Object-Orientated Programming: An Overview

Eckart Bindewald

How to Represent a “Gene” in Code? (C++)

```
int kras_id = 3845;
string kras_symbol = “KRAS”;

// general function for printing gene information:
void print_gene(int id, string symbol) {
  cout << “ID: “ << id << “ symbol: “ << symbol << endl;
}


// call function:
print_gene(kras_id, kras_symbol);
```

Problems with this approach

- Need to “bundle” related data elements
- No easy way to “bundle” related functions
- Difficult to keep gene information consistent (change of gene id must be accompanied by change of gene symbol)
- Difficult to implement code for printing all current and future data structures

4 Pillars of Object-Oriented Programming

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

https://codewithmosh.com/p/object-oriented-programming-in-javascript/

A “Class” Representing a Gene (C++)

```
class Gene {
 public:
  int id = -1;
  string symbol = “undefined”;
}
// general function for printing gene information:
void print_gene(Gene gene) { // improved code: const Gene& gene
  cout << “ID: “ << gene.id << “ symbol: “ << gene.symbol << endl;
};


Gene kras;
kras.id = 3845;
kras.symbol = “KRAS”;
// call print function:
print_gene(kras);
```

A **class** is a data-structure that can bundle data and code

How to Represent a “Gene” in Code? (C++)

```
class Gene {
 public:
  int id = -1;
  string symbol = “undefined”;
}
// general function for printing gene information:
void print_gene(Gene gene) { // improved code: const Gene& gene
  cout << “ID: “ << gene.id << “ symbol: “ << gene.symbol << endl;
};

Gene kras;
kras.id = 3845;
kras.symbol = “KRAS”;
// call print function:
print_gene(kras);
```

Variable kras is an **object** (a.k.a. instance) of **class** Gene

Achieved “**Encapsulation**”: simple form of bundling data!

A **class** is a data-structure that can bundle data and code

From Functions to Methods

```
class Gene {
 public:
  int id = -1;
  string symbol = “undefined”;
  // general ~~function~~ method for printing gene information:
 void print() {
  cout << “ID: “ << id << “ symbol: “ << symbol << endl;
 }
};


Gene kras;
kras.id = 3845;
kras.symbol = “KRAS”;
// call print method:
kras.print();
```

A **method** is a function that is part of the definition of a class

Object-Oriented Programming

Object-oriented programming uses data-structures that bundles **data** with the **code** that can operates on the data

Constructors Simplify Creation of Objects

```
class Gene {
 public:
   int id = -1;
   string symbol = “undefined”;
   // Constructor:
   Gene(int _id, const string& _symbol) { id = _id; symbol = _symbol; }
   // Method for printing gene information:
   void print() {
     cout << “ID: “ << id << “ symbol: “ << symbol << endl;
   }
};


Gene kras(3845, “KRAS”);
// call print function:
kras.print();
```

A **constructor** is a special method called for creating an object

Constructors Simplify Creation of Objects :

```
class Gene {
 public:
   int id = -1;
   string symbol = “undefined”;
   // Constructor:
   Gene(int _id, const string& _symbol) { id = _id; symbol = _symbol; }
   // Method for printing gene information:
   void print() {
     cout << “ID: “ << id << “ symbol: “ << symbol << endl;
   }
};
```

Achieved “Abstraction”: simplified way to manipulate data

```
Gene kras(3845, “KRAS”);
// call print method:
kras.print();
```

A **constructor** is a special method called for creating an object

Using Commonalities via "Inheritance"

- Imagine you wan to program a genome browser
- You want to display genomic regions associated with genes
- Not all genomic regions of interest are "genes" (repeat elements, promoter regions, regulatory elements etc.)
- Different display code for each type of genomic element?

Commonality Between All Genomic Elements

```
class GenomicElement {
 public:
  string chromosome = “undefined”;
  long start = -1;
  long end = -1;
  string strand = “+”;

  void print() { cout << “chromosome: “ << chromosome << “:“
                                << start << “-” << end << endl;  }

}
```

Methods have access to Superclass:

```
class Gene : public GenomicElement {
 public:
  int id = -1;
  string symbol = "undefined";

  // Constructor:
  Gene(int _id, const string& _symbol) { id = _id; symbol = _symbol; }

  // Method for printing gene information:
  void print() {
    cout << "Gene ID: " << id << " symbol: " << symbol
         << "Chromosome:" << chromosome << endl;
  }
};

void print_elements(const GenomicElement& x) {
  x.print(); // which version of print method is being called?
}

Gene kras(3845, "KRAS");
kras.chromosome = "chr12"; // defined in superclass "GenomicElement"!

// call print function:
kras.print(); // which version of "print" is called? Derived class

print_elements(kras); // which version of "print" is called? Superclass - Unexpected problem!

// output (incorrect):
> chr12:-1--1
```

Have achieved inheritance: use commonality between different types of data

Polymorphism: Swap Code as Needed

```
class Gene : GenomicElement {
 public:
  int id = -1;
  string symbol = “undefined”;

  // Constructor:
  Gene(int _id, const string& _symbol) { id = _id; symbol = _symbol; }

  // Re-defining print method (“overloading”)
  virtual void print() {

    cout << “Gene ID: “ << id << “ symbol: “ << symbol

         << “Chromosome:” << chromosome << endl;

  }
};

void print_elements(const GenomicElement& x) {

  x.print(); // which version of print method is being called?

}


Gene kras(3845, “KRAS”);
kras.chromosome = “chr12”; // defined in superclass “GenomicElement”!

// call print methods in 2 ways:
kras.print();
print_elements(kras); // “print” method from derived class as desired!


// output (correct):
> Gene ID: 3845 symbol: KRAS chromosome: chr12
```

Polymorphism: Use of Superclass + Derived Class

```
class Gene : public GenomicElement {
 public:
  int id = -1;
  string symbol = “undefined”;

  // Constructor:
  Gene(int _id, const string& _symbol) { id = _id; symbol = _symbol; }

  // Re-defining print method (“overloading”)
  virtual void print() {

    GenemicElement::print(); // explicitely call print method from superclass
    cout << “Gene ID: “ << id << “ symbol: “ << symbol

         << “Chromosome:” << chromosome << endl;

  }
};


Gene kras(3845, “KRAS”);
kras.chromosome = “chr12”; // defined in superclass “GenomicElement”!

// call print function:
kras.print(); // which version of ”print” is called? From derived class as desired!

// output (correct):
> chr12:-1--1
> Gene ID: 3845 symbol: KRAS chromosome: chr12
```

Polymorphism: Use of Superclass + Derived Class (Java)

```
class Gene : public GenomicElement {

  public int id = -1;
  public String symbol = "undefined";

  // Constructor:
  public Gene(int _id, String _symbol) { id = _id; symbol = _symbol; }

  // Re-defining print method ("overloading")
  public void print() {

    super.print(); // explicitly call print method from superclass
    System.out.println("Gene ID: " + id + " symbol: " + symbol + "Chromosome:" + chromosome); }
}


Gene kras = new Gene(3845, "KRAS");
kras.chromosome = "chr12"; // defined in superclass "GenomicElement"!

// call print function:
kras.print(); // which version of "print" is called? From derived class as desired!

// output (correct):
> chr12:-1--1
> Gene ID: 3845 symbol: KRAS chromosome: chr12
```

4 Pillars of Object-Oriented Programming

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

https://codewithmosh.com/p/object-oriented-programming-in-javascript/

4 Pillars of Object-Oriented Programming

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

https://codewithmosh.com/p/object-oriented-programming-in-javascript/

Potential Pitfalls…

Deep Copy vs Shallow Copy – C++

```
class Gene {
 public:
  int id = 1;
};
```

```
Gene a;
Gene b = a; // b is bytewise-copy
b.id = 2;

a.id == 1; // a has not changed!
```

Deep Copy vs Shallow Copy – C++

```
class Gene {
 public:
  int id = 1;
  String symbol = “default”;
};
```

```
Gene a;
Gene b = a; // b is bytewise-copy. a and b different in memory (but not attribute ‘symbol’)
b.id = 2;
b.symbol = “KRAS”;
```

a.id == 1; // a has **not** changed!

a.symbol == “KRAS”; // *symbol* **has** changed because is was shallow copy (points to same address in memory)

Byte-wise Copying: Deep and Shallow Copies of Primitive and Complex Attributes Respectively

Gene a:
 int id= 1
 string symbol;

Gene b:
 int id = 1
 string symbol;

‘default’

For variable b: Changed id to 2 and symbol to “KRAS”:

Gene a:
 int id = 1
 string symbol;

Gene a:
 int id = 2
 string symbol;

‘KRAS’

Likely an inconsistent state for variable a! Variable a likely ‘corrupted’!

Bytewise Copying: Shallow Copies of Complex Attributes May Lead to Crash

Gene a:
int id = 1
string symbol;

Gene b:
int id = 1
string symbol;

'default'

If we want to dispose variable b (call "destructor" method): may free common memory of "symbol":

Gene a:
int id = 1
string symbol;

Gene a:
int id = 2
string symbol;

'default'

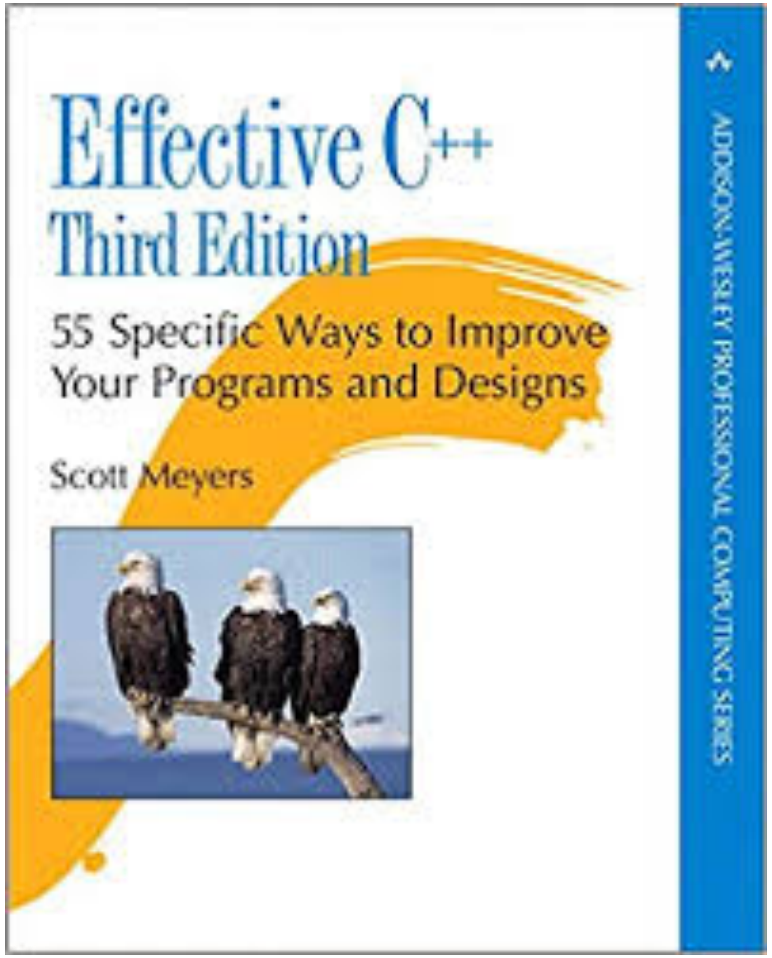
Attempting to access a.symbol will lead to **crash** of program!!

Potential Pitfalls of Copying Objects (C++):

- Default behavior of compiler: assignment operator ("=") corresponds to bytewise copying of objects
- Bytewise copying of objects may be OK if all attributes are primitive datatypes (int, double) without classes or pointers
- Bytewise copying may be disastrous if class contains attributes that are themselves classes or contain pointers
- Different behavior in different programming languages (C++,Java,Python)

Solution: Take Full Control over Life-Cycle (Construction, Copying, Destruction)

- Define constructor(s)
- Define assignment operator
- Define destructor
- “Best practices” from book Effective C++ by Scott Meyers



FIXED Deep Copy vs Shallow Copy – C++

```
class Gene {
 public:
  int id = 1;
  string symbol = “default”;

  // Regular constructor
  Gene(int _id, const String& _symbol) { id = _id; symbol = _symbol; }
  // Copy constructor: Ensures that a(b)  is deep copy
  Gene(const Gene& other) { copy(other); }

  // Assignment operator: ensures that a = b is deep copy
  virtual Gene& operator = (const Gene& other) {

    if (this != &other) { // avoid self-assignment (example: a=a)
      copy(other);

    }
    return *this;

  }

 // destructor . “virtual” destructor ensures that correct destructor is called for derived class
  virtual ~Gene() { }

  virtual void copy(const Gene& other) {

    id = other.id;
    symbol = other.symbol;

  }
};
```

FIXED Deep Copy vs Shallow Copy – C++

Using improved class Gene will lead to improved behavior:

```
Gene a;
Gene b = a; // b is is deep copy because we implemented assignment operator
b.id = 2;
b.symbol = "KRAS";

a.id == 1; // a has not changed!
a.symbol == "default"; // symbol not changed because is was deep copy (points to different address in memory)


// destruction of object b will not affect object a
```

Deep versus Shallow Copy – Python Example

```
[3]  a = [5,9]
     b = a
     print("a:",a,"id(a):",id(a))
     print("b:",b, "id(b):",id(b))
```

```
a: [5, 9] id(a): 139626418300936
b: [5, 9] id(b): 139626418300936
```

Setting b[0]=7 has **sideeffect** on a:

```
[4]  b[0] = 7
     print("a:",a,"id(a):",id(a))
     print("b:",b, "id(b):",id(b))
```

```
a: [7, 9] id(a): 139626418300936
b: [7, 9] id(b): 139626418300936
```

Reproduce in Browser using Python Jupyter Notebook via Google Colab (requires Gmail sign in):
https://colab.research.google.com/drive/1Q9GqfCi1jIf7n-_Qj0SZifc5fx9it6Km

Deep versus Shallow Copy – Python Example

- Even though we only changed variable `b`, the content of variable `a` is also changed (!)
- Reason: `b` is *shallow* copy of variable `a` (points to same address in memory)
- Sometimes desired (very fast copy), sometimes source of confusion and errors
- Solution: `deepcopy` method

```
[7]  import copy
     c = copy.deepcopy(a)
```

Now we can safely change new variable `c` without changing the variable it was copied from:

```
[6]  c[1] = 11
     print("a:",a,"id(a):",id(a))
     print("b:",b, "id(b):",id(b))
     print("c:",c, "id(c):",id(c))
```

```
a: [7, 9] id(a): 139626418300936
b: [7, 9] id(b): 139626418300936
c: [7, 11] id(c): 139626418397384
```

Deep versus Shallow Copy - Java

- Gene a = new Gene();
- Gene b = a; // copies only reference – shallow copy
- Gene c = a.clone(); // deep copy (depending on clone method implementation)

Sticking to “Best Practices” Avoids Pitfalls

- Take full control over lifecycle: constructor, copying, destructor
- Avoid default implementations: bytewise copying of objects will be disastrous for non-trivial classes
- Use or imitate “boiler-plate code”
- Book: Effective C++ by Scott Meyers

Highlights from “Effective C++”

- A class should be minimal but complete
- Minimal: no methods that do not really need special access
- Complete:
  Copy constructor
  Assignment operator
  get and set methods [example: getId(); setId(int id)]
  “destructor”
  All methods with keyword “virtual”

Violation of “minimal” principle:

```
class Gene {
 public:
  int id = 1;
  String symbol = “default”;


  // standard code
  // …
  // …

 // probably too specialized; introduces dependency
  String searchPubmed() {
    return “code_should_be_outside_of_class”;
  }

};
```

Violation of “minimal” principle - solution:

```
class Gene {
 public:
  int id = 1;
  string symbol = “default”;

  // standard code
  // …
  // …
};

 // special use cases are outside of class:
  string searchPubmed(const Gene&  gene) {
    return “to_be_implemented”;
  }
```

Violation of “minimal” principle - solution:

```
class Gene {
 public:
  int id = 1;
  string symbol = “default”;

  // standard code
  // …
  // …
};

 // special use cases are outside of class:
  class GeneTools {
    public:
    string searchPubmed(const Gene&  gene) {
    return “to_be_implemented”;
  }
 }
```

Deep Copy vs Shallow Copy – Java++

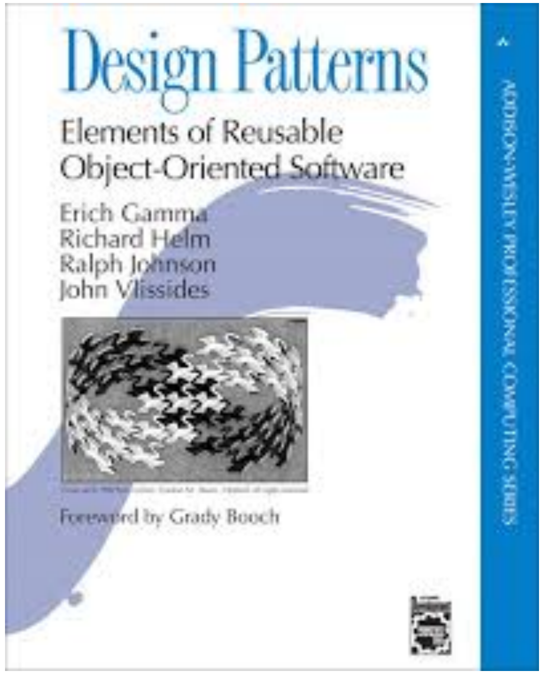
```
class Gene {
 public int id = 1;
}

Gene a = new Gene();
Gene b = a; // b is reference to a – shallow copy
b.id = 2;
a.id == 2; // a has changed !
```

Fixed: Deep Copy vs Shallow Copy – Java++

```
class Gene {
  public int id = 1;
  public Gene clone(Gene other) {
    Gene result = new Gene();
    result.id = other.id;
    return result;
  }
}

Gene a = new Gene();
Gene b = a.clone(); // b is deep copy
b.id = 2;
a.id == 1; // a has not changed !
```

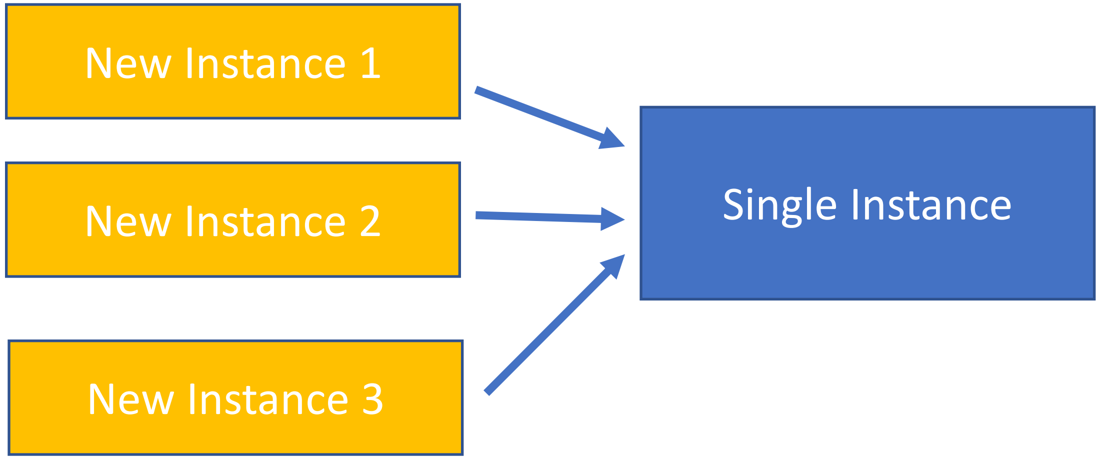


Design Patterns

- Authors analyzed successful software projects
- Found reoccurring “patterns” of object-oriented designs
- Defines common language of patterns

Example Design Pattern: Singleton

- Sometimes, we really just want one object of an entity, not unrelated copi
- Examples: database connection, random number generator



Example Design Pattern: Singleton (Java)

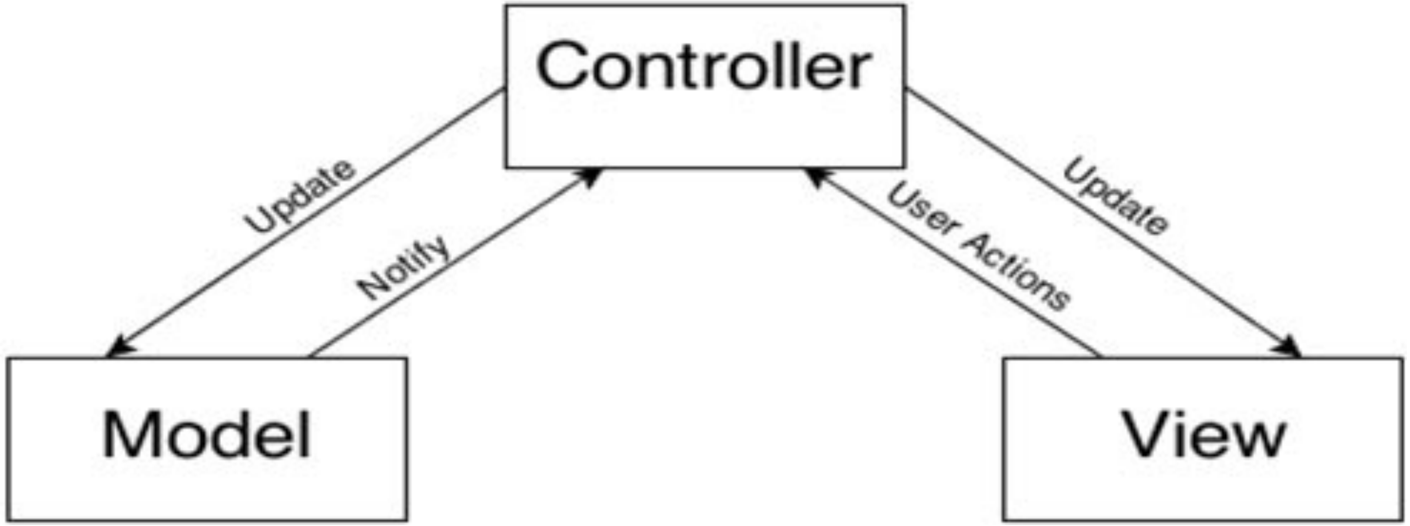
- Sometimes, we really just want one object of an entity, not unrelated copies
- Implementation: make constructor inaccessible from outside (private)

```
public class Singleton
   {
       private static Singleton instance;
       private Singleton() { }
       public static Singleton GetInstance()
       {
         if (instance == null)
         {
            instance = new Singleton();
         }
         return instance;

       }
   }
```

Design Pattern: Model-View-Controller

- Interprets user commands
- Updates data
- Updates user view



- Database
- State Variables

- User interface
- Web interface
- Command line interface

Big Picture

- Object oriented ideas are uses in most programming languages
- Tool for designing software with structured complex data or interfaces
- *Using* well-designed system of code classes can be productive
- *Creating* well-designed system code classes can be challenging
- In science, we often can make do with traditional functions (input, algorithm, output; no “user-loop”)
- More important for application software and larger systems
- Different takes on concept of object orientation in different languages
- Design patterns define common terminology for high-level object-oriented designs.