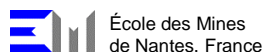


Three musketeers to the rescue

Meta-modelling, Logic Programming, and Explanation-based
Constraint Programming for Pattern Description and Detection

Yann-Gaël Guéhéneuc
guehene@emn.fr



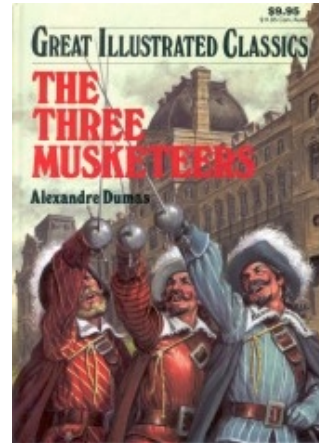
Hello, my name is ... from the École des Mines de Nantes.

My work is partly funded by OTI.

Today, I present three distinct meta-declarative programming techniques we develop and use for our research on patterns.

The musketeers were French...

- Do not hesitate to interrupt!
- Questions are most welcome!



2/18

Well, as the Three Musketeers, I am also French.

My accent might be a bit of a problem...

Do not hesitate to interrupt me and to ask questions!



Objectives of this presentation

- Present our use of three declarative meta-programming techniques
 - Meta-modelling
 - Meta-logic programming
 - Explanation-based constraint programming
 - Context, example, advantages/drawbacks
- Discuss these techniques, their limitations, their use in other contexts

In our research, we use three declarative meta-programming techniques :

-Meta-modelling.

-Meta-logic programming.

-Explanation-based constraint programming.

We use these techniques for the description, application, and detection of design patterns and design defects.

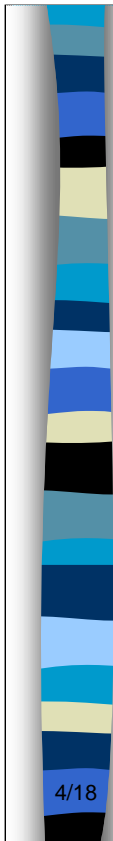
First, I introduce the context of our research.

Then, I present each of these declarative meta-programming techniques.

I also give an example and the reasons why it is a declarative meta-programming technique.

I also discuss some of its advantages and drawbacks.

Finally, I wish to discuss with you these techniques, their limitations, other better techniques, other possible applications...



Context of our research

- Design patterns
 - ⇒ Design defects
 - ⇒ Patterns
- Patterns description
- Pattern application
(design patterns only! 😊)
- Pattern detection

We work on design patterns and design defects.

Design patterns are “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” ([Gamma, 1994], page 3).

Design patterns describe a problem, a solution, and the consequences when applying the solutions.

The solution of a design pattern is generally represented with OMT diagrams and source code examples; That is the solution of a design pattern is generally represented with a micro-architecture exemplifying the solution.

We argue that:

-Design defects are similar to design patterns and we can represent them as design patterns.

-A micro-architecture close but not identical to the solution of a design pattern represent a possible design defects.

We call design patterns and design defect “patterns”.

We want to describe patterns (the micro-architectures representing the solutions of design patterns, or representing design defects).

We want to apply design patterns from descriptions of design patterns.

We want to detect complete forms of design patterns, distorted forms of design patterns, and design defects.



Meta-modelling

(1/4)

- **Pattern Description Language (PDL):**
 - A meta-model
 - Abstract models of patterns
- **An abstract model is a first class entity:**
 - We can parameterize it (PatternsBox)
 - It can generate its corresponding code
 - It can detect complete micro-architectures that are similar to itself

We develop Pattern Description Language (PDL), a meta-model to describe abstract models of patterns.

An abstract model is declared using constituents of PDL.

An abstract model represents the micro-architecture solution of a pattern.

We can parameterize an abstract model to obtain a concrete model of a pattern, using for example PatternsBox.

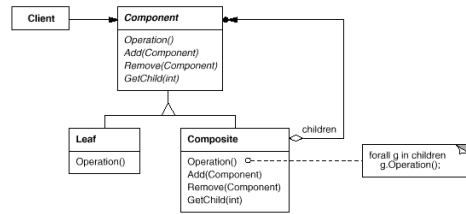
We can ask an abstract model (or a concrete model) to generate itself, for example as Java source code, as constraint system, or as an input for other tools.

We can ask an abstract model to detect itself in a given Java program architecture (source code/class files).

When an abstract model detects itself, we obtain concrete models of the pattern.

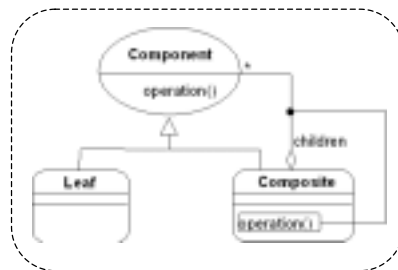
Meta-modelling







(2/4)



Informal descriptions from [Gamma, 1994]

Translates into



-  Instance of Pattern
-  Instance of Interface
-  Instance of Class
-  Instance of Association
-  Instance of Delegation
-  Instance of Method

6/18

Here is a simple example of an abstract model.

This abstract model describes the Composite design pattern.

We describe the micro-architecture solution advocated by the Composite design pattern and the informal information using constituents of the meta-model.

Meta-modelling

(3/4)

```
public Composite() throws
CloneNotSupportedException,
PatternDeclarationException {
    super();

    final Class aClass;
    final Interface anInterface;
    final DelegatingMethod aDelegatingMethod;
    final Method abMethod;

    // Interface Component.
    anInterface = new Interface("Component");
    this.addEntity(anInterface);
    abMethod = new Method("Operation");
    anInterface.addElement(abMethod);

    // Association children.
    final Composition aComposition =
    new Composition("children", anInterface, 2);
}

// Class Composite.
aClass = new Class("Composite");
this.addEntity(aClass);
aClass.addImplementedEntity(anInterface);
aClass.addElement(aComposition);
aDelegatingMethod = new DelegatingMethod(
    "Operation",
    aComposition);
aDelegatingMethod.attachTo(abMethod);
aClass.addElement(aDelegatingMethod);

// Class Leaf.
this.addLeaf(new String[] { "Leaf" });
}
```

Here is the source code describing the Composite design pattern.

It is declarative because each abstract model is declared by a sequence of statements.

It is meta because the statements are instances of constituents of the meta-model.

It is programming because we use the abstract model programmatically to reason about the design pattern solution.



Meta-modelling

(4/4)

■ Advantages:

- Patterns are first-class entities
- Patterns embody all the logic

■ Drawbacks:

- The meta-model only describes one perspective on patterns

Advantages:

-Patterns are first-class entities.

-Patterns descriptions embody all the logic related to patterns (application, detection).

Drawbacks:

-The meta-model only describes one perspective on patterns. For example, PDL only describes the structure of patterns (and some behavioural information).



Meta-logic programming (1/4)

- Caffeine is a program for trace analysis of Java programs
- Caffeine is based on:
 - A model of trace
 - A model of execution events that abstracts the program execution
 - The JPDA
 - A Prolog engine (JIProlog)

Caffeine is a tool for the analysis of Java programs.

It defines a model of trace: A trace is a history of execution events.

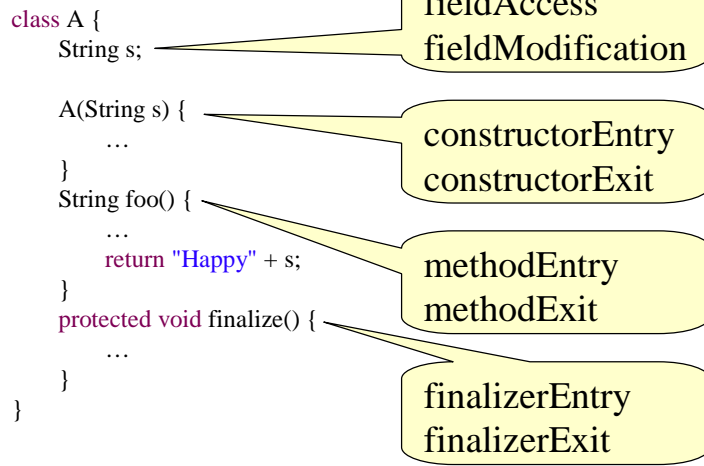
It defines a model of execution events: Execution events are emitted by the program being analyzed and abstract the program behaviour.

It is a program that runs as a co-routine of a program to analyze.

It drives the program being analyzed using the JPDA (Java Platform Debug Architecture).

It receives events from the program being analyzed as Prolog predicates and can reflect on these events.

Meta-logic programming (2/4)



classLoad
+ classUnload
programEnd

Here is the list of execution events that can be emitted by the program being analyzed.

All these execution events might not be of interest, so we can filter out the events we are not interested in.

Filtering out events improves slightly the performances of the program being analyzed.

Meta-logic programming (3/4)

```
nextPlaceNextQueen :-
    nextMethodEntryEvent(N),
    isPlaceNextQueen(N).

isPlaceNextQueen(placeNextQueen) :- !.
isPlaceNextQueen(removeQueen)   :- !, fail.
isPlaceNextQueen(setQueen)       :- !, fail.
isPlaceNextQueen(_)              :- nextPlaceNextQueen.

query(N, M) :-
    % I go to the next
    %   setQueen(...)
    %   placeNextQueen(...)
    %   removeQueen(...)
    % triplet of events:
    nextMethodEntryEvent(setQueen),
    nextPlaceNextQueen,
    nextMethodEntryEvent(removeQueen),
    % I count the number of time I execute the query:
    !,
    N1 is N + 1,
    query(N1, M),
    query(N, N).
```

11/18

Here is the source code counting the number of times the program being analyzed performs a sequence of method invocations.

It is declarative because the analysis is declared by a sequence of predicates.

It is meta because the predicates abstract the execution of the program being analyzed.

It is programming because we use this analysis programmatically to reason about the program execution.

LINK WITH SOUL (<http://prog.vub.ac.be/research/DMP/soul/soul2.html>)



Meta-logic programming (4/4)

■ Advantages:

- Powerful

■ Drawbacks:

- No universal quantifiers
- No temporal relationships
- Performances

Advantages:

-Powerful and natural expressions of queries.

Drawbacks:

-No universal quantifiers.

-No temporal relationships.

-Performances.



Explanation-based constraint programming (1/4)

- Abstract models of design defects
- ONE Assumption:
 - Micro-architectures similar but not identical to design pattern solutions might be:
 - Distorted because clumsily implemented
 - Distorted because not fit
 - Something else, sorry! 😊
- An explanation-based constraint solver: Ptidej Solver (PaLM)

Using PDL, we describe abstract model of design defects.

We also make one assumption: Micro-architectures that are similar but not identical to design pattern solutions are:

- Either distorted design patterns because the design patterns are clumsily implemented.
- Or distorted design patterns because the design patterns do not fit in the architecture.
- Or something else, but it is still worth to know that it looked like some design pattern!

So, we want to detect distorted micro-architectures in a given program architecture. We use explanation-based constraint programming. Explanation-based constraint programming is basically constraint programming, except that when there is no more solutions (i.e., a contradiction), the solver knows why: It knows the constraints that led to the contradiction. We can remove from the constraint systems the constraints leading to the contradiction and obtain more distorted solutions to the problem.

We develop an explanation-based constraint solver: Ptidej Solver. Ptidej Solver is based on the PaLM explanation-based constraint solver, PaLM is the reference implementation regarding explanation-based constraint programming.

Explanation-based constraint programming

(2/4)

```
[ac4ProblemForCompositePattern() : PtidejProblem ->
  let pb := makePtidejProblem("Composite Pattern Problem", length(listOfEntities), 90),
      compositeRoot := makePtidejIntVar(pb, "CompositeRoot", 1, length(listOfEntities)),
      component := makePtidejIntVar(pb, "Component", 1, length(listOfEntities)),
      composite := makePtidejIntVar(pb, "Composite", 1, length(listOfEntities), false),
      leaf := makePtidejIntVar(pb, "Leaf", 1, length(listOfEntities), false) in (

    post(pb, makeCompositionAC4Constraint("CompositeRoot <--> Component", "...", compositeRoot, component), 90),
    post(pb, makeInheritanceAC4Constraint("CompositeRoot -> Component", "...", compositeRoot, component), 60),
    post(pb, makeInheritanceAC4Constraint("Composite -> CompositeRoot", "...", composite, compositeRoot), 90),
    post(pb, makeInheritancePathAC4Constraint("Leaf -> ... -> Component", "...", leaf, component), 90),
    post(pb, makeIgnoranceAC4Constraint("Component -> Leaf", "...", component, leaf), 10),
    post(pb, makeIgnoranceAC4Constraint("Leaf -> Composite", "...", leaf, composite), 30),
    post(pb, makeNotEqualAC4Constraint("Component <> Leaf", "...", component, leaf), 100),
    post(pb, makeNotEqualAC4Constraint("Composite <> Leaf", "...", composite, leaf), 100),
    post(pb, makeNotEqualAC4Constraint("CompositeRoot <> Leaf", "...", compositeRoot, leaf), 100),

    pb
  )
)
```

14/18

Here is an example of the constraint system we use to detect complete and distorted micro-architecture identical or similar to the Composite design pattern.

This constraint system is directly obtained from the abstract model of the Composite design pattern.

First, we declare the problem.

Then, we declare the different variables.

Each variable represents an actor in the pattern.

In particular, the Composite and Leaf actor are declared as *not* enumerated, because their roles may be filled by more than one entity in the software architecture.

Finally, we post the different constraints representing the Composite design pattern.

We associate a weight with each constraint to tell the Ptidej Solver which constraints it can relax and which it can not.

It is declarative because the constraint system is declared by a sequence of constraints.

It is meta because the constraints apply on the architecture of the program.

It is programming because we use constraint systems programmatically to detect complete and distorted forms of design patterns.



Explanation-based constraint programming (4/4)

■ Advantages:

- Complete forms of patterns
- Constraint systems directly from patterns
- Explanations on relaxed constraints

■ Drawbacks:

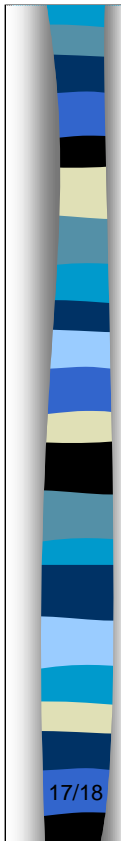
- Fine tuning (but less than fuzzy nets! 😊)
- Weights

Advantages:

- Constraint systems describe complete forms. The solver takes care of computing distorted forms.
- We obtain constraint systems directly from the pattern description.
- We have explanations on the constraints that have been relaxed.

Drawbacks:

- Fine tuning (but less than fuzzy nets! 😊).
- Weights.



Putting it all together

- With Ptidej, we can
 - Load a program (PDL)
 - Load dynamic information (Caffeine)
 - Choose a pattern to detect (PDL)
 - Load concrete patterns (Ptidej Solver)
- Demo?

Finally, we develop a tool, Ptidej (Pattern Trace Identification, Detection, and Enhancement in Java) that puts all the declarative meta-programming techniques together.

With Ptidej, we can:

- Load a program and represent its architecture using PDL.
- Load dynamic information related to the program. We obtain the dynamic information from Caffeine.
- Choose a pattern to detect. We obtain pattern descriptions from PDL (and PatternsBox).
- Detect and load the results. We obtain the results of the detection from Ptidej Solver.

MENTION TRANSFORMATIONS AND REFACTORINGS.



Discussion

- Design pattern and design defects
 - Meta-modelling
 - Meta-logic programming
 - Explanation-based constraint programming
- Comments? Ideas? Questions?

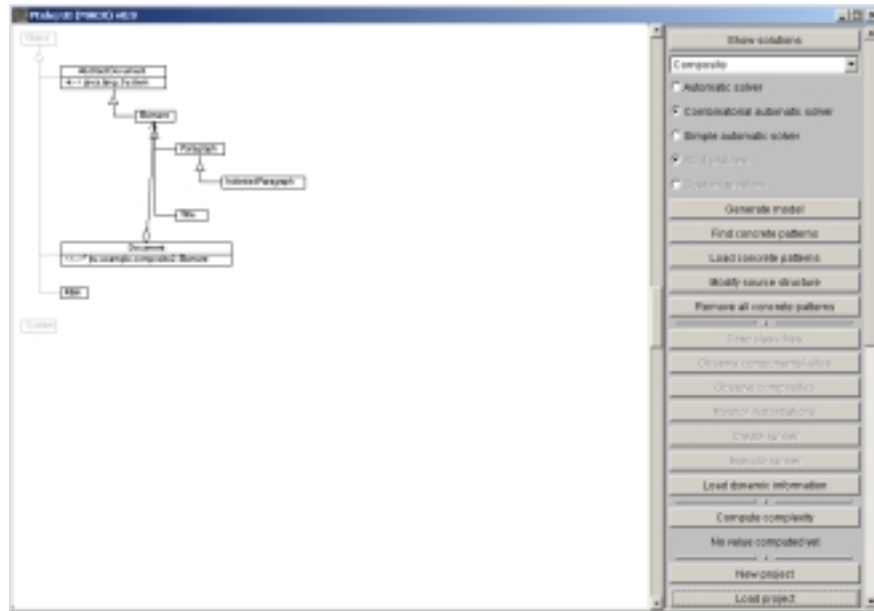
18/18

To conclude, we introduced three declarative meta-programming techniques. We use each of these meta-programming techniques in a different context. Each of these techniques has advantages and drawbacks.

Now, I would like to discuss with you these techniques, to have your opinions, to listen your comments, to answer your questions.

Paper demo: Ptidej

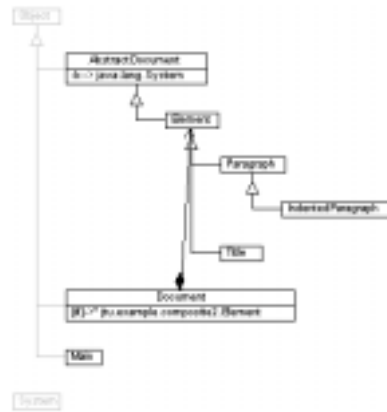
(1/4)



19/18

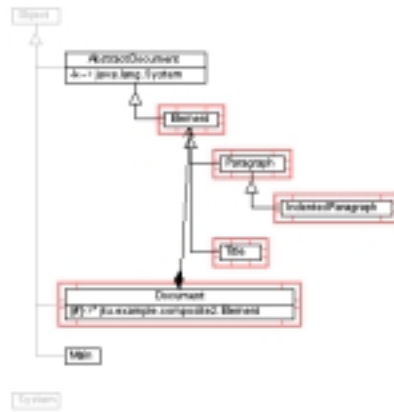
Paper demo

(2/4)



Paper demo

(3/4)



Paper demo

(4/4)

