

# Vishnu Genesis

## [Attack of the clones]

Jean-Sébastien Mouret  
js@lrde.epita.fr

Séminaire du LRDE, 30 Octobre 2002



# Table des matières

<b>Introduction</b> .....	<b>3</b>
<b>Visual Language Dream</b> .....	<b>4</b>
<b>Considerations on writing components</b> .....	<b>10</b>
<b>OpenC++ Overview</b> .....	<b>14</b>
<b>Extensions for writing components</b> .....	<b>18</b>
<b>Adding an event system</b> .....	<b>22</b>
<b>Extensions for data-flow support</b> .....	<b>29</b>
<b>A working example</b> .....	<b>34</b>
<b>Further work</b> .....	<b>39</b>

**Table des matières**

---

**Conclusion** ..... 40

**Questions ?** ..... 41

# Introduction

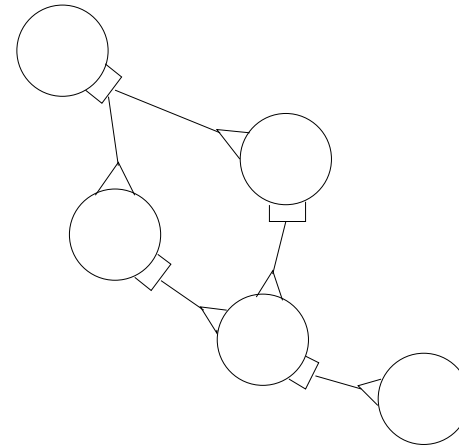
# Visual Language Dream

Previously on Lrde's seminar :

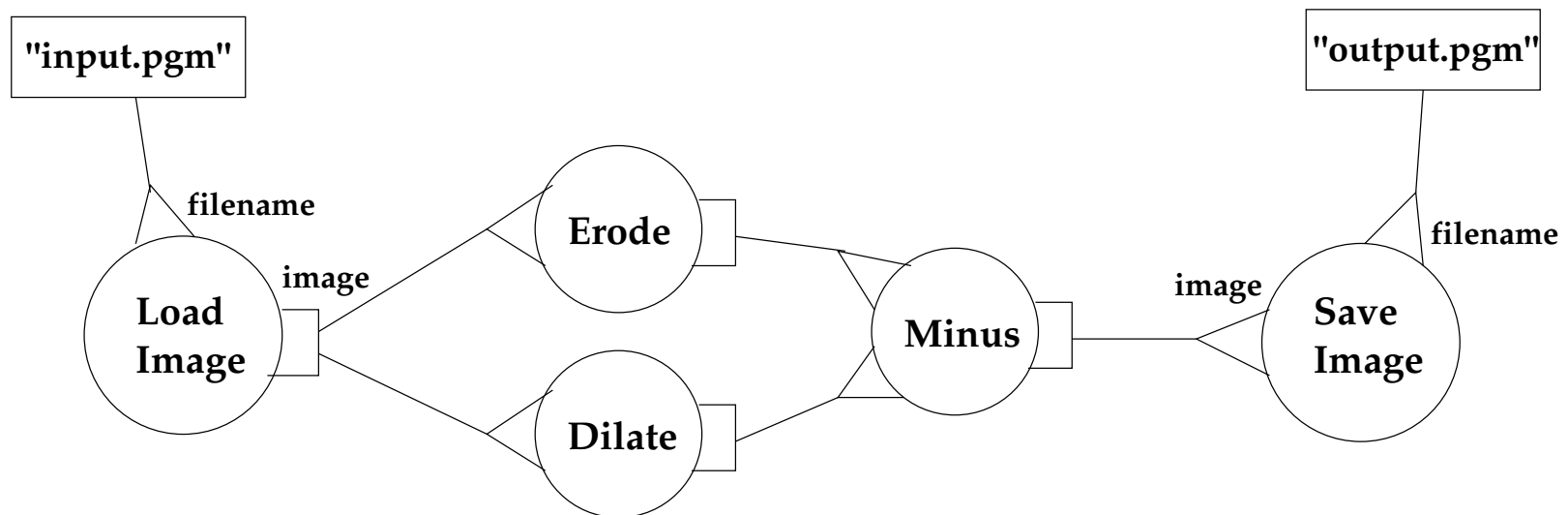
- imagine a world with reusable components
- that can be easily linked together
- that can be automatically optimized
- with an intuitive interface

# Graph Grammar

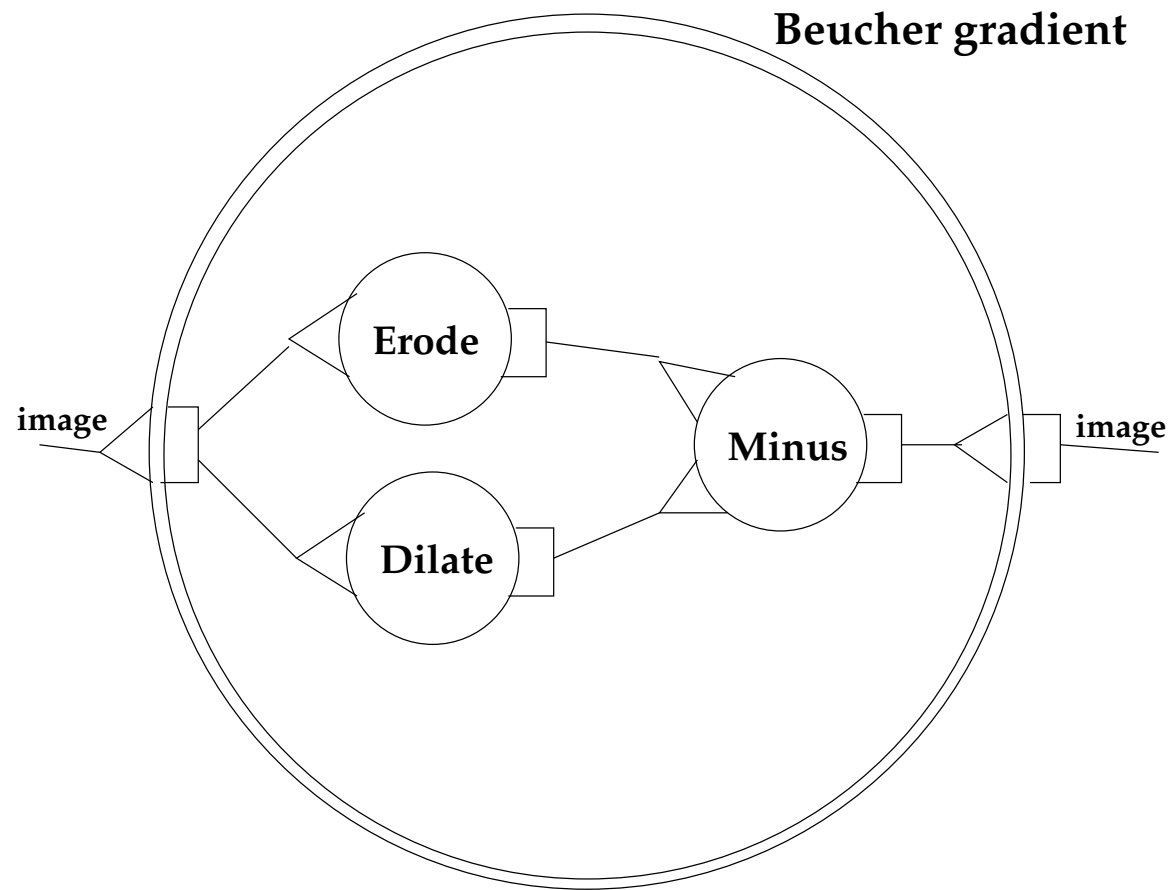
- Intuitive : Dataflow
- Nervous : Event-driven
- suitable to any data processing
- natural for Gui WidgetSet



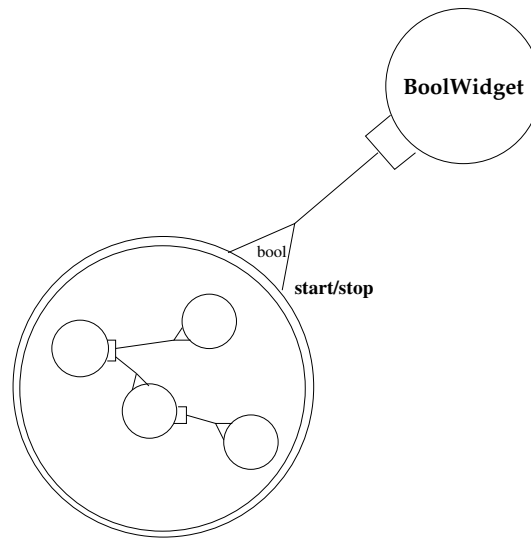
# Simple image processing algorithm



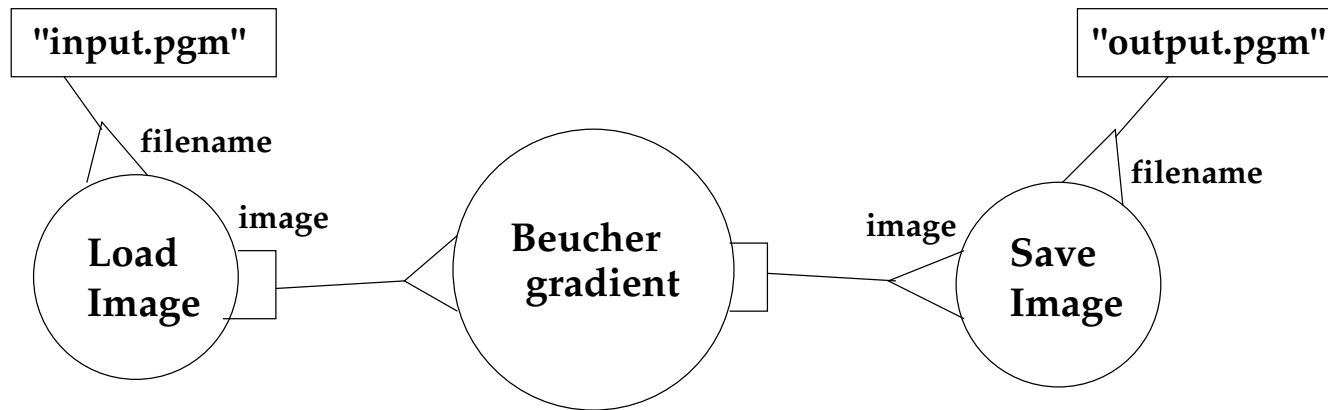
# Encapsulation



# Execution Control



# Optimisation / Cristallisation



## Considerations on writing components

Goals :

- write reusable components
- with dynamic loading
- and efficient cristallisation

## Classic interface description

```
struct Interface
```

```
{  
  virtual void foo () = 0;  
  virtual int bar () = 0;  
};
```

```
struct Implementation : public Interface
```

```
{  
  void foo()  
  {  
    // ...  
  }  
};
```

```
void foo(Interface * impl)
```

```
{  
  for ( int i = 0; i < 10000; ++i)  
    impl->foo();  
};
```

- + can be loaded at runtime
- suffers virtual indirection cost

## Static interface

```
template<typename Bottom>
struct Interface_
{
    void foo () { self (). foo (); }
    int bar () { return self (). bar (); }

protected:
    Bottom& self () { return static_cast<Bottom&>(*this); }
};

struct Implementation : Interface_<Implementation>
{
    void foo()
    {
        // ...
    }
};
```

```
template<typename Bottom>
void foo(Interface_<Bottom>* impl)
{
    for (int i = 0; i < 10000; ++i)
        impl->foo();
};
```

- + no interface penalty, maximum inlining
- no separate compilation, dynamic loading...

## What do we want ?

- ▷ write virtual and static interfaces in one shot to maintain one code

```
abstract struct Interface
{
    void foo ();
    int bar ();
};
```

```
struct Interface
{
    virtual void foo () = 0;
    virtual int bar () = 0;
};

template<typename Bottom>
struct Interface_
{
    void foo () { self (). foo (); }
    int bar () { return self (). bar (); }
protected:
    Bottom& self() { return static_cast<Bottom&>(*this); }
};
```

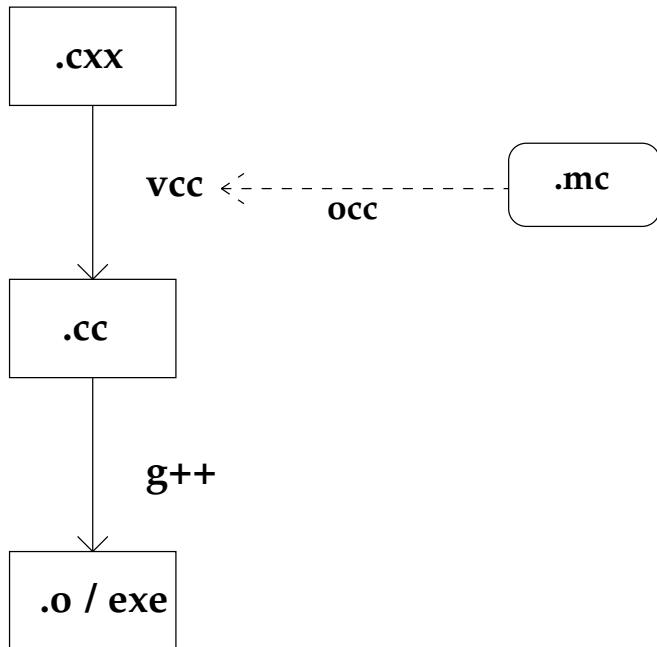
⇒ we need C++ extensions ⇒ we need a C++ parser

# OpenC++ Overview

<http://www.csg.is.titech.ac.jp/~chiba/openc++.html>

# What is this ?

"A tool of source-code translation for C++"



.cxx : extended c++  
.cc : regular c++  
.mc : openc++ source

occ : openc++ compiler  
vcc : vishnu compiler  
g++ : (put your favorite c++ compiler here)

## User keywords places

```
class-modifier struct Foo
{
  access-specifier:
    member-modifier void method(argument-modifier int param)
    {
      Foo* f = new-modifier new Foo();
      f->while-statement (true)
        { // ... }
      f->for-statement (it = begin; it != end; ++it)
        { // ... }
      f->closure-statement(int x, int y)
        { // ... }
    }
};
```

## How to write transformations ?

```
#include <mop.h>

class Abstract : public Class
{
    static bool Initialize ()
    {
        RegisterMetaclass("abstract","Abstract");
    }

    void TranslateClass(Environment* env)
    {
        Class* c = new Class(env, Ptree::qMake("Name()'_ '"));

        Member m;
        int i = -1;
        while (NthMember(++i,m))
        {
            if (m.IsFunction())
            {
                m.SetFunctionBody(Ptree::qMake("{ return self().'m.Name()'; }" ));
                c->AppendMember(m);
                m.SetFunctionBody(Ptree::qMake(" = 0;"));
                ChangeMember(m);
            }
        }
    }

    c->AppendMember(Ptree::Make(
        "protected:\n"
        " Bottom& self () { return static_cast<Bottom&>(*this); }\n" ));

    AppendAfterToplevel(env, Ptree::Make("template<typename Bottom>\n"));
    AppendAfterToplevel(env, c);
};
```

# Extensions for writing components

## What is a component ?

A component is a class that :

- exports several interfaces
- may implement its own behavior for each interface

```
struct Component
{
  struct Implementation1 : public Interface1
  {
    // override methods
  } a;
  struct Implementation2 : public Interface2
  {
    // override methods
  } b;
};
```

## Problem

not easy to get a pointer on the owning component in an implementation.

```
struct Component
{
  struct Implementation1 : public Interface1
  {
    void meth()
    {
      owner().component_data = 51;
    }
  } a;
  int component_data;52
};
```

- in Implementation1, Component is not yet a complete type
- must add a new constructor or something to set a 'owner' pointer

## Solution : the `refine` keyword

```
struct Component
{
  refine struct Interface1
  {
    void meth()
    {
      owner().component_data = 51;
    }
  } a;

  int component_data;
};
```

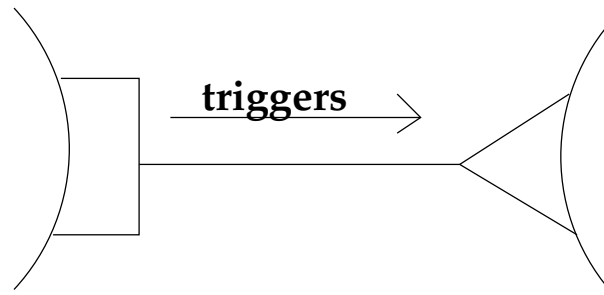
How does the generated source code look like? You don't want to know...

# Adding an event system

## The two headed abstract

Goals :

- Define a kind of protocol :
  - that can still be transformed to virtual or static interface
  - permits easy declaration of **events**



## New access specifier : `trigger`

```
abstract struct Int
{
  trigger :
  void changed(int i);

  public:
  int get ();
};
```

```
struct Input_Int
{
  virtual void on_changed(int i) {}
};

struct Output_Int
{
  void send_changed(int i)
  {
    for_all (observers o) o.on_changed(i);
  }

  virtual int get () = 0;
};
```

## Another example

```
abstract struct Mouse
{
  trigger :
  void leftClick ();
  void rightClick ();
  void doubleClick();
public:
  bool isPressed(int button);
};
```

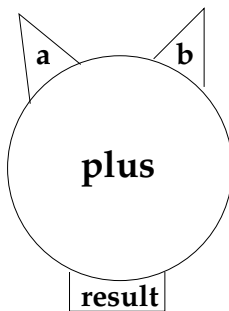
```
struct Input_Mouse
{
  virtual void on_leftClick () {}
  virtual void on_rightClick () {}
  virtual void on_doubleClick() {}
};

struct Output_Mouse
{
  void send_leftClick () { // ... }
  void send_rightClick () { // ... }
  void send_doubleClick() { // ... }

  virtual bool isPressed(int button) = 0;
};
```

## Our custom component : node

- New access specifiers : `input` and `output`
- Naive component description



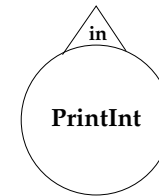
```
node struct Plus
{
  input:
    Int a, b;
  output:
    Int result;
};
```

```
struct Plus
{
  Input_Int a, b;

  Output_Int result;
};
```

## Responding to triggers

- by using the `refine` construct



```
node struct PrintInt
{
input:
  refine struct Int
  {
    void on_changed(int v)
    {
      cout << "value changed : "
        << v << endl;
    }
  } in;
};
```

```
struct PrintInt
{
  struct Int_Refined : public Input_Int
  {
    void on_changed(int v)
    {
      cout << "value changed : "
        << v << endl;
    }
  } in;
};
```

## Another example

```
node struct MouseController
{
output:
    Mouse mouse;

void foo()
{
    mouse.send_leftClick();
}
};
```

```
node struct Widget
{
input:
    refine struct Mouse
    {
        void on_leftClick ()
        {
            cout « "youpi!" « endl;
        }
    } mouse_in;
};
```

# Extensions for data-flow support

## Defining a value

Lightweight abstract :

```
abstract struct Value
{
  trigger :
  void changed(const int&);

  public:
  inline const int& operator()() const { return get (); }
  const int& get() const;
};
```

## A constant node

```
node struct ConstValue
{
  ConstValue(const int& v) : _value(v) {}

  output:
  refine struct Value
  {
    const int& get() const { return owner()._value; }
  } o;

  public:
  inline const int& operator>()() const { return o.get (); }

  protected:
  int _value;
};
```

## Basic data node

```
node struct BaseValue
{
  BaseValue() {}
  BaseValue(const int& v) : _value(v) {}

  output:
  refine struct Value
  {
    const int& get() const { return owner()._value; }
  } o;

  public:
  inline const int& operator()() const { return o.get (); }
  inline void operator=(const int& i)
  {
    _value = i;
    o.send_changed(i);
  }

  protected:
  int _value;
};
```

## A sensitive input

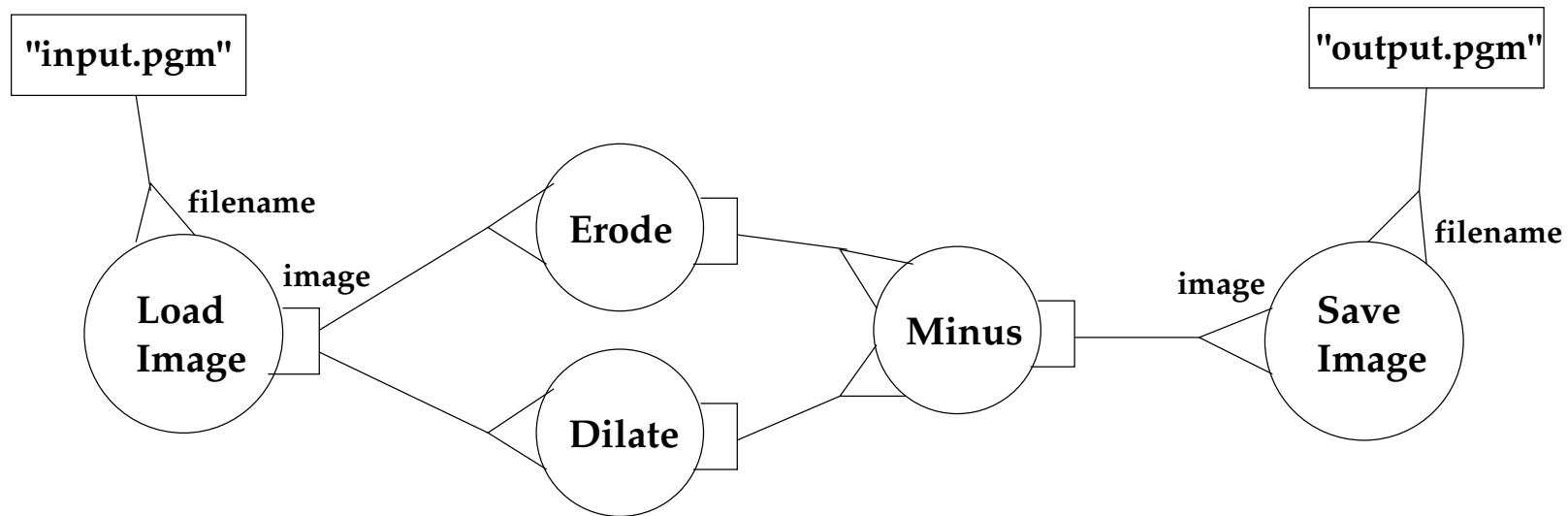
```
node struct WatchValue
{
  WatchValue() : _absent(false) {}

  input:
    refine struct Value
    {
      void on_changed(const int& v)
      {
        owner().absent(false);
      }
    } i;

  public:
    inline const int& operator>()() const { return i . link (). get (); }
    inline bool ready () { return i . ready (); }
    inline bool present() const { return !_absent; }
    inline bool absent() const { return _absent; }
    inline bool absent(bool b) { _absent = b; return _absent; }
    inline void done () { absent(true); }

  protected:
    bool _absent;
};
```

# A working example



## First Part

```
#include "vishnu.hxx"

include("iostream")
include(" string ")
using namespace std;

include("oln/basics2d.hh")
include("oln/morpho/dilation.hh")
include("oln/morpho/erosion.hh")
include("oln/arith/ops.hh")
include("copy.hh")
using namespace oln;

MakeValue(String,string)
MakeValue(Image2d,image2d<int_u8>)
```

- Two include levels
- MakeValue macro

## I/O Part

```
node struct LoadImage
{
  input:
    WatchString filename;
  output:
    BasImage2d out;

  void compute()
  {
    if (filename.present())
    {
      out = io :: load(filename ());
      filename.done();
    }
  }
};
```

```
node struct SaveImage
{
  input:
    WatchString filename;
    WatchImage2d img;

  void compute()
  {
    if (img.present() && filename.ready())
    {
      io :: save(img(), filename ());
      img.done();
    }
  }
};
```

## Treatment Part

```
node struct Erode
{
  input:
    refine struct Image2d
    {
      void on_changed(const image2d<int_u8>& i)
      {
        owner().out = morpho::erosion(i, win_c8p());
      }
    } img;

  output:
    BasImage2d out;
};

node struct Dilate
{
  // ...
};
```

```
node struct Minus
{
  input:
    WatchImage2d a, b;
  output:
    BasImage2d out;

  void compute()
  {
    if ((a.present () || b.present()) && a.ready() && b.ready())
    {
      out = arith :: minus(convert::stretch<int_u8>(),a (), b ());
      a.done(); b.done();
    }
  }
};
```

## And the main

```
int main()
{
  // loading
  ConstString filename("lena.pgm");
  LoadImage loader;
  loader.filename.i.connect(&filename.o);

  // treatment
  Dilate dilate;
  Erode erode;
  Minus minus;
  dilate .img.connect(&loader.out.o);
  erode.img.connect(&loader.out.o);
  minus.a.i.connect(&dilate.out.o);
  minus.b.i.connect(&erode.out.o);

  // saving
```

```
  ConstString filename_out("lena_out.pgm");
  SaveImage saver;
  saver.filename.i.connect(&filename_out.o);
  saver.img.i.connect(&minus.out.o);

  // run
  vishnu::run();
}
```



## Further work

- improve generated code features
  - introspection
  - static interface
  - template support
- replace openc++ by bob's stratego suite
- standard node library
- visual editor !

## Conclusion

- Low-level features are now specified.
  - A rough implementation is working.
- ⇒ Future looks great !

Questions ?

---

# Questions ?