Generative Programming

Paolo Costa

paolo.costa@polimi.it

Outline

- Introduction
- Generative Programming
- Domain Engineering
- Generic Programming
- Code Generators:
  - Static Template Metaprogramming
  - Dynamic Metaprogramming
  - Intentional Programming

Industrial Revolution

1826
interchangeable parts
successfully introduced
by John Hall (after 25 years
of unsuccessful attempts!)

1901
assembly lines
introduced
by Ransom Olds;
popularized and refined
by Henry Ford in 1913

1980s
automated assembly lines
first industrial robot
installed in 1961
at General Motors;
1970's advance of
microchips

The Current State of Software

“You get all the parts necessary to assemble the car yourself. Actually, maybe some of the parts are not a perfect fit, and you have to do some cutting and filing to make them fit”

Product Line Architecture

“The programmer states what he wants in abstract terms and the generator produces the desired system or component”

The Vision

- Think & program
  - ➔ *“one of a kind” programming*
- Survey & assembly
  - ➔ *component-based programming*
- Order & generate
  - ➔ *generative programming*

Generative Programming

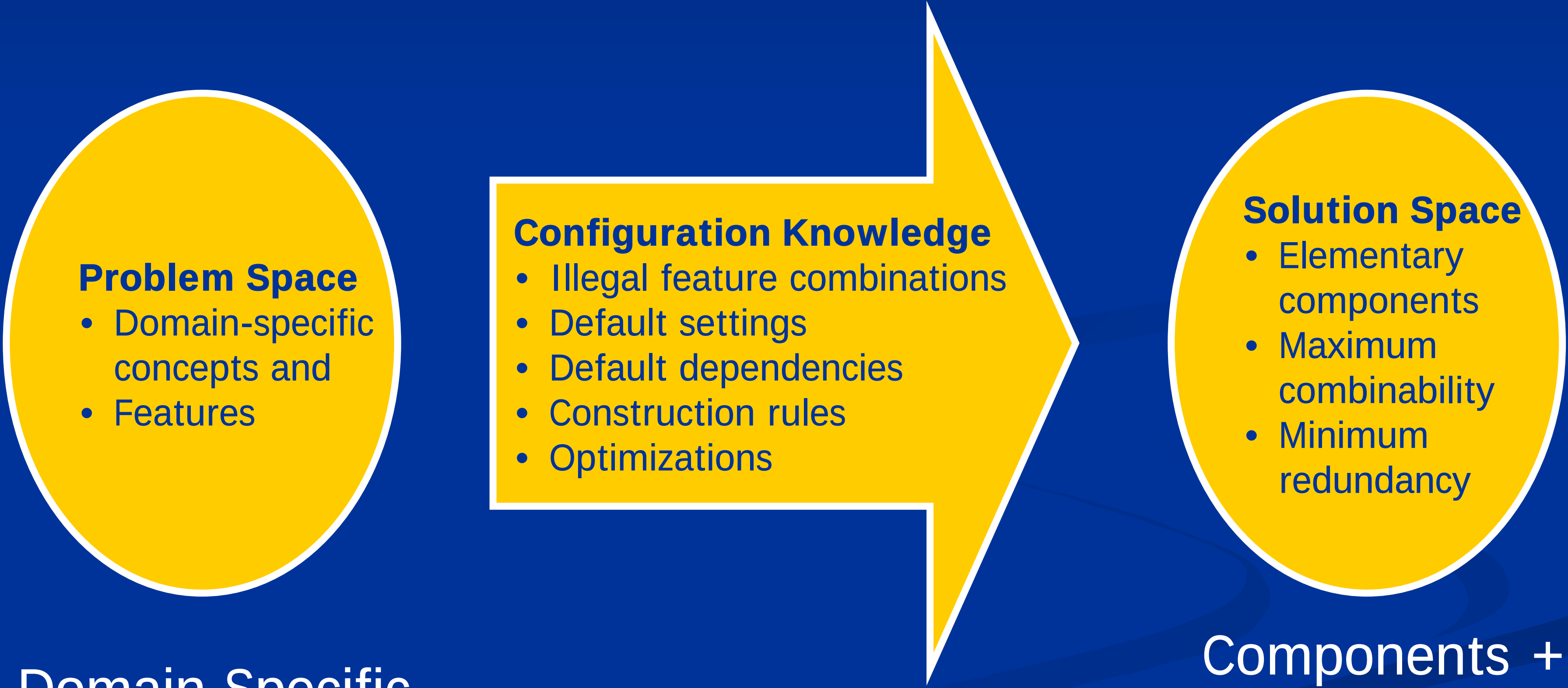
- A software engineering paradigm in which
  - given a particular requirements specification
  - a highly customized and optimized intermediate or end-product
  - can be automatically manufactured on demand
  - from elementary, reusable implementation components
  - by means of configuration knowledge

[Carnecki & Eisenecker, 2000]

Three Fundamental Steps

1. Design the implementation components to fit a common product-line architecture
2. Model the configuration knowledge starting how to translate abstract requirements into specific set of components
3. Implement the configuration knowledge using generators

Generative Domain Model



Problem Space
- Domain-specific concepts and
- Features

Configuration Knowledge
- Illegal feature combinations
- Default settings
- Default dependencies
- Construction rules
- Optimizations

Solution Space
- Elementary components
- Maximum combinability
- Minimum redundancy

Domain Specific Languages (DSLs)

Generators

Components + System Family Architecture

Why current SE methodology is not enough

- No distinction between engineering for reuse and engineering with reuse
- No domain scoping phase
- No differentiation between modeling variability within one application and between several application
- No implementation-independent means of variability modeling

A Meta-Paradigm

- Aspect-Oriented Programming
- Subject-Oriented Programming
- Software Transformation Technologies
- Domain Engineering
- Generic Programming
- Code Generators
- Intentional Programming

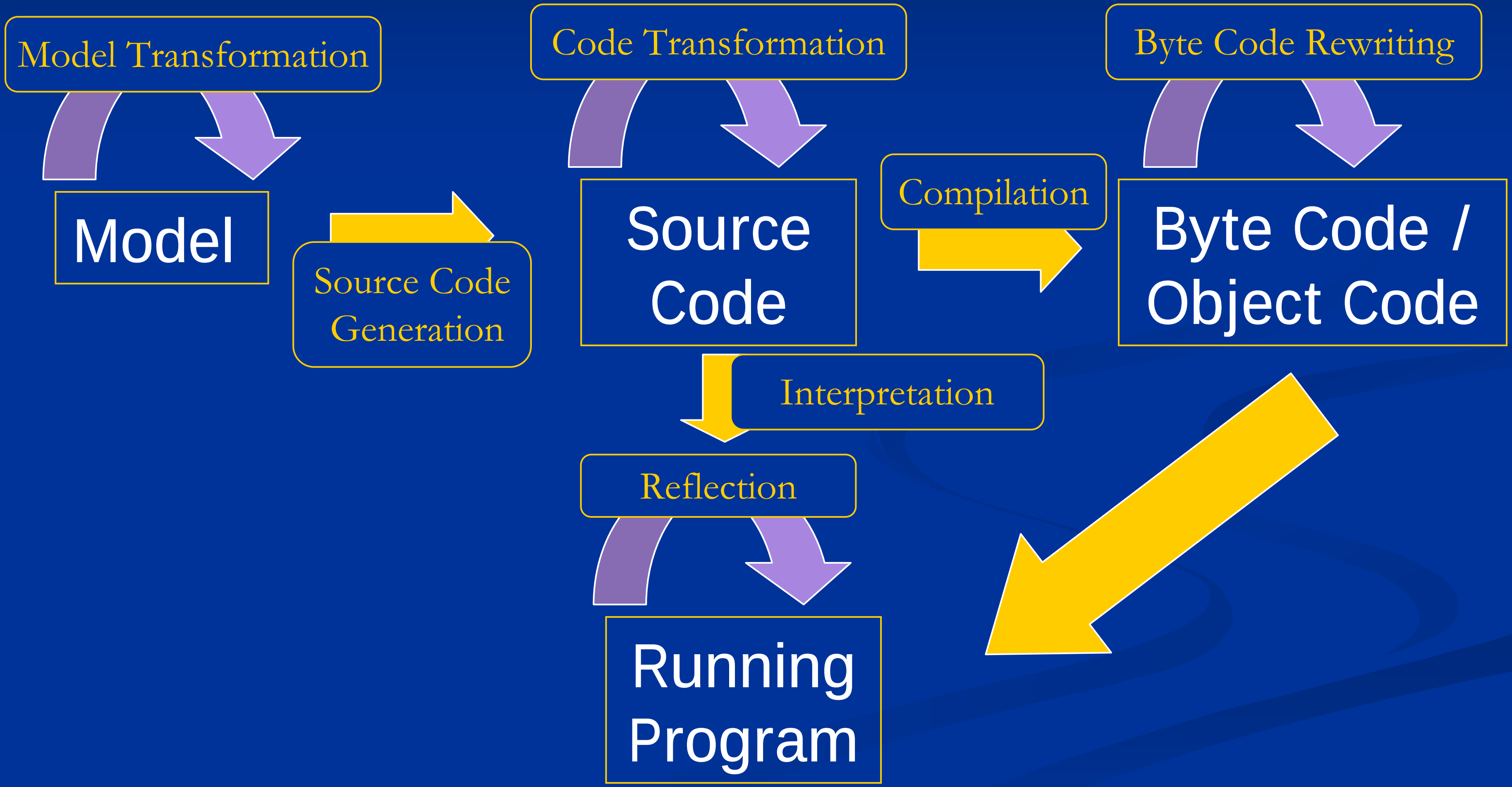
System Family Approach

- Domain Engineering (development for reuse)
  - analysis: scoping, common and variable features, feature dependencies (*FODA feature diagrams*)
  - design: common architecture for the system family domain-specific languages, configuration generators
  - implementation: reusable components, domain-specific languages, configuration generators
- Application Engineering (development with reuse)
  - production of concrete, highly customized systems and components using the above-mentioned results

Towards Automation

- Automatic manufacturing of software products can be achieved in different ways:
  - object-orientation and polymorphism
  - frameworks
  - reflection
  - code generation

Code Generation: Where, When and What



Generic Programming

- Generic Programming is a subdiscipline of computer science that deals with finding abstract representations of efficient algorithms, data structures and other software concepts
- Algorithms are expressed with minimal assumptions about data abstraction and vice versa

Generic Parameters

- Function for squaring a number:

```
sqr (x) { return x * x; }
```

- C version:

```
int sqr(int x) { return x * x; }
```

- Multiple versions:

```
int sqrInt(int x) { return x * x; }
```

- C++ overloading:

```
int sqr(int x) { return x * x; }
double sqrt(double x) { return x * x; }
```

C++ Templates

```
template<class T>
T sqr(T x) { return x * x; }
```

- Compiler automatically generates a version for each parameter type used by a program:

```
int a = 3;
double b = 3.14;
int aa = sqr(a);
double bb = sqr(b);
```

Compile-time Checking

```cpp
template<class T>
void swap(T& a, T& b) {
        const T temp = a;
        a = b;
        b = temp;
}

// …
int a = 5, b = 9;
swap(a, b);   // OK
double c = 9.0;
swap(a, c);   // error
```

C++ Standard Template Library

- Goal: represent algorithms in as general form as possible without compromising efficiency
- Extensive use of templates
- Only uses static binding (an inlining)
- Use of iterators for decoupling algorithms from containers
- Iterator: abstraction of pointers

STL Organization

- Containers
  - vector, deque, list, set, map, ...
- Algorithms
  - for_each, find, transform, sort
- Iterators
  - forward_iterator, reverse_iterator, istream_iterator, ...
- Function Objects
  - plus, equal, logical_and, project1

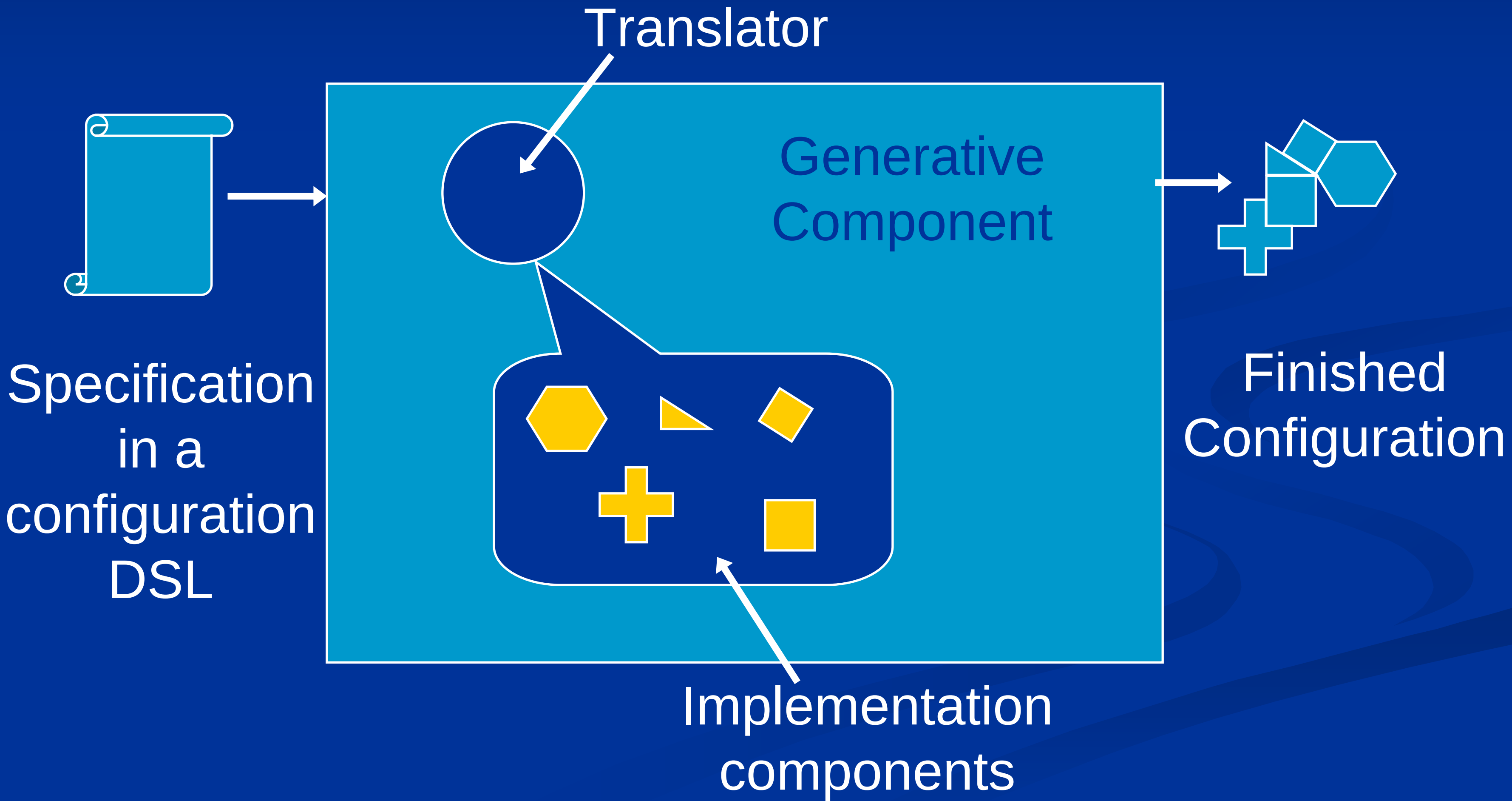
BubbleSort Definition

```
template<class T>
void bubblesort(T a[], unsigned size, bool (*comp)(T&,
T&)) {
   for (unsigned i = 0; i < size; ++i)
      for (unsigned j = i+1; j < size; ++j)
             if (comp(a[i], a[j])
                   swap(a[i], a[j]);
};

int x[] = {-1, -2, -3, -4, -5};
bubblesort(x, sizeof(x)/sizeof(x[0]), greater<int>());
bubblesort(x, sizeof(x)/sizeof(x[0]), less<int>());
```

Generic vs Generative

- Generic Programming focuses on representing families of domain concepts
  - Problem of manual assembly in STL
    - extensive knowledge of implementation detail needed
    - implementation components, illegal & optimal configurations
    - poor error reporting
- Generative Programming also includes the process of creating concrete instances of concepts

Something more is required


Translator
Generative Component
Specification in a configuration DSL
Finished Configuration
Implementation components

Why Generators ?

- Raise the intentionality of system descriptions
  - E.g. using domain specific notation
- Produce an efficient implementation
  - Nontrivial mapping into implementation concepts
- Avoid the library scaling problem
  - Library built as concrete component double in size for each new added feature

Template Metaprogramming in C++

- Metaprogramming is about representing and manipulating components that implement the basic functionality of a system
- C++ template mechanism allows us to write code which is executed by C++ compiler
- The static code can be used to manipulate the dynamic code

An example: Factorial

```
int factorial(int n) {
    if(n == 0)
         return 1;
    else
        return n*factorial(n-1);
}
```

Recursive factorial function

```
template<int n>
struct Factorial {
  enum { RET = Factorial<n-1>::RET * n };
}

template<>
struct Factorial<0> {
  enum { RET = 1 };
}

// …
cout << Factorial<7>::RET << endl;
```

Static code for computing the factorial at compile time

A Turing-complete Language

- Static C++ is Turing-complete because it is equipped with a conditional and a loop construct and consequently it can be used to implement a Turing machine

I always knew C++ templates were the work of the Devil and now I'm sure :-)

- Cliff Click, 1998

Implementation of IF

```
template<bool condition, class Then, class Else>
struct IF
{ typedef Then RET;
};

//specialization for condition==false
template<class Then, class Else>
struct IF<false, Then, Else>
{ typedef Else RET;
};

//…
IF<(1+2>4), short, int>::RET i;
```

but also FOR, WHILE, DO-WHILE and SWITCH can be implemented

Code Generation

```
int power(const int& m, int n)
{ int r = 1;
  for (; n>0; --n)
     r *= m;
  return r;
}
```

Function for raising m to the power of n

```
template<int n>
int power(const int& m)
{ return power<n-1>(m) * m; }

template<>
int power<1>(const int& m)
{ return m; }

template<>
int power<0>(const int& m)
{ return 1; }

//...
cout << power<3>(2) << endl;
```

Raising a number to the power of n where n is known at compile time

C++ preprocessor

- C++ preprocessor allows the programmer to do some computation too but...
  - it does not support recursion or looping
  - it cannot use data embedded in C++ program

Runtime Code Generation

- How to evaluate a polynomial quickly
  - Determine Y for a given value of X, where

$$Y = A_0 + A_1 * X + A_2 * X^2 + \ldots + A_n * X^n$$

Simple (loop) Implementation

- Approach:
  - Refactor the polynomial to:

    $Y = A_0 + X * (A_1 + X * (A_2 \dots))$
  - Loop:

```
val = val * x;
result = result + val * coefficient[I];
```

Custom Code Implementation

- Goal
  - Get rid of looping overhead
- Approach:
  - Write a C# class with a function that evaluates the polynomial directly:

```
y = 1.5 + x * (3.5 + x * (133.2 + x * (3288)));
```

  - Write the class to a file, compile it, and then call the evaluation function through reflection or interface (faster)

Generating code in .NET

- CodeDom
  - A set of classes that describe code
  - Can generate different languages
    - Generators for VB and C#
  - Fairly difficult to use
    - Like generating a parse tree

Generating code in .NET (2)

- CodeDom Example:
  Consider generating the following code
  `public class Veichle : Object { }`

```
CodeNamespace n = …

CodeTypeDeclaration c = new CodeTypeDeclaration(“Veichle”);

c.IsClass = true;
c.BaseTypes.Add(typeof (System.Object));
c.TypeAttributes = TypeAttributes.Public;
n.Types.Add(c);

// Generating the code
ICodeGenerator cg = … // code generator for intend language

cg.GeneratedCodeFromNamespace(n,textWriter,odeGeneratorOptions);
```

Generating code in .NET (3)

- Reflection.Emit
  - Goal:
    - Get rid of overhead of compiling the C# file.
  - More difficult to debug and IL knowledge is required
  - Approach:
    1. Create a class that implements the IPolynomial interface
    2. Write a function in IL that performs the evaluation
    3. Call through an interface as before

Generating code in .NET (4)

- Reflection.Emit Example:
  Consider generating the following code
  `public class Veichle : Object { }`

```
AssemblyName an = new AssemblyName();
an.Name = “MyOwnAssembly”;

AssemblyBuilder abuilder =
Thread.GetDomain().DefineDynamicAssembly(an,
AssemblyBuilderAccess.Save);

ModuleBuilder module =
abuilder.DefineDynamicModule(“Example,”Example.DLL”);
TypeBuilder myClass = module.DefineType(“ExampleClass”,
TypeAttributes.Public);
```

Generating code in .NET (5)

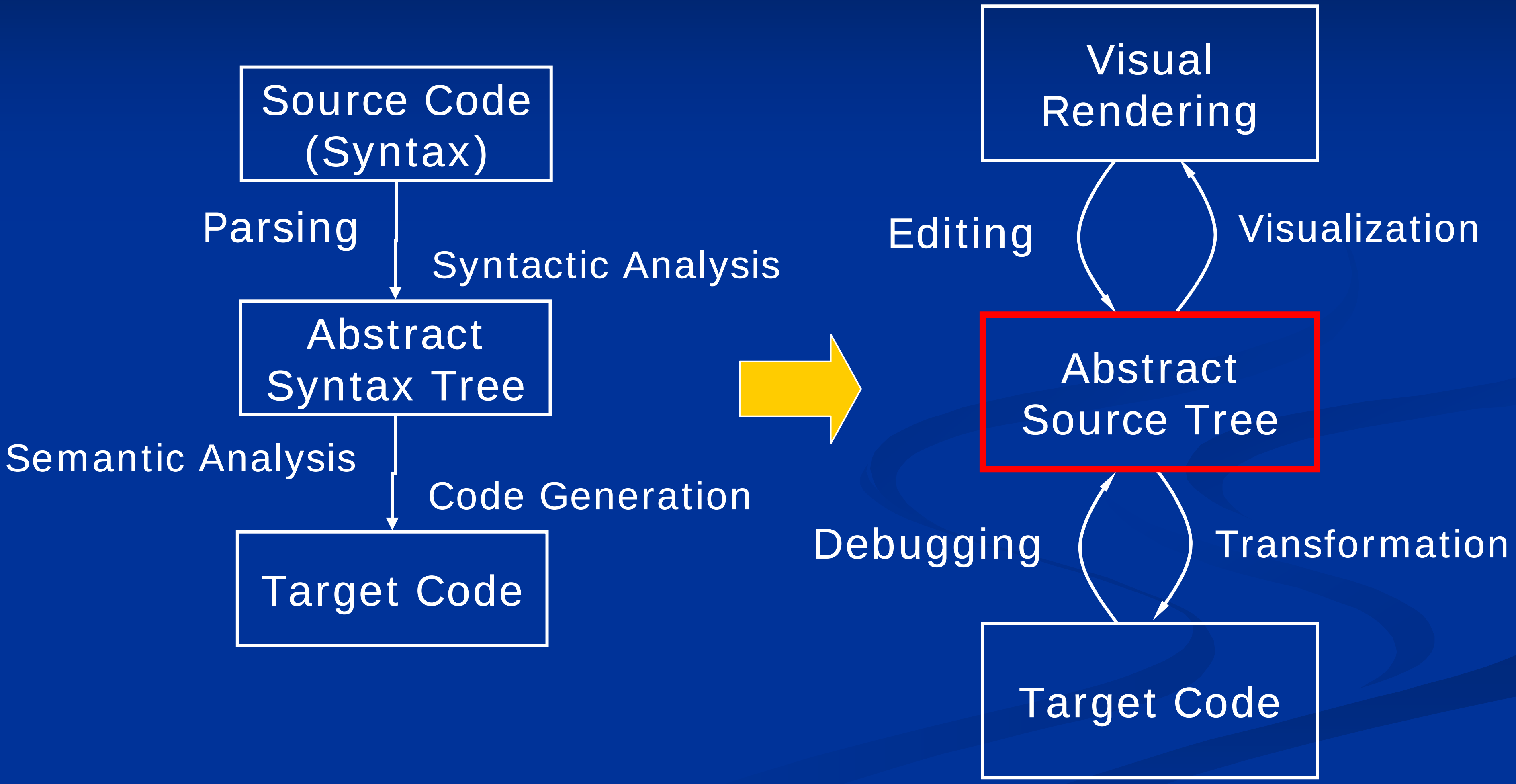
- Reflection.Emit Example (continued):
  Adding a method

```
Type[] params = new Type[2];
params[0] = typeof(int);

params[1] = typeof(int);

Type returnType = typeof(int);


// Creating the method

MethodBuilder addMethod = myClass.DefineMethod(“Add”,
MethodAttributes.Public | MethodAttributes.Virtual, returnType,
params);

// Implementing the method

ILGenerator ilg = addMethod.GetILGenerator();

ilg.Emit(OpCodes.Ldc_T4, 0);
```

Active Libraries

- The idea of putting compile-time metacode into domain-specific libraries
- Active libraries – in addition to classes and functions – also contain metacode for configuration, generation, optimization, error reporting, debugging and profiling, editing and visualization of code, code refactoring, versioning, …
- Extensible programming environments
  - e.g. Intentional Programming (Microsoft Research)

Intentional Programming



A WYSIWYG Programming ?

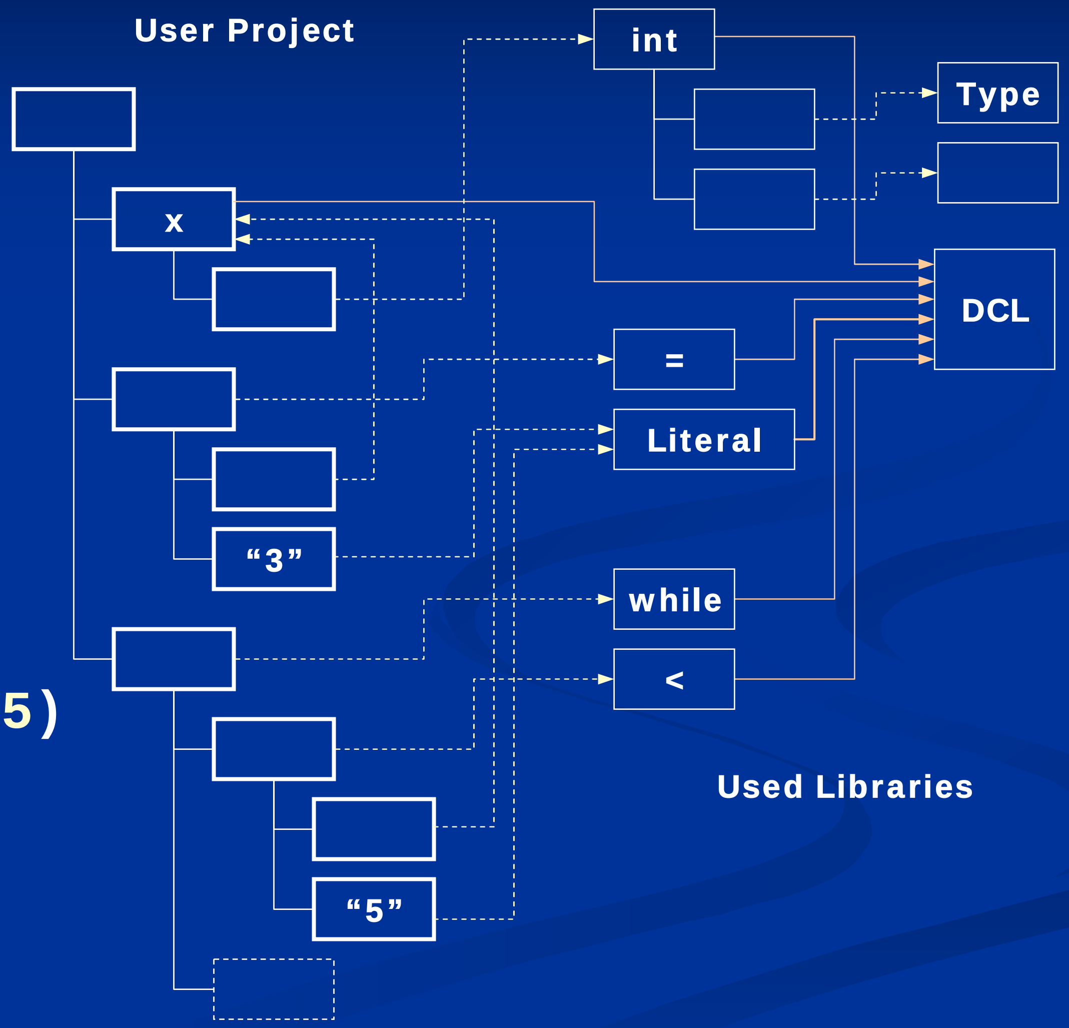
- The source code captures the intentions of the programmer, but does so at a very low level.
- The idea of intents is that a representation of the 'intent' of the programmer should be the best way to store the code, and these 'intents' should also be 'viewable' in the language of your choice.

The Code is the Data

- All code is stored in a database and the user will only see a text-like (or graphical layout) rendering of the information stored in that database.
- Physically IP systems consist of a database of symbols and their definitions, and the manipulations on them.
- The system then generates source code from this database, which is then compiled as normal. The system can generate any sort of source the user desires as long as they have an appropriate translator
- In this respect IP systems are a type of code generator

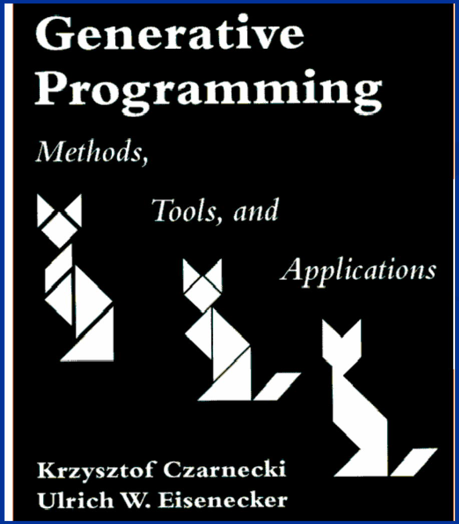
Example Tree

```
{
  int x;

  x = 3;

  while (x < 5)
  {
  }

}
```



The Book

- *"Generative Programming: Methods, Tools, and Applications"*
  Krzysztof Czarnecki, Ulrich Eisenecker



http://www.generative-programming.org

Internet Resources

- Generative Programming Wiky
  http://www.program-transformation.org/Transform/GenerativeProgrammingWiki
- Generative and Component-based Software Engineering
  http://www.prakinf.tu-lmenau.de/~czarn/generate/engl.html
- Intentional Programming Wiky
  http://c2.com/cgi/wiki?IntentionalProgramming

References

- International Conference on Generative Programming and Component Engineering (GPCE) (next will be located at Vancouver CA, 24-28 October 2004)
- K. Czarnecki and U. Eisenecker. "*Components and Generative Programming*." Invited talk, (ESEC/FSE'99, Toulouse, France, September 1999)
- C. Simonyi, "*The Death of Computer Languages, the Birth of Intentional Programming*," The Future of Software, Univ. of Newcastle upon Tyne, England, Dept. of Computing Science, 1995
- Markus Voelter, "*A Catalog of Patterns for Program Generation", Tech. Report, Heidenheim, Germany 2003