Micropatterns in Grammars

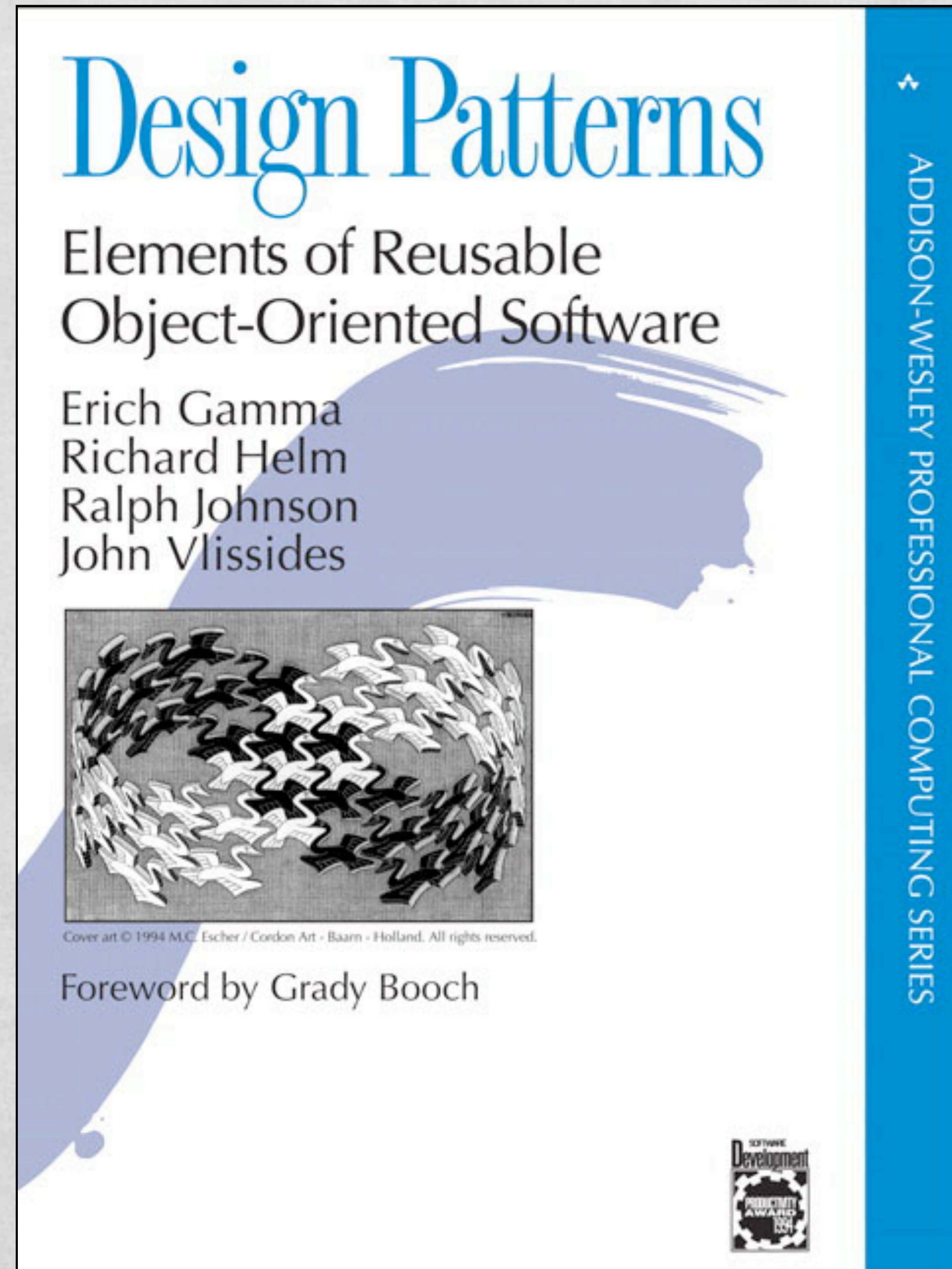
Dr. Vadim Zaytsev, SWAT, CWI

grammarware @ SLE 2013 @ SPLASH

Related work

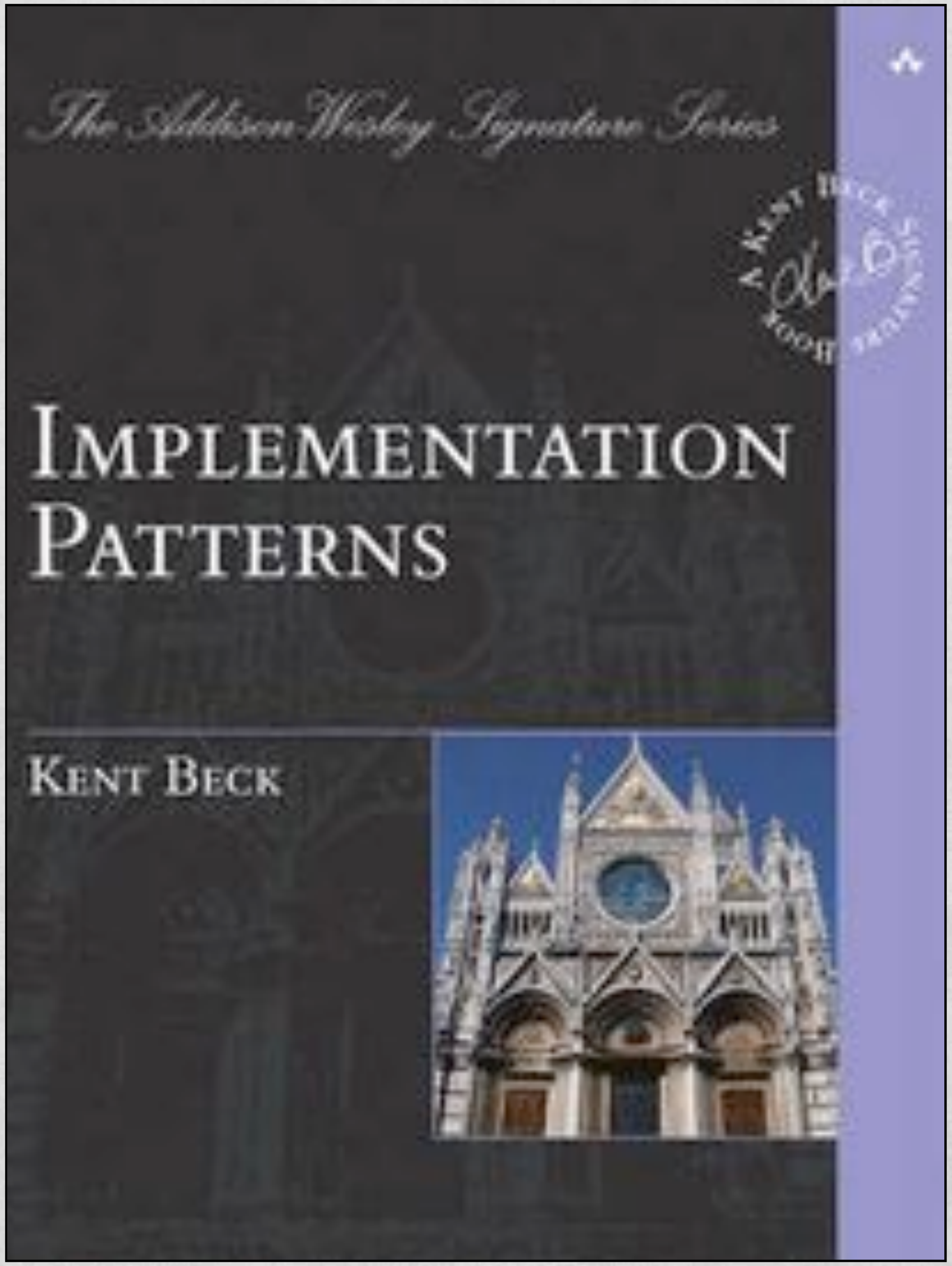
… Patterns

- ~~Pattern matching~~
- Design Patterns



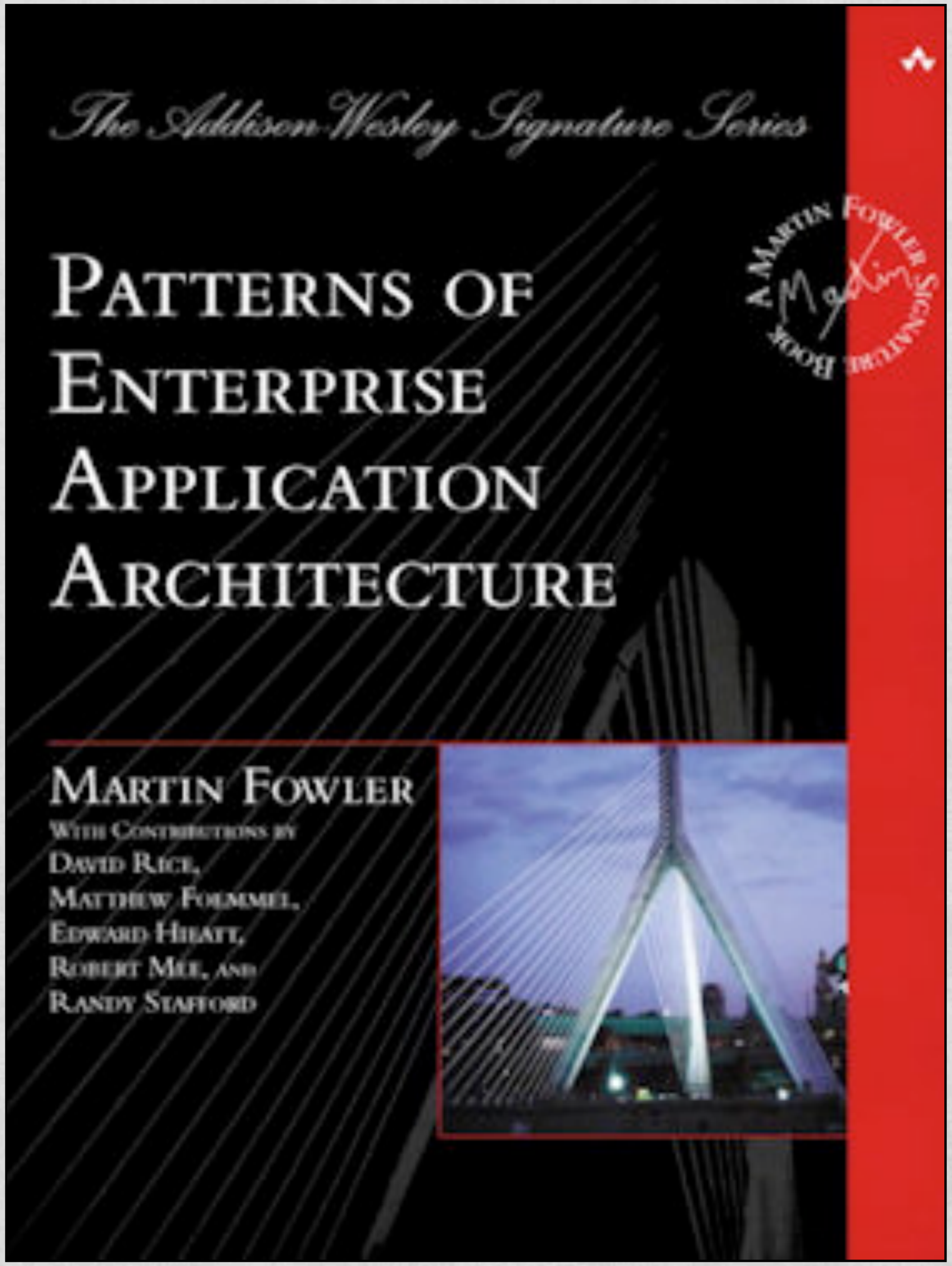
... Patterns

- ~~Pattern matching~~
- Design Patterns
- Implementation Patterns



… Patterns

- ~~Pattern matching~~
- Design Patterns
- Implementation Patterns
- Architectural Patterns



… Patterns

- ~~Pattern matching~~
- Design Patterns
- Implementation Patterns
- Architectural Patterns
- Micro Patterns

Micro Patterns in Java Code*

Joseph (Yossi) Gil† Itay Maman
Department of Computer Science
Technion—Israel Institute of Technology
{ yogi | imaman } @ cs.technion.ac.il

Abstract

Micro patterns are similar to design patterns*, except that micro patterns stand at a lower, closer to the implementation, level of abstraction. Micro patterns are also unique in that they are mechanically recognizable, since each such pattern can be expressed as a formal condition on the structure of a class.*

This paper presents a catalog of 27 micro-patterns defined on JAVA *classes and interfaces. The catalog captures a wide spectrum of common programming practices, including a particular and (intentionally restricted) use of inheritance, immutability, data management and wrapping, restricted creation, and emulation of procedural-, modular-, and even functional- programming paradigms with object oriented constructs. Together, the patterns present a set of prototypes after which a large portion of all* JAVA *classes and interfaces are modeled. We provide empirical indication that this portion is as high as 75% .*

A statistical analysis of occurrences of micro patterns in a large software corpus, spanning some 70,000 JAVA *classes drawn from a rich set of application domains, shows, with high confidence level that the use of these patterns is not random. These results indicate consciousness and discernible design decisions, which are sustained in the software evolution. With high confidence level, we can also show that the use of these patterns is tied to the specification, or the purpose, that the software realizes.*

The traceability, abundance *and the* statistical significance *of micro pattern occurrence raise the hope of using the classification of software into these patterns for a more founded appreciation of its design and code quality.*

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages

General Terms

Design, Object-Oriented Programming

Keywords

Program Analysis, Design Patterns, Implementation Patterns

1. Introduction

We all know what makes one algorithm better than another: time, space, random-bits, disk access, etc. are established, *objective* and well defined metrics [14] to be employed in making such a judgement. In contrast, the assessment of quality of software design is an illusive prospect. Despite the array of books and research articles on the topic (see e.g., [11, 12, 28, 30]), a question such as "*Is Design A better than Design B?*" can, still, only be decided by force of the argumentation, and ultimately, by the personal and *subjective* perspective of the judge.

The research described in this paper is concerned with the important, yet so recalcitrant, problem of finding sound objective methods of assessment of design. Medical experiments can prove that a certain medication is better than another in treating a specific ailment. We all want to carry similar controlled experiments to prove that certain design methods are more likely to produce better software than others. However, in contrast with many other natural sciences, experiments on large scale software development are so prohibitively costly that much of the research on the topic abandoned this hope.

Our attack on this multiple Gordian knot is by taking a different angle at it: Rather than subjecting the development process to experimentation, we apply statistical tools to *existing* artifacts of the development. Instead of dealing with "is *A better than B?*" sort of questions, our research should help in rigorously determining *"how is A different than B?*'. We can also show that certain design techniques are more common than others. The judgement of the quality of design can perhaps then be reduced to the judgement of the abundance of the design, and the quality of the software that uses it.

This angle is made possible by the bountiful class structure of JAVA [3], together with the colossal, publicly available, base of software in the language, which opens the road for sound claims and understanding of the way people write software (more precisely, on the software written by people). We argue that this class structure makes it possible to find traces of design, specifically of what we shall call *micro patterns*.

1.1 Traceability of Design

Can design be traced and identified in software? The prime candidates of units of design to look for in the software are obviously *design patterns* [22]. However, despite the dozen years that passed since the original publication [21], and the voluminous research ensuing it, attempts to automate and formalize design patterns are scarce. Systems like DisCo [31], LePUS [16,17], SPINE and HEDGEHOG [6], constraint diagrams [27], Elemental Design Patterns [39],

*Research supported in part by Israel Science Foundation (ISF) grant no. 2004460.
†Work is supported in part by the IBM faculty award.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

	Main Category	Pattern	Short description	Additional Category
Degenerate Classes	Degenerate State and Behavior	Designator	An interface with absolutely no members.	
		Taxonomy	An empty interface extending another interface.	
		Joiner	An empty interface joining two or more superinterfaces.	
		Pool	A class which declares only static final fields, but no methods.	
	Degenerate Behavior	Function Pointer	A class with a single public instance method, but with no fields.	
		Function Object	A class with a single public instance method, and at least one instance field.	
		Cobol Like	A class with a single static method, but no instance members	
	Degenerate State	Stateless	A class with no fields, other than static final ones.	
		Common State	A class in which all fields are static.	
		Immutable	A class with several instance fields, which are assigned exactly once, during instance construction.	
	Controlled Creation	Restricted Creation	A class with no public constructors, and at least one static field of the same type as the class	
		Sampler	A class with one or more public constructors, and at least one static field of the same type as the class	
Containment	Wrappers	Box	A class which has exactly one, mutable, instance field.	
		Compound Box	A class with exactly one non primitive instance field.	
		Canopy	A class with exactly one instance field that it assigned exactly once, during instance construction.	Degenerate State
	Data Managers	Record	A class in which all fields are public, no declared methods.	Degenerate Behavior
		Data Manager	A class where all methods are either setters or getters.	
		Sink	A class whose methods do not propagate calls to any other class.	
Inheritance	Base Classes	Outline	A class where at least two methods invoke an abstract method on "this"	Degenerate State
		Trait	An abstract class which has no state.	
		State Machine	An interface whose methods accept no parameters.	Degenerate State and Behavior
		Pure Type	A class with only abstract methods, and no static members, and no fields	
		Augmented Type	Only abstract methods and three or more static final fields of the same type	
		Pseudo Class	A class which can be rewritten as an interface: no concrete methods, only static fields	
	Inheritors	Implementor	A concrete class, where all the methods override inherited abstract methods.	
		Overrider	A class in which all methods override inherited, non-abstract methods.	
		Extender	A class which extends the inherited protocol, without overriding any methods.	

Table 1: Micro patterns in the catalog

J. Gil and I. Maman. Micro Patterns in Java Code. OOPSLA'05, pages 97–116. ACM, 2005.

… Patterns

- ~~Pattern matching~~
- Design Patterns
- Implementation Patterns
- Architectural Patterns
- Micro Patterns
- Nano Patterns
- Milli Patterns

Micro Patterns in Java Code*

Joseph (Yossi) Gil† Itay Maman
Department of Computer Science
Technion—Israel Institute of Technology
{ yogi | imaman } @ cs.technion.ac.il

Abstract

Micro patterns are similar to design patterns*, except that micro patterns stand at a lower, closer to the implementation, level of abstraction. Micro patterns are also unique in that they are mechanically recognizable, since each such pattern can be expressed as a formal condition on the structure of a class.*

This paper presents a catalog of 27 micro-patterns defined on JAVA *classes and interfaces. The catalog captures a wide spectrum of common programming practices, including a particular and (intentionally restricted) use of inheritance, immutability, data management and wrapping, restricted creation, and emulation of procedural-, modular-, and even functional- programming paradigms with object oriented constructs. Together, the patterns present a set of prototypes after which a large portion of all* JAVA *classes and interfaces are modeled. We provide empirical indication that this portion is as high as 75% .*

A statistical analysis of occurrences of micro patterns in a large software corpus, spanning some 70,000 JAVA *classes drawn from a rich set of application domains, shows, with high confidence level that the use of these patterns is not random. These results indicate consciousness and discernible design decisions, which are sustained in the software evolution. With high confidence level, we can also show that the use of these patterns is tied to the specification, or the purpose, that the software realizes.*

The traceability, abundance *and the* statistical significance *of micro pattern occurrence raise the hope of using the classification of software into these patterns for a more founded appreciation of its design and code quality.*

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages

General Terms

Design, Object-Oriented Programming

*Research supported in part by Israel Science Foundation (ISF) grant no. 2004460.
†Work is supported in part by the IBM faculty award.

Keywords

Program Analysis, Design Patterns, Implementation Patterns

1. Introduction

We all know what makes one algorithm better than another: time, space, random-bits, disk access, etc. are established, *objective* and well defined metrics [14] to be employed in making such a judgement. In contrast, the assessment of quality of software design is an illusive prospect. Despite the array of books and research articles on the topic (see e.g., [11,12,28,30]), a question such as "*Is Design A better than Design B?*" can, still, only be decided by force of the argumentation, and ultimately, by the personal and *subjective* perspective of the judge.

The research described in this paper is concerned with the important, yet so recalcitrant, problem of finding sound objective methods of assessment of design. Medical experiments can prove that a certain medication is better than another in treating a specific ailment. We all want to carry similar controlled experiments to prove that certain design methods are more likely to produce better software than others. However, in contrast with many other natural sciences, experiments on large scale software development are so prohibitively costly that much of the research on the topic abandoned this hope.

Our attack on this multiple Gordian knot is by taking a different angle at it: Rather than subjecting the development process to experimentation, we apply statistical tools to *existing* artifacts of the development. Instead of dealing with "is *A better than B?*" sort of questions, our research should help in rigorously determining *"how is A different than B?*'. We can also show that certain design techniques are more common than others. The judgement of the quality of design can perhaps then be reduced to the judgement of the abundance of the design, and the quality of the software that uses it.

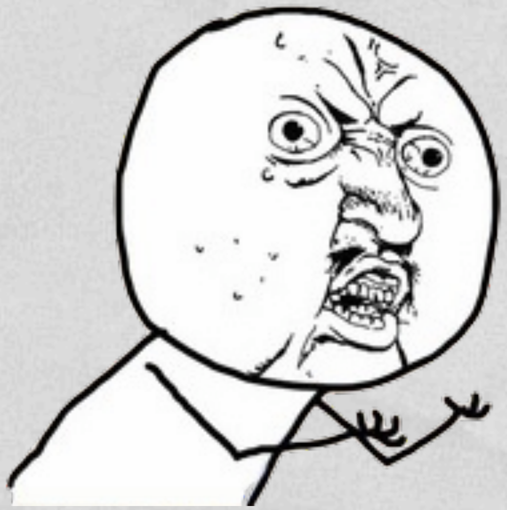
This angle is made possible by the bountiful class structure of JAVA [3], together with the colossal, publicly available, base of software in the language, which opens the road for sound claims and understanding of the way people write software (more precisely, on the software written by people). We argue that this class structure makes it possible to find traces of design, specifically of what we shall call *micro patterns*.

1.1 Traceability of Design

Can design be traced and identified in software? The prime candidates of units of design to look for in the software are obviously *design patterns* [22]. However, despite the dozen years that passed since the original publication [21], and the voluminous research ensuing it, attempts to automate and formalize design patterns are scarce. Systems like DisCo [31], LePUS [16,17], SPINE and HEDGEHOG [6], constraint diagrams [27], Elemental Design Patterns [39],

Why micropatterns?

- Is design A better than design B?
- How design A is better than design B?
- How to design C?

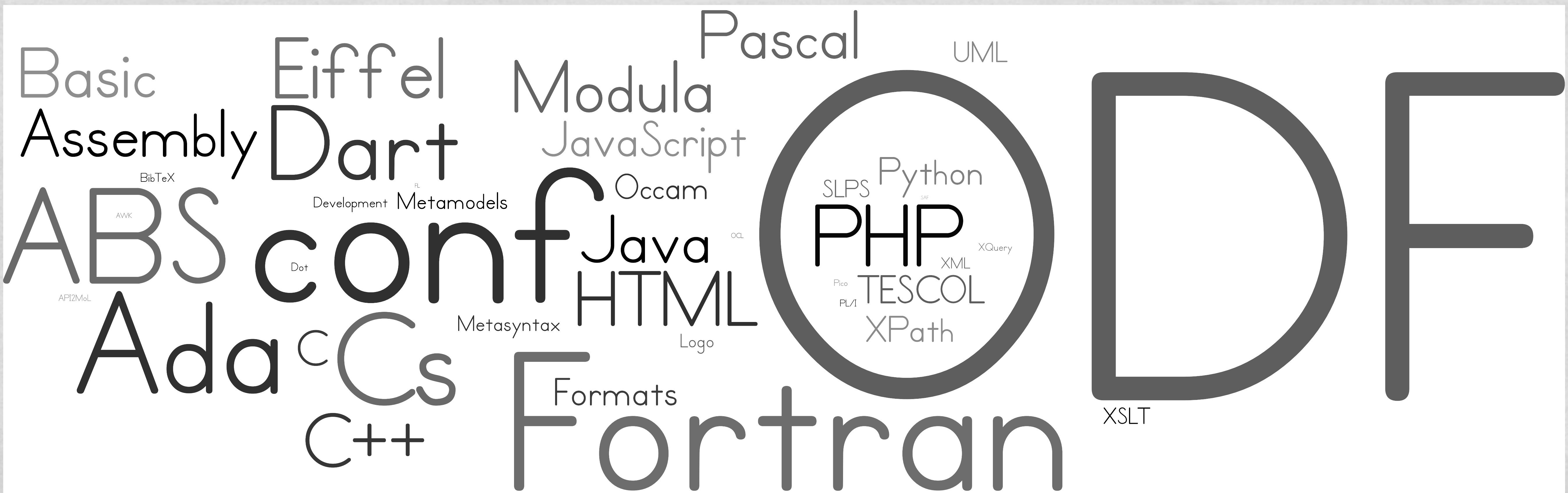


SLE design BoK

- Syntactic Structures (Ch, 1957)
- Orthogonal Design and Description of a FL (AvW, 1965)
- Go To Statement Considered Harmful (D, 1968)
- Minority Report (D, 1968)
- Hints on PL Design (H, 1973)
- On the Design of PLs (W, 1974)
- On the Design of PLs Including MINI ALGOL 68 (Ammeraal, 1975)
- Designing a Beginners' PL (Geurts, Meertens, 1976, 1980)
- The Design of Elegant Languages (Meertens, 1993)
- When and How to Develop DSLs (Mernik, Heering, Sloane, 2005)
- Evolving a DSL Implementation (Tratt, 2007)
- SLE: Creating DSLs Using Metamodels (Kleppe, 2008)
- DSLs (Fowler, 2010)
- Language Implementation Patterns (Parr, 2010)
- Semantics First! (Erwig, Walkingshaw, 2012)
- DSL Engineering: Designing, Implementing and Using DSLs (Völter et al, 2013)

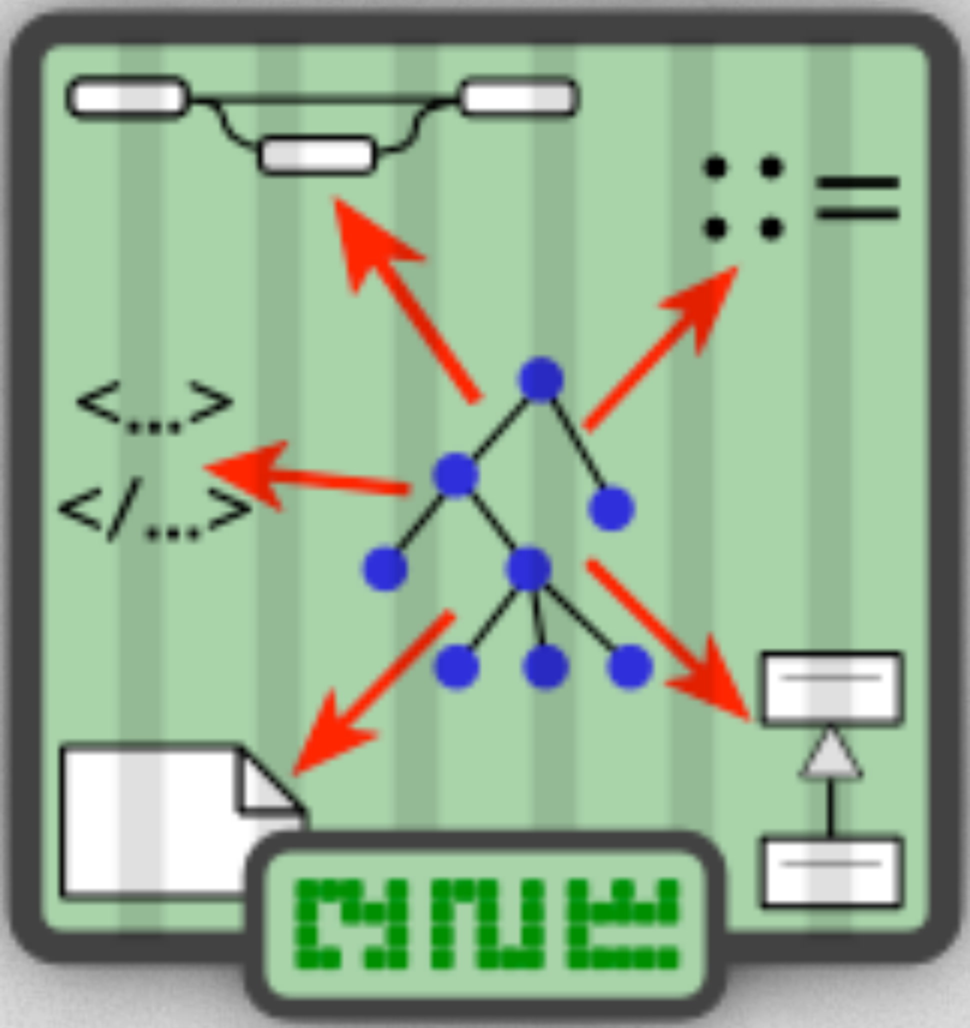
Grammar Zoo

http://slps.github.io/zoo



Grammar Zoo

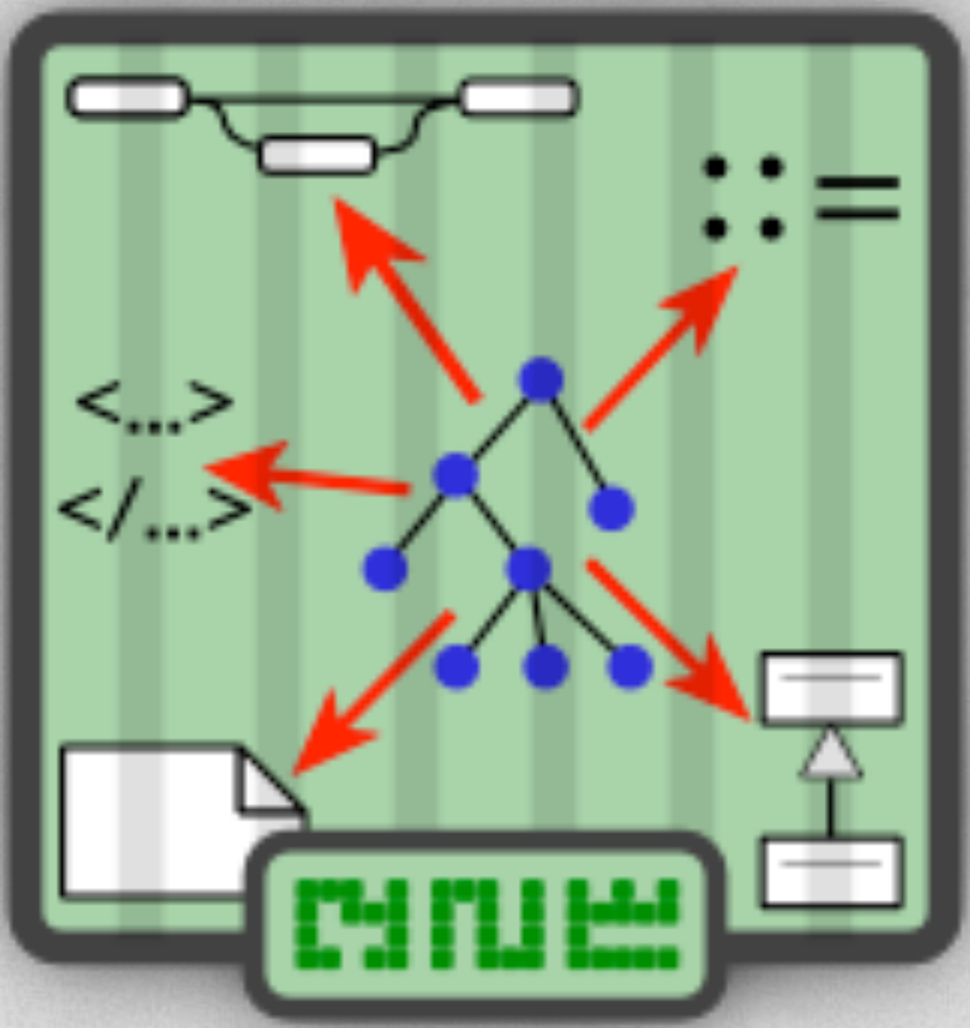
http://slps.github.io/zoo



- Language documentation
  - ISO, ECMA, W3C, OMG
- Document schemata
  - XML Schema, RELAX NG, Ecore
- Concrete syntax specs
  - Rascal library
  - SDF library
  - TXL library
  - ANTLR library

Grammar Zoo

http://slps.github.io/zoo



- Coursework
  - TESCOL, FL
- Versioning system
  - BGF, XBGF, EDD, LCF, LDF, XLDF
- Metamodels
  - the entire AtlantEcore Zoo
- Other collections
  - books; test suites
  - mining/hunting/crawling

Back to the grammars

Grammatical micropatterns

- Recognisable
- Purposeful
- Prevalent
- Simple
- Local
- Evidently used

Grammatical micropatterns

- Is an isolated micropattern useful?
- How high is the coverage?
- Mining what?

Global & structure

Category	Pattern	Matches	Prevalence
Global	Root	563	1.37%
	Leaf	9,467	23.07%
	Top	3,245	7.91%
	MultiRoot	1	0.002%
	Bottom	1,311	3.19%
	Total coverage	**12,459**	**30.36%**
Structure	Disallowed	69	0.17%
	Singleton	29,134	70.99%
	Vertical	3,697	9.01%
	Horizontal	6,043	14.73%
	ZigZag	784	1.91%
	Total coverage	**39,727**	**96.81%**

Global & structure

```
Expression2:
        Expression3 Expression2Rest?
Expression2Rest:
        (Infixop Expression3)*
Expression2Rest:
        Expression3 "instanceof" Type
```

}vertical

horizontal:

```
Modifier:
      "public" | "protected" | "private" | "static" | "abstract"
            | "final" | "native" | "synchronized" | "transient"
            | "volatile" | "strictfp"
```

Metasyntax micropatterns

Category	Pattern	Matches	Prevalence
Metasyntax	ContainsEpsilon	4,185	10.20%
	ContainsFailure	69	0.17%
	ContainsUniversal	825	2.01%
	ContainsString	1,889	4.60%
	ContainsInteger	343	0.84%
	ContainsOptional	6,554	15.97%
	ContainsPlus	4,586	11.18%
	ContainsStar	3,080	7.51%
	ContainsSepListPlus	55	0.13%
	ContainsSepListStar	142	0.35%
	ContainsDisjunction	2,804	6.83%
	ContainsSelectors	17,328	42.22%
	ContainsLabels	132	0.32%
	ContainsSequence	19,447	47.39%
	AbstractSyntax	29,299	71.39%
	Total coverage	**36,522**	**89.00%**

Sugary micropatterns

Category	Pattern	Matches	Prevalence	Frequency
Sugar	FakeOptional	134	0.33%	10.89%
	FakeSepList	624	1.52%	50.69%
	ExprMidLayer	349	0.85%	28.35%
	ExprLowLayer	39	0.10%	3.17%
	YaccifiedPlusLeft	354	0.86%	28.76%
	YaccifiedPlusRight	6	0.01%	0.49%
	YaccifiedStarLeft	0	0.00%	0.00%
	YaccifiedStarRight	0	0.00%	0.00%
	Total coverage	**1,231**	**3.00%**	

ExprMidLayer

ExprMidLayer

```
logical-or-expression ::= logical-and-expression
 | logical-or-expression "||" logical-and-expression ;
logical-and-expression ::= inclusive-or-expression
 | logical-and-expression "&&" inclusive-or-expression ;
```

... (12 layers skipped) ...

```
primary-expression ::= literal | "this"
 | "(" expression ")" | id-expression ;
```

(ISO/IEC 14882:1998(E) C++)

ExprLowLayer

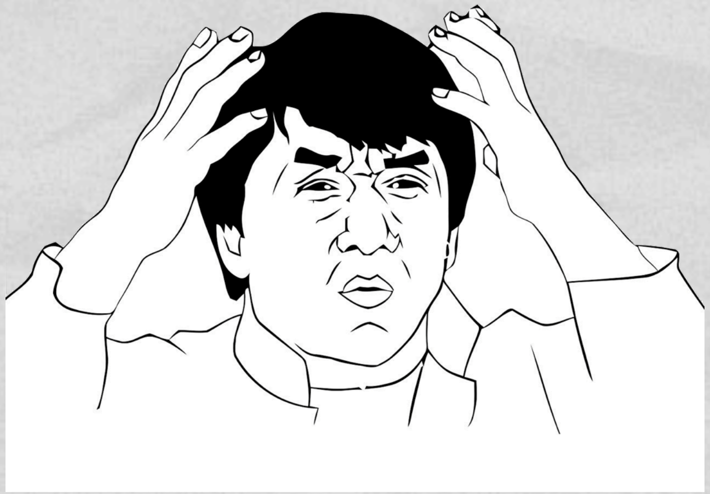
Naming micropatterns

Category	Pattern	Matches	Prevalence
Naming	CamelCase	16704	40.70%
	LowerCase	3323	8.10%
	MixedCase	1706	4.16%
	MultiWord	31816	77.53%
	UpperCase	2073	5.05%
	Total coverage	**40,562**	**98.84%**
Naming, lax	CamelCaseLax	18332	44.67%
	LowerCaseLax	17840	43.47%
	MixedCaseLax	1969	4.80%
	MultiWordLax	32290	78.68%
	UpperCaseLax	2412	5.88%
	Total coverage	**41,038**	**100.00%**

All clasified!

All clasified?

Express_metamodel::Core::GeneralARRAYType



Concrete syntax

Category	Pattern	Matches	Prevalence	Frequency
Concrete	Preterminal	3249	7.92%	100.00%
	Keyword	906	2.21%	27.89%
	Keywords	1774	4.32%	54.60%
	Operator	1001	2.44%	30.81%
	Operators	1190	2.90%	36.63%
	OperatorsMixed	110	0.27%	3.39%
	Words	40	0.10%	1.23%
	Tokens	34	0.08%	1.05%
	Modifiers	19	0.05%	0.58%
	Range	730	1.78%	22.47%
	NumericLiteral	51	0.12%	1.57%
	LiteralSimple	15	0.04%	0.46%
	LiteralFirstRest	62	0.15%	1.91%
	EmptyStatement	30	0.07%	0.92%
	Total coverage	**3,249**	**7.92%**	

```
exit_qualifier ::= ("__exit" | "exit__" | "exit" | "__exit__") ;
```

Keywords

(TXL C Basis Grammar 5.2)

```
typeModifier ::= ("opt" | "repeat" | "list" | "attr" | "see" | "not"
 | "push" | "pop" | ":" | "~" | ">" | "<") ;
```

OperatorsMixed

(TXL Basis Grammar for TXL 10.5)

```
op ::= (">" | "<" | "<=" | ">=" | "<>" | "=" | "in" | "is" | "+" | "-"
 | "or" | "xor" | "*" | "/" | "div" | "mod" | "and" | "shl" | "shr"
 | "DIV" | "AND") ;
```

(TXL Basis Grammar for Borland Delphi Object Pascal 1.1)

```
simpleDerivationSet ::= "#all" | ("list" | "union" | "restriction")*
```

Words

(RELAX NG schema for XML Schema)

```
mml.lines.datatype ::= ("none" | "solid" | "dashed")+
```

(TESCOL 10001)

LiteralFirst Rest

```
IDENT ::= ("a" | ... | "z" | "A" | ... | "Z" | "_" | "$")
 ("a" | ... | "z" | "A" | ... | "Z" | "_" | "0" | ... | "9" | "$")* ;
```

(Michael Studman Java 5)

```
VARID ::= ("A" | ... | "Z" | "a" | ... | "z")
 ("A" | ... | "Z" | "a" | ... | "z" | "0" | ... | "9" | "_")* ;
```

(TESCOL 10110)

Normal forms

Category	Pattern	Matches	Prevalence
Normal	CNF	5,365	13.07%
	GNF	3,074	7.49%
	ANF	26,269	64.01%
	Total coverage	**28,168**	**68.64%**

Folding/unfolding

Category	Pattern	Matches	Prevalence	Frequency
Folding	Empty	3,028	7.38%	32.56%
	Failure	69	0.17%	0.74%
	JustOptional	48	0.12%	0.52%
	JustPlus	199	0.48%	2.14%
	JustStar	130	0.32%	1.40%
	JustSepListPlus	28	0.07%	0.30%
	JustSepListStar	32	0.08%	0.34%
	JustChains	1,045	2.55%	11.24%
	JustOneChain	2,065	5.03%	22.20%
	ReflexiveChain	0	0.00%	0.00%
	ChainOrTerminal	145	0.35%	1.56%
	ChainsAndTerminals	290	0.71%	3.12%
	Total coverage	**9,300**	**22.66%**	

Category	Pattern	Matches	Prevalence	Frequency
Template	Constructor	657	1.60%	13.56%
	Bracket	132	0.32%	2.73%
	BracketedFakeSepList	56	0.14%	1.16%
	BracketedFakeSLStar	10	0.02%	0.21%
	BracketedOptional	117	0.29%	2.42%
	BracketedPlus	6	0.01%	0.12%
	BracketedSepListPlus	8	0.02%	0.17%
	BracketedSepListStar	24	0.06%	0.50%
	BracketedStar	15	0.04%	0.31%
	Delimited	81	0.20%	1.67%
	ElementAccess	25	0.06%	0.52%
	PureSequence	2,999	7.31%	61.91%
	DistinguishByTerm	933	2.27%	19.26%
	Total coverage	**4,844**	**11.80%**	

Bracket

```
Explicit_creation_type ::= "{" Type "}" ;
Actual_generics ::= "[" Type_list "]" ;
Parenthesized ::= "(" Expression ")" ;
External_system_file ::= "<" Simple_string ">" ;
```

(ISO/IEC 25436:2006(E) Eiffel)

BracketedFakeSL

```
typeParameters ::= "<" typeParameter ("," typeParameter)* ">" ;
namedFormalParameters ::= "[" defaultFormalParameter
    ("," defaultFormalParameter)* "]" ;
```

(ANTLR Google Dart)

```
template ::= "{{" title ("|" part)* "}}" ;
tplarg ::= "{{{" title ("|" part)* "}}}" ;
```

(EBNF MediaWiki)

Delimited

```
RecordType ::= "RECORD" Fields "END" ;
LoopStmt ::= "LOOP" Stmts "END" ;
```

(SDF Modula 3)

ElementAccess

```
slice ::= prefix "(" discrete_range ")" ;
```

(LNCS 4348, Ada 2005)

```
libraryDefinition ::= LIBRARY "{" libraryBody "}" ;
```

(ANTLR Google Dart)

```
ArrayDeclarator ::= VariableName "(" ArraySpec ")" ;
StructureConstructor ::= TypeName "(" ExprList ")" ;
```

(TXL Fortran 77/90)

Conclusion

- Experiment is successful
- Concise in Rascal

```
set[str] check4mp(bracketSLPlus(), GGrammar g) = {n | str n <- g.nts,
  [production(n,sequence([
     terminal(str x),
     seplistplus(nonterminal(_),terminal(_)),
     terminal(str y)]))
  ] := normanon(g.prods[n]),
  bracketpair(x,y)};
```

- Empirical evidence is weak
- Usefulness needs support

coming soon:

Grammar Smell Detection

Tijs van der Storm, Jurgen Vinju, Vadim Zaytsev,
SWAT, CWI @ SLE 2014

Questions?

- Summary:
  - Mining a big repo of grammars in a broad sense
  - Recognising purposeful, prevalent, simple, local patterns in grammars
  - …
  - find me: vadim@grammarware.net