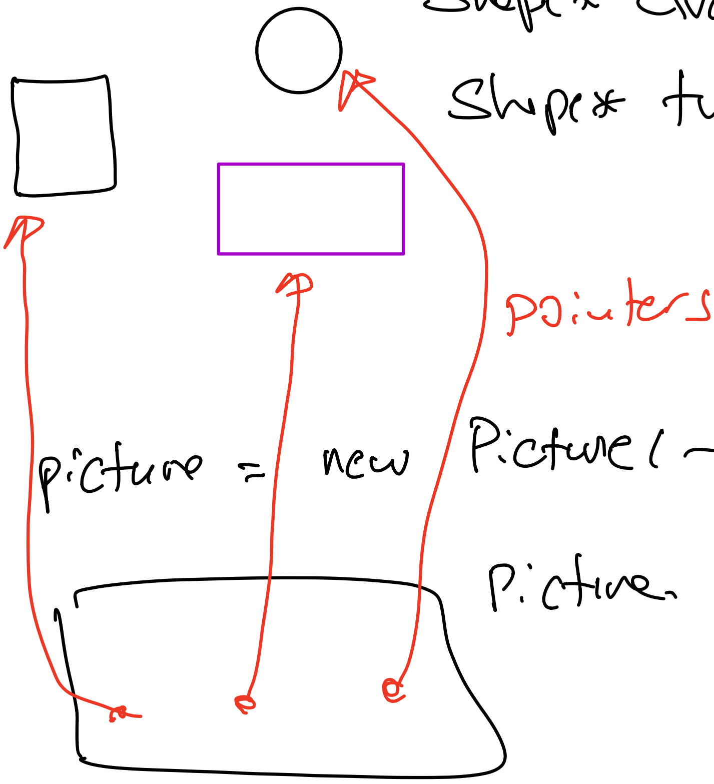
Composite

Shape* rect = new Rectangle(--)

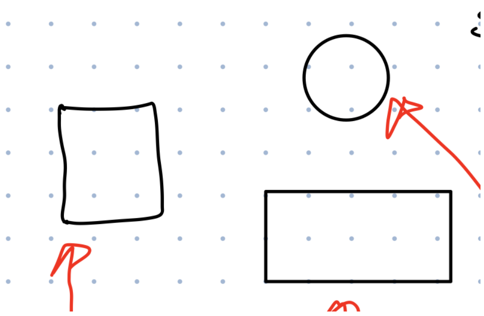
Shape* Circle = new Circle(-)

Shape* tri = new Triangle(-)



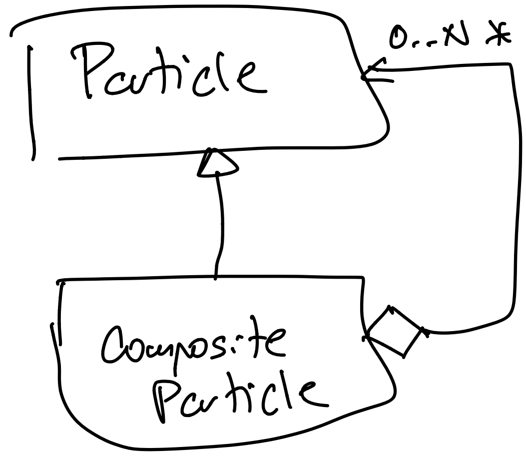
Shape* picture = new Picture(—)

IF Picture owned Shapes



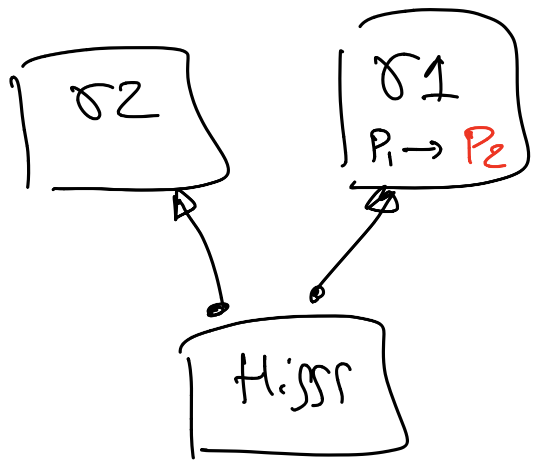
Composite Pattern with Particles

$H \longrightarrow \gamma\gamma$ $\quad\quad \underline{P}_H = \underline{P}_{\gamma_1} + \underline{P}_{\gamma_2}$ $\quad$ 4-momentum




```
Particle* P1 = new Particle(—)
Particle* P2 = new Particle(—)
Particle* H = new CompositeParticle(—)
```

```
H->add(&P1)
H->add(&P2)
```



H->mass() m_1

$m = \sqrt{|\underline{P}_H|^2}$

$\vec{P}_{\gamma_1} = (100, 20, 20)$

$\vec{P}_{\gamma_2} = (-100, -20, +50)$

$\vec{P}_{\gamma_2} = (-150, -30, +20)$

$\underline{P}_H = \underline{P}_{\gamma_1} + \underline{P}_{\gamma_2}$

```
P1->setMomentum(vector3D(-150, -30, +20))
```

H->mass() m_2

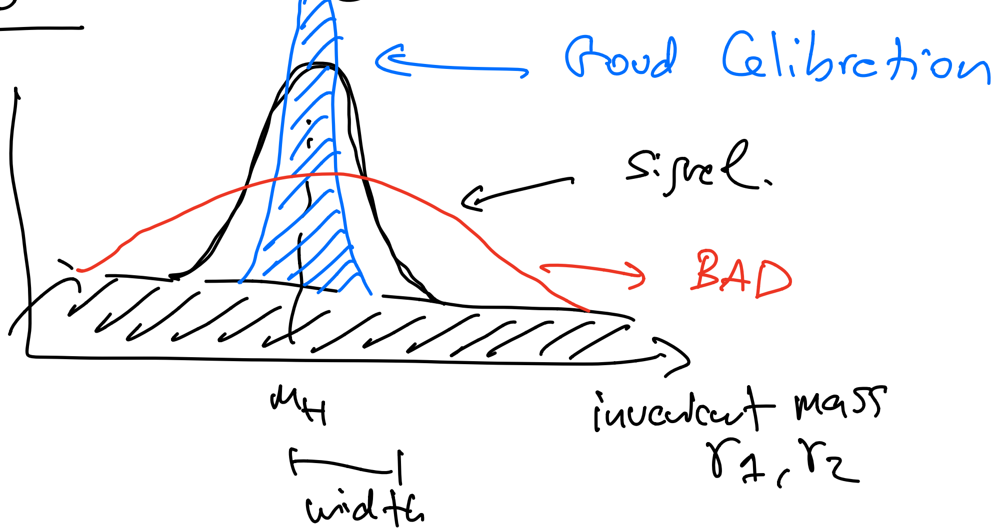
Day 1

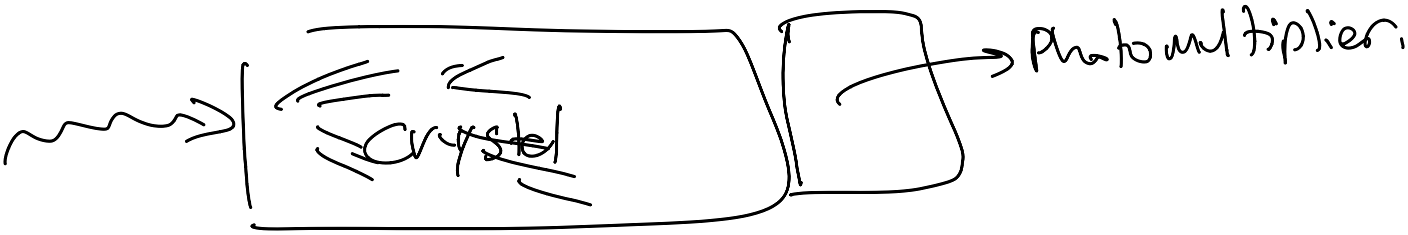
γ_1
γ_2

calibrate detector - -

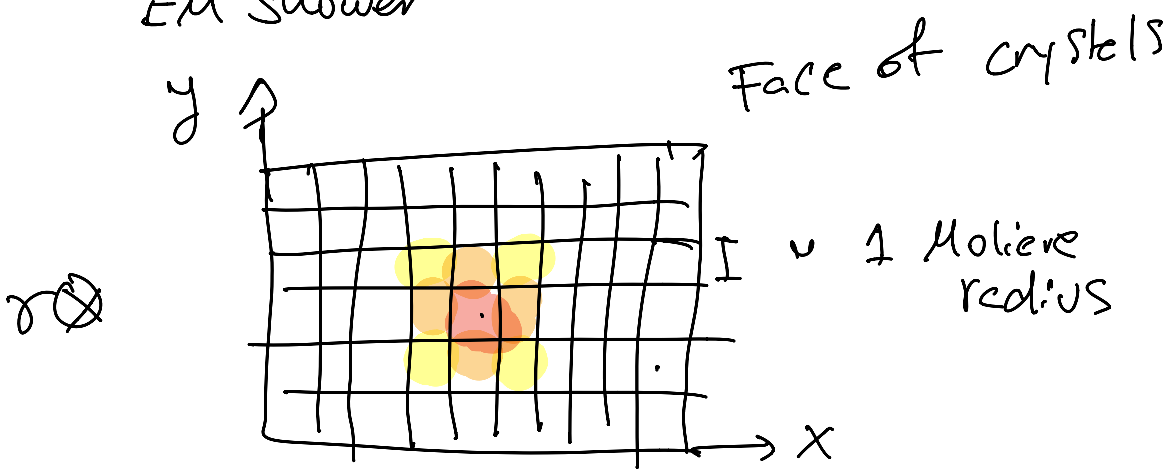
Day 20

γ_1
γ_2





EM shower



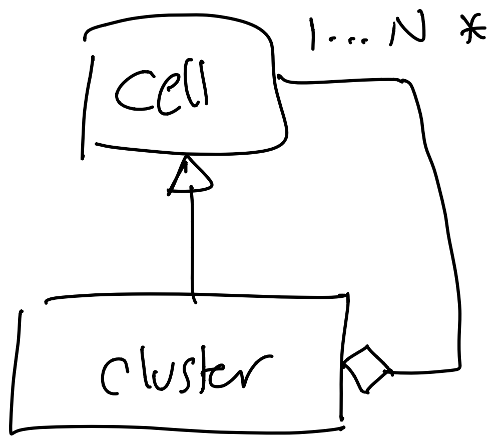
$$E_\gamma = \sum_{i=1}^{N} E_{crystal_i}$$

Cluster of energy in EM Calorimeter

1 cell: energy()
x()
y()

1 cluster: energy()
x()
y()

Cluster: $X = \dfrac{\sum_i E_i \cdot x_i}{\sum E_i}$



Composite Pattern.

$H \longrightarrow Z \; Z^*$

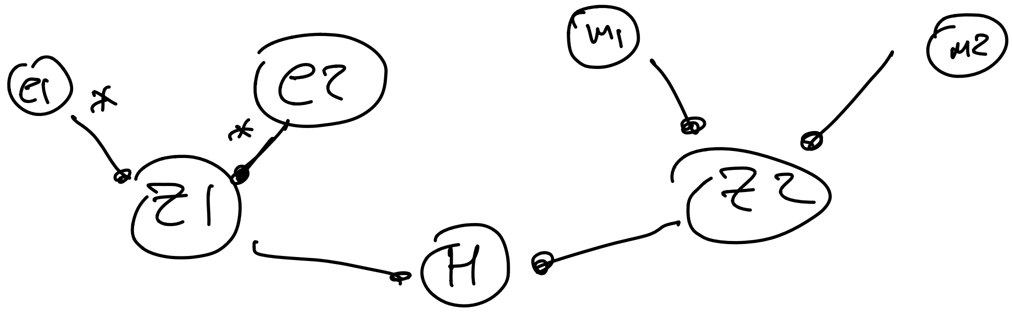
$Z^* \to e^+ e^-$

$Z \to \mu^+ \mu^-$

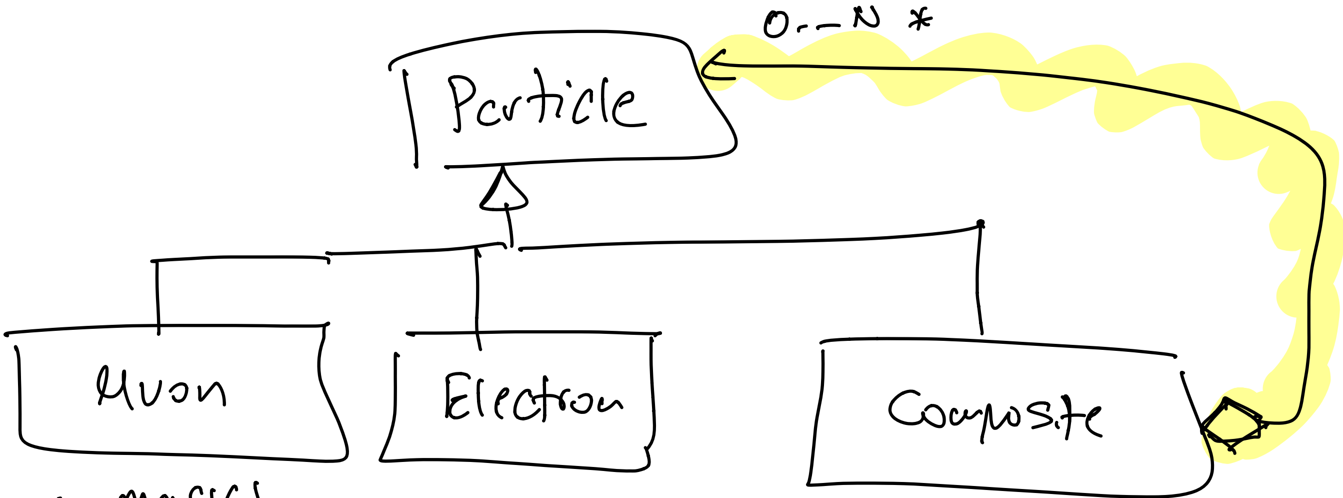
$e^\pm$: EM Calorimeter

$\mu^\pm$: muon Chambers.

```
Particle* Z1 = new Composite(--)
Particle* e1 = new Electorn(--)
Particle* e2 = new Electron(--)
Z1 -> add(&e1)
Z1 -> add(&e2)
Particle* Z2 = new Composite(--)
Particle* m1 = new Muon(--)
Particle* m2 = new Muon(--)
Z2 -> add(&m1) ; Z2 -> add(&m2)
```



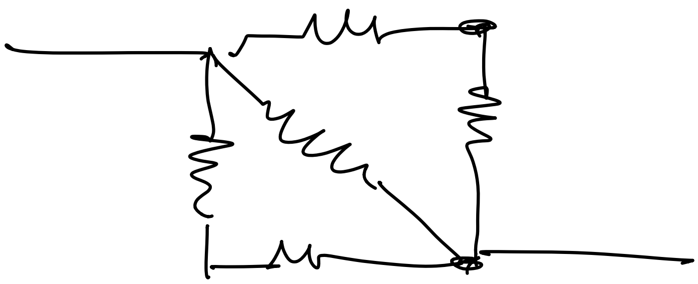

```
Particle* H = new Composite(--)
H -> add(&Z1)
H -> add(&Z2)

H -> mass()
```



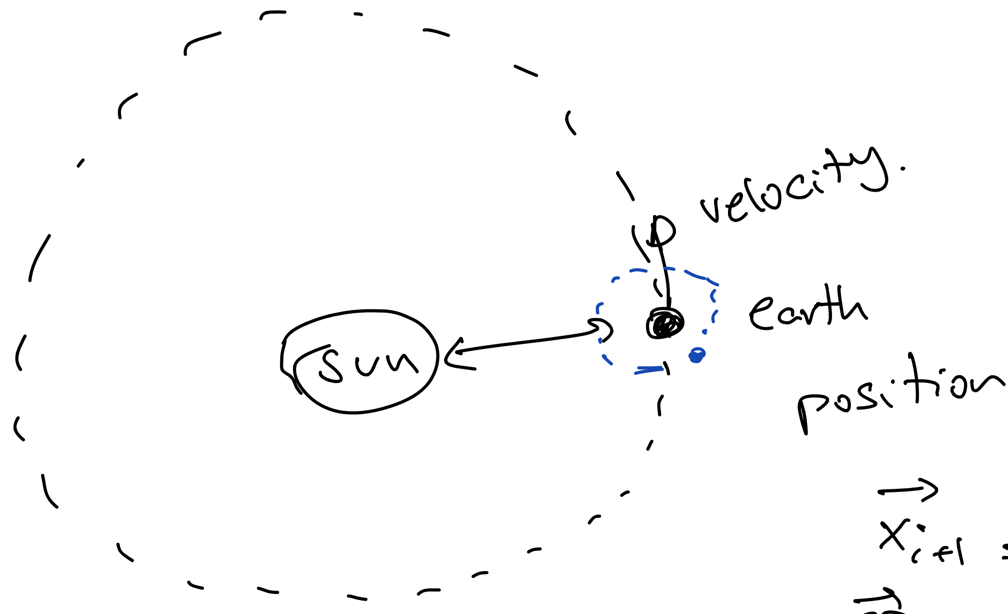
+ mass() . —

+ 4momentum() —

+ add(--)



Solar System



$$\vec{x}_{i+1} = \vec{x}_i + \vec{v}_i \cdot \Delta t$$

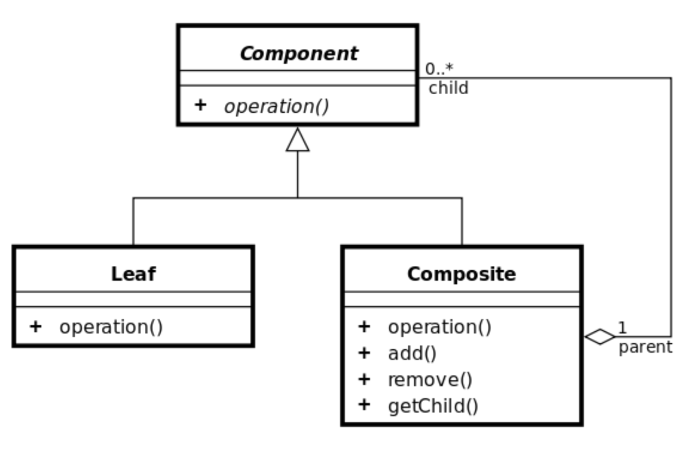
$$\vec{v}_{i+1} = \vec{v}_i + \vec{a}_i \cdot \Delta t$$

$$\vec{F} = G \frac{M_\odot M_{Earth}}{r^3} \vec{r}$$

$$\vec{a} = \frac{\vec{F}}{M_{earth}}$$

Composite Pattern (Composition)

Composition is a design patten to treat a group of objects the same way as a single object. It is an example of a part-whole hierarchy where simple objects are composed into a composite object. Clients treat simple and composite objects share the same interface.



The UML diagram for the composite pattern is

A simple example of composition is with graphical objects, e.g. square, circle, rectangle, as basic objects and a picture as a composed object containing a list of simple of objects.

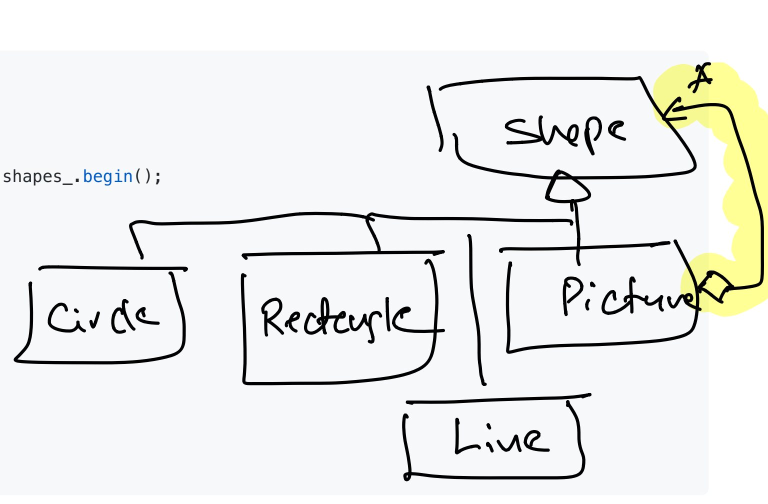
Simple objects inherit from Shape

```
class Shape {
public:
  virtual void draw() const = 0;
  virtual void move() = 0;
  virtual ~Shape() {}
};
```

pure virtual

The composite objects such as Picture contain a list of pointers (not copies) of simple objects in order to allow us to take advantage of polymorphic behavior of different objects.

```
class Picture : public Shape {
public:
  void draw() const {
    // call  draw() for each shape in the list
    for( std::list<Shape*>::const_iterator it = shapes_.begin();
      it != shapes_.end(); it++ )  {
          it->draw();
    }
  }

  void add(Shape *s) {
    shapes_.push_back(s);
  }

private:
  std::list<Shape*> shapes_;
};
```



Now we can define a bunch of specific shapes

```
class Rectangle : public Shape {
public:
  Rectangle(const std::string& name) : Shape(name) {}
  virtual void draw() const {
      std::cout << "calling Rectangle::draw() for " << name() << std::endl;
  }
};

class Circle : public Shape {
public:
  Circle(const std::string& name) : Shape(name) {}
  virtual void draw() const {
      std::cout << "calling Circle::draw() for " << name() << std::endl;
  }
};

class Line : public Shape {
public:
  Line(const std::string& name) : Shape(name) {}
  virtual void draw() const {
      std::cout << "calling Line::draw() for " << name() << std::endl;
  }
};
```

and now we can test the application appShapes.cc

```
int main() {

  Rectangle rect1("r1");
  Circle cir1("cir1");

  Rectangle rect2("r2");
  Circle cir2("cir2");
  Line l2("l2");

  Picture pic1("pic1");
  pic1.add( &rect2);
  pic1.add( &cir2);
  pic1.add( &l2);

  rect1.draw();
  cir1.draw();

  pic1.draw();

  return 0;
}
```

we can now link and run the example

```
$ g++ -o /tmp/appShapes appShapes.cc
$ /tmp/appShapes
calling Rectangle::draw() for r1
calling Circle::draw() for cir1
drawing picture pic1
calling Rectangle::draw() for r2
calling Circle::draw() for cir2
calling Line::draw() for l2
```

Note that in this example we only use a base class for simple shapes and a composite class. So our `Leaf` and `Component` classes are the same.

Exercise

1. implement proper constructors to take as argument the information needed to create each type of objects
2. implement the `move()` method

Simulation of solar system with composite pattern

This example is inspired by work described by Giovanni Organtini in the freely available appendix to the book Programmazione Scientifica (by Barone, Marinari, Organtini, Ricci-Tersenghi). This book is the reference book for the course **Laboratorio di Calcolo** and **Laboratorio di Fisica Comutazionale** in the Laurea Triennale in Fisica in Rome.

The idea is to simulate the motion of the earth around the sun with a simple class representing a generic celestial body subject to the gravitation force. The Euler method is used to integrate the equation of motion. You can use the Strategy pattern to implement a diffrent integration scheme, eg. with the Runge-Kutta method.

The composite pattern is used to extend the model to simulate the motion of the moon around the Earth. The same model can then be used to add all planets of the solar system and their satellites.

Note that with the addition of many bodies, the simple Euler method wiull become insufficient and better and more precise computation is needed.

However, all classes continue to work and you only need to update or improve the `move()` method or even better use the Strategy pattern.

generic celestial body

The base class `Body` (Body.h, Body.cc) is abstract and mainly meant to define the interface for all celestial bodies.

All methods should be usable by both simple and composite objects.

```
class Body {
  public:
    Body(const std::string& name) { name_ = name; }
    virtual void move(const Vector3D& Force, double dt) = 0;
    virtual double mass() const = 0;
    virtual Vector3D position() const = 0;
    virtual Vector3D velocity() const = 0;
    virtual void setPosition(const Vector3D& p) = 0;
    virtual void setVelocity(const Vector3D& v) = 0;
    virtual void translate(const Vector3D& dr) = 0;
    virtual void addVelocity(const Vector3D& dv) = 0;

    virtual Vector3D forceOn(const Body* obj) const = 0;
    virtual Vector3D forceFrom(const Body* source) const = 0;

    virtual std::string name() const {
        return name_;
    }

    virtual void print() const {
        std::cout << "class Body with name_: " << name_ << std::endl;
    }

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

Simple body

First we implement the `SimpleBody` (SimpleBody.h, SimpleBody.cc) class to simulate the motion of a body, both planets or satellites.

```
class SimpleBody : public Body {
  public:
    SimpleBody(const std::string& name, const double mass, const Vector3D& x0=0);
    virtual void move(const Vector3D& F, double dt);
    virtual double mass() const { return mass_;}
    virtual Vector3D position() const { return pos_; }
    Vector3D velocity() const { return vel_; }
    virtual void setVelocity(const Vector3D& v) { vel_ = v; };
    virtual void setPosition(const Vector3D& pos) {pos_ = pos; };
    virtual void translate(const Vector3D& dr) { pos_ += dr; }
    virtual void addVelocity(const Vector3D& dv) { vel_ += dv; }
    virtual Vector3D forceOn(const Body* obj) const;
    virtual Vector3D forceFrom(const Body* source) const;
    virtual void print() const;

  private:
    Vector3D pos_;
    Vector3D vel_;
    double mass_;
};
```

Note how all methods from `Body` are now implemented. There are no additional methods, but the class is defined by its new data members: the mass, the position, and the velocity.

We are using the Vector3D class to do vector calculations for position and velocity.

```
class Vector3D {
  public:
    Vector3D(const double& x=0, const double& y=0, const double& z=0) ;


    const Vector3D& operator=(const Vector3D& rhs);
    const Vector3D& operator+=(const Vector3D& rhs);

    const Vector3D operator+(const Vector3D& rhs) const;
    const Vector3D operator-(const Vector3D& rhs) const;

    const Vector3D& operator=(const double& a); // set all components  = a

    const Vector3D operator/(const double a) const ; // vector / scalar

    const Vector3D operator*(const double a) const; // scalar x double

    friend const Vector3D operator*(const double a, const Vector3D& rhs); // scalar x vector

    friend  std::ostream& operator<<(std::ostream& os, const Vector3D& rhs);

    double mod() const;
    double distance(const Vector3D& r0) const;

  private:
    double x_[3];

};
```

NB: the implementation (.cc file) of `Vector3D` is not provided here. You should by now have already this class implemented. If not, take the header file and implement all the functions as an exercise.

The key of the code is in the `move()` method which implements the Euler method.

```
void SimpleBody::move(const Vector3D& F, double dt) {
  pos_ += vel_ * dt;
  Vector3D acc = F/mass_;
  vel_ += acc * dt;

}
```

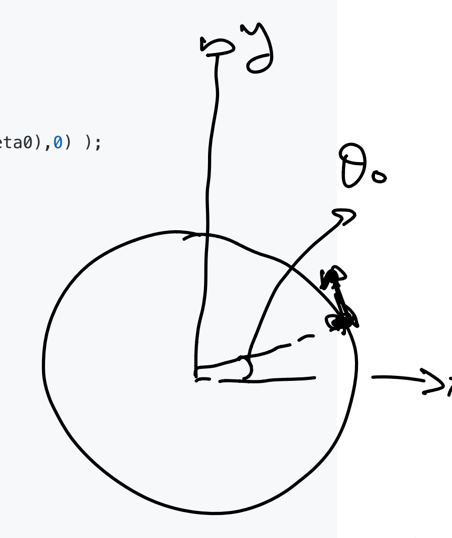
$\vec{x}_{i+1} = \vec{x}_i + \vec{v}_i \Delta t$

$\vec{v}_{i+1} = \vec{v}_i + \vec{a} \cdot \Delta t$

where you provide the force on the object and move it accordingly.

We are now ready to test our simulation in appSimple.cc

```
int main() {


  double sunMass = 2.e30; // kg
  SimpleBody sun("sun", sunMass, Vector3D(0,0,0) );

  double theta0 = M_PI/10; // random position along the orbit
  double orbitalVel = 30.e3; // 30 km/s
  double astrUnit = 150.e9; // 150 x 10^6 km
  double earthMass = 6.e24; // kg
  SimpleBody earth("earth", earthMass, Vector3D(astrUnit*cos(theta0),astrUnit*sin(theta0),0) );
  earth.setVelocity( Vector3D(-orbitalVel*sin(theta0), orbitalVel*cos(theta0), 0) );

  sun.print();
  earth.print();

  const double G = 6.673e-11; // Newton's constant

  cout << std::setprecision(4) << std::setw(10);

  // estimate position every minute
  int days(380), hours(24), mins(60);
  for(int i=0; i<= days*hours*mins; ++i) {
    // compute force
    Vector3D dr = sun.position() - earth.position();
    Vector3D force =  G * sun.mass() * earth.mass() * dr / pow(dr.mod(),3);
    //force = force/(pow(dr.mod(),3));

    if( i%(hours*mins) == 0 ) {
      cout << " ---- Day " << i/(hours*mins) << " ----- " << endl;
      cout << "earth position: " << earth.position()
           << endl;

    }
    earth.move(force, 1);
  }
  return 0;

}
```



class Constants?
public:
static const double
G = – –
?
constants::G

You can also write a method to compute this force always correctly

```
Vector3D SimpleBody::forceOn(const Body* obj) const {
  const  double Grav_Const = 6.673e-11; // Newton's constant
  Vector3D dr = position() - obj->position();
  Vector3D force =  Grav_Const * mass() * obj->mass() * dr / pow(dr.mod(),3);
  return force;
}

Vector3D SimpleBody::forceFrom(const Body* source) const {
  const  double Grav_Const = 6.673e-11; // Newton's constant
  Vector3D dr = source->position() - position();
  Vector3D force =  Grav_Const * source->mass() * mass() * dr / pow(dr.mod(),3);
  return force;
}
```

Link and run the executable

```
$ g++ -o /tmp/appSimple appSimple.cc SimpleBody.cc Vector3D.cc
$ /tmp/appSimple
===== beginning of simulation
===== class: SimpleBody  name: sun       mass: 2e+30 kg =====
current position (0 , 0 , 0)     distance from origin: 0 m
current velocity (0 , 0 , 0)     0 m/s
===== class: SimpleBody  name: earth     mass: 6e+24 kg =====
current position (1.42658e+11 , 4.63525e+10 , 0)         distance from origin: 1.5e+11 m
current velocity (-9270.51 , 28531.7 , 0)        30000 m/s
 ---- Day 0 -----
earth position: (1.427e+11 , 4.635e+10 , 0)
 ---- Day 1 -----
earth position: (1.426e+11 , 4.639e+10 , 0)
 .....

 ---- Day 379 -----
earth position: (1.368e+11 , 6.162e+10 , 0)
---- Day 380 -----
earth position: (1.368e+11 , 6.166e+10 , 0)
===== End of simulation
===== class: SimpleBody  name: earth     mass: 6e+24 kg =====
current position (1.368e+11 , 6.166e+10 , 0)     distance from origin: 1.5e+11 m
current velocity (-1.23e+04 , 2.736e+04 , 0)     3e+04 m/s
```

Composite Body

We now define a `CompositeSystem` (CompositeSystem.h, CompositeSystem.cc) class to represent an object that contains a list of pointers to `Body`. This can be for example the sun-earth system, the earth-moon, system, or the sun-(earth-moon) system. It can also be used to simulate the complete solar system.

```
class CompositeSystem : public Body {
  public:
    CompositeSystem(const std::string& name, Body* center=0);
    virtual double mass() const;
    virtual Vector3D position() const;
    virtual Vector3D velocity() const;
    virtual void move(const Vector3D& F, double dt);
    virtual void print() const;
    virtual void setPosition(const Vector3D& pos);
    virtual void setVelocity(const Vector3D& vel);
    virtual void translate(const Vector3D& dr);
    virtual void addVelocity(const Vector3D& dv);
    virtual Vector3D forceOn(const Body* obj) const;
    virtual Vector3D forceFrom(const Body* source) const;

    virtual void add(Body* b, const Vector3D& p0=Vector3D(0,0,0), const Vector3D& v0=Vector3D(0,0,0));
    virtual const std::list<Body*> bodies() const { return bodies_; }

    void evolve(const double& dt);
/*
    virtual const get(const std::string& name) const;
    virtual const remove(const std::string& name) const;
    virtual const remove(const Body& b) const;
*/

  private:
    std::list<Body*> bodies_;
    Body* center_;
};
```

The example is appComposite.cc

```
int main() {



  double theta0 = M_PI/10; // random position along the orbit
  double earthVel = 30.e3; // 30 km/s
  double astrUnit = 150.e9; // 150 x 10^6 km
  double earthMass = 6.e24; // kg
  SimpleBody earth("earth", earthMass, Vector3D(0,0,0) );
  earth.setVelocity( Vector3D(0, 0, 0) );

  double moonVel = 1.e3; // 1 km/s
  double moonDist = 384399.e3; // 384 399 km
  double moonMass = 7.e22; // kg
  SimpleBody moon("moon", moonMass, Vector3D(moonDist*cos(theta0),moonDist*sin(theta0),0) );
  moon.setVelocity( Vector3D(-moonVel*sin(theta0), moonVel*cos(theta0), 0) );

  CompositeSystem earthSystem("earth+moon", &earth);
  earthSystem.add(&moon);


  double sunMass = 2.e30; // kg
  SimpleBody sun("sun", sunMass, Vector3D(0,0,0) );


  CompositeSystem solarSystem("solarSystem", &sun);
  solarSystem.add( &earthSystem,
                   Vector3D(astrUnit*cos(theta0),astrUnit*sin(theta0),0),
                   Vector3D(-earthVel*sin(theta0), earthVel*cos(theta0), 0) );

  solarSystem.print();


  cout << std::setprecision(4) << std::setw(10);

  // estimate position every minute
  int days(600), hours(24), mins(60);
  for(int i=0; i<= days*hours*mins; ++i) {

    if( i%(hours*mins*10) == 0 ) {
      cout << " ---- Day " << i/(hours*mins) << " ----- " << endl;
      cout << "earth position: " << earth.position()
           << endl;
    }
    solarSystem.evolve(60);
  }
  solarSystem.print();

  return 0;
```