Presentation of TC-3

Assistants 2009

May 6, 2014

Presentation of TC-3

Overview of the tarball

The tree structure of TC-3

- Only ‘`src/bind`’ directory has been added.
- That is where the Binder lays.
- Implement it entirely.
- Pay attention to ’`break`’: it must be used in a loop only. If not, exit with a bind error code (4).

```
echo break | _build/src/tc -B -A --prelude= -
=> standard input:1.0-4: ‘break’ outside any loop
echo \$?
=> 4
```

The tree structure of TC-3

- Only ‘`src/bind`’ directory has been added.
- That is where the Binder lays.
- Implement it entirely.
- Pay attention to ’`break`’: it must be used in a loop only. If not, exit with a bind error code (4).

```
echo break | _build/src/tc -B -A --prelude= -
=> standard input:1.0-4: ‘break’ outside any loop
echo \$?
=> 4
```

The tree structure of TC-3

- Only ‘`src/bind`’ directory has been added.
- That is where the Binder lays.
- Implement it entirely.
- Pay attention to ’`break`’: it must be used in a loop only. If not, exit with a bind error code (4).

```
echo break | _build/src/tc -B -A --prelude= -
=> standard input:1.0-4: ‘break’ outside any loop
echo \$?
=> 4
```

The tree structure of TC-3

- Only ‘`src/bind`’ directory has been added.
- That is where the Binder lays.
- Implement it entirely.
- Pay attention to ’`break`’: it must be used in a loop only. If not, exit with a bind error code (4).

```
echo break | _build/src/tc -B -A --prelude= -
=> standard input:1.0-4: ‘break’ outside any loop
echo \$?
=> 4
```

misc::ScopedMap

- Implement it entirely.
- There is more than one way to do it:
  - Functionnal approach
  - Imperative approach

misc::ScopedMap

- Implement it entirely.
- There is more than one way to do it:
  - Functionnal approach
  - Imperative approach

misc::ScopedMap

- Implement it entirely.
- There is more than one way to do it:
  - Functionnal approach
  - Imperative approach

misc::ScopedMap

- Implement it entirely.
- There is more than one way to do it:
  - Functionnal approach
  - Imperative approach

Simple stack

- A simple stack with markers for scope delimiters.
- Advantages:
  - Easy to implement.
- Drawbacks:
  - Slow: You have to walk through the stack each time you want to check if an element exists. If it does not exist you have to walk the entire stack.

Simple stack

- A simple stack with markers for scope delimiters.
- Advantages:
  - Easy to implement.
- Drawbacks:
  - Slow: You have to walk through the stack each time you want to check if an element exists. If it does not exist you have to walk the entire stack.

Simple stack

- A simple stack with markers for scope delimiters.
- Advantages:
  - Easy to implement.
- Drawbacks:
  - Slow: You have to walk through the stack each time you want to check if an element exists. If it does not exist you have to walk the entire stack.

Simple stack

- A simple stack with markers for scope delimiters.
- Advantages:
  - Easy to implement.
- Drawbacks:
  - Slow: You have to walk through the stack each time you want to check if an element exists. If it does not exist you have to walk the entire stack.

Simple stack

- A simple stack with markers for scope delimiters.
- Advantages:
  - Easy to implement.
- Drawbacks:
  - Slow: You have to walk through the stack each time you want to check if an element exists. If it does not exist you have to walk the entire stack.

Stack of maps

- Each level in the stack represents a scope.
- Within each level, a symbol can be found immediately since it is a hash map.
- Advantages:
  - Faster than the previous solution.
- Drawbacks:
  - Harder to implement.
  - You still have to walk through the whole stack if a symbol does not exist.

Stack of maps

- Each level in the stack represents a scope.
- Within each level, a symbol can be found immediately since it is a hash map.
- Advantages:
  - Faster than the previous solution
- Drawbacks:
  - Harder to implement
  - You still have to walk through the whole stack if a symbol does not exist

Stack of maps

- Each level in the stack represents a scope.
- Within each level, a symbol can be found immediately since it is a hash map.
- Advantages:
  - Faster than the previous solution
- Drawbacks:
  - Harder to implement
  - You still have to walk through the whole stack if a symbol does not exist

Stack of maps

- Each level in the stack represents a scope.
- Within each level, a symbol can be found immediately since it is a hash map.
- Advantages:
  - Faster than the previous solution
- Drawbacks:
  - Harder to implement.
  - You still have to walk through the whole stack if a symbol does not exist.

Stack of maps

- Each level in the stack represents a scope.
- Within each level, a symbol can be found immediately since it is a hash map.
- Advantages:
  - Faster than the previous solution
- Drawbacks:
  - Harder to implement.
  - You still have to walk through the whole stack if a symbol does not exist.

Stack of maps

- Each level in the stack represents a scope.
- Within each level, a symbol can be found immediately since it is a hash map.
- Advantages:
  - Faster than the previous solution
- Drawbacks:
  - Harder to implement.
  - You still have to walk through the whole stack if a symbol does not exist.

Stack of maps

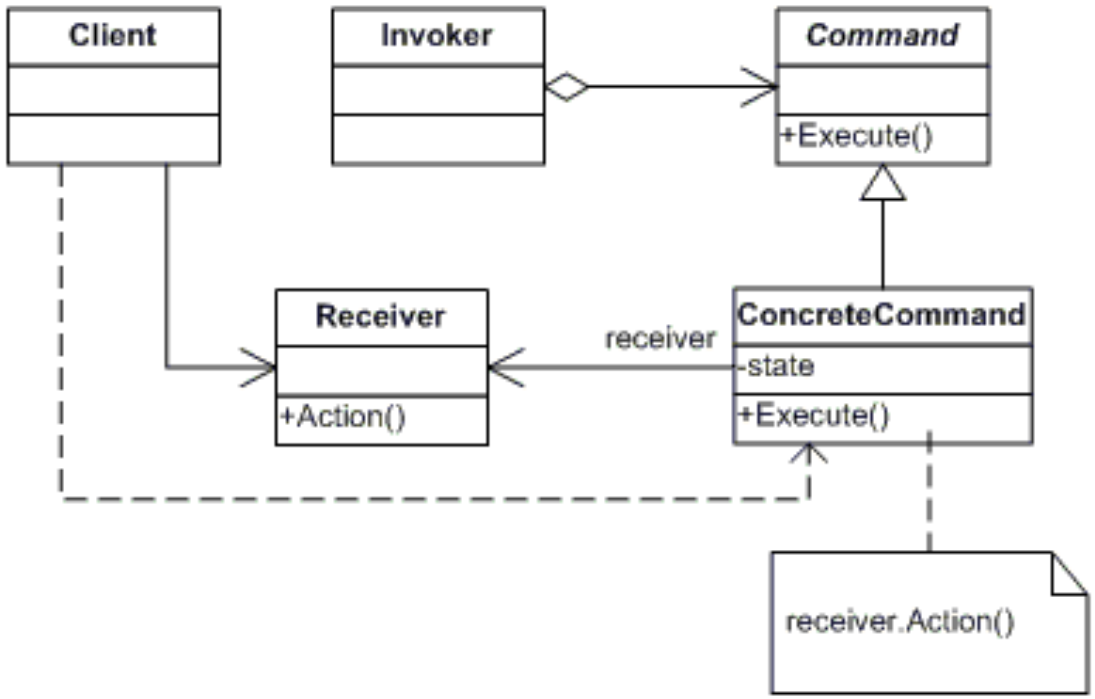
- Each level in the stack represents a scope.
- Within each level, a symbol can be found immediately since it is a hash map.
- Advantages:
  - Faster than the previous solution
- Drawbacks:
  - Harder to implement.
  - You still have to walk through the whole stack if a symbol does not exist.

Task module

The tasks module

The design pattern *command*:

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



The tasks module

- Clients are defined in the tasks module and instantiated in each '`<module>/tasks.*`'.
- `enable_action` will be called from argp (command-line parser): it is the invoker.
- `register_action` will play the role of the client.

The tasks module

- Clients are defined in the tasks module and instantiated in each '`<module>/tasks.*`'.
- `enable_action` will be called from argp (command-line parser): it is the invoker.
- `register_action` will play the role of the client.

The tasks module

- Clients are defined in the tasks module and instantiated in each '`<module>/tasks.*`'.
- `enable_action` will be called from argp (command-line parser): it is the invoker.
- `register_action` will play the role of the client.

Method pointer

Method pointer

- C and C++ offer function pointer.
- C++ introduces a new sort of function: function members.
- Pointers to those function members are called *function member pointers*.

Method pointer

- C and C++ offer function pointer.
- C++ introduces a new sort of function: function members.
- Pointers to those function members are called *function member pointers*.

Method pointer

- C and C++ offer function pointer.
- C++ introduces a new sort of function: function members.
- Pointers to those function members are called *function member pointers*.

Syntax of function member pointers: declaration

```
/// Member manipulator signature.
typedef void (Error::*member_manip_type) ();

/// Hook for member manipulators.
Error& operator<< (member_manip_type f);
```

Syntax of method pointer: usage

```
// Calling a function member using a function member pointer.
inline Error&
Error::operator<< (member_manip_type f)
{
  (this->*f) ();
  return *this;
}

// Passing a function member pointer as argument
// (excerpt from scantiger.ll).
error_ << misc::Error::failure
 << program_name
 << ": cannot open `" << filename_ << "': "
 << strerror (errno) << std::endl
 << &misc::Error::exit;
```

Templates

Traits

Template parameterized by templates

- Templates can be parameterized by a scalar value, a type, ...
- ... and another template :)!
- Example [1]:

```
template <class T, template <class> class C>
class Xrefd
{
  C<T>  mems;
  C<T*> refs;
};

// This instanciates a vector of "Entries"
// and  a vector of "Entries*"
Xrefd<Entry, std::vector> x1;
```

Template parameterized by templates

- Templates can be parameterized by a scalar value, a type, ...
- ... and another template :)!
- Example [1]:

```
template <class T, template <class> class C>
class Xrefd
{
  C<T>  mems;
  C<T*> refs;
};

// This instanciates a vector of "Entries"
// and  a vector of "Entries*"
Xrefd<Entry, std::vector> x1;
```

Template parameterized by templates

- Templates can be parameterized by a scalar value, a type, ...
- ... and another template :)!
- Example [1]:

```
template <class T, template <class> class C>
class Xrefd
{
  C<T>  mems;
  C<T*> refs;
};

// This instanciates a vector of "Entries"
// and  a vector of "Entries*"
Xrefd<Entry, std::vector> x1;
```

Traits

- Traits are classes that encapsulate properties of types.
- Example: is a type a pointer type?
- First define default value (Excerpt from 'lib/misc/traits.hxx').

```
/// Use is_pointer<T>::value.
template<typename T>
struct is_pointer
{
  static const bool value = false;
};
```

- Then use template specialization for pointers:

```
template<typename T>
struct is_pointer<T*>
{
  static const bool value = true;
};
```

Traits

- Traits are classes that encapsulate properties of types.
- Example: is a type a pointer type?
- First define default value (Excerpt from 'lib/misc/traits.hxx').

```
/// Use is_pointer<T>::value.
template<typename T>
struct is_pointer
{
  static const bool value = false;
};
```

- Then use template specialization for pointers:

```
template<typename T>
struct is_pointer<T*>
{
  static const bool value = true;
};
```

Traits

- Traits are classes that encapsulate properties of types.
- Example: is a type a pointer type?
- First define default value (Excerpt from 'lib/misc/traits.hxx').

```
/// Use is_pointer<T>::value.
template<typename T>
struct is_pointer
{
  static const bool value = false;
};
```

- Then use template specialization for pointers:

```
template<typename T>
struct is_pointer<T*>
{
  static const bool value = true;
};
```

Traits

- Traits are classes that encapsulate properties of types.
- Example: is a type a pointer type?
- First define default value (Excerpt from 'lib/misc/traits.hxx').

```
/// Use is_pointer<T>::value.
template<typename T>
struct is_pointer
{
  static const bool value = false;
};
```

- Then use template specialization for pointers:

```
template<typename T>
struct is_pointer<T*>
{
  static const bool value = true;
};
```

Traits: another example

- Traits can compute a type from another type. Here is a sample of 'misc/select-const.hh'

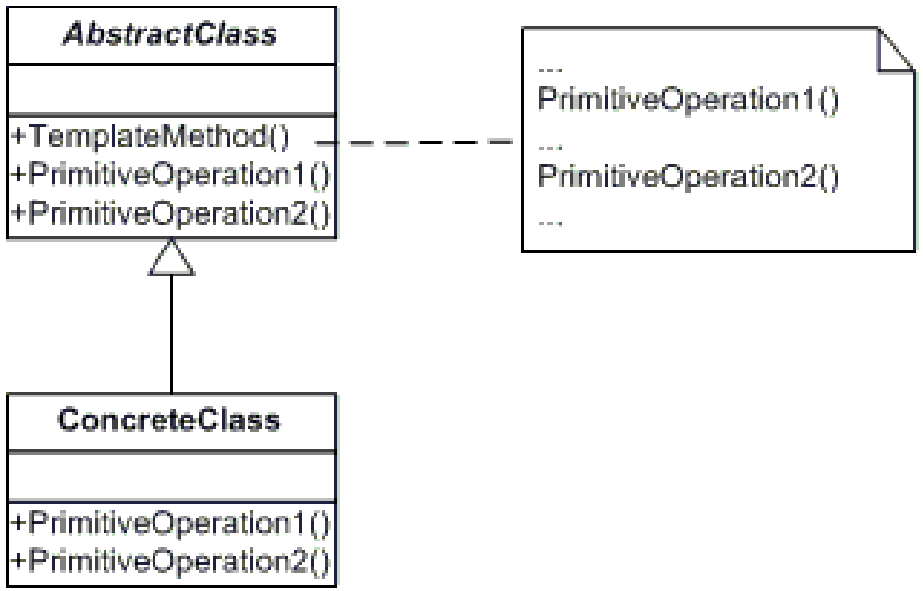
```
/// The iterator over a non const structure
/// is plain iterator.
template<typename T>
struct select_iterator
{
  typedef typename T::iterator type;
};

/// The iterator over a const structure
/// is a const_iterator.
template<typename T>
struct select_iterator<const T>
{
  typedef typename T::const_iterator type;
};
```

Template Methods

The design pattern *Template Method*

The design pattern *Template Method*:

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



The design pattern *Template Method*

- In the Tiger project it is used to visit the chunks.
- Visiting a chunk of types or functions is basically the same:
  1. Declare type or function prototypes.
  2. Process their definitions.
- warning: Implemented thanks to template, and not class inheritance as usual in C++.
- warning: Do not mix up template method and method template. (Hint: the last word is the noun and the other is the adjective).
- Likewise, "class template" is correct but "template class" is not.

The design pattern *Template Method*

- In the Tiger project it is used to visit the chunks.
- Visiting a chunk of types or functions is basically the same:
  - Declare type or function prototypes.
  - Process their definitions.
- warning: Implemented thanks to template, and not class inheritance as usual in C++.
- warning: Do not mix up template method and method template. (Hint: the last word is the noun and the other is the adjective).
- Likewise, "class template" is correct but "template class" is not.

The design pattern *Template Method*

- In the Tiger project it is used to visit the chunks.
- Visiting a chunk of types or functions is basically the same:
  - Declare type or function prototypes.
  - Process their definitions.
- warning: Implemented thanks to template, and not class inheritance as usual in C++.
- warning: Do not mix up template method and method template. (Hint: the last word is the noun and the other is the adjective).
- Likewise, "class template" is correct but "template class" is not.

The design pattern *Template Method*

- In the Tiger project it is used to visit the chunks.
- Visiting a chunk of types or functions is basically the same:
  - Declare type or function prototypes.
  - Process their definitions.
- warning: Implemented thanks to template, and not class inheritance as usual in C++.
- warning: Do not mix up template method and method template. (Hint: the last word is the noun and the other is the adjective).
- Likewise, "class template" is correct but "template class" is not.

The design pattern *Template Method*

- In the Tiger project it is used to visit the chunks.
- Visiting a chunk of types or functions is basically the same:
  - Declare type or function prototypes.
  - Process their definitions.
- warning: Implemented thanks to template, and not class inheritance as usual in C++.
- warning: Do not mix up template method and method template. (Hint: the last word is the noun and the other is the adjective).
- Likewise, "class template" is correct but "template class" is not.

The design pattern *Template Method*

- In the Tiger project it is used to visit the chunks.
- Visiting a chunk of types or functions is basically the same:
  - Declare type or function prototypes.
  - Process their definitions.
- warning: Implemented thanks to template, and not class inheritance as usual in C++.
- warning: Do not mix up template method and method template. (Hint: the last word is the noun and the other is the adjective).
- Likewise, "class template" is correct but "template class" is not.

The design pattern *Template Method*

- In the Tiger project it is used to visit the chunks.
- Visiting a chunk of types or functions is basically the same:
  - Declare type or function prototypes.
  - Process their definitions.
- warning: Implemented thanks to template, and not class inheritance as usual in C++.
- warning: Do not mix up template method and method template. (Hint: the last word is the noun and the other is the adjective).
- Likewise, "class template" is correct but "template class" is not.

Implementation

- decsvisit: the skeleton that performs the double traversal.
- visitDecHeader: visit the declaration to register in the current environment.
- visitDecBody: visits the body.

Implementation

- decsvisit: the skeleton that performs the double traversal.
- visitDecHeader: visit the declaration to register in the current environment.
- visitDecBody: visits the body.

Implementation

- decsvisit: the skeleton that performs the double traversal.
- visitDecHeader: visit the declaration to register in the current environment.
- visitDecBody: visits the body.

Interfaces

Interfaces

- Abstract classes
- Contain only pure function members.
- Used for:
  - defining a protocol of behavior used by two entities for interacting.
  - prolong implementation of the class to its client.
  - gathering several unrelated classes which expose similar behaviors.
- Examples:
  - Printable
  - Clonable

Interfaces

- Abstract classes
- Contain only pure function members.
- Used for:
  - defining a protocol of behavior used by two entities for interacting.
  - bridge implementation of the class to its client.
  - gathering several unrelated classes which expose similar behaviors.
- Examples:
  - Printable
  - Clonable

Interfaces

- Abstract classes
- Contain only pure function members.
- Used for:
  - defining a protocol of behavior used by two entities for interacting.
  - hiding implementation of the class to its client.
  - gathering several unrelated classes which expose similar behaviors.
- Examples:
  - Printable
  - Clonable

Interfaces

- Abstract classes
- Contain only pure function members.
- Used for:
  - defining a protocol of behavior used by two entities for interacting.
  - hiding implementation of the class to its client.
  - gathering several unrelated classes which expose similar behaviors.
- Examples:
  - Printable
  - Cleanable

Interfaces

- Abstract classes
- Contain only pure function members.
- Used for:
  - defining a protocol of behavior used by two entities for interacting.
  - hiding implementation of the class to its client.
  - gathering several unrelated classes which expose similar behaviors.
- Examples:
  - Printable
  - Cloneable

Interfaces

- Abstract classes
- Contain only pure function members.
- Used for:
  - defining a protocol of behavior used by two entities for interacting.
  - hiding implementation of the class to its client.
  - gathering several unrelated classes which expose similar behaviors.
- Examples:
  - Printable
  - Cloneable

Interfaces

- Abstract classes
- Contain only pure function members.
- Used for:
  - defining a protocol of behavior used by two entities for interacting.
  - hiding implementation of the class to its client.
  - gathering several unrelated classes which expose similar behaviors.
- Examples:
  - Printable
  - Cloneable

Interfaces

- Abstract classes
- Contain only pure function members.
- Used for:
  - defining a protocol of behavior used by two entities for interacting.
  - hiding implementation of the class to its client.
  - gathering several unrelated classes which expose similar behaviors.
- Examples:
  - Printable
  - Cloneable

Interfaces

- Abstract classes
- Contain only pure function members.
- Used for:
  - defining a protocol of behavior used by two entities for interacting.
  - hiding implementation of the class to its client.
  - gathering several unrelated classes which expose similar behaviors.
- Examples:
  - Printable
  - Cloneable

Interfaces

- Some languages offer interfaces (Java, C#), others let the user do the job (C++).
- In Java:

```
public interface Serializable
{
  public string serialize ();
}
```

- In C++:

```
/// Interface.
class Serializable
{
  public:
    std::string serialize () const = 0;
}
```

Interfaces

- Some languages offer interfaces (Java, C#), others let the user do the job (C++).
- In Java:

```
public interface Serializable
{
  public string serialize ();
}
```

- In C++:

```
/// Interface.
class Serializable
{
  public:
    std::string serialize () const = 0;
}
```

Interfaces

- Some languages offer interfaces (Java, C#), others let the user do the job (C++).
- In Java:

```
public interface Serializable
{
  public string serialize ();
}
```

- In C++:

```
/// Interface.
class Serializable
{
  public:
    std::string serialize () const = 0;
}
```

Multiple inheritance

- Classes can derive from several classes.
- They inherit members and function members from all classes.
- In fact, in languages that do not support multiple inheritance, developers emulate it with interfaces.
- As we wish to factor code, we do not always follow the interface concept.

Multiple inheritance

- Classes can derive from several classes.
- They inherit members and function members from all classes.
- In fact, in languages that do not support multiple inheritance, developers emulate it with interfaces.
- As we wish to factor code, we do not always follow the interface concept.

Multiple inheritance

- Classes can derive from several classes.
- They inherit members and function members from all classes.
- In fact, in languages that do not support multiple inheritance, developers emulate it with interfaces.
- As we wish to factor code, we do not always follow the interface concept.

Multiple inheritance

- Classes can derive from several classes.
- They inherit members and function members from all classes.
- In fact, in languages that do not support multiple inheritance, developers emulate it with interfaces.
- As we wish to factor code, we do not always follow the interface concept.

Bindable

- To implement bindings in the ast, attributes and accessors must be created.
- Problems:
  - Several classes must have those attributes.
  - Their closest common ancestor is Ast.
  - Some nodes that derive from Ast do not need bindings.
  - Strong type checking is required.
- As a consequence, use an interface: *Bindable*.

Bindable

- To implement bindings in the ast, attributes and accessors must be created.
- Problems:
  - Several classes must have those attributes.
  - Their closest common ancestor is Ast.
  - Some nodes that derive from Ast do not need bindings.
  - Strong type checking is required.
- As a consequence, use an interface: *Bindable*.

Bindable

- To implement bindings in the ast, attributes and accessors must be created.
- Problems:
  - Several classes must have those attributes.
  - Their closest common ancestor is Ast.
  - Some nodes that derive from Ast do not need bindings.
  - Strong type checking is required.
- As a consequence, use an interface: *Bindable*.

Bindable

- To implement bindings in the ast, attributes and accessors must be created.
- Problems:
  - Several classes must have those attributes.
  - Their closest common ancestor is Ast.
  - Some nodes that derive from Ast do not need bindings.
  - Strong type checking is required.
- As a consequence, use an interface: *Bindable*.

Bindable

- To implement bindings in the ast, attributes and accessors must be created.
- Problems:
  - Several classes must have those attributes.
  - Their closest common ancestor is Ast.
  - Some nodes that derive from Ast do not need bindings.
  - Strong type checking is required.
- As a consequence, use an interface: *Bindable*.

Bindable

- To implement bindings in the ast, attributes and accessors must be created.
- Problems:
  - Several classes must have those attributes.
  - Their closest common ancestor is Ast.
  - Some nodes that derive from Ast do not need bindings.
  - Strong type checking is required.
- As a consequence, use an interface: *Bindable*.

Bindable

- To implement bindings in the ast, attributes and accessors must be created.
- Problems:
  - Several classes must have those attributes.
  - Their closest common ancestor is Ast.
  - Some nodes that derive from Ast do not need bindings.
  - Strong type checking is required.
- As a consequence, use an interface: *Bindable*.

Bindable: one interface?

- Each node referring to a type, function or variable implements Bindable.
- Pros:
  - Easy to implement.
  - Code is completely factorized.
- Cons:
  - No strong type checking: the destinations could be anything. Do not FunctionDecl, TypeDef or VarDecl.

Bindable: one interface?

- Each node referring to a type, function or variable implements Bindable.
- Pros:
  - Easy to implement.
  - Code is completely factorized.
- Cons:
  - No strong type checking, the destructure is not an error. Does not FunctionDecl, TypeDef or VarDec

Bindable: one interface?

- Each node referring to a type, function or variable implements Bindable.
- Pros:
  - Easy to implement.
  - Code is completely factorized.
- Cons:
  - No strong type checking, the destructure is not the correct. Do not FunctionDecl, TypeDef or VarDec

Bindable: one interface?

- Each node referring to a type, function or variable implements Bindable.
- Pros:
  - Easy to implement.
  - Code is completely factorized.
- Cons:
  - No strong type checking, the definitions must be generic Dec, not FunctionDec, TypeDec or VarDec.

Bindable: one interface?

- Each node referring to a type, function or variable implements Bindable.
- Pros:
  - Easy to implement.
  - Code is completely factorized.
- Cons:
  - No strong type checking, the definitions must be generic Dec, not FunctionDec, TypeDec or VarDec.

Bindable: one interface?

- Each node referring to a type, function or variable implements Bindable.
- Pros:
  - Easy to implement.
  - Code is completely factorized.
- Cons:
  - No strong type checking, the definitions must be generic Dec, not FunctionDec, TypeDec or VarDec.

Bindable: three interfaces?

- Each node referring to a type, function or variable implements respectively TypeBindable, FunctionBindable and VariableBindable.
- Pros:
  - Easy to implement.
  - Strong type checking.
- Cons:
  - Code is duplicated three times, the only difference being the

Bindable: three interfaces?

- Each node referring to a type, function or variable implements respectively TypeBindable, FunctionBindable and VariableBindable.
- Pros:
  - Easy to implement.
  - Strong type checking.
- Cons:
  - Code is duplicated three times, the only difference being the

Bindable: three interfaces?

- Each node referring to a type, function or variable implements respectively TypeBindable, FunctionBindable and VariableBindable.
- Pros:
  - Easy to implement.
  - Strong type checking.
- Cons:
  - Code is duplicated three times, the only difference being the

Bindable: three interfaces?

- Each node referring to a type, function or variable implements respectively TypeBindable, FunctionBindable and VariableBindable.
- Pros:
  - Easy to implement.
  - Strong type checking.
- Cons:
  - Code is duplicated three times, the only difference being types.

Bindable: three interfaces?

- Each node referring to a type, function or variable implements respectively TypeBindable, FunctionBindable and VariableBindable.
- Pros:
  - Easy to implement.
  - Strong type checking.
- Cons:
  - Code is duplicated three times, the only difference being types.

Bindable: three interfaces?

- Each node referring to a type, function or variable implements respectively TypeBindable, FunctionBindable and VariableBindable.
- Pros:
  - Easy to implement.
  - Strong type checking.
- Cons:
  - Code is duplicated three times, the only difference being types.

Bindable: one template interface!

- Each node referring to a type, function or variable implement respectively TypeBindable, FunctionBindable and VariableBindable, but those classes are written using a template.
- Pros:
  - Strong type checking.
  - Code is completely factorized.
- Cons:
  - A bit harder to implement.

Bindable: one template interface!

- Each node referring to a type, function or variable implement respectively TypeBindable, FunctionBindable and VariableBindable, but those classes are written using a template.
- Pros:
  - Strong type checking.
  - Code is completely factorized.
- Cons:
  - A bit harder to implement.

Bindable: one template interface!

- Each node referring to a type, function or variable implement respectively TypeBindable, FunctionBindable and VariableBindable, but those classes are written using a template.
- Pros:
  - Strong type checking.
  - Code is completely factorized.
- Cons:
  - A bit harder to implement.

Bindable: one template interface!

- Each node referring to a type, function or variable implement respectively TypeBindable, FunctionBindable and VariableBindable, but those classes are written using a template.
- Pros:
  - Strong type checking.
  - Code is completely factorized.
- Cons:
  - A bit harder to implement.

Bindable: one template interface!

- Each node referring to a type, function or variable implement respectively TypeBindable, FunctionBindable and VariableBindable, but those classes are written using a template.
- Pros:
  - Strong type checking.
  - Code is completely factorized.
- Cons:
  - A bit harder to implement.

Bindable: one template interface!

- Each node referring to a type, function or variable implement respectively TypeBindable, FunctionBindable and VariableBindable, but those classes are written using a template.
- Pros:
  - Strong type checking.
  - Code is completely factorized.
- Cons:
  - A bit harder to implement.

Bindable: one template interface!

Same kind of declaration than for DefaultVisitor and DefaultConstVisitor.

```
template <class DefinitionClassType>
class Bindable
{
  // FIXME.
}
// Aliases for template classes.
typedef Bindable<FunctionDec> FunctionBindable;
```

Bibliography I


Bjarne Stroustrup.
The C++ programming language third edition, 1997.