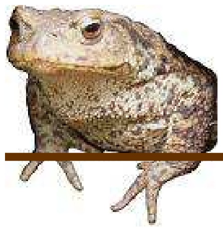
Course Wrap-up: the Past and Future of Objects

Principles of Software System Construction

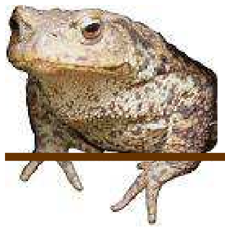
Jonathan Aldrich and Charlie Garrod

Fall 2013



Outline

- The Beginnings of Objects
  - Simulation in Simula
- Pure OO in Smalltalk
  - Historical context, demo
- OO Design Approaches
  - Refactoring and related design principles
- Self: Objects Without Classes
- Actors: Concurrent Objects
- Emerald: Distributed Objects
- The Engineering Significance of Objects
- Current OO Research at CMU



Simulation at the NCC in 1961

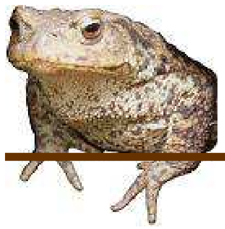
- Context: Operations research
  - Goal: to improve decision-making by **simulating complex systems**
    - Discrete-event simulations like Rabbit world, but in domains like traffic analysis
  - Kristin Nygaard and Ole-Johan Dahl at the Norwegian Computing Center

Dahl and Nygaard at the time of Simula's development

- Development of SIMULA I
  - Goal: SIMULA "should be **problem-oriented** and not computer-oriented, even if this implies an appreciable increase in the amount of work which has to be done by the computer.“
  - Modeled simulations “as a variable **collection of interacting processes**"
  - Design approach: "Instead of deriving language constructs from discussions of the described systems combined with implementation considerations, **we developed model system properties** suitable for portraying discrete event systems, considered the implementation possibilities, **and then settled the language constructs**."

SIMULA: a Motivating Problem

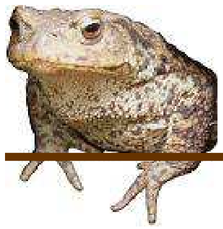
- Need to store vehicles in a toll booth queue.
- Want to store vehicles in a linked list to represent the queue
- Each vehicle is either a car, a truck, or a bus.
- Different kinds of vehicles interact with the toll booth in different ways



Needs Motivating OOP

- Issues with SIMULA I
  - Since each object in a simulation was a process, it was awkward to get **attributes** of other objects
  - "We had seen many useful applications of the process concept to represent **collections of variables and procedures**, which functioned as natural units of programming" motivating more direct support for this
  - "When writing simulation programs we had observed that processes often **shared a number of common properties**, both in data attributes and actions, but were structurally different in other respects so that they had to be described by separate declarations."
  - "**memory space** [was] our most serious bottleneck for large scale simulation."

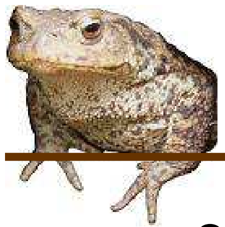
[source: Kristen Nygaard and Ole-Johan Dahl, The Development of the SIMULA Languages, History of Programming Languages Conference, 1978]



Needs Motivating OOP

- Issues with SIMULA I
  - Since each object in a simulation was a process, it was awkward to get **attributes** of other objects
  - "We had seen many useful applications of the process concept to represent **collections of variables and procedures**, which functioned as natural units of programming" motivating more direct support for this
  - "When writing simulation programs we had observed that processes often **shared a number of common properties**, both in data attributes and actions, but were structurally different in other respects so that they had to be described by separate declarations."
  - "**memory space** [was] our most serious bottleneck for large scale simulation."

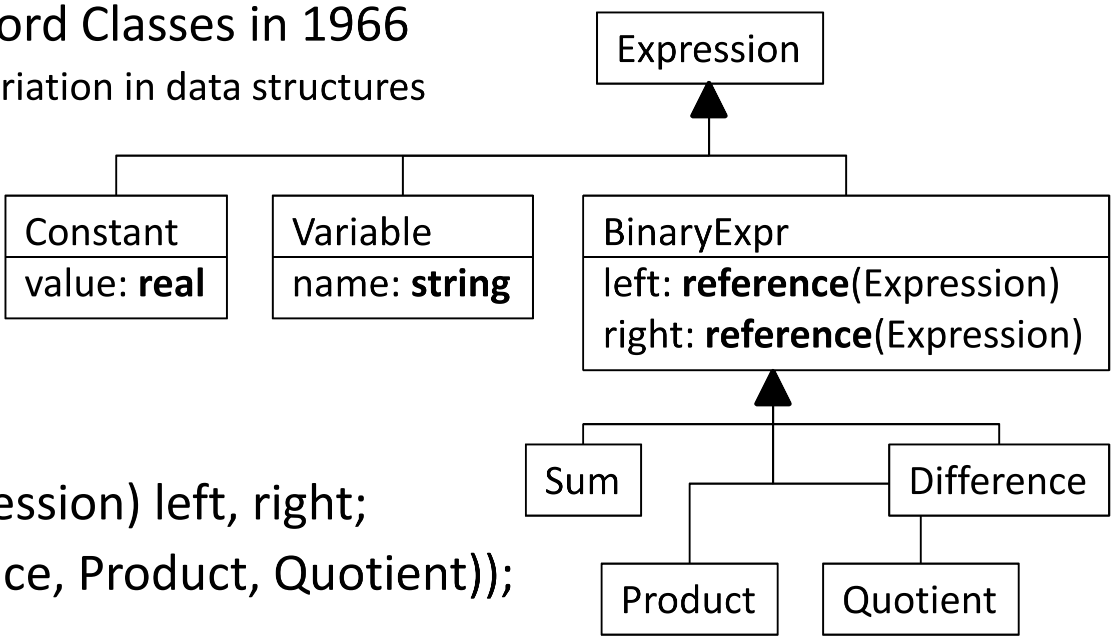
Garbage collection was a good technology for the memory problem. The others required new ideas.



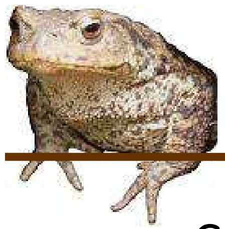
Hoare's Record Classes

- C. A. R. Hoare proposed Record Classes in 1966
  - Goal: capture similarity and variation in data structures

```
record class Expression (
    subclasses
        Constant(real value),
        Variable(string name),
        BinaryExpr(reference(Expression) left, right;
            subclasses Sum, Difference, Product, Quotient));
```



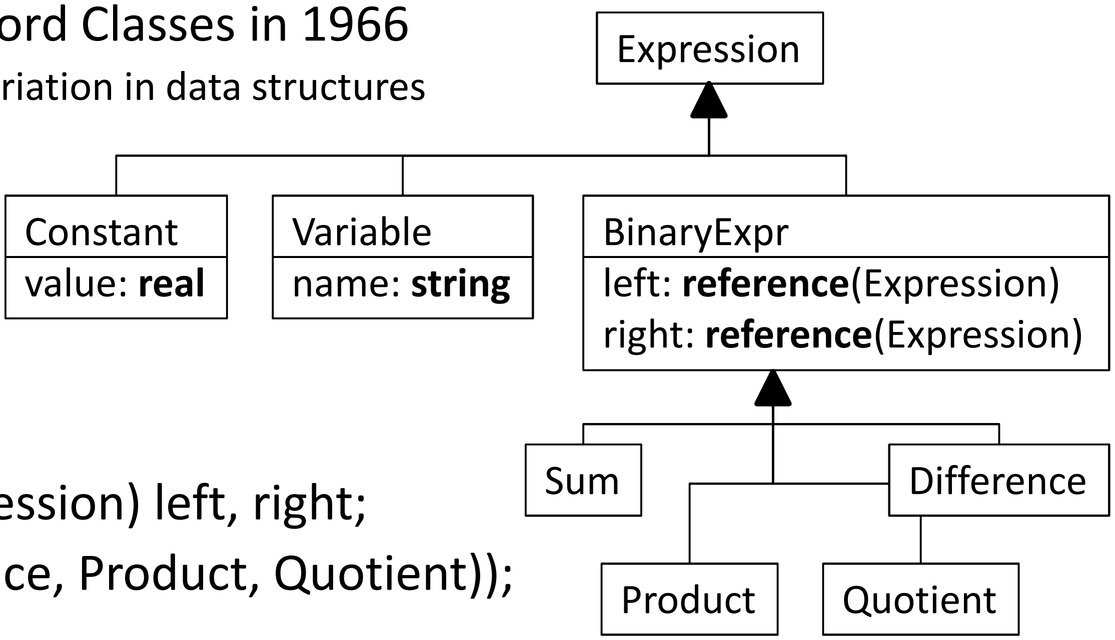
- Each **class** described a particular record structure
- A **subclass** shared fields from its parent
- Variables could take any type in the subclass hierarchy
- A **record class discriminator** provided case analysis on the record type



Hoare's Record Classes

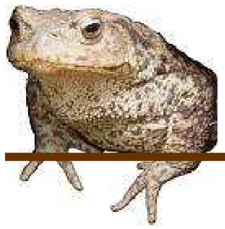
- C. A. R. Hoare proposed Record Classes in 1966
  - Goal: capture similarity and variation in data structures

record class Expression (
    subclasses
        Constant(**real** value),
        Variable(**string** name),
        BinaryExpr(**reference**(Expression) left, right;
            subclasses Sum, Difference, Product, Quotient));

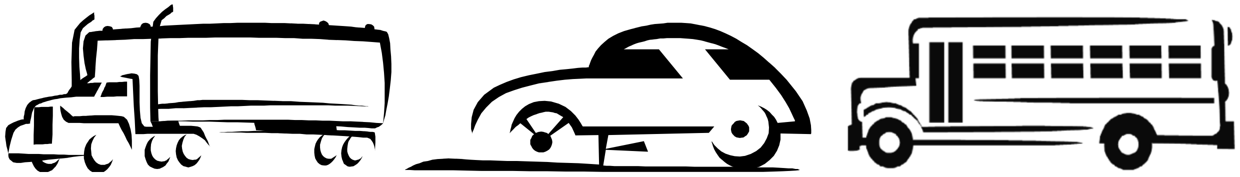


Dahl and Nygaard's observations on record classes:

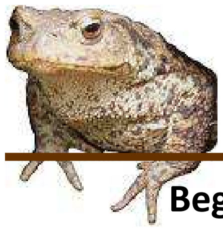
- "We needed subclasses of processes with...actions...not only of pure data records"
- "We also needed to group together common process properties in such a way that they could be applied later, in a variety of different situations not necessarily known in advance"



SIMULA 67’s Class Prefix Idea



- Create a Link class to represent the linked list
- Add the Link class as a **prefix** to vehicles, which are subclasses
  - Today we would say this is not a good design—but it nevertheless was enough to motivate a good idea
- As in Hoare’s design, subclassing is hierarchical
  - Car, Truck, etc. are subclasses of Vehicle
- Unlike Hoare’s classes, Simula classes can have **virtual procedures**
  - Allows subclasses to override behavior for the toll booth
- Unlike in Hoare’s design, each class was **declared separately**
  - Link could be reused for other linked lists, not just lists of vehicles
  - Supports extensibility: can add RVs later as a subclass of Vehicle

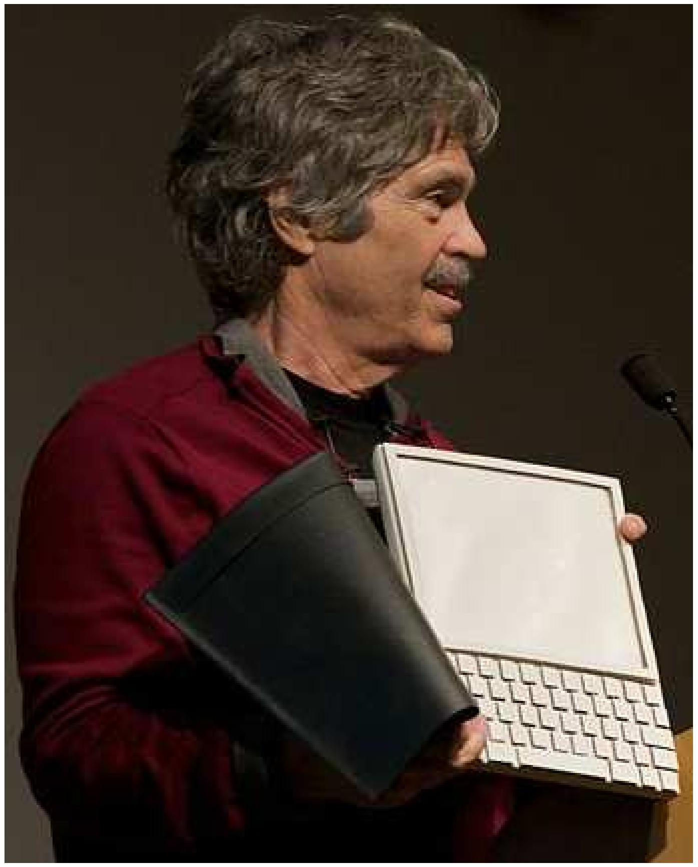


Sample Simula 67 Code

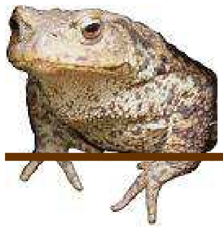
```
Begin
    Class Glyph;
        Virtual: Procedure print Is Procedure print;;
    Begin End;

    Glyph Class Char (c);
        Character c;
    Begin
        Procedure print;
            OutChar(c);
    End;

    Glyph Class Line (elements);
        Ref (Glyph) Array elements;
    Begin
        Procedure print;
        Begin
            Integer i;
            For i:= 1 Step 1 Until UpperBound (elements, 1) Do
                elements (i).print;
            OutImage;
        End;
    End;

    Ref (Glyph) rg;
    Ref (Glyph) Array rgs (1 : 4);

    ! Main program;
    rgs (1):- New Char ('A');
    rgs (2):- New Char ('b');
    rgs (3):- New Char ('b');
    rgs (4):- New Char ('a');
    rg:- New Line (rgs);
    rg.print;
End;
```

Smalltalk Context: Personal Computing

- The Dynabook at Xerox PARC: "A Personal Computer for Children of All Ages"
- Funded by US Govt (ARPA, the folks who brought you the internet) to facilitate portable maintenance documentation
- Alan Kay's goal
  - Amplify human reach
  - Bring new ways of thinking to civilization (*CMU still pursuing this goal with computational thinking*)

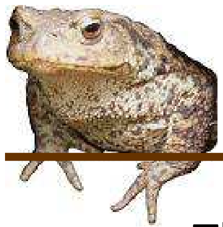

Alan Kay with a Dynabook prototype

Smalltalk and Simula



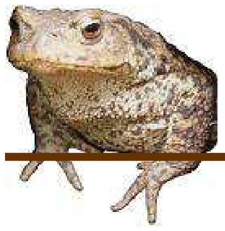
"What I got from Simula was that you could now replace bindings and assignment with *goals*. The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as sites of higher level behaviors more appropriate for use as dynamic components."

- Alan Kay, The early history of Smalltalk. In History of programming languages—II, 1993.

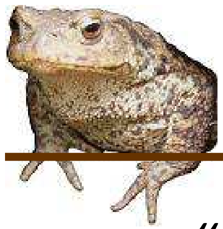


Smalltalk

- The name
  - "Children should program in…"
  - "Programming should be a matter of…"
- Pure OO language
  - Everything is an object (including true, "hello", and 17)
  - All computation is done via sending messages
    - 3 + 4 sends the "+" message to 3, with 4 as an argument
    - To create a Point, send the "new" message to the Point class
      - Naturally, classes are objects too!
- Garbage collected
  - Following Lisp and Simula 67
- Reflective
  - Smalltalk is implemented (mostly) in Smalltalk
    - A few primitives in C or assembler
  - Classes, methods, objects, stack frames, etc. are all objects
    - You can look at them in the debugger, which (naturally) is itself implemented in Smalltalk

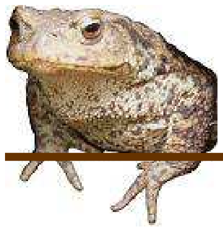


Smalltalk Demo



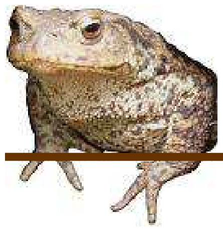
Smalltalk, according to Alan Kay

- "In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole—like data structures, procedures, and functions which are the usual paraphernalia of programming languages—**each Smalltalk object is a recursion of the entire possibilities of the computer**.
- "…everything we describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.
- "Thus [Smalltalk's] semantics are a bit like having thousands and thousands of computers all hooked together in a very fast network."



Dan Ingalls' perspective

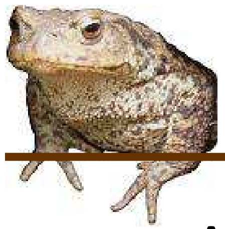
- Computing should be viewed as an intrinsic capability of objects that can be uniformly invoked by sending messages... Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires.
  - Daniel Ingalls, Design Principles Behind Smalltalk (1981)



Impact of Smalltalk and Simula

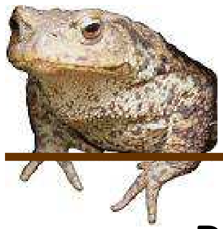
- Mac (and later Windows): inspired by Smalltalk GUI
- GUI frameworks
  - Smalltalk MVC → MacApp → Cocoa, MFC, AWT/Swing, …
- C++: inspired by Simula 67 concepts
- Objective C: borrows Smalltalk concepts, syntax
- Java: garbage collection, bytecode from Smalltalk
- Ruby: pure OO model almost identical to Smalltalk
  - All dynamic OO languages draw from Smalltalk to some extent

- Design and process ideas impacted by Smalltalk
  - Patterns, **Refactoring**, Extreme programming/Agile movement



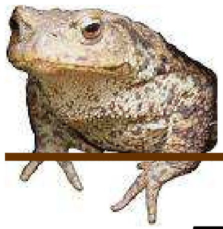
Refactoring

- A "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior" – Martin Fowler, *Refactoring: Improving the Design of Existing Code.*
- Goal: improve maintainability
  - Increase readability, reduce complexity, support extensibility
- Typically done in an incremental way
  - Many basic "micro-refactorings" applied in sequence
- May be triggered by ***code smells***
  - Code duplication, large classes or methods, high coupling, etc.
  - A good proxy: *what would your TA think?*



Sample Refactorings

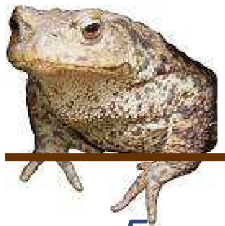
- Rename method
  - Tricky/tedious to do manually: all references must be updated
- Encapsulate field
  - Replace with getter/setter, allowing representation change and interposed behavior
- Replace conditional with polymorphism
  - Makes code more extensibe
- Extract class or method
  - Move reusable code into a construct that supports reuse
- Move method
  - When the method is in the wrong place
- Pull up/push down methods or fields
  - To reuse a method more widely, or to remove methods where they are not used



Refactoring in Practice

- Tool support helps a lot
  - Although only simple refactorings are typically automated
  - Eclipse demo
- Supported by good unit tests, version control
  - Easy to have confidence in changing code if you have good tests, and can easily revert to a known state
- Dovetails with agile methodology
  - Value continuous code improvement over big design up front
  - Often, you don't know enough about the problem to do the big design anyway—so fixing what you get wrong is essential
  - *More in 15-313!*

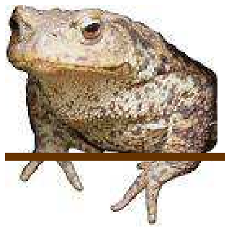
More Design Principles



Don’t Repeat Yourself

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

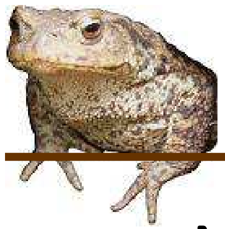
- Duplicate code: the most obvious symptom
  - Refactor to merged close duplicates into methods
  - Create abstract classes
  - Apply design patterns
  - Use code generation when PL abstractions are not good enough
    - But never edit the generated code! [no longer a single representation]
- Watch out for other sources of duplication
  - Text files, etc.
- Also known as “Once and Only Once”
  - Some argue for a “Rule of Three” – refactor when a piece of code appears three times. In practice it depends on how much code is duplicated.
  - Quality Rationale [Yaron Minksy, Jane Street]: *you can’t pay people enough to read boring/repetitive code carefully*
  - Evolution Rationale: difficult to evolve repeated code, especially maintaining consistency



Worse is Better [Gabriel]

Contrasting design styles:

- The “Right Thing”
  - Correctness is an absolute requirement
  - Consistency is also required, even at the cost of simplicity/completeness
  - Completeness: cover as many important situations as is practical
  - Simplicity is desirable, especially in the interface – but OK to sacrifice
- “Worse is Better”
  - Simplicity, especially in the implementation, is most important
  - Correctness is essential, but it is better to be simple than correct
  - Consistency is a goal; OK to sacrifice for simplicity, but better to leave out functionality
  - Completeness is desirable, but can be sacrificed for any other quality

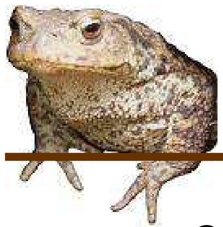


Worse is Better: ***Simplicity***

- Memorable, but oxymoronic slogan
  - Meaning is that less functionality ("worse") is often preferable ("better")

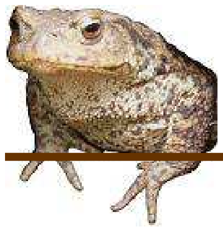
- Rationale
  - If the initial program is good, it will be easier to implement and adapt
  - Use will spread rapidly, but users are conditioned to expect imperfection
  - Pressure, and capacity, to improve due to widespread use

"Therefore, the worse-is-better software first will gain acceptance, second will condition its users to expect less, and third will be improved to a point that is almost the right thing. In concrete terms, even though Lisp compilers in 1987 were about as good as C compilers, there are many more compiler experts who want to make C compilers better than want to make Lisp compilers better."



Simplicity or Planning for Change?

- 214 emphasizes *planning for change* in design
  - But adding flexibility often has a cost: e.g. interface indirection
  - Almost every design pattern has a complexity cost
  - OO keeps costs lower than other paradigms, but the cost is still there
- Simplicity itself facilitates change
  - It is easier to change a system if it is small and well-understood
  - KISS: Keep it simple stupid
    - Coined by Kelly Johnson at Lockheed: challenge to fix a jet plane with a handful of tools
- Are you *sure* you are going to need it?
  - Often developers, and even users, are certain they need functionality that doesn't turn out to be important
  - So common there's an acronym: YAGNI (you ain't gonna need it)



Next Time

- Self: Objects Without Classes
- Actors: Concurrent Objects
- Emerald: Distributed Objects
- The Engineering Significance of Objects
- Current OO Research at CMU