[home page] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Structured data programming**

opens door to unprecedented advances in programming

including fully automatic persistence

**Idea:**

Remove declarations of data structures from the classes where they are burried now,

and move them into a block of statements resembling a database schema. Note the similarity:

*Structured programming Structured data programming*

Brings order to the messy codeBrings order to the messy data

*goto* statements not allowed pointer/reference members not allowed

in the application code in the application classes

Example [LINK]

Benefits [LINK]

Available Software[LINK]

Not Recommended Use [LINK]

[page example] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Example**

This C++ example shows the new style of representing relations among classes Faculty, Department, Teacher, and Course, repeatedly using the following data organization. The doubly linked list of Children can be NULL ending or a ring.

Parent

Child

Child

Child

This often occuring data organization cannot be represented with the existing templates/generics. For simplicity of this presentation, we will call it Aggregate.

// Data organization is specified in one block of declarations (schema)

// which is so clear it does not need any explanations

Association Aggregate<Faculty,Department> departments;

Association Aggregate<Department,Teacher> teachers;

Association Aggregate<Department,Course> courses;

Association Aggregate<Teacher,Course> teaches;

// The members that implement the individual data structures are

// transparently injected under ZZds

class Faculty {

ZZ\_Faculty ZZds; // see Note 1

…. all other code as usual

};

class Department {

ZZ\_Department ZZds; //see Note 1

…. all other code as usual

};

// …. and so on for all the other application classes

**Note 1:** This line must be in every application, it just repeats the class name. Simple template processor combines the schema with a special data structure library, and creates a file with declaration of classes such as ZZ\_Faculty or ZZ\_Department which hide all the pieces that implement the data structures. There is no code generator that would modify the application code.

// The application uses methods such as addTail() or remove() from the library

int main () {

Department\* d=new Department;

Teacher\* t=new Teacher;

Course\* c=new course;

teachers::addTail(d,t); // see Note 2

courses::addTail(d,c);

teaches::addTail(t,c);

…

Department\* dn=new Department;

courses::remove(c);

courses::addTail(dn,c);

….

This works also in other C languages including Objective C and in Java.

**Note 2:** Note the improved order of the parameters in these statements compared to the standard libraries. First the most important – the data organization identified by its name, then the operation to perform, finally the participating objects.

**Change of schema**

Let’s assume that we want the Teachers to be grouped under the Faculty, not under Department as in the original schema, and instead of the Aggregate, we want Teachers to be in a hash table for fast searches. We first change the schema which is self explanatory. The changes are highlighted in red:

Association Aggregate<Faculty,Department> departments;

Association Hash<Faculty,Teacher> teachers;

Association Aggregate<Department,Course> courses;

Association Aggregate<Teacher,Course> teaches;

This may require minor changes within the application code, but if you miss some the compiler(!) will tell you about them without using a debugger and walking through individual objects.

[page benefits] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Benefits**

1. At all times, all the designers and coders are fully aware of the data organization. This avoids conceptual errors, inefficient use of the data, and duplication. It tremendously improves communication among teams and maintenance. If there is a software architect, that would likely be the person in charge of the schema.
2. Our program Layout reads the schema and generates an eye-pleasing UML class diagram which is never outdated. Here are the diagrams generated for the two schemas used on the Example [LINK]page.
3. Objects and relations are persistent – can be just copied out or in from the disk – except for pointers embedded in them. Because all pointers are now handled transparently, implementation of the persistence is simple, elegant, efficient and safe, once for ever implemented within the library. No custom serialization functions are required.

We strongly believe this technology will revolutionize programming, and we are looking for programmers interested in expanding our open source libraries.

For all technical details, programming tricks, and extensive benchmarks, see book by Soukup&Machacek [LINK]

Benefits:

1. Instead of being burried in the application classes, relations have the same visibility as classes, improving clarity and the ease of modifications.
2. The coordination between the software architect and the coders is improved. At all times, the Schema is the concise description of the data organization.
3. The Schema is essentially a textual version of the UML class diagram, and can be used as the input for the program whichgenerates the visual UML class diagram.
4. Libraries designed in this style can include important data structures such as bi-directional aggregate or graphs based on pointer/reference chains. The current standard libraries cannot represent such intrusive data structures.
5. When pointer chains are implemented as rings, we get a better protection against pointer errors than in Java, which protects only about references to deleted objects.
6. Because that are no explicit pointers/references in application classes, all application objects can be automatically persistent. No custom coded serialization functions,another potential source of difficult errors, are required.
7. We manage internal data structures as if they were a DB but maintaining their original performance.
8. On many projects, the persistent data can be used as a fast memory-resident DB with the minimal footprint. This DB is not restricted to a limited number number of relations or data organizations (relational, graphs, etc).

The compounded effect of all this is so huge that if we tell you how much it shortens the development time and speeds up the data processing, you would not take it seriously. The results of extensive benchmarks are in Soukup&Macháček. [LINK]

[PAGE] Implementation

**Example 1: Removing raw pointers from the application classes in C++.**

Assume a C++ code dealing with Companies, Departments, and Employees.

The data representation with the STL library may look like this:

class Company {

std::unordered\_set<int> depts;

};

class Department {

int deptNo;

std::unordered\_set<std::string> empls;

};

class Employee {

std::string emplName;

Department \*myDept’; **// raw pointer**

};

In this representation , string and unordered\_set insert pointers into their host classes, but these pointers are transparent and safely managed by the library. However, container empls and pointer myDept form the bi-direction associaton A, which cannot be stored in the STL library as a template; it requires to insert members into more than a single class.

The new data representation declares separately the classes and the relations. String is considered one of the relations.

class Company {

\_Company exp;

};

class Department {

\_Department exp;

int deptNo;

};

class Employee {

\_Employee exp;

};

// ------------------------

String<Company> compName;

Set<Company,Department> depts;

String<Employee> emplName;

Aggregate<Department,Employee> empls;

Classes \_Company, \_Department, and \_Employee are created automatically by a simple code generator or eventually by the compiler. They insert piece that the application classes need for individual relations. For more details, see Soukup&Machacek [LINK].

**Example 2: Intrusive representation of the same data, using chains of pointers.**

Existing templates do not support intrusive data structures inolving more than just one class, so using the existing methods we just may code it from scratch:

class Company {

Department \*firstDept;

};

class Department {

int deptNo;

Department \*nextDept;

Department \*prevDept;

Employee \*firstEmpl;

};

class Employee {

char \*emplName;

Employee \*nextEmpl;

Employee \*prevEmpl;

Department \*myDept;

};

Committing an error when managing these pointers, is relatively easy.

The representation of the same data logic with the new library looks hides the pointers from the application domain:

class Company {

\_Company exp;

};

class Department {

\_Department exp;

int deptNo;

};

class Employee {

\_Employee exp;

};

// ------------------------

String<Company> compName;

IntrusiveCollection<Company,Department> depts;

String<Employee> emplName;

IntrusiveAggregate<Department,Employee> empls;

[PAGE]

Available software: