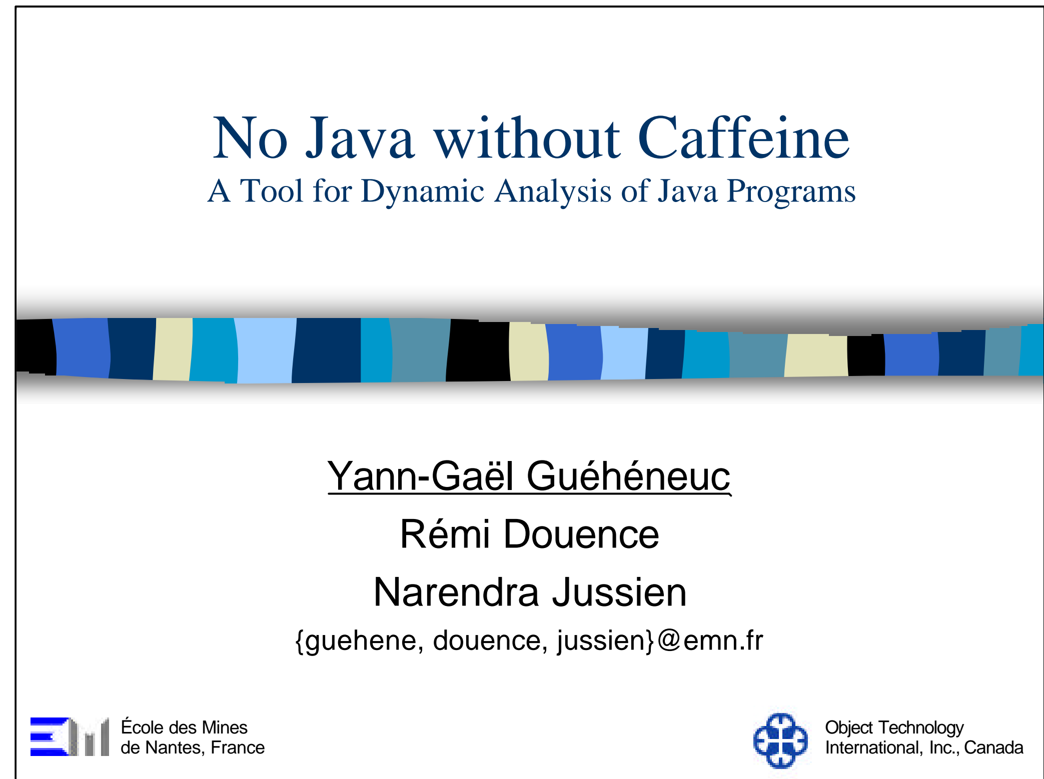


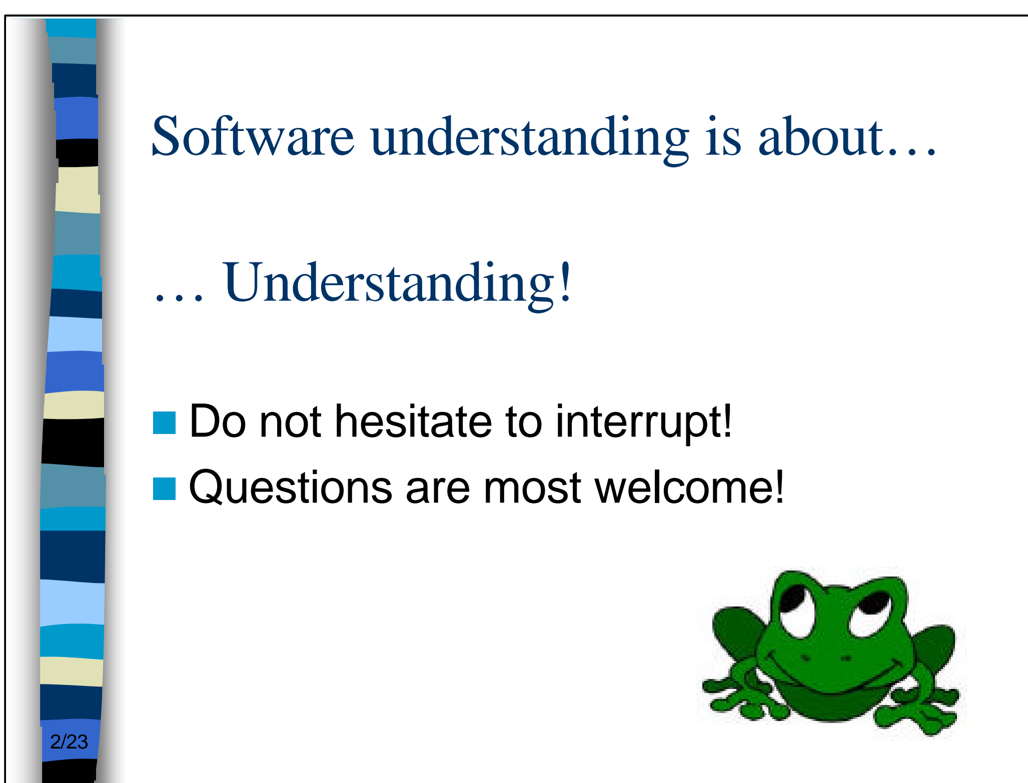
Ladies and gentlemen…

My name is Yann-Gaël Guéhéneuc.

I present today our work on Caffeine, a tool for the dynamic analysis of Java programs.

We develop Caffeine with Rémi Douence and Narendra Jussien from the EMN.

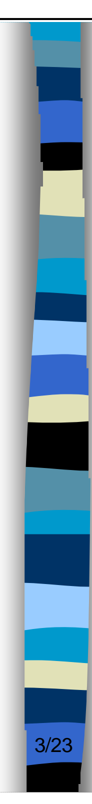
My work is partly funded by Object Technology International, Inc.



Today, we are all attending a session on software understanding.

We should not forget that software understanding is first and foremost about understanding.

Being from France, my accent might be a bit of a problem, so please, do not hesitate to interrupt me, questions are most welcome.



Objectives

- Motivate the need for dynamic analysis
- Present the prototype models
- Detail the prototype implementation
- Describe a more complex example
- Summarize limitations and future work

Today, in my presentation:

First, I motivate the need for dynamic analysis, using on a very simple example: The analysis of a n-queens algorithm.

Second, I present the models used in our prototype.

Third, I detail our prototype implementation. I put the emphasis on issues and limitations.

Fourth, I introduce a real-life example that (I hope) highlights the usefulness of dynamic analyses and presents the integration of our tool with other tools for software understanding.

Finally, I summarize the usefulness and the limitations of our tools and present some future work.

Motivation

- Software understanding: [Soloway *et al.*, 1988]
  - Code structure
  - Conjecture
- Assistant
- Dynamic behaviour of Java programs
- Least modifications (source code, JVM)

Our motivation is software understanding.

Soloway *et al.* characterized software understanding as composed of inquiry episodes.

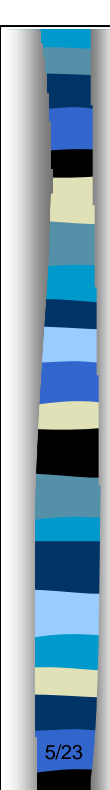
The maintainer, who tries to understand a program, first reads some code, asks a question about this code, conjectures an answers, and searches the code and the documentation for a confirmation of the conjecture.

Our goal is to develop an assistant which the maintainer uses to verify conjectures on the runtime behaviour of the program with respect to the program structure.

We want our assistant to work with Java.

We need to analyze the dynamic behaviour of Java programs.

(I hope that I will convince you of the need for dynamic analyses in Java, in contrast with the results presented in the Nokia experience report yesterday…)



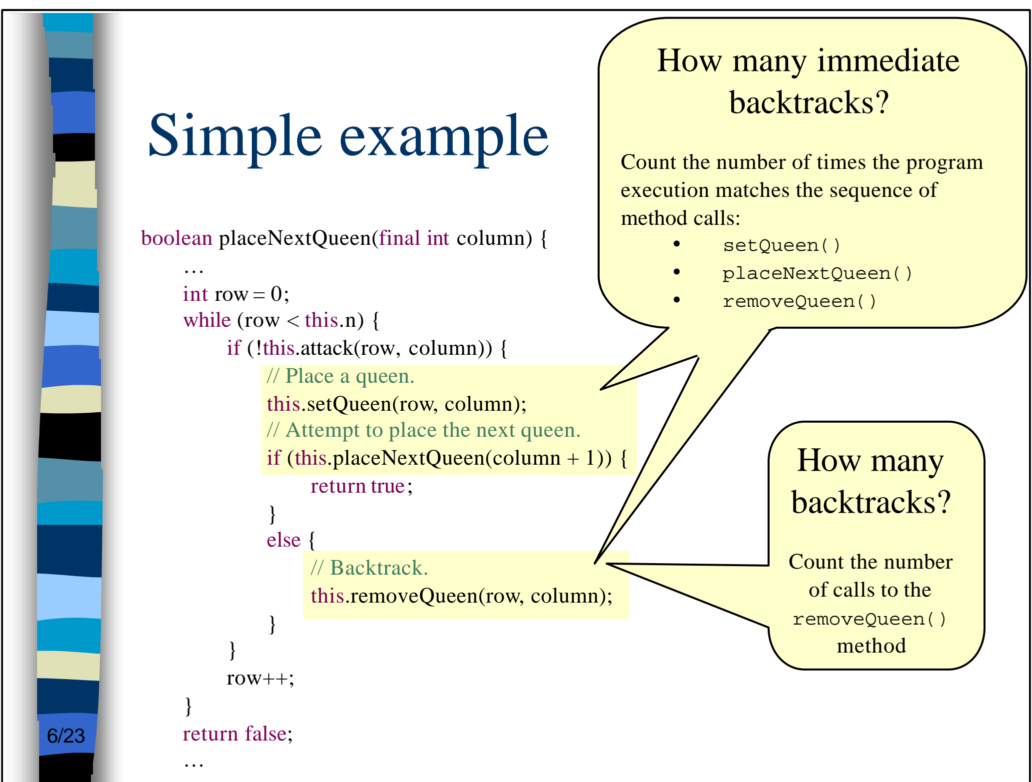
Simple example (1/2)

- The n-queens problem:
  - A $n \times n$ chessboard
  - n queens
  - An algorithm to place the n queens

Take for example a simple Java program to solve the n-queens problem.

This program declares a board of $n \times n$ cells and puts n queens on that board using a simple algorithm with backtrack.

This is a very simple example… I present a more complex example at the end of this presentation.

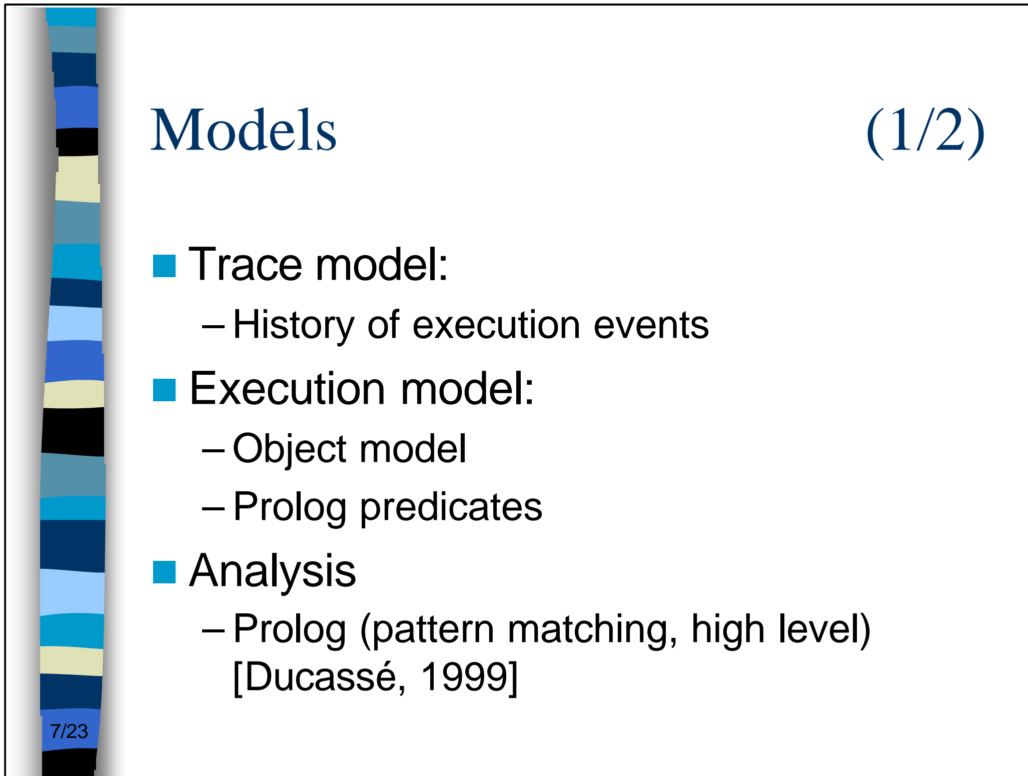


The algorithm is presented here.

Now, let's imagine the maintainer who gets that program wants to know the number of time the algorithm backtracks and the number of time the algorithm does an immediate backtracks, i.e., remove a queen just after having placed her.

The maintainer needs to analyze the behaviour of the program…

We could then replace immediate backtracks with, for example, a propagation algorithm.



To perform the dynamic analysis of Java programs we need both a trace model, which model the process of program execution, and an execution model, which models what happens during the execution.

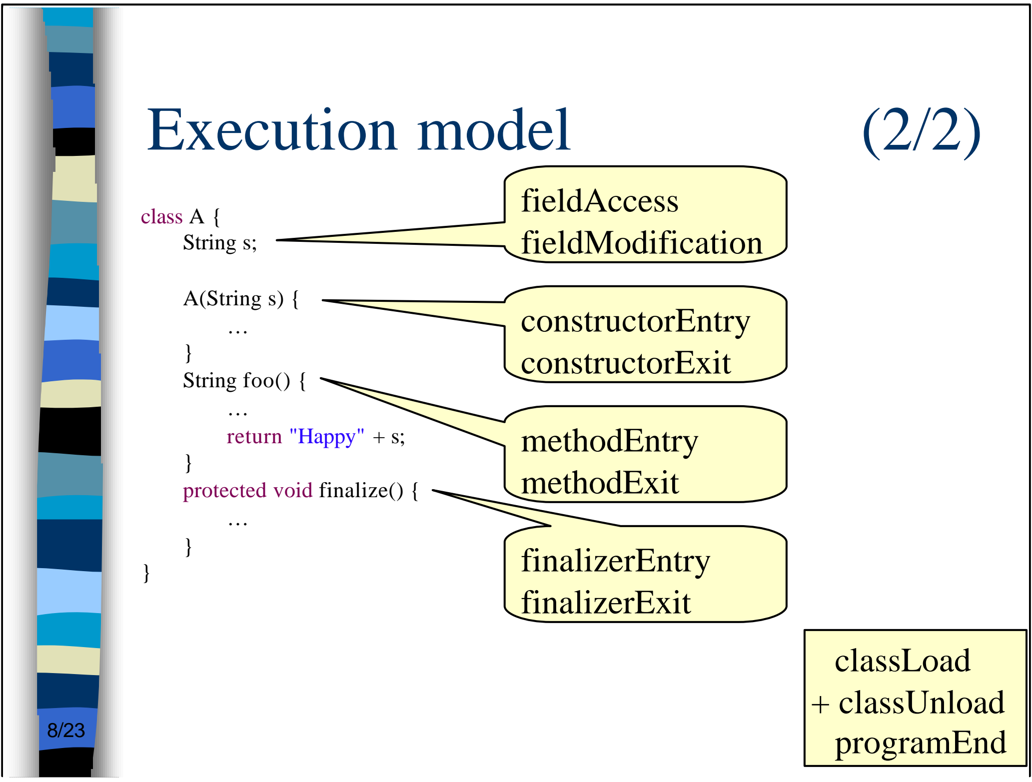
A trace is an history of execution events. We assume that the trace is built on the fly and that there is not storage mechanism (but it is possible to store execution events).

Our execution model is based on the object model of the Java programming language. This means that we have execution events related to operations performed on and by classes and their instances.

We describe these execution events as Prolog predicates.

We use Prolog to perform analyses on traces of program executions.

We chose Prolog because it is powerful, it has pattern matching capabilities, and we can express queries in a quite natural way. Prolog has already been chosen to perform trace analysis: For example in Mireille Ducassé's work on Opium and Coca.



This list presents the different execution events defined by our execution model.

Remember that these execution events are modelled as Prolog predicates.

We have execution events for almost everything that can happens to classes and their instances.

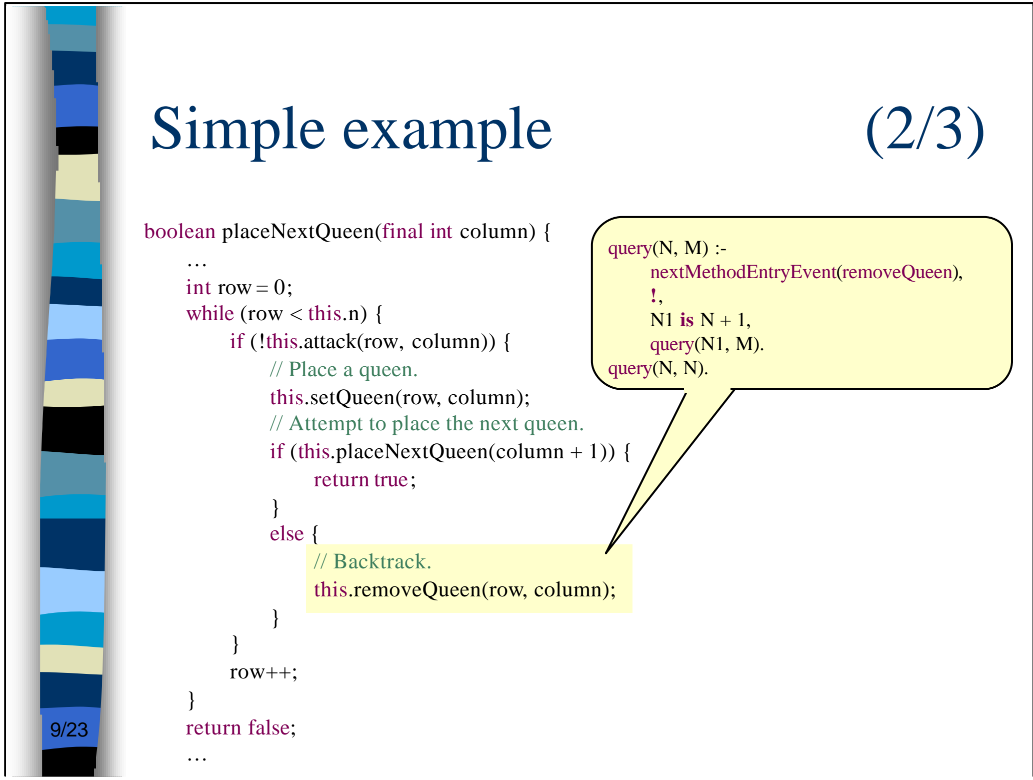
We have field accesses and modification…

We have class load and unload…

We have…

We have method exit execution event, including the returned value.

We have…



So, let's get back to our simple example.

Remember we wanted to count the number of backtracks?

We write the following Prolog query on the program execution.

The `nextMethodEntryEvent/1` predicate requests the program to run until method `removeQueen()` is called. When the `removeQueen()` method is called, the program execution stops and the control flow goes to the Prolog engine, which then increases a counter and loops.

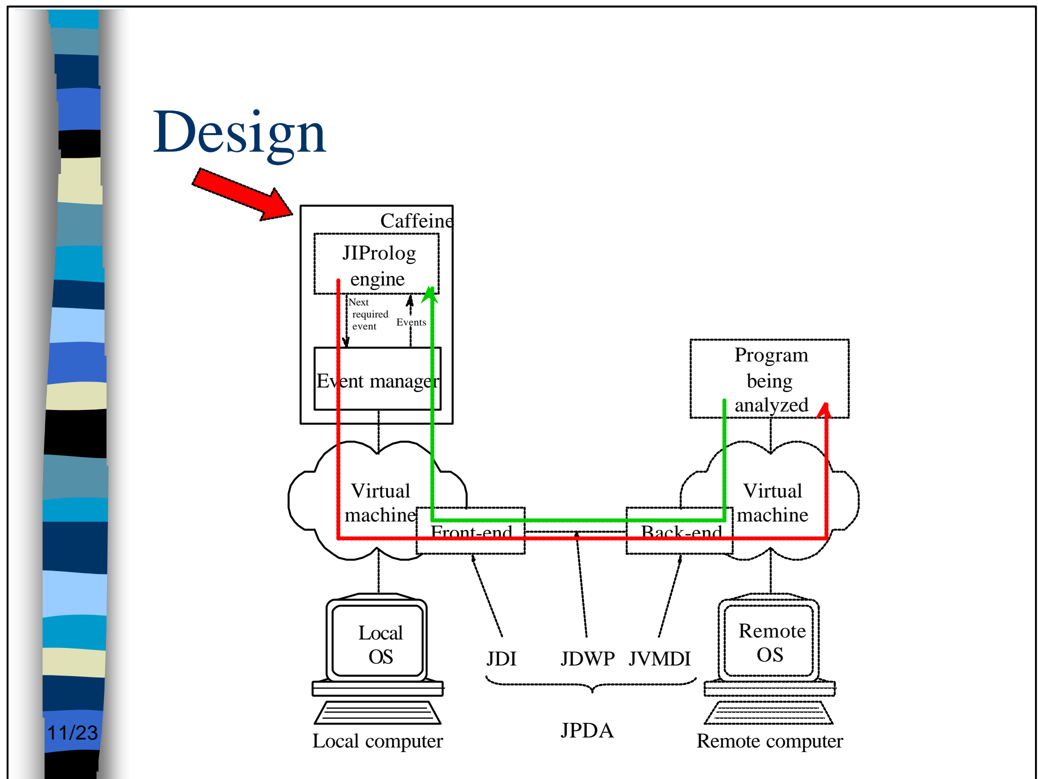
Simple…

```
query(N, M) :-
        nextMethodEntryEvent(setQueen),
        nextPlaceNextQueen,
        nextMethodEntryEvent(removeQueen),
        !, N1 is N + 1, query(N1, M).
query(N, N).
nextPlaceNextQueen :-
        nextMethodEntryEvent(NAME),
        isPlaceNextQueen(NAME).
isPlaceNextQueen(placeNextQueen)        :- !.
isPlaceNextQueen(removeQueen)           :- !, fail.
isPlaceNextQueen(setQueen)              :- !, fail.
isPlaceNextQueen(_)                     :- nextPlaceNextQueen.
```

```
boolean placeNextQ…
    …
    int row = 0;
    while (row < th…
        if (!this.atta…
            // Place a queen.
            this.setQueen(row, column);
            // Attempt to place the next queen.
            if (this.placeNextQueen(column + 1)) {
                return true;
            }
            else {
                // Backtrack.
                this.removeQueen(row, column);
            }
        }
        row++;
    }
    return false;
    …
```


We also wanted to count the number of immediate backtracks.

We write the following query, base on the same principles as for counting backtracks. The query is a bit more complicated and I do not detail it here, but you can find in the paper the details that prove that we obtain the number of immediate backtracks quite nicely.



Okay, so far I gave you a flavour of what our tool, Caffeine, is all about.

I now detail the tool implementation of our tool.

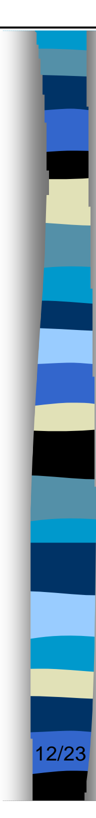
Basically, we have a Prolog engine, JIProlog, which runs as a co-routine of the program being analyzed.

The Prolog engine controls and receives execution events from the program through the Java Debug Interface (JDI) of the Java Debug Platform Architecture (JPDA).

We chose to use the JPDA because it seems quite natural to use the debug architecture to analyze, programmatically though, program executions; It alsoreduces the need for modifications of both the source code and the JVM.

This is quite simple to implement at first sight.

However, some issues quickly show up. In particular with the following five points.



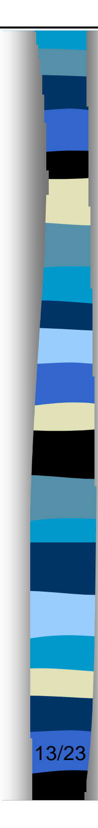
Implementation

- Issues:
  - Returned values
    - Parameter values
    - Instrument `return` statements
      - `return CaffeineWrapper(<previously returned value>);`
  - Finalizers
    - `System.runFinalizersOnExit(boolean)`
    - Instrument class files to add `finalize()` methods and to call the GC

Here are implementation techniques we use to overcome the limitations of the JDI.

To obtain returned value, we instrument the class files at load-time, to replace any return statement by a return statement with a call to a special method with a single parameter (the returned value), which in turn we can monitor to obtain its parameter and thus the returned value.

To obtain execution events related to finalizers, we use the deprecated method `System.runFinalizersOnExit(boolean)`. We also add `finalize()` method to all loaded class and add calls to the garbage collector. All these manipulations dramatically slow down the program execution.



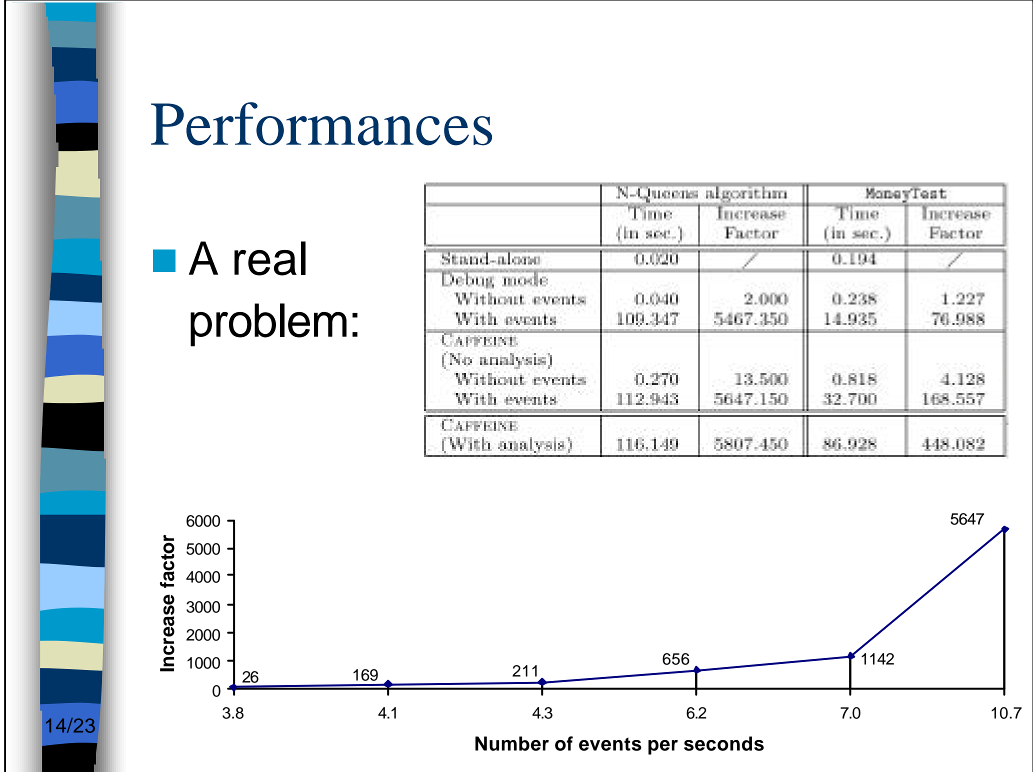
Implementation

- Issues:
  - Program end
    - Instrument `System.exit(int)`
    - Count user thread starts/ends
      - Assumption about the thread group
  - Unique identifiers
    - Instrument hierarchy root

To obtain program ends, we need to monitor calls to the `System.exit(int)` method, which we replace by our own wrapper method. We also count the number of user threads that start and terminate. We need to count user threads only because the JVM starts its own threads.

To obtain unique identifiers, we instrument the loaded class to insert our own super-class which holds a unique identifier for each class and instances created.

All these tricks solve most of the problem we met when developing Caffeine. It's all fun and good. However, a last issue remains, and this issue is of most importance.



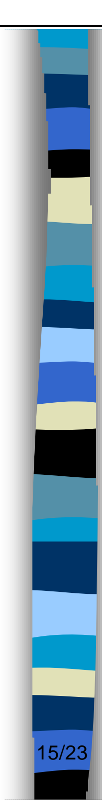
	N-Queens algorithm		MoneyTest	
	Time (in sec.)	Increase Factor	Time (in sec.)	Increase Factor
Stand-alone	0.020	/	0.194	/
Debug mode				
Without events	0.040	2.000	0.238	1.227
With events	109.347	5467.350	14.935	76.988
CAFFEINE (No analysis)				
Without events	0.270	13.500	0.818	4.128
With events	112.943	5647.150	32.700	168.557
CAFFEINE (With analysis)	116.149	5807.450	86.928	448.082

The performance of our tool is way to low!

On this table, for two different programs, we show that the performances are dramatically decreased. The main slow down factor is the event generation.

We perform some measurements, and we notice a exponential increase of execution time according to the number of execution events. That sounds reasonable, nothing comes free. However, the increase is of many orders of magnitude.

This performance issue is very bad and impedes the use of our analysis tool.



More complex example

- Object-oriented programs:
  – Classes (and their instances)
  – Relationships among classes (and among their instances)
- Two binary class relationships:
  – Aggregation: [A ◇— B] (structural)
  – Composition: [A ◆— B] (behavioural)

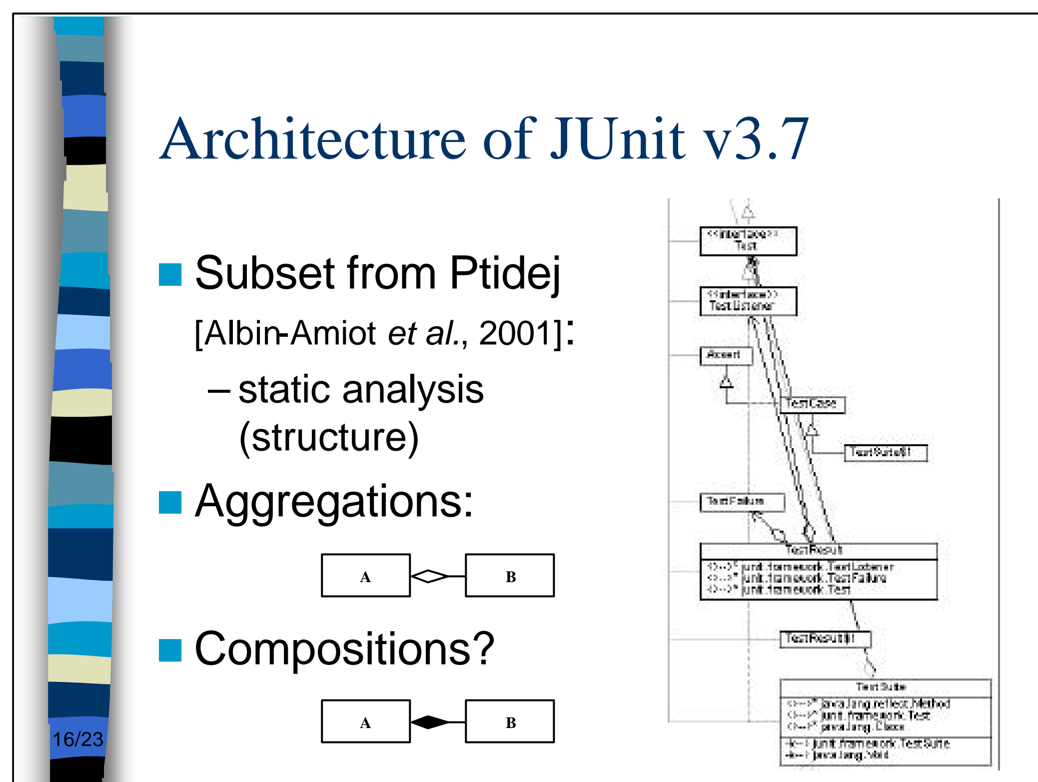
I present now a real-life example of software understanding.

We all know that object-oriented programs are about classes (and their instances) and about relationships among classes (and among instances).

There are two *famous* binary class relationships: The aggregation and the composition relationships. There are no consensual definitions of those relationships, so I give my own:

-An aggregations relationship between two classes `A` and `B` expresses that class `A` declares a field (or a collection) of instances of class `B`.

-A composition relationship between two classes `A` and `B` expresses that an aggregation relationship links those two classes and that the lifetime of instances of `B` is shorter that (and exclusive to) the lifetime of instances of `A`.



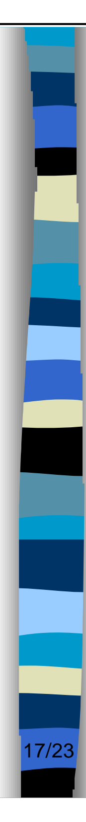
Let’s consider the following subset of JUnit v3.7 architecture.

We extract this architecture using our tool Ptidej and a static (structural) analysis.

We presented Ptidej last year at ASE.

We notice there are some aggregation relationships found.

We would like to have more information about this architecture. In particular, we would like to know if there are any composition relationships among classes of the JUnit.



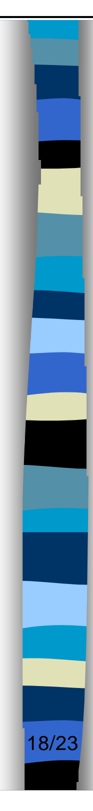
Composition relationship

- Static (structure):
  - `A` declares a collection (field) of `B`
  - `A` declares handling methods
- Dynamic (behaviour):
  - Instances of `B` must belong **exclusively** to instances of `A` (*exclusivity*)
  - Lifetime of instances of `B` must **not exceed** the lifetime of instances of `A` (*lifetime*)

A composition relationship has two aspects:

-A structure (static), a field (a collection, generally) and some methods to handle adding and removing of instances of `B`.

-A behaviour (dynamic), instances of…

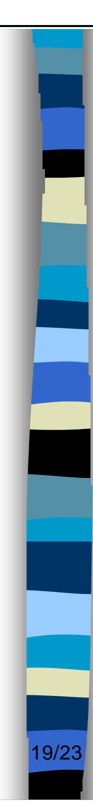


Execution events

- Assignations of instances of `B` to any field in class `A`
- Assignation of instances of `B` to any collection in class `A`
- Finalization of instances of `B` and `A`

Using Caffeine, our dynamic analysis tool, we can monitor assignations of instances of `B` in any field or collection of class `A`.

We can also monitor the finalizations of instances of `A` and `B` and their ordering.

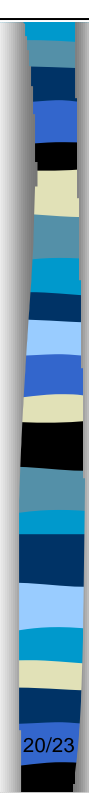


Composition analysis

- About 50 predicates:
  - About 300 lines of Prolog
  - Mostly lists handling
  - Complexity n^2
- Analysis written once and for all
  - Exclusivity property
  - Lifetime property

We wrote Prolog predicates to monitor and analyze the composition relationships between classes. It is not that complex, the Prolog file is about 50 predicates, 300 lines of Prolog, and is mostly about list handling.

This analysis is written once and for all, it can be reused for other Java programs. It decomposes into the two properties Exclusivity and Lifetime.

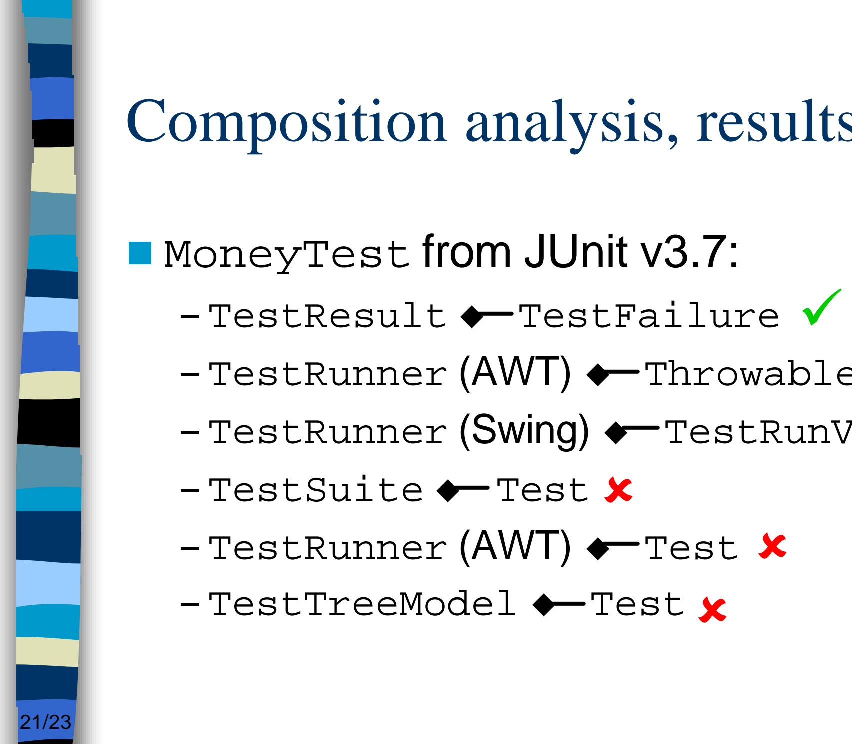


Composition analysis, example

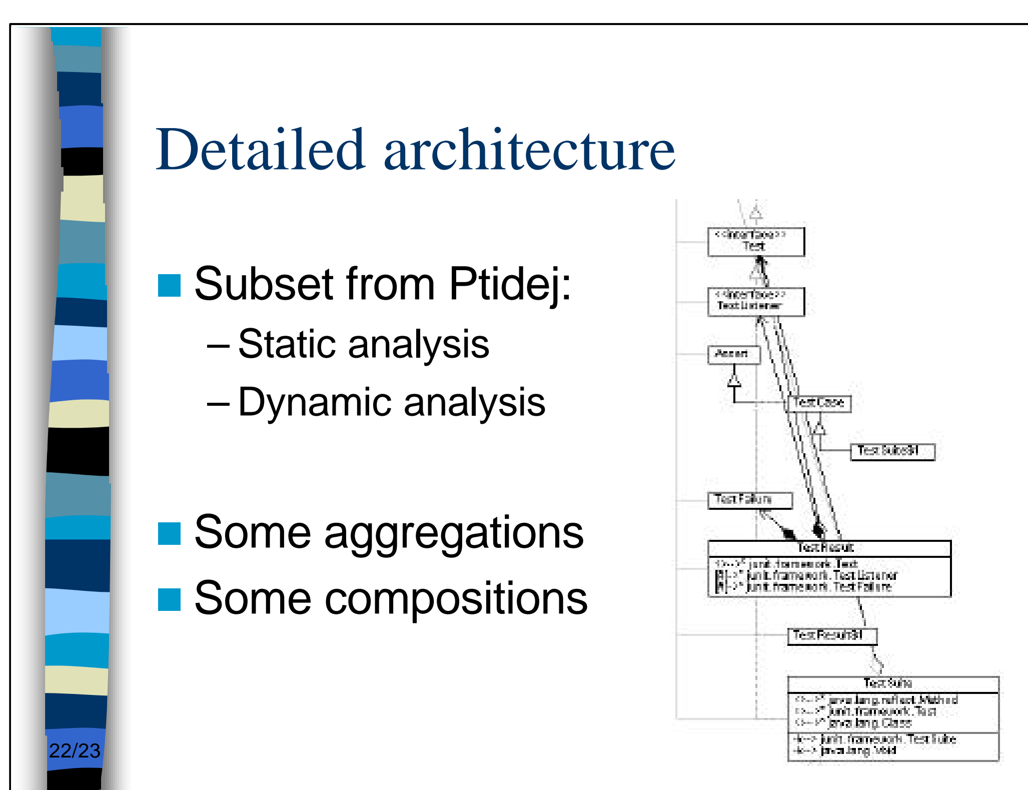
- `MoneyTest` from JUnit v3.7:
  - `TestResult` ◆— `TestFailure`?
  - `TestRunner` (AWT) ◆— `Throwable`?
  - `TestRunner` (Swing) ◆— `TestRunView`?
  - `TestSuite` ◆— `Test`?
  - `TestRunner` (AWT) ◆— `Test`?
  - `TestTreeModel` ◆— `Test`?

We apply Caffeine and our composition relationship analysis on JUnit v3.7, using the provided `MoneyTest` class.

We monitored the following classes because they all have aggregation relationships.

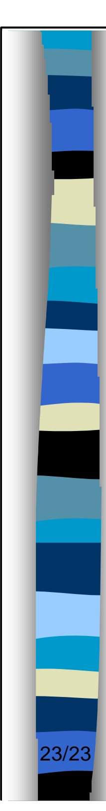


The results are as follow…



Finally, from the static and dynamic analyses, we can improve our understanding of the architecture of JUnit.

We have now a better understanding of JUnit architecture. On one hand, we know that the `TestResult` and `TestFailure` classes are tightly coupled, through a composition relationships: Instances of class `TestFailure` live and die with instances of class `TestResult`. On the other hand, instances of class `Test` are shared among several classes and have lives on their own.



Conclusion and future work

- Useful!
- Limitations:
  - Performances
  - Code coverage
- Future work:
  - Implementation rework
  - More analyses
  - Runtime description language

To conclude, dynamic analysis is a useful thing to have and to do!

I presented Caffeine, a tool for the dynamic analysis of Java programs.

I gave some examples of our use of dynamic analyses.

However, our tool, as it is, is too limited with respect to performances and to code coverage analysis.

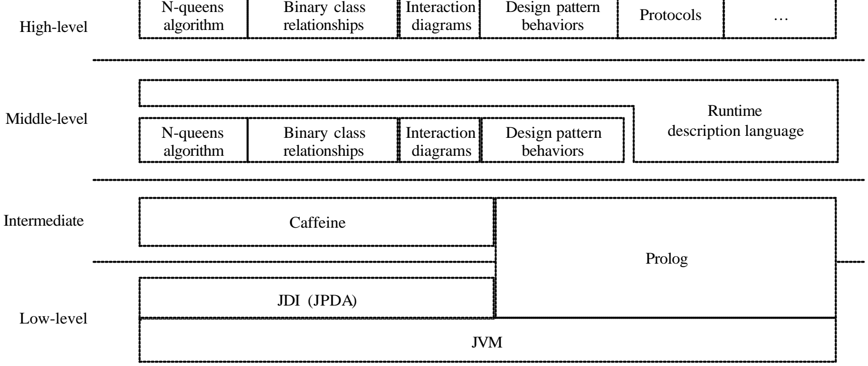
Future work includes:

-Implementation rework, we would like to obtain only interesting events, no all events of a certain kind, and we would like to remove the dependency on the debug architecture that decreases performances.

-We also want to add more analyses to the current library. In particular, we want to develop more properties.

-We plan to implement a *runtime description language* to express properties and to compose easily properties into complete analyses. We consider using temporal logic or logic programming.

Future work


High-level
N-queens algorithm
Binary class relationships
Interaction diagrams
Design pattern behaviors
Protocols
…
Middle-level
N-queens algorithm
Binary class relationships
Interaction diagrams
Design pattern behaviors
Runtime description language
Intermediate
Caffeine
Prolog
Low-level
JDI (JPDA)
JVM

Exclusivity property

```
checkEXProperty(
    assignation(_, A, AID, B, BID),
    LIST,
    NLIST) :-
    updateEXProperties(A, AID, B, BID, LIST, NLIST, true).
checkEXProperty(_, LIST, LIST).

updateEXProperties(
    A, AID, B, BID,
    [],
    [exclusivityProperty(A, AID, B, BID, true)],
    true).
updateEXProperties(
    A, AID, B, BID,
    [],
    [exclusivityProperty(A, AID, B, BID, false)],
    false).
…
```

Lifetime property

```
checkLTProperty(
    assignation(_, A, AID, B, BID),
    LIST,
    NLIST) :-
          append(LIST, [pendingAssignation(A, AID, B, BID, [])], NLIST).
checkLTProperty(
    finalization(EID, A, AID),
    LIST,
    NLIST) :-
    doesLTPropertyHold(finalization(EID, A, AID), LIST, NLIST).
checkLTProperty(
    programEnd(_),
    LIST,
    NLIST) :-
    convertPendingAssignations(LIST , NLIST).
checkLTProperty(_, LIST, LIST).
```

Composition analysis

```
checkComposition([], [], []).
checkComposition([], _, []) :-
    write('Problem! Lists of properties have different size!'), nl, false.
checkComposition(_, [], []) :-
    write('Problem! Lists of properties have different size!'), nl, false.
checkComposition(
    [exclusivityProperty(A, AID, B, BID, OKAY) | EXPREST],
    [lifetimeProperty(A, AID, B, BID, OKAY) | LTPREST],
    [composition(A, AID, B, BID, OKAY) | NLIST]) :-
        checkComposition(EXPREST, LTPREST, NLIST).
checkComposition(
    [exclusivityProperty(A, AID, B, BID, _) | EXPREST],
    [lifetimeProperty(A, AID, B, BID, _) | LTPREST],
    [composition(A, AID, B, BID, false) | NLIST]) :-
        checkComposition(EXPREST, LTPREST, NLIST).
```