# Modularidade, Componentes, Arquitetura e Reuso de Software: de Parnas aos LLMs

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco

pauloborba.cin.ufpe.br

# Software Modularity, Components, Architecture, and Reuse: from Parnas to LLMs

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

# It all begun with Parnas!

## Modularity and its benefits

Independent development

Independent maintenance

Independent understanding



# On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

**This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.**

**Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design**

**CR Categories: 4.0**

Copyright © 1972, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

**Introduction**

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

**A Brief Status Report**

The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above.

[1] Reprinted by permission of Prentice-Hall, Englewood Cliffs, N.J.



David L. Parnas. On the criteria to be used in decomposing systems into modules (CACM 1972).

“In this context module is considered to be a **responsibility assignment** rather than a **sub-program**”

David L. Parnas

In this context module is considered to be a task (implement, fix, improve…) rather than a class, package, component…

# It all begun with Parnas… but was popularized by language designers!

## Modularity in the New Millenium: A Panel Summary

Premkumar Devanbu, (*Panel Chair*)
Bob Balzer, Don Batory, Gregor Kiczales, John Launchbury, David Parnas, Peri Tarr (*Panelists*)

### 1 Introduction (Chair's Statement)

Parnas' seminal work [2] on *separation of concerns* in design has led to diverse innovations in programming language design, to support modularity. However, there has been a growing sentiment in many quarters that there are some concerns that stubbornly resist tidy confinement, when using established modularization mechanisms in programming languages. A diverse set of new approaches have emerged in response: aspects [1], monads [5], mixin layers [3], and multi-dimensional separation of concerns [4]. These approaches arose more or less independently of each other, and have (to varying degrees) developed technical maturity, real-world credibility and strong user bases. We are also now beginning to see strong scholarly comparisions of the intellectual foundations and practical utility of these different aproaches. This panel aims to support this trend.

In this panel, we bring together leading experts (Profs. Batory, Kiczales, and Launchbury, and Dr. Tarr) in these different areas. Each represents a particular perspective on how to evolve and adapt the old idea of modularization to deal with new challenges such as security, fault-tolerance, distribution, and auditing. In addition, we also have two pioneering researchers (Profs. Balzer and Parnas) to provide us with a historical perspective on the evolution (sic) of program modularization and evolution techniques.

Position statements of some of the panelists follow, presented in alphabetical order of their names:

### 2 Panel Statement: Don Batory

The future of software engineering lies in automation. Perhaps the most successful example of automated software engineering is relational query optimization (RQO). A query is specified in a declarative domain-specific language (SQL), mapped to an inefficient relational algebra expression, optimized by rewrite rules, and then an efficient query evaluation program is generated. RQO is also a great example of the holy grail problem of automatic programming transforming a declarative specification to an efficient program. Feature Oriented Programming (FOP) aims to generalize this powerful paradigm to other software domains. An FOP domain model is a set of operators that define an algebra. Each operator implements a feature that is characteristic of programs in that domain; compositions of these operators define a specific program. FOP is ideally suited for product-lines and program generators. AHEAD is both a model and tool suite for FOP that is based on step-wise refinement, the paradigm that builds complex applications from simple applications by progressively adding features. The AHEAD tool suite has been bootstrapped, where its 150+K LOC are being generated from simple, declarative specifications. Optimizations in our tool building process are now being introduced to make it like RQO.

### 3 "It's the cross-cutting": Gregor Kiczales

The idea of separation of concerns is not new; it is a basic element of modern thought.

We are however, constantly renewing our tools for separating, composing and otherwise operating on concerns throughout the lifecycle. We have seen several such evolutions, including the most recent major success, object-orientation.

The critical idea in aspect-oriented programming (AOP) is that no single decomposition can capture all the concerns in a complex system in a modular way. This incredibly simple idea is, in retrospect, nearly obvious. But until AOP we spent countless hours (days, weeks...) refactoring a system over-and-over trying to get a structure in which all concerns are modular. The frustration of working with single decompositions of complex systems led to the realization that choosing a nice structure for modularizing many concerns inherently means that some others cannot be modular, if they have to live within that primary structure. We call such concerns crosscutting, because they cut across the natural lines of the rest of the system.

The rapid uptake of AOP in industry is because developers find that adopting AOP tools like AspectJ is simple and natural their own experience helps them understand crosscutting concerns at a deep intuitive level.

### 4 Panel Statement: David Parnas

To a man with a hammer, everything looks like a nail. To a Computer Scientist, everything looks like a language design problem. Languages and compilers are, in their opinion, the only way to drive an idea into practice.

0-7695-1877-X/03 $17.00 © 2003 IEEE

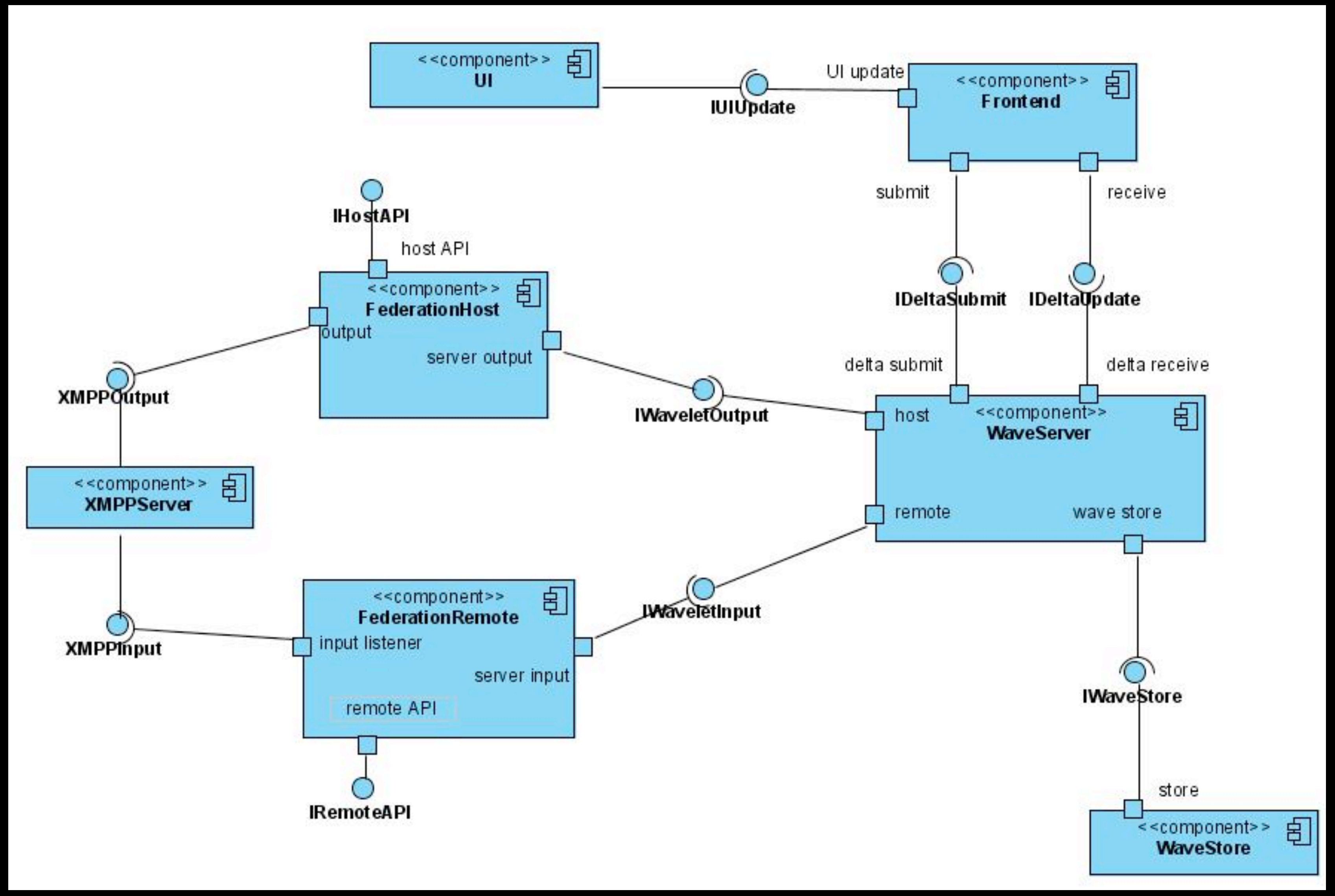


Premkumar Devanbu, Bob Balzer, Don Batory, Gregor Kiczales, John Launchbury, David Parnas, Peri Tarr. Modularity in the New MiHenium: A Panel Summary (ICSE 2003).

# A modular system is...

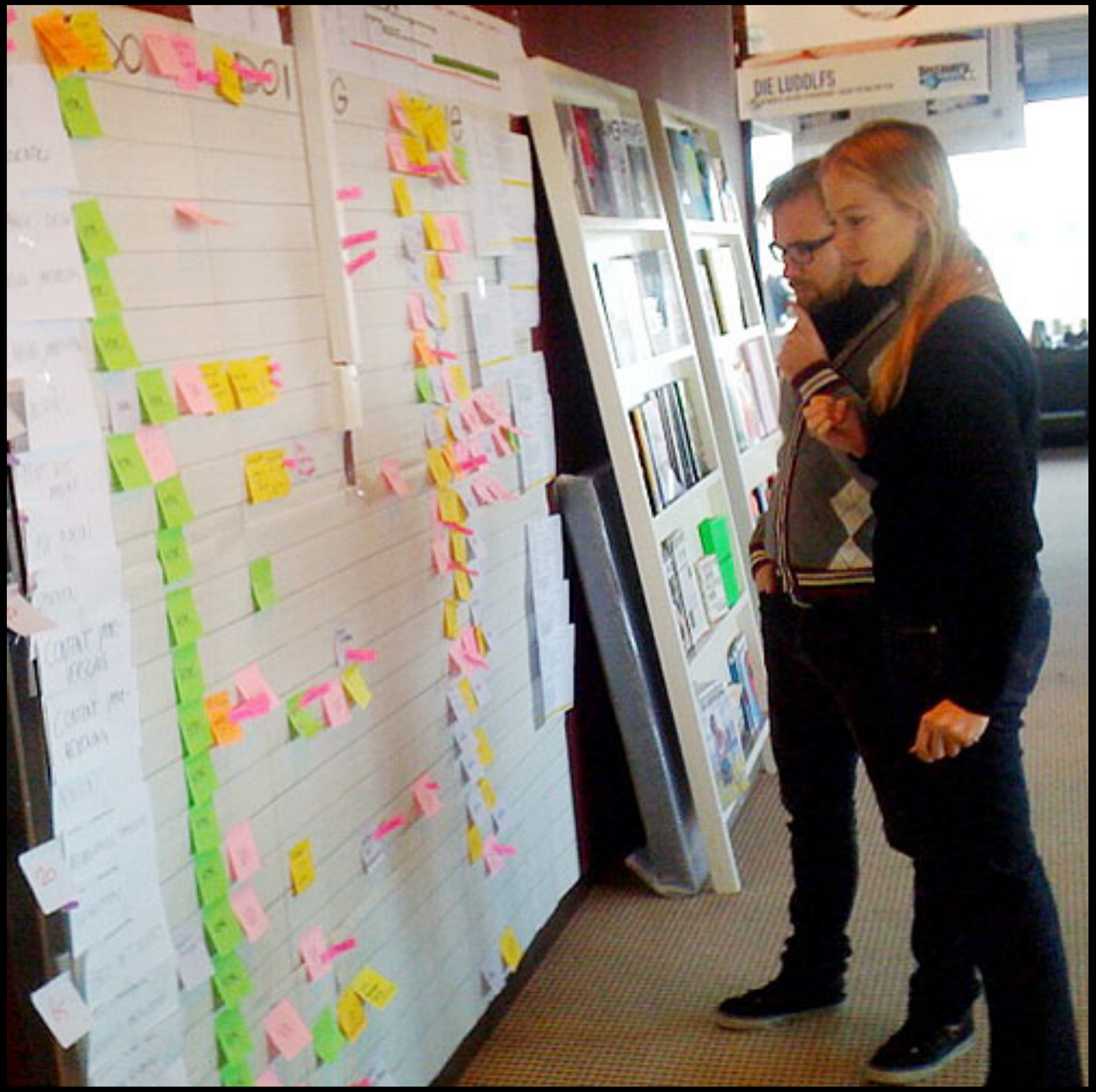
## (code modularity view)

- split into separate
    - code units
    - language elements
    - modules
    - with well defined interfaces




http://docs.google.com/Doc?docid=0Aeq-cxjLYT32ZGRicGpuYl8zM3d0cDI0NmRr&hl=en

# A modular system is...
## (work modularity view)



http://static-c.frankwatching.com/wp-content/uploads/2010/12/SNS-Reaal-Scrum-board.jpg

- developed by separate
  - work assignments
  - design elements
  - modules
  - with well defined interfaces

# A modular system is...

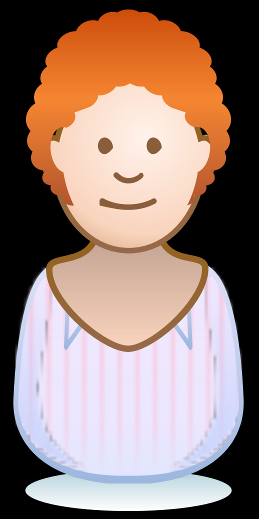
- split into separate
  - code units
  - language elements
  - modules
  - with well defined interfaces

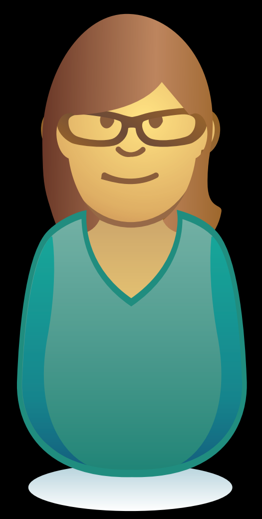
- developed by separate
  - work assignments
  - design elements
  - modules
  - with well defined interfaces

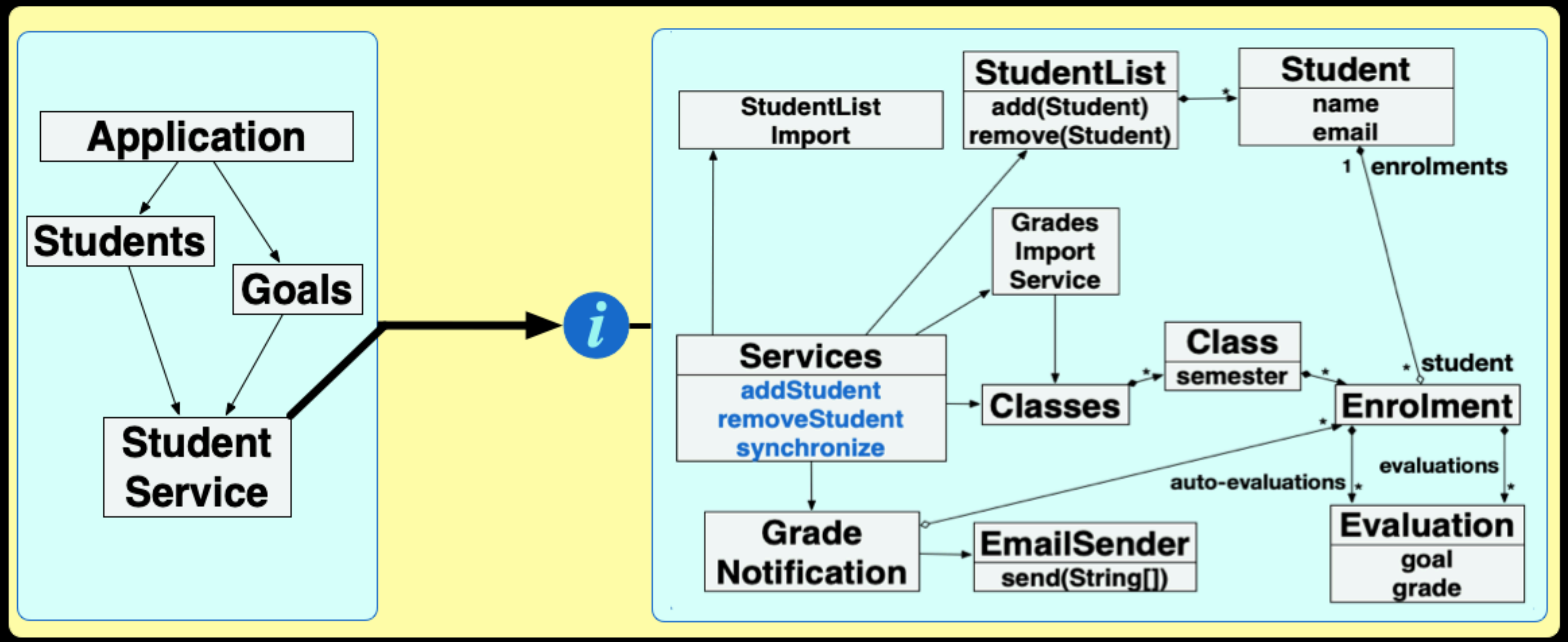
new (old) vision

We should go beyond the more restricted, and dominant, code modularity vision

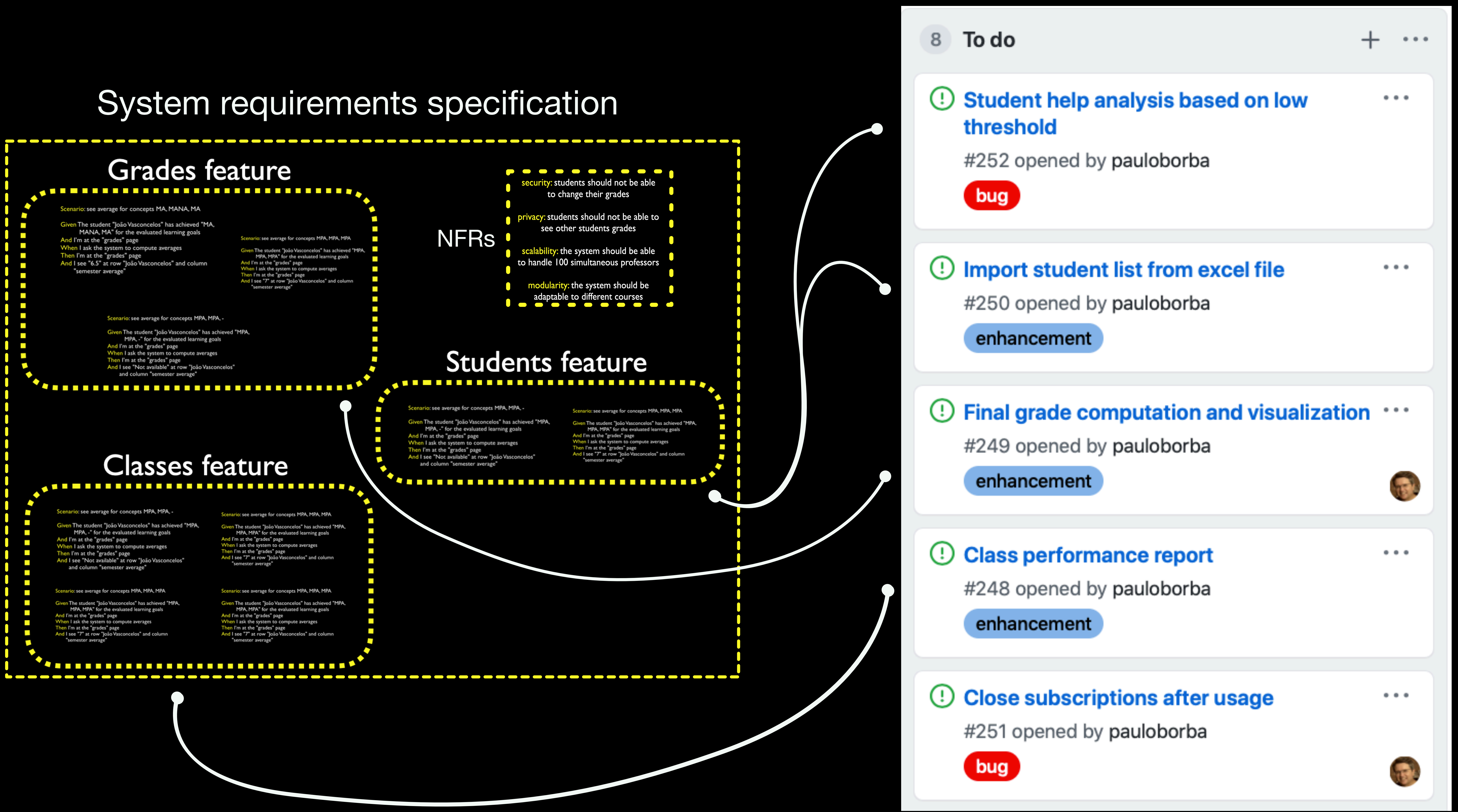
# Collaborative software development



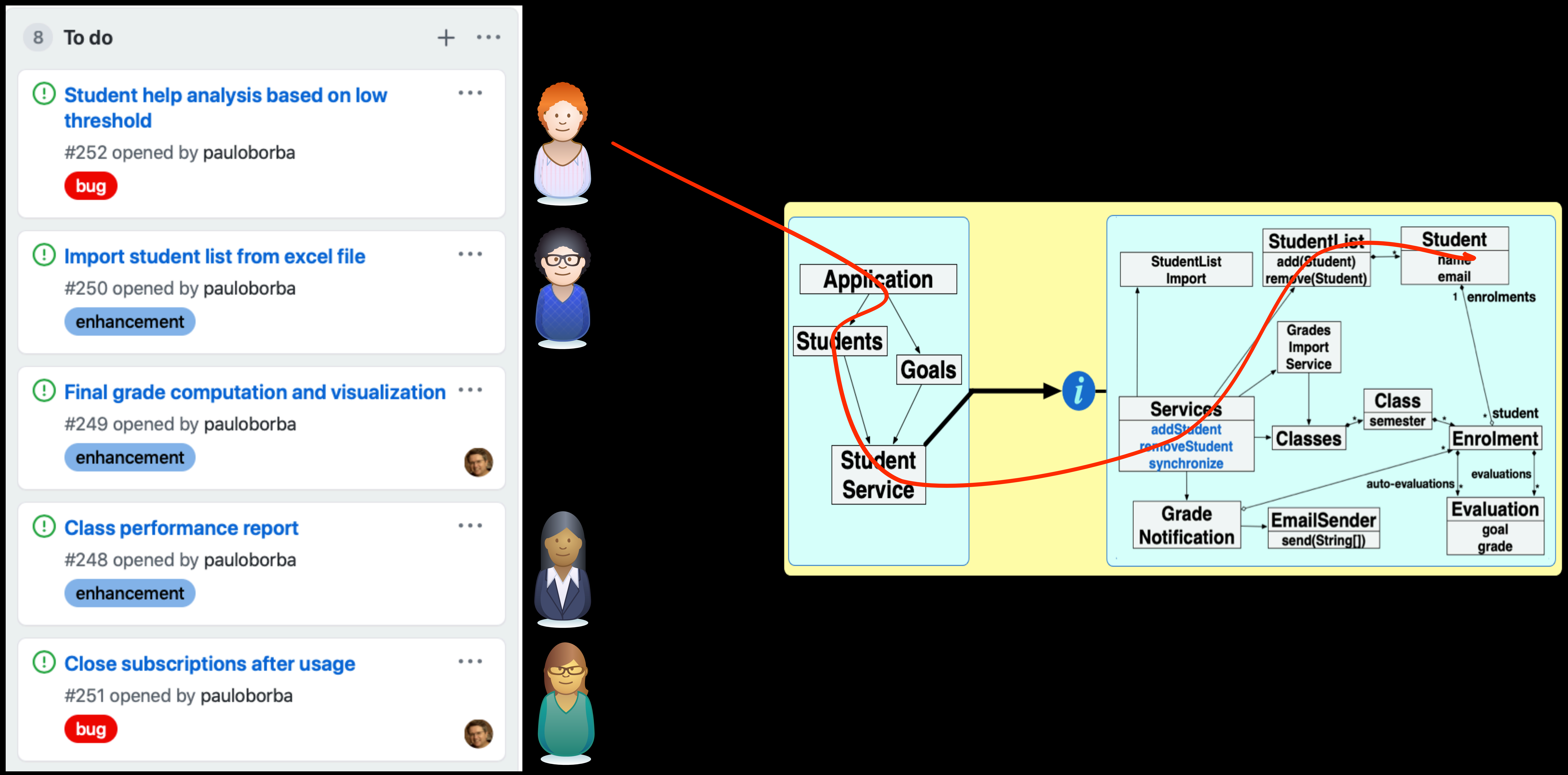




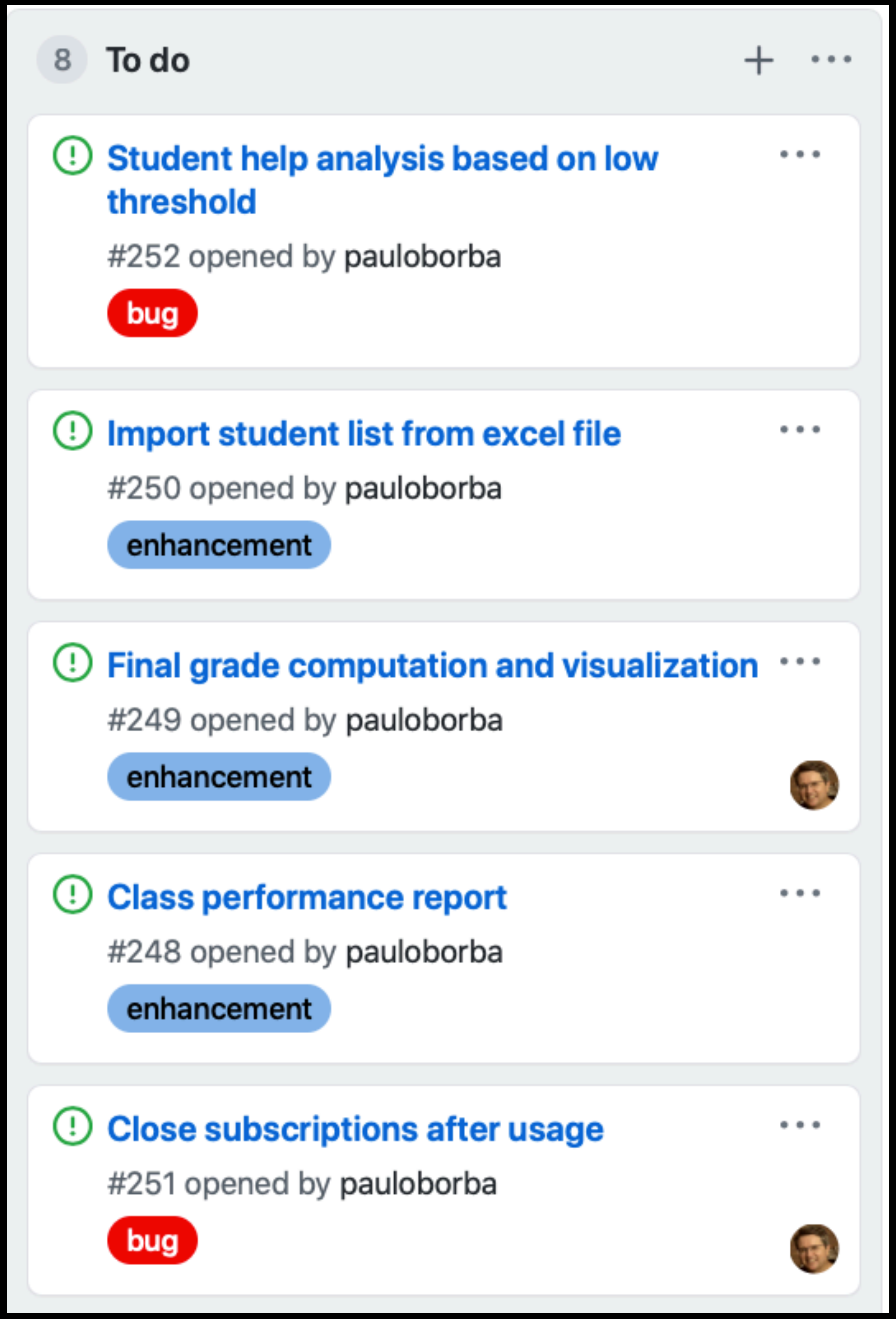

# Task structure is often derived from requirements structure

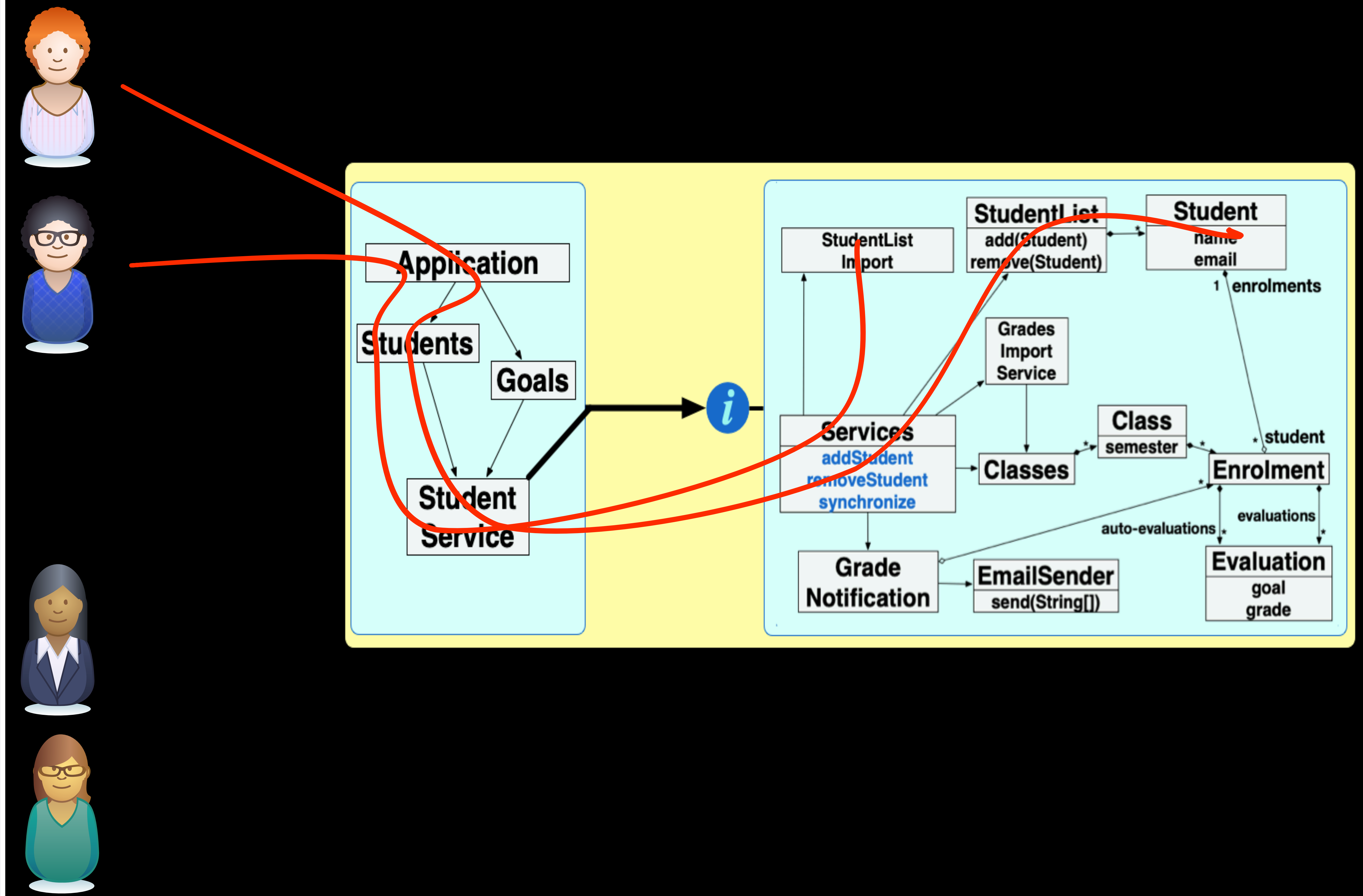


# Tasks are often crosscutting

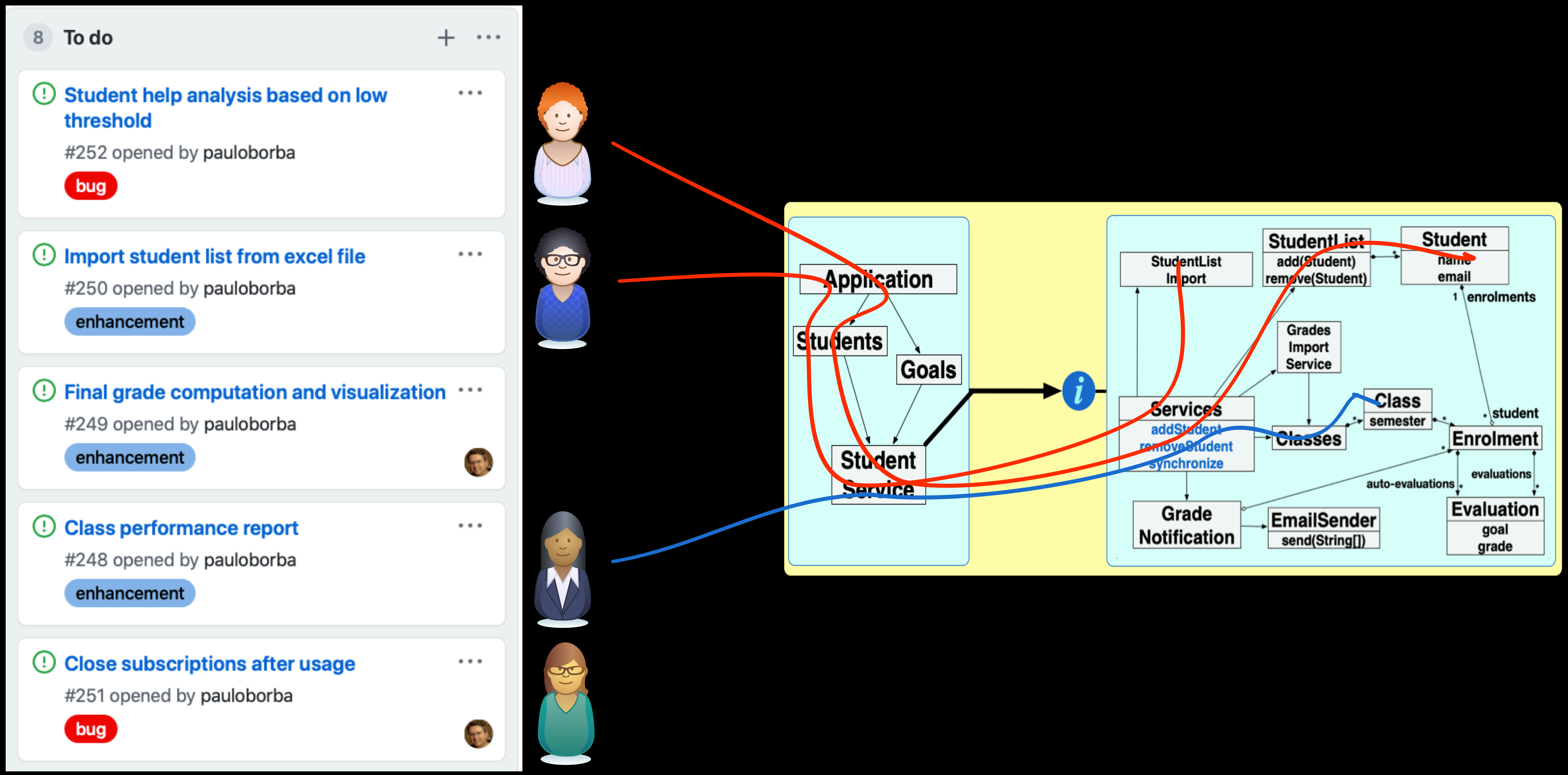


# Tasks might involve changing classes and components in common

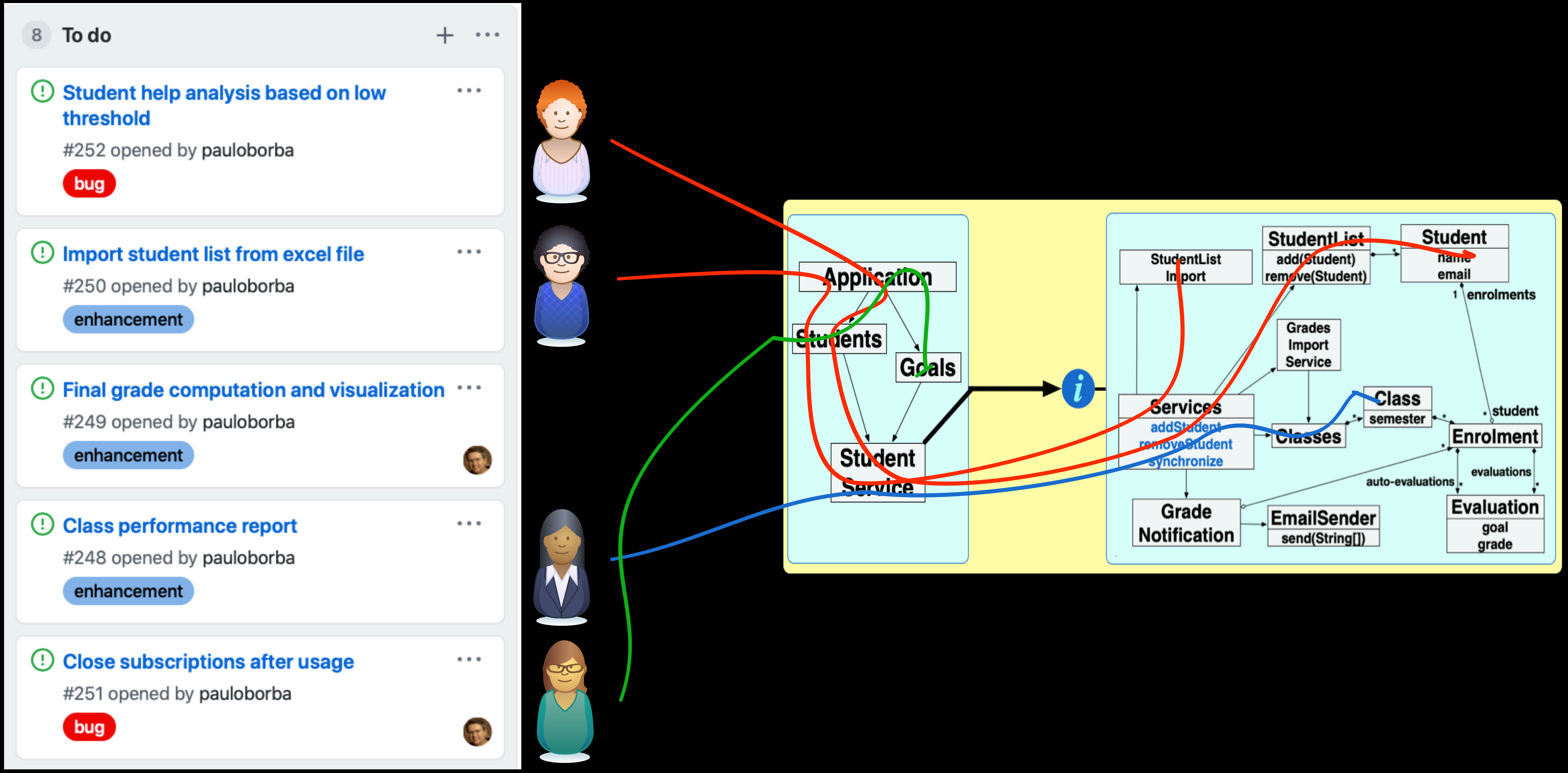


# Task structure often does not match code structure



# Modular development is not always possible, no matter the investment in code modularity



Good code modularity
is a relative concept

Code modularity depends on what you intend to independently develop, maintain, and understand!

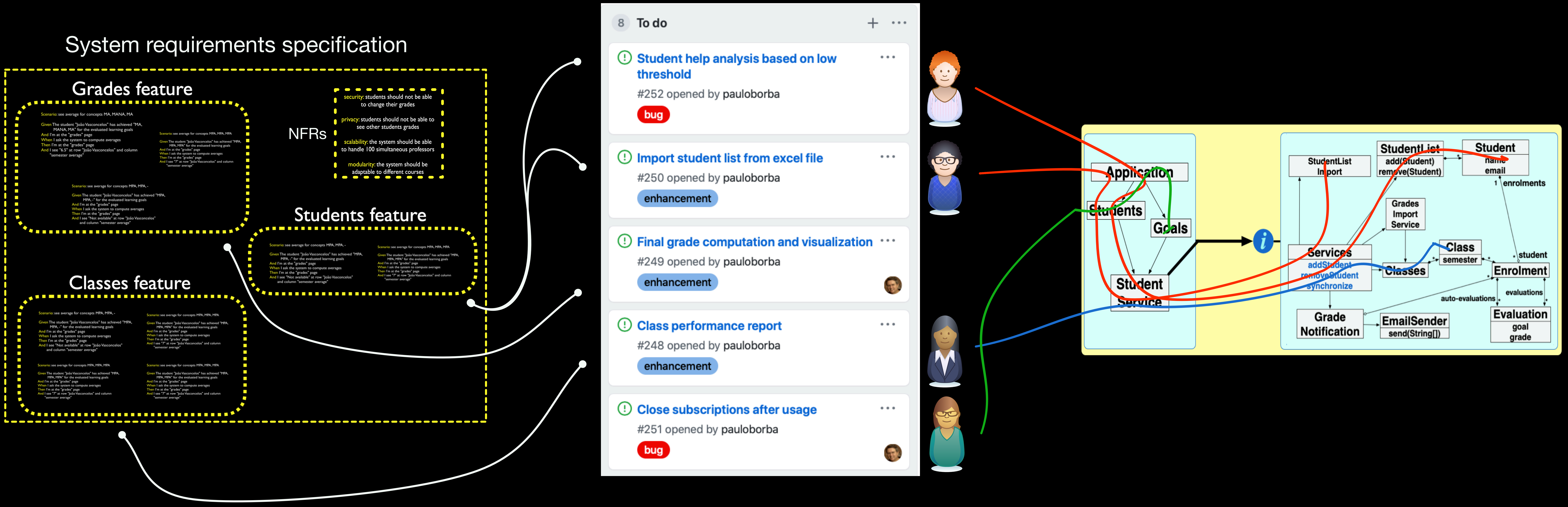
Depends on what more likely changes or varies!

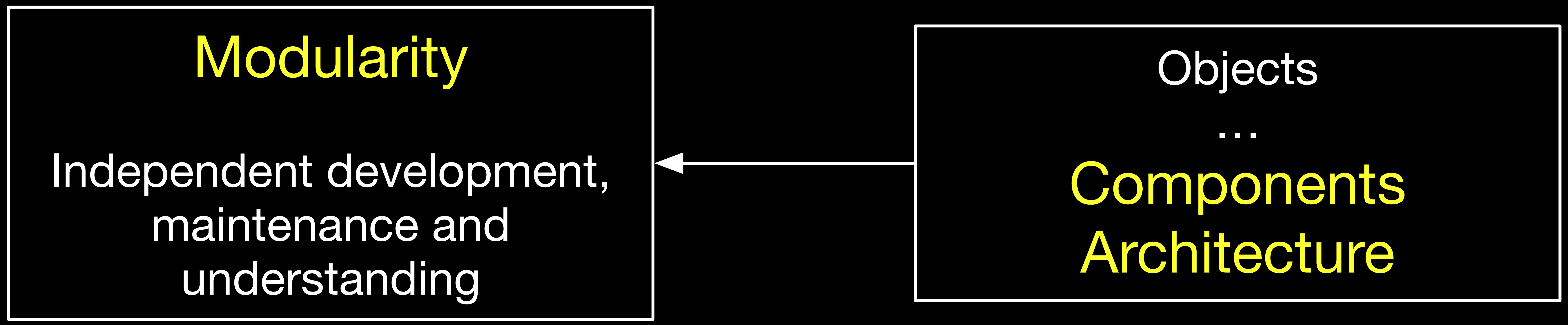
# Different tasks (evolution, integration, customization) are favored by different modular designs

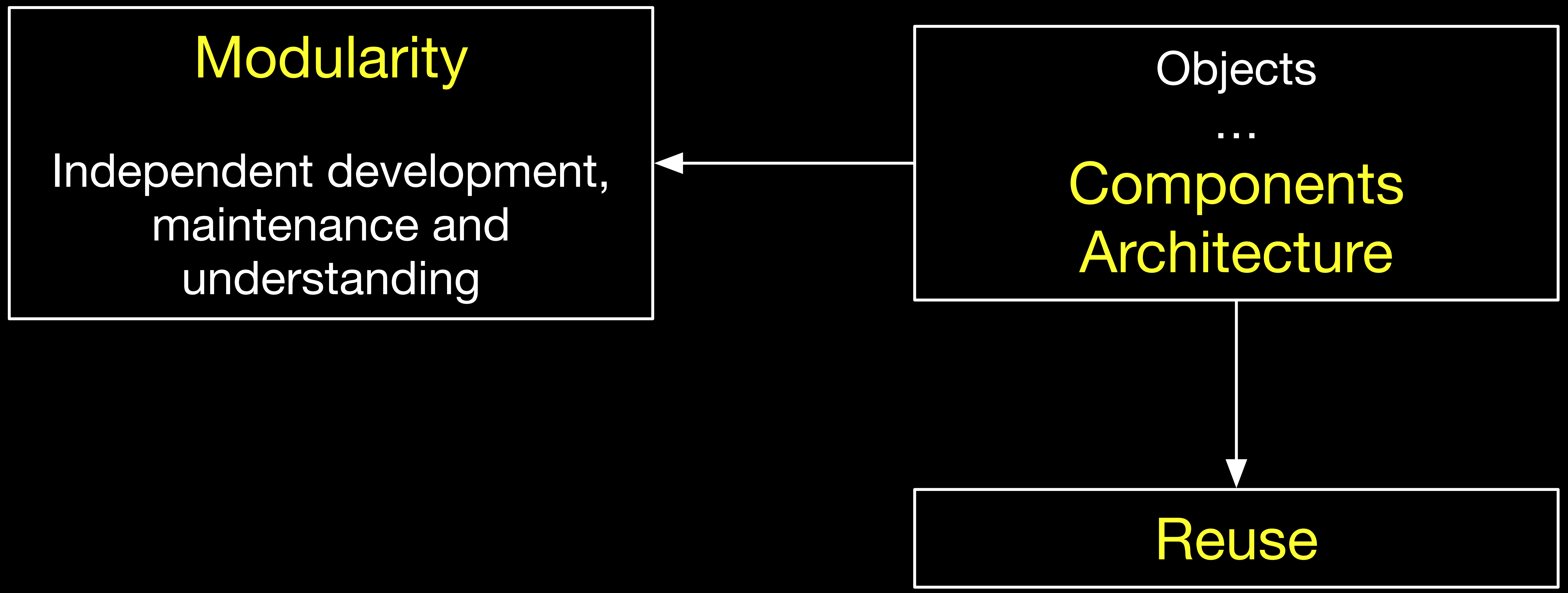
Fernanda d'Amorim and Paulo Borba. Modularity analysis of use case implementations (JSS 2012).

Fernanda d'Amorim, Paulo Borba. Modularity Analysis of Use Case Implementations (SBCARS 2010).

# The limits of code modularity: different modular structures for different artifacts

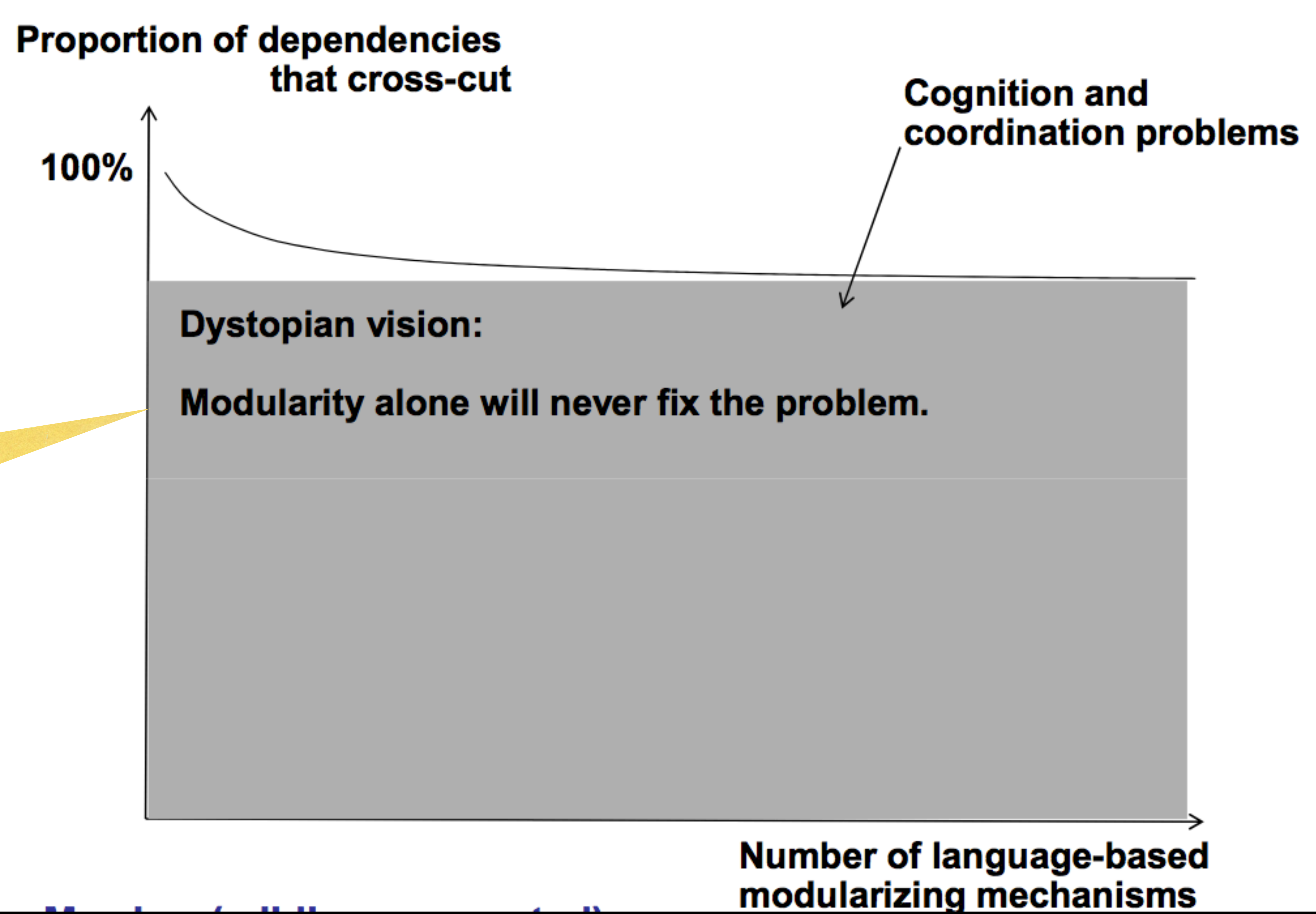



Modularity
Independent development, maintenance and understanding
Objects
…
Components
Architecture


Modularity
Independent development, maintenance and understanding
Objects
...
Components
Architecture
Reuse

# This is not an issue to be solved only by language mechanisms

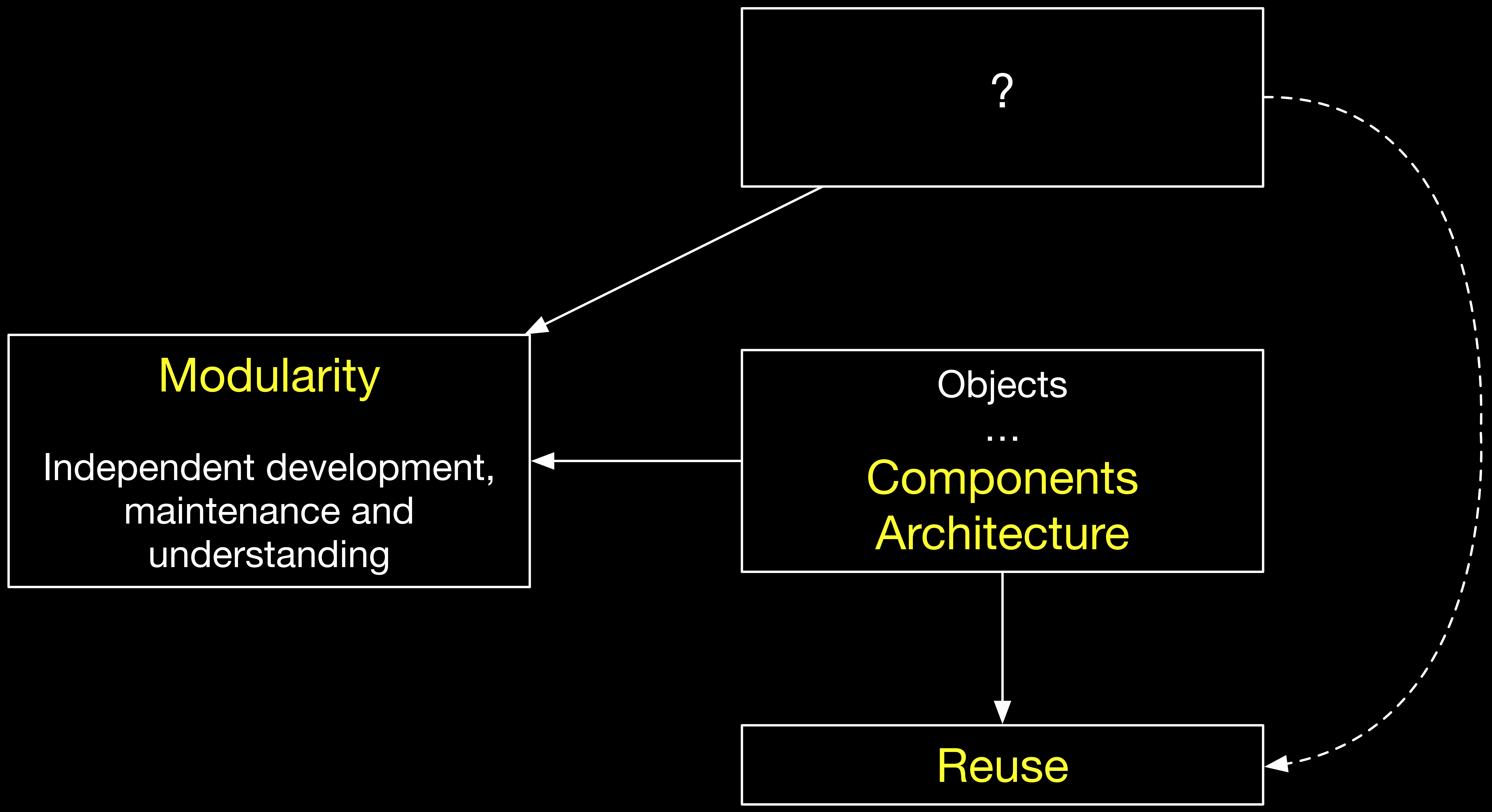
language modularity

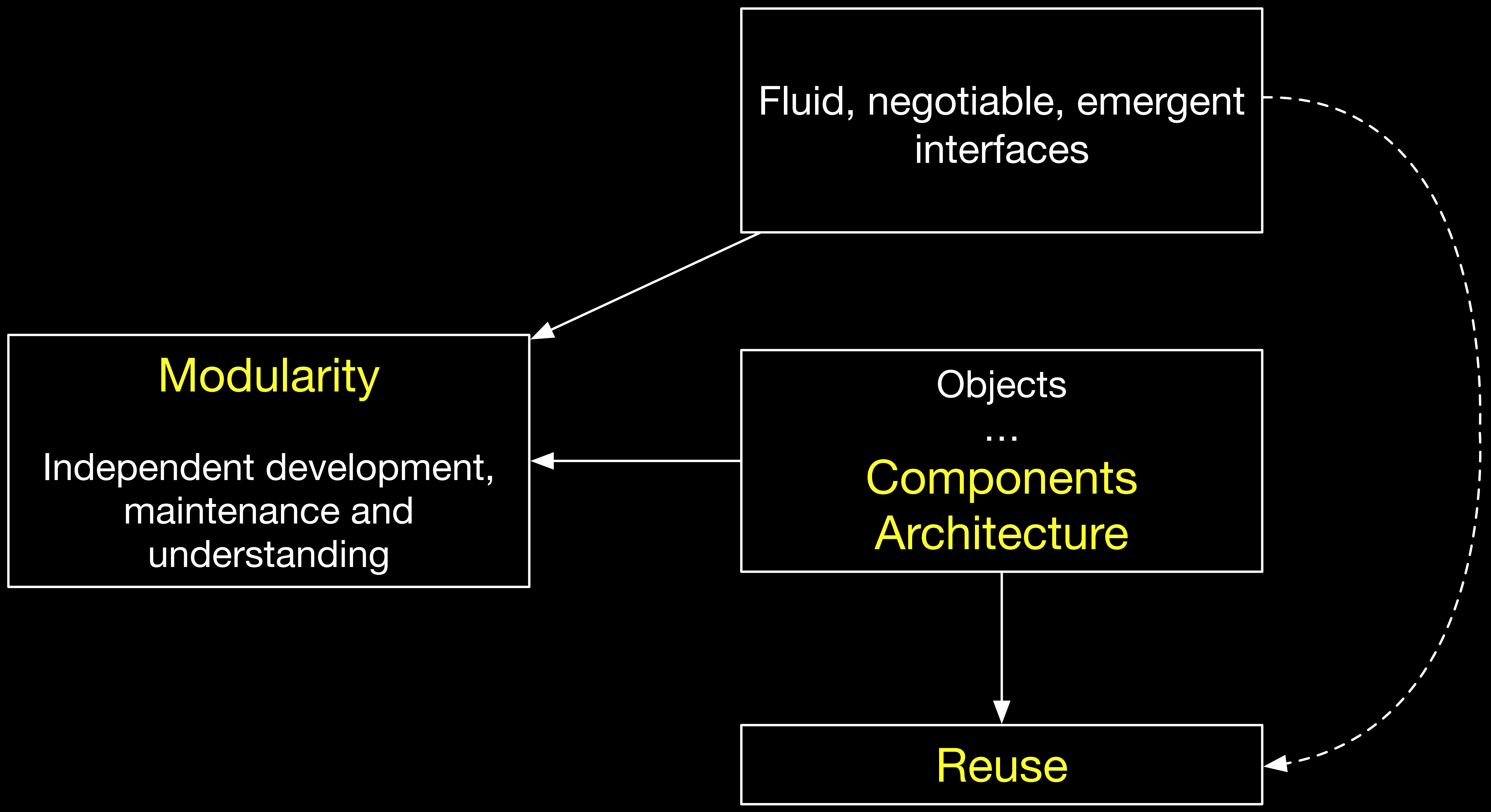



James Herbsleb, http://aosd.net/2011/files/keynotes/Herbsleb-AOSD-2011-v1-dist.pdf

conjecture

Accept the limits of language and pattern modularity


?
Modularity
Independent development, maintenance and understanding
Objects
...
Components
Architecture
Reuse


Fluid, negotiable, emergent interfaces
Modularity
Independent development, maintenance and understanding
Objects
...
Components
Architecture
Reuse

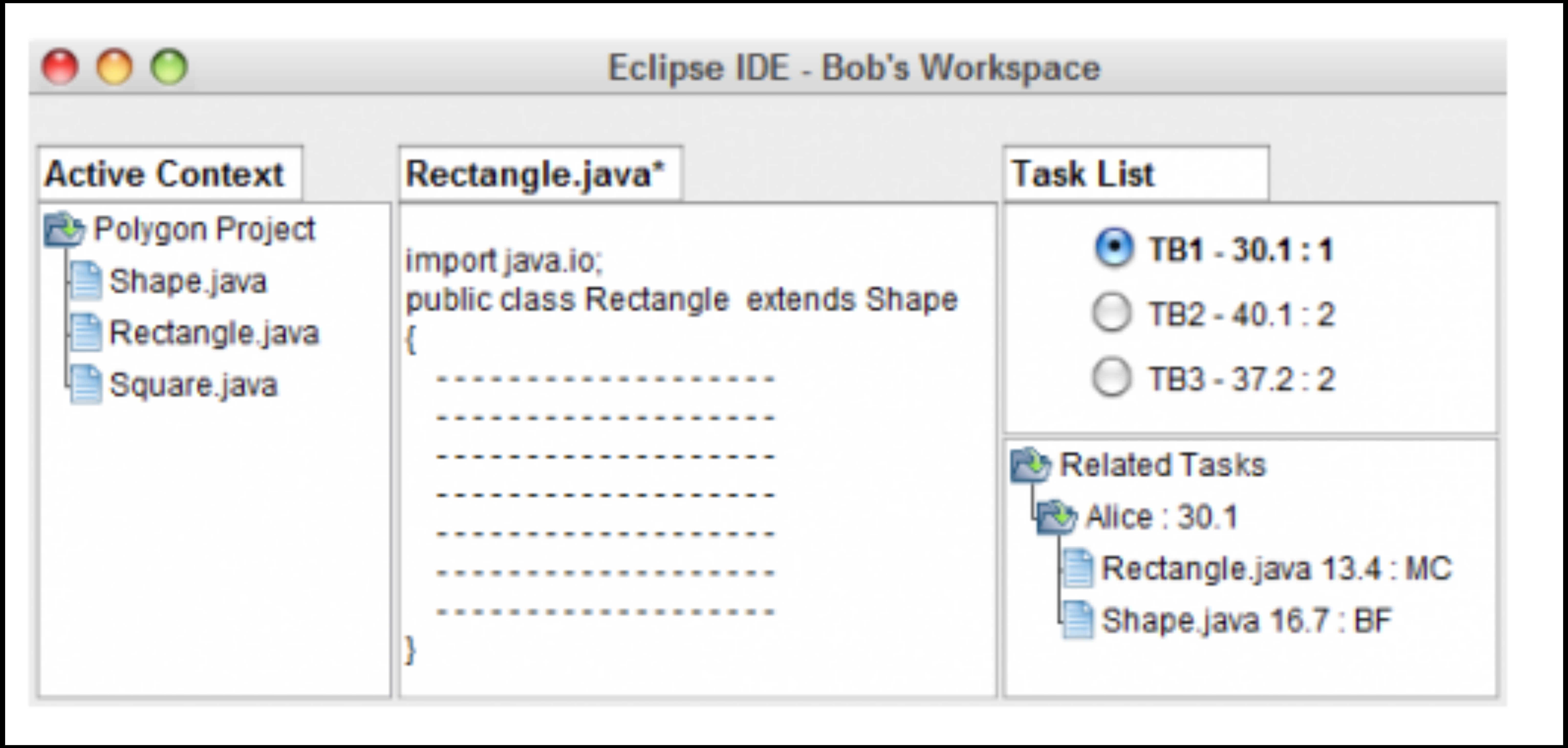
# Task, non language, modularity

Tools enable modular development, understanding, and maintenance by supporting task scheduling and independent task realization

# Task scheduling

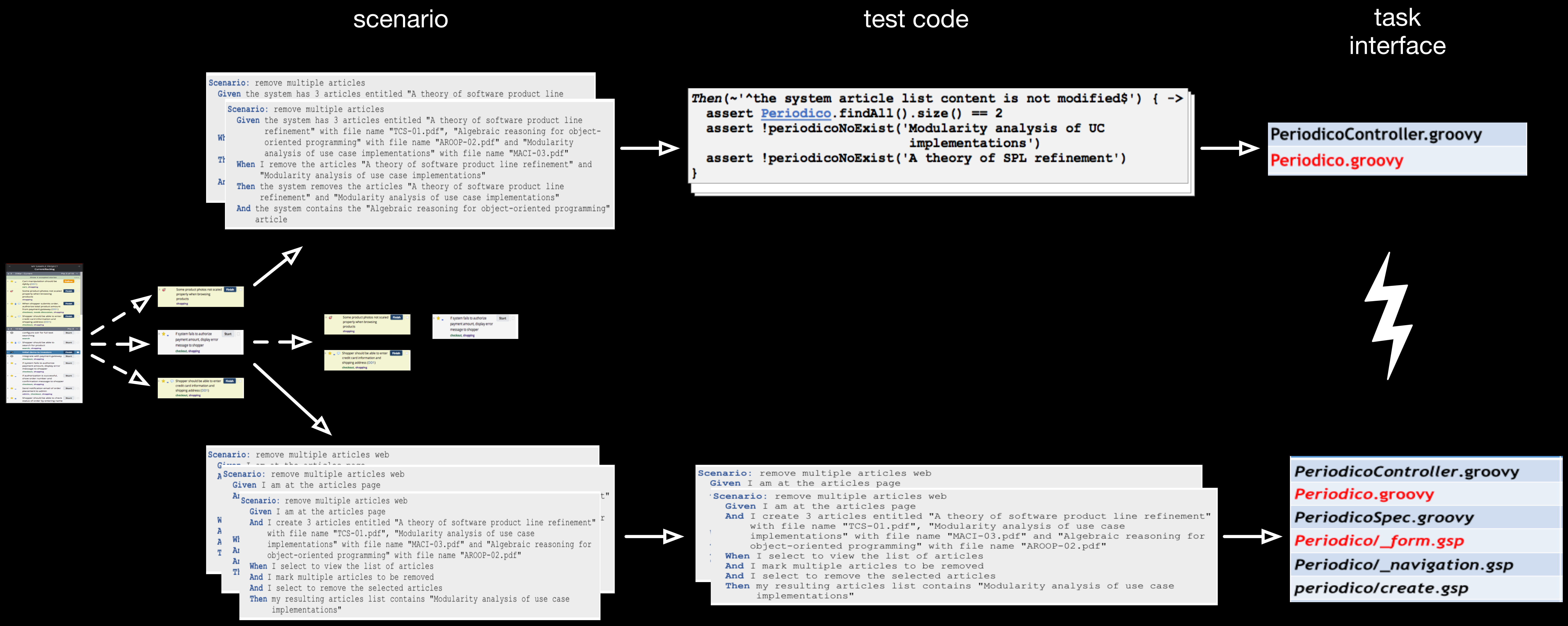
schedule tasks so that collaboration conflicts can be avoided

# Cassandra: optimized task scheduling, for experienced developers




Bakhtiar Khan Kasi and Anita Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling (ICSE 2013).

# Taiti: scenarios and tests, in BDD contexts




Thaís Rocha, Paulo Borba. Using acceptance tests to predict merge conflict risk (ESE 2023).

Thaís Rocha, Paulo Borba, João Santos. Using acceptance tests to predict files changed by programming tasks (JSS 2019).

João Santos, Thaís Rocha, Paulo Borba. Improving the prediction of files changed by programming tasks (SBCARS 2019).

# Tasks with non-disjoint TestI interfaces more likely modify files in common

They are **2.07 times** more likely to change a file in common

Test-first context

&

significant test coverage

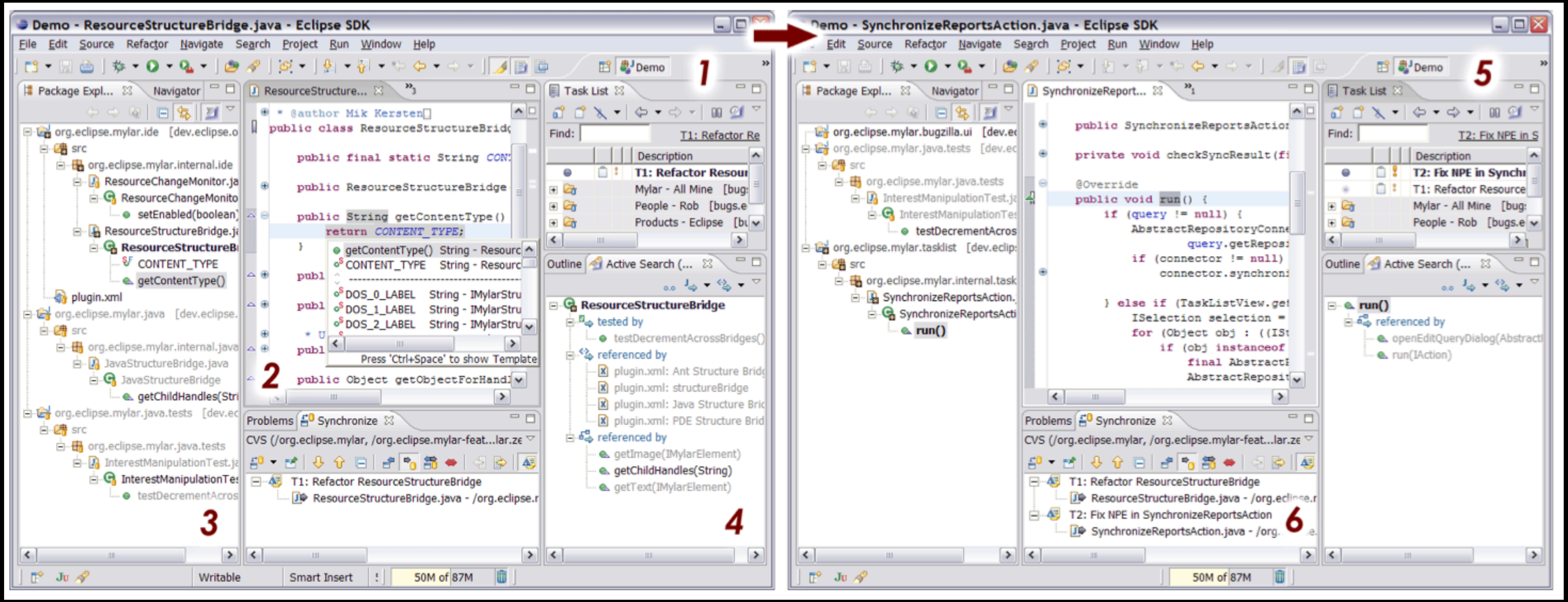
$\Longrightarrow$

Test-based task interfaces as an additional factor to consider for scheduling programming tasks

# Independent task realization

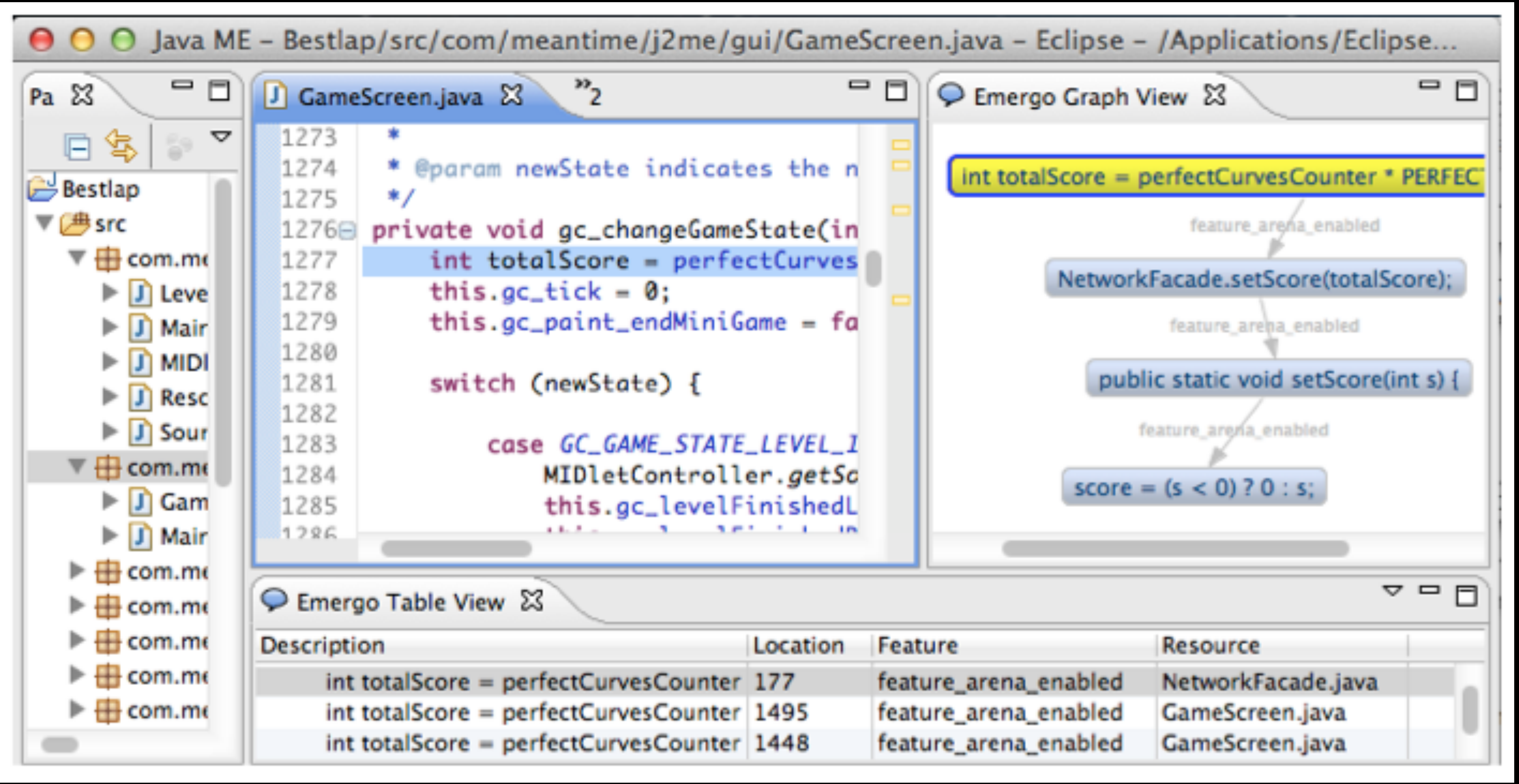
bring context, dependencies and risks to the attention of developers so they can act appropriately

# Mylyn: task context, fluid modules, but no interfaces




Kersten, M., & Murphy, G. C. (2006). Using task context to improve programmer productivity.

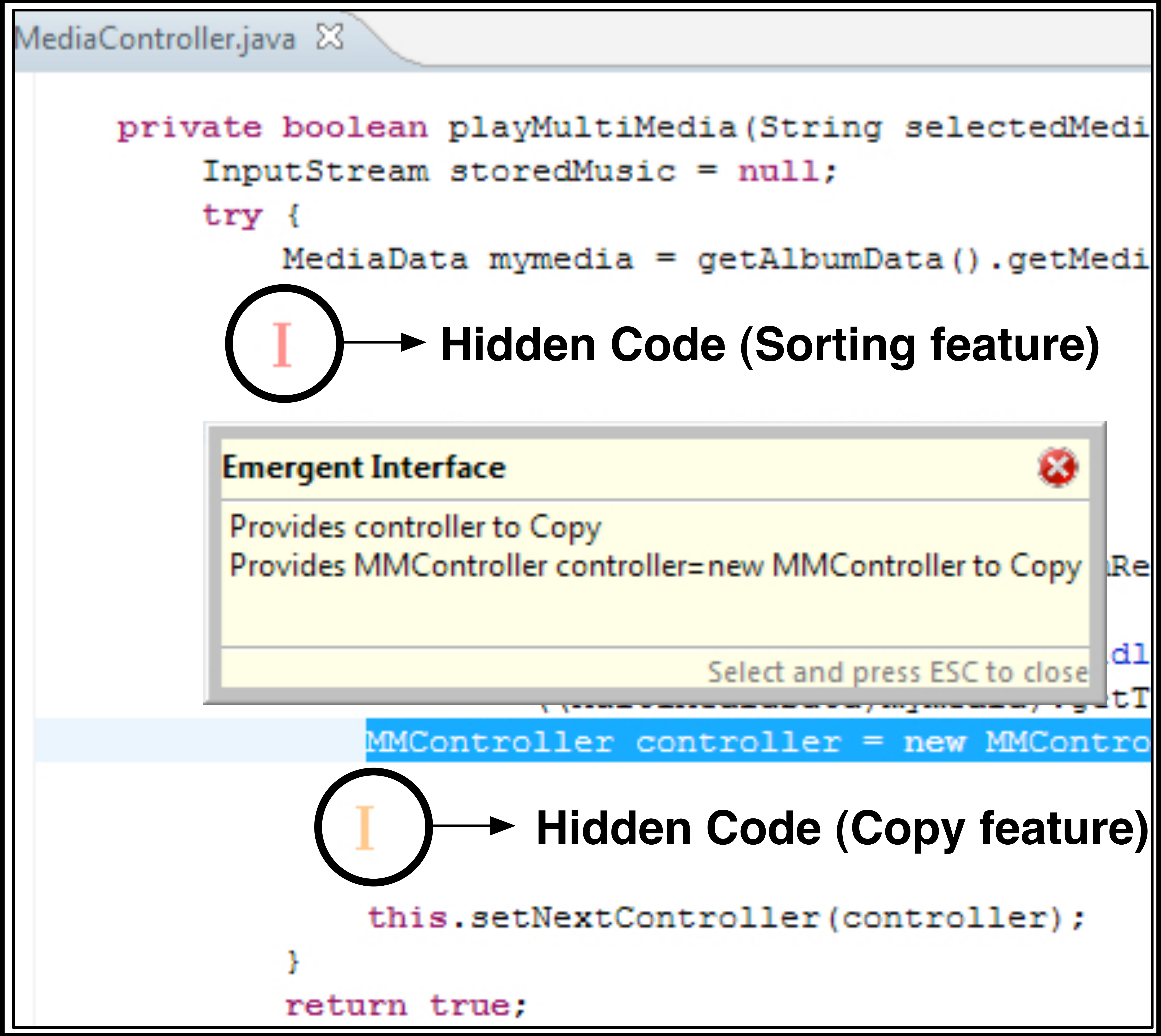
# Emergo: feature modules, task dependencies and emergent interfaces





Márcio Ribeiro, Paulo Borba, Christian Kästner. Feature Maintenance with Emergent Interfaces (ICSE 2014).

Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, Paulo Borba. Emergent Feature Modularization (Onward! 2010).



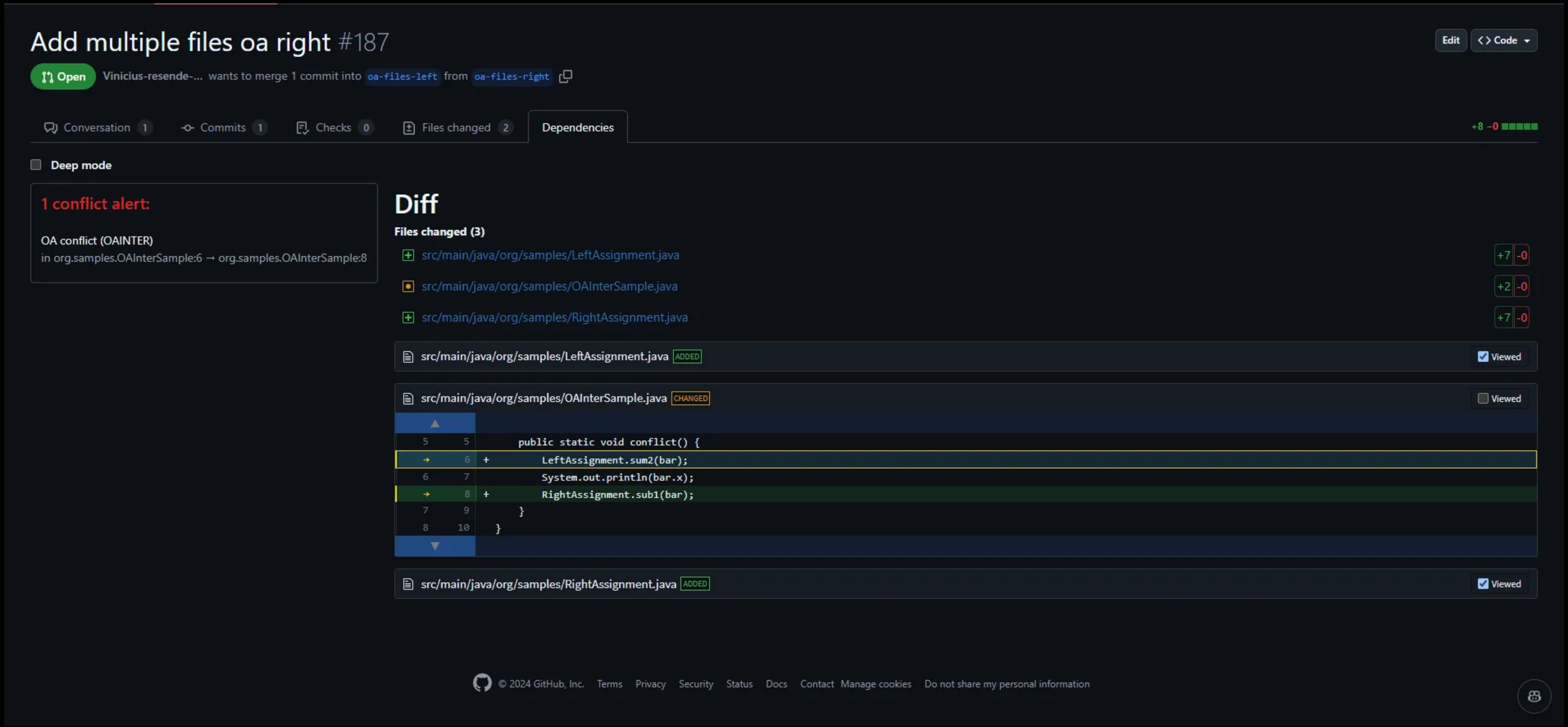

Claus Brabrand, Márcio Ribeiro, Társis Toledo, Paulo Borba (2012). Intraprocedural Dataflow Analysis for Software Product Lines (AOSD 2012).

Eric Bodden, Társis Toledo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, Mira Mezini. SPLlift: Transparent and Efficient Reuse of IFDS-based Static Program Analyses for Software Product Lines (PLDI 2013).

A significant (3x) decrease in code-change effort (time) by emergent interfaces, when faced with interprocedural dependencies

A reduction in errors made during code-change tasks when using emergent interfaces

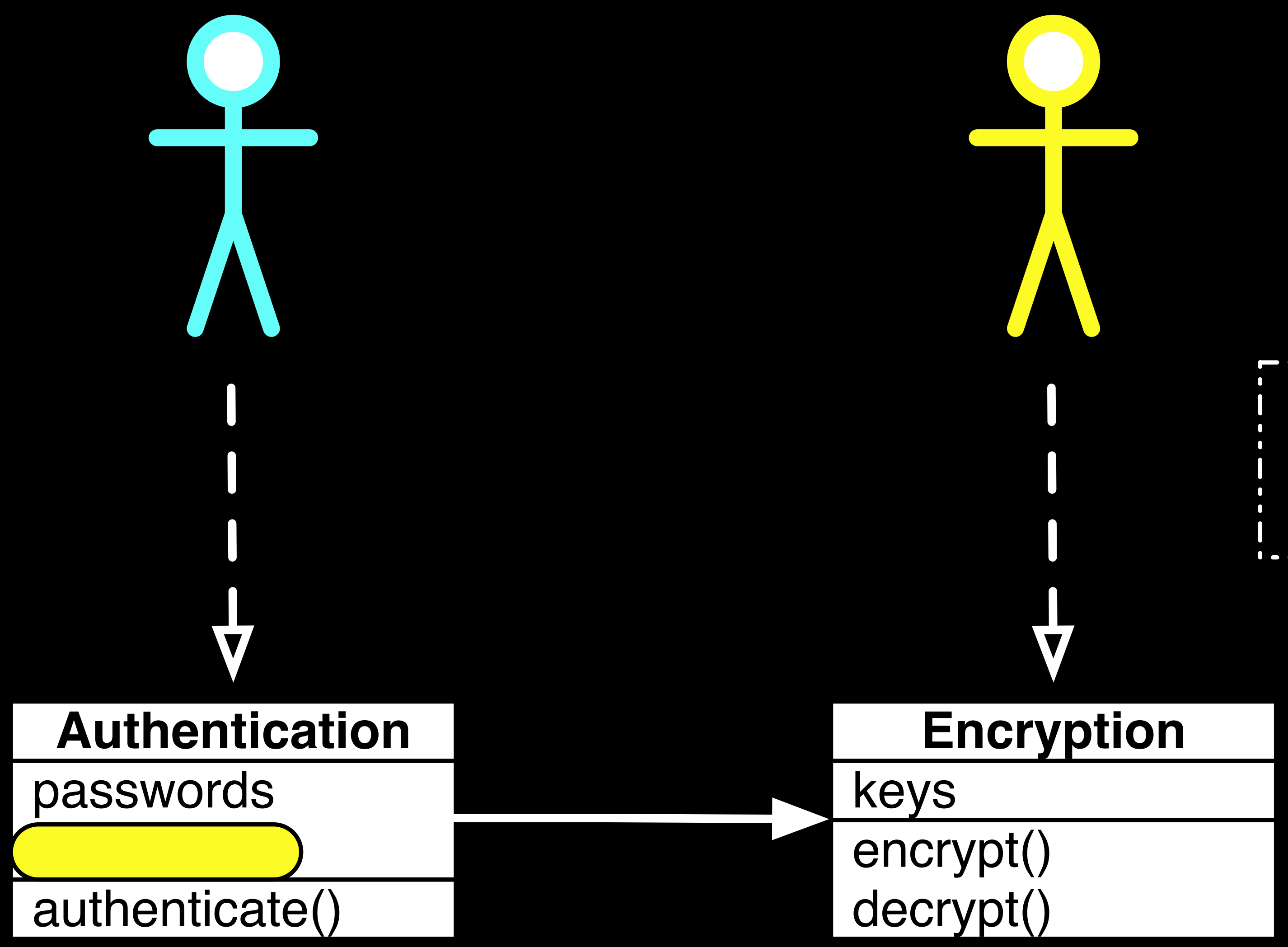
# Visualizing dependencies and interfaces for code contributions



# Inferring, negotiating and managing interfaces

hide non task elements

infer and changes task interfaces




feature F in encrypt() being changed by Blue.
Interface changes: x != null

freeze interface

encrypt() being changed by Blue.
Interface changes: new parameter y

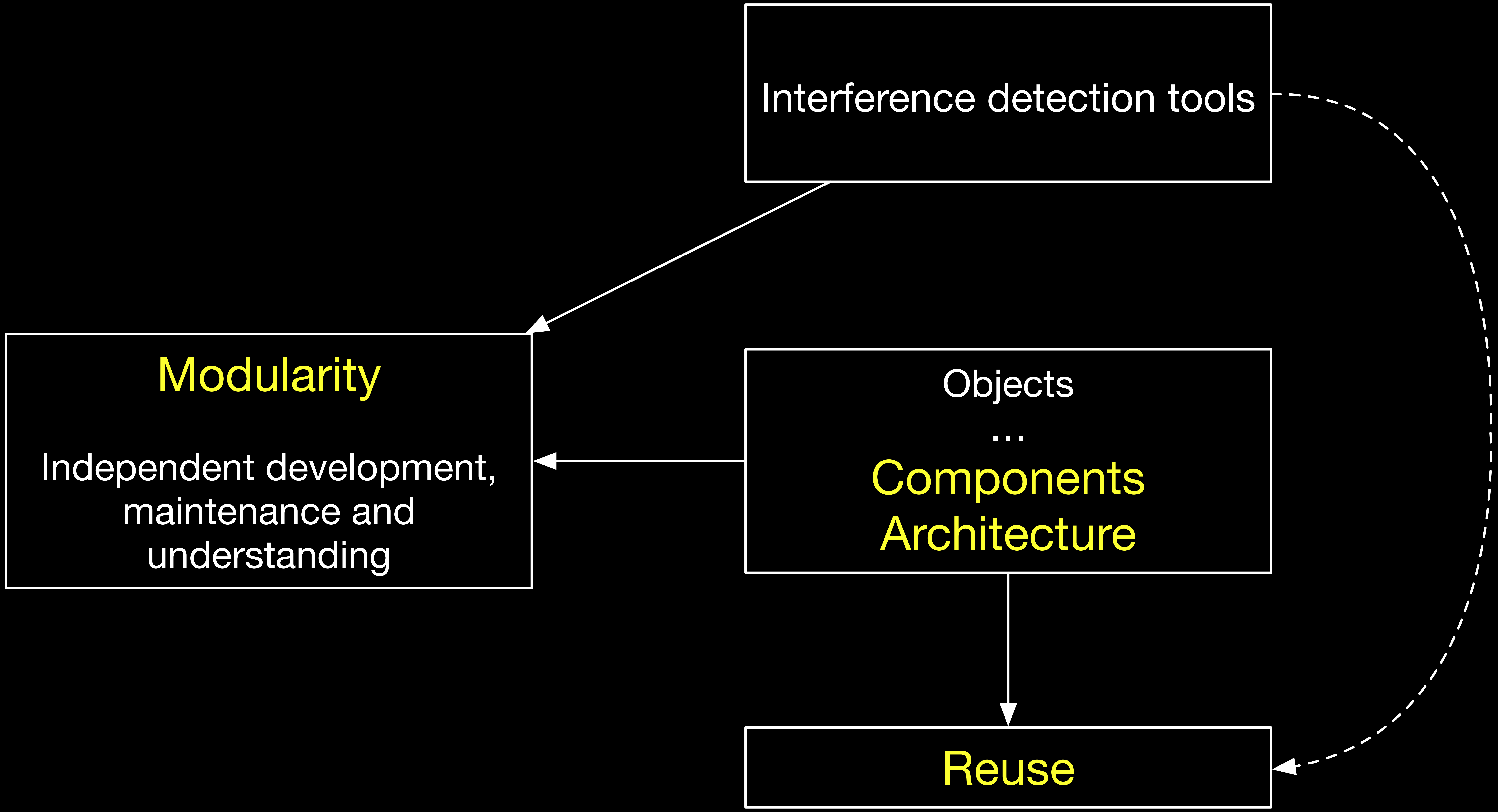
don't care

you are changing encrypt() interface,
impacts interface items: x != null

propose interface

your commit possibly conflicts
with Blue's commit

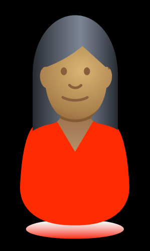
request previous interface


Interference detection tools
Modularity
Independent development, maintenance and understanding
Objects
...
Components
Architecture
Reuse

# Task, non language, modularity

Tools enable modular development and maintenance by supporting the integration of task results

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```



```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    removeDuplicateWords();
  }
}
```

merge

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

~~resulting text has no duplicate whitespace~~

resulting text has no duplicate words

```
Text t = new Text();
t.text = "the␣the␣␣dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```

**FAILS!**

# Feature interaction is a particular case of this problem

```
#ifdef NewFeature
  x++;
  ...
#endif
...
#ifdef AnotherNewFeature
  x--;
  ...
#endif
print(x);
```

testing or
production issues!

# Detecting interference with static analysis


Galileu Santos de Jesus, Paulo Borba, Rodrigo Bonifácio, Matheus Barbosa de Oliveira. Detecting Semantic Conflicts using Static Analysis (arXiv 2024).

Roberto Souto Maior de Barros Filho and Paulo Borba. Using Information Flow to estimate interference between developers same method contributions (arXiv 2024).

Matheus Barbosa, Paulo Borba, Rodrigo Bonifacio, Galileu Santos de Jesus. Semantic conflict detection with overriding assignment analysis (SBES 2022).

# Analyze only the merged program version, annotated with the origin of the changes

```
class Text {
  String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

# Detecting data flow between developers changes

```
class Text {
  String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

rd/wr

rd/wr

a path from a yellow to a red command, or vice-versa, indicates interference risk

# Detecting overriding assignments involving developers changes

```
class Text {
  String text;
  int fixes;
  …
  int countFixes() {
    countDupWhitespace();
    countComments();
    countDupWords();
    return fixes;
  }
}
```

write paths, without intermediate assignments, to a common target indicates interference risk

# Detecting control dependences involving developers changes

```
class Text {
  String text;
  …
  void cleanText() {
    if (text != null &&
        hasWhitespace()) {
      normalizeWhitespace();
      removeDuplicateWords();
    }
  }
}
```

# Detecting interference with testing

Léuson Silva, Paulo Borba, Toni Maciel, Wardah Mahmood, Thorsten Berger, João Moisakis, Aldiberg Gomes, Vinícius Leite). Detecting semantic conflicts with unit tests (JSS 2024).

Léuson Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, João Moisakis. Detecting Semantic Conflicts via Automated Behavior Change Detection (ICSME 2020).

Toni Maciel, Paulo Borba, Léuson Silva, Thaís Burity. Explorando a detecção de conflitos semânticos nas integrações de código em múltiplos métodos (SBES 2024).

# Changed behavior is not preserved

✘

```
Text t = new Text();
t.text = "the␣the␣␣dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```

✔

```
Text t = new Text();
t.text = "the␣the␣␣dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```

✘

```
Text t = new Text();
t.text = "the␣the␣␣dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

Unchanged behavior
is not preserved

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```

✔

```
Text t = new Text();
t.text = "...";
t.cleanText();
assertTrue(...);
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
  }
}
```

✔

```
Text t = new Text();
t.text = "...";
t.cleanText();
assertTrue(...);
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    removeDuplicateWords();
  }
}
```

✔

```
Text t = new Text();
t.text = "...";
t.cleanText();
assertTrue(...);
```

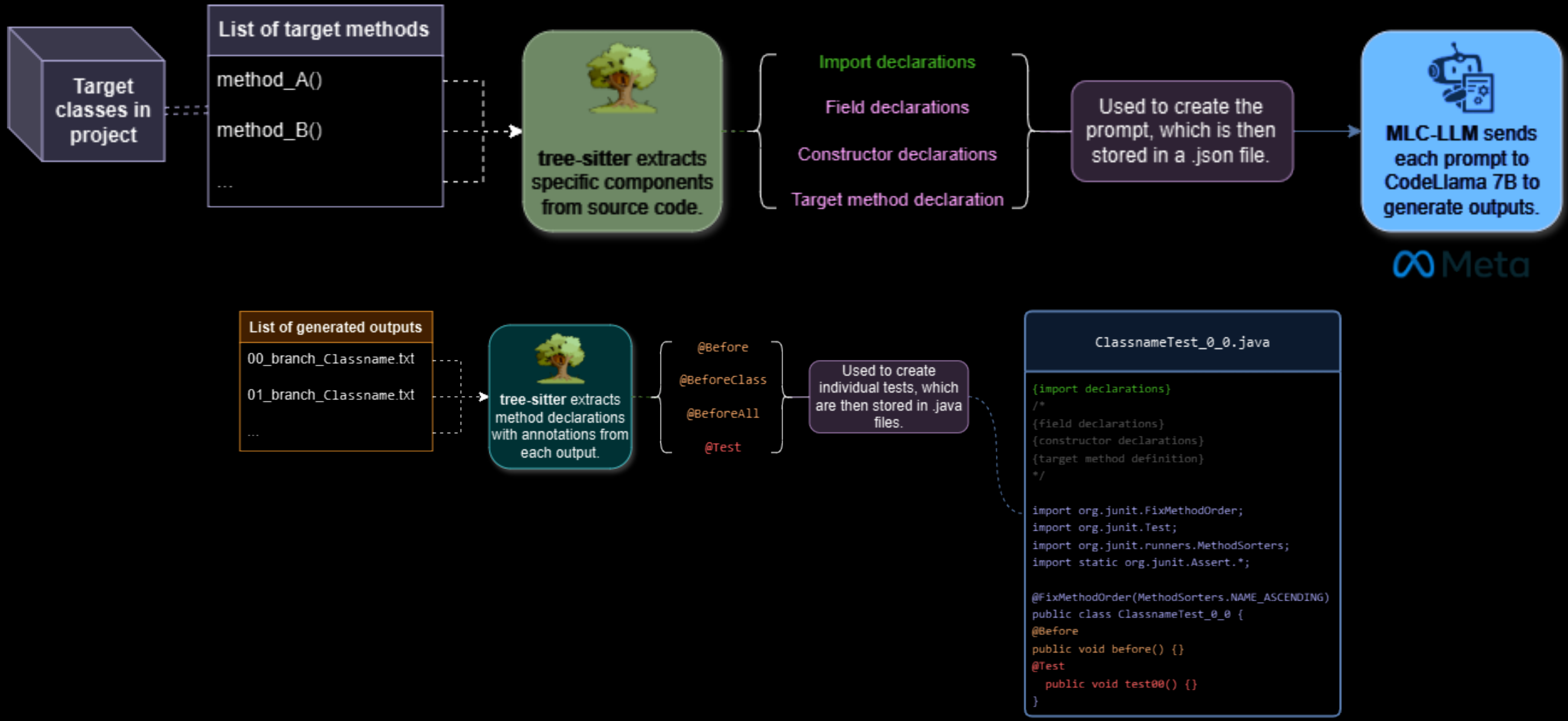
merge

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

✘

```
Text t = new Text();
t.text = ...;
t.cleanText();
assertTrue(...);
```

# Using LLMs to generate unit tests for interference detection

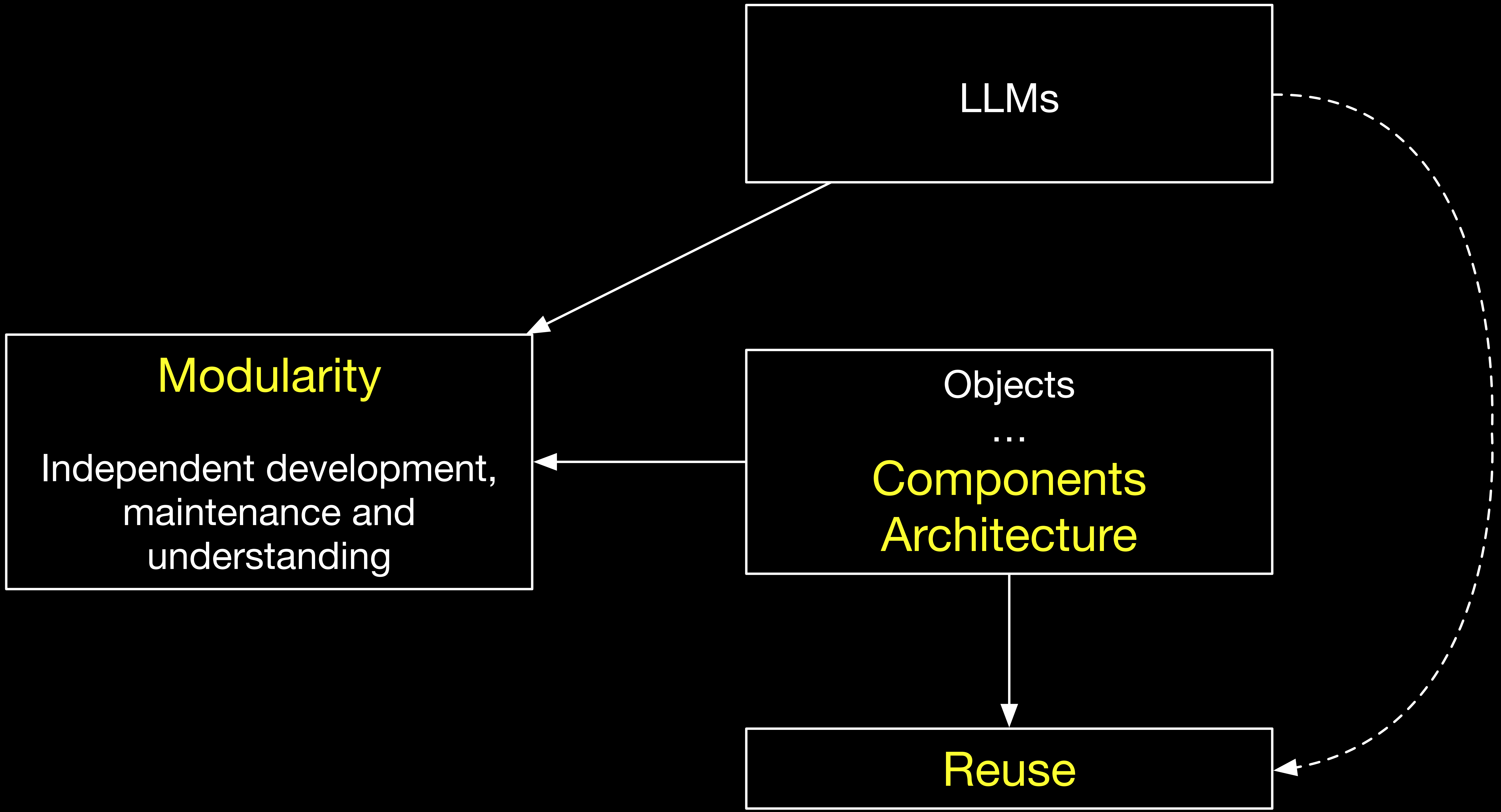


# Interference detection potential to support modularity

| Metrics | Techniques | | | | | | |
|---|---|---|---|---|---|---|---|
| | SA | SA | TA | SA | IF | SA | VA |
| Precision | 0.33 | 0.32 | 0.80 | 0.52 | 0.45 | 0.17 | 0.22 |
| Recall | 0.52 | 0.50 | 0.17 | 0.86 | - | 0.17 | 0.33 |
| F1 Score | 0.40 | 0.39 | 0.28 | 0.65 | - | 0.17 | 0.27 |
| Accuracy | 0.55 | 0.51 | 0.72 | 0.58 | - | 0.67 | 0.63 |
| Units | 99 | 75 | 75 | 31 | 31 | 30 | 30 |

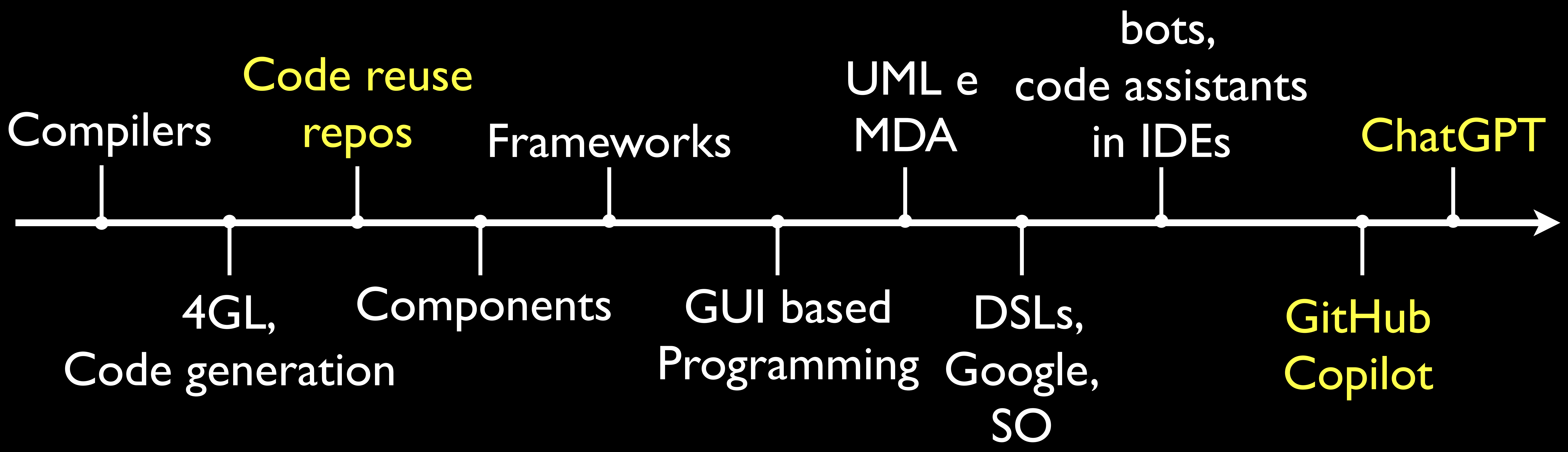
# Detecting interference with dynamic analysis

Amanda Moraes, Paulo Borba, Léuson Da Silva. Semantic conflict detection via dynamic analysis (SBLP 2024).


LLMs
Modularity
Independent development, maintenance and understanding
Objects
...
Components
Architecture
Reuse

LLM’s don’t solve the problem unless you assume a context in which (human) collaborative development is not necessary!

# Code generation and reuse timeline



# Code reuse and LLMs

| Code Reuse Repos | Copilot and Chat GPT |
|---|---|
| Curated limited repo | The repo is the web |
| Specification based structured search | NL based non structured search |
| Functions, components, systems | Code snippets and combinations |

Stack overflow replacement

Generator of repetitive code

Natural language coding

Risk

Creator of APIs, abstractions, modular structures, etc.

Interference detection

(no scientific evidence!)

# Acknowledgments


Galileu Santos


Matheus Barbosa


Victor Lira


Rodrigo Bonifácio


Gabriela Sampaio


Vinicius Barbosa


Roberto Barros


Ykaro dos Santos


Henrique Oliveira


Rafael Alves


Léuson Silva


Toni Maciel


Hugo Cardoso

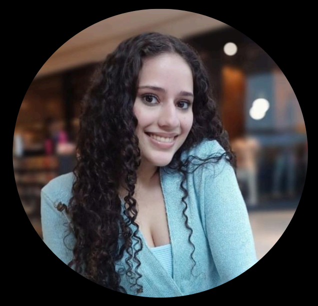

Thorsten Berger


Nathalia Barbosa


João Moisakis


Thaís Burity


Amanda Moraes


Victoria Figueiredo

SPG

ines

# Software Modularity, Components, Architecture, and Reuse: from Parnas to LLMs

Paulo Borba
Informatics Center
Federal University of Pernambuco

pauloborba.cin.ufpe.br

50 anos Centro de Informática UFPE

CBSOFT'25
XVI CONGRESSO BRASILEIRO DE SOFTWARE: TEORIA E PRÁTICA
22 A 26 DE SETEMBRO | RECIFE/PE

SPG

ines