Talking ‘bout Code Generation

Nicola Musatti

Nicola.Musatti@gmail.com

ACCU Conference 2008

Copyright 2008 Nicola Musatti

“I would rather write programs to help me write programs than write programs.”

Richard L. Sites

In our job we’re sometimes faced with tedious activities. Tedium is often a good indication that automation is possible

Copyright 2008 Nicola Musatti

Programme

- An introduction to code generators
- The design of code generators
- A case study: Perceval
- Tools and techniques
- Lessons learned
- Conclusions

Copyright 2008 Nicola Musatti

Scope of this presentation

- Practical advice drawn from direct experience
- Little or no theory
- No rocket science
  - You already had too much of it anyway ;-)
- No CASE, Generative Programming, MDA, etc.
  - No connection with the Code Generation conference, but what a coincidence!
- Some Python, some C++

Copyright 2008 Nicola Musatti

A Success Story

- A 4,000,000+ code lines application was ported to an altogether different platform, with a different RDBMS, by a single person.
  - From IBM System I (aka iSeries, aka AS/400) to Windows Server 2003
  - From DB/2 to Oracle
- ***How?***

Copyright 2008 Nicola Musatti

A Success Story

- A 4,000,000+ code lines application was ported to an altogether different platform, with a different RDBMS, by a single person.
  - From IBM System I (aka iSeries, aka AS/400) to Windows Server 2003
  - From DB/2 to Oracle
- ***How?***
  - By coding in a truly portable language
    - Hint: it's not Java…

Copyright 2008 Nicola Musatti

A Success Story

- A 4,000,000+ code lines application was ported to an altogether different platform, with a different RDBMS, by a single person.
  - From IBM System I (aka iSeries, aka AS/400) to Windows Server 2003
  - From DB/2 to Oracle
- ***How?***
  - By coding in a truly portable language
    - Hint: it's not Java… It's COBOL!

Copyright 2008 Nicola Musatti

A Success Story

- A 4,000,000+ code lines application was ported to an altogether different platform, with a different RDBMS, by a single person.
  - From IBM System I (aka iSeries, aka AS/400) to Windows Server 2003
  - From DB/2 to Oracle

➢ ***How?***
  - By coding in a truly portable language
    - Hint: it's not Java… It's COBOL!
  - By converting automatically the relatively few differences, i.e. by writing a code generator!

Copyright 2008 Nicola Musatti

Another Success Story

- At the onset of a new C++ project in early 1999 we decided to write our own persistence layer
- It meant we had to write the interface for all the 35 tables in our database
- Instead I convinced our team leader that we should write a code generator
- In over seven years our application grew a little:
  - Over 300 tables
  - Over 700,000 lines of code
  - Of which some 75,000 automatically generated!
  - But I'll tell you later about the one mistake I made…

Copyright 2008 Nicola Musatti

Code Generators are everywhere

- The compiler is a code generator
- Your favourite IDE’s wizards are code generators
- Web applications are often code generators of sorts
- Similar techniques may be applied to other areas of software development
  - Basically, each time you either need to retrieve information from text or to provide information in textual form!

Copyright 2008 Nicola Musatti

So why doesn't everybody use them?

- Code Generators are a kind of meta-programming
  - The step to a meta level is unsettling for some people
  - They may introduce an additional language/toolset combination
- Reflection and generic programming reduce the use of CG's
  - C++ template meta-programming solves many of the problems
  - Dynamic languages let you define types at run-time
- However, the resulting code is often more complicated

Copyright 2008 Nicola Musatti

Intensive vs. Extensive Programming

- Intensive code has:
  - Perfect factoring: Every concept is expressed once
  - Awful localization: The basic steps of a complex operation are scattered everywhere
- Extensive code has:
  - Awful factoring: lots of repetition
  - Good localization: you see what is happening
- Code generators may give you the best of both worlds

Copyright 2008 Nicola Musatti

Our First Code Generator

```
#include <cctype>
#include <iostream>
#include <string>

int main()
{
    std::string s;
    std::cout << "Who do you want to greet? ";
    std::getline(std::cin, s);
    s[0] = toupper(s[0]);
    std::cout << "#include <iostream>\n";
    std::cout << "\n";
    std::cout << "int main()\n";
    std::cout << "{\n";
    std::cout << "\tstd::cout << \"Hello, " + s + "!\\n\";\n";
    std::cout << "}\n";
}
```

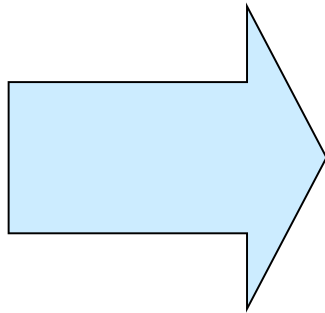

Copyright 2008 Nicola Musatti

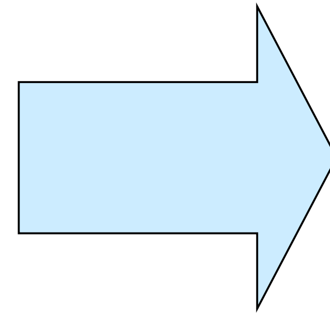
Structure of a code generator

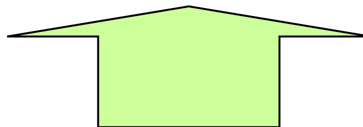
Acquisition

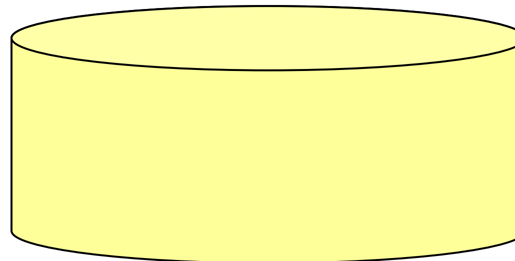
Transformation

Generation









Schema

Copyright 2008 Nicola Musatti

Acquisition

- Acquisition is the collection of the information that characterizes each instantiation of the schema patterns
- It is often dominated by the infrastructure required to handle the external representation of the information
- Some examples:
  - Parsing a textual description
    - Code, perhaps?
  - Querying a database
    - To explore its schema
    - To retrieve project wide conventional values

Copyright 2008 Nicola Musatti

Transformation

- Transformation builds an internal representation of the collected information
- Specific domains have consolidated representations
  - e.g. compilers have augmented syntax trees
- In general, however, you’re on your own

Copyright 2008 Nicola Musatti

Generation

- Generation navigates the internal representation and “splices” the information onto the schema patterns to produce an original instance
- This area too is dominated by the infrastructure required to handle the external representation
- Ideally it should combine the internal representation with a structured representation of the desired output

Copyright 2008 Nicola Musatti

Passive vs. Active Generators

- Hunt and Thomas [Hunt2000] distinguish *passive* from *active* generators
- Passive generators can only be run once for each output instance
- Active generators may be executed as many times as desired

Copyright 2008 Nicola Musatti

Passive Generators

- Usually generate code that requires manual modification
  - IDE wizards are a typical example
- Good for boilerplate code
  - Company standard header comments
  - Custom IDE project types
- Good for getting the easy 80% done quickly

Copyright 2008 Nicola Musatti

Active generators

- May produce successive versions of an output instance
  - GUI builders should be active generators!
- The key is the separation between the generated code and its customizations
  - “Do not modify above this line” is a very fragile approach

Copyright 2008 Nicola Musatti

The GENERATION GAP pattern

- Formulated by the GoF and documented byJohn Vlissides [Vlissides1998]
- Put the generated code in a base class
- Customize it by subclassing it
- Examples:
  - ICS's Builder xCessory
  - C# partial classes

Copyright 2008 Nicola Musatti

Our case study: Perceval

- A C++ Object Relational mapping
- For each DB table there are:
  - A class that directly represents it: one column – one data member
  - A set of class templates that handle reading and writing
  - A factory/container of instances that handles caching and on demand reading
  - An internal representation class that uses the persistent state
  - A factory of such internal representation
  - A GUI/report oriented representation class
  - Possibly a GUI frame

Copyright 2008 Nicola Musatti

The “Fake Template” idiom

- The declaration is generic
- All implementations are specific
- Good for providing building blocks for generic programming
- Extremely tedious

Copyright 2008 Nicola Musatti

The original DB table

```
create table cambio_maf (
  id                   numeric(11)      identity,
  cod_cambio           char(10)         not null,
  dta_reference        datetime         not null,
  num_exchange_rate    numeric(30,10)   not null,
  data_load            datetime         not null,
  constraint cambio_maf42 primary key (id)
)
go
```


Copyright 2008 Nicola Musatti

The corresponding persistent class

```
class CambioMaf : public Owf::Port::Pers::Persistent
{
    public:
        std::string         getCodCambio() const { return codCambio; }
        Owf::DateTime       getDtaReference() const { return dtaReference; }
        Owf::Pers::Money    getNumExchangeRate() const { return numExchangeRate; }
        Owf::DateTime       getDataLoad() const { return dataLoad; }

        void                setCodCambio(const std::string & arg) { codCambio = arg; }
        void                setDtaReference(Owf::DateTime arg) { dtaReference = arg; }
        void                setNumExchangeRate(Owf::Pers::Money arg)
                                                               { numExchangeRate = arg; }
        void                setDataLoad(Owf::DateTime arg) { dataLoad = arg; }

    private:
        friend class Owf::Port::Pers::PRecord<CambioMaf>;

        std::string         codCambio;
        Owf::DateTime       dtaReference;
        Owf::Pers::Money    numExchangeRate;
        Owf::DateTime       dataLoad;
};
```


Copyright 2008 Nicola Musatti

A first try: handwritten C++

- Formal language recognition is one of the most well understood fields in computer science
- The architecture is standard
  - Valid “words” are recognized by a lexical analyzer, or scanner
  - The “sentences” in which they are combined are recognized by a parser

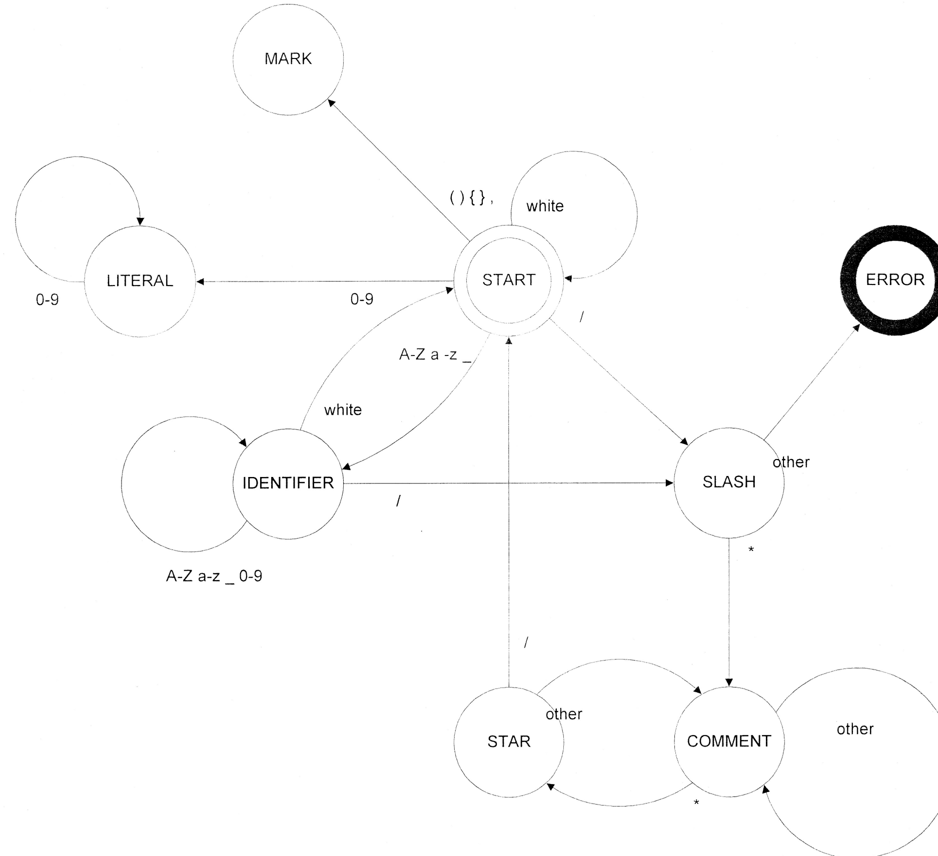
Copyright 2008 Nicola Musatti

The scanner

- The primitive elements of programming languages can be recognized by regular expressions
  - E.g. identifiers: [A-Za-z_][A-Za-z_0-9]*
- Simple ones are easily implemented by hand:
  - Draw the Deterministic Automaton diagram
  - Code the loop around a switch, with one case per state

Copyright 2008 Nicola Musatti

A Deterministic Finite Automaton



Copyright 2008 Nicola Musatti

The parser

- Two big families
  - Bottom up are great for automatic generation
  - Top down are more easily coded by hand
- There are caveats
  - It's easy to introduce infinite recursion
  - Some languages take a long time to parse (On^2)
- Languages should be kept simple
  - Python is among the simplest ones
  - Java is simple enough for tools to handle it
  - C++ is extremely hard

Copyright 2008 Nicola Musatti

Hand writing the parser

- SQL DDL is a tractable language
  - Declarative languages often are
- Recursive descent is simple and regular
  - One function (or class) for each symbol
  - If the language is recursive, so is the parser
- “The parser is the AST”
  - One class per symbol
  - Each symbol’s children are recognized in the parent’s constructor

Copyright 2008 Nicola Musatti

Generation

- Make a “tracer bullet”: hand code a working example of the whole source code sequence for a single table
- Prepend “std::cout <<“ to all your source lines!
- Seek a methodology that limits the modifications you need to apply to your reference code

Copyright 2008 Nicola Musatti

Here my trouble began

- We switched from Sybase ASE to Sybase ASA
  - Same supplier, different syntax
  - It turned out to be simpler to convert the syntax than to change the generator
- Over time we thought of improvements to our reference architecture
  - Wading through a miriad of ‘std::cout << …;’ statements wasn’t any fun

Copyright 2008 Nicola Musatti

Code Generators revisited

- Beware of change: C++ handwriting may be OK for version 1.0, but takes too long to maintain
- It's also a matter of timing: at the onset of a new project all deadlines appear so far away…
- Why don't we also…
  - If you have a regular architecture, there's always at least one other thing you can generate from the same data

Copyright 2008 Nicola Musatti

The right approach

- Establish an architecture!
- Acquisition: use a parser generator, or at least a well devised set of regular expressions
- Generation: use a macro processor, or a template engine
- Overall: use an agile language
  - If performance is an issue for your generator, you're in deep trouble

Copyright 2008 Nicola Musatti

Perceval 2.0

- Acquisition: use a parser generator
  - PLY: A Python implementation of Lex & Yacc
- Generation: use a template engine
  - Cheetah: A non XML, non HTML specific engine
- Python is the language
  - Pleasant to code in
  - Maintainable
  - “Batteries included” and many more within easy reach

Copyright 2008 Nicola Musatti

Conclusions

- Writing code generators is easy!
  - The domain is our own
  - We control the requirements… sort of
- Use tools!
- Use a suitable language!

Copyright 2008 Nicola Musatti

Q & A

Copyright 2008 Nicola Musatti

Bibliography

[Hunt1999]	Hunt, A. Thomas, D., *The Pragmatic Programmer*, Addison Wesley 1999
[Vlissides1998]	Vlissides, J., *Pattern Hatching – Design Patterns Applied*, Addison Wesley, 1998
PLY	http://ply.dabeaz.com
Cheetah	http://www.cheetahtemplate.org

Copyright 2008 Nicola Musatti

Colophon

- The presentation title was inspired by the refrain from The Who's first and greatest hit single: "My Generation"
- The presentation scheme was inspired by the work of Swiss artist Max Bill
- This colophon was inspired by the ones you find in all O'Reilly books :-)

Copyright 2008 Nicola Musatti