Today: Templates, STL and Design Patterns

- So far
  - Abstraction: classes define a clear interface, details are irrelevant to the user
  - **Polymorphism**: same interface for multiple classes => “virtual” calls
  - **Inheritance**: reuse common features of multiple classes
  - **Generic programming**: same code for different types => “**template**” code
  - C++ **STL** (Standard Template Library) => solves common problems
- Today
  - More on generic programming and STL
  - **Design Patterns** => abstract solution to recurrent software design problems
- Remember the usual disclaimer: Use with care!

Iterators

- Not all STL containers allow for random access like “v[10]”
  - but you can always iterate through the elements in some order
- STL introduces iterators to loop over any STL container
  - An iterator is “like a pointer”... but it works even if objects are spread in memory
  - Each STL container class has an iterator defined for it
  - Iterators can be incremented/decremented etc. like pointers
  - Allows for generic loops and algorithms on any container
- C++ style loop:

```
vector<int> v(10,1);  // an int vector
// iterate over int vector
for(vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
  cout << *it << endl;
}
```

Iterators

Generally, template-type-names can become very long
=> use “typedef” for readability
Typedef defines a new name “X” for an existing type “T”:
`typedef T X;`

```
typedef vector<int> myvec;  // we use int vectors
myvec v(10,1);
// iterate over the vector
for(myvec::iterator it = v.begin(); it != v.end(); ++it) {
  cout << *it << endl;
}
```

Access elements by dereferencing iterator (like a pointer)

begin() points to first element

end() points one past (!) the last element
(don’t dereference end(); it points to nowhere!)

We prefer ++it over it++. “it++” makes a temporary object which will be returned and then increments the iterator. “++it” does not require this temporary object...

Different kinds of Iterators

- random access iterators: `i+=3; --i; ++i`
- bidirectional iterators: `++i; --i` (read/write)
- forward iterators: `++i` (read/write)
- input iterators: `++i` (read-only)

- Depending on the container object, the type of the iterator defined for it may vary
- Example: lists do not have random access iterators; const objects only have input iterators

Practical iterators

- The standard `iterator`: goes from beginning to end
- The `const_iterator` does the same, but does not allow changing the entries in the container
- The `reverse_iterator` goes from the end to the beginning
  - use rbegin() and rend() for start and end (instead of begin() and end())

Iterators - Example


```
int main() {
  // initialize v
  vector<int> v;
  for (int k = 0; k < 7; ++k) v.push_back(k); // grows autom.
  display(v); // 0 1 2 3 4 5 6

  // use standard iterator to add 3 to each element
  vector<int>::iterator i;
  for (i = v.begin(); i != v.end(); ++i) {
    *i += 3;
  }
  display(v); // 3 4 5 6 7 8 9

  // use const iterator for display (cannot change elements!)
  vector<int>::const_iterator ci;
  for (ci = v.begin(); ci != v.end(); ++ci) {
    cout << *ci << ' ';  // *ci -= 3; won't compile
  }
  cout << endl;    // 3 4 5 6 7 8 9

  // check reverse iterator
  vector<int>::reverse_iterator ri;
  for (ri = v.rbegin(); ri != v.rend(); ++ri) {
    *ri -= 3;    // supports write access
    cout << *ri << ' ';
  }
  cout << endl; // 6 5 4 3 2 1 0

  return 0;
}
```

see iterators.cpp

Iterators to insert/remove elements

- **vector** can efficiently access/overwrite elements in O(1) and add/remove elements at the end of the vector in O(1)
- one can also insert/remove elements anywhere else, but computational cost is O(N), where N is size of vector
  - other containers like **list** do this in O(1), but may not have (fast) random access
- Commands insert and erase work with iterators

```
int main() {
  // initialize v
  vector<int> v;
  for (int k = 0; k < 7; ++k) v.push_back(k); // grows autom.
  display(v); // 0 1 2 3 4 5 6

  // insert/erase elements
  v.insert(v.begin()+2,10); // insert before 3rd element
  v.erase(v.begin()+4);     // erase 5th element
  display(v); // 0 1 10 2 4 5 6

  return 0;
}
```

Note:
“begin()+i” only works if iterator of container supports random access

see iterators.cpp

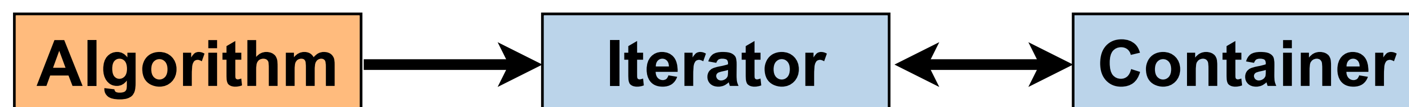
Iterators - Example (display)

- What is the “display” function? It prints the contents of a vector
- We can make it **work for any STL container** using iterators and generic programming!
- Generic programming also works for complex data types such as entire templates!

```
template <typename Container>
// “const” limits to input iterators -> always defined.
// call by reference avoids copy of large data
void display(const Container &c) {
  // "typename" needed due to template type (explained later...)
  typename Container::const_iterator i;
  for (i = c.begin(); i != c.end(); ++i) cout << *i << " ";
  cout << endl;
}
```

STL algorithms

- STL provides templated implementations of numerous algorithms of common interest
- Pretty much all of them take and return iterators
- They act on the contents of containers using iterators



- Selection of algorithms:
  - Copying
  - Searching unsorted sequences
  - Searching sorted sequences
  - Replacing/removing elements
  - Reordering
  - Sorting
  - Merging sorted sequences
  - Set operations
  - Heap operations
  - Minimum/maximum
  - Permutations
  - Transforming and generating
  - Numeric

Algorithm demo...

CSElab
www.cse-lab.ethz.ch

```
#include <iostream>
#include <vector>       // include STL vector definition
#include <algorithm>    // include STL algorithms

using namespace std;

// functor: can call s(d) on object s of type Sum
struct Sum {
  Sum() { sum = 0; }
  void operator()(double n) { sum += n; }
  double sum;
};

// global functions -> can also be used instead of functors
void printme(double x) { cout << x << " "; }
double reciprocal(double x) { return 1/x; }

int main() {
  // initialize v
  vector<double> v;
  for (int k = 1; k < 10; ++k) v.push_back(k); // grows autom.

  // display
  for_each(v.begin(), v.end(), printme); cout << endl;

  // sum up numbers (src: cppreference)
  Sum s = for_each(v.begin(), v.end(), Sum());
  cout << "Sum: " << s.sum << endl;

  // reverse
  reverse(v.begin(), v.end());
  for_each(v.begin(), v.end(), printme); cout << endl;

  // transform
  transform(v.begin(), v.end(), v.begin(), reciprocal);
  for_each(v.begin(), v.end(), printme); cout << endl;

  return 0;
}
```

see algorithms.cpp

- Use `#include <algorithm>`
- Methods commonly work on iterators (would also work with `list<double> v`)
- Functions or function objects (“functors”) may be passed to perform actions
- transform and reverse overwrite input
- Can you predict the output without running the code?

Algorithms in real life...

- **Goal**: I want to find an element in a vector of strings and add a new string before it
  - class **string** allows to store strings and compare them with “==”
- Setup a test for your problem
  - small vector with a few strings
  - check http://en.cppreference.com/w/cpp/algorithm for an appropriate algorithm
  - compare possible methods (find? find_if? search?)
  - apply method to our little test
  - then: use it wherever you actually needed it...
  - see find.cpp for result...

More algorithms


```cpp
#include <iostream>
#include <vector>     // include STL vector definition
#include <algorithm>  // include STL algorithms
#include <string>     // include STL strings

using namespace std;

// what does this do?
template <typename Container>
bool is_something(const Container &input) {
  return equal(input.begin(),input.end(),input.rbegin());
}

int main() {
  // initialize v
  vector<double> v(5);
  v[0] = 2.4; v[1] = 122; v[2] = 5; v[3] = 3.1; v[4] = 50;

  // some statistics
  sort(v.begin(),v.end());  // first we sort
  // min/max easy to get
  cout << "min/max: " << v.front() << " " << v.back() << endl;

  // same with C-arrays?
  double v2[5] = {2.4, 122, 5, 3.1, 50};
  sort(v2,v2+5);
  cout << "min/max: " << v2[0] << " " << v2[4] << endl;

  // check something
  string str = "HALLO";
  if (is_something(str)) {
    cout << str << " is quite something" << endl;
  } else {
    cout << str << " is not something" << endl;
  }

  return 0;
}
```

see algorithms.cpp

- Can use pointers instead of iterators for most algorithms
  - begin() => pointer to first
  - end() => pointer one past end
  - how does this work?
    - template-type iterator used in algorithm. Syntax for iterators and for pointers are the same!
- What does is_something method do?
  - Note: equal(first1,last1,first2) checks two vectors for equality of their contents

Fazit on STL

- STL offers many commonly used data structures and algorithms
- Let the compiler do the overloading work for you - use generic programming and templates!
- Save time and lines of code
- Make your code more readable
- Do not “re-implement the wheel”
- Make less mistakes: someone else has already tested and debugged the library functions for you
- **STL is MUCH MORE than what I could cover here!**
  - **check cppreference etc for anything you may use from there**

Intermezzo on code structure

- **Main** program (commonly main.cpp)
  - contains main function (“int main()” or “int main(int argc, const char * argv[])”)
- Header files ***.h**
  - declare classes and functions <= *HINT: usually enough to understand what it does*
- Source files ***.cpp**
  - define functions (and global data) <= *HINT: only look at it if really, really necessary*
- Compilation with Makefile (provided by us in exercises)
  - compile with “make” => convenient and fast
  - each cpp file compiled independently and combined (“linked”) in the end
    - length of single cpp file (+ included .h files) defines compile-time
    - linking is usually fast (scales with number of functions and not with lines of code)
    - only cpp files (+ included .h files) that changed must be recompiled

Template classes vs code structure

- Reminder: template class doesn't define a class (yet)
  - only once we use it and define the template-arguments, the class exists
  - compiler cannot generate any code before it knows the template-arguments!
  - conclusion: **template classes must be defined in headers**!
- But we can split class declaration and definition:

```
template <typename X>
class Container {
  int size;
  X * data;
public:
  // constructor with initialization list
  Container(int N): size(N) { data = new X[N]; }
  // destructor
  ~Container() { delete[] data; }
  // access operator for non-const Container
  X& operator[](unsigned int i) {
    return data[i];
  }
};
```

```
// class declaration
template <typename X>
class Container {
  int size;
  X * data;
public:
  Container(int N);
  ~Container();
  X& operator[](unsigned int i);
};

// class definition
template <typename X>
Container<X>::Container(int N): size(N) { data = new X[N]; }

template <typename X>
Container<X>::~Container() { delete[] data; }

template <typename X>
X& Container<X>::operator[](unsigned int i) {
  return data[i];
}
```

(all of this needs to go in a header file)

Templates vs Inheritance

- **Both help you avoid duplicated code**
- Generic arguments: must be known at compile-time
  - good: compiler can optimize, bad: you cannot set them at run-time...
- Generic classes/functions must be defined in headers
  - big headers = big code to compile for each cpp file = long compile time
- Inheritance: can use base pointers
  - can have containers with pointers to different types of objects
- How to define requirements for type of arguments:
  - templates: not clear => any type could work if correct methods defined

    ```
    template <typename T> void doIt(T* t) { ... }
    ```

  - polymorphism: base class defines requirements

    ```
    void doIt(Base* t)
    ```

Caveats with templates

Inheritance

need to access members/methods of base class with "this->"

```
template <typename T>
class Foo
{
public:
  T bla;
  void what() { std::cout << bla << std::endl; }
};

template <typename T>
class Bar: public Foo<T>
{
public:
  void test() {
    what();          // FAILS
    this->what();    // OK
    bla = 10;        // FAILS
    this->bla = 10; // OK
  }
};
```

Nested types of template type

need keyword "typename"

```
template <typename Container>
void display(Container &c) {
  Container::iterator i;            // FAILS
  typename Container::iterator i;   // OK
  for (i = c.begin(); i != c.end(); ++i) {
    cout << *i << " ";
  }
  cout << endl;
}
```

Here "iterator" is defined as a nested class or with a "typedef" in the type passed as "Container". E.g.:

```
// example Container
class MyC {
  double data[10];
public:
  typedef double* iterator;
  iterator begin() { return data; }
  iterator end() { return data+10; }
};

int main() {
  MyC c;
  display(c);

  return 0;
}
```

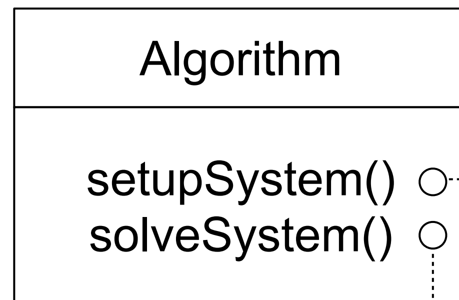
Why is this necessary? Answer is way out of scope for this lecture...
More info in [35.18] and [35.19] at: http://www.parashift.com/c++-faq-lite/templates.html

Design Patterns

- A **design pattern** is an **abstract formalization of the solution** to a recurrent software design problem
  - Just as novelists use story patterns and prototypical characters, software engineers use recurrent constellations of objects and classes
  - They provide flexible solutions to frequent design challenges and make life easier
- **Related to polymorphism**, not to generic programming
  - even though some patterns could also be used with templates...
- Catalog of patterns has been created and presented in the book “Design Patterns” (E. Gamma, R. Helm, R. Johnson, J. Vlissides)
  - Many of the design patterns had been in use to enhance modularity, reusability and maintainability of large object-oriented codes
- **In this course**: basic motivation, overview of existing design patterns and an illustrative example

Design Patterns - Example

- **Task**: design an algorithm with exchangeable components
  - here: discretize PDE and solve the resulting linear system of equations
- Algorithm class:

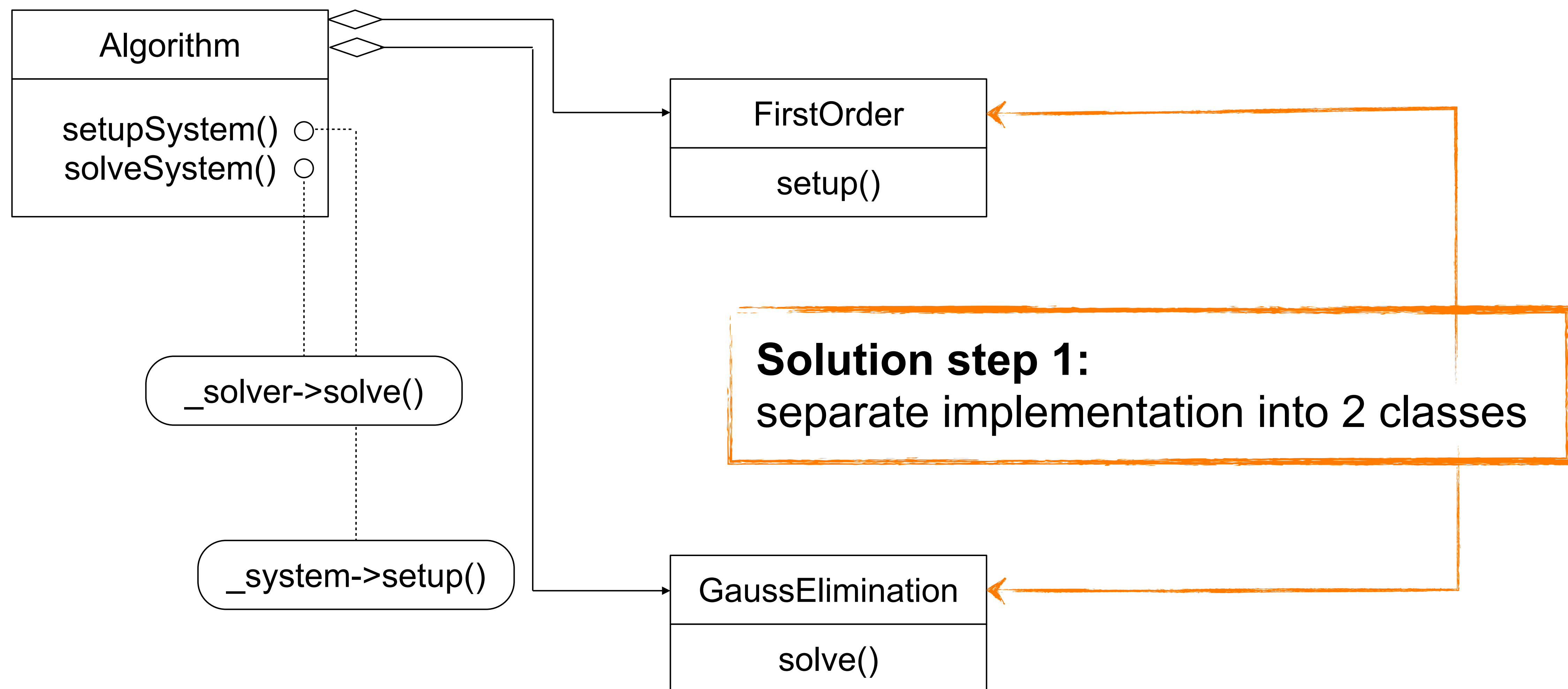


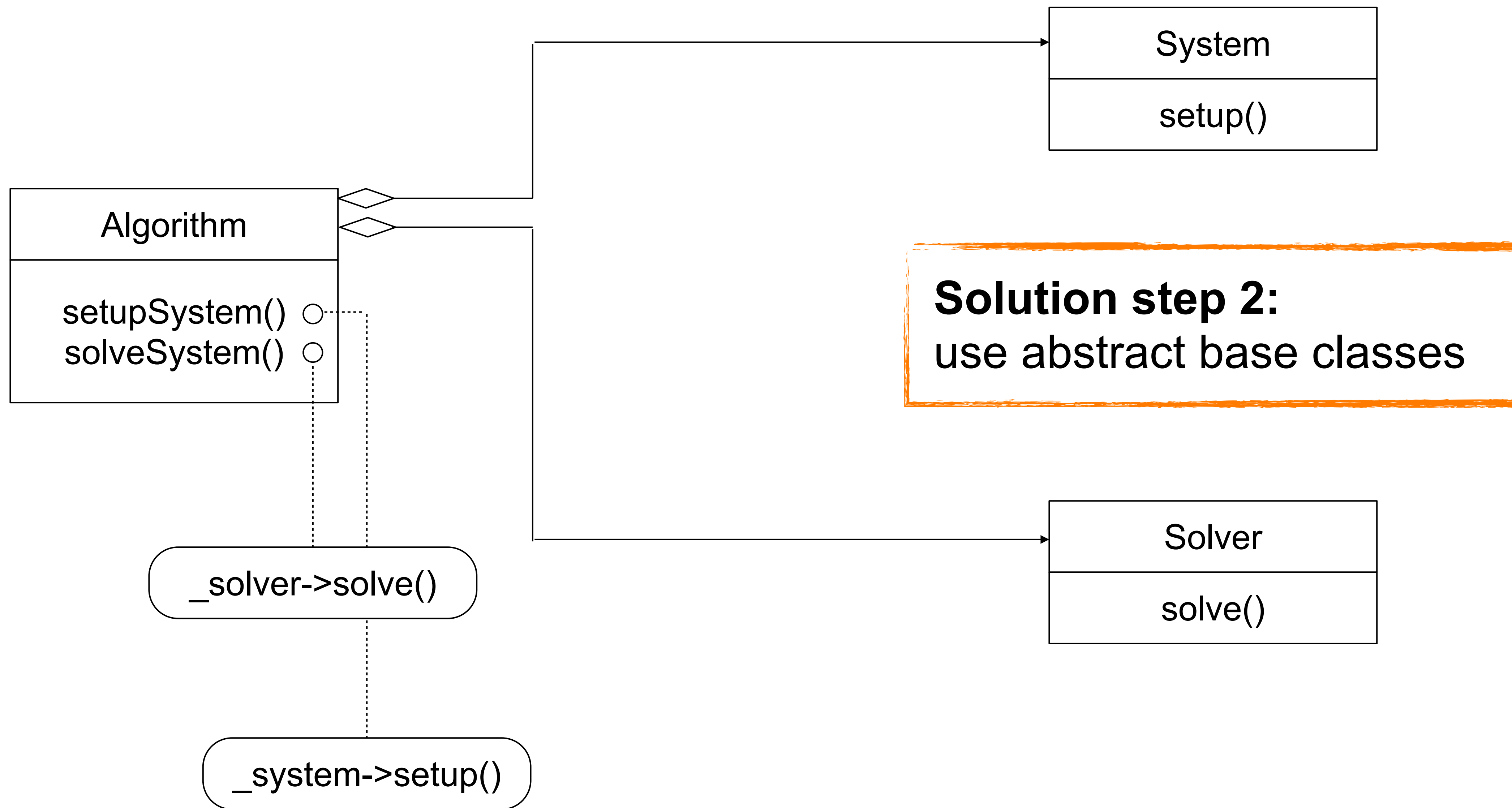
BUT: you may want to use other discretization and/or other linear solvers

Gauss elimination

1st order discret.

- **Task**: design an algorithm with exchangeable components
  - here: discretize PDE and solve the resulting linear system of equations
- Algorithm class:

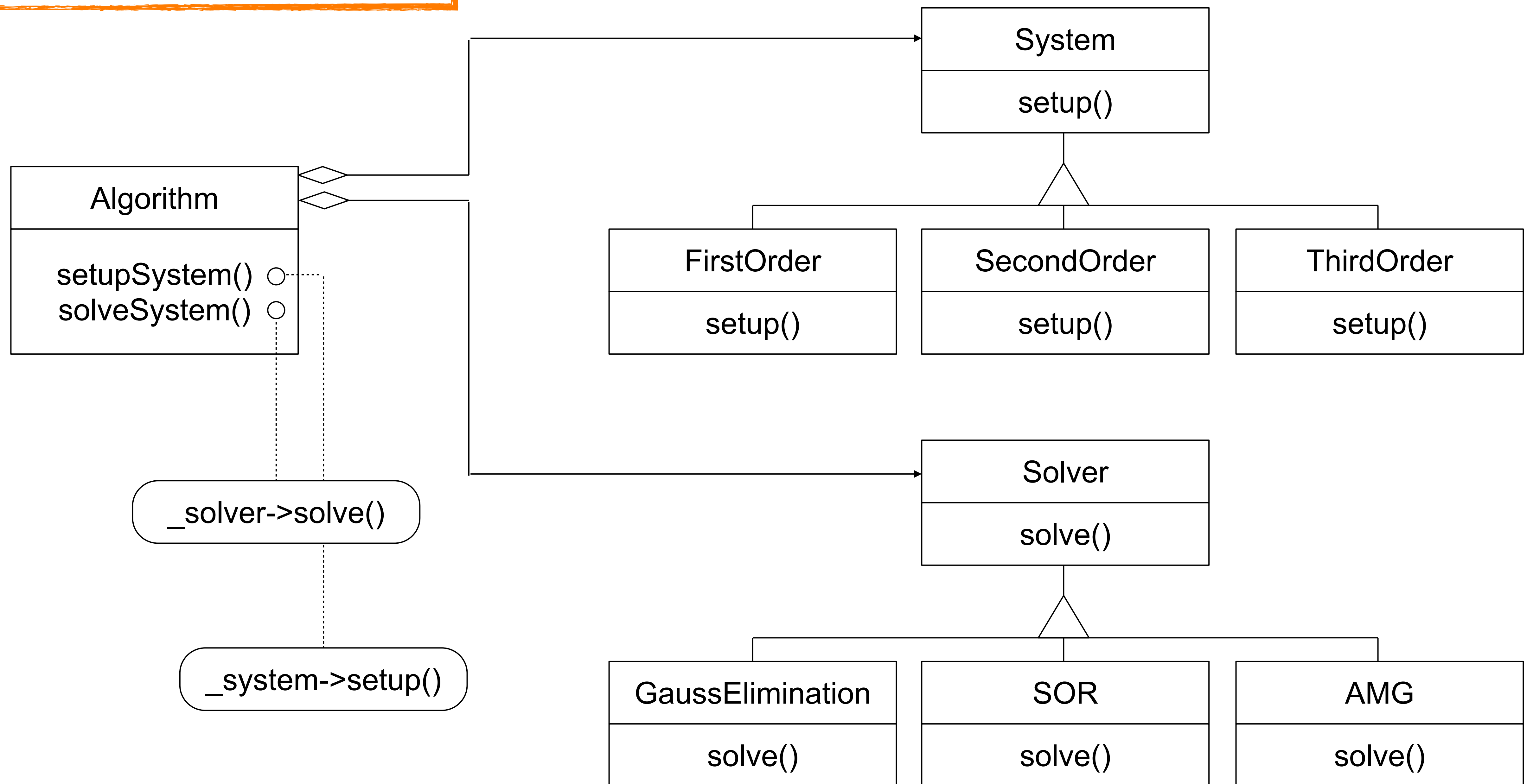




Solution step 2:
use abstract base classes

Design Patterns - Example

Solution step 3:
implement different versions as derived classes



This is called the "**Strategy Pattern**"

Design Patterns - Example

Algorithm

```
// generic algorithm
class Algorithm {
  System * _system;
  Solver * _solver;
public:
  Algorithm(System * system, Solver * solver) {
    _system = system;
    _solver = solver;
  }
  void run() {
    _system->setup();
    _solver->solve();
  }
};
```

Use case

```
int main() {
  // declarations
  System * system;
  Solver * solver;
  int which_order;
  int which_solver;
  // choose discretization and solver
  cout << "Enter 1 for 1st order only (else 2nd order) => ";
  cin >> which_order;
  cout << "Enter 1 for AMG (else SOR)                  => ";
  cin >> which_solver;
  // init system and solver
  if (which_order == 1) system = new System_FirstOrder;
  else                  system = new System_SecondOrder;
  if (which_solver == 1) solver = new Solver_AMG;
  else                   solver = new Solver_SOR;
  // run algorithm
  Algorithm algorithm(system,solver);
  algorithm.run();
  // clean up
  delete system;
  delete solver;

  return 0;
}
```

System classes

```
// abstract base class
class System {
public:
  virtual void setup() = 0;
};
// derived classes
class System_FirstOrder: public System {
public:
  void setup() {
    cout << "Setup 1st order discretization" << endl;
  }
};
class System_SecondOrder: public System {
public:
  void setup() {
    cout << "Setup 2nd order discretization" << endl;
  }
};
```

Solver classes

```
// abstract base class
class Solver {
public:
  virtual void solve() = 0;
};
// derived classes
class Solver_SOR: public Solver {
public:
  void solve() {
    cout << "Solve using SOR" << endl;
  }
};
class Solver_AMG: public Solver {
public:
  void solve() {
    cout << "Solve using AMG" << endl;
  }
};
```

see strategy.cpp

Strategy Pattern

- System- and Solver-trees can also be used in other codes, where the Algorithm class is not involved
  => **modular design enhances reusability**

- System- and Solver-trees can be developed in isolation; only interfaces and objective have to be known
  => **simplifies complex code development and maintainability**, since developer does not need to understand everything

- New solvers and new discretization schemes can be added without affecting rest of code
  => **allows to extend software** with state-of-the-art developments

Design Patterns - Catalog

- Types of design patterns:
  - **Creational**: about instantiating objects
  - **Structural**: compose objects into larger structures
  - **Behavioral**: program control flow
- Elements of a pattern
  - **Name** (catalog): enhances communication and allows for high level of abstraction
  - **Problem**: typical examples fulfilling a list of conditions
  - **Solution**: design, relationships, responsibilities and collaborations of and between classes
  - **Consequences**: results and trade-offs, e.g. impact on a system's flexibility, extensibility, or portability, but also regarding memory and computational cost
- Note: patterns often overlap and/or solve similar problems

Creational Design Patterns

- Creational Patterns deal with **instantiation of objects**
- Examples:
  - **Abstract Factory**: interface for creating objects without specifying a concrete class (commonly implemented using the Factory Method pattern below)
  - **Builder**: separate construction, such that it can be used to create different complex objects
  - **Factory Method**: interface for creating objects, but subclasses decide which object to instantiate
  - **Prototype**: specify interface of objects to be created and a prototype instance
  - **Singleton**: ensures that only one instance of a class can be created, which is globally accessible
- Example task: algorithm needs to create new particles, but Particle-class is to be chosen at runtime

=> use (Abstract) Factory or Prototype Pattern

Structural Design Patterns

- How classes and objects are combined to form larger structures
- Examples:
  - **Adapter**: convert interface of a class (wrapper)
  - **Bridge**: decoupling of abstraction and implementation
  - **Composite**: treat individual objects and composites thereof uniformly
  - **Decorator**: extend functionality of subclass
  - **Facade**: provide uniform interface of subsystem
  - **Flyweight**: support large numbers of fine-grained objects
  - **Proxy**: placeholder for another object to control access to it
- Example task: we are given a FFT library which uses its own grid structure etc. We want to use it with our own data structures in our existing simulation code => use Adapter Pattern

Behavioral Design Patterns

- **Control program flow** (e.g. choice of algorithms)
  - not only patterns of objects/classes, but also patterns of interactions between them
- Examples:
  - **Chain of responsibilities**: Process and/or pass on requests along a chain of objects. Decide at run-time what to do
  - **Command**: Encapsulate requests in objects so they can be passed as parameters, stored in lists, or manipulated
  - **Interpreter**: Represent a grammar (language, formulas, ...) as a class hierarchy and implement an interpreter on it
  - **Iterator**: Abstract the way an aggregate (e.g., a Composite pattern or an STL container) is traversed
  - **Mediator**: Introduce mediator objects between interaction peers in order to provide indirection (phone operator)
  - **Memento**: Capture and externalize (without violating encapsulation!) an object's state so that it can be restored later

Behavioral Design Patterns

- More Examples:
  - **Observer**: defines and maintains dependencies between objects (e.g., notify certain objects when the state of some other objects has changed)
  - **State**: Encapsulate the state of an object in a separate object, so the object can change its behavior when its corresponding state object changes
  - **Strategy**: encapsulate an algorithm in an object and allow changing the algorithm at run-time (as shown earlier)
  - **Template method**: Abstract definition of an algorithm's interface. Subclass implements the specific algorithm. Alternative to Strategy pattern without separate class hierarchies for components (see particle motion example in week5/6)
  - **Visitor**: Encapsulate behavior that is distributed across classes
- Example task: particles move in space at irregular intervals, other particles may be affected by it => use Observer Pattern