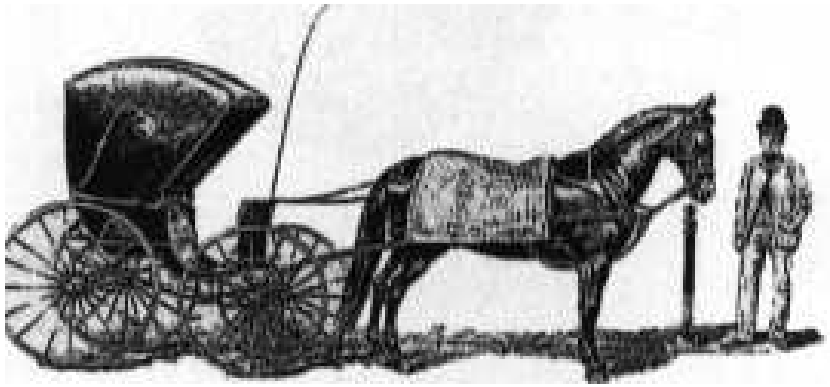
OBJECT ORIENTED APPROACHES IN EVOLUTIONARY COMPUTATION

Marc Schoenaeuer/ J. J. Merelo
Polytechnique/UGranada

SUMMARY

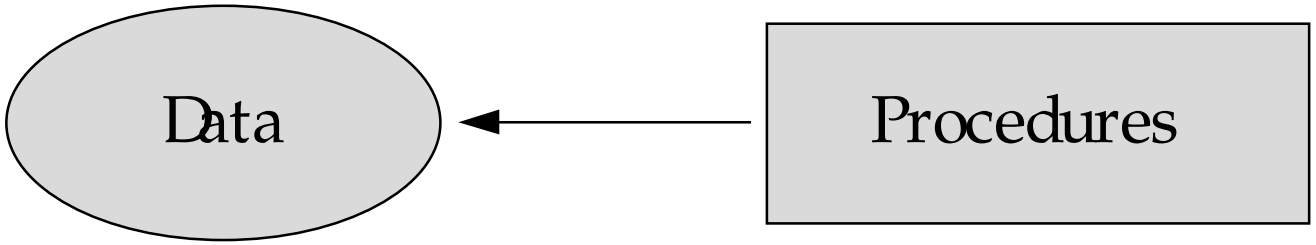
- Procedural/object oriented programming
- What is OOP and C++
- OO-Genetic Algorithms (and EC) and GAGS

PROCEDURAL PROGRAMMING



- Procedures act on data structures

```
struct coord {
    unsigned x,y;
} aCoord;
move(&aCoord);
```



OBJECT ORIENTED PROGRAMMING

Object

Data
Procedures = Methods = Messages

- Data and methods that handle it are together in *objects. Objects* with the same behavior are grouped in *classes*.

```
class coord {
    unsigned x,y;
    void move( void );
}
aCoord.move();
```

OOP FEATURES: ENCAPSULATION

- An object can be manipulated only through its *interface.*

```
class Stack {
public:
    void push( Object anObject);
    Object pop(void);
private:
    List aList;
}
```

➡ An object of class Stack can only be “pushed” or “popped”.

- This leads to a modular design: OO programs are easy to debug (if guidelines are followed)

☑ Reduces programming time.

OOP FEATURES: INHERITANCE

- Implements relation is-a.
  - ➡ Animals
    - » Breathe(Air& air)
    - » Die(void)
  - ➡ Mammals
    - » Suckle (Mom& aMom)
  - ➡ Persons
    - » PayTaxes(IRS& aVampire)
  - ➡ Men
    - » WatchSoccer(TV& aTV, popCornBag& aBag, Beer& aBottle)
  - ➡ Men can Breathe, Die, Suckle, PayTaxes and WatchSoccer, since aMan is-a Person, which is-a Mammal, which is-a animal : behavior and/or interface can be inherited.

OOP FEATURES: INHERITANCE II

- Inheritance leads to *code and design reuse:* once an object is designed and coded, its interface can be resued by creating a whole hierarchy of objects with the same characteristics, reusing the code or overriding it, but reusing the interface.
- Reduces design time.

OOP FEATURES: SUMMING UP

- Encapsulation : reduces coding time.
- Inheritance: reduces design time.
- ☑ Once a programmer has reached the Zen of OOP, program development time is quite shortened.

OO LANGUAGES AND OSES

- SmallTalk, Modula-3: Mainly academical
- CLISP: Used in symbolic AI.
- OO-COBOL (COOBOOL?), PERL5 by Larry Wall, Python by Guido Van Rossum.
- C++, ObjectiveC (and Objective C++), derived from C
- Operating systems: based on a document-oriented interface (DOI), as opposed to a file-program oriented paradigm: Plan9, BeOS, NextStep

WHITHER C++?

- Derived from a widely used language: C.
- ANSI standard coming up (right now, Draft White Paper: DWP).
- C++ is strongly typed, Objective C weakly typed and closer to C and SmallTalk.

C++ FEATURES

- ✓ Templates and Standard Template Library: type-safe type-independent programming.
- ✓ Multiple inheritance
- ✓ Exceptions (not quite there yet...)
- ✓ friends: objects can peek into other objects guts.
- ✓ Operator overloading: created types can have the same syntax than built-in types.
- ✓ Portable, wide and kowledgeable user base, good, efficient and portable compilers and libraries.

SOLVING PROBLEMS USING C++

- Indentify objects, classes and relationships among them: interfaces.
- OOP CASE Methodologies: Booch et al.
- *Advice: use common sense and experience*
- C++ is not C plus one: C++ is programmed in a completely different way than C.

OOGA,OOGA

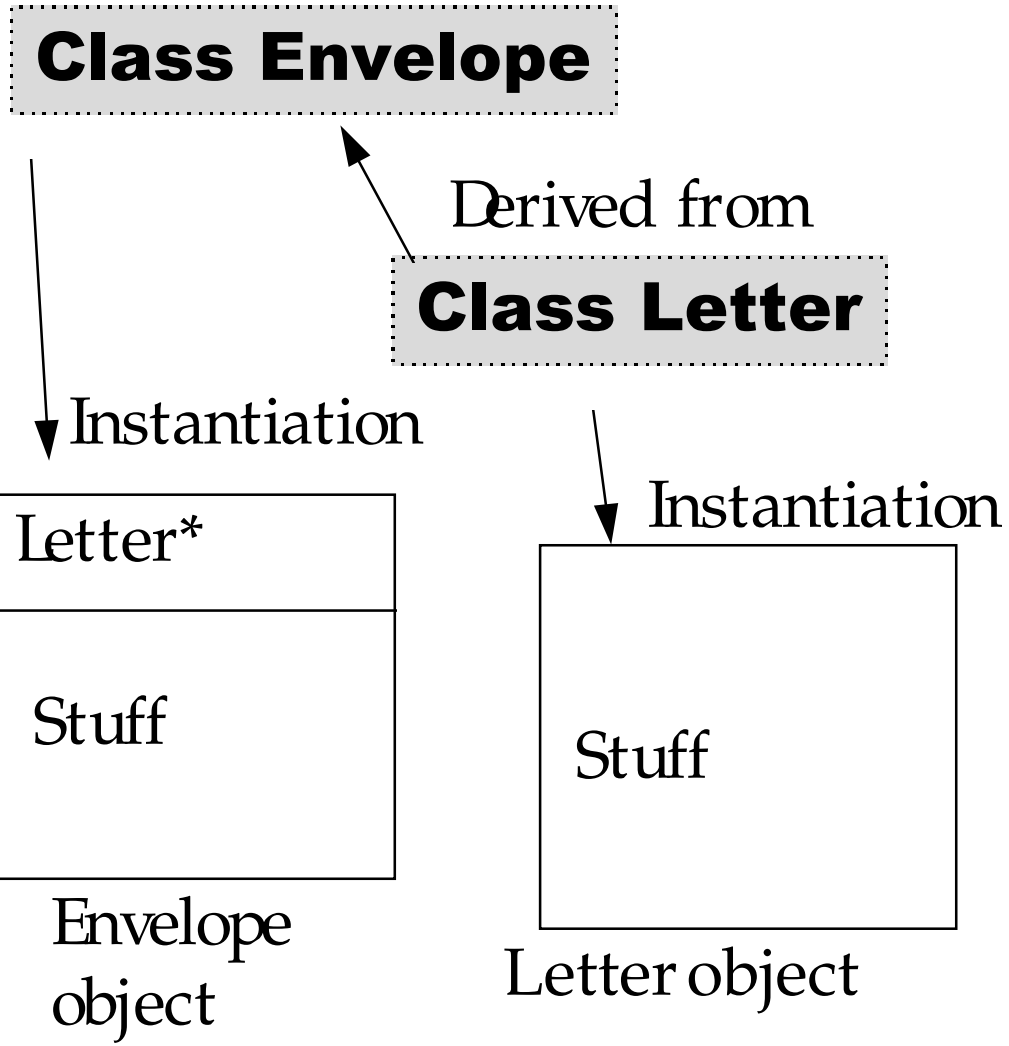
- Several publicly released EC object-oriented freeware packages: Matthew´s GAlib, Tang´s TOLKIEN, GAGS, and GAlib in Swarm (written in Objective C).
- **Common sense**: two classes: chromosome and population, auxiliary classes (file I/O, log, vector & array).
- **Experience:** genetic operators, selection and reproduction operators, fitness objects, views.

CLASSICAL CHROMOSOME

```
class chromosome {
    public:
        chromosome();
        mutate();
        crossOver(
          chromosome&
          otherChrom);
        print();
}
```

- What about "myStrangeMutationOp"? Or creep operators? Or "fixing" operators (used in constrained search)?
  - ➡ No can do: a new class, with new methods, must be created (inheriting maybe from chromosome).
  - ➡ Fixing is difficult, since it involves "directed" application of operators: violation of encapsulation.

LETTER/ENVELOPE CLASSES



- Taken from Coplien Advanced C++: programming styles and idioms. Related to *state* object in “Design patterns” (Gamma)
- Letter classes are derived from envelope classes. Envelope objects include a pointer to a letter object.
- Letter objects are handled from envelope.
- Allows changing object class at runtime, among other things.

CHROMOSOMES IN GAGS

```
class GAGSchrom {
public:
      GAGSchrom();
      unsigned getLength();
      unsigned operator[](unsigned);
protected:
      //bitString manipulation ops
}
// Genetic operators -- letter
class GAGSop {
public:
      virtual void operator() (GAGSchrom&);
      virtual void operator()
(List<GAGSChrom>&);
      virtual void applyAt(GAGSchrom&,
unsigned);
private:
      GAGSop* op;
}
// Genetic operators -- envelope
class Mutator: public GAGSop{
      virtual void operator()...
}
```

- Operators separated from the chromosome, and in its own hierarchy.
- Uses *letter-envelope* concept (Coplien): genetic operators are handled thru the base class, which includes objects of derived classes.
- Operators are *functors*: objects with function syntax.

USING CHROMOSOMES

```
// Create chromosome
GAGSchrom aChrom( NumberOfBits ),
  anotherChrom( anotherNumberOfBits );
//Create genetic operators
GAGSop Mutator(0.01), Crosser( 0.01,
  GAGSop:GXOVER);
// Example of application of genetic operators
Mutator( aChrom );
Crosser( aChrom, anotherChrom);
```

WHERE'S THE BIG DEAL?

- Flexible genetic operator class hierarchy, with well-designed interface. New operator? Create a new one, inheriting from GAGSop, redefine two functions.
- Operators are objects: they can be created, destroyed, altered at runtime: variable operator rates are possible, as well as testing different operators in the middle of a GA run.
- Chromosome decoding is another object: raw chromosome bits can be seen as whatever using *views.*

```
view<float> vista (SIZEGENES, -1.0, 2.0);
```

➡ Takes SIZEGENES bits and converts to range -1.0, 1.0

POPULATION CONCEPTS

- Same operator-object concept than chromosomes: operators that act on the population (selection and reproduction operators) should be outside it, as independent objects.
- *Fitness:* should be another object, since fitness is not always a floating point number: in general, it is a vector (used sometimes for multiobjective optimization).
  - ➡ Population is a template on the fitness type.
  - ➡ Fitness is an object with comparison operations defined (<, >, ==).

MORE POPULATION CONCEPTS

- Population includes dynamic lists with chromosomes *and genetic operators*.
- *Iterators* or cursors run over the chromosomes and operators, and manipulate them.
  - ➡ Operators and chromosomes can be added or eliminated dynamically at runtime: variable population size and island models are supported.
  - ➡ Genetic operator fitness is also supported, as well as variable operator rate: the operator list includes a key with the operator probability.

USING POPULATION OBJECTS

```
// Create Population
Population<fitness_t> myPop( genSize, SIZEGENES,
   popSize, mutRate );
// Set steady state parameter
myPop.setElite( 0.2 );
// Create iterator over the population
popIter<fitness_t> censo( myPop );
// create genetic operators (variable-size ops) and add them to
   population, with a rate
genOp adder( SIZEGENES*2, genOp::DUP);
myPop.addOp( &adder, 0.01 );
genOp killer( SIZEGENES*2, genOp::KILL);
myPop.addOp( &killer, 0.01 );
```

WHERE'S THE EC?

- Any chromosome and operator classes can be used on the population object: it is only a container for ranked population of objects that are substituted by new objects created from them.
  - ➡ A chromosome must only have a copy ctor, and genetic operators receive one or two chromosomes as operators.
- This provides hooks for different classes of chromosomes: real-valued, matrix-arranged, GP, and different selection and reproduction methods.

CONCLUSIONS

- OO can be profitably applied to EC programming.
- There are many and good-quality libraries avaiable for that purpose.
- Another library: GAGS, places an emphasis on portability and flexibility.

REFERENCES

- HitchHiker´s guide to evolutionary computation, by Beasley and Heitkötter:

`http://krypton.ugr.es/~encore`

- GAGS class library:

`http://kal-el.ugr.es/GAGS`