Announcements

- Final exam: Friday, December 19, 14:00, at the GYM

  `http://www.mcgill.ca/student-records/exam/`

- Final exam tutorials

- Assignment 6 deadline: Saturday, December 6

Review

- Linked-lists: nodes with data and pointer to the next
- Traversing a linked-list

```
Node p = first;
while (p != null && ...) {
  //...
  p = p.get_next();
}
```

- Difference between a linked-list and an array
  - An array has a fixed size
  - A linked-list is dynamic: its size increases each time we add a new element and we don't have to worry about running out of space

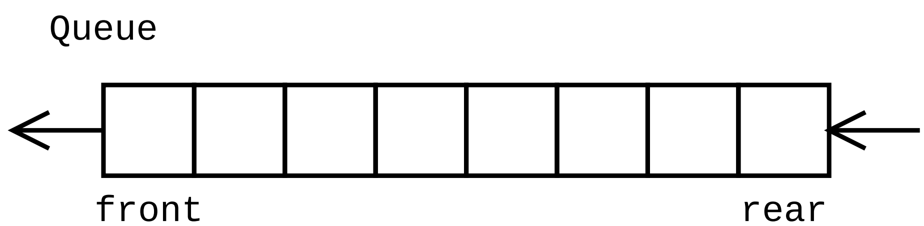
Queues and Stacks

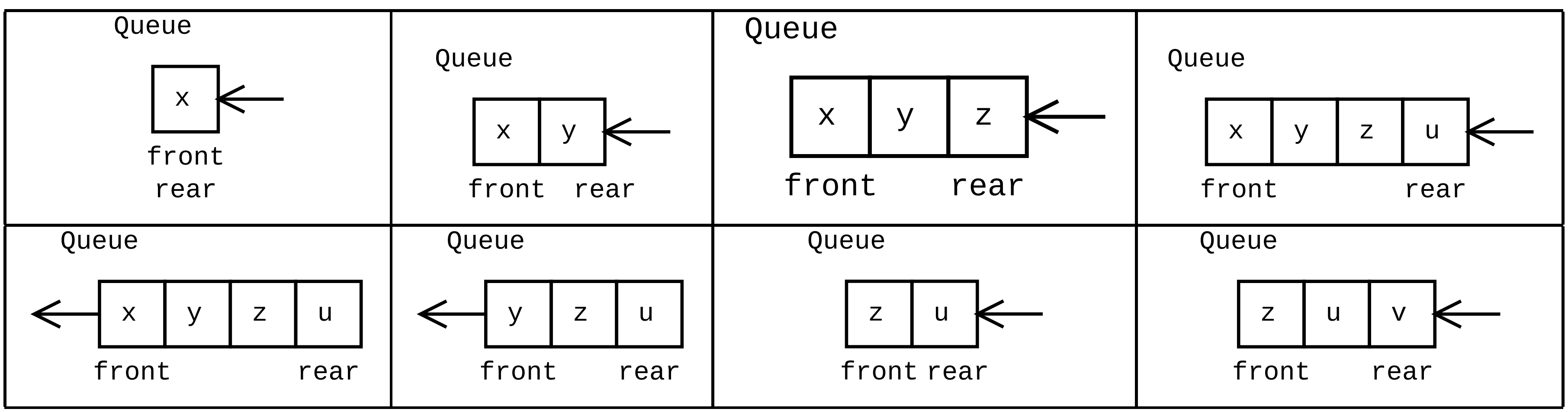
- Queues and Stacks are ADTs representing linear collections with particular operations

- A *queue* (FIFO) is a (dynamic) linear collection with (at least) the following operations:
  - *enqueue*: adds an item at the end of the sequence
  - *dequeue*: removes the first item of the sequence
  - *peek*: gets the first item of the sequence without removing it
  - *isempty*: returns true if the sequence has no items

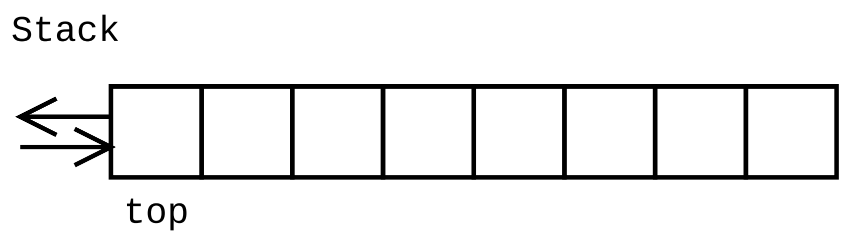
- A *stack* (LIFO, or FILO) is a (dynamic) linear collection with (at least) the following operations:
  - *push*: adds an item at the "top" of the sequence
  - *pop*: removes the "top" item of the sequence
  - *top*: returns the top item without removing it
  - *isempty*: returns true if the sequence has no items

Queues

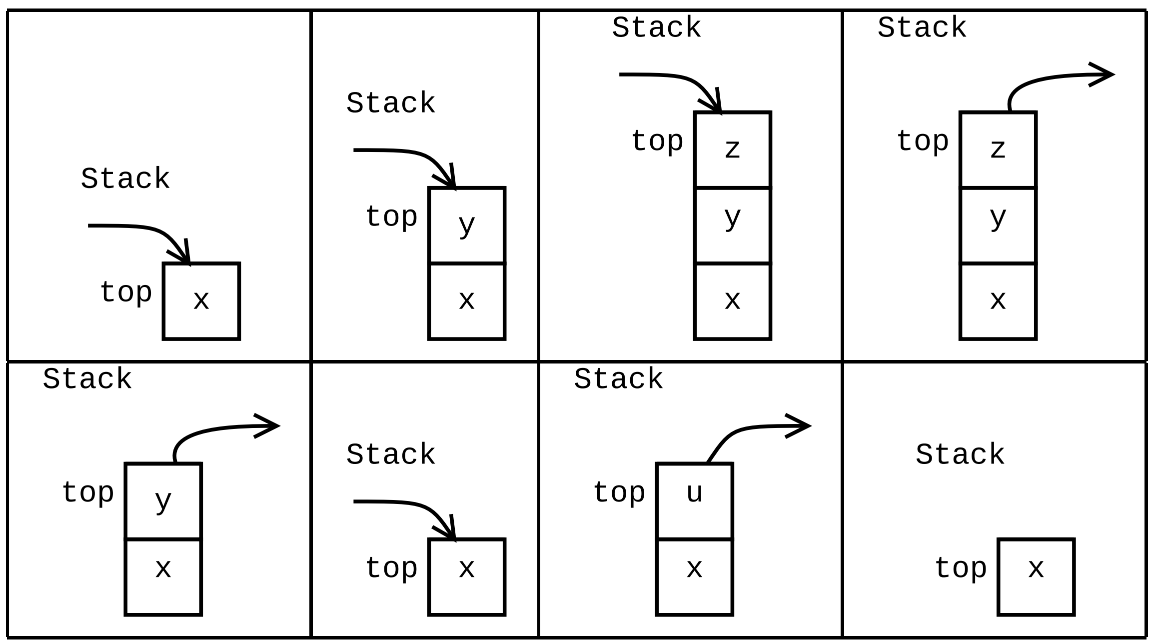

Queue
front
rear


Queue
x
front
rear
Queue
x y
front rear
Queue
x y z
front rear
Queue
x y z u
front rear
Queue
x y z u
front rear
Queue
y z u
front rear
Queue
z u
front rear
Queue
z u v
front rear

Stacks



Stack
top



Stack
top
x
Stack
top
y
x
Stack
top
z
y
x
Stack
top
z
y
x
Stack
top
y
x
Stack
top
x
Stack
top
u
x
Stack
top
x

Queues and Stacks

- Queues and Stacks are ADTs so they can be implemented in many different ways
- For example, they could be implemented using
  - Linked-lists
  - Arrays
  - Doubly-linked-lists
  - Linked-lists with front and rear pointers
  - Vectors/Growing arrays
  - "Circular" arrays
- Each implementation has advantages and disadvantages with respect to efficiency
- Queues implemented as fixed-size circular arrays are commonly called *buffers*

Implementing Queues

```
class LinkedList {
  //...
  public LinkedList() { ... }
  public void insert_at(Object o, int index) { ...
  public void remove_at(int index) { ... }
  public Object element_at(int index) { ... }
  public int length() { ... }
}
```

Implementing Queues

```
class Queue {
  private LinkedList list;
  public Queue() { list = new LinkedList(); }
  public void enqueue(Object obj)
  {
    list.insert_at(obj, list.length());
  }
  public void dequeue()
  {
    list.remove_at(0);
  }
  public Object peek()
  {
    return list.element_at(0);
  }
  public boolean isempty()
  {
    return list.length() == 0;
  }
}
```

Implementing Queues

```
class Queue {
  private LinkedList list;
  public Queue() { list = new LinkedList(); }
  public void enqueue(Object obj)
  {
    list.insert_at(obj, 0);
  }
  public void dequeue()
  {
    list.remove_at(list.length());
  }
  public Object peek()
  {
    return list.element_at(list.length());
  }
  public boolean isempty()
  {
    return list.length() == 0;
  }
}
```

Implementing Queues

```
class Queue {
  private LinkedList list;
  public Queue() { list = new LinkedList(); }
  public void enqueue(Object obj)
  {
    list.add_at_end(obj);
  }
  public void dequeue()
  {
    list.remove_first();
  }
  public Object peek()
  {
    return list.element_at(0);
  }
  public boolean isempty()
  {
    return list.length() == 0;
  }
}
```

Implementing Stacks

```
class Stack {
  private LinkedList list;
  public Stack() { list = new LinkedList(); }
  public void push(Object obj)
  {
    list.insert_at(obj, 0);
  }
  public void pop()
  {
    list.remove_at(0);
  }
  public Object top()
  {
    return list.element_at(0);
  }
  public boolean isempty()
  {
    return list.length() == 0;
  }
}
```

Implementing Stacks

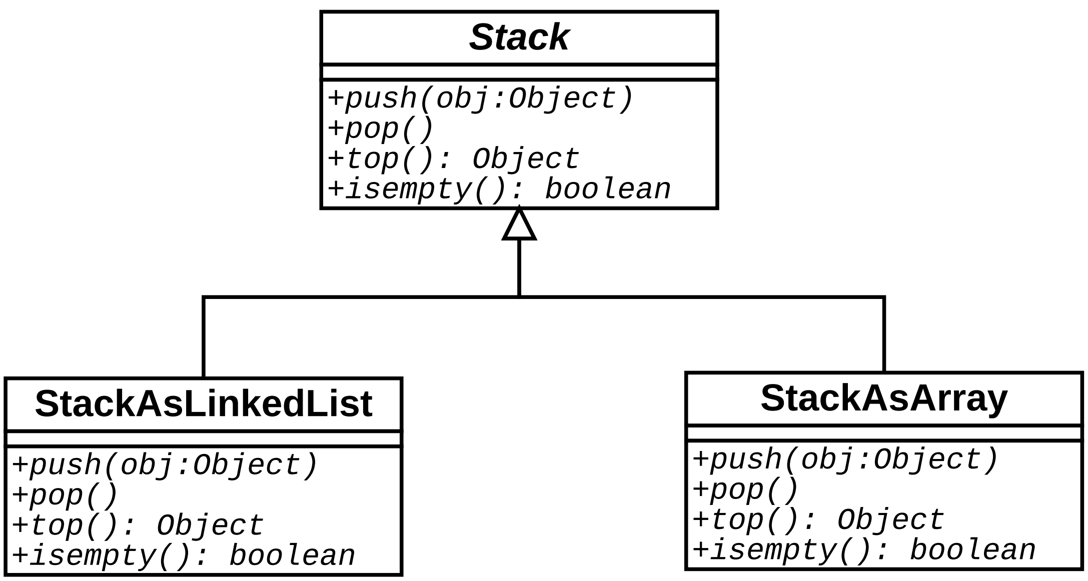
```
class Stack {
  private Object[] list;
  private int top;

  public Stack()
  {
    list = new Object[1000];
    top = 0;
  }
  public void push(Object obj)
  {
    if (top >= list.length)
      grow_array(100);
    list[top] = obj;
    top++;
  }
```

Implementing Stacks

```
  public void pop()
  {
    top--;
  }
  public Object top()
  {
    return list[top];
  }
  public boolean isempty()
  {
    return top == 0;
  }
  private void grow_array(int n)
  {
    ...
  }
} // End of Stack
```

ADTs and abstract classes

- Stacks and queues constrain the operations on a list
- An ADT should be declared as an interface or abstract class, and the concrete implementations should implement the interface or extend the abstract class



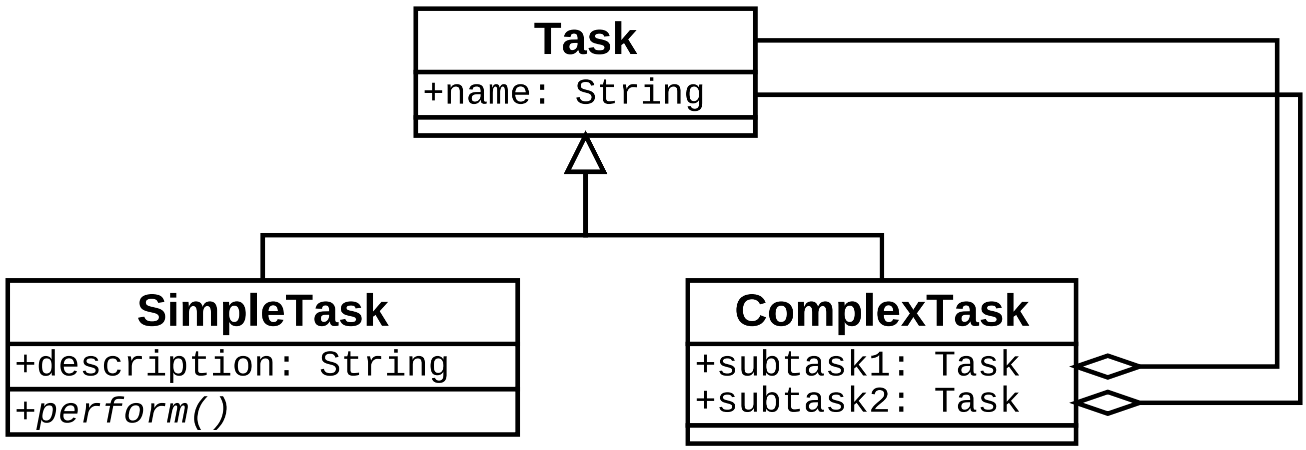
Applications (Simulation)

```
class Customer { ... }
class SuperMarket {
  Queue line;
  SuperMarket() { line = new Queue(); }
  void process(Customer c) { ... }
  void run()
  {
    while (true) {
      int coin = (int)(Math.random() * 2);
      if (coin == 1) {
        Customer first = line.peek();
        process(first);
        line.dequeue();
      }
      else {
        line.enqueue(new Customer());
      }
    }
  }
}
```

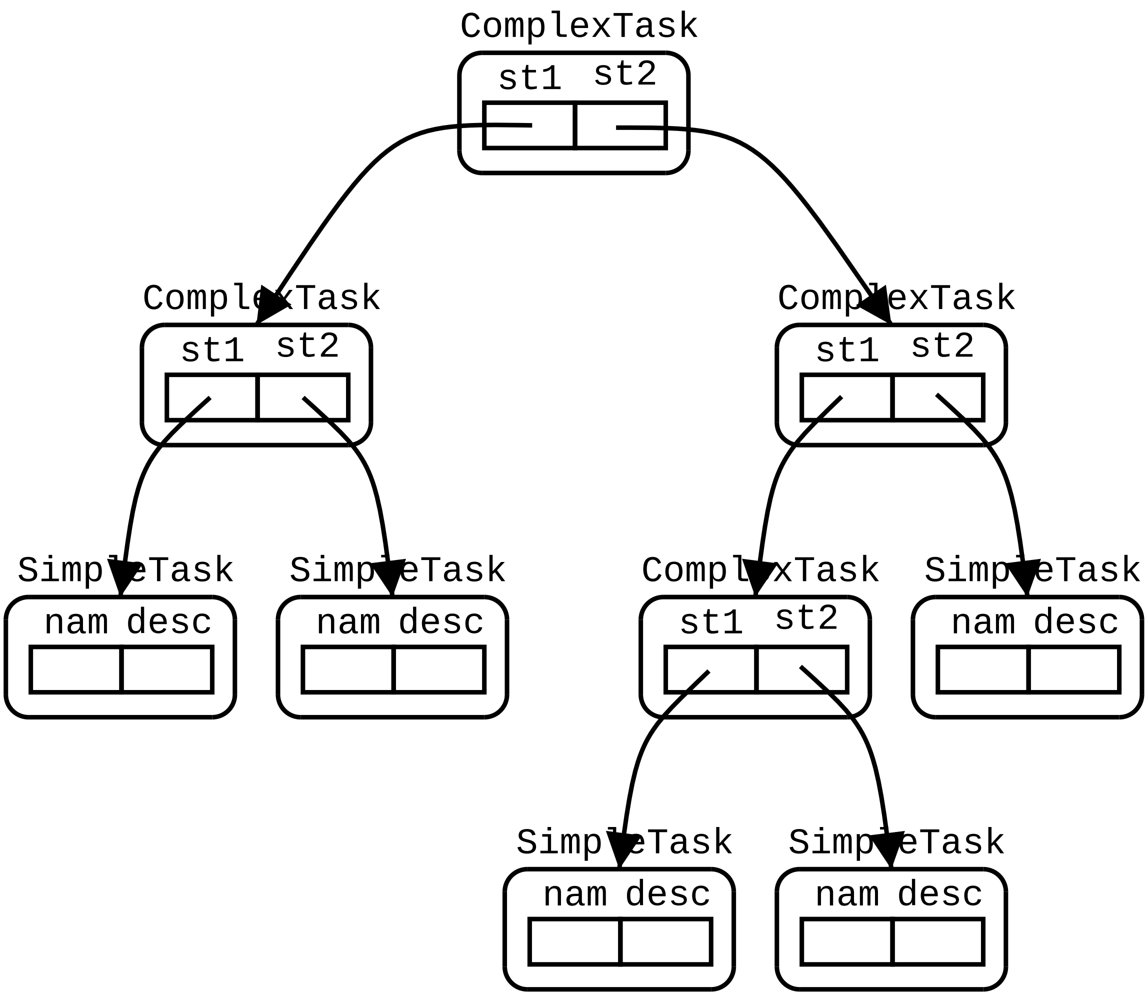
Applications (reverse)

```
static String reverse(String s)
{
  String r = "";
  Stack stack = new Stack();
  int i = 0;
  while (i < s.length()) {
    stack.push(new Character(s.charAt(i)));
    i++;
  }
  while (!stack.isempty()) {
    Character c = (Character)stack.top();
    r = r + c.charValue();
    stack.pop();
  }
  return r;
}
```

Binary Trees



Binary Trees



Binary Trees

```
abstract class Task {
  String name;
}

class SimpleTask extends Task {
  String description;
  void perform()
  {
    System.out.println(name+“:”+description);
    //...
  }
}

class ComplexTask extends Task {
  Task subtask1, subtask2;
}
```

Binary Trees

- Processing trees using recursion

```
class Worker {
  void work(Task t)
  {
    if (t instanceof SimpleTask) {
      ((SimpleTask)t).perform();
    }
    else if (t instanceof ComplexTask) {
      work(((ComplexTask)t).subtask1);
      work(((ComplexTask)t).subtask2);
    }
  }
}
```

Binary Trees

- Processing trees using stacks

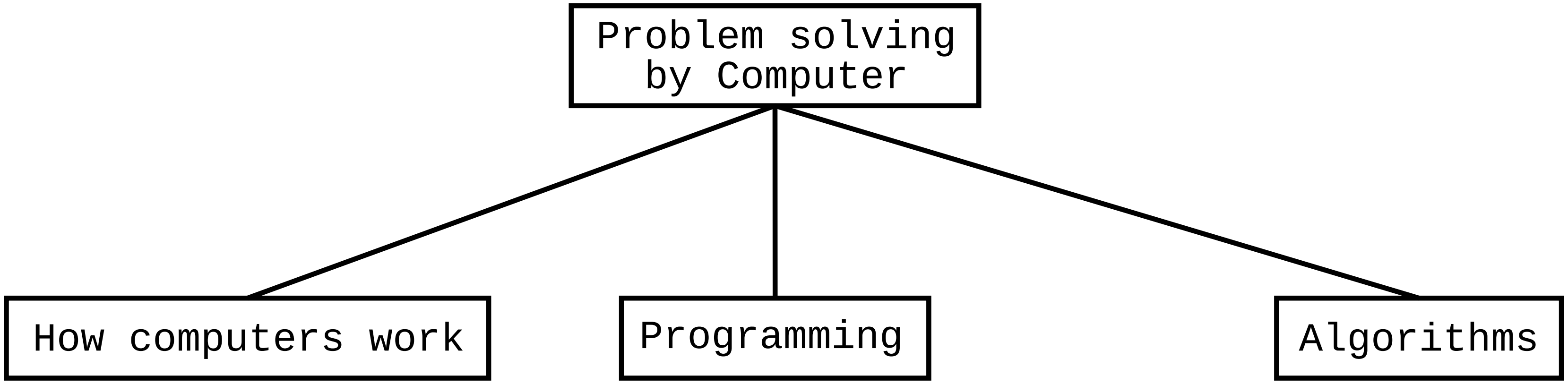
```
class Worker {
  void work(Task t)
  {
    Stack s = new Stack();
    s.push(t);
    while (!s.isempty()) {
      Task temp = s.top();
      s.pop();
      if (temp instanceof SimpleTask) {
        ((SimpleTask)t).perform();
      }
      else {
        s.push(((ComplexTask)temp).subtask2);
        s.push(((ComplexTask)temp).subtask1);
      }
    }
  }
}
```

Data structures zoo

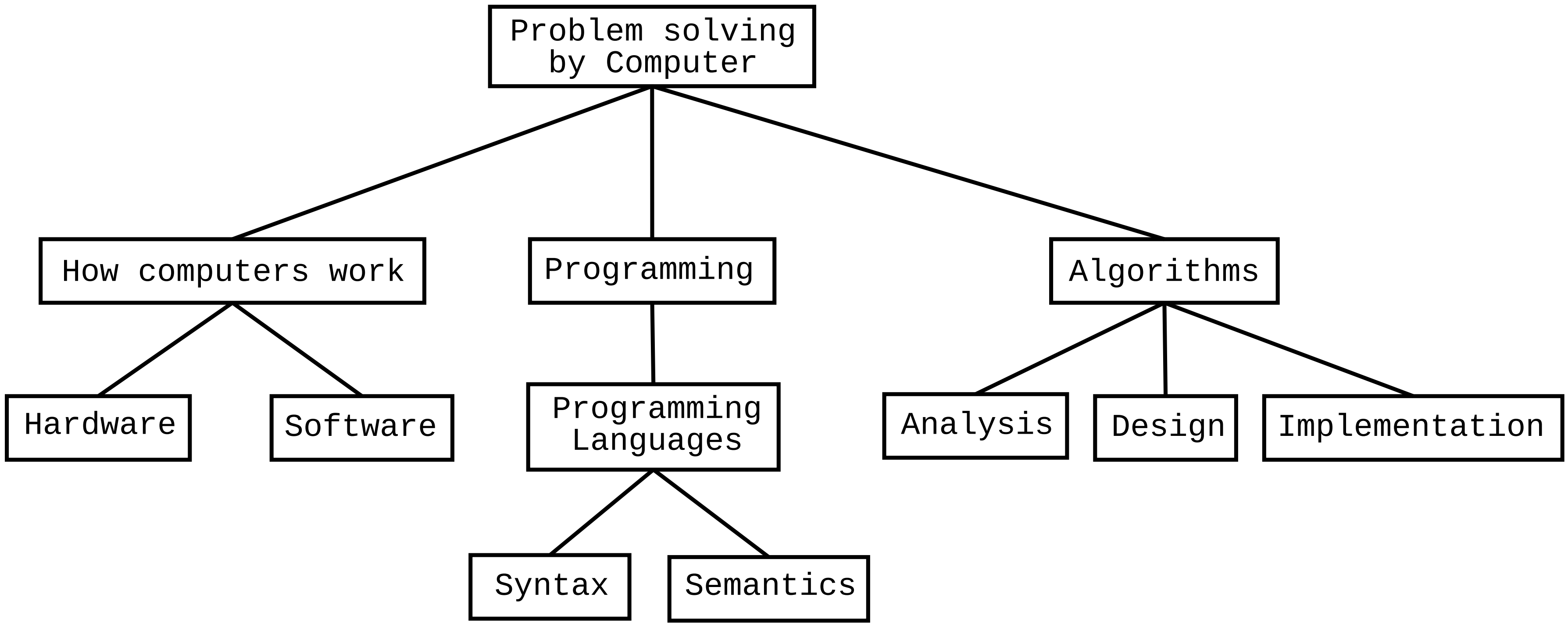
- Other data-structures: sets, bags, priority queues, heaps, binary trees, n-ary trees, red-black trees, AVL trees, graphs, hyper-graphs, hi-graphs, dictionaries/mappings, etc.

- The selection of data-structure has a major impact on the efficiency of an algorithm.

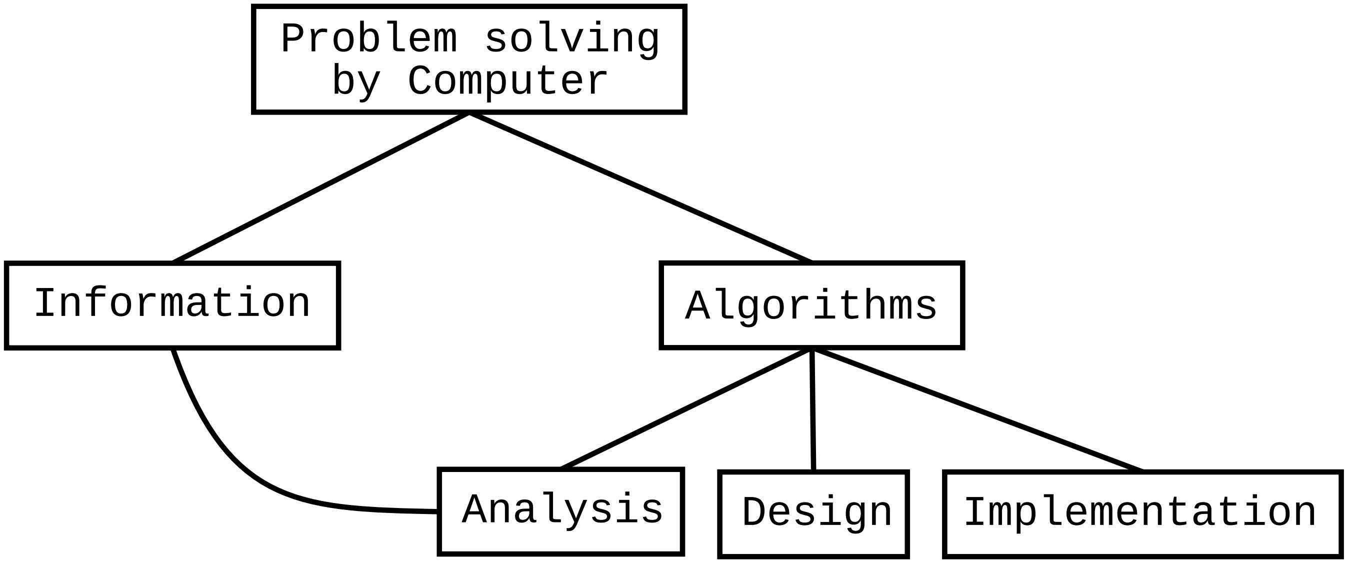
The big picture



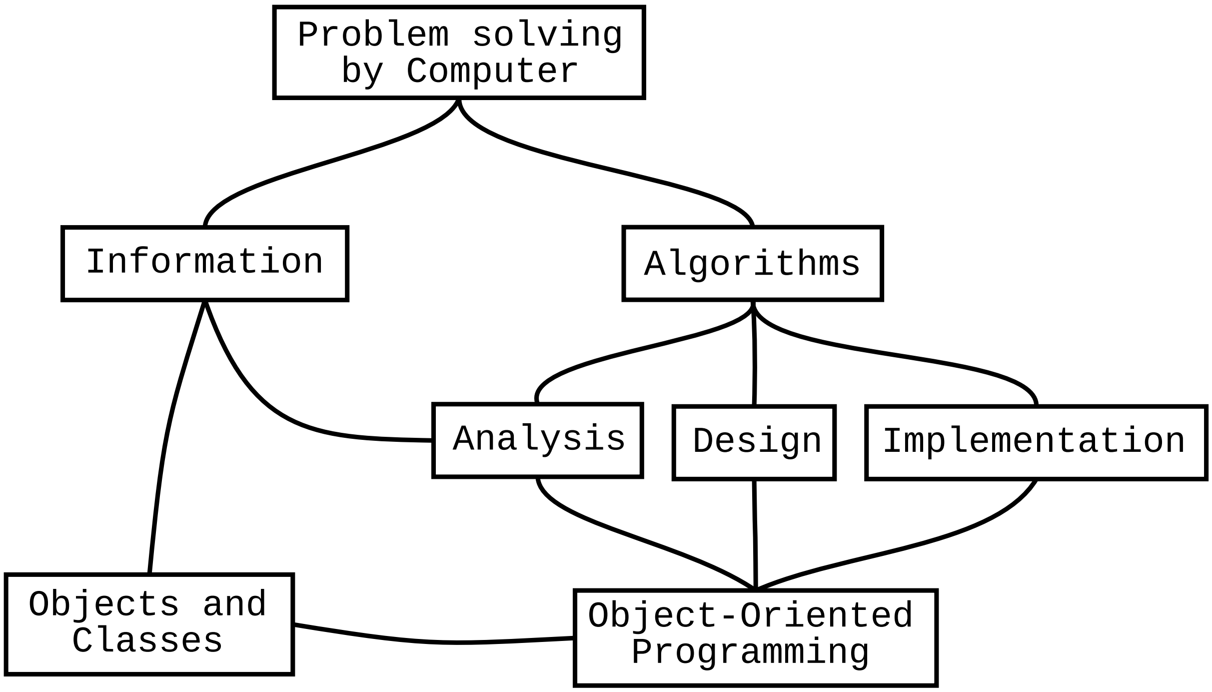
The big picture



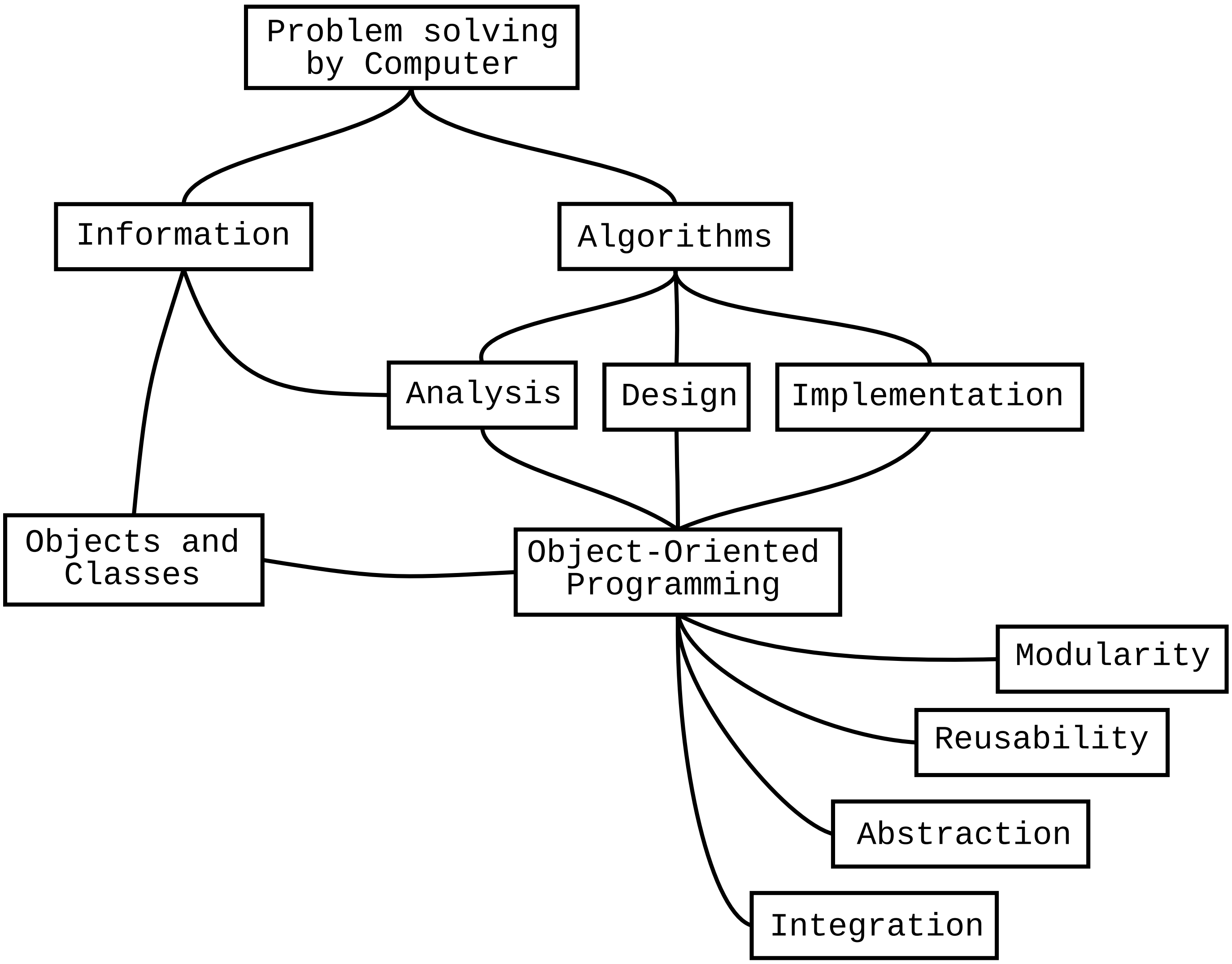
The big picture



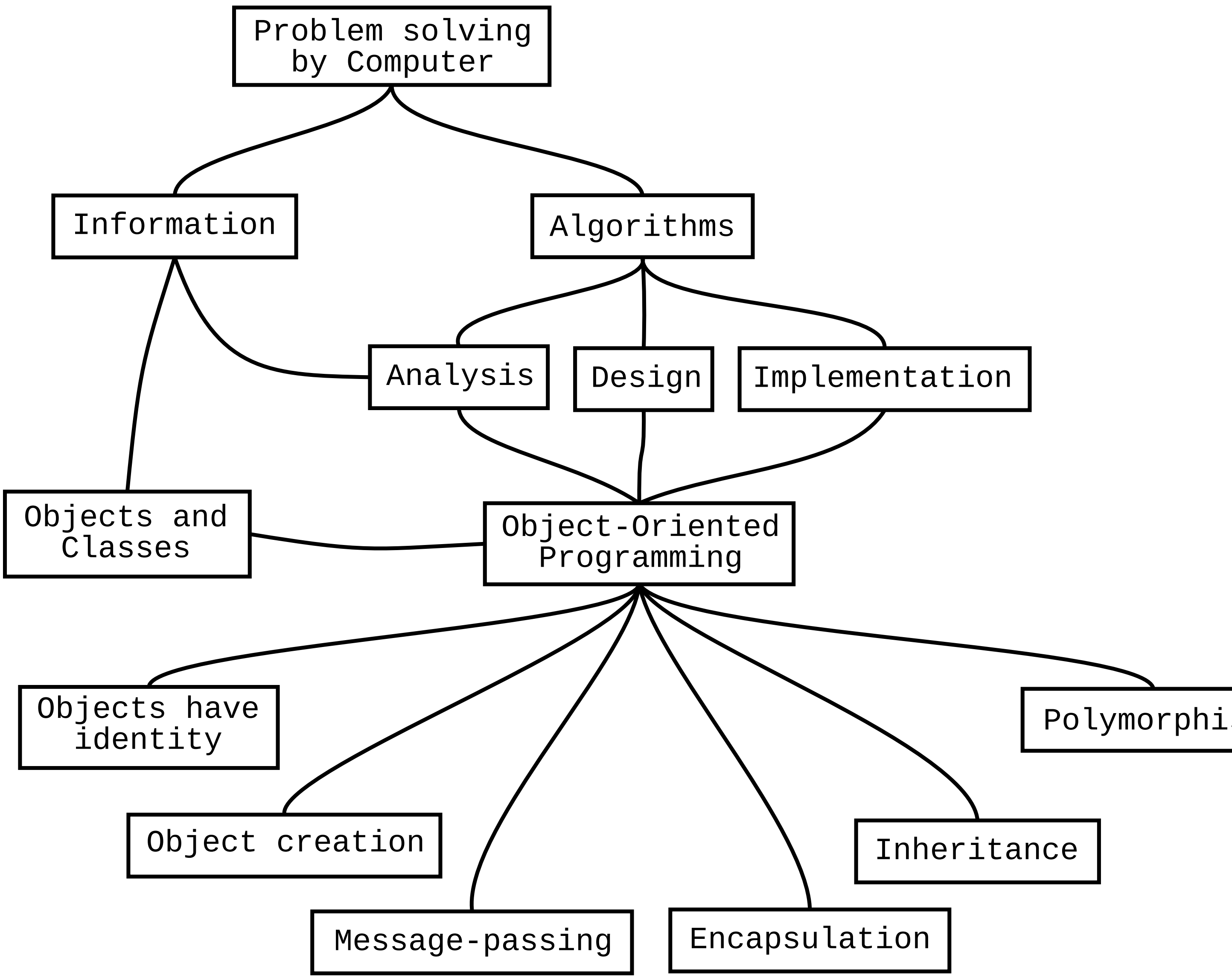
The big picture



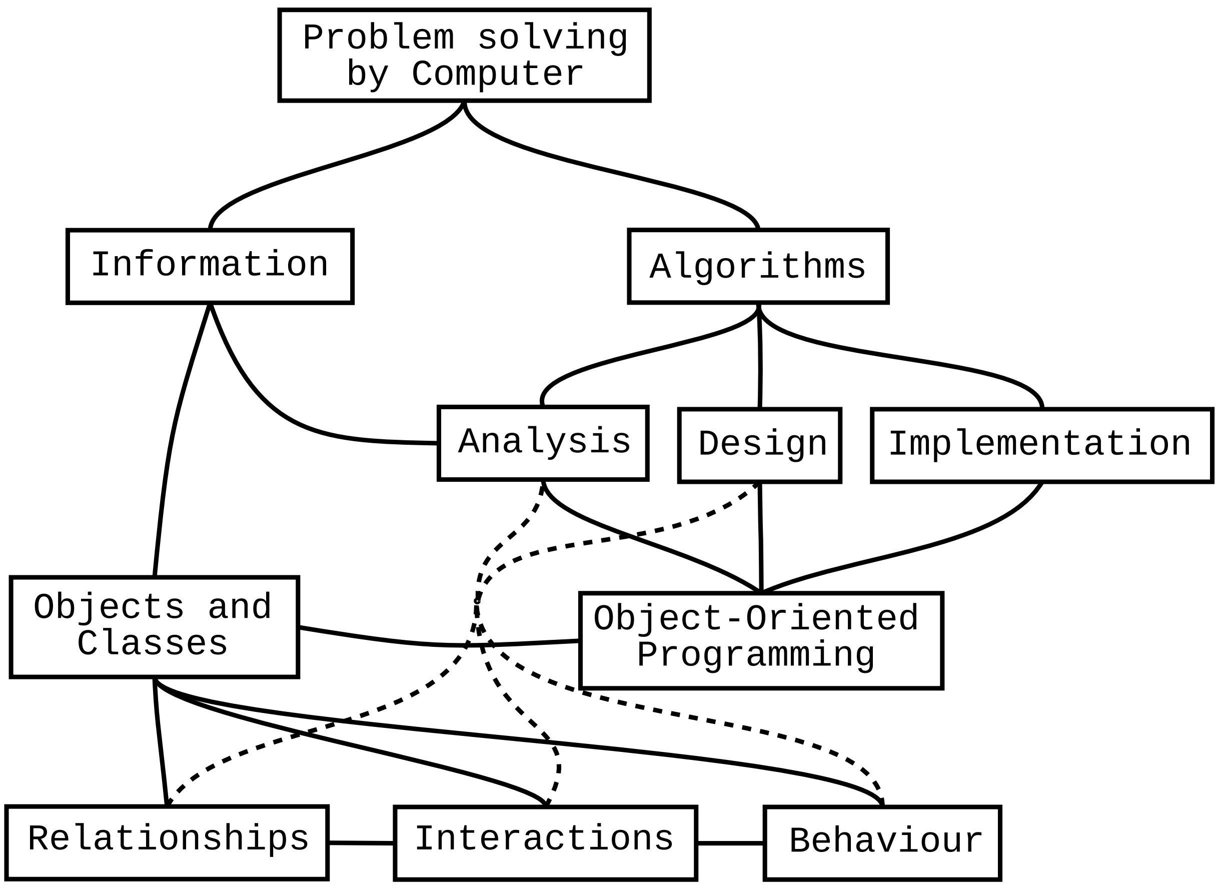
The big picture



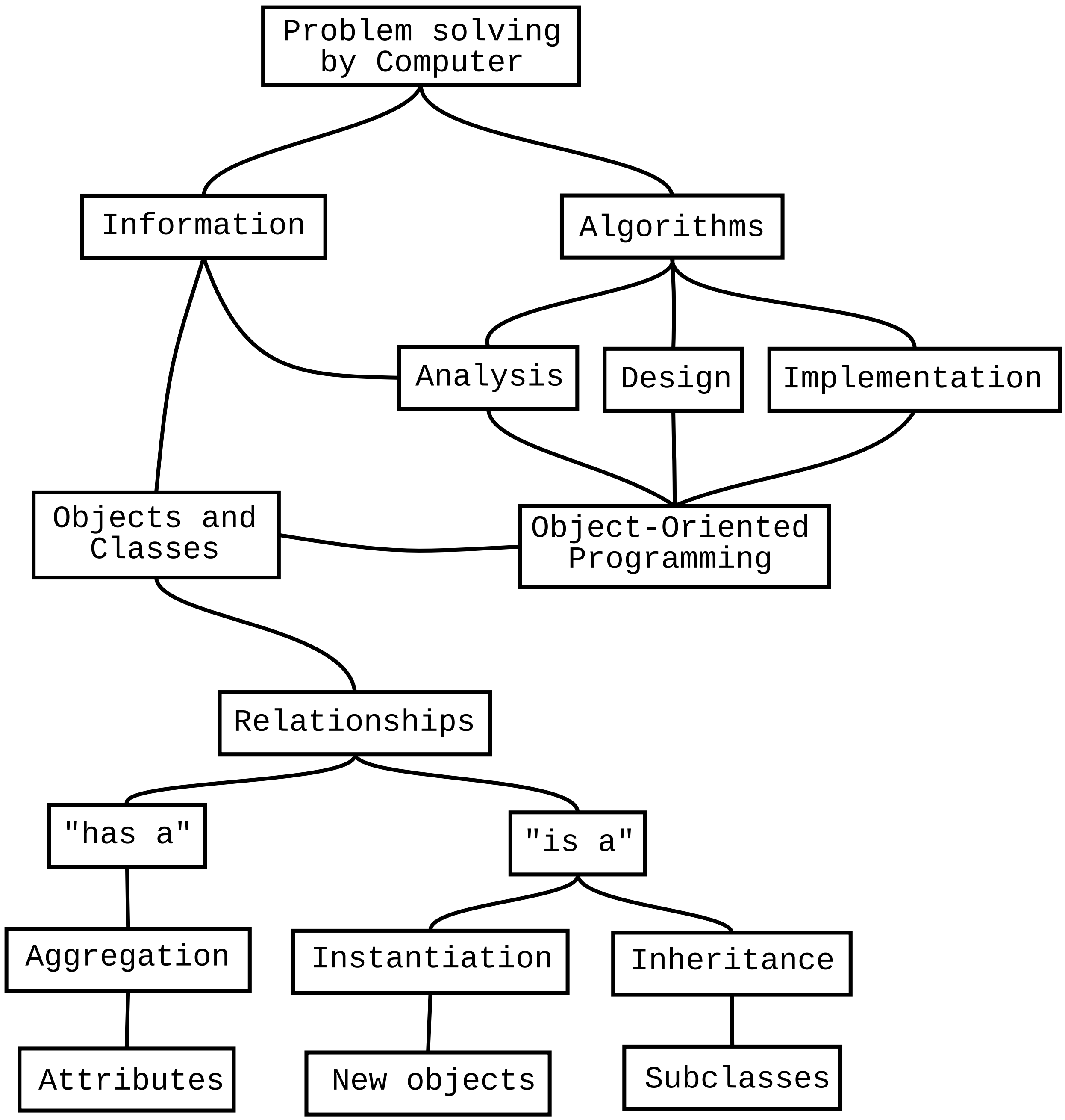
The big picture



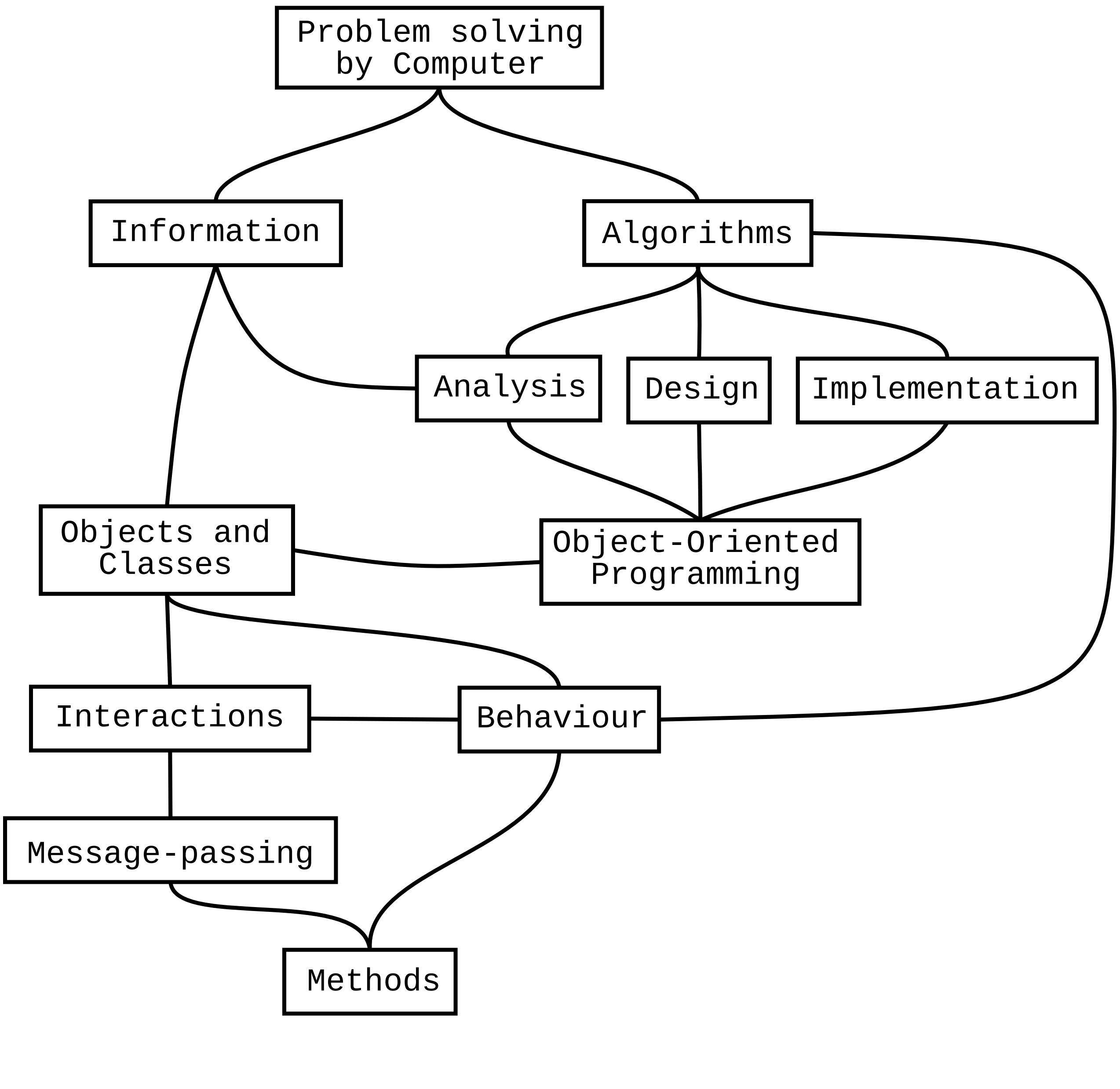
The big picture



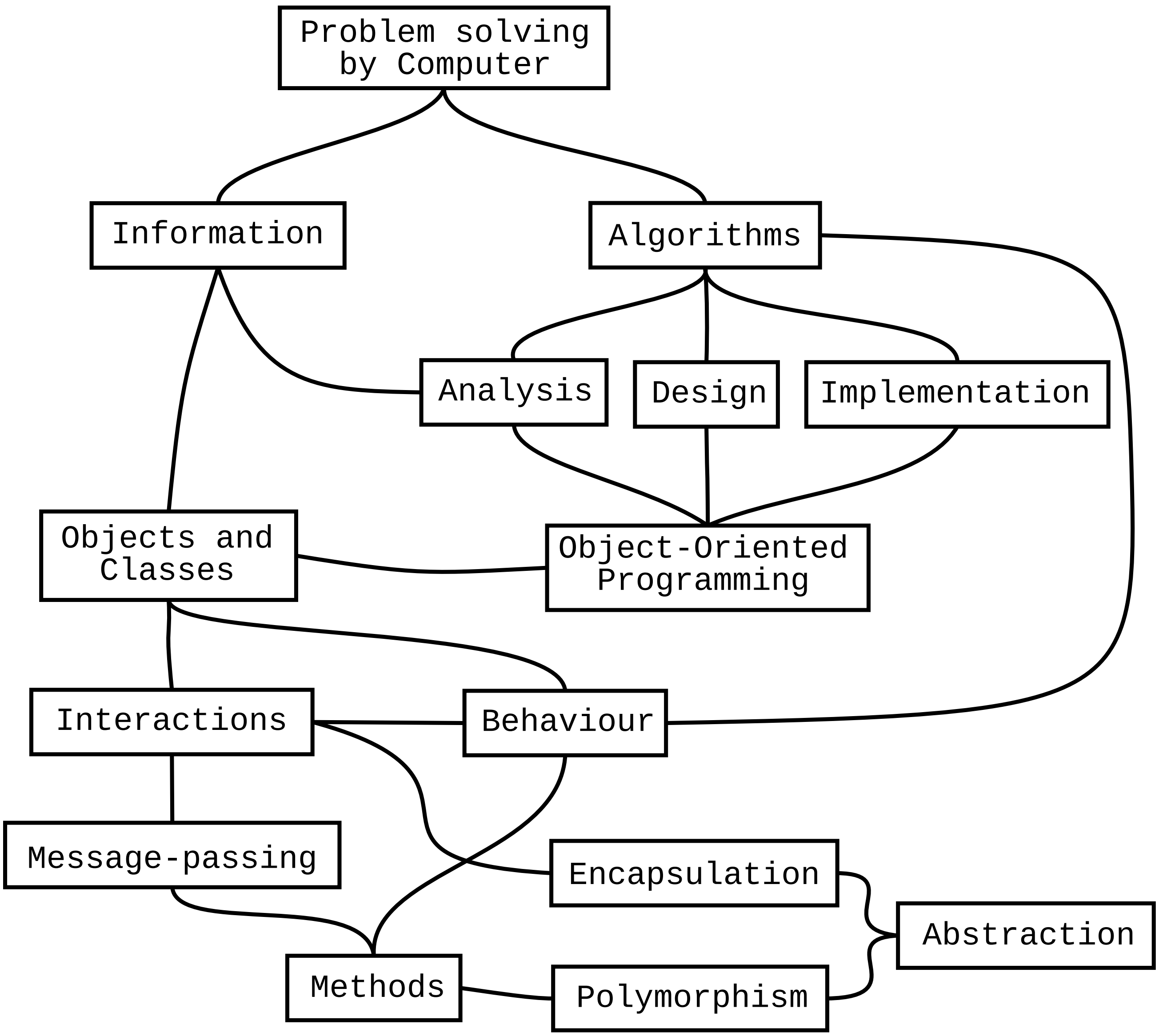
The big picture



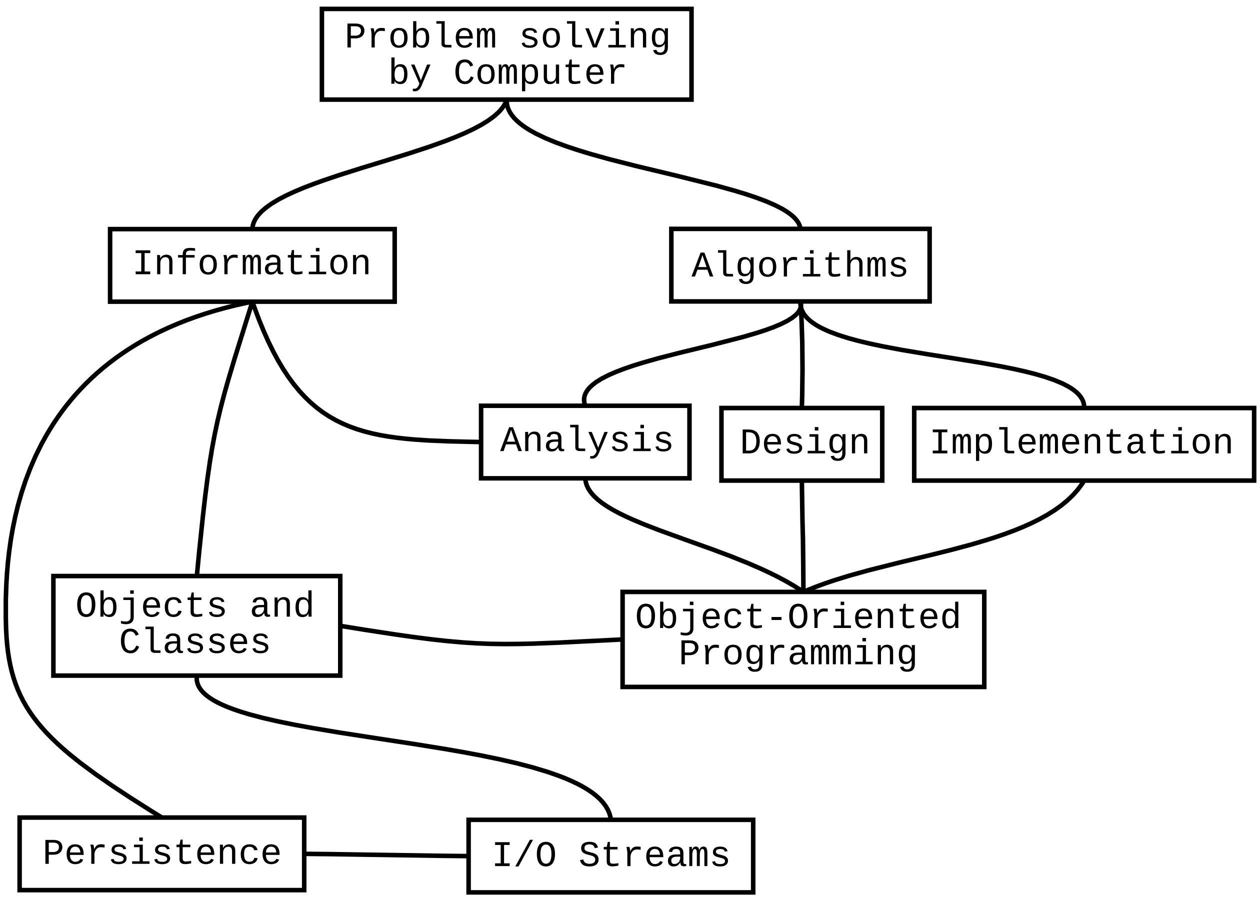
The big picture



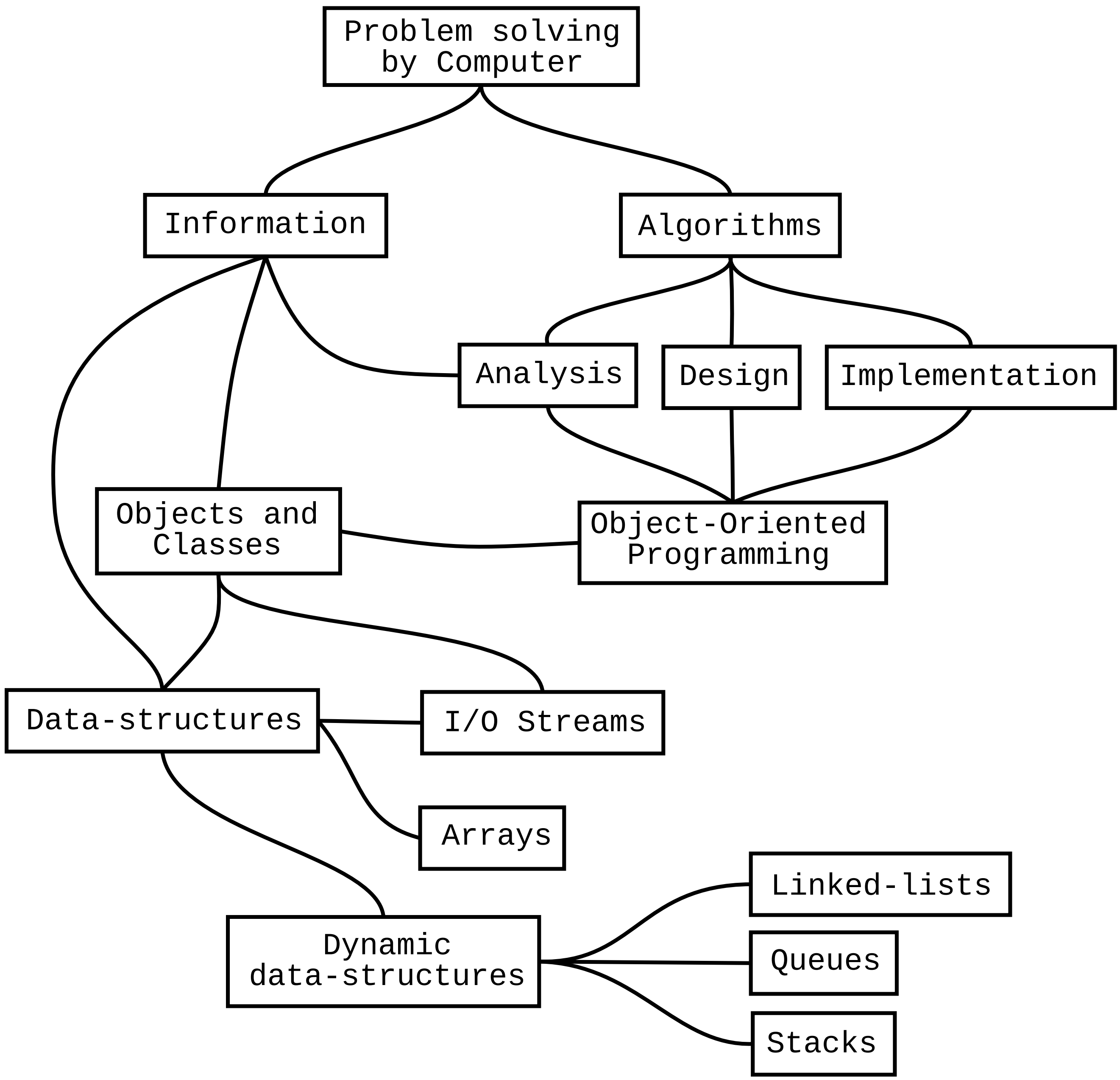
The big picture



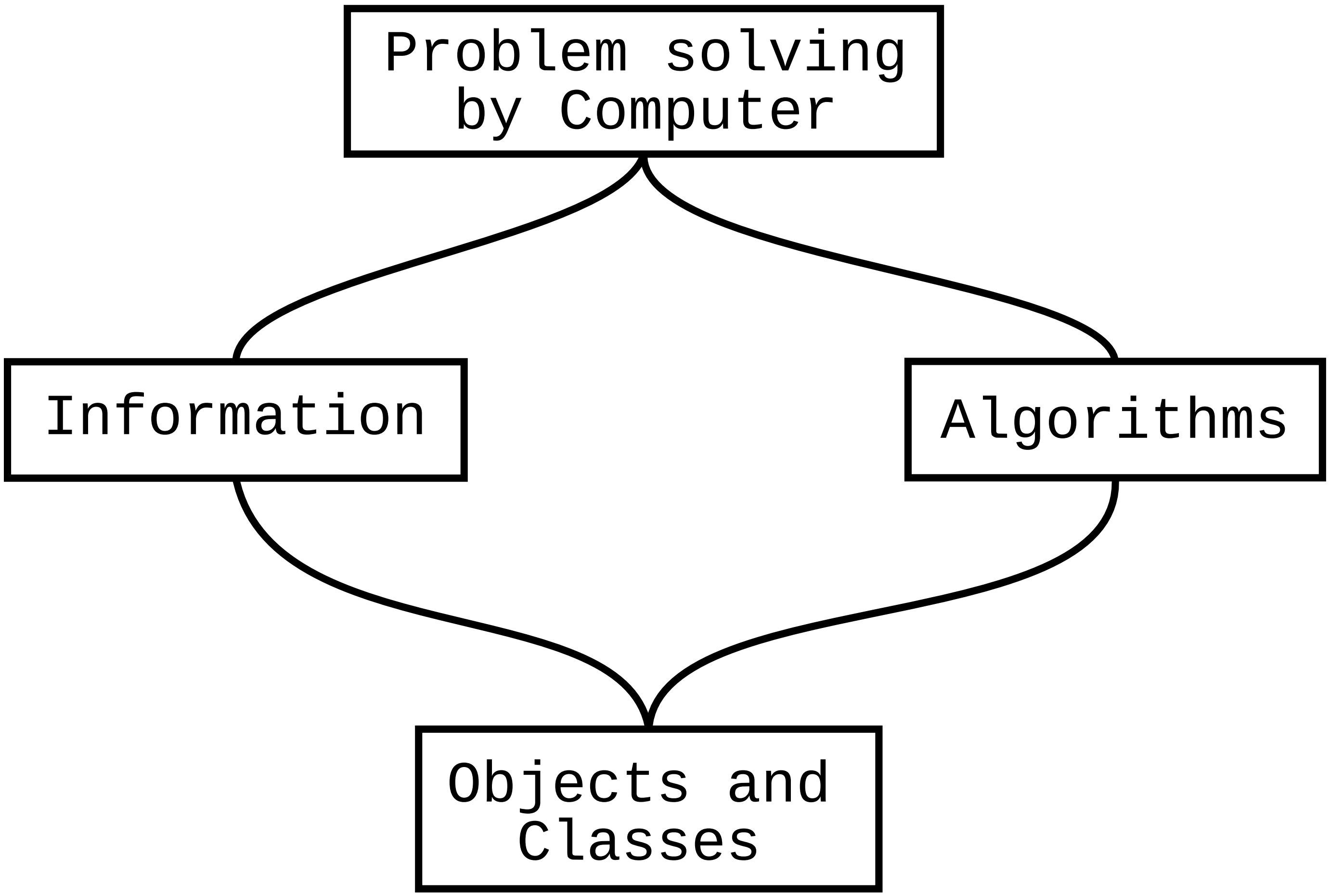
The big picture



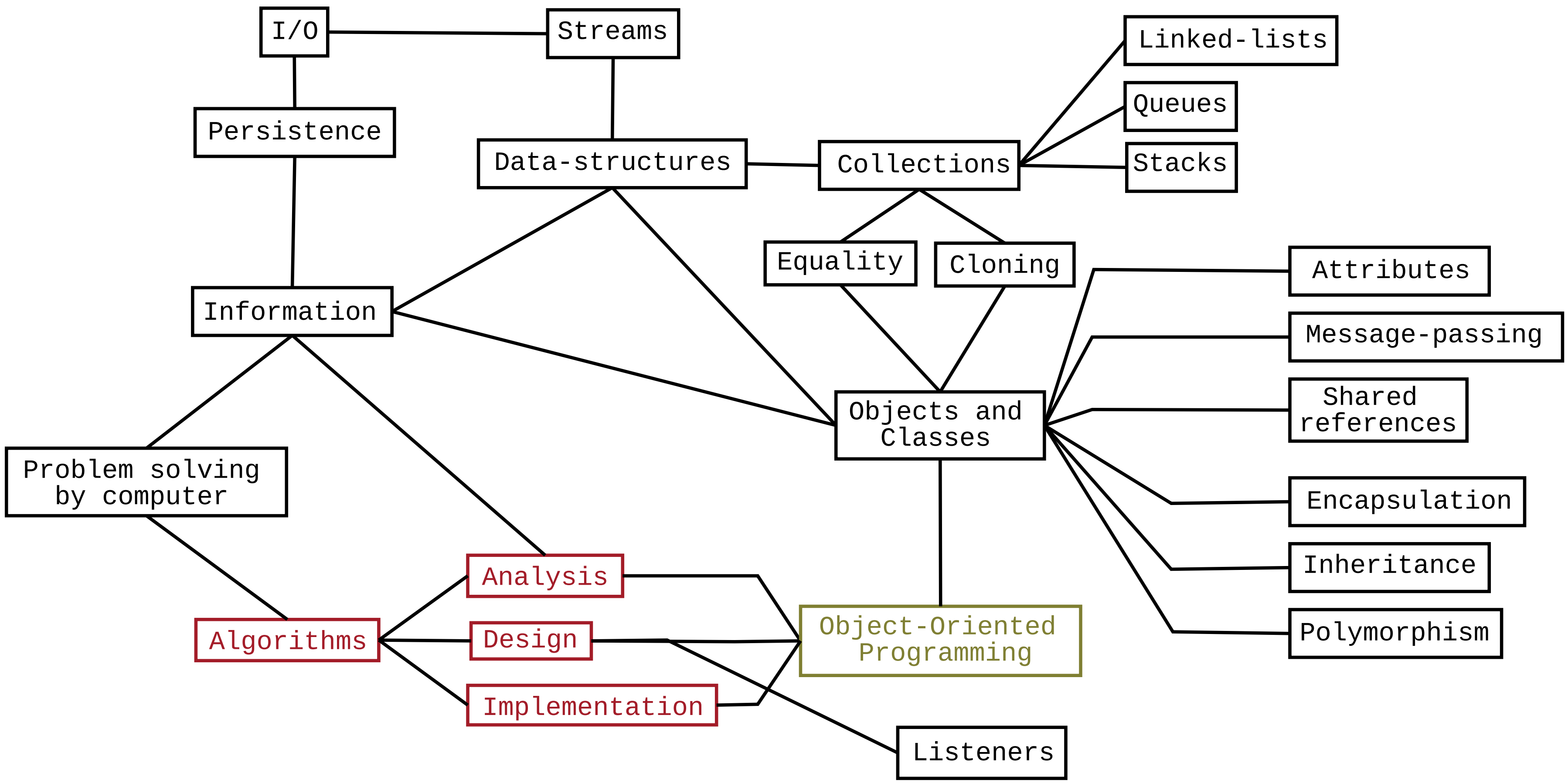
The big picture



The big picture



The big picture



McGill