University of St Andrews | FOUNDED 1413

Object-oriented programming

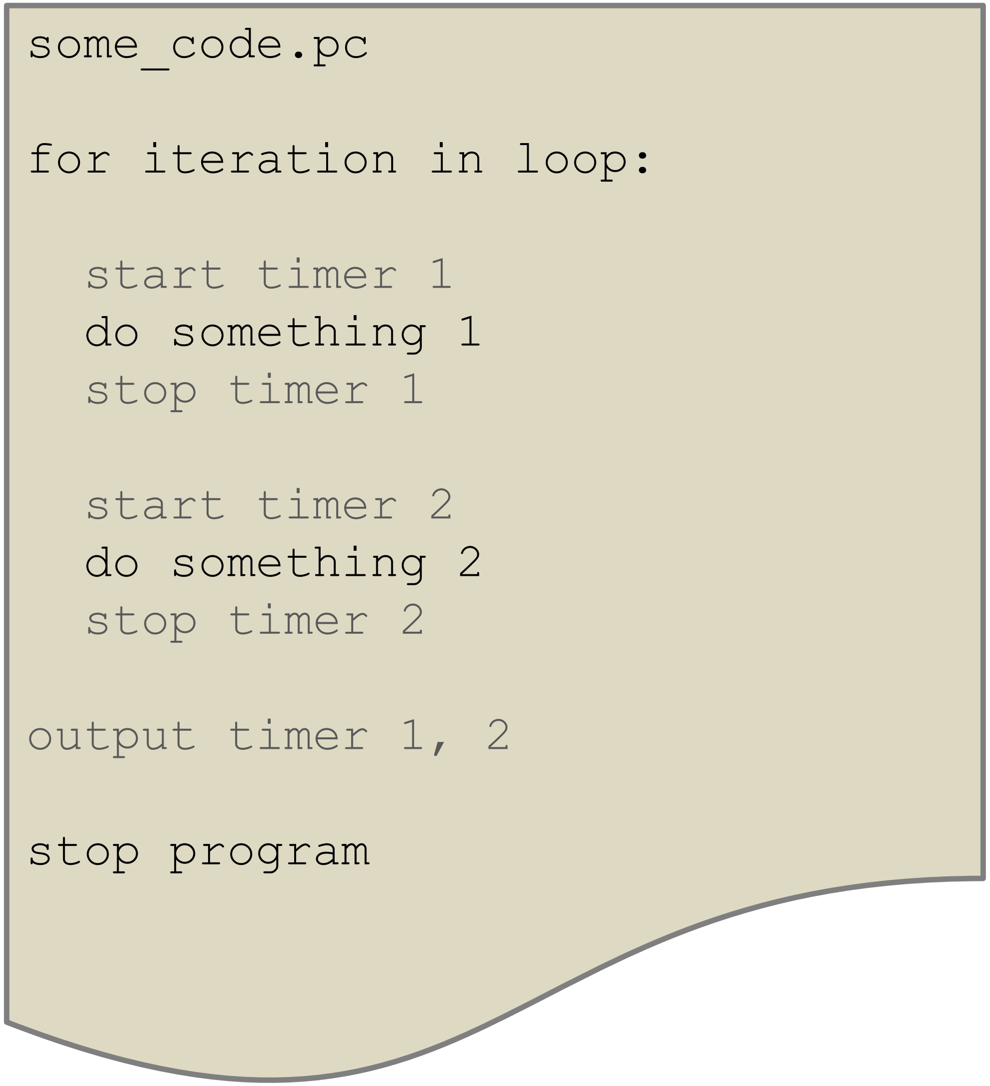
Bert Vandenbroucke

bv7@st-andrews.ac.uk

A simple use case: timer

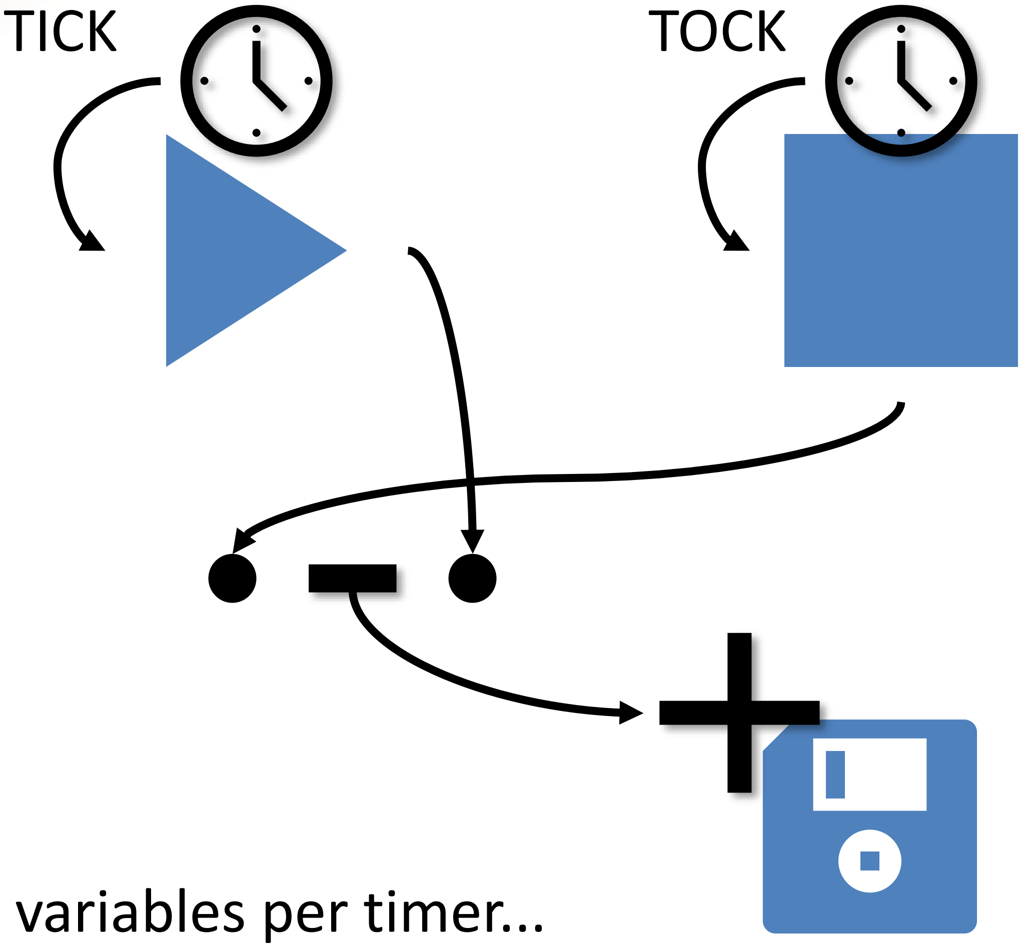
```
some_code.pc

for iteration in loop:


  do something 1



  do something 2




stop program
```

A simple use case: timer



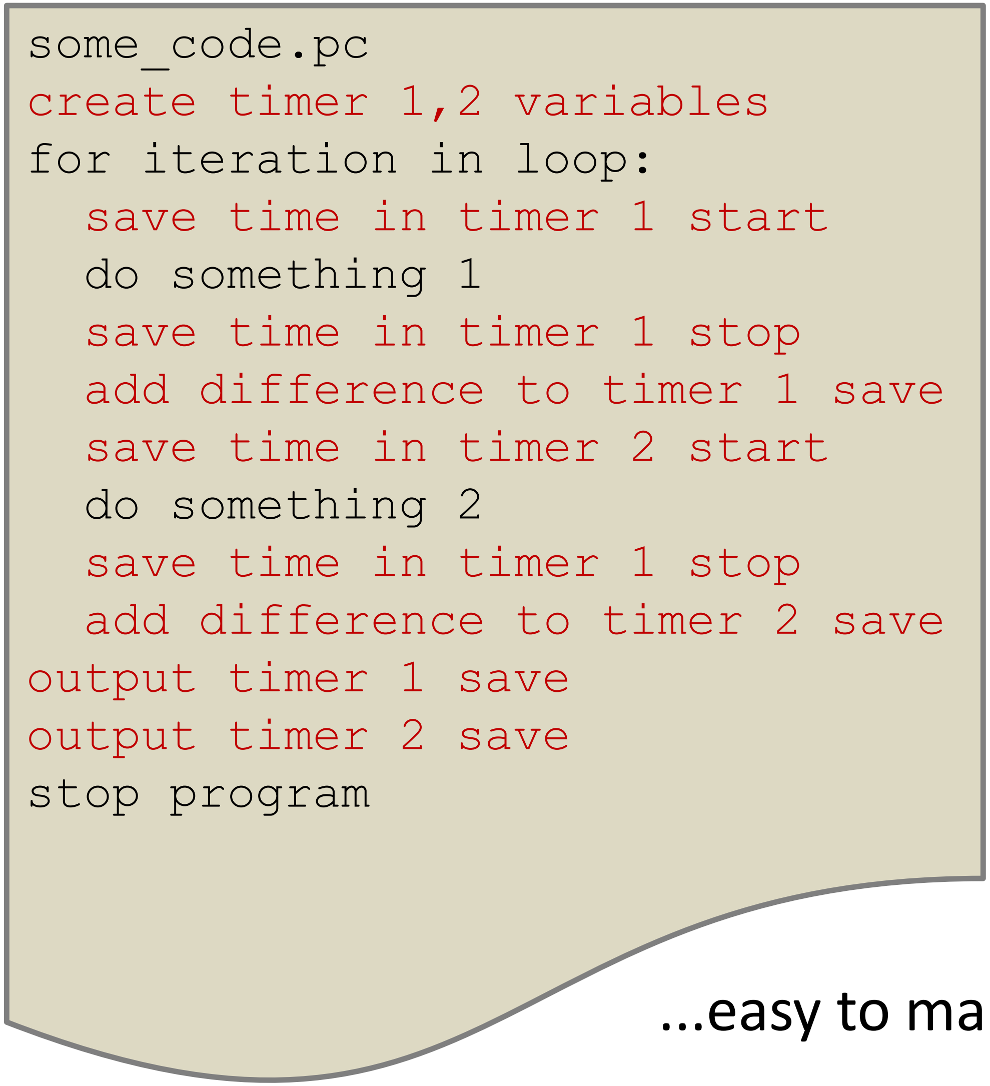
```
some_code.pc

for iteration in loop:

  start timer 1
  do something 1
  stop timer 1

  start timer 2
  do something 2
  stop timer 2

output timer 1, 2

stop program
```

A basic timer



We need 3 variables per timer...

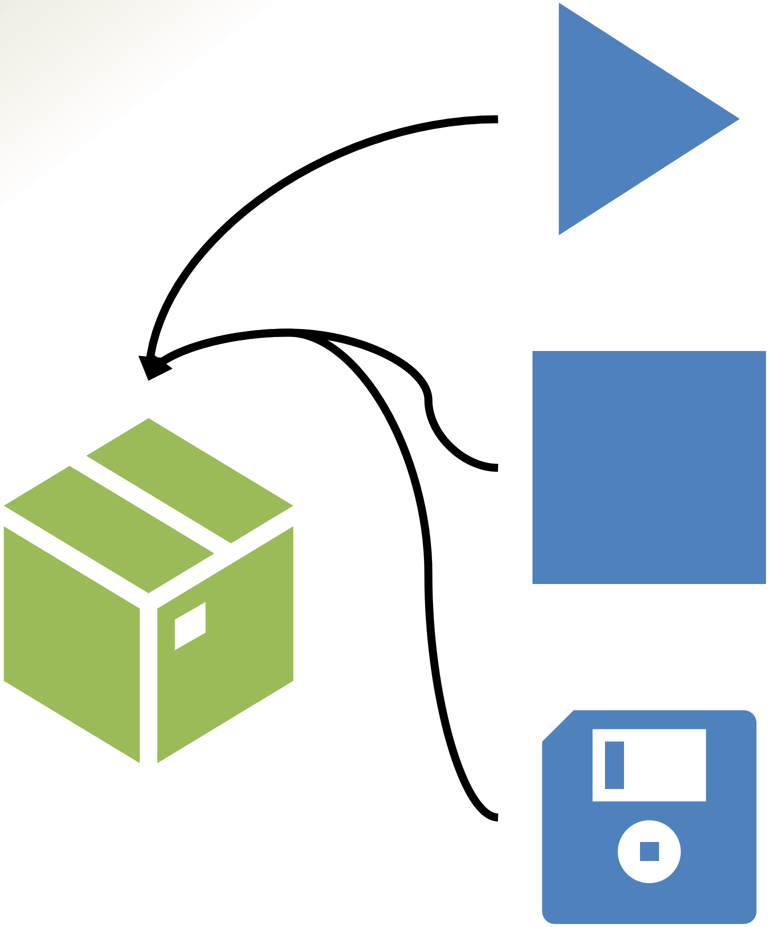
A simple use case: timer



```
some_code.pc
create timer 1,2 variables
for iteration in loop:
   save time in timer 1 start
   do something 1
   save time in timer 1 stop
   add difference to timer 1 save
   save time in timer 2 start
   do something 2
   save time in timer 1 stop
   add difference to timer 2 save
output timer 1 save
output timer 2 save
stop program
```

...easy to make mistakes...

Variable groups



We can group variables together to make things more organised

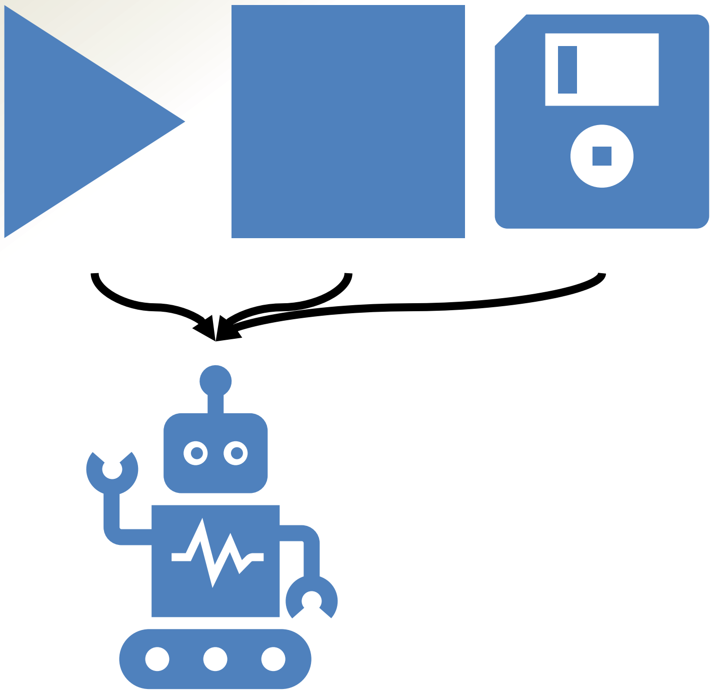
e.g. C struct,
F77 COMMON block (not really),
F90 module, C++ class,
Python class...

A simple use case: timer

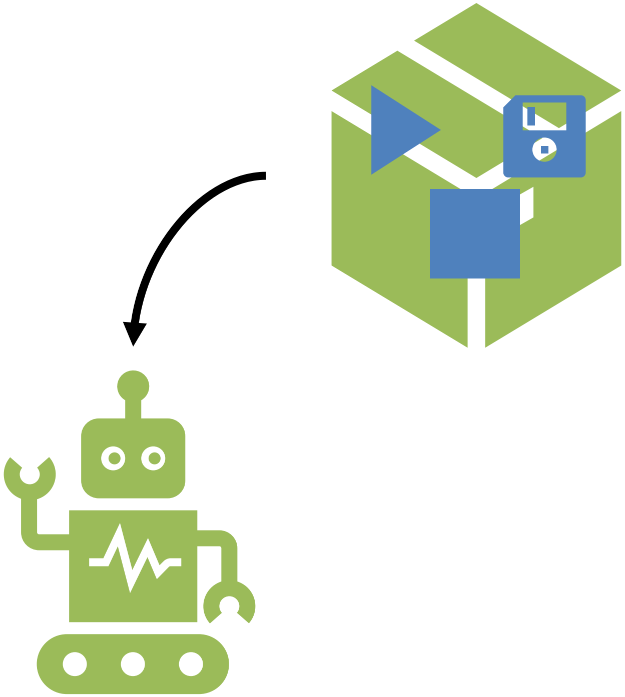
```
some_code.pc
create timer 1,2 boxes
for iteration in loop:
   save time in timer 1 box start
   do something 1
   save time in timer 1 box stop
   add diff to timer 1 box save
   save time in timer 2 box start
   do something 2
   save time in timer 1 box stop
   add diff to timer 2 box save
output timer 1 box save
output timer 2 box save
stop program
```

doesn't really help...

Using variable groups as variables



VARIABLE PARAMETER
TICK/TOCK FUNCTION



GROUP PARAMETER
TICK/TOCK FUNCTION

A simple use case: timer

```
some_code.pc
create timer 1,2 boxes
for iteration in loop:
   tick timer 1 box
   do something 1
   tock timer 1 box

   tick timer 2 box
   do something 2
   tock timer 2 box

output timer 1 box
output timer 2 box
stop program
```

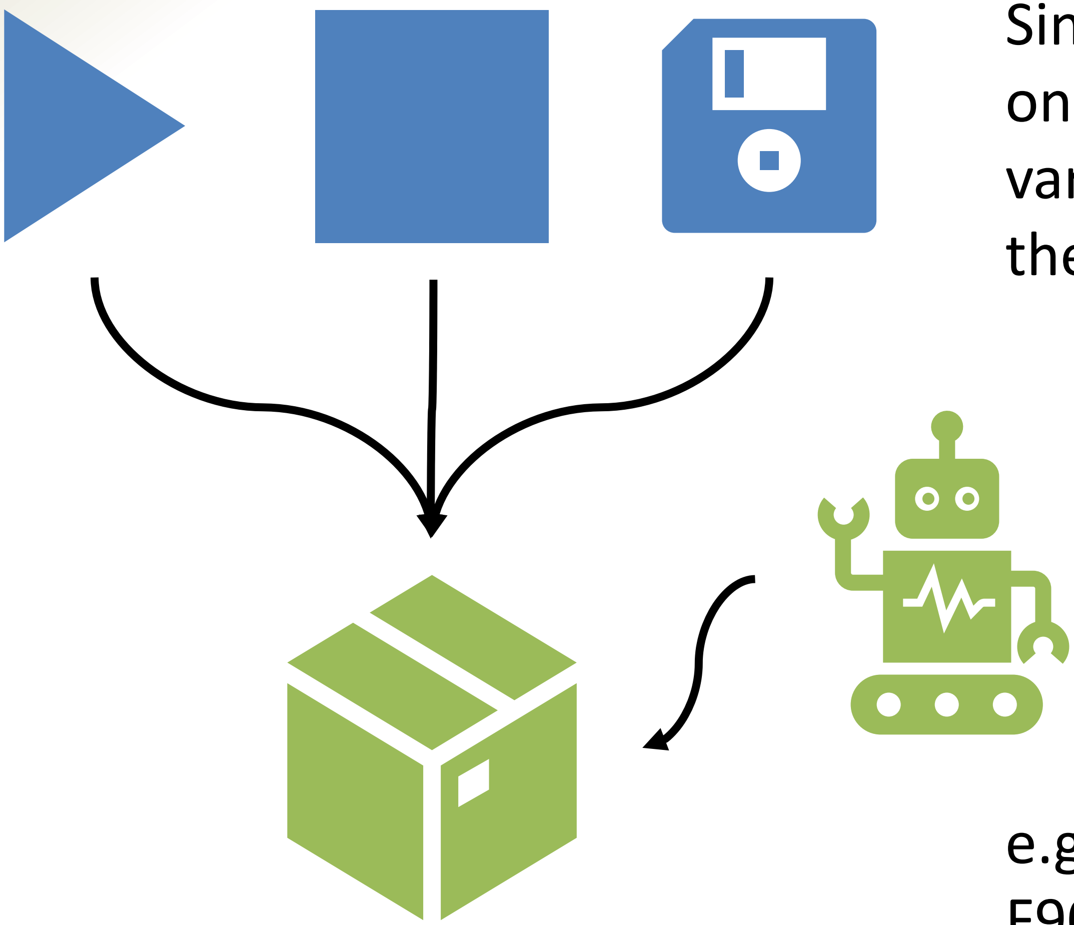
better, but... we could still mess with our timer...

A simple use case: timer

```
some_code.pc
create timer 1,2 boxes
for iteration in loop:
   tick timer 1 box
   do something 1
   tock timer 1 box
   change timer 1 save
   tick timer 2 box
   do something 2
   tock timer 2 box

output timer 1 box
output timer 2 box
stop program
```

better, but… we could still mess with our timer…

Group functions inside group



Since tick/tock functions only deal with group variables, we can make them part of the group

e.g. C++/Python class, F90 module...

A simple use case: timer

```
some_code.pc
create timer 1,2
for iteration in loop:
   timer1.tick()
   do something 1
   timer1.tock()

   timer2.tick()
   do something 2
   timer2.tock()

timer1.output()
timer2.output()
stop program
```

variables are completely hidden (and inaccessible)!

Objects and classes

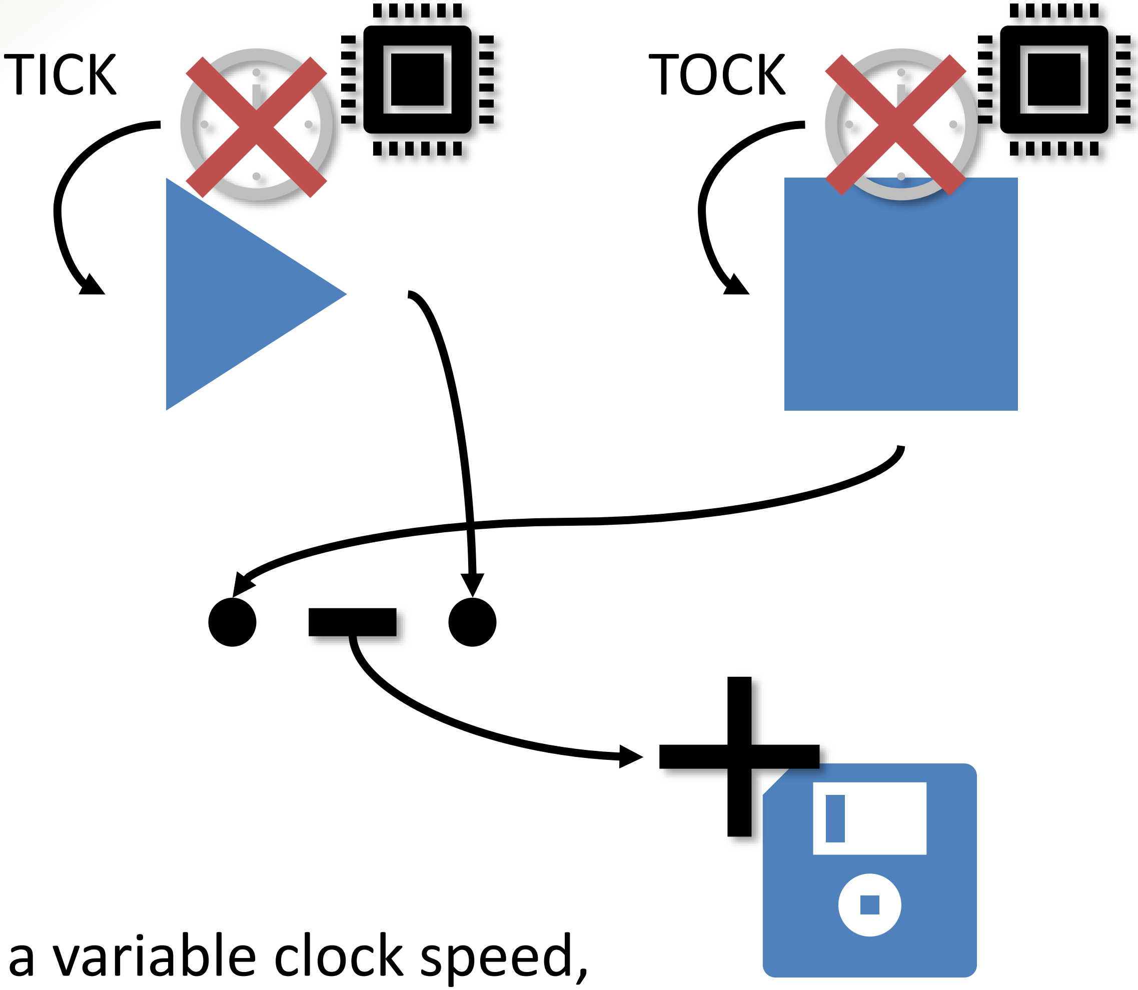
- Classes/modules group together variables and functionality that uses these variables
- Classes OWN their variables: no messing around with variables outside the class (unless you allow it; you shouldn't)
- Classes hide what happens internally from the rest of the program
- An instance of a class/module is an *object*

Why should I use this?

- Modularity: keep variables/code logic for a specific purpose in a separate place
- Avoids unnecessary code duplication
- Makes typos less likely or more obvious
- Results in more readable code (if member functions have clear names)

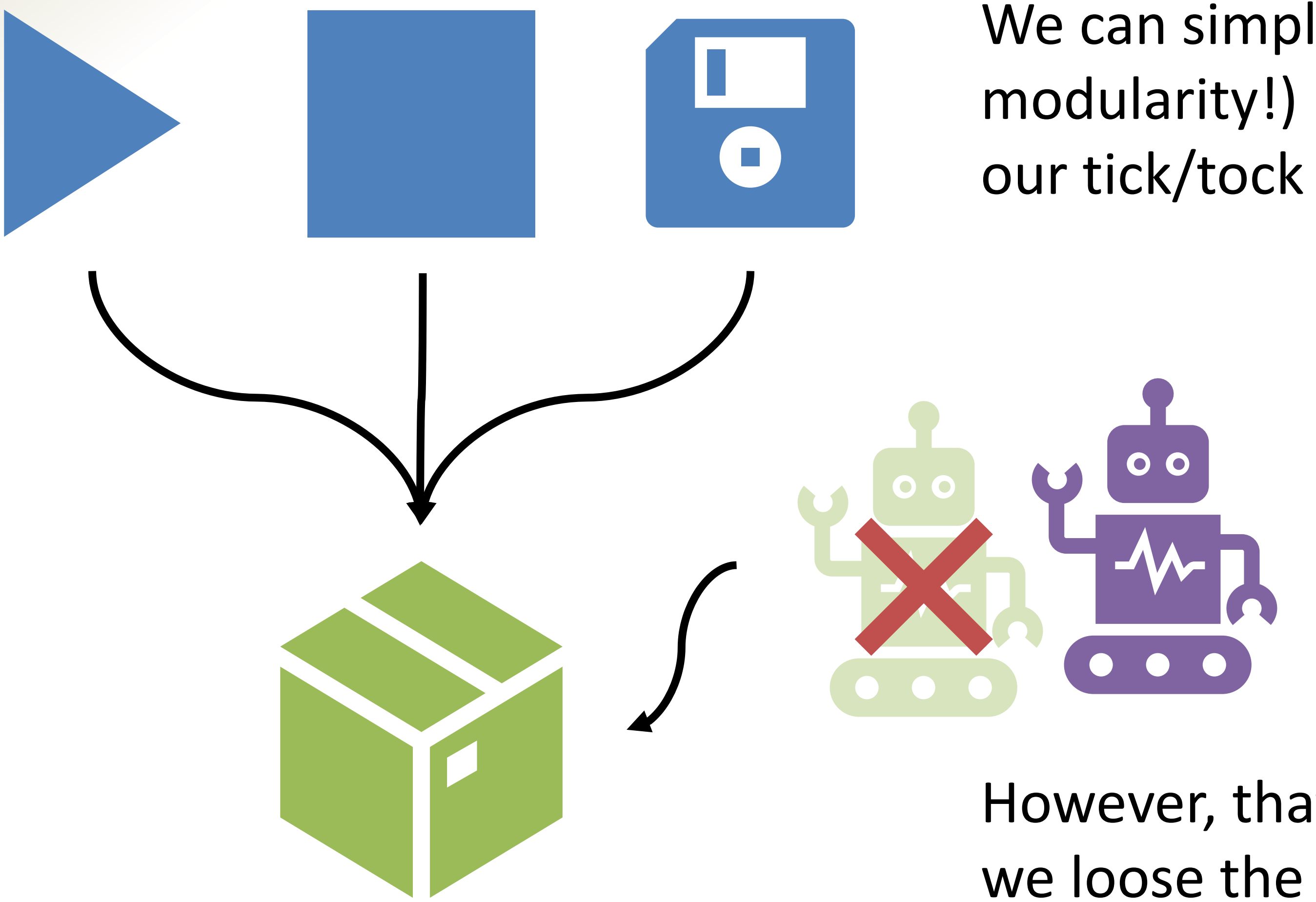
There is more...

An alternative timer



CPUs have a variable clock speed,
we want to measure CPU time

Alternative timer class

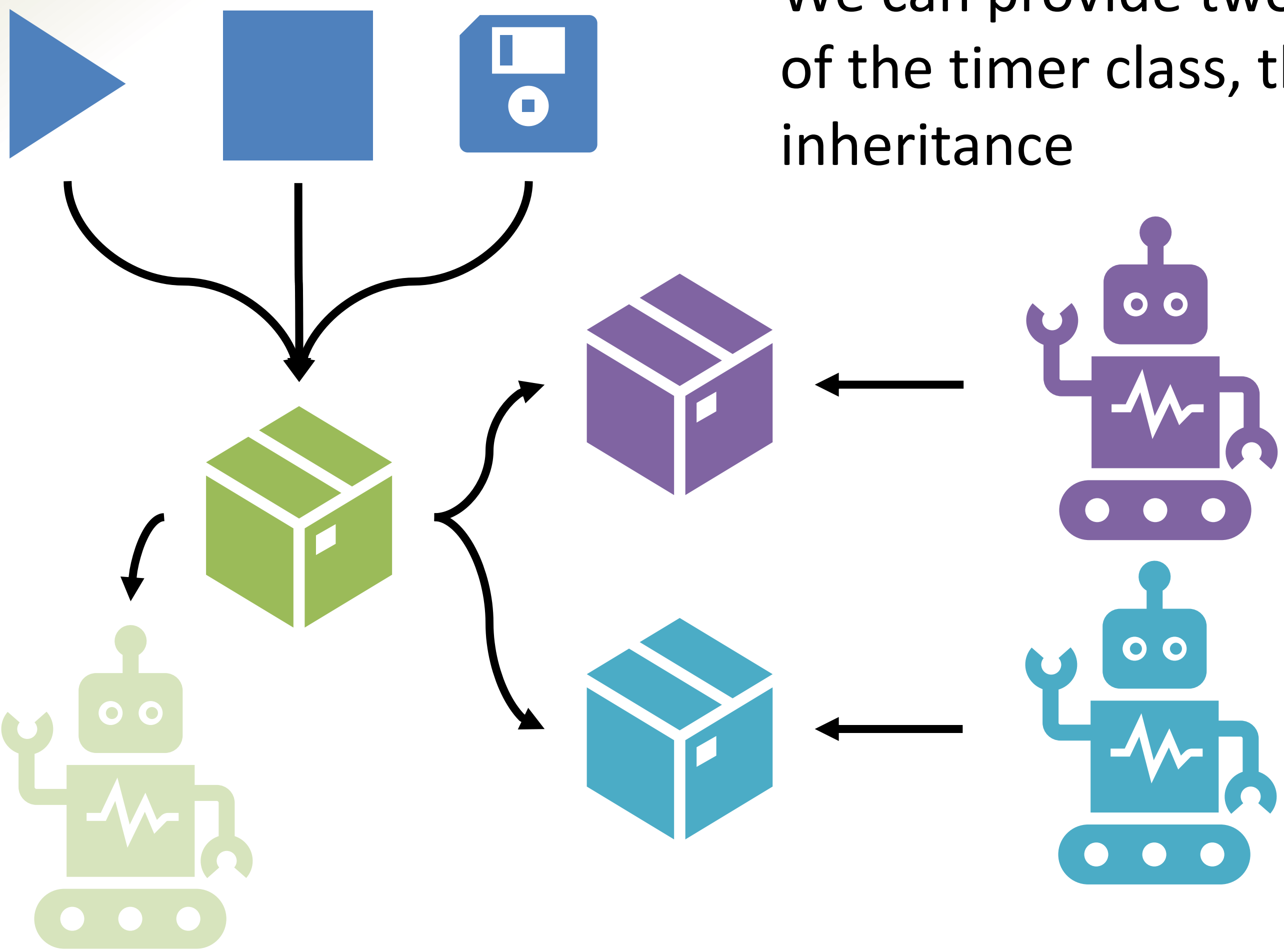


We can simply (yay modularity!) replace our tick/tock functions

However, that means we loose the capability of measuring real time...

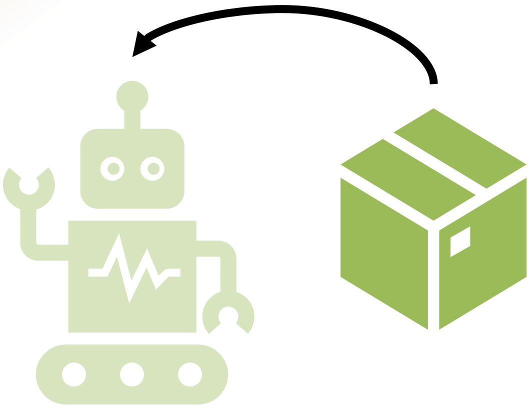
Abstract classes

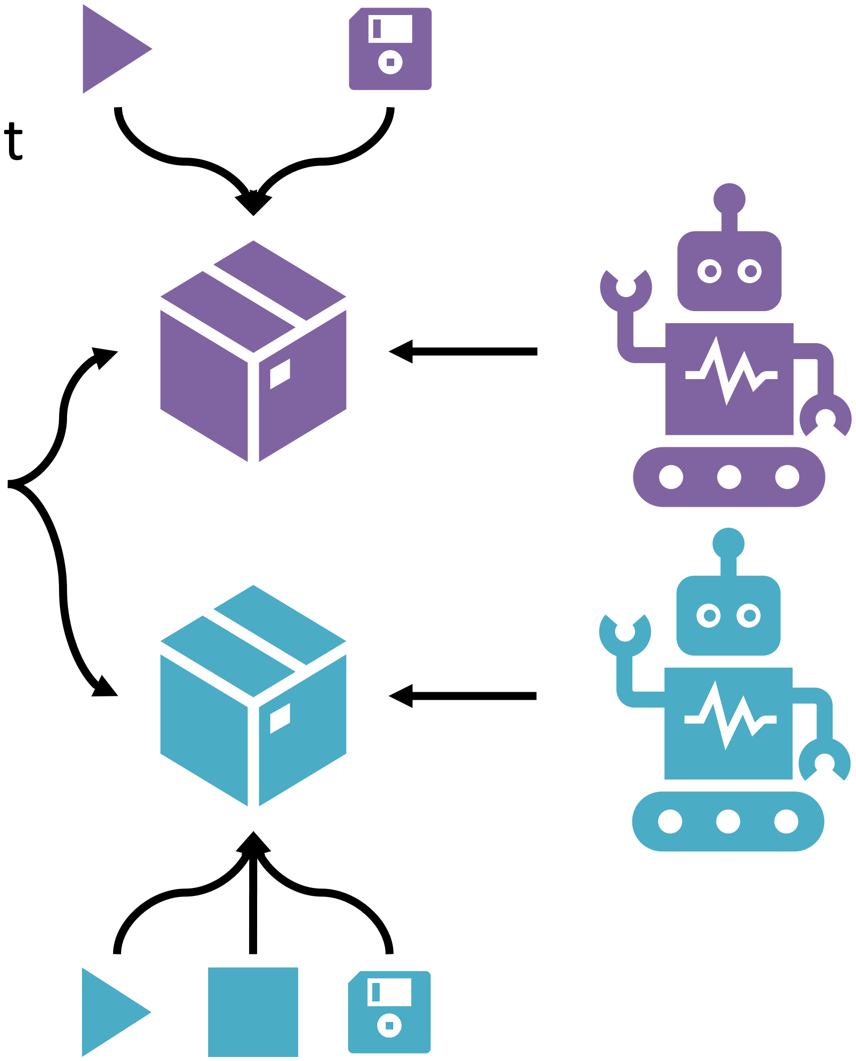
We can provide two versions of the timer class, this is called inheritance



Interfaces

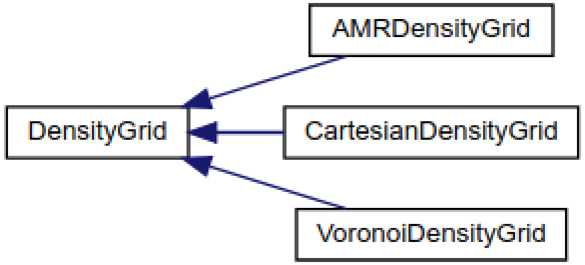
A parent class does not need to implement any functionality





Different child classes can be completely different internally

Real world examples

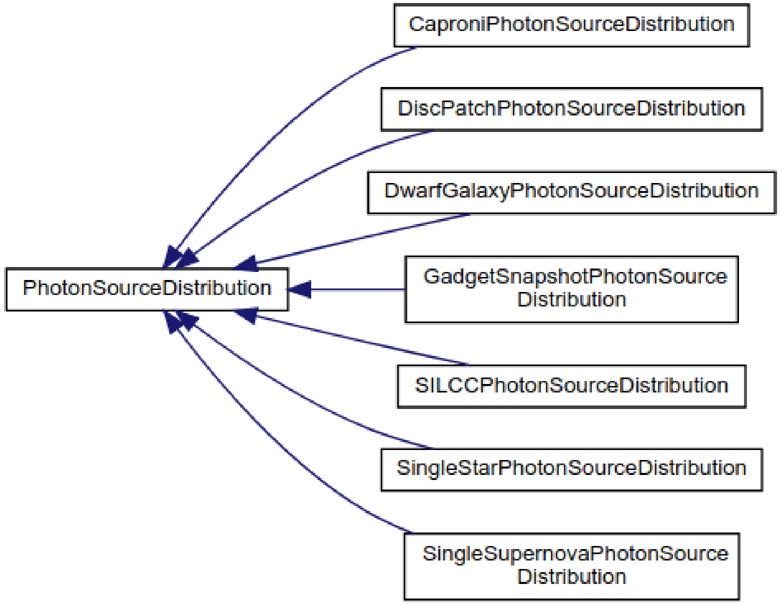


DensityGrid

AMRDensityGrid

CartesianDensityGrid

VoronoiDensityGrid



PhotonSourceDistribution

CaproniPhotonSourceDistribution

DiscPatchPhotonSourceDistribution

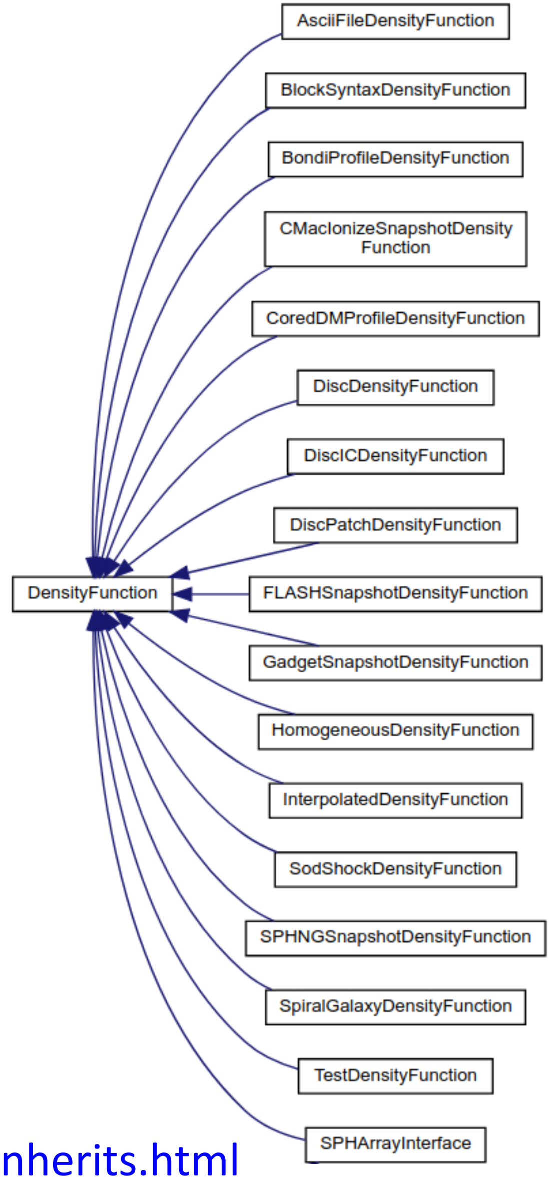
DwarfGalaxyPhotonSourceDistribution

GadgetSnapshotPhotonSource Distribution

SILCCPhotonSourceDistribution

SingleStarPhotonSourceDistribution

SingleSupernovaPhotonSource Distribution



DensityFunction

AsciiFileDensityFunction

BlockSyntaxDensityFunction

BondiProfileDensityFunction

CMacIonizeSnapshotDensity Function

CoredDMProfileDensityFunction

DiscDensityFunction

DiscICDensityFunction

DiscPatchDensityFunction

FLASHSnapshotDensityFunction

GadgetSnapshotDensityFunction

HomogeneousDensityFunction

InterpolatedDensityFunction

SodShockDensityFunction

SPHNGSnapshotDensityFunction

SpiralGalaxyDensityFunction

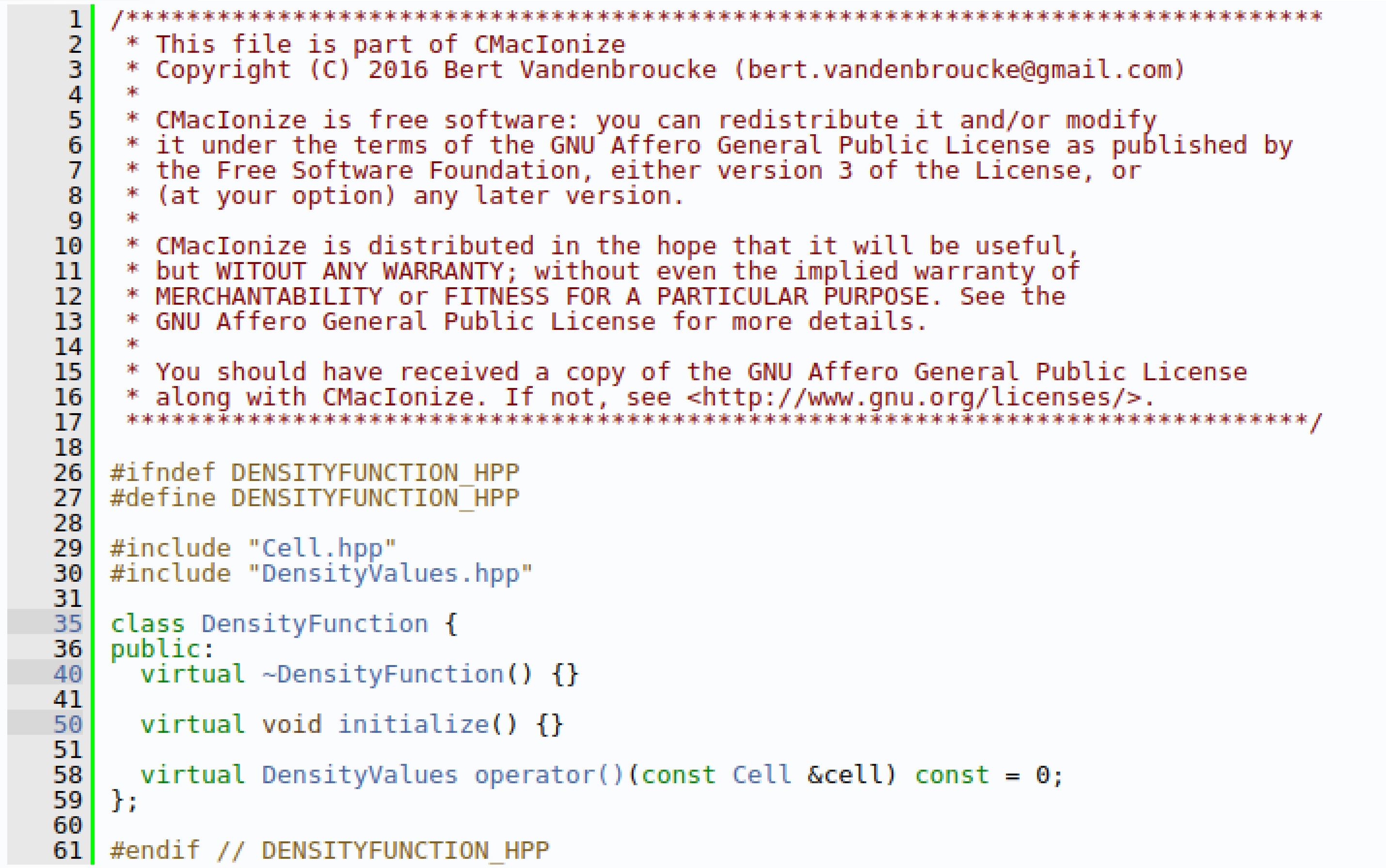
TestDensityFunction

SPHArrayInterface

CMacIonize

http://www-star.st-and.ac.uk/~bv7/CMacIonize_documentation/inherits.html

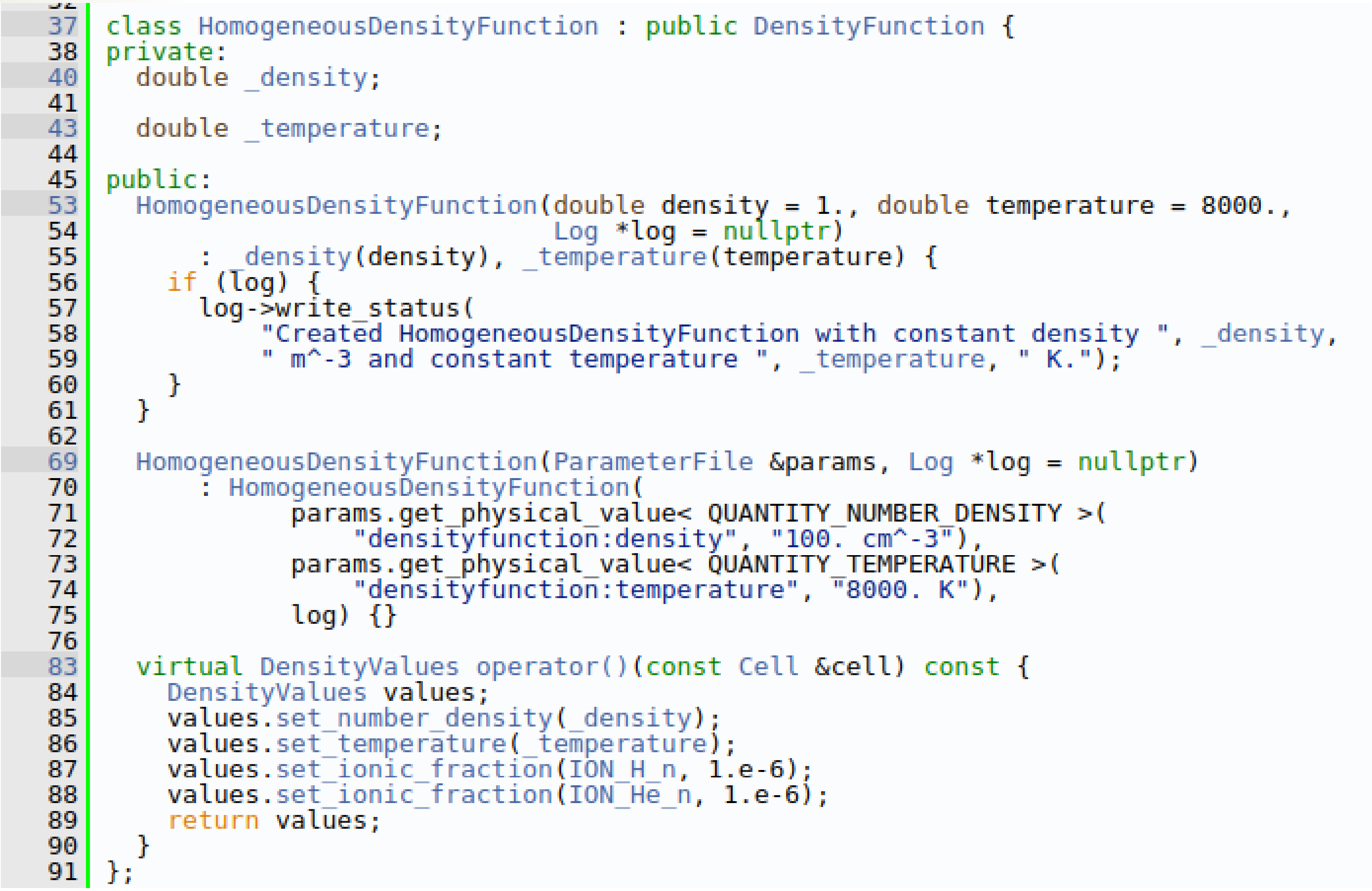
Real world examples (2)



```
/*******************************************************************************
 * This file is part of CMacIonize
 * Copyright (C) 2016 Bert Vandenbroucke (bert.vandenbroucke@gmail.com)
 *
 * CMacIonize is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Affero General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * CMacIonize is distributed in the hope that it will be useful,
 * but WITOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU Affero General Public License for more details.
 *
 * You should have received a copy of the GNU Affero General Public License
 * along with CMacIonize. If not, see <http://www.gnu.org/licenses/>.
 ******************************************************************************/

#ifndef DENSITYFUNCTION_HPP
#define DENSITYFUNCTION_HPP

#include "Cell.hpp"
#include "DensityValues.hpp"

class DensityFunction {
public:
  virtual ~DensityFunction() {}

  virtual void initialize() {}

  virtual DensityValues operator()(const Cell &cell) const = 0;
};

#endif // DENSITYFUNCTION_HPP
```

http://www-star.st-and.ac.uk/~bv7/CMacIonize_documentation/DensityFunction_8hpp_source.html

Real world examples (3)



```
class HomogeneousDensityFunction : public DensityFunction {
private:
  double _density;

  double _temperature;

public:
  HomogeneousDensityFunction(double density = 1., double temperature = 8000.,
                             Log *log = nullptr)
      : _density(density), _temperature(temperature) {
    if (log) {
      log->write_status(
          "Created HomogeneousDensityFunction with constant density ", _density,
          " m^-3 and constant temperature ", _temperature, " K.");
    }
  }

  HomogeneousDensityFunction(ParameterFile &params, Log *log = nullptr)
      : HomogeneousDensityFunction(
            params.get_physical_value< QUANTITY_NUMBER_DENSITY >(
                "densityfunction:density", "100. cm^-3"),
            params.get_physical_value< QUANTITY_TEMPERATURE >(
                "densityfunction:temperature", "8000. K"),
            log) {}

  virtual DensityValues operator()(const Cell &cell) const {
    DensityValues values;
    values.set_number_density(_density);
    values.set_temperature(_temperature);
    values.set_ionic_fraction(ION_H_n, 1.e-6);
    values.set_ionic_fraction(ION_He_n, 1.e-6);
    return values;
  }
};
```

http://www-star.st-and.ac.uk/~bv7/CMacIonize_documentation/HomogeneousDensityFunction_8hpp_source.html

Why I really like objects (and C++)

- In C++, operators (+, -, *, /...) are also functions
- The C++ syntax allows you to *overload* these operators for objects, i.e.

  ```
  a = b + c;
  ```

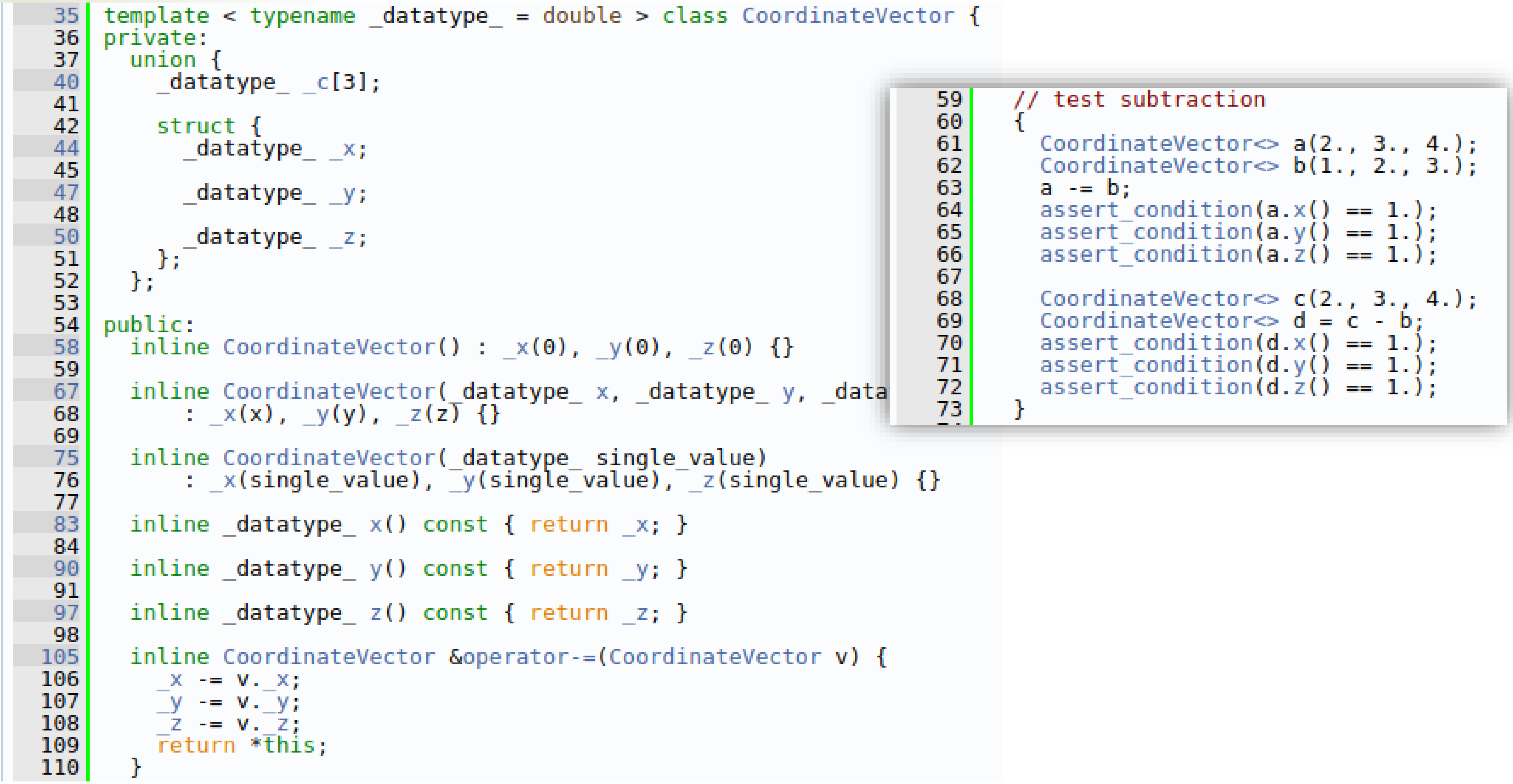
  equals

  ```
  a = operator+(b, c);
  ```

Why I really like objects (and C++)

- In-place operations (e.g. `a += b`) are overloaded by member functions of the class they act on
- We can disguise function calls as basic operators

More real world examples



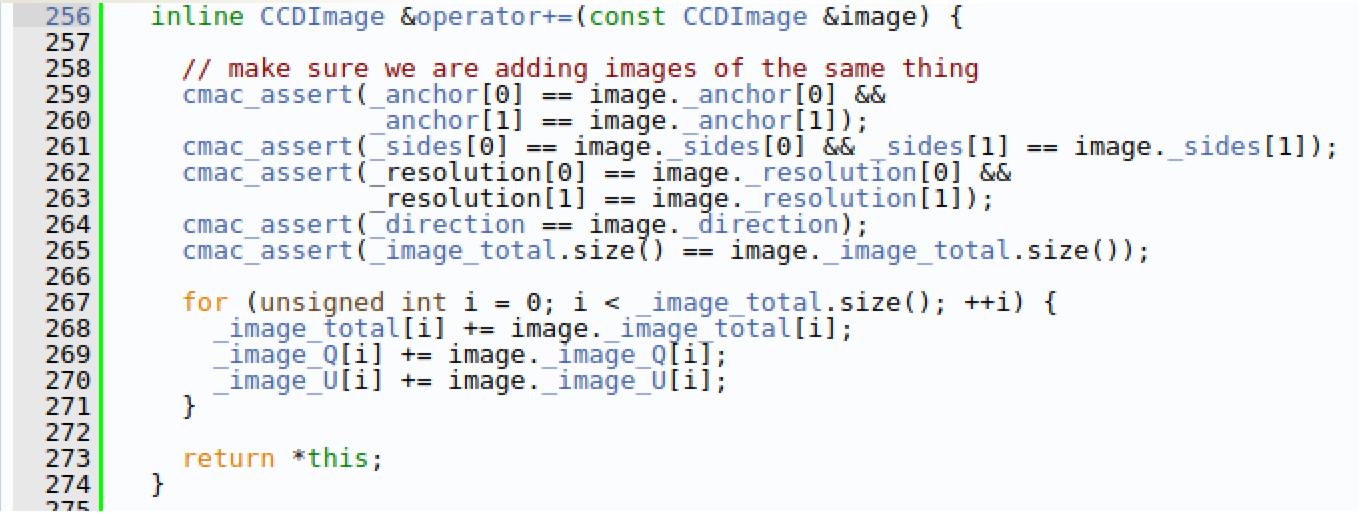
```
template < typename _datatype_ = double > class CoordinateVector {
private:
  union {
    _datatype_ _c[3];

    struct {
      _datatype_ _x;

      _datatype_ _y;

      _datatype_ _z;
    };
  };

public:
  inline CoordinateVector() : _x(0), _y(0), _z(0) {}

  inline CoordinateVector(_datatype_ x, _datatype_ y, _data
      : _x(x), _y(y), _z(z) {}

  inline CoordinateVector(_datatype_ single_value)
      : _x(single_value), _y(single_value), _z(single_value) {}

  inline _datatype_ x() const { return _x; }

  inline _datatype_ y() const { return _y; }

  inline _datatype_ z() const { return _z; }

  inline CoordinateVector &operator-=(CoordinateVector v) {
    _x -= v._x;
    _y -= v._y;
    _z -= v._z;
    return *this;
  }
```

```
// test subtraction
{
  CoordinateVector<> a(2., 3., 4.);
  CoordinateVector<> b(1., 2., 3.);
  a -= b;
  assert_condition(a.x() == 1.);
  assert_condition(a.y() == 1.);
  assert_condition(a.z() == 1.);

  CoordinateVector<> c(2., 3., 4.);
  CoordinateVector<> d = c - b;
  assert_condition(d.x() == 1.);
  assert_condition(d.y() == 1.);
  assert_condition(d.z() == 1.);
}
```

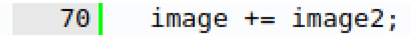
http://www-star.st-and.ac.uk/~bv7/CMacIonize_documentation/CoordinateVector_8hpp_source.html

http://www-star.st-and.ac.uk/~bv7/CMacIonize_documentation/testCoordinateVector_8cpp_source.html

More real world examples (2)



```
inline CCDImage &operator+=(const CCDImage &image) {

  // make sure we are adding images of the same thing
  cmac_assert(_anchor[0] == image._anchor[0] &&
              _anchor[1] == image._anchor[1]);
  cmac_assert(_sides[0] == image._sides[0] && _sides[1] == image._sides[1]);
  cmac_assert(_resolution[0] == image._resolution[0] &&
              _resolution[1] == image._resolution[1]);
  cmac_assert(_direction == image._direction);
  cmac_assert(_image_total.size() == image._image_total.size());

  for (unsigned int i = 0; i < _image_total.size(); ++i) {
    _image_total[i] += image._image_total[i];
    _image_Q[i] += image._image_Q[i];
    _image_U[i] += image._image_U[i];
  }

  return *this;
}
```



```
image += image2;
```

http://www-star.st-and.ac.uk/~bv7/CMacIonize_documentation/CCDImage_8hpp_source.html

http://www-star.st-and.ac.uk/~bv7/CMacIonize_documentation/testCCDImage_8cpp_source.html

Some additional thoughts

- Modularity makes *unit testing* very easy
- Classes add an abstraction layer to your program that makes it more intuitive:
  - actions rather than lists of instructions
  - logical entities rather than individual variables
- Inheritance makes implementing new functionality that is an alternative for existing functionality very easy

Summary

- Classes group together variables and functions with a specific functionality
- Classes provide modularity and abstraction to your program
- Classes make your code cleaner and easier to read and help you avoid making mistakes
- Classes are supported by C++, Python and modern Fortran, so no excuse not to use them