

# Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together

Hervé Albin-Amiot ([albin@emn.fr](mailto:albin@emn.fr))

Pierre Cointe ([cointe@emn.fr](mailto:cointe@emn.fr))

Yann-Gaël Guéhéneuc ([guehene@emn.fr](mailto:guehene@emn.fr))

Narendra Jussien ([jussien@emn.fr](mailto:jussien@emn.fr))



École des Mines de Nantes, France



Object Technology  
International Inc., Canada



Soft-Maint S.A., France

Welcome

My name is...

And this is...

We are PhD students at the EMN.

Today, I come to talk about design pattern application and detection.



# Highlights

- Context
- Goal
- Solution:
  - Related works
  - Architecture
  - Experimentations
  - Limitations
- Open questions

Today's presentation follows this plan:

Context = Software engineering (obviously)

Goal = To automate design pattern application and detection, to help implementing, understanding, and re-engineering software systems.

We propose our solution.

We position ourselves relatively to the literature.

Then...

We finish with open questions, still unanswered.



# Context

- What?

- To improve OO software systems quality

- How?

- In helping OO software practitioners in:

- Implementing their design
    - Understanding their software systems
    - Re-engineering their software systems

Read the slide...



## Goal

- To provide two assistants to help in:
  - Implementing the design:
    - Applying design patterns
  - Understanding:
    - Detecting complete versions of design patterns
  - Re-engineering:
    - Detecting distorted versions of design patterns
    - Transforming source code
- Assistants must be simple and efficient

We want to automate the design patterns for the implementation phase, at the source code level.

We do not work on the requirements or design phases.

Assistants must be kept simple.

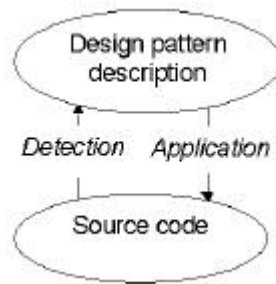
Let's take a word-processor:

Example with a wizard to make letter:

This wizard must not be bigger or more complex than the word-processor itself !!!

# Our solution

- *Round-trip* based on a common formalization of the design patterns



Concept of round-trip in OUR case = Common design pattern description/formalization:

- To apply design patterns
- To detect design patterns





# Our two assistants

## ■ PatternsBox:

- To implement the design:
  - To apply design patterns
- To understand:
  - To detect complete versions of design patterns

## ■ Ptidej:

- To understand and to re-engineer:
  - To detect distorted versions of design patterns
  - To transform source code

Our solution is made of two assistants.

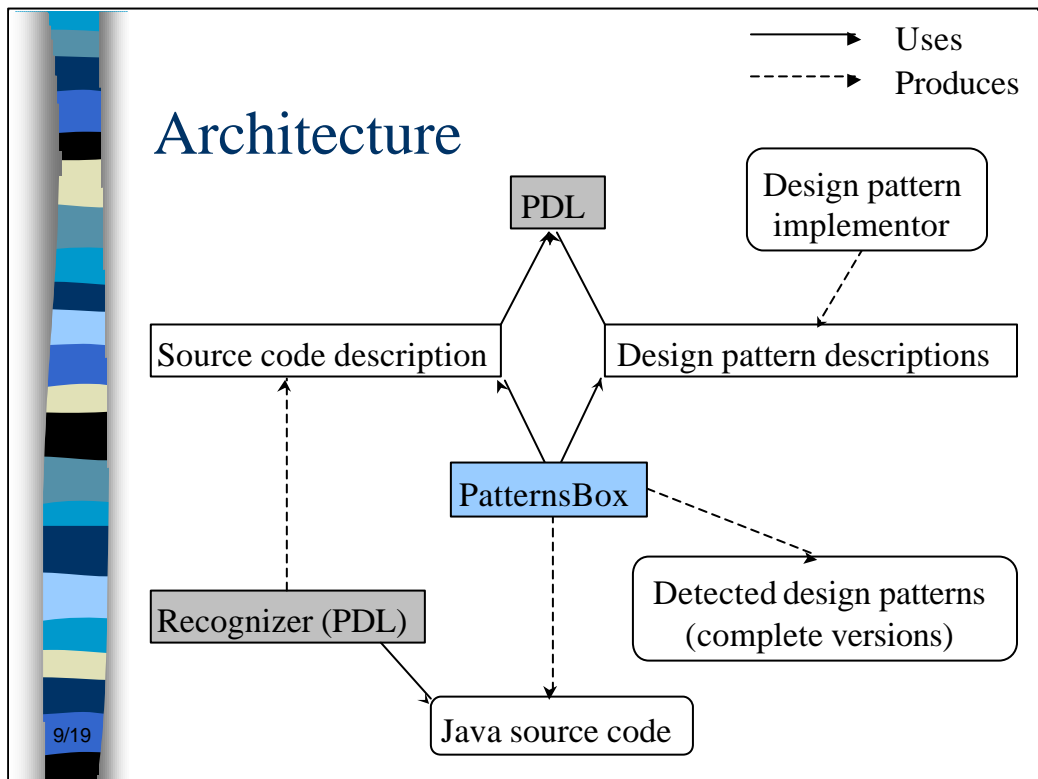


# Architecture

- **PDL** (Pattern Description Language):
  - A meta-model for design pattern formalization
- **JavaXL** (Java eXtended Language):
  - A Java source transformation engine
- **PaLM** (Propagation and Learning with Move):
  - An explanation-based constraint solver

Those two assistants are based on three components...

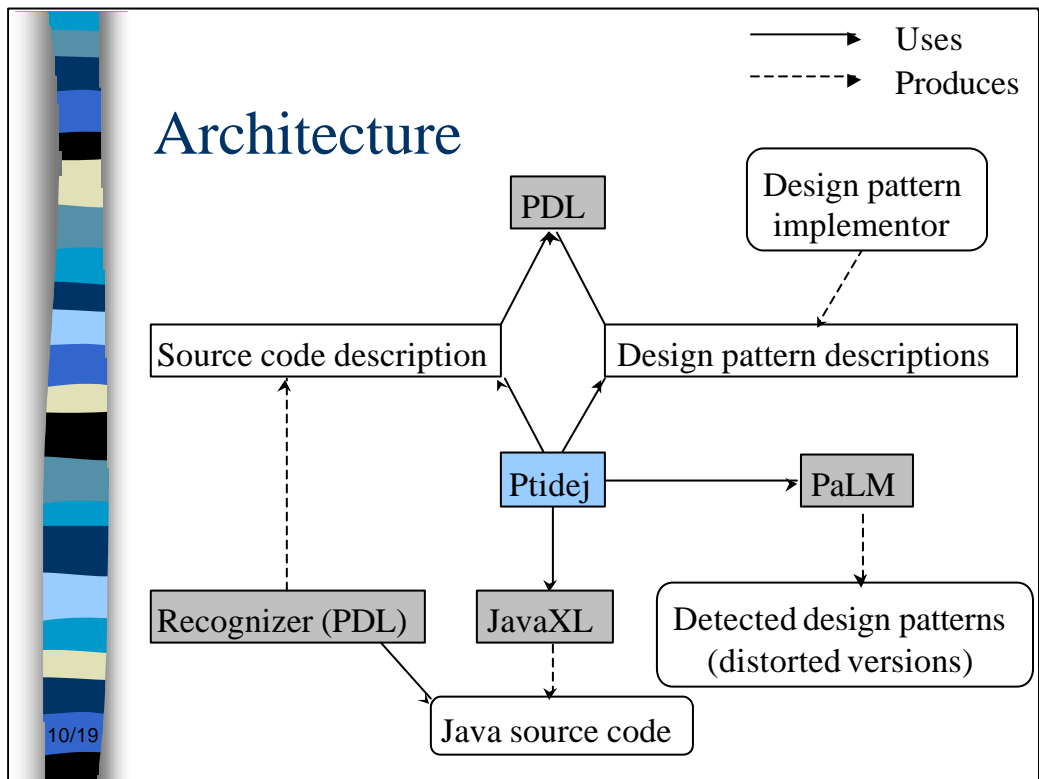
PDL = Design pattern descriptions.



On one hand...

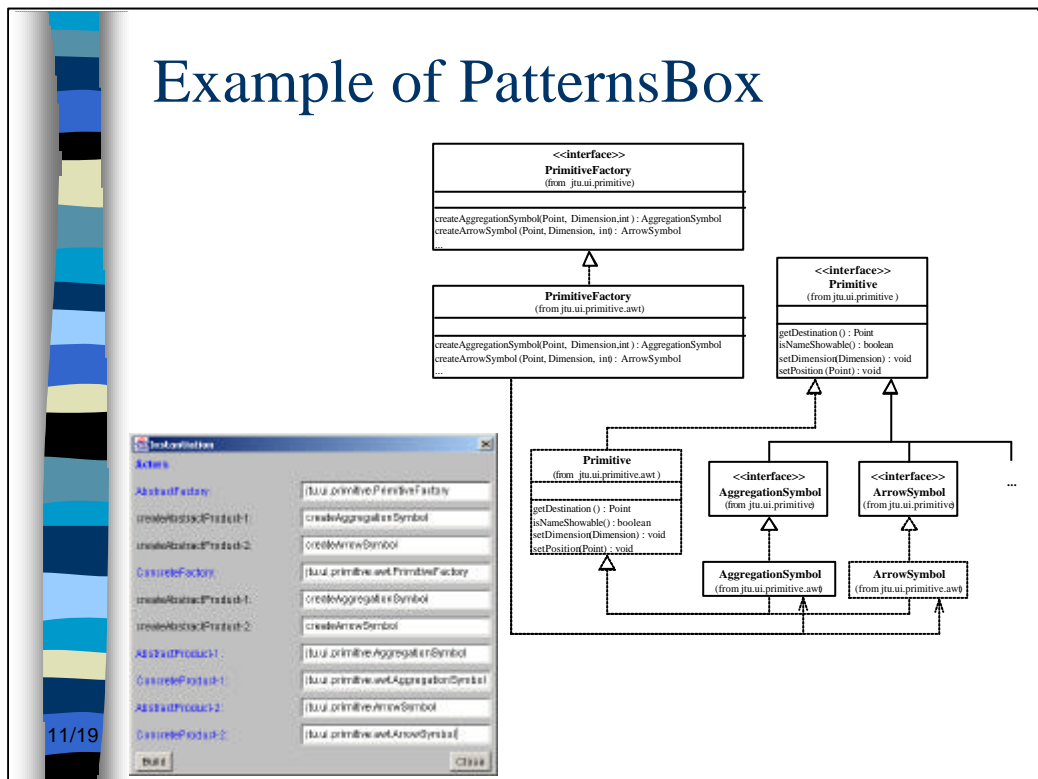
PatternsBox detects complete versions of design patterns using arc-consistency or specific and language-dependent detection algorithms.

PatternsBox is open enough to accept any customized detection algorithm to improve design pattern detection.



On the other hand...

# Example of PatternsBox



We presented our two assistants and their architecture.

We present now some experimentations.

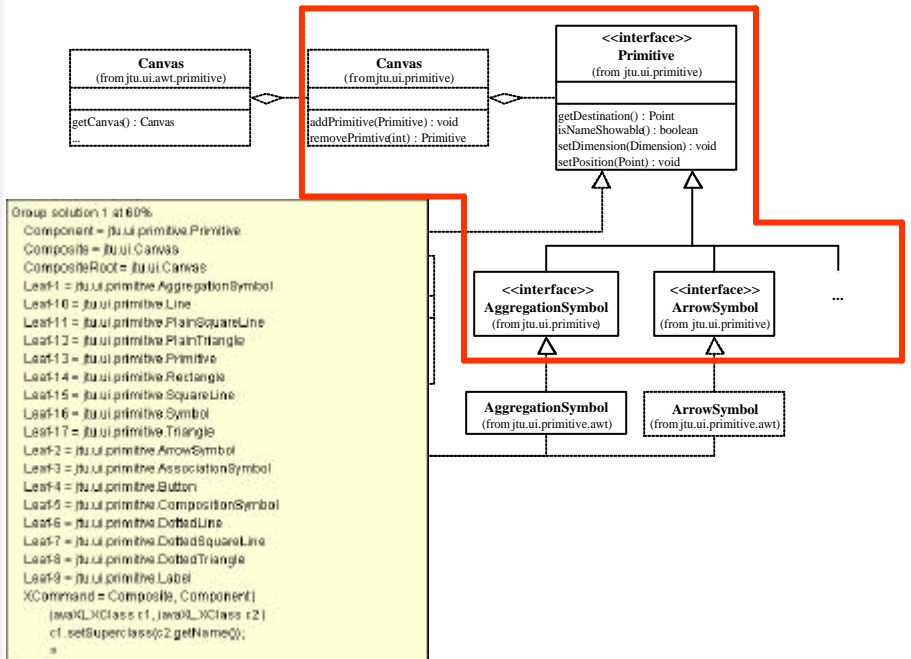
However, it is difficult to show usage experimentations, because it is hard on slides to show you how tools work.

Here is an example of how we applied the design pattern Abstract Factory.

This is a real case used in Ptidej...

Here, we present the instantiation window if the Abstract Factory design pattern...

# Example of Ptidej



Here, you have another related part of the Ptidej architecture.

Briefly, the Abstract Factory creates widgets that are aggregated into a Canvas, and Ptidej...

We used Ptidej to re-engineer its own architecture...

Now, we are going to show you some results on the understanding and re-engineering parts of our assistants.

# Results

Design patterns	Frameworks	NOC	Existing	Understanding (PatternsBox)				Reengineering (Ptidej)				
				Hits	Missed	False hits	Time (sec.)	Complete			Distorted	Time (sec.)
								Hits	Missed	False hits		
Composite	java.awt.*	121	1	1			1	1			7	82,6
	JHotDraw	155	1	1			0,3	1			3	65,5
	JUnit	34	1		1		0,4	1			7	22,9
	JEdit	248					1,4					29,5
	PatternsBox	52					0,3				3	40,3
Decorator	java.awt.*	121					0,2			1	7	82,6
	JHotDraw	155	1	3		2	0,4	1			3	65,5
	JUnit	34	1	1			0,2	1			7	22,9
	JEdit	248					0,4					29,5
	PatternsBox	52					0,3				3	40,3
Factory Method	java.awt.*	121	3				1,1	3		8	1	7
	JHotDraw	155	2	2	1	1	0,5	2		3	1	103,7
	JUnit	34								1	1	23
	JEdit	248					0,1				6	29,7
	PatternsBox	52					0,2			7	1	13,5
Iterator	java.awt.*	121									12	75,6
	JHotDraw	155	3	3			0,1		3	100		231,1
	JUnit	34									8	22,7
	JEdit	248	1	1			0,1				1	28,5
	PatternsBox	52								7		36,5
Observer	java.awt.*	121					3,4	4		4		73,5
	JHotDraw	155	2	2			3,2	2	2	2		61,4
	JUnit	34	4	4			2,5					19,9
	JEdit	248	3	3			13		3			27,8
	PatternsBox	52	1	1			2,8	2	1	2		31,5
Singleton	java.awt.*	121	3	3			0,7					
	JHotDraw	155	2	2			0,5					
	JUnit	34					0,4					
	JEdit	248					1					
	PatternsBox	52	1	1			0,5					
		610	29	28	2	3	1,3	18	9	262	50	50,7

In the following table, we apply our detection mechanism to different frameworks...

Namely...

And the design patterns... WHICH ARE COMMON.

Our understanding tool is fast and accurate, thanks to its specific and language-dependent algorithms.

Our re-engineering tool is slower and less accurate but provide useful information on distorted design patterns, and helps efficiently in re-engineering tools, like we showed in the previous experimentations...

Very useful...

Requires design knowledge...

The summary is that among the 5 frameworks, 610 classes, but only 29 instances of those 6 very common design patterns.

Drawbacks = 610 classes, 12 design patterns in JHotDraw...



## Current limitations

- PatternsBox does not transform source code, it only generates skeletons
- Ptidej is slow and not accurate enough
- Only for Java

We presented our two assistants and real-life experimentations but the tools have...



# Open questions

## ■ Formalization:

- Is a unique meta-model sufficient to formalize design patterns?
- How to represent trade-offs?

## ■ Application:

- How to represent usefully source code?
- Is source transformation possible and interesting for any design pattern?



# Open questions

## ■ Understanding:

- Is it possible to detect complete versions of any design pattern?
- Is it better:
  - To use a common formalization to detect complete versions of design patterns?
  - To use specific algorithms?
  - To mix both approaches?



# Open questions

- Re-engineering:
  - How to distinguish a distorted design pattern from a non-design pattern micro-architecture?
  - How to present the interesting pieces of information on distorted versions of design patterns?



## Future

- Complete catalogue of design patterns:
  - More frameworks
  - Detection of complete versions
  - Detection of distorted versions
- More on design pattern application:
  - Source transformation

We propose a set of tools to:

- Apply design patterns
- Detect complete and distorted design patterns micro-architectures
- Correct design defect

These tools work, are useful but have some limitations...

Thank you for your attention

Questions?

Comments?



 École des Mines de Nantes  
 Object Technology International Inc.  
 Soft-Maint S.A.

Hervé Albin-Amiot ([albin@emn.fr](mailto:albin@emn.fr))  
Pierre Cointe ([cointe@emn.fr](mailto:cointe@emn.fr))  
Yann-Gaël Guéhéneuc ([guehene@emn.fr](mailto:guehene@emn.fr))  
Narendra Jussien ([jussien@emn.fr](mailto:jussien@emn.fr))