Abstract Class

Pure Virtual Functions

Based on `examples/11/Function.h` and `Function.cc`

- `virtual` functions with no implementation
  - All derived classes **are required** to implement these functions
- Typically used for functions that cannot be implemented (or at least in an unambiguous way) in the base case
- **Abstract class**: a class with at least one pure virtual method

```
class Function {
  public:
    Function(const std::string& name);
    virtual double value(double x) const = 0;
    virtual double integrate(double x1, double x2) const = 0;
    virtual void print() const;
    virtual std::string name() const { return name_; }

  private:
    std::string name_;
};
```

= 0 is called pure specifier

```
#include "Function.h"
#include <iostream>

Function::Function(const std::string& name) {
  name_ = name;
}
```

ConstantFunction

Based on `examples/11/ConstantFunction.*`

```
#include "ConstantFunction.h"

ConstantFunction::ConstantFunction(const std::string& name, double value):
  Function(name) {
  value_ = value;
}

double ConstantFunction::value(double x) const {
  return value_;
}

double ConstantFunction::integrate(double x1, double x2) const {
  return (x2-x1)*value_;
}
```

```
#ifndef ConstantFunction_h
#define ConstantFunction_h

#include <string>
#include "Function.h"

class ConstantFunction : public Function {
  public:
    ConstantFunction(const std::string& name, double value);
    virtual double value(double x) const;
    virtual double integrate(double x1, double x2) const;

  private:
    double value_;
};
#endif
```

Typical Error with Abstract Class

Based on `examples/11/Abstract1.cpp`

```
#include <string>
#include <iostream>
using namespace std;

#include "Function.h"

int main() {

  Function* gauss = Function("Gauss");

  return 0;
}
```

```
$ g++ -o Abstract1 Abstract1.cpp Function.cc
Abstract1.cpp:9:22: error: allocating an object of abstract class type 'Function'
  Function* gauss  = Function("Gauss");
                     ^
./Function.h:9:20: note: unimplemented pure virtual method 'value' in 'Function'
    virtual double value(double x) const = 0;
                   ^
./Function.h:10:20: note: unimplemented pure virtual method 'integrate' in 'Function'
    virtual double integrate(double x1, double x2) const = 0;
                   ^
1 error generated.
```

- Cannot make an object of an abstract class!
- Pure virtual methods not implemented and the class is effectively incomplete

`virtual` and Pure `virtual`

- No default implementation for pure virtual
  - Requires explicit implementation in derived classes
- Use pure virtual when
  - Need to enforce policy for derived classes
  - Need to guarantee public interface for all derived classes
  - You expect to have certain functionalities but too early to provide default implementation in base class
  - Default implementation can lead to error
    - User forgets to implement correctly a virtual function
    - Default implementation is used in a meaningless way
- Virtual allows polymorphism
- Pure virtual forces derived classes to ensure correct implementation

Abstract and Concrete Classes

- Abstract classes are incomplete
  - At least one method not implemented
  - Compiler has no way to determine the correct size of an incomplete type

- **Cannot instantiate an object of Abstract class**

- Usually abstract classes are used in higher levels of hierarchy
  - Focus on defining policies and interface
  - Leave implementation to lower level of hierarchy

- Abstract classes used typically as pointers or references to achieve polymorphism
  - Point to objects of sub-classes via pointer to abstract class

Example of Bad Use of `virtual`

Based on `examples/11/BadFunction.cpp`

```
class BadFunction {
  public:
    BadFunction(const std::string& name){
      name_ = name;
    }
    virtual double value(double x) const { return 0; }
    virtual double integrate(double x1, double x2) const { return 0; }

  private:
    std::string name_;
};

class Gauss : public BadFunction {
  public:
    Gauss(const std::string& name, double mean, double width) : BadFunction(name) {
      mean_ = mean;
      width_ = width;
    }
    virtual double value(double x) const {
      double pull = (x-mean_)/width_;
      double y = (1/sqrt(2.*3.14*width_)) * exp(-pull*pull/2.);
      return y;
    }

  private:
    double mean_;
    double width_;
};
```

Default dummy implementation

Implement `value()` correctly but use default `integrate()`

```
int main() {

  Gauss g1("g1",0.,1.);
  cout << "g1.value(2.): " << g1.value(2.) << endl;
  cout << "g1.integrate(0.,1000.): "
       << g1.integrate(0.,1000.) << endl;

  return 0;
}
```

```
$ g++ -o BadFunction BadFunction.cpp
$ ./BadFunction
g1.value(2.): 0.0540047
g1.integrate(0.,1000.): 0
```

We can use ill-defined `BadFunction` and wrongly use **`Gauss`**!

Function and **BadFunction**

```
class Function {
  public:
    Function(const std::string& name);
    virtual double value(double x) const = 0;
    virtual double integrate(double x1, double x2) const = 0;
    virtual void print() const;
    virtual std::string name() const { return name_; }

  private:
    std::string name_;
};
```

```
class BadFunction {
  public:
    BadFunction(const std::string& name){
      name_ = name;
    }
    virtual double value(double x) const { return 0; }
    virtual double integrate(double x1, double x2) const { return 0; }

  private:
    std::string name_;
};
```

Cannot instantiate `Function` because abstract

`BadFunction` can be used

Try it on your own!

Use of `virtual` in Abstract Class `Function`

Based on `examples/11/Function.h` and `Function.cc`

```
class Function {
  public:
    Function(const std::string& name);
    virtual double value(double x) const = 0;
    virtual double integrate(double x1, double x2) const = 0;
    virtual void print() const;
    virtual std::string name() const { return name_; }

  private:
    std::string name_;
};
```

```
#include "Function.h"
#include <iostream>

Function::Function(const std::string& name) {
  name_ = name;
}

void Function::print() const {
  std::cout << "Function with name " << name_ << std::endl;
}
```

- Default implementation of `name()`
  - Unambiguous functionality: user will always want the name of the particular object regardless of its particular subclass
- `print()` can be overriden in sub-classes to provide more details about sub-class, but still a function with a name

Concrete Class `Gauss`

Based on `examples/11/Gauss.*` and `Abstract2.cpp`

```
#include "Gauss.h"
#include <cmath>
#include <iostream>
using std::cout;
using std::endl;

Gauss::Gauss(const std::string& name, double mean, double width) : Function(name) {
  mean_ = mean;
  width_ = width;
}

double Gauss::value(double x) const {
  double pull = (x-mean_)/width_;
  double y = (1/sqrt(2.*3.14*width_)) * exp(-pull*pull/2.);
  return y;
}

double Gauss::integrate(double x1, double x2) const {
  cout << "Sorry. Gauss::integrate(x1,x2) not implemented yet..."
       << "returning 0. for now..." << endl;
  return 0;
}

void Gauss::print() const {
  cout << "Gaussian with name: " << name()
       << " mean: " << mean_
       << " width: " << width_
       << endl;
}
```

```
#ifndef Gauss_h
#define Gauss_h

#include <string>
#include "Function.h"

class Gauss : public Function {
  public:
    Gauss(const std::string& name, double mean, double width);

    virtual double value(double x) const;
    virtual double integrate(double x1, double x2) const;
    virtual void print() const;

  private:
    double mean_;
    double width_;
};
#endif
```

```
int main() {

  Function* g1 = new Gauss("g1",0.,1.);
  g1->print();
  double x = g1->integrate(0., 3.);

  return 0;
}
```

```
$ g++ -o Abstract2 Abstract2.cpp Function.cc Gauss.cc
$ ./Abstract2
Gaussian with name: g1 mean: 0 width: 1
Sorry. Gauss::integrate(x1,x2) not implemented yet...returning 0. for now...
```

Problem with Destructors

Based on `examples/11/Abstract3.cpp`

- We now want to properly delete the `Gauss` object

```
$ g++ -o Abstract3 Abstract3.cpp Function.cc Gauss.cc
Abstract3.cpp:14:3: warning: delete called on 'Function' that is abstract but has non-virtual destructor [-Wdelete-abstract-non-virtual-dtor]
  delete g1;
  ^

1 warning generated.
./Abstract3
Gaussian with name: gauss mean: 0 width: 1
Sorry. Gauss::integrate(x1,x2) not implemented yet...returning 0. for now...
Trace/BPT trap: 5
```

```
int main() {

  Function* g1 = new Gauss("gauss",0.,1.);
  g1->print();
  double x = g1->integrate(0., 3.);

  delete g1;

  return 0;
}
```

- In general with polymorphism and inheritance it is a **very good** idea to use virtual destructors

- Particularly important when using dynamically allocated objects in constructors of polymorphic objects

Revisit `Person` and `Student`

Based on `examples/11/Polymorphism7.cpp`

```
int main() {

  Person* p1 = new Student("Susan", 123456);
  Person* p2 = new GraduateStudent("Paolo", 9856, "Physics");

  delete p1;
  delete p2;

  return 0;
}
```

```
Person::~Person() {
  cout << "~Person() called for " << name_ << endl;
}
```

```
Student::~Student() {
  cout << "~Student() called for name:" << name()
       << " and id: " << id_ << endl;
}
```

```
GraduateStudent::~GraduateStudent() {
  cout << "~GraduateStudent() called for name:" << name()
       << " id: " << id()
       << " major: " << major_ << endl;
}
```

```
$ g++ -o Polymorphism7 Polymorphism7.cpp {Person,Student,GraduateStudent}.cc
$ ./Polymorphism7
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
~Person() called for Susan
~Person() called for Paolo
```

- Note that `~Person()` is called and not the destructor of the derived class!
- We did not declare the destructor to be virtual
- Handle type and not object type determines the destructor called! Non-polymorphic behaviour

Virtual Destructors

- Derived classes might allocate memory dynamically
  - Derived-class destructor (if correctly written!) will take care of cleaning up memory upon destruction
  - Base-class destructor will not do the proper job if called for a derived-class object
- Declaring destructor to be virtual is a simple solution to prevent memory leak using polymorphism
- **Virtual destructors ensure that memory leaks do not occur when one deletes an object via base-class pointer**

Simple Example of `virtual` Destructor

Based on `examples/11/*VirtualDtor.cpp`

```
#include <iostream>

using std::cout;
using std::endl;

class Base {
  public:
  Base(double x) {
    x_ = new double(x);
    cout << "Base(" << x << ") called" << endl;
  }
  ~Base() {
    cout << "~Base() called" << endl;
    delete x_;
  }
  private:
   double* x_;
};

class Derived : public Base {
  public:
  Derived(double x) : Base(x){
    cout << "Derived("<<x<<") called" << endl;
  }
  ~Derived() {
    cout << "~Derived() called" << endl;
  }
};

int main() {
  Base* a = new Derived(1.2);
  delete a;
  return 0;
}
```

Destructor not virtual

```
$ g++ -o -Wall -o NoVirtualDtor NoVirtualDtor.cpp
$ ./NoVirtualDtor
Base(1.2) called
Derived(1.2) called
~Base() called
```

```
#include <iostream>

using std::cout;
using std::endl;

class Base {
  public:
  Base(double x) {
    x_ = new double(x);
    cout << "Base(" << x << ") called" << endl;
  }
  virtual ~Base() {
    cout << "~Base() called" << endl;
    delete x_;
  }
  private:
   double* x_;
};

class Derived : public Base {
  public:
  Derived(double x) : Base(x){
    cout << "Derived("<<x<<") called" << endl;
  }
  virtual ~Derived() {
    cout << "~Derived() called" << endl;
  }
};

int main() {
  Base* a = new Derived(1.2);
  delete a;
  return 0;
}
```

Virtual destructor

Virtual destructor

```
$ g++ -o -Wall -o VirtualDtor VirtualDtor.cpp
$ ./VirtualDtor
Base(1.2) called
Derived(1.2) called
~Derived() called
~Base() called
```

Revised Class `Student`

Based on `examples/12/Revised/Student.h`

```
class Student : public Person {
  public:
    Student(const std::string& name, int id);
    ~Student();
    virtual void print() const;
    int id() const { return id_; }

    void addCourse(const std::string& course);
    const std::vector<std::string>* getCourses() const;
    void printCourses() const;

  private:
    int id_;
    std::vector<std::string>* courses_;
};
```

New methods and data member

Revised Class `Student`

Based on `examples/12/Revised/Student.cc`

New methods

```
void Student::addCourse(const std::string& course) {
  courses_->push_back( course );
}

void Student::printCourses() const {
  cout << "student " << name()
       << " currently enrolled in following courses:"
       << endl;

  for(int i=0; i<courses_->size(); ++i) {
    cout << (*courses_)[i] << endl;
  }
}

const std::vector<std::string>* Student::getCourses() const {
  return courses_;
}
```

Changes to pre-existing methods

```
Student::Student(const std::string& name, int id) : Person(name) {
  id_ = id;
  courses_ = new std::vector<std::string>();
  cout << "Student(" << name << ", " << id << ") called" << endl;
}

Student::~Student() {
  delete courses_;
  courses_ = 0; // null pointer
  cout << "~Student() called for name:" << name() << " and id: “
       << id_ << endl;
}

void Student::print() const {
  cout << "I am Student " << name() << " with id " << id_ << endl;
  cout << "I am now enrolled in " << courses_->size() << " courses." <<
endl;
}
```

Example of Memory Leak with `Student`

Based on `examples/12/Revised/StudentMemLeak.cpp`

```
#include <string>
#include <iostream>
using namespace std;

#include "Person.h"
#include "Student.h"

int main() {

  Student* p1 = new Student("Susan", 123456);
  p1->addCourse(string("algebra"));
  p1->addCourse(string("physics"));
  p1->addCourse(string("Art"));
  p1->printCourses();

  Student* paolo = new Student("Paolo", 9856);
  paolo->addCourse("Music");
  paolo->addCourse("Chemistry");

  Person* p2 = paolo;

  p1->print();
  p2->print();

  delete p1;
  delete p2;

  return 0;
}
```

```
$ g++ -Wall -o StudentMemLeak StudentMemLeak.cpp {Student,GraduateStudent,Person}.cc
StudentMemLeak.cpp:25:3: warning: delete called on non-final 'Student' that has virtual functions but non-virtual destructor [-Wdelete-non-abstract-non-virtual-dtor]
  delete p1;
  ^
StudentMemLeak.cpp:26:3: warning: delete called on non-final 'Person' that has virtual functions but non-virtual destructor [-Wdelete-non-abstract-non-virtual-dtor]
  delete p2;
  ^
2 warnings generated.
$ ./StudentMemLeak
Person(Susan) called
Student(Susan, 123456) called
student Susan currently enrolled in following courses:
algebra
physics
Art
Person(Paolo) called
Student(Paolo, 9856) called
I am Student Susan with id 123456
I am now enrolled in 3 courses.
I am Student Paolo with id 9856
I am now enrolled in 2 courses.
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Person() called for Paolo
```

Memory leak when deleting `p2` because nobody deletes Paolo’s `courses_`

Need to extend polymorphism also to destructors to ensure that object type and not pointer determines correct destructor to be called

Virtual Destructor for `Person` and `Student`

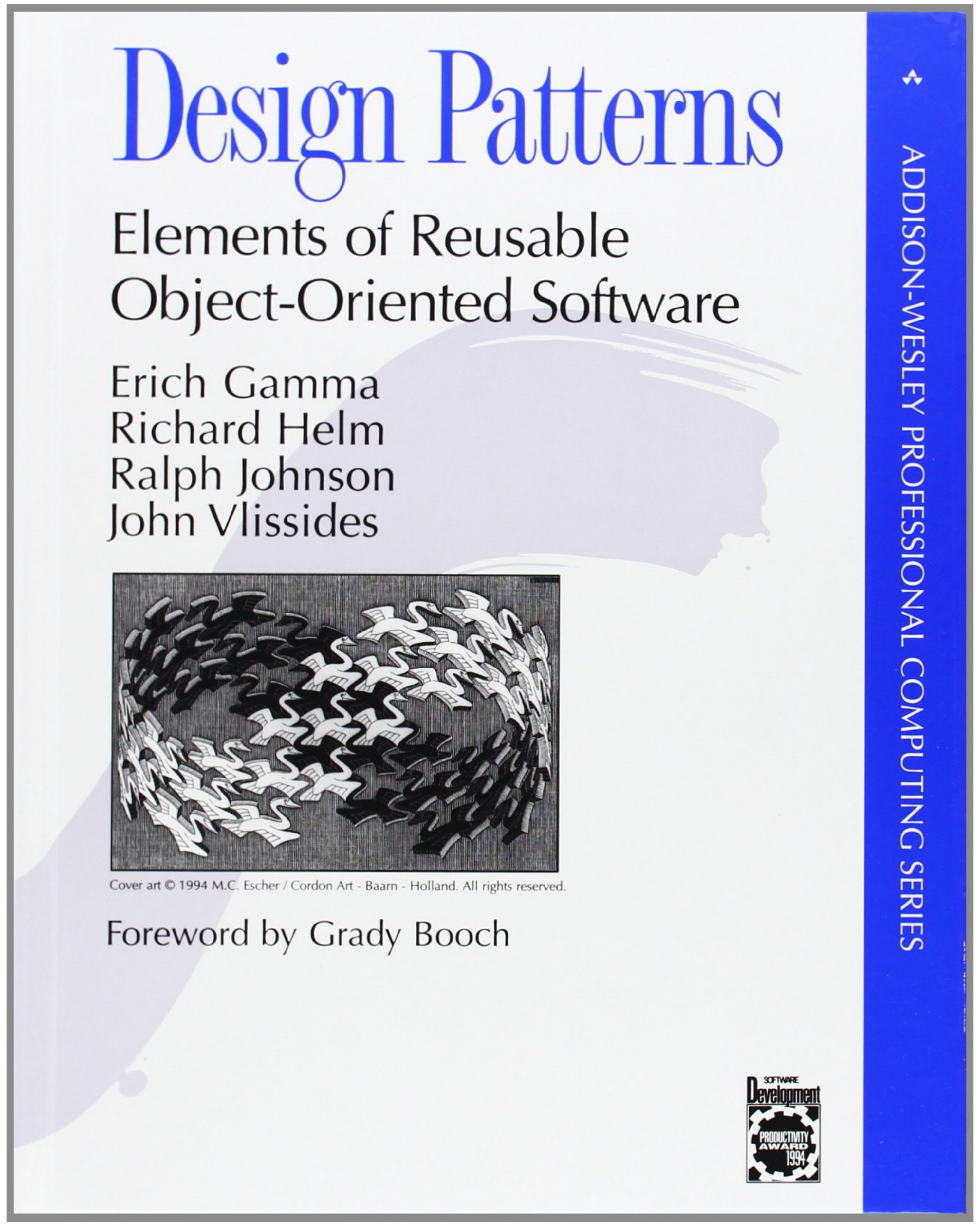
Based on `examples/12/Revised/*`

```
class Student : public Person {
  public:
    // ...
    virtual ~Student();
    // ...
};
```

```
class Person {
  public:
    // ...
    virtual ~Person();
    // ...
};
```

Correct destructor is now being called when using the base-class pointer to a `Student` instance

```
$ g++ -Wall -o StudentMemLeak StudentMemLeak.cpp {Student,GraduateStudent,Person}.cc
$ ./StudentMemLeak
Person(Susan) called
Student(Susan, 123456) called
student Susan currently enrolled in following courses:
algebra
physics
Art
Person(Paolo) called
Student(Paolo, 9856) called
I am Student Susan with id 123456
I am now enrolled in 3 courses.
I am Student Paolo with id 9856
I am now enrolled in 2 courses.
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Student() called for name:Paolo and id: 9856
~Person() called for Paolo
```



Strategy Pattern

Strategy Pattern: Overview

- The **strategy pattern** (a.k.a. **policy pattern**) is a behavioural software design pattern that enables selecting an algorithm at runtime

- Instead of implementing a single algorithm directly, the code receives runtime instructions as to which algorithm to use among a family of **interchangeable** algorithms
  - Each one is encapsulated as an object
  - The algorithm concretely used varies independently from client to client

- Deferring until runtime the decision about which algorithm to use allows the calling code to be more flexible and reusable

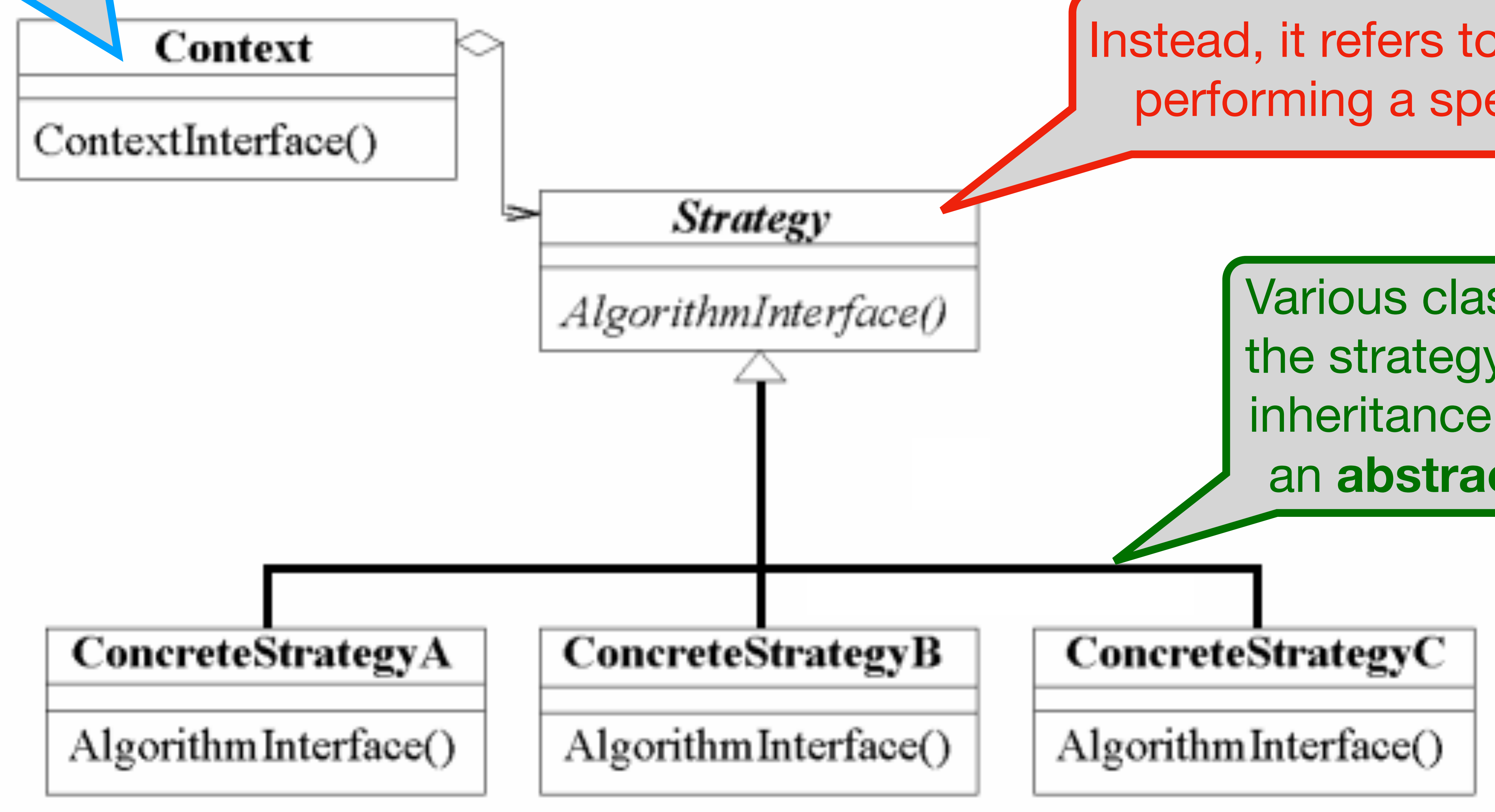
- Typically, the strategy pattern stores a reference to some code in a data structure and retrieves it

Strategy Pattern: Overview

Usually the client/application making use of the strategy: it does not implement an algorithm directly

Instead, it refers to an **interface** for performing a specific algorithm

Various classes implement the strategy interface in an inheritance structure (from an **abstract base class**)



Strategy Pattern Application: Integrating

Pick a concrete integration strategy among various implemented strategies (algorithms)
- midpoint or rectangles rule
- trapezoidal rule
- Simpson
- Monte-Carlo
- ...

$$\int_a^b f(x)dx \approx I$$

Algorithm interface
- We know we will have an integration interval
- We know we will have a function to integrate

Integration Interface

Based on `examples/12/StrategyPattern/Integrator.h`

```
class Integrator {

  public:
    Integrator() {
      integrand_ = 0;
    }
    void setIntegrand( double(*f)(double) ) {
        integrand_ = f;
    }
    double integrand(double x) const {
        return integrand_(x);
    }
    virtual double integrate(double xlo, double xhi) const = 0;

  private:
    double (*integrand_)(double);

};
```

- Using pointer to a function to integrate (provided by the user)
    - Has to be a standard single-value C function
- `integrate()` is pure virtual, so the integration interface is an abstract class
    - We cannot perform any calculations yet

A Concrete Integration Strategy

Based on `examples/12/StrategyPattern/MCIntegrator.h`

```
class MCIntegrator : public Integrator {

  public:
    MCIntegrator(int n=1000);
    void setNPoints( int n ) {
        npoints_ = n;
    }
    virtual double integrate(double xlo, double xhi) const;

  private:
    int npoints_;
    double uniform(double a, double b) const;

};
```

- Public inheritance from `Integrator`
- Constructor with default value
- `integrate` implemented in `MCIntegrator.cc` (no need for the user to see details)

Context 1: Integrating `Exp()`

Based on `examples/12/StrategyPattern/Context1.cpp`

```
MCIntegrator mcalgo(100);
mcalgo.setIntegrand(exp);

double a,b;
cout << "Program to integrate the exponential function over [a,b]" << endl;
cout << "a: ";
cin >> a;
cout << "b: ";
cin >> b;

double analyticalIntegral = mcalgo.integrand(b) - mcalgo.integrand(a);
cout << "analytical integral: " << analyticalIntegral << endl;

for(int n=10; n<1e8; n*=10) {
    mcalgo.setNPoints(n);
    double sum = mcalgo.integrate( a, b ); // numerical integral from a -> b
    cout << "# points: " << setw(10) << n
         << "\t Integral: " << setprecision(6) << sum
         << "\t residual: "
         << sum - analyticalIntegral
         << "\t fractional difference: " << setprecision(3)
         << 100*(sum - analyticalIntegral)/analyticalIntegral << " %"
         << endl;
}
```

- The exponential is a nice test case in which we know the result analytically
  - Provide comparisons to assess algorithm precision, convergence, etc.

Context 1: Output

Based on `examples/12/StrategyPattern/Context1.cpp`

```
$ g++ -c MCIntegrator.cc
$ g++ -Wall -o Context1 Context1.cpp MCIntegrator.o
$ ./Context1
Program to integrate the exponential function over [a,b]
a: 1
b: 10
analytical integral: 22023.7
# points:         10   Integral: 6968.75     residual: -15055      fractional difference: -68.4 %
# points:        100   Integral: 18843.9     residual: -3179.81    fractional difference: -14.4 %
# points:       1000   Integral: 22762.8     residual: 739.064     fractional difference: 3.36 %
# points:      10000   Integral: 21768       residual: -255.767    fractional difference: -1.16 %
# points:     100000   Integral: 22065.1     residual: 41.3194     fractional difference: 0.188 %
# points:    1000000   Integral: 22047.1     residual: 23.3573     fractional difference: 0.106 %
# points:   10000000   Integral: 22013.6     residual: -10.1308    fractional difference: -0.046 %
```

Context 2: Integrating `sin()`

Based on `examples/12/StrategyPattern/Context2.cpp`

```
MCIntegrator mcalgo(100);
mcalgo.setIntegrand(sin);

double a,b;
cout << "Program to integrate the sinus function over  [a x Pi, b x Pi]" << endl;
cout << "a: ";
cin >> a;
cout << "b: ";
cin >> b;

double analyticalIntegral = -cos(b*M_PI) - (-cos(a*M_PI));
cout << "analytical integral: " << analyticalIntegral << endl;

for(int n= 10; n< 1e8; n *=10 ) {
    mcalgo.setNPoints(n);
    double sum = mcalgo.integrate( a*M_PI, b*M_PI ); // integral from a -> b
    cout << "# points: " << setw(10) << n
         << "\t Integral: " << setprecision(6) << sum
         << "\t residual: "
         << sum - analyticalIntegral
         << "\t fractional difference: " << setprecision(3)
         << 100*(sum - analyticalIntegral)/analyticalIntegral << " %"
         << endl;
}
```

- Changing the integrand to sin function and scaling integration interval with **π** (`M_PI` from `cmath`)
  - Comparison to analytical solution is again possible

Integration Context 2: Output 1

Based on `examples/12/StrategyPattern/Context2.cpp`

```
$ g++ -Wall -o Context2 Context2.cpp MCIntegrator.o
$ ./Context2
Program to integrate the sinus function over  [a x Pi, b x Pi]
a: 1
b: 10
analytical integral: -2
# points:        10   Integral: -3.3557     residual: -1.3557     fractional difference: 67.8 %
# points:       100   Integral: -0.838648   residual: 1.16135     fractional difference: -58.1 %
# points:      1000   Integral: -1.1051     residual: 0.894905    fractional difference: -44.7 %
# points:     10000   Integral: -2.13556    residual: -0.135559   fractional difference: 6.78 %
# points:    100000   Integral: -1.89243    residual: 0.107569    fractional difference: -5.38 %
# points:   1000000   Integral: -2.0053     residual: -0.00530445 fractional difference: 0.265 %
# points:  10000000   Integral: -2.01286    residual: -0.0128573  fractional difference: 0.643 %
```

No need to recompile the concrete strategies when context changes!

Integration Context 2: Output 2

Based on `examples/12/StrategyPattern/Context2.cpp`

```
$ ./Context2
Program to integrate the sinus function over  [a x Pi, b x Pi]
a: 1
b: 9
analytical integral: 0
# points:         10   Integral: -3.75307    residual: -3.75307    fractional difference: -inf %
# points:        100   Integral: -2.98412    residual: -2.98412    fractional difference: -inf %
# points:       1000   Integral: 0.490921    residual: 0.490921    fractional difference: inf %
# points:      10000   Integral: -0.0772875  residual: -0.0772875  fractional difference: -inf %
# points:     100000   Integral: 0.0251424   residual: 0.0251424   fractional difference: inf %
# points:    1000000   Integral: 0.0278078   residual: 0.0278078   fractional difference: inf %
# points:   10000000   Integral: -0.00500983 residual: -0.00500983 fractional difference: -inf %
```

No need to recompile the context
if we change integration interval

What happened here?

Exercise

1. Write the missing classes to implement more integration methods

2. Study the difference among the integral values estimated by the various methods

3. Plot the integration errors as a function of number of points (or interval divisions): `gnuplot`, `ROOT (TH1F)`, `Python`...

Integration Interface for Custom Functions

Based on `examples/12/StrategyPattern/CustomIntegrator.h`

```
class Integrator {

  public:
    Integrator() {
      integrand_ = 0;
    }
    void setIntegrand( double(*f)(double) ) {
        integrand_ = f;
    }
    double integrand(double x) const {
        return integrand_(x);
    }
    virtual double integrate(double xlo, double xhi) const = 0;

  private:
    double (*integrand_)(double);

};
```

Integration Interface for Custom Functions

Based on `examples/12/StrategyPattern/CustomIntegrator.h`

```
#include "Function.h"

class CustomIntegrator {

  public:
    CustomIntegrator() {
      integrand_ = 0;
    }
    void setIntegrand( Function* f ) {
        integrand_ = f;
    }
    double integrand(double x) const {
        return integrand_->value(x);
    }

    Function* integrand() const {
        return integrand_;
    }
    virtual double integrate(double xlo, double xhi) const = 0;

  private:
    Function* integrand_;

};
```

- Integrand function treated via a pointer to an instance of the `Function` abstract class
  - The user will have to comply to the rules set by the `Function` abstract class and provide a concrete function
- `integrate()` is again pure virtual
  - This was removed from `Function` compared to the previous lecture: leave it to the strategy pattern to handle integration

Revisiting the Concrete Integration Strategy

Based on `examples/12/StrategyPattern/CustomMCIntegrator.h`

```
class MCIntegrator : public Integrator {

  public:
    MCIntegrator(int n=1000);
    void setNPoints( int n ) {
        npoints_ = n;
    }
    virtual double integrate(double xlo, double xhi) const;

  private:
    int npoints_;
    double uniform(double a, double b) const;

};
```

Revisiting the Concrete Integration Strategy

Based on `examples/12/StrategyPattern/CustomMCIntegrator.h`

```
class CustomMCIntegrator : public CustomIntegrator {

  public:
    CustomMCIntegrator(int n=1000);
    void setNPoints( int n ) {
        npoints_ = n;
    }
    virtual double integrate(double xlo, double xhi) const;

  private:
    int npoints_;
    double uniform(double a, double b) const;

};
```

- Public inheritance from `CustomIntegrator`
  - Upgrade of interface is trivial
  - Upgrade of implementation is too

Context 3: Integrating the Concrete `Function` `Gauss`

Based on `examples/12/StrategyPattern/Gauss.h`

```
#include <string>
#include "Function.h"

class Gauss : public Function {
  public:
    Gauss(const std::string& name, double mean, double width);

    virtual double value(double x) const;
    virtual void print() const;

  private:
    double mean_;
    double width_;
};
```

Context 3: Integrating the Concrete `Function Gauss`

Based on `examples/12/StrategyPattern/Context3.cpp`

```
CustomMCIntegrator cinteg = CustomMCIntegrator();
Function*  g1 = new Gauss("g1", 0., 1.);
cinteg.setIntegrand( g1 );

double a,b;
cout << "Program to integrate the Gaussian function over [a x sigma, b x sigma]" << endl;
cout << "a: ";
cin >> a;
cout << "b: ";
cin >> b;

for(int n= 10; n< 1e8; n *=10 ) {
    cinteg.setNPoints(n);
    double sum = cinteg.integrate( a, b ); // integral from a -> b
    cout << "# points: " << setw(10) << n
         << "\t Integral: " << setprecision(6) << sum
         << endl;
}
```

Not providing analytical comparison

We could provide a test integrating from 0 to `L`>>1: this would depend on the number of integration points (`n`) **and** the upper integration limit (`L`) going to infinity

Context 3: 1-sigma Output

Based on `examples/12/StrategyPattern/Context3.cpp`

```
$ g++ -c CustomMCIntegrator.cc
$ g++ -c Function.cc
$ g++ -c Gauss.cc
$ g++ -Wall -o Context3 Context3.cpp CustomMCIntegrator.o Function.o Gauss.o
$ ./Context3
Program to integrate the Gaussian function over [a x sigma, b x sigma]
a: -1
b: 1
# points:         10   Integral: 0.678152
# points:        100   Integral: 0.673835
# points:       1000   Integral: 0.686308
# points:      10000   Integral: 0.682816
# points:     100000   Integral: 0.682718
# points:    1000000   Integral: 0.682672
# points:   10000000   Integral: 0.682664
```

Context 3: 2-sigma Output

Based on `examples/12/StrategyPattern/Context3.cpp`

```
$ ./Context3
Program to integrate the Gaussian function over [a x sigma, b x sigma]
a: -2
b: 2
# points:          10   Integral: 1.12849
# points:         100   Integral: 0.973729
# points:        1000   Integral: 0.957521
# points:       10000   Integral: 0.955094
# points:      100000   Integral: 0.953609
# points:     1000000   Integral: 0.954173
# points:    10000000   Integral: 0.954674
```

Context 3: 3-sigma Output

Based on `examples/12/StrategyPattern/Context3.cpp`

```
$ ./Context3
Program to integrate the Gaussian function over [a x sigma, b x sigma]
a: -3
b: 3
# points:          10   Integral: 1.40873
# points:         100   Integral: 0.972365
# points:        1000   Integral: 0.977208
# points:       10000   Integral: 1.00923
# points:      100000   Integral: 0.995503
# points:     1000000   Integral: 0.999204
# points:    10000000   Integral: 0.997157
```

Exercise

1. Add new methods to `Gauss` to modify its parameters after it has been created

2. Add a few more `Function` derivate classes, e.g. line, exponential, famous polynomials, …

3. Implement more integration methods inheriting from `CustomIntegrator`

4. Study the difference among the integral values estimated by the various methods, for a given `Function`

5. Make plots of the differences with `gnuplot`, `ROOT (TH1F)`, `Python…`