DERRIC
Model-Driven Engineering in Digital Forensics

Jeroen van den Bos

jeroen@infuse.org

Centrum Wiskunde & Informatica
CWI

Nederlands Forensisch Instituut
Ministerie van Veiligheid en Justitie

DERRIC

Model-Driven Engineering in Digital Foren

Jeroen van den Bos

jeroen@infuse.org

Centrum Wiskunde & Informatica

CWI

Nederlands F

Ministerie van

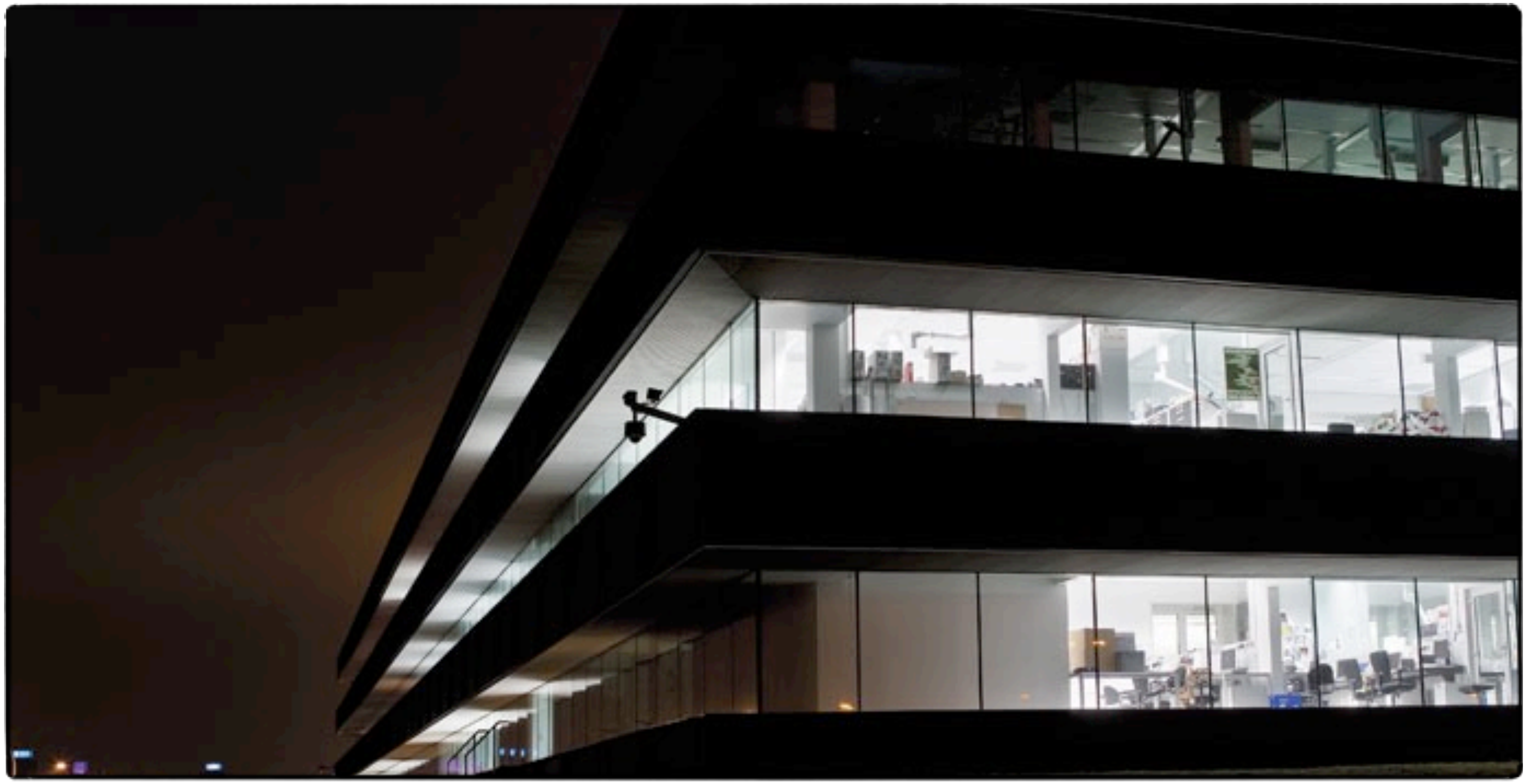
Experience:
1998: software engineer
2002: architect at NFI
2009: researcher at NFI/CWI
Education:
BICT CS/MSc SE

Contents

- How we decided to build a DSL
- What DSL we built
- How we built it

How we decided to build a DSL

Or: Domain analysis of and our experience in digital forensics



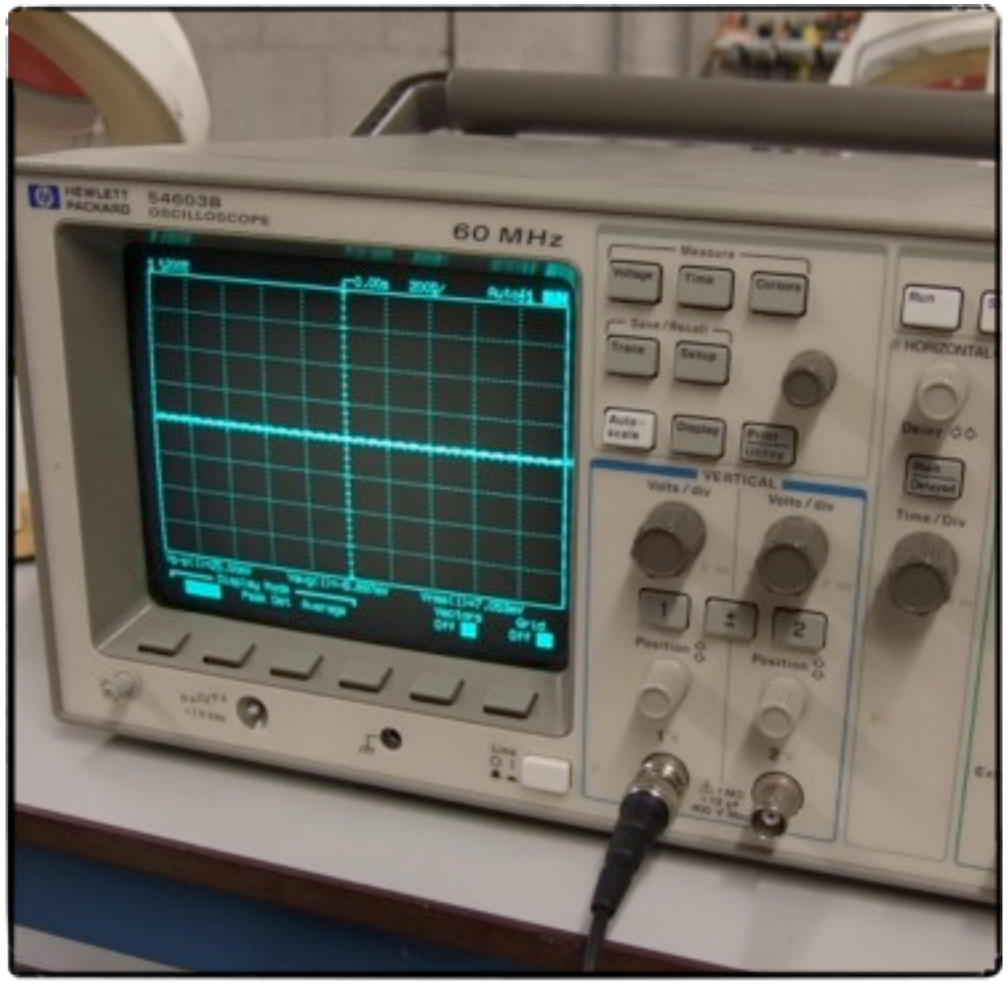
Netherlands Forensic Institute
“Improve our clients’ information position through high-quality forensic services”

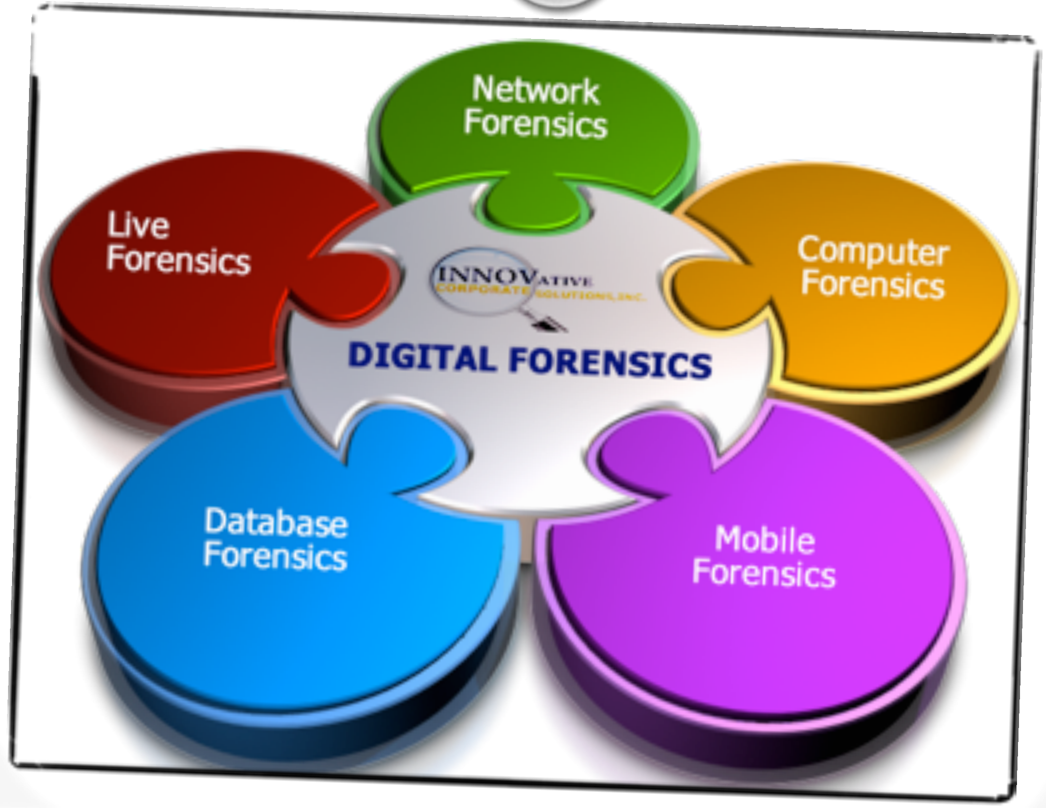
What is “digital forensics”?

From Wikipedia:
“Digital forensics is a branch of forensic science encompassing the recovery and investigation of material found in digital devices, often in relation to computer crime.”

Do we need (custom) software (engineering)?

- Software: yes, there is no other way to do digital forensics.
- Custom software: yes, because we have specific requirements.
- Software engineering: yes, for legal, business and engineering reasons.



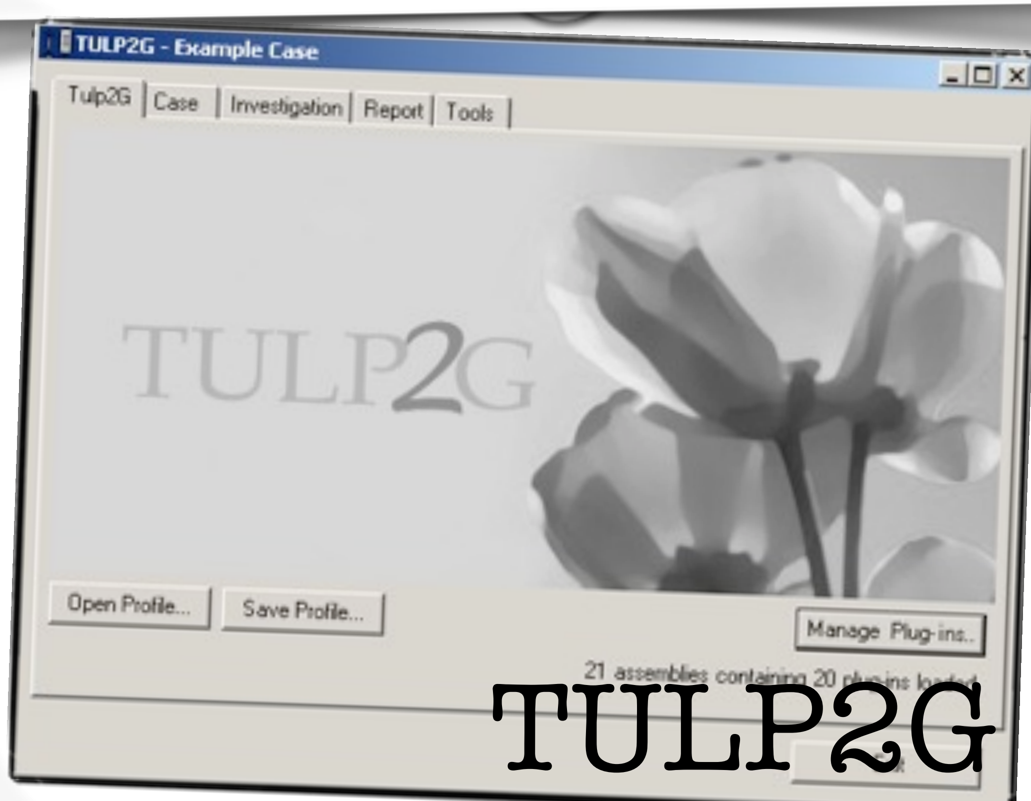


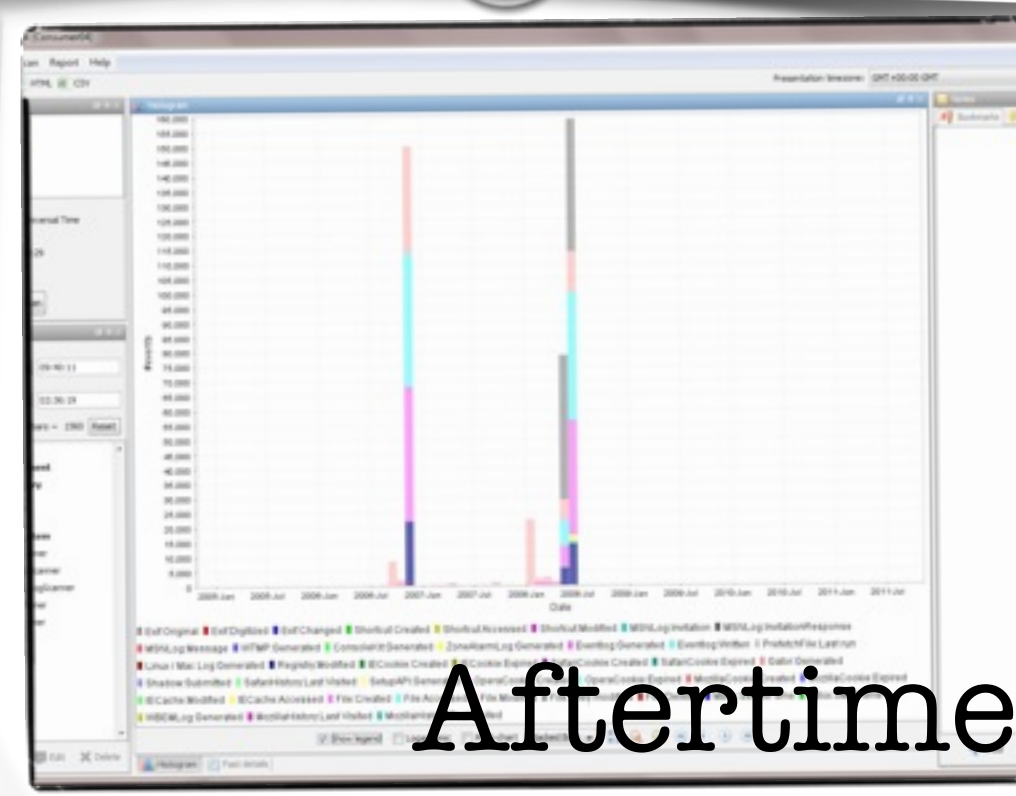
What to develop?

Stage 1 estigation preparation	a. identify the purpose of investigatio
	b. identify resources required
Stage 2 Evidence acquisition	a. identify sources of digital evidence
	b. preserve digital evidence
Stage 3 Analysis of evidence	a. identify tools and techniques to us
	b. process data
	c. interpret analysis results
Stage 4 Results dissemination	a. report findings
	b. present findings

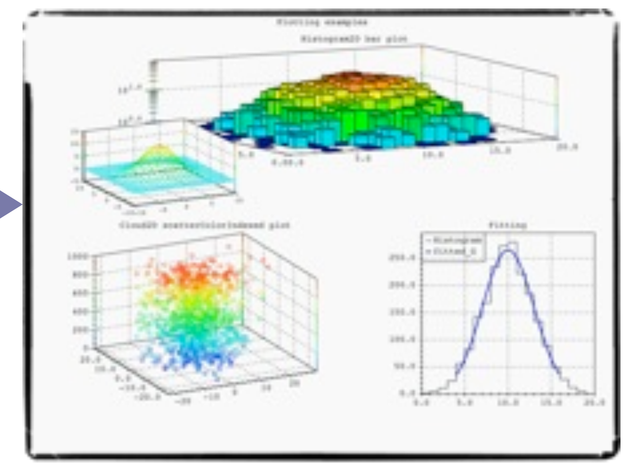

rdd-copy [local options] infile [outfile]
rdd-copy -C [client options] <local file> <remote file>
rdd-copy -S [server options]
Options:
-?, --help Print this message
-C, --client Run rdd as a network client
-F, --fault-simulation <file> simulate read errors specified in <file>
-M, --max-read-err <count> Give up after <count> read errors
-P, --progress <sec> Report progress every <sec> seconds
-S, --server Run rdd as a network server
-V, --version Report version number and exit
-b, --block-size <count>[kKmMgG] Read blocks of <count> [KMG]byte at a time
-c, --count <count>[kKmMgG] Read at most <count> [KMG]bytes
-f, --force Ruthlessly overwrite existing files
-i, --inetd rdd is started by (x)inetd
-l, --log-file <file> Log messages in <file>
-m, --min-block-size <count>[kKmMgK]
Minimum read-block size is <count> [KMG]byte
-n, --nretry <count> Retry failed reads <count> times
-o, --offset <count>[kKmMgG] Skip <count> [KMG] input bytes
-p, --port <portnum> Set server port to <port>
-q, --quiet Do not ask questions
-r, --raw Read from a raw device (/dev/raw/raw[0-9])
-s, --split <count>[kKmMgG] Split output, all files < <count> [KMG]bytes
-v, --verbose Be verbose
-z, --compress Compress data sent across the network
-H, --histogram <file> Store histogram-derived stats in <file>
-h, --histogram-block-size <size>
Histogramming block size
--checksum, --adler32 <file> Compute and store Adler32 checksums in <file>
--checksum-block-size, --adler32-block-size <size>
Adler32 uses <size>-byte blocks
--crc32, --crc32 <file> Compute and store CRC32 checksums in
--crc32-block-size, --crc32-block-size <size>
CRC32 uses <size>-byte blocks
--md5, --md5 Compute and print MD5 hash
RDD


Defraser


TULP2G - Example Case
Tulp2G
Case
Investigation
Report
Tools
TULP2G
Open Profile...
Save Profile...
Manage Plug-ins...
TULP2G


Aftertime

Main activities



Acquisition

Securing the data

Recovery

Turning data into information

Analysis

Finding relevant information

One clear bottleneck

- Data acquisition
  - Hardware and hardware interfaces are slow-moving.
- Data recovery
  - New platforms, apps and versions emerge daily.
  - Lots of variants due to vendor-specific implementations.
  - Underlying storage approaches are slow-moving.
- Data analysis
  - Analyses and types of questions are slow-moving.

Critical factors

- High time pressure
- Performance/ scalability concerns
- Last-minute tool modifications
- Changes required in several places
- Division of work

Requirements

1. Data structure definitions that are easy to develop and modify.

2. Highest possible runtime performance and scalability.

3. Reuse of changes across applications.

4. Separation forensic investigation and software engineering concerns.

What DSL we built

Or: DERRIC, a user’s perspective

A DERRIC description

1. Header
Name and encoding/ type defaults

```
Format PNG

strings ascii
size 1
unit byte
sign false
type integer
order lsb0
endian little
```

2. Sequence
Data structure ordering

```
Sequence

Signature
IHDR
(ITXT ICMT)*
PLTE?
IDAT
IDAT*
IEND
```

3. Structures
Layout of individual data structures

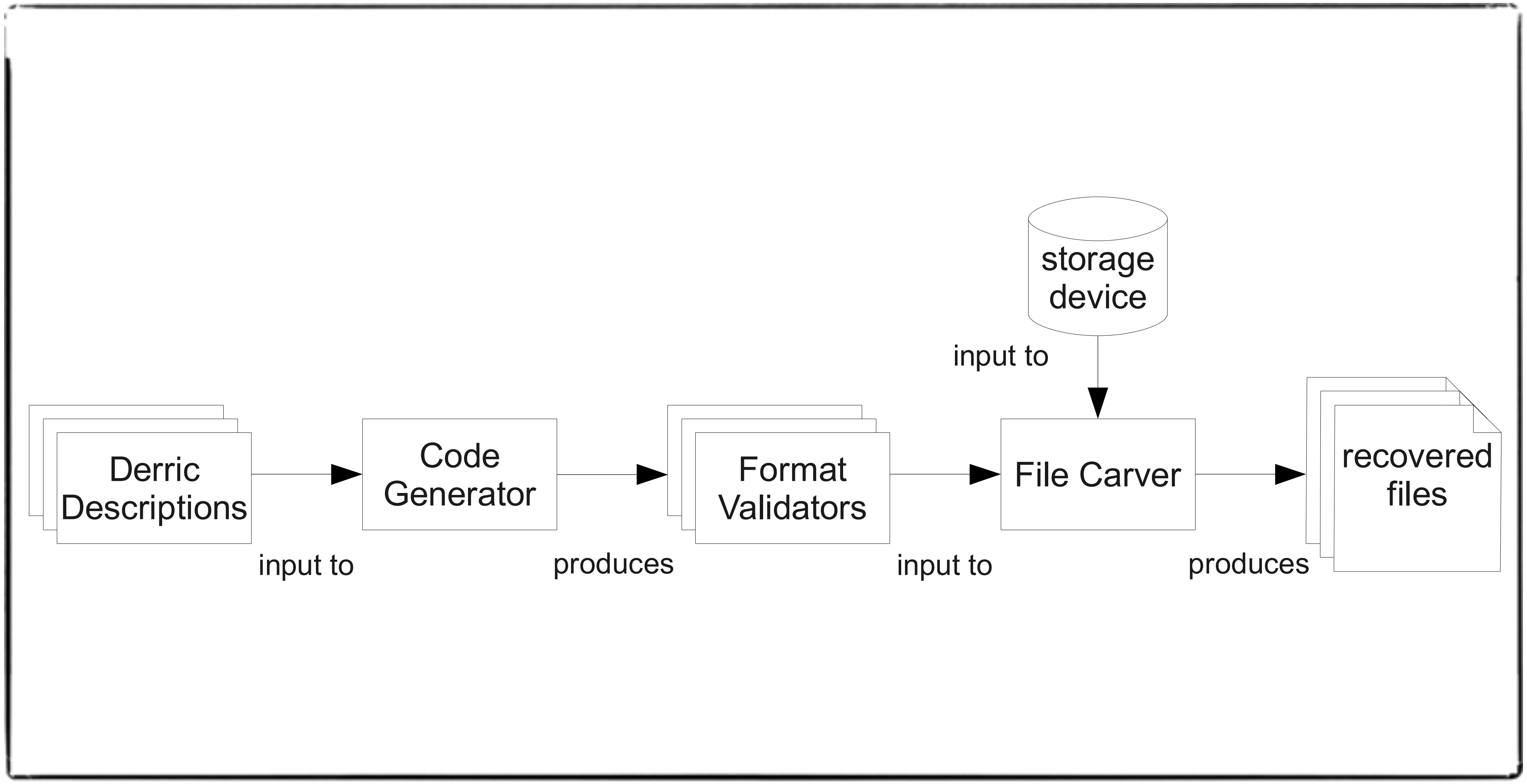
```
Structures

IHDR {
 l: lengthOf(d)
      size 4;
 n: “IHDR”;
 d: { ... }
 c: checksum
(...) size 4; }
```

Structures

```
Chunk {
  length: lengthOf(chunkdata) size 4;
  chunktype: type string size 4;
  chunkdata: size length;
  crc: checksum(algorithm="crc32-ieee",
              fields=chunktype+chunkdata) size 4;
}
IHDR = Chunk {
  chunktype: "IHDR";
  chunkdata: {
    width: !0 size 4;
    height: !0 size 4;
    bitdepth: 1|2|4|8|16;
    imagesize: (width*height*bitdepth)/8 size 4;
    colourtype: 0|2|3|4|6;
    interlace: 0|1;
  }
}
```

Applying Derric

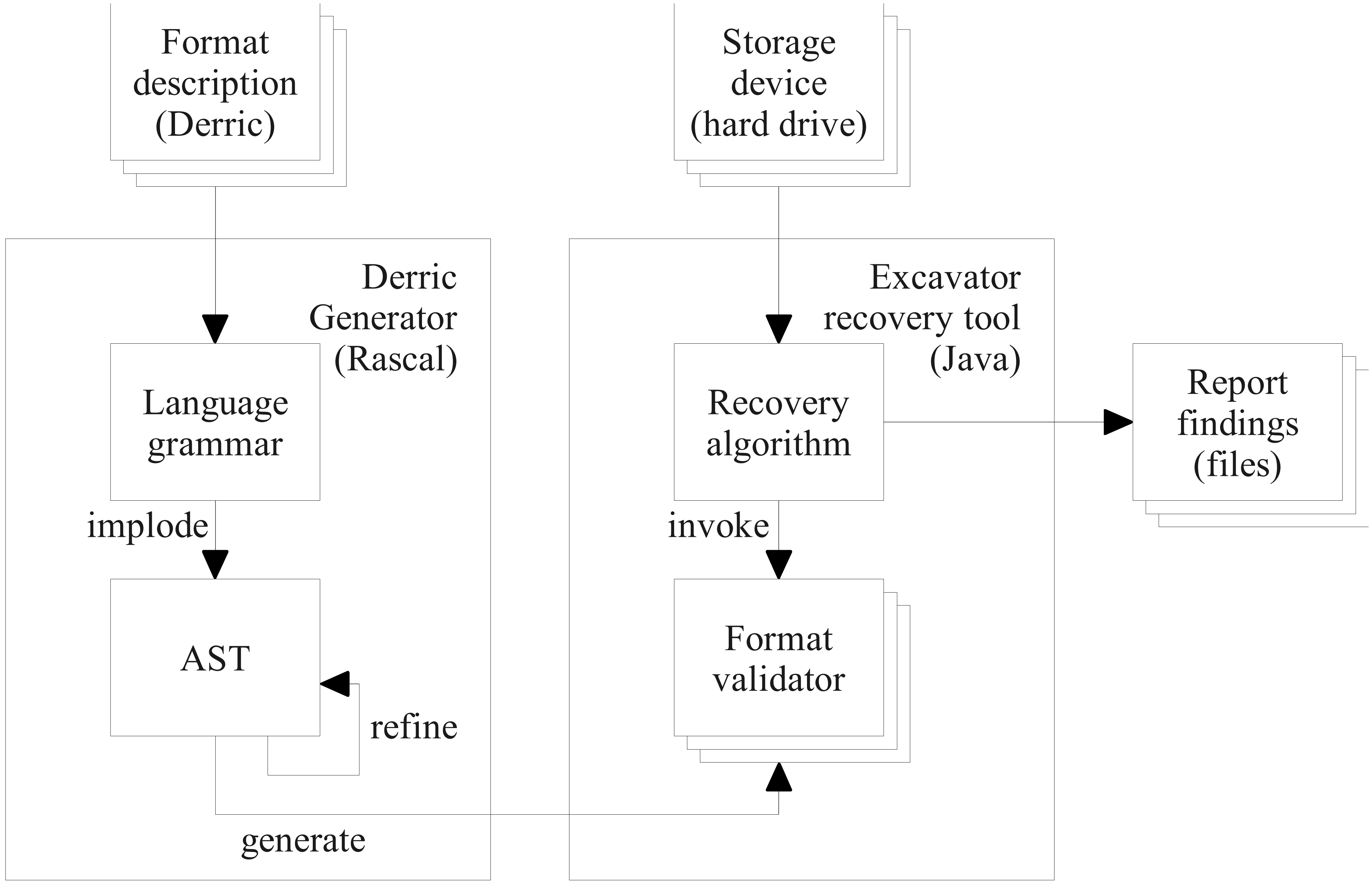
- Each format has one/several descriptions.
- Code generator uses descriptions:
  - Applies (domain-specific) optimizations/transformations.
  - Runs quickly, so easy to rerun after changes.
  - May output for multiple targets.
- Runtime application uses components:
  - Recognizes, extracts or ignores files.
  - Implements algorithms and common optimizations.



Excavator/Derric architecture

How we built it

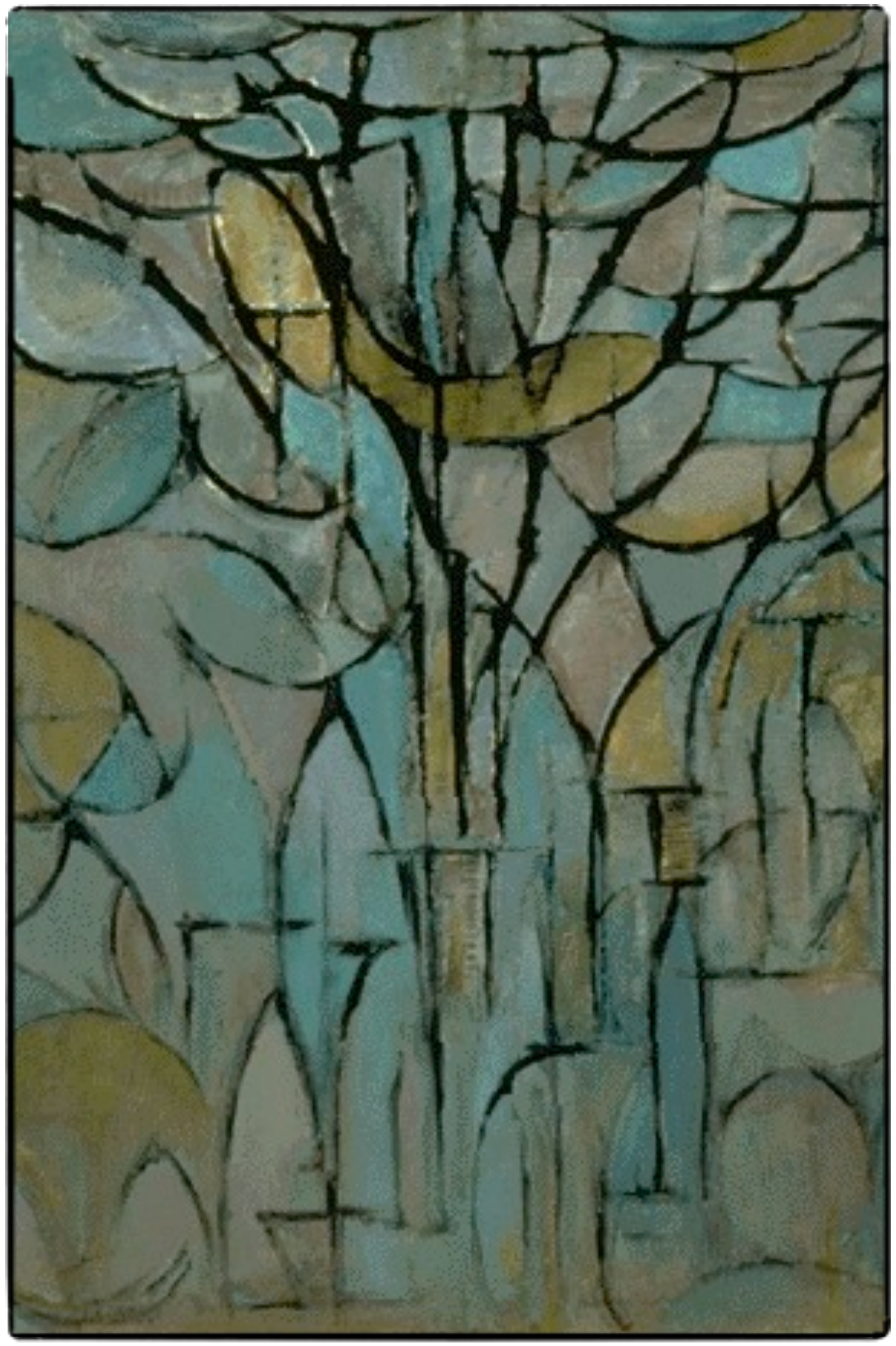
Or: Implementing a DSL



Format description (Derric)
Storage device (hard drive)
Derric Generator (Rascal)
Excavator recovery tool (Java)
Language grammar
implode
AST
refine
generate
Recovery algorithm
invoke
Format validator
Report findings (files)

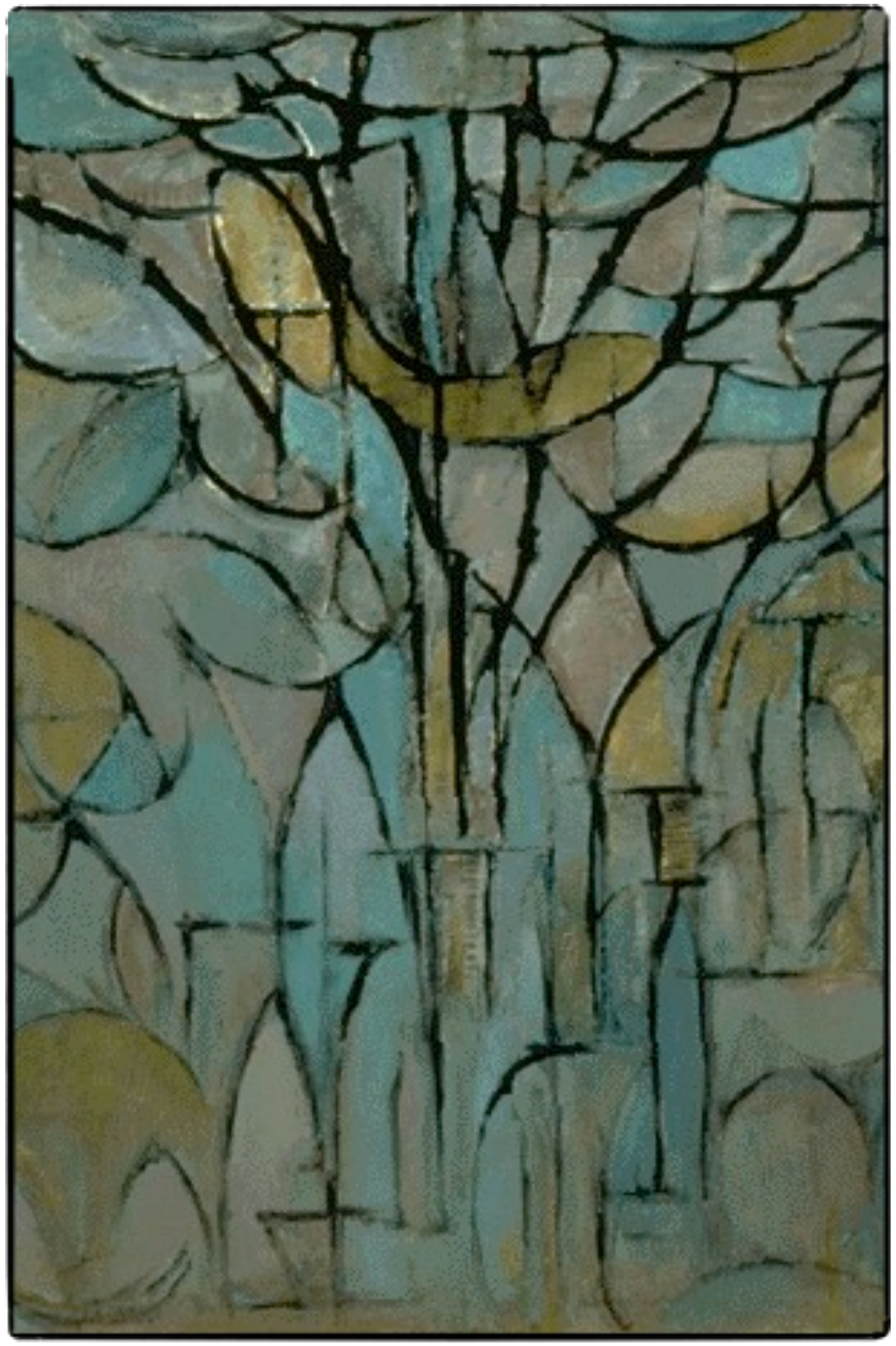
Compilation stages

1. Parse (concrete syntax)
2. Implode (FileFormat AST)
3. Propagate defaults
4. Desugar (x5)
5. Propagate constants
6. Fold constants (5/6 chg? 5)
7. Annotate expressions
8. Transform (Validator AST)
9. Generate code (Java)



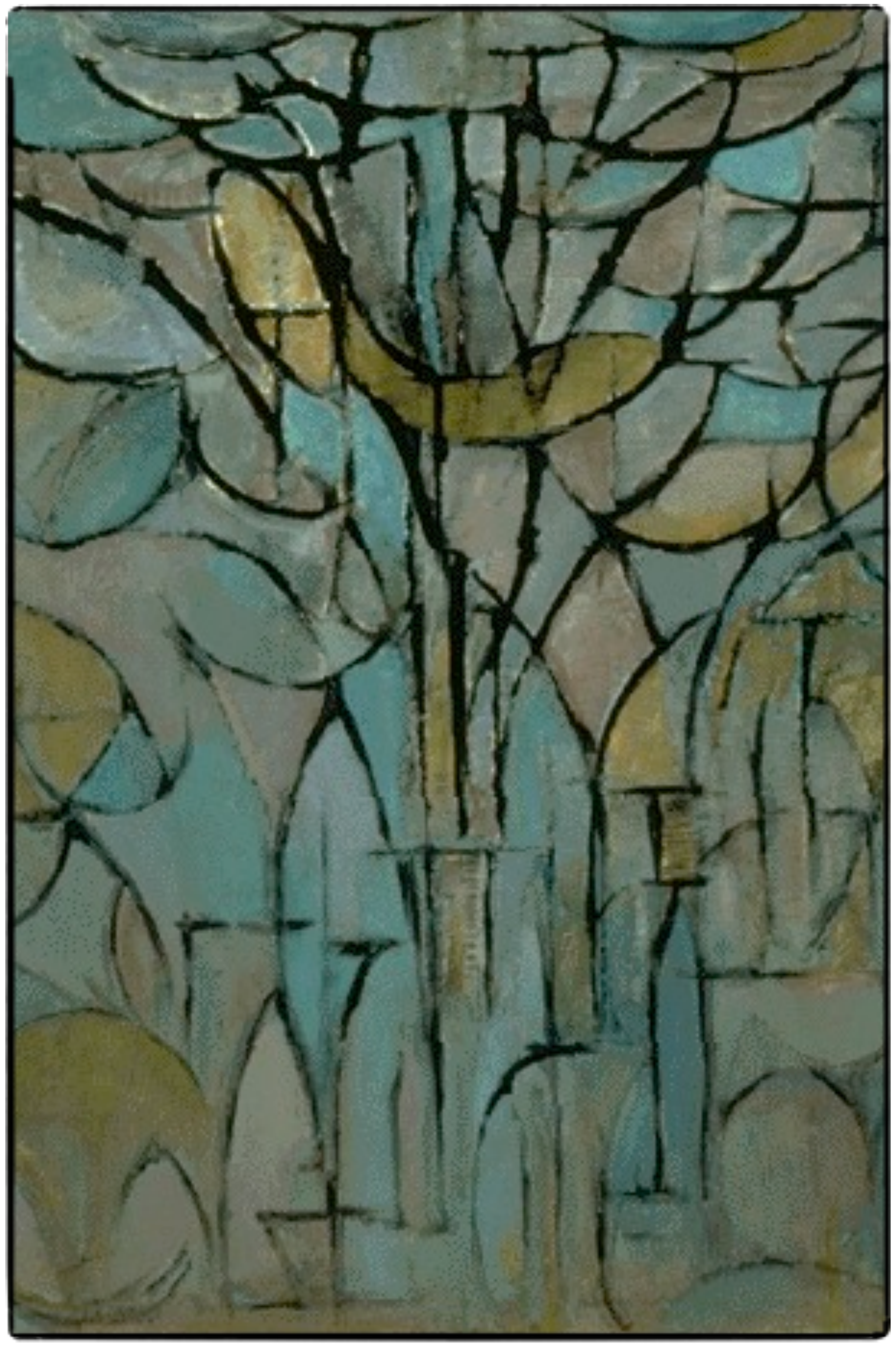
Tree transforms

1. Parse (concrete syntax)
2. Implode (FileFormat AST)
3. Propagate defaults
4. Desugar (x5)
5. Propagate constants
6. Fold constants (changed? 5)
7. Annotate expressions
8. Transform (Validator AST)
9. Generate code (Java)



Tree refinements

1. Parse (concrete syntax)
2. Implode (FileFormat AST)
3. Propagate defaults
4. Desugar (x5)
5. Propagate constants
6. Fold constants (changed? 5)
7. Annotate expressions
8. Transform (Validator AST)
9. Generate code (Java)



Tree refinement pattern

- Input tree.
- Build data structure with required knowledge.
- Traverse tree, removing/modifying nodes.
- Return tree.

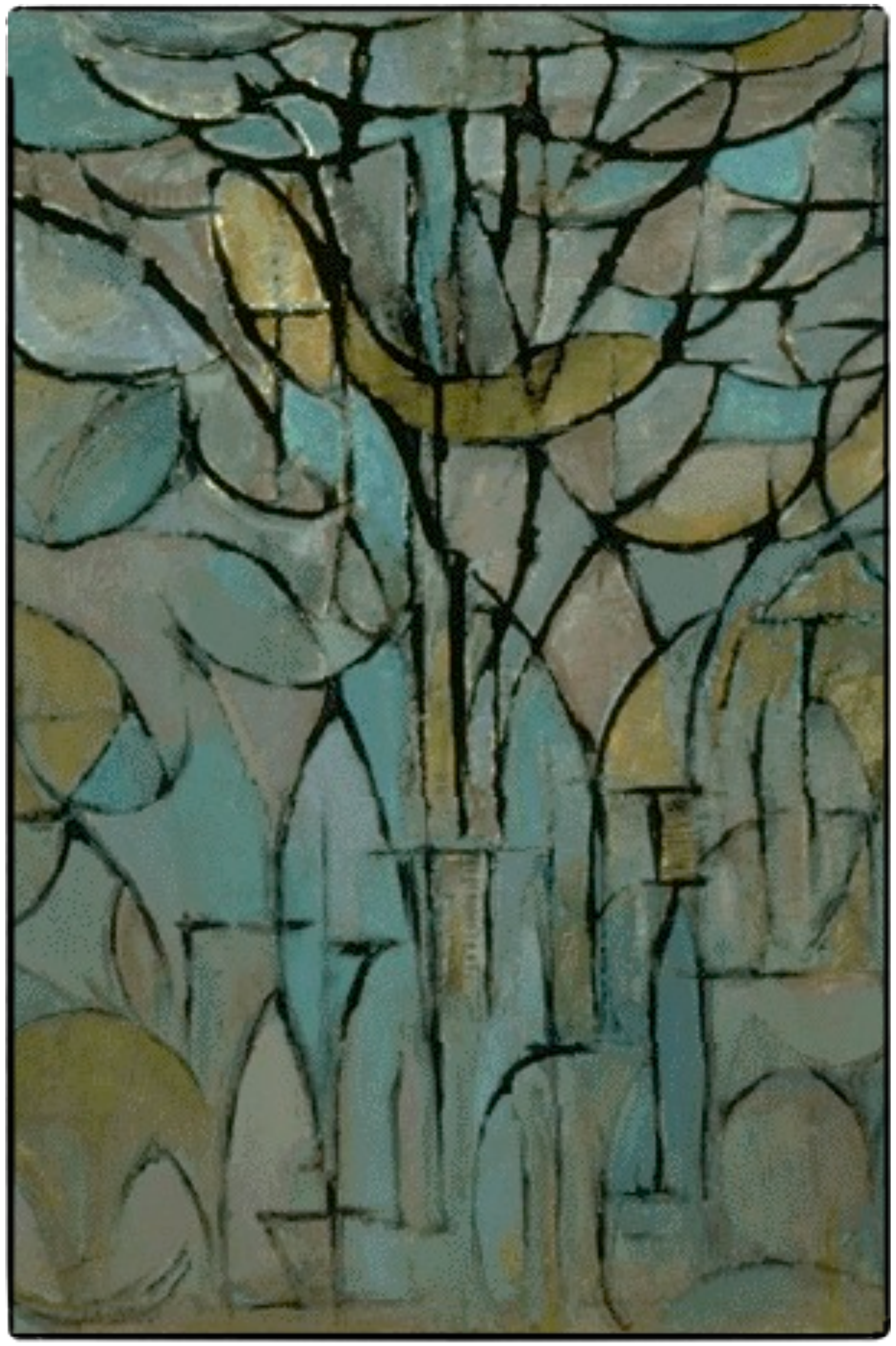
```
private AST performRefinement(AST ast) {

  rel[str, str, str, int] env = { };
  for (term <- ast.terms) {
    // build the environment
  }

  Term fixTerm(Term term) {
    // transform node using environment
    return term;
  }

  return top-down visit(ast) {
    case t:term(list[Field] fields) =>
      fixTerm(t)
    case t:term(str name, list[Field] fields) =>
      fixTerm(t)
  }

}
```

Refinement examples

- Remove inheritance
  - Find root, walk back
  - Overwrite/add encountered fields
- Remove offset()-keyword
  - For each field, determine all preceding fields
  - Replace offset()s with lengthOf()s of preceding fields
- Remove strings
  - Replace each string with a list of byte values obtained from an encode function.

Tree annotations

1. Parse (concrete syntax)
2. Implode (FileFormat AST)
3. Propagate defaults
4. Desugar (x5)
5. Propagate constants
6. Fold constants (changed? 5)
7. Annotate expressions
8. Transform (Validator AST)
9. Generate code (Java)



Structures

```
Chunk {
  length: lengthOf(chunkdata) size 4;
  chunktype: type string size 4;
  chunkdata: size length;
  crc: checksum(algorithm="crc32-ieee",
              fields=chunktype+chunkdata) size 4;
}
IHDR = Chunk {
  chunktype: "IHDR";
  chunkdata: {
    width: !0 size 4;
    height: !0 size 4;
    bitdepth: 1|2|4|8|16;
    imagesize: (width*height*bitdepth)/8 size 4;
    colourtype: 0|2|3|4|6;
    interlace: 0|1;
  }
}
```

Source

```
length:
    lengthOf(chunkdata)
    size 4
    @checkAfter(chunkdata)
chunkdata:
    unknown
    size length
    @ref(global)
crc:
    checksum
    size 4
```

Target

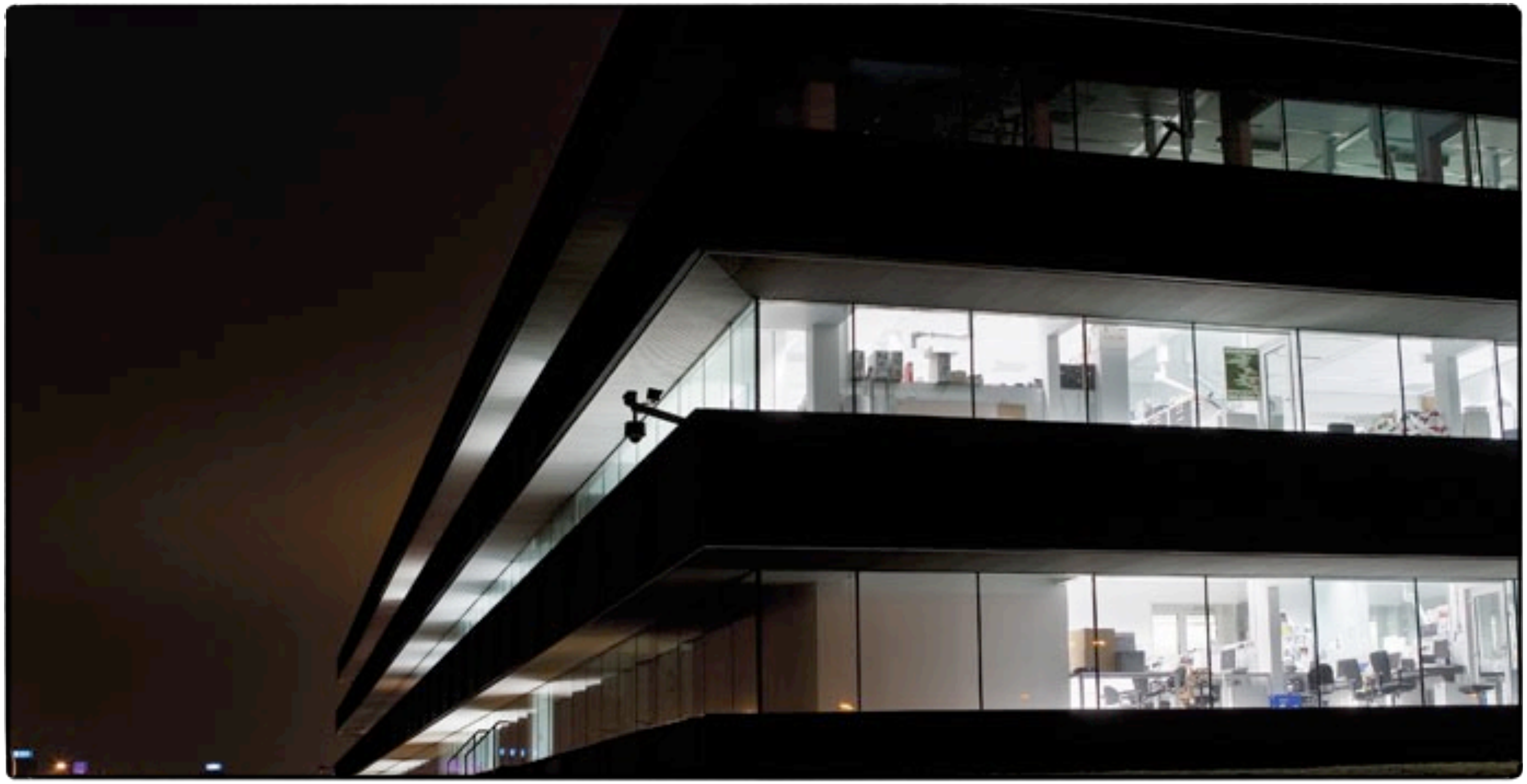
```
length:
    declare variable l
    read into l
    declare variable l2
    calculate l2
chunkdata:
    declare global variable d
    read into d
    check lengthOf(d) = l2
crc:
    declare variable c
    read into c
    declare variable c2
    calculcate c2
    check c = c2
```

Rules of thumb

- Transform from representation to representation until you reach target
  - For each node, a simple (typically one-line) transformation
- When too complex, refine current representation
  - A single step at a time
- Use annotations to guide transformations

General conclusions

- Don’t invest in DSL/MDE unless you have extensive background in the problem domain.
- Software is hard, metasoftware is harder.
- Implement DSLs using a large amount of individual steps that are themselves simple.
- It actually works!



Questions?
Jobs @ NFI: http://www.holmes.nl/