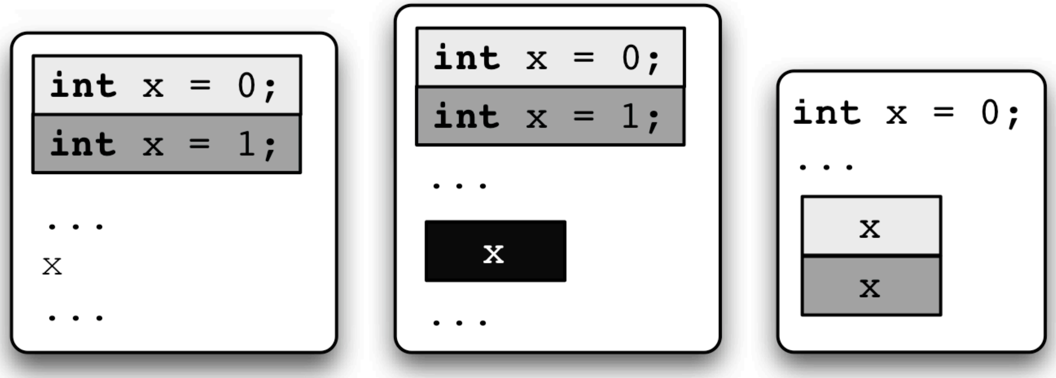
Feature dependencies our tool considers



Declaration/Assignment - Use:
One **#ifdef**

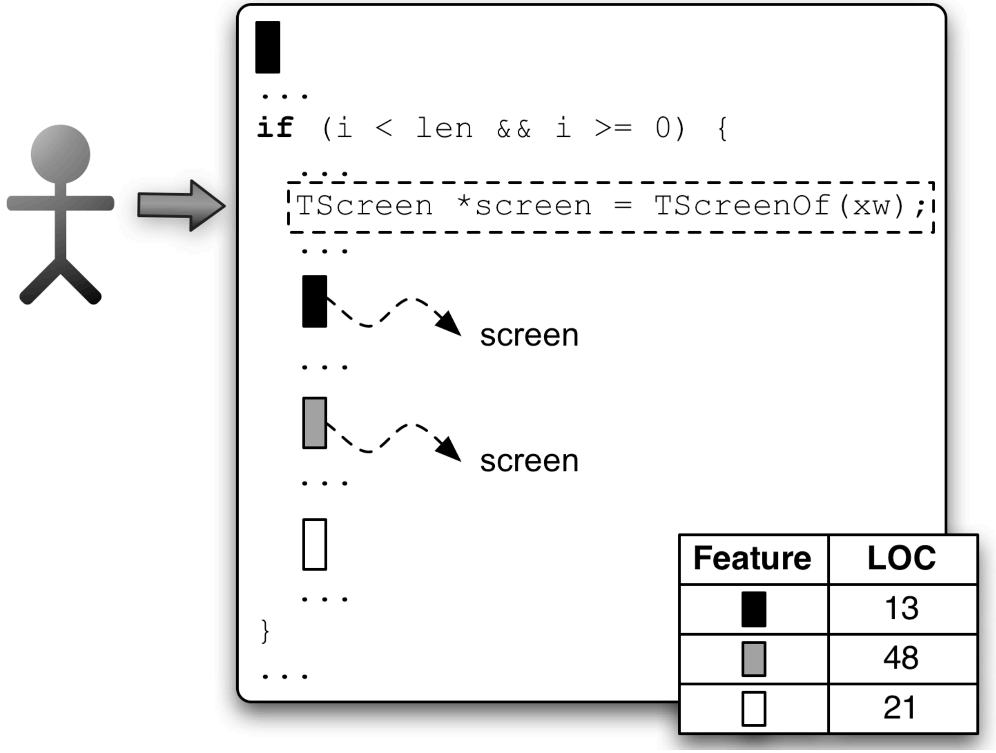


Declaration/Assignment - Use:
Nested **#ifdefs**



Alternative features:
#ifdef (light gray) followed by **#else** (dark gray)

Effort estimation (= or ≠)



```
■
...
if (i < len && i >= 0) {
  ...
  TScreen *screen = TScreenOf(xw);
  ...
  ■
  ...
  ▨
  ...
  □
  ...
}
...
```

Feature	LOC
■	13
▨	48
□	21

Number of Fragments

Number of Features

Approach	LOC	NoFa	NoFe
VSoC	82	3	3
Emergent	61	2	2

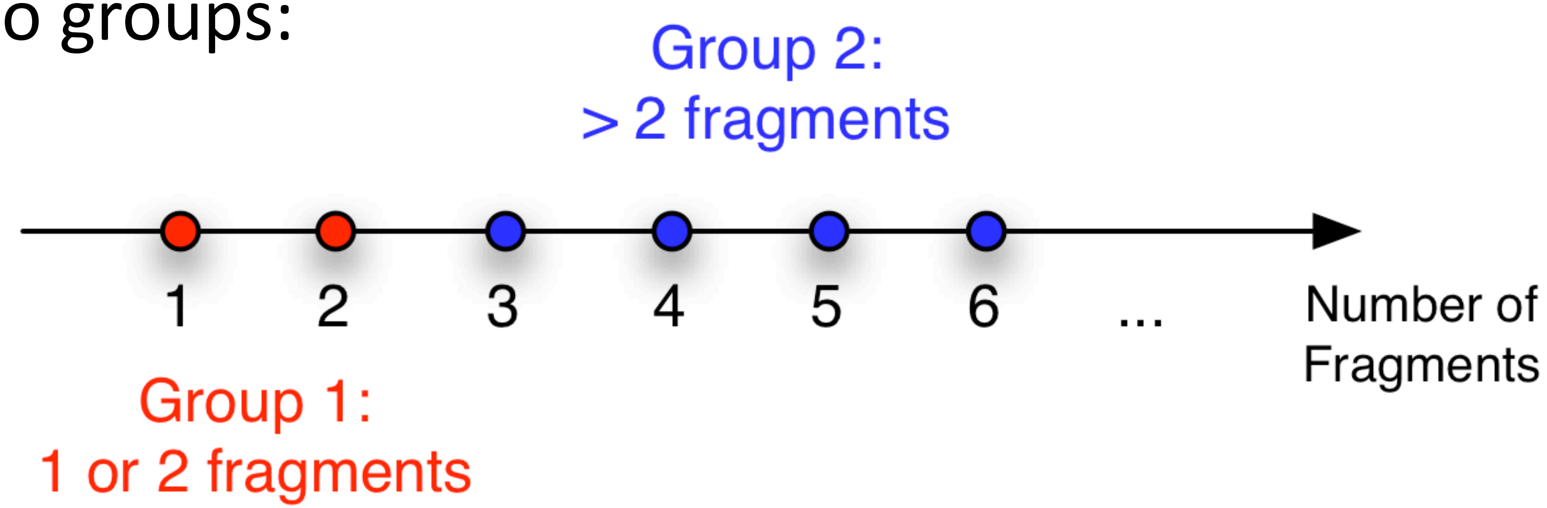
Methods selection

- Randomly methods selection
  - Only methods that contain dependencies (our focus)

- Which methods should we select?
  - Many fragments: favoring emergent interfaces
  - Few fragments: no differences

Groups

- Two groups:



- Why 2 as a threshold?
  - Differences between both approaches appear from 2
  - 1: both approaches have always the same effort estimation

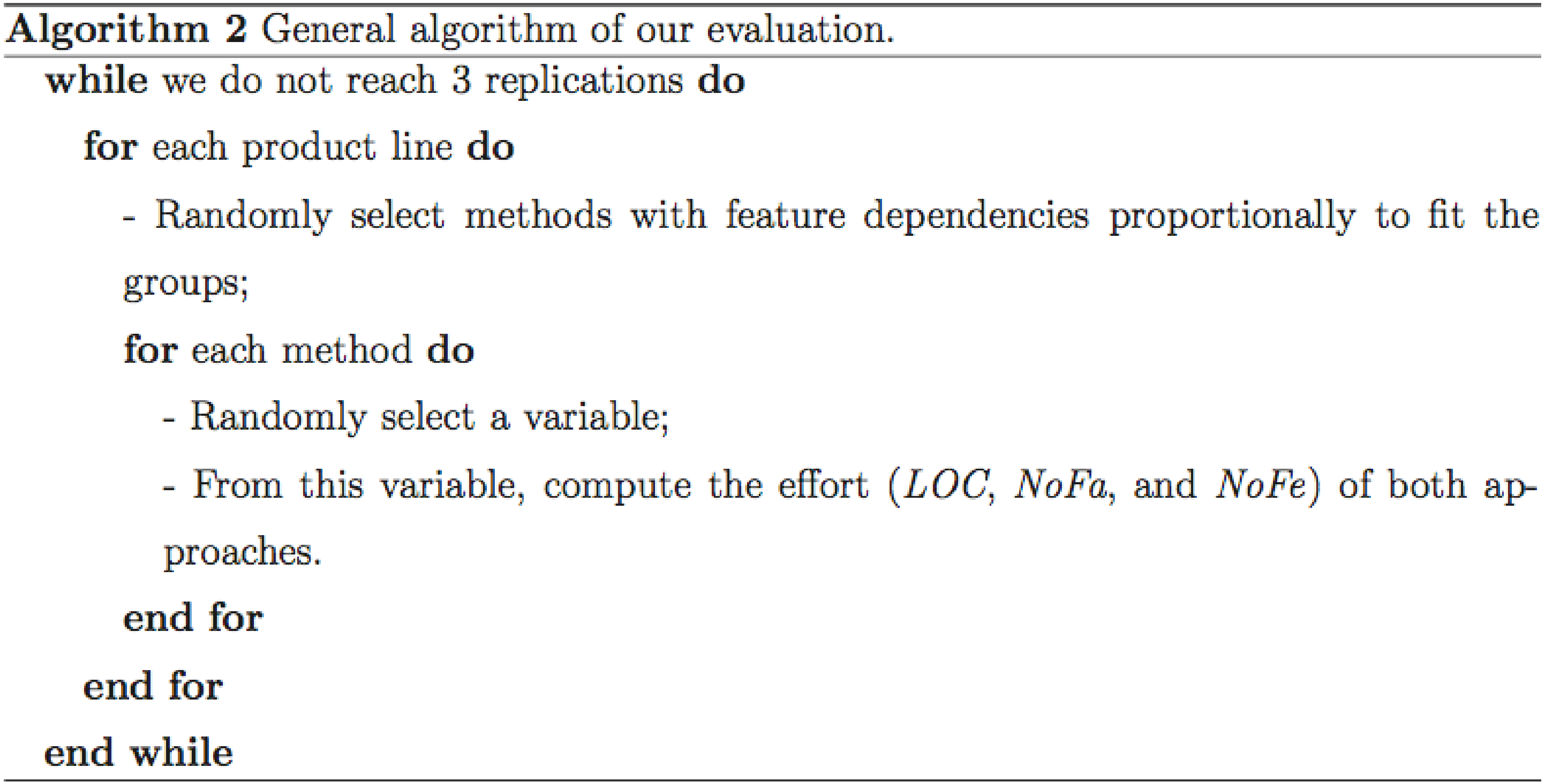
Methods selection to fit the groups

- Proportional selection according to each SPL

- Example: *libxml2*
  - Group 1: 125 methods (1 method selected)
  - Group 2: 953 methods (8 methods selected)

- Majority
  - 1:1 (28 product lines)

General algorithm



Algorithm 2 General algorithm of our evaluation.

```
while we do not reach 3 replications do
  for each product line do
    - Randomly select methods with feature dependencies proportionally to fit the groups;
    for each method do
      - Randomly select a variable;
      - From this variable, compute the effort ($LOC$, $NoFa$, and $NoFe$) of both approaches.
    end for
  end for
end while
```

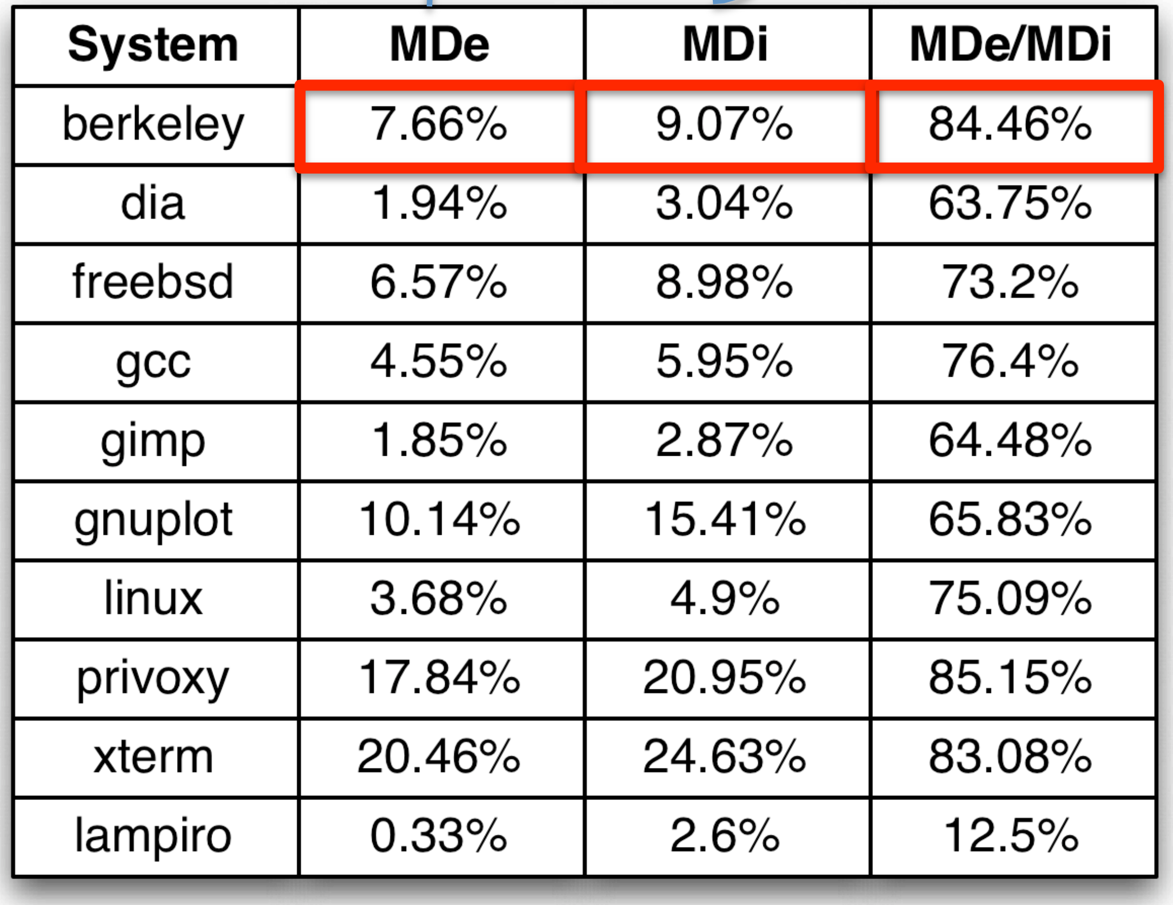
Results

Question 1: how often methods with preprocessor directives contain feature dependencies?

Frequency of feature dependencies

Methods with Dependencies

Methods with Directives



System	MDe	MDi	MDe/MDi
berkeley	7.66%	9.07%	84.46%
dia	1.94%	3.04%	63.75%
freebsd	6.57%	8.98%	73.2%
gcc	4.55%	5.95%	76.4%
gimp	1.85%	2.87%	64.48%
gnuplot	10.14%	15.41%	65.83%
linux	3.68%	4.9%	75.09%
privoxy	17.84%	20.95%	85.15%
xterm	20.46%	24.63%	83.08%
lampiro	0.33%	2.6%	12.5%

Our data reveal that…

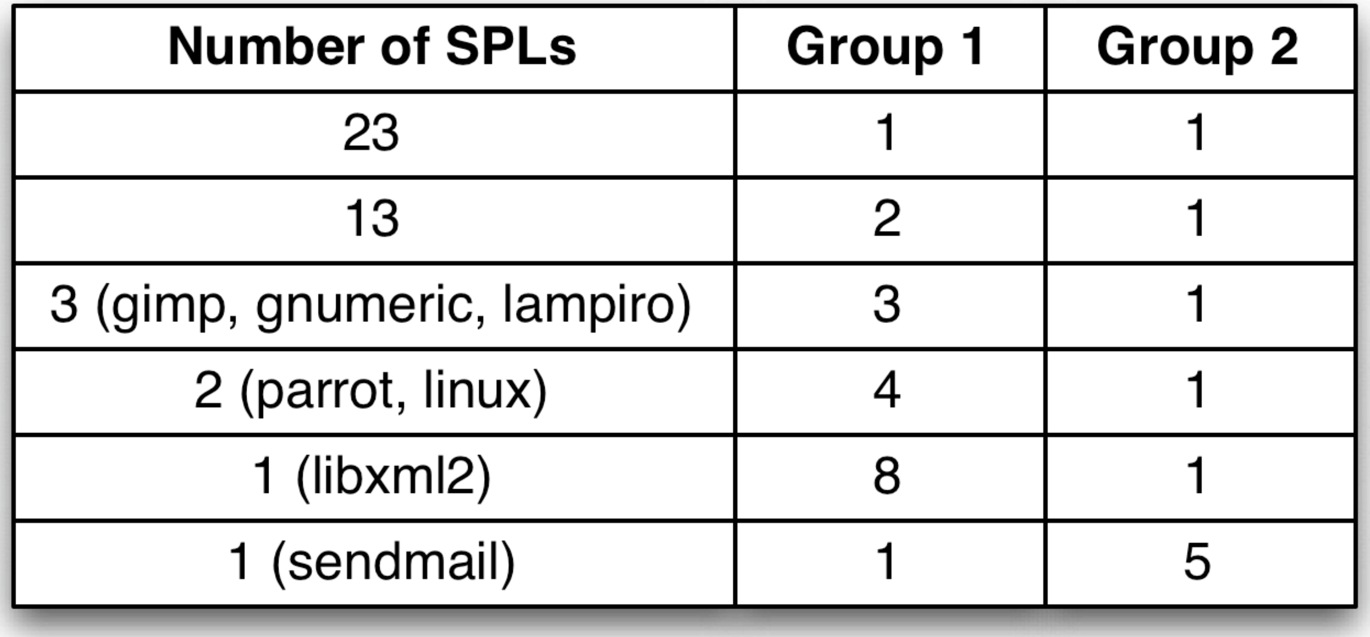
- 11.26% ± 7.13% of the methods use preprocessors
- 65.92% ± 18.54% of the methods with directives also have dependencies
- So, the feature dependencies we considered are indeed **common** in the 43 SPLs we studied

Question 2: how feature dependencies impact maintenance effort when using VSoC and emergent interfaces?

Selection: methods, groups, SPLs

- For each replication: 115 methods

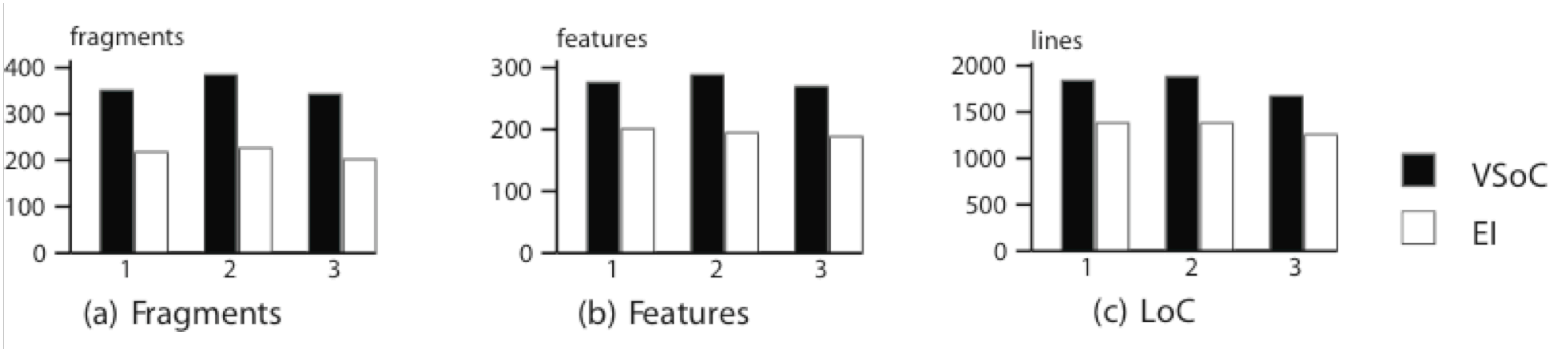
- Methods selection according to each product line



Number of SPLs	Group 1	Group 2
23	1	1
13	2	1
3 (gimp, gnumeric, lampiro)	3	1
2 (parrot, linux)	4	1
1 (libxml2)	8	1
1 (sendmail)	1	5

Global effort estimation

- Emergent interfaces: effort reduction in all replications



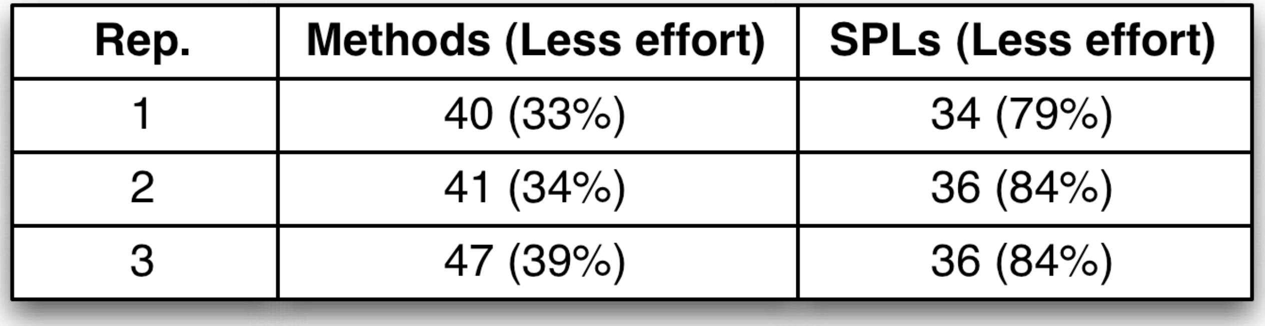
(a) Fragments

(b) Features

(c) LoC

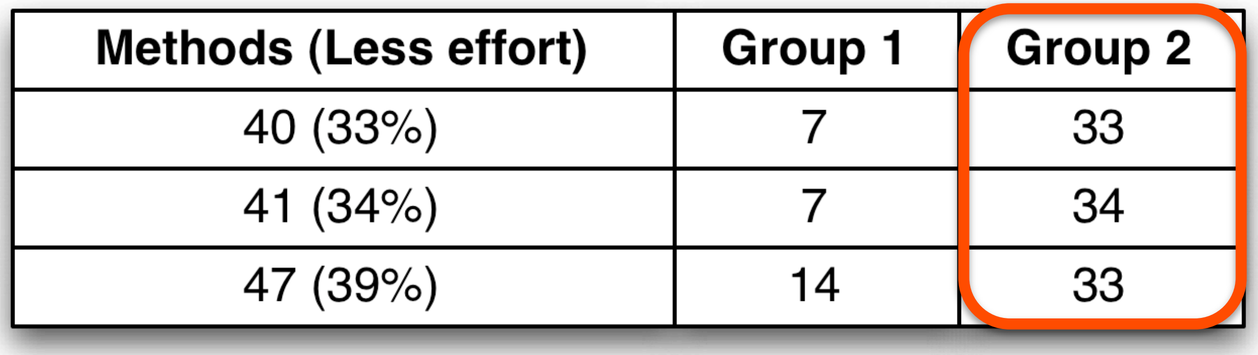
Emergent Interfaces effort reduction

- Effort reduction in the majority of the SPLs



Rep.	Methods (Less effort)	SPLs (Less effort)
1	40 (33%)	34 (79%)
2	41 (34%)	36 (84%)
3	47 (39%)	36 (84%)

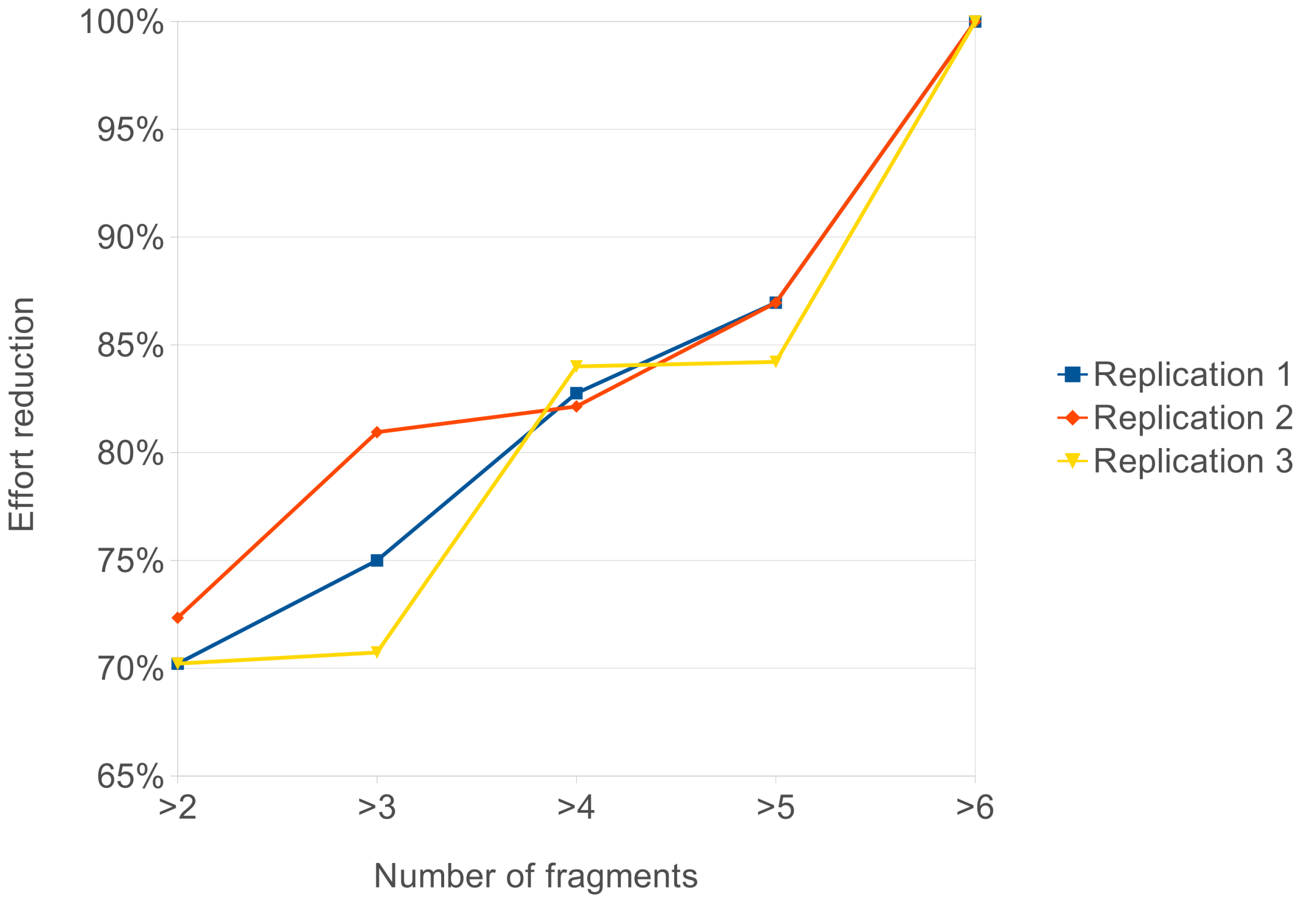
- Distribution by groups:



Methods (Less effort)	Group 1	Group 2
40 (33%)	7	33
41 (34%)	7	34
47 (39%)	14	33

When increasing the number of fragments…

- … the percentage of methods where Emergent Interfaces achieve effort reduction also increases…



Threats to validity

- Metrics and effort estimation
  - Overhead to compute emergent interfaces
  - Time better measure effort

- Highlighting tools
  - Do not consider dataflow analysis
  - We cannot hide features

- Dependencies
  - Interprocedural, chain of assignments... (not computed)

Concluding remarks

- How often feature dependencies occur in practice?
  - 65.92% ± 18.54%
  - Reasonably common in the SPLs we studied

- Emergent interfaces achieve effort reduction:
  - Methods: 35.25% ± 3.6%;
  - Majority (64.75%): same effort of VSoC
  - So, the negative impact of VSoC is **not** so common

- More significant effort reductions: methods with many fragments