

# Towards More Sophisticated Access Control

József Mihalicza    Norbert Pataki    Zoltán Porkoláb  
Ádám Sipos



Dept. Programming Languages and Compilers  
Eötvös Loránd University, Budapest, Hungary

{pocok, patakino, gsd,  
shp}@inf.elte.hu

SPLST 2009

# Contents

- 1 Introduction and Motivation
- 2 C++ Concepts
- 3 Sophisticated Access Control
- 4 Conclusion

# Encapsulation, Information Hiding

- Encapsulation: Fundamental Concept in Object-Oriented Programming
- Minimizing the dependencies between among separataly-written modules
- Information hiding is not a mandatory part of encapsulation
- Information hiding via specific access control rules

# Motivating example – C++

```
class Person
{
public:
    string name;        // public for All
private:
    string address;    // public for Pal, Bank
    string nickname;   // public for Pal
    string accno;      // public for Bank
    string dreams;     // public for None
};
```

- public, private, protected: not enough sophisticated
- friend: does not help

# Access control in programming languages

	PRI	PUB	PRO	PAC	FRI	SEL	OBJ
Simula	+	+	+	-	-	-	-
Java	+	+	+	+	-	-	-
C#	+	+	+	+	-	-	-
C++	+	+	+	-	+	-	-
D	+	+	+	+	+	-	-
Eiffel	+	+	+	-	-	+	+

# Requirements for fine-grained visibility rules

- Access from a specific client
- Access from a group of clients
- Access from a distinct inheritance hierarchy
- Access from the inheritance hierarchy of the same class
- Access by role
- Interface function defined outside of a class
- Interface function for multiple classes
- Access by self
- Without runtime overhead

# Motivation

## Solution: Eiffel's selective visibility

- The designer of a class controls which clients may call its features
- Feature declarations are organised in groups according to their visibility
- special constructs, like `none`, `any`

# C++ Now

```
template <class T>
T min(const T& a, const T& b)
{
    return a < b ? a : b;
}
```

# C++ Now

- Only implementation shows the requirements of the template parameters
- Template declarations don't describe them
- Comprehensible error diagnostics, when the argument violate these requirements

# C++ Concepts

```
concept LessThanComparable<typename T>
{
    bool operator<(T a, T b);
};
```

- Concept maps: allow types to be explicitly bound to a concept.

# C++ Concepts

```
template<typename T>
requires LessThanComparable<T>
T min(const T& a, const T& b)
{
    return a < b ? a : b;
};
```

```
template<LessThanComparable T>
T min(const T& a, const T& b)
{
    return a < b ? a : b;
};
```

- Concepts can be combined with `operator!`, `operator&&`, `operator||`

# C++ Concepts

- ConceptGCC: translates concept-enabled code to current C++ standard code.
- Concepts have been removed from C++0x
- Delayed

# C++ Template Metaprograms

- template-s and specialization: functional framework
- Program parts that work in compilation-time
- Checks in compilation time (able to stop the compilation process)
- Create flexible libraries
- Speed-up the runtime programs

# Idea

- pass `this` (as an extra parameter)
- check it in compilation-time if this parameter's type is proper to the rules
- Template metaprograms
- No runtime overhead, just syntactical
- We have not found a library-based solution

# Specification of sophisticated access rules

```
class Person {
public:
    string& name(); /* accessible for all */
    accessible InType<_, Typelist(Pal,Bank)>
    string& address(); /* accessible from Pal or Ba
    accessible SameType<_,Pal>
    string& nickname(); /* accesible from Pal */
    accessible SameType<_,Bank>
    string& accno(); /* accesible from Bank */
```

# Specification of sophisticated access rules

```
private:  
    string& dreams(); /* not accessible */  
    string name_;  
    string address_;  
    string nickname_;  
    string accno_;  
    string dreams_;  
};
```

# Specification of sophisticated access rules

- new keyword: `accessible`
- `_`: the caller

# Solution

- 1 We hide the keyword `access` from the code but the concept inside remains. This code is passed to ConceptGCC.
  - 2 Set back the keyword, and EDG C++ Frontend Compiler analyze the code.
- We just catch the access violations

# Conclusion

- Modern OO languages do not support sophisticated access specification
- Eiffel: memberwise, selective access rules
- Our C++ approach: using template metaprograms to check access rules
- Library based vs. Concepts

# Future work

- Quantitative analysis based on bugreports, experience, etc.
- Library-based solution