Object-Oriented Programming

What does that even mean?

Designing object-oriented software is hard, and
designing reusable object-oriented software is even harder.

— GoF, *Design Patterns* (1994)

Mathias Laurin

https://github.com/Synss

Object-oriented programming

- Programming paradigms?
- History and flavors of OOP: Simula, Smalltalk, and Eiffel
- On storing state with procedures: objects and abstract data types
- A definition in seven words
- OOP in C++ with examples

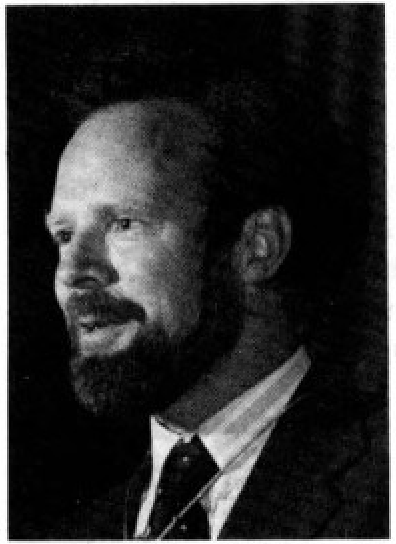
some early work on compiler optimization. Subsequently, in the	in its capabilities.

Programming paradigm

" The archetype, paradigm, exemplar, and idea, according to which all things were made.

The Paradigms of Programming

Robert W. Floyd
Stanford University



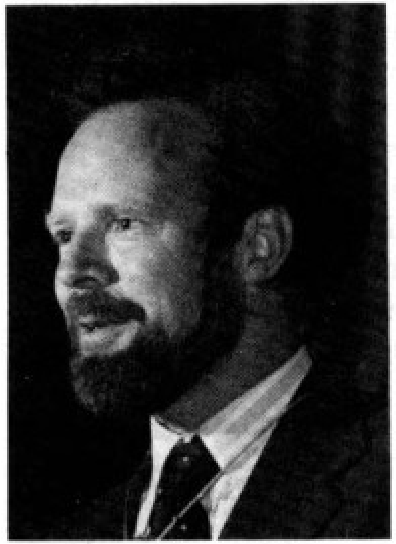
Paradigm(pæ·radim, —dəim) . . . [a. F. *paradigme*, ad. L. *paradigma*, a. Gr. παραδειγμα pattern, example, f. παραδεικνυ·ναι to exhibit beside, show side by side. . .]
1. A pattern, exemplar, example.
1752 J. Gill *Trinity* v. 91
The archetype, paradigm, exemplar, and idea, according to which all things were made.
From the Oxford English Dictionary.

Today I want to talk about the paradigms of programming, how they affect our success as designers of computer programs, how they should be taught, and how they should be embodied in our programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology. Structured programming, as formulated by Dijkstra [6], Wirth [27, 29], and Parnas [21], among others, consists of two phases.

In the first phase, that of top-down design, or stepwise refinement, the problem is decomposed into a very small number of simpler subproblems. In programming the solution of simultaneous linear equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-substitution in the triangularized system. This gradual decomposition is continued until the subproblems that arise are simple enough to cope with directly. In the simultaneous equation example, the back substitution process would be further decomposed as a backwards iteration of a process which finds and stores the value of the *i*th variable from the *i*th equation. Yet further decomposition would yield a fully detailed algorithm.

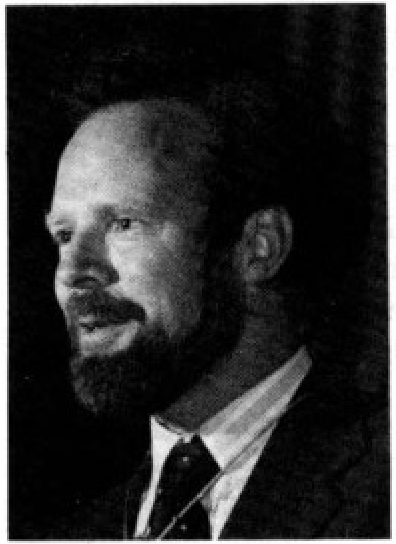
Language support



“ When a language makes a paradigm convenient, I will say that the language *supports* the paradigm. When a language makes a paradigm feasible, but not convenient, I will say the language *weakly supports* the paradigm.

— R. W. Floyd , *The Paradigms of Programming*, 1978 ACM Turing Award Lecture

Language support



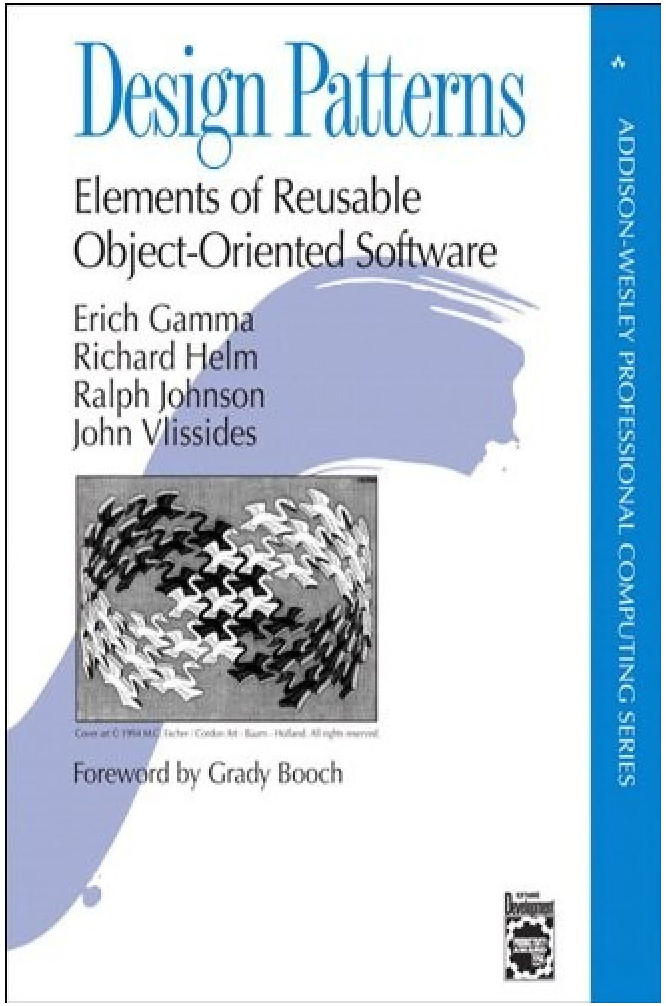
“ When a language makes a paradigm convenient, I will say that the language *supports* the paradigm. When a language makes a paradigm feasible, but not convenient, I will say the language *weakly supports* the paradigm.

— R. W. Floyd , *The Paradigms of Programming*,
1978 ACM Turing Award Lecture

The OO programming paradigm

“ The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily.
If we assumed procedural languages, we might have included design patterns called “Inheritance”, “Encapsulation”, and “Polymorphism.”

— GoF, *Design Patterns* (1994)



OOPL vs. OOP

Object-oriented programming language

- Requirement on the language:
  the language is implemented with objects.
- Not a requirement on the user.

Object-oriented style

- Requirement on the program:
  the program is implemented with objects.
- Not a requirement on the programming language:
  language agnostic.

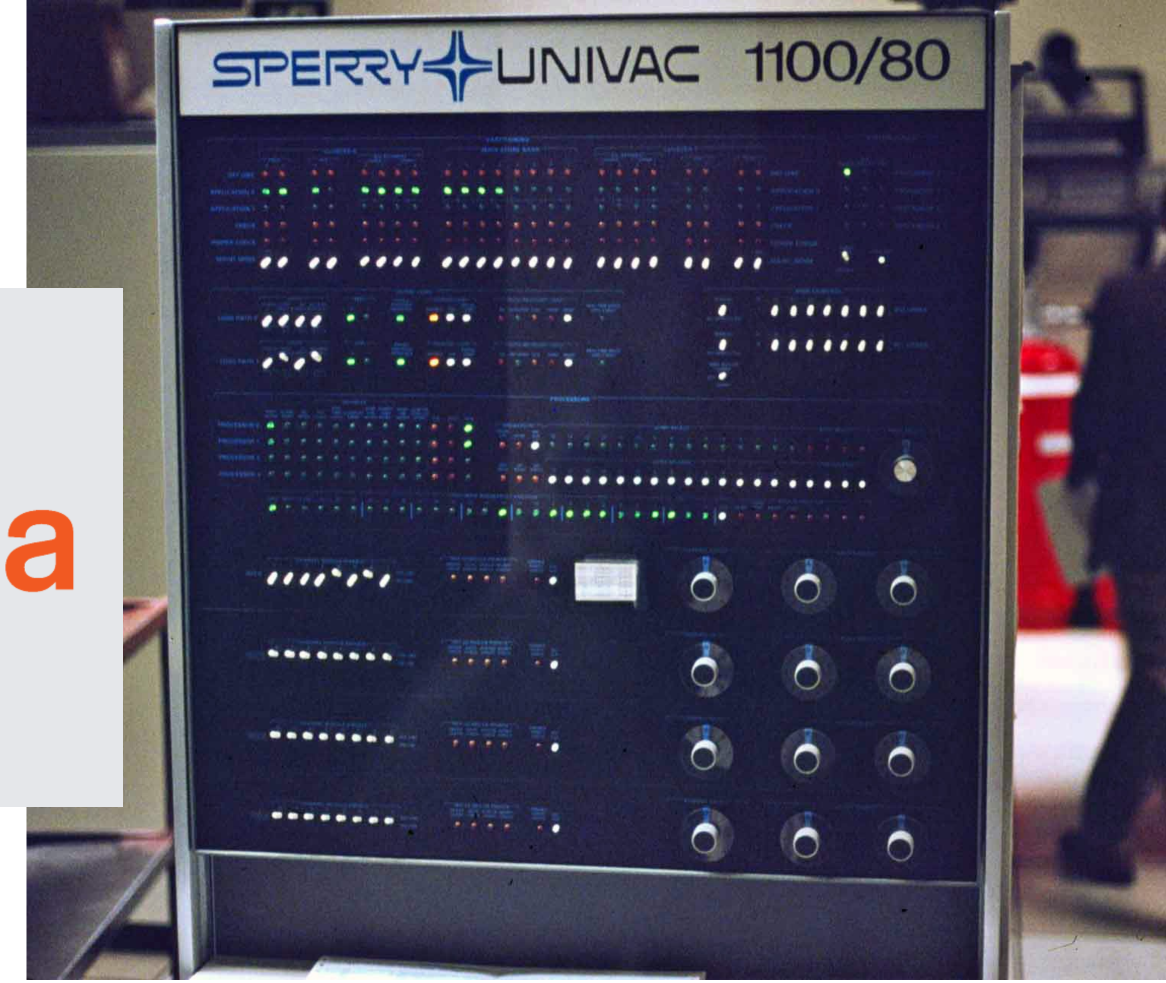
```
(define-syntax <<p>>
  (extend-shadow <<o>> (x y)
    ([init
       (method (x^ y^)
         (set! x x^)
         (set! y y^)
         (init super))]
     [move
       (method (dx dy)
         (set! x (+ x dx))
         (set! y (+ y dy)))]
     [get-loc
       (method ()
         (list x y))]
     [diag
       (method (a)
         (move it a a))])))

(define <p> (create-class <<p>> <o>))
```

OO in Scheme? → Daniel P. Friedman, *Object-Oriented Style* (1999)

History

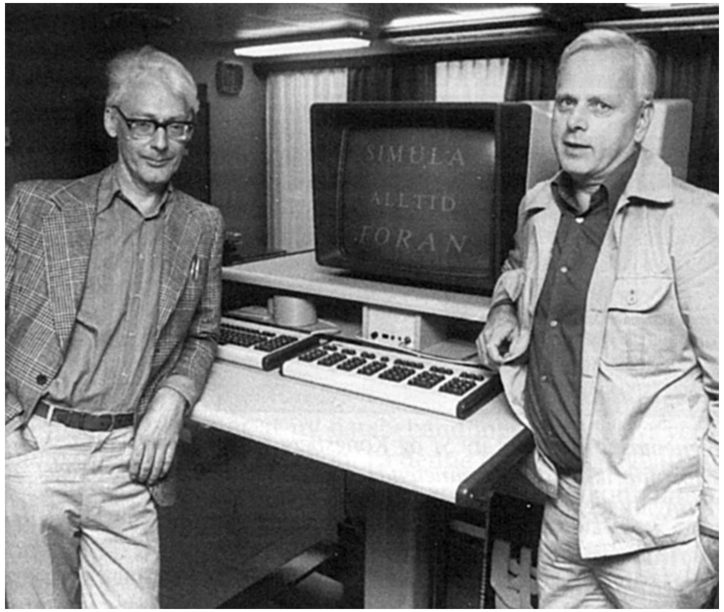
simula



Eiffel

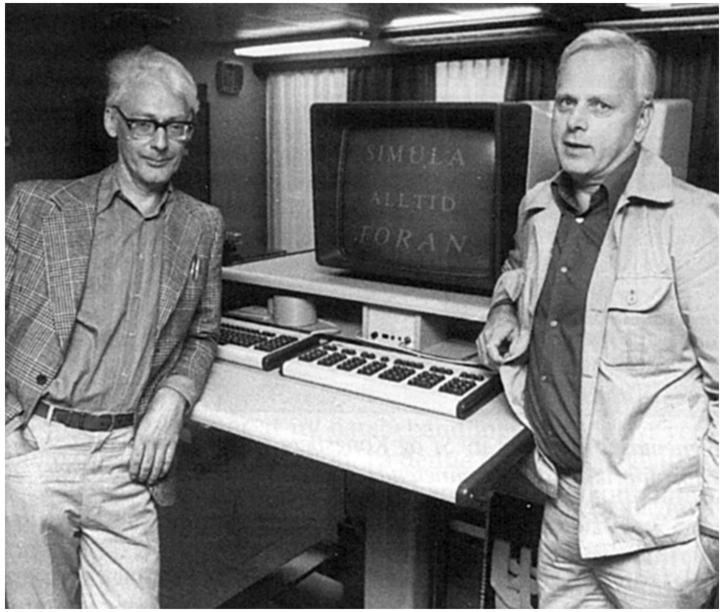
Origins: Simula

- Ole-Johan Dahl (1931–2002) and
  Kristen Nygaard (1926–2002)
  Turing Award 2001
  Dahl–Nygaard Prize for researchers since 2005
- Norwegian Computing Center
- Discrete event simulation
- Objects, classes, inheritance, subclasses, virtual procedures, coroutines…
- 1962? (1965)–1967



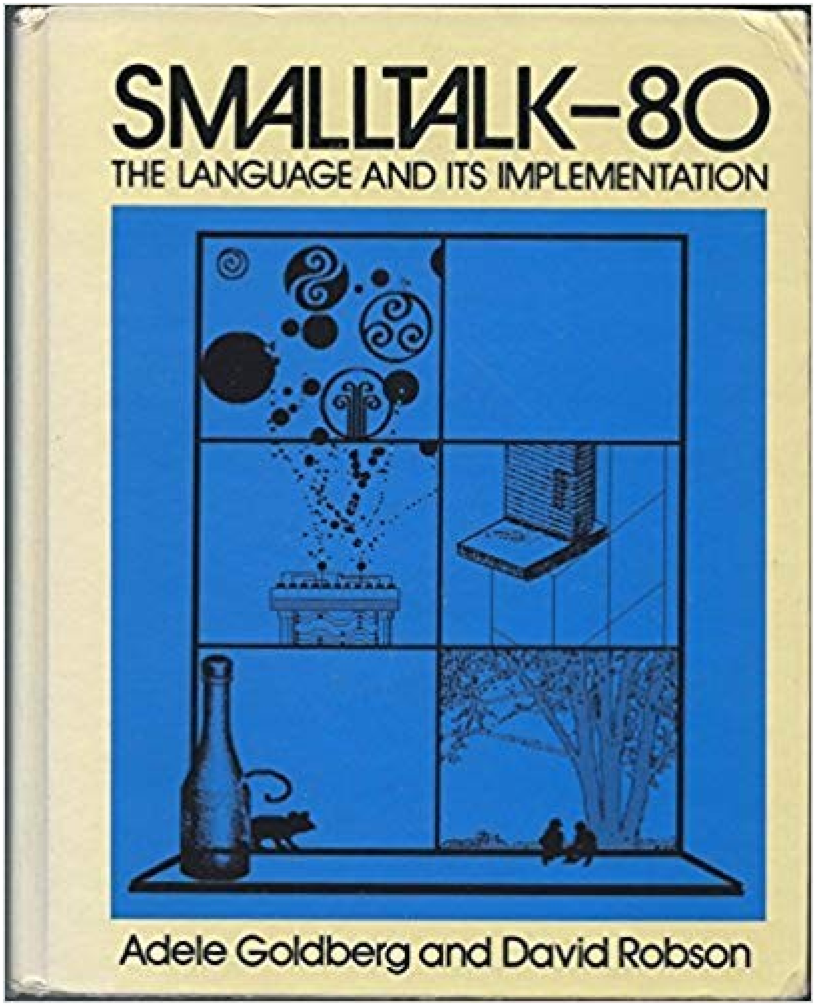
Origins: Simula

- Ole-Johan Dahl (1931–2002) and
  Kristen Nygaard (1926–2002)
  Turing Award 2001
  Dahl–Nygaard Prize for researchers since 2005
- Norwegian Computing Center
- Discrete event simulation
- Objects, classes, inheritance, subclasses, virtual procedures, coroutines…
- 1962? (1965)–1967



Origins: Smalltalk

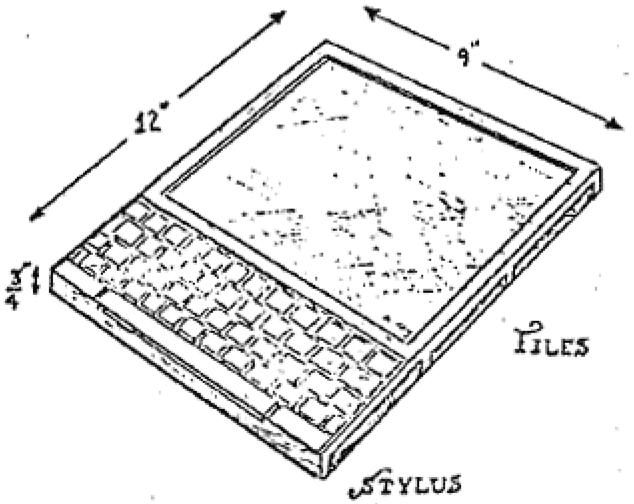
- Alan Kay (2003 Turing Award), Dan Ingalls, Adele Goldbert, and more.
- Xerox PARC
- Educational use / constructionist learning
- Message passing, actor model, metaclasses…
- 1969 (1972)–1980

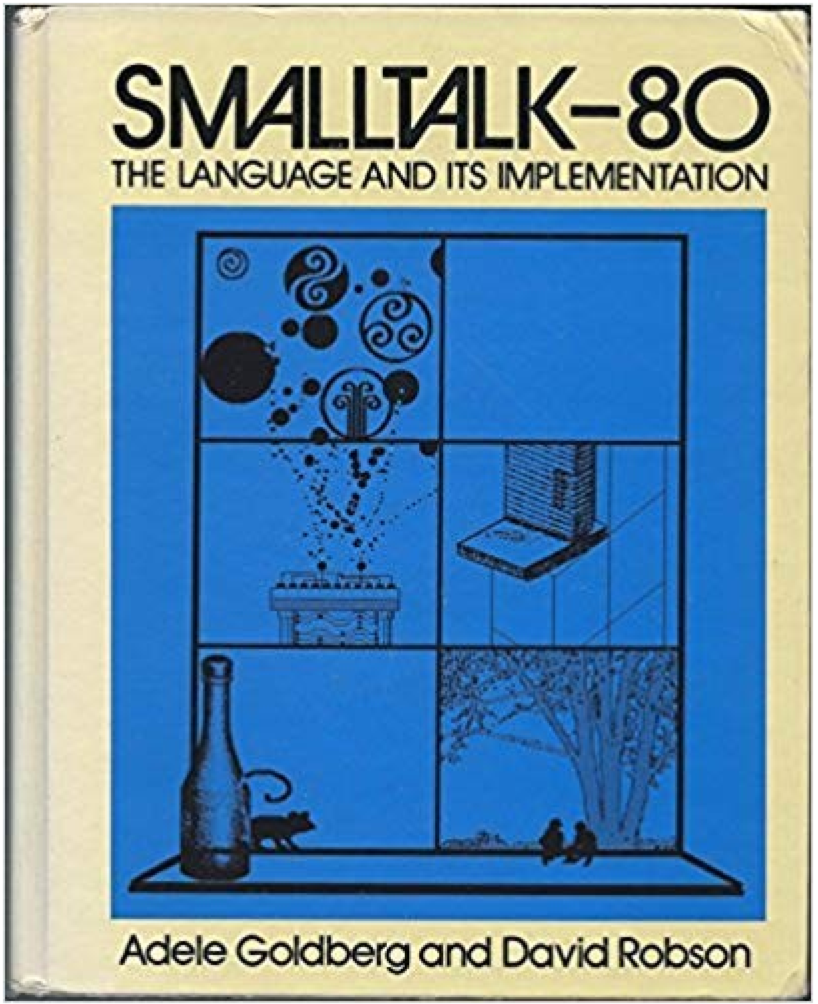


Origins: Smalltalk

- Alan Kay (2003 Turing Award), Dan Ingalls, Adele Goldbert, and more.
- Xerox PARC
- Educational use / constructionist learning
- Message passing, actor model, metaclasses…
- 1969 (1972)–1980

Software component of the Dynabook: "A Personal Computer For Children of All Ages."

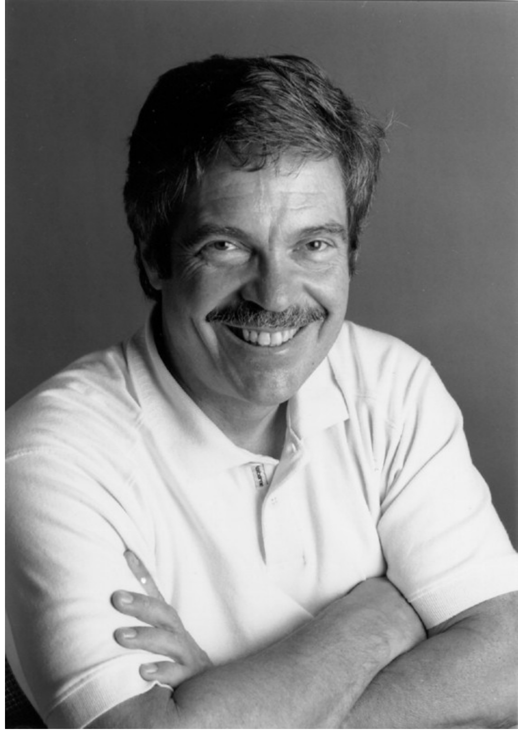




Flavors of OOP

- Object based
- Class based
- (Multimethod based)

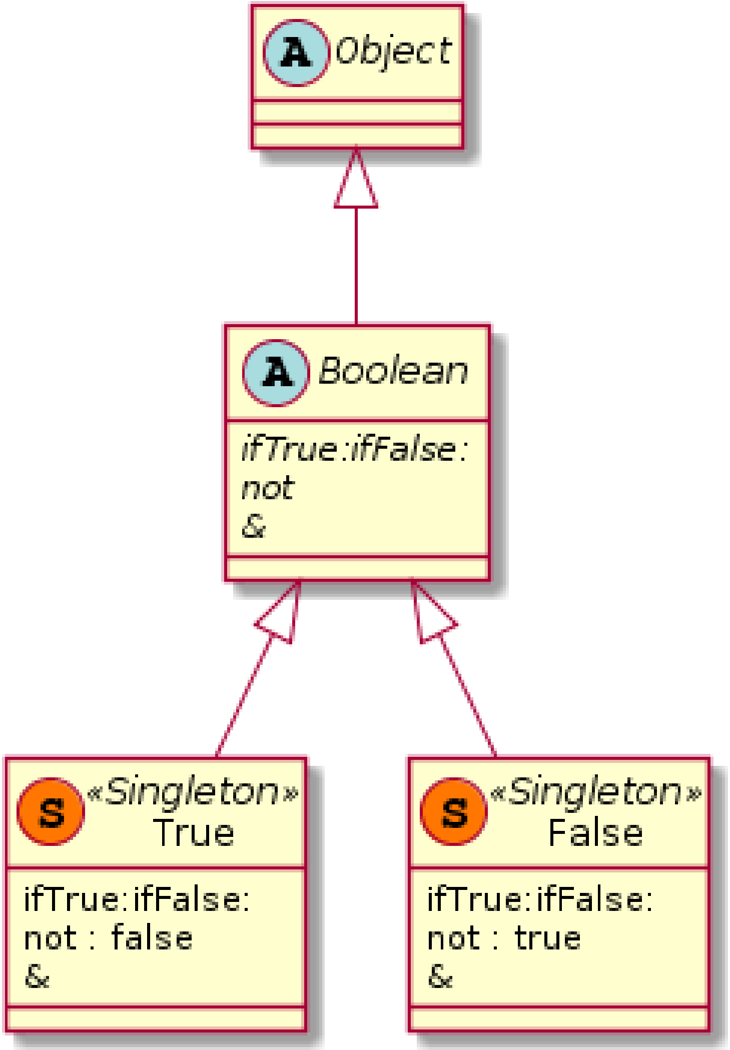
OOP according to Alan Kay



- Everything is an object
- Objects communicate by sending and receiving messages (in terms of objects)
- Objects have their own memory (in terms of objects)
- Every object is an instance of a class (which must be an object)
- The class holds the shared behavior for its instances (in the form of objects in a program list)
- To eval a program list, control is passed to the first object and the remainder is treated as its message

— Alan Kay, *The Early History of Smalltalk,* History of programming languages—II ACM (1996)

OOP according to Alan Kay

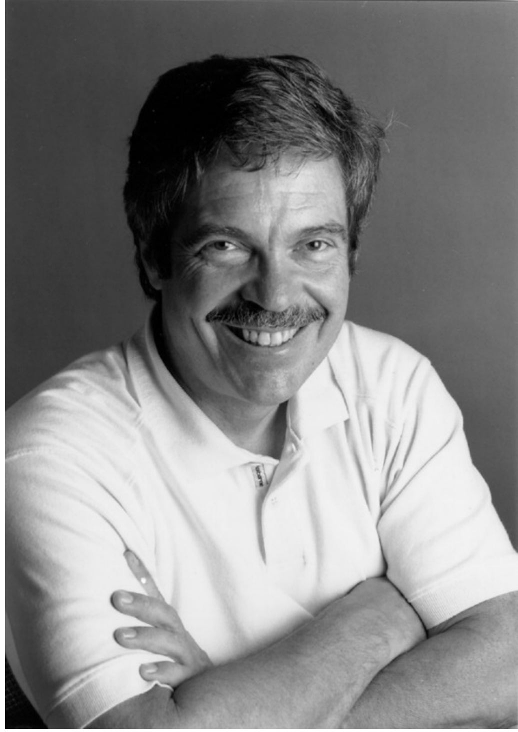



```
class True
  ifTrue: a ifFalse: b
    ^ a value

class False
  ifTrue: a ifFalse: b
    ^ b value
```

$$\text{true} \equiv \lambda a.\lambda b.a$$
$$\text{false} \equiv \lambda a.\lambda b.b$$

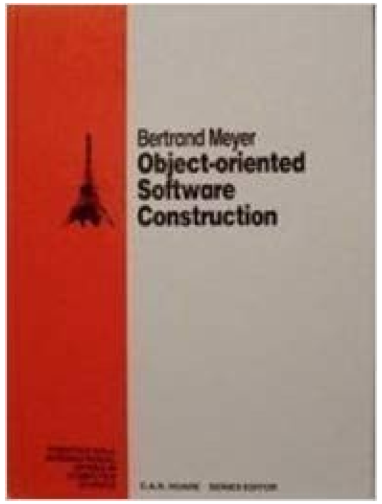
OOP according to Alan Kay



“ OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them.

— Alan Kay, *Dr. Alan Kay on the meaning of “Object-Oriented Programming”*,
http://www.purl.org/stefan_ram/pub/doc_kay_oop_en

OOP according to B. Meyer



- Level 1: *Object-based modular structure*
- Level 2: *Data abstraction*
- Level 3: *Automatic memory management*
- Level 4: *Classes*
- Level 5: *Inheritance*
- Level 6: *Polymorphism and dynamic binding*
- Level 7: *Multiple and repeated inheritance*

— Bertrand Meyer, Object-oriented software construction (1988)

On packing procedures with state: Objects and abstract data types

Object or Abstract Data Type

On Understanding Data Abstraction, Revisited

William R. Cook
University of Texas at Austin
wcook@cs.utexas.edu

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On un-

So what is the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't

“ Cook contrasted the concepts of Abstract Data Types (ADTs) and objects in Object-Oriented Languages (OOLs). He argued that although both are based on the idea of data abstraction, they are fundamentally different. In short, an ADT may be understood as a set *with* operations, whereas objects are sets *of* operations.

“ In an ADT, abstraction is achieved by hiding the type of the representation. This hiding is often modeled using an existential quantifier. In OOLs, abstraction is achieved by letting the operations themselves represent the data. The interfaces of the operations need not have any direct relation to the underlying representation at all.

- Andrew Black and Jens Palsberg, *Foundations of OOL, ACM SIGPLAN Notices* (1994)
- William R. Cook, *On understanding data abstraction, revisited* (2009)

Object or Abstract Data Type

On Understanding Data Abstraction, Revisited

William R. Cook
University of Texas at Austin
wcook@cs.utexas.edu

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called “On un-

So what is the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't

“ Cook contrasted the concepts of Abstract Data Types (ADTs) and objects in Object-Oriented Languages (OOLs). He argued that although both are based on the idea of data abstraction, they are fundamentally different. In short, an ADT may be understood as a set *with* operations, whereas objects are sets *of* operations.

“ In an ADT, abstraction is achieved by hiding the type of the representation. This hiding is often modeled using an existential quantifier. In OOLs, abstraction is achieved by letting the operations themselves represent the data. The interfaces of the operations need not have any direct relation to the underlying representation at all.

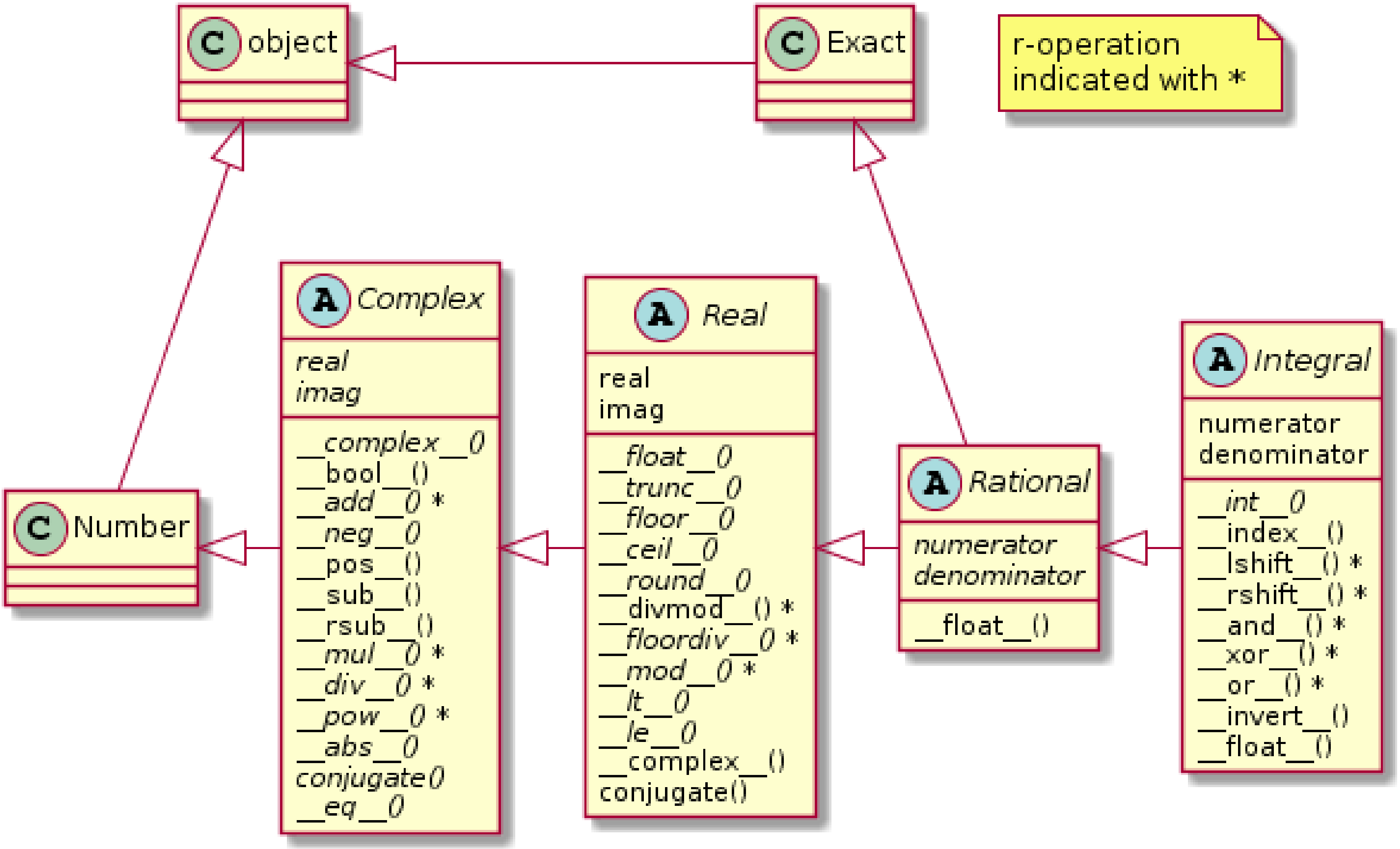
- Andrew Black and Jens Palsberg, *Foundations of OOL, ACM SIGPLAN Notices* (1994)
- William R. Cook, *On understanding data abstraction, revisited* (2009)


BRAIN
EXPLODED

Abstract data type: **int** in C

- **int** (int$N$_t) is a *set of values* in two’s complement $[-2^{N-1}..2^{N-1}-1]$
- **int** has *operators* that modify or access this representation directly.
  - arithmetic operators: +, -, *, /, %, ++, --;
  - relational operators: ==, !=, >, <, >=, <=;
  - logical operators: &&, ||, !;
  - bitwise operators: &, |, ^, ~, <<, >>;
  - assignment operators: =, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=;
  - other operators: sizeof(), & (address of), * (pointer to), ?:.

Object: **Integral** type in Python



Example: ADT vs. Object

C

```c
/* Signed addition: X = A + B */

int mbedtls_mpi_add_mpi(
  mbedtls_mpi *X,
  const mbedtls_mpi *A,
  const mbedtls_mpi *B );


/* Compute G^X mod P */

err = mbedtls_mpi_exp_mod( &ctx->GX, &ctx->G,

  &ctx->X, &ctx->P , &ctx->RP )
```

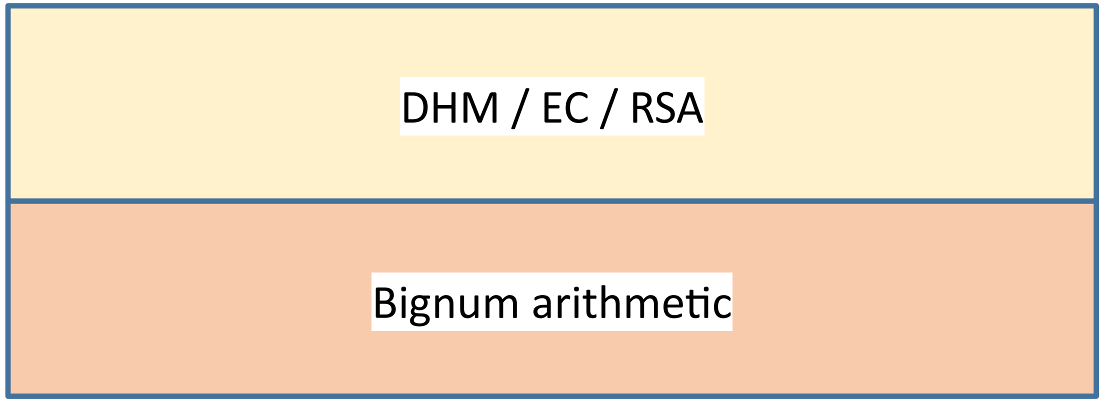
Python

```python
# Signed addition

result = a + b

# Compute G^X mod P

public_key = (generator ** private_key) % modulus
```



Example: ADT vs. Object

C

```c
/* Signed addition: X = A + B */

int mbedtls_mpi_add_mpi(
  mbedtls_mpi *X,
  const mbedtls_mpi *A,
  const mbedtls_mpi *B );


/* Compute G^X mod P */

err = mbedtls_mpi_exp_mod( &ctx->GX, &ctx->G,
  &ctx->X, &ctx->P , &ctx->RP )
```

Python

```python
# Signed addition

result = a + b

# Compute G^X mod P

public_key = (generator ** private_key) % modulus
```

Difficulty shifts from user to implementer

- Increased for implementer
- Decreased for user (when done well!)

Paradigm shift for the user

from “I know how it works” to “I know what it does”

See Also: Dijkstra, Structured Programming (1972)

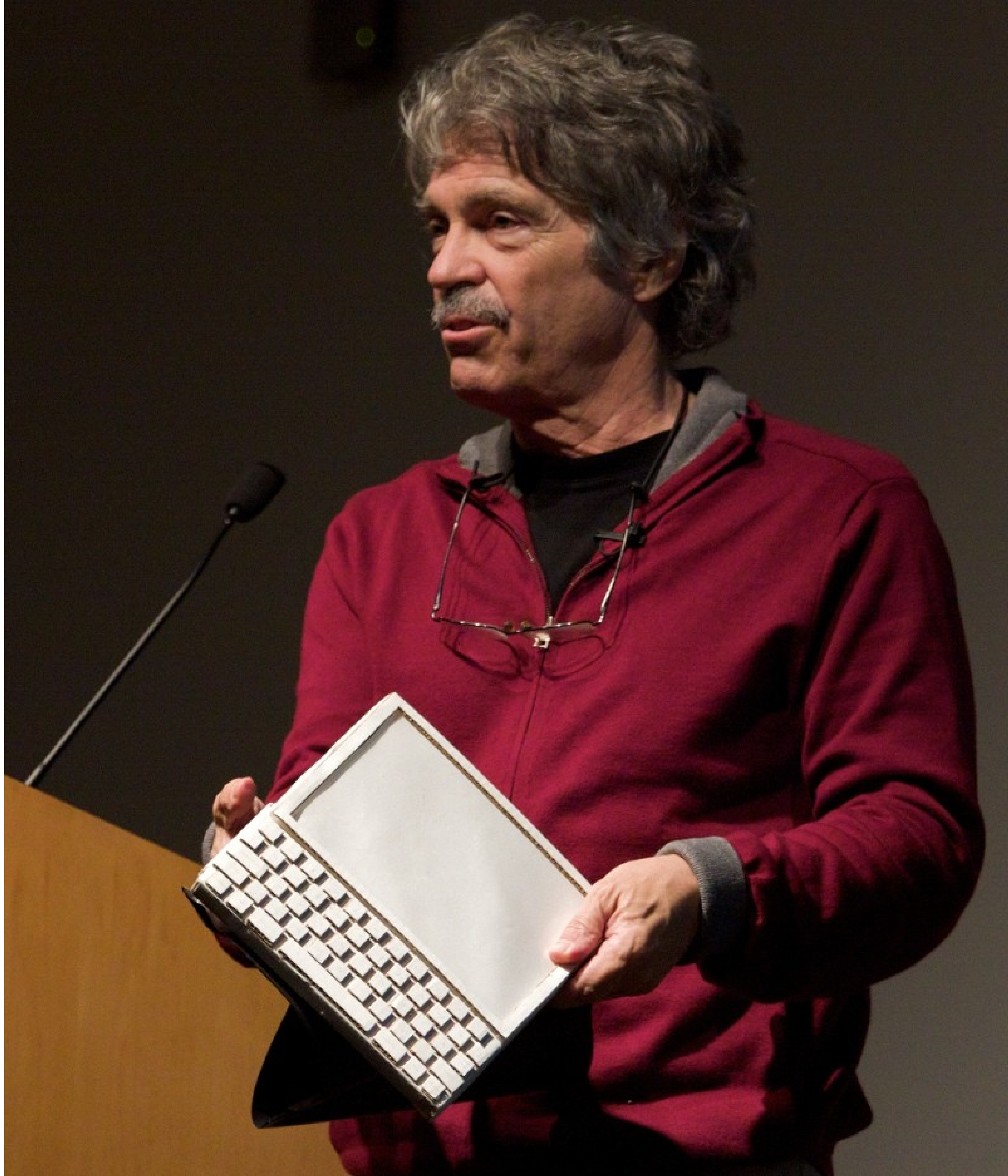
Conclusion

An object is an *interface* providing *services*.

OOP in C++

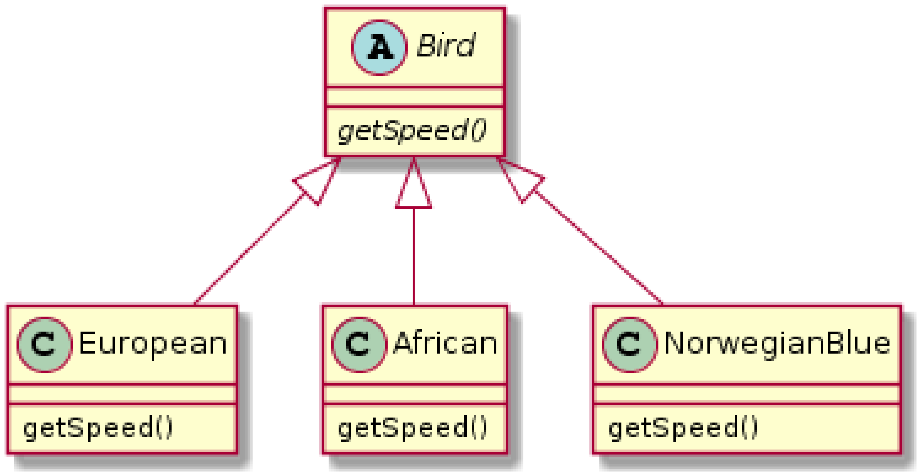
“ Actually I made up the term “object oriented”, and I can tell you I did not have C++ in mind.

— Alan Kay, *The computer revolution hasn't happened yet*, OOPSLA 1997



Replace Conditional with Polymorphism

```
double getSpeed() {
  switch (_type) {
    case EUROPEAN:
      return getBaseSpeed();
    case AFRICAN:
      return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
    case NORWEGIAN_BLUE:
      return (_isNailed) ? 0 : getBaseSpeed(_voltage);
  }
  throw new RuntimeException ("Should be unreachable");
}
```



— Martin Fowler, *Refactoring*, 1999 [link]

Replace Conditional with Polymorphism

```
double getSpeed() {
 switch (_type) {
  case EUROPEAN:
   return getBaseSpeed();
  case AFRICAN:
   return getBaseSpeed() - getLoadFactor() * _numberOfCoco
  case NORWEGIAN_BLUE:
   return (_isNailed) ? 0 : getBaseSpeed(_voltage);
 }
 throw new RuntimeException ("Should be unreachable");
}
```

— Martin Fowler, *Refactoring*, 1999 [link]

Main function

```
int main() {
  european eurobird{12.0};
  african afribird{12.0, 0.1, 5};
  norwegian_blue deadbird{220.0};
  // deadbird.nail();

  {
    auto out_iter = std::ostream_iterator<double>{std::cout, "\n"};
    std::apply(
        [&out_iter](auto const &... bird) {
          ((*out_iter++ = bird.speed()), ...);
        },
        std::make_tuple(eurobird, afribird, deadbird));
  }

  return 0;
}
```

Algebraic types: Product type

```
int main() {
  european eurobird{12.0};
  african afribird{12.0, 0.1, 5};
  norwegian_blue deadbird{220.0};
  // deadbird.nail();

  {
    auto out_iter = std::ostream_iterator<double>{std::cout, "\n"};
    std::apply(
      [&out_iter](auto const&... bird) {
        ((*out_iter++ = bird.speed()), ...);
      },
      std::make_tuple(eurobird, afribird, deadbird));
  }

  return 0;
}
```

Interface is statically checked.

Algebraic types: Sum type

```
int main() {
  european eurobird{12.0};
  african afribird{12.0, 0.1, 5};
  norwegian_blue deadbird{220.0};
  // deadbird.nail();

  using bird_t = std::variant<european, african, norwegian_blue>;
  std::vector<bird_t> birds { eurobird, afribird, deadbird };
  {
    auto out_iter = std::ostream_iterator<double>{std::cout, "\n"};
    for (auto && bird : birds)
    {
      *out_iter++ = std::visit(
        [](auto && bird) { return bird.speed(); },
        bird
      );
    }
  }
  return 0;
}
```

Interface is statically checked.

“reversed” inheritance

“Subclassing” Base class

```
class bird {
public:
  explicit bird(double speed) : speed_{speed} {}
  virtual ~bird() = default;
  virtual double speed() const { return speed_; }

private:
  double speed_;
};
```

“Subclassing” European bird

```
class bird {
public:
  explicit bird(double speed) : speed_{speed} {}
  virtual ~bird() = default;
  virtual double speed() const { return speed_; }

private:
  double speed_;
};

class european : public bird {
public:
  explicit european(double speed) : bird{speed} {}
};
```

“Subclassing” African bird

```
class bird {
public:
  explicit bird(double speed) : speed_{speed} {}
  virtual ~bird() = default;
  virtual double speed() const { return speed_; }

private:
  double speed_;
};

class african : public bird {
public:
  african(double speed, double load, int coconuts = 0)
     : bird(speed), load_{load}, coconuts_{coconuts} {}
  double speed() const final {
    return bird::speed() - load_ * coconuts_;
  }

private:
  double load_;
  int coconuts_;
};
```

“Subclassing” Norwegian blue

```
class bird {
public:
  explicit bird(double speed) : speed_{speed} {}
  virtual ~bird() = default;
  virtual double speed() const { return speed_; }

private:
  double speed_;
};

class norwegian_blue : public bird {
public:
  explicit norwegian_blue(double voltage)
    : bird{voltage} {}
  double speed() const final {
    return is_nailed_ ? 0 : 0.1 * bird::speed();
  }
  void nail() { is_nailed_ = true; }

private:
  bool is_nailed_ = false;
};
```

“Subclassing” Norwegian blue

```
class bird {
public:
  explicit bird(double speed) : speed_{speed} {}
  virtual ~bird() = default;
  virtual double speed() const { return speed_; }

private:
  double speed_;
};

class norwegian_blue : public bird {
public:
  explicit norwegian_blue(double voltage)
    : bird{voltage} {}
  double speed() const final {
    return is_nailed_ ? 0 : 0.1 * bird::speed();
  }
  void nail() { is_nailed_ = true; }

private:
  bool is_nailed_ = false;
};
```

Scott Meyers, *More Effective C++* (1996)

- Item 33 "Make non-leaf classes abstract."

Scott Meyers, *Effective C++* (2005)

- Item 34 "Differentiate between inheritance of interface and inheritance of implementation."

“Subtyping” Base class

```
struct bird {
  virtual ~bird() = default;
  virtual double speed() const = 0;
};
```

“Subtyping” European bird

```
struct bird {
  virtual ~bird() = default;
  virtual double speed() const = 0;
};

class european : public bird {
public:
  explicit european(double speed) : speed_{speed} {}
  double speed() const final { return speed_; }

private:
  double speed_;
};
```

“Subtyping” African bird

```
struct bird {
  virtual ~bird() = default;
  virtual double speed() const = 0;
};

class african : public bird {
public:
  african(double speed, double load, int coconuts = 0)
      : speed_(speed), load_{load}, coconuts_{coconuts} {}
  double speed() const final { return speed_ - load_ * coconuts_; }

private:
  double speed_;
  double load_;
  int coconuts_;
};
```

“Subtyping” Norwegian blue

```
struct bird {
  virtual ~bird() = default;
  virtual double speed() const = 0;
};

class norwegian_blue : public bird {
public:
  explicit norwegian_blue(double voltage) : voltage_{voltage} {}
  double speed() const final {
    return is_nailed_ ? 0 : 0.1 * voltage_;
  }
  void nail() { is_nailed_ = true; }

private:
  double voltage_;
  bool is_nailed_ = false;
};
```

“ Within C++, there is a much smaller and cleaner language struggling to get out.

— Bjarne Stroustrup, *The Design and Evolution of C++* (1994)

“ Within C++, there is a much smaller and cleaner language struggling to get out.

— Bjarne Stroustrup, *The Design and Evolution of C++* (1994)

- James O. Coplien, *Curiously Recurring Template Patterns* (1995)
- P. Canning, W. Cook, W. Hill, W. Olthoff, J.C. Mitchell, *F-Bounded Polymorphism for Object-Oriented Programming* (1989)

CRTP
Base class

```
template <class T> struct bird {
  double speed() const {
    return static_cast<T const *const>(this)->_speed();
  }
};
```

CRTP
European bird

```
template <class T> struct bird {
  double speed() const {
    return static_cast<T const *const>(this)->_speed();
  }
};

class european : public bird<european> {
  friend struct bird;

public:
  explicit european(double speed) : speed_{speed} {}

private:
  double _speed() const { return speed_; }
  double speed_;
};
```

CRTP
African bird

```
template <class T> struct bird {
  double speed() const {
    return static_cast<T const *const>(this)->_speed();
  }
};

class african : public bird<african> {
  friend struct bird;

public:
  african(double speed, double load, int coconuts = 0)
      : speed_(speed), load_{load}, coconuts_{coconuts} {}

private:
  double _speed() const { return speed_ - load_ * coconuts_; }
  double speed_;
  double load_;
  int coconuts_;
};
```

CRTP
Norwegian blue

```
template <class T> struct bird {
  double speed() const {
    return static_cast<T const *const>(this)->_speed();
  }
};

class norwegian_blue : public bird<norwegian_blue> {
  friend struct bird;

public:
  explicit norwegian_blue(double voltage) : voltage_{voltage} {}
  void nail() { is_nailed_ = true; }

private:
  double _speed() const { return is_nailed_ ? 0 : 0.1 * voltage_; }
  double voltage_;
  bool is_nailed_ = false;
};
```

CRTP

```
template <class T> struct bird {
  double speed() const {
    return static_cast<T const *const>(this)->_speed();
  }
};
```

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder.

— GoF, *Design Patterns* (1994)

Reusable abstraction

```
template <class T> class movable_t {
public:
  auto speed() const { return derived()._speed(); }
  auto position() const { return derived()._position(); }
private:
  T const& derived() const {
    return *static_cast<T const * const>(this);
  }
  T& derived() {
    return *static_cast<T * const>(this);
  }
};
```