The Design in Question

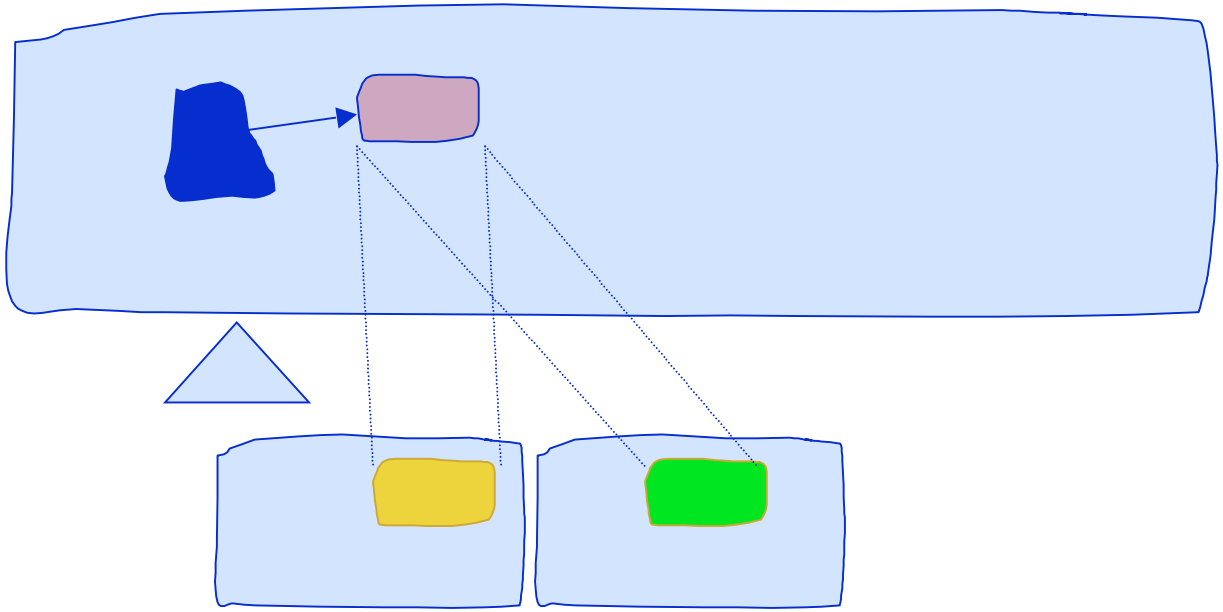
- The Basic Idea behind Frameworks
- Subclassing vs SubTyping
- Design Heuristics
- Design Symptoms

Inheritance as Parameterization

- Subclass customizes template method by implementing (abstract) operations
- Any method acts as a parameter
- Methods are unit of reuse
- Abstract class -- one that must be customized before it can be used

Methods are Unit of Reuse

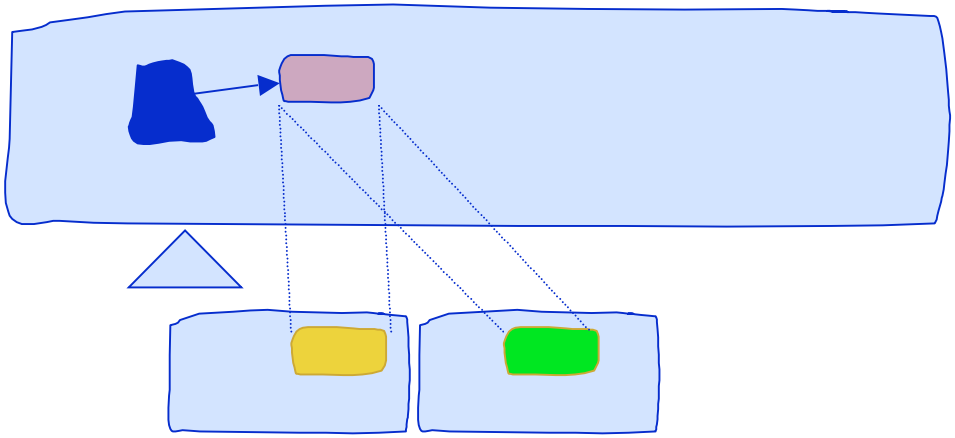
- self send are plans for reuse



-  can be abstract or not

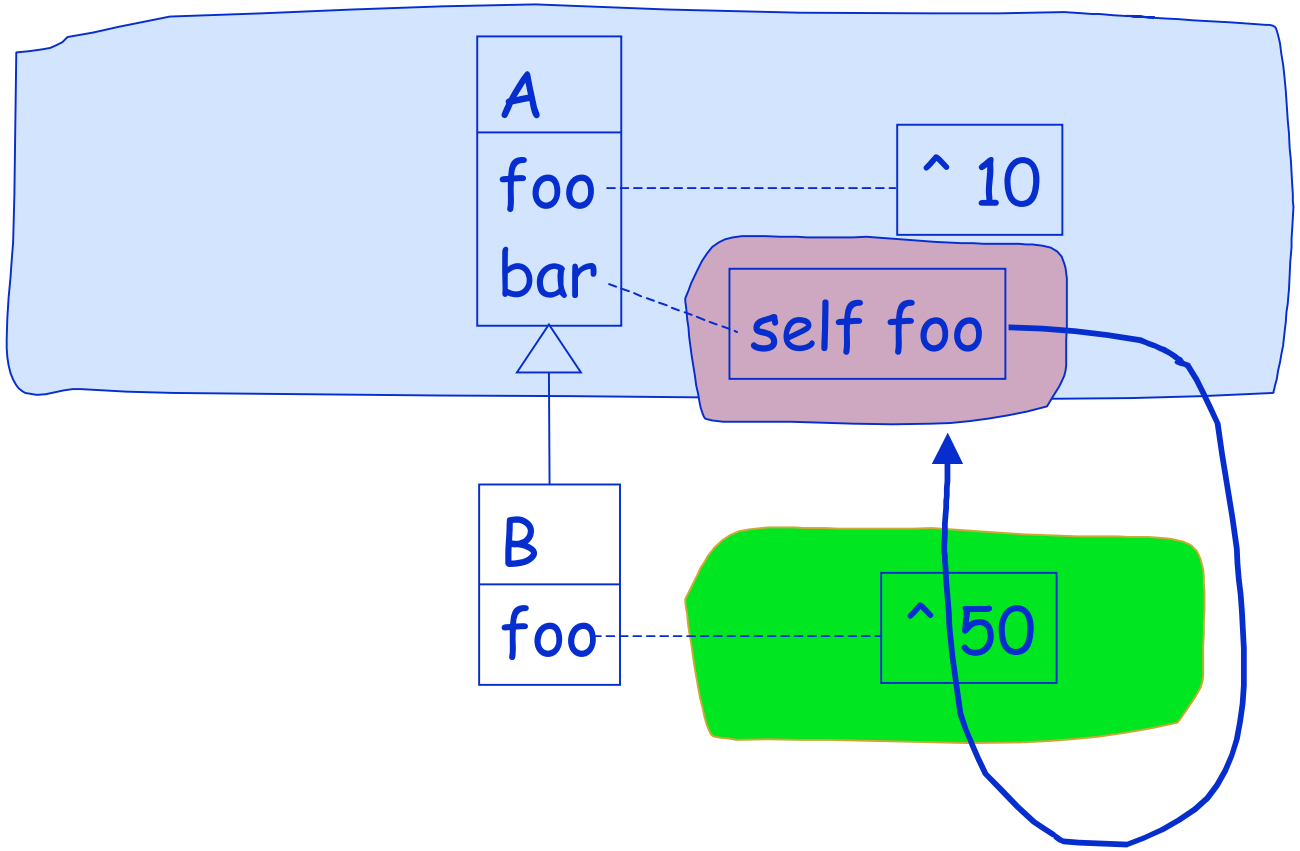
Frameworks vs. Libraries

- Libraries
  - You call them
  - Callback to extend them
- Framework
  - Hollywood principle: Don't call me I will call you
  - Three applications before a framework!



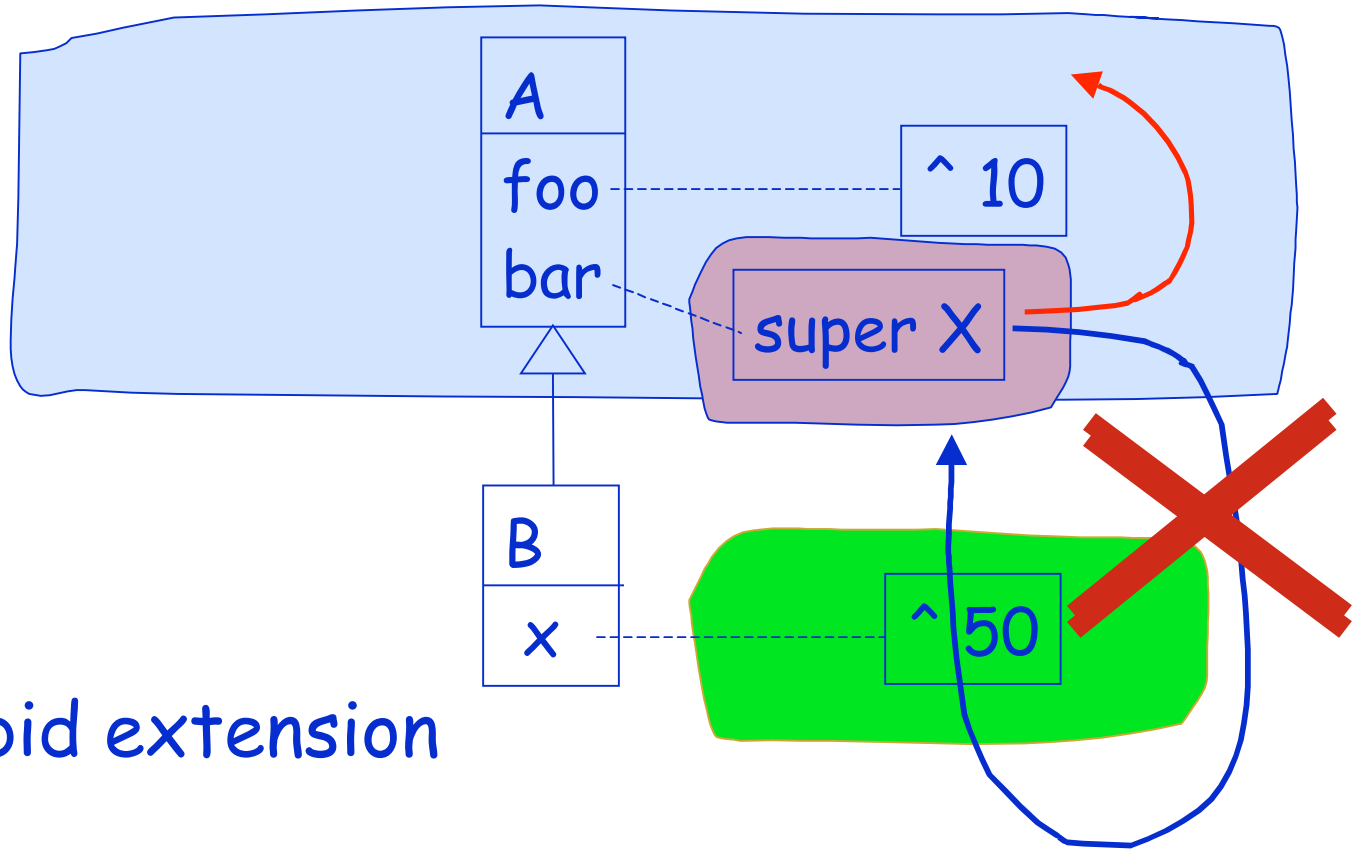
You remember self…

- self is dynamic
- self acts as a hook



You remember super...

- super is static



- super forbid extension

By the way…

- Frameworks design
  - Need at least 3 applications to support the generalization
- Smile if somebody tell that they start implementing a framework

RoadMap

- The Basic Idea behind Frameworks
- Subclassing vs SubTyping
- Design Heuristics
- Design Symptoms

Inheritance and Polymorphism

- Polymorphism works best with standard interfaces
- Inheritance creates families of classes with similar interfaces
- Abstract class describes standard interfaces
- Inheritance helps software reuse by making polymorphism easier

Specification Inheritance (Subtyping)

- Reuse of specification
  - A program that works with Numbers will work with Fractions.
  - A program that works with Collections will work with Arrays.
- A class is an abstract data type (Data + operations to manipulate it)

Inheritance for code reuse (Subclassing)

- Dictionary is a subclass of Set
- Semaphore is a subclass of LinkedList

- Subclass reuses code from superclass, but has a different specification. It cannot be used everywhere its superclass is used. Usually overrides a lot of code.
- ShouldNotImplement use is a bad smell...

Inheritance for code reuse (Subclassing)

Inheritance for code reuse is good for

- rapid prototyping
- getting application done quickly.

Bad for:

- easy to understand systems
- reusable software
- application with long life-time.

Quizz

- How to implement a Stack and a Queue?
  - Subclass of OrderedCollection?
  - Using an orderedCollection?
- Circle subclass of Point?
- Implement Stack and Queue

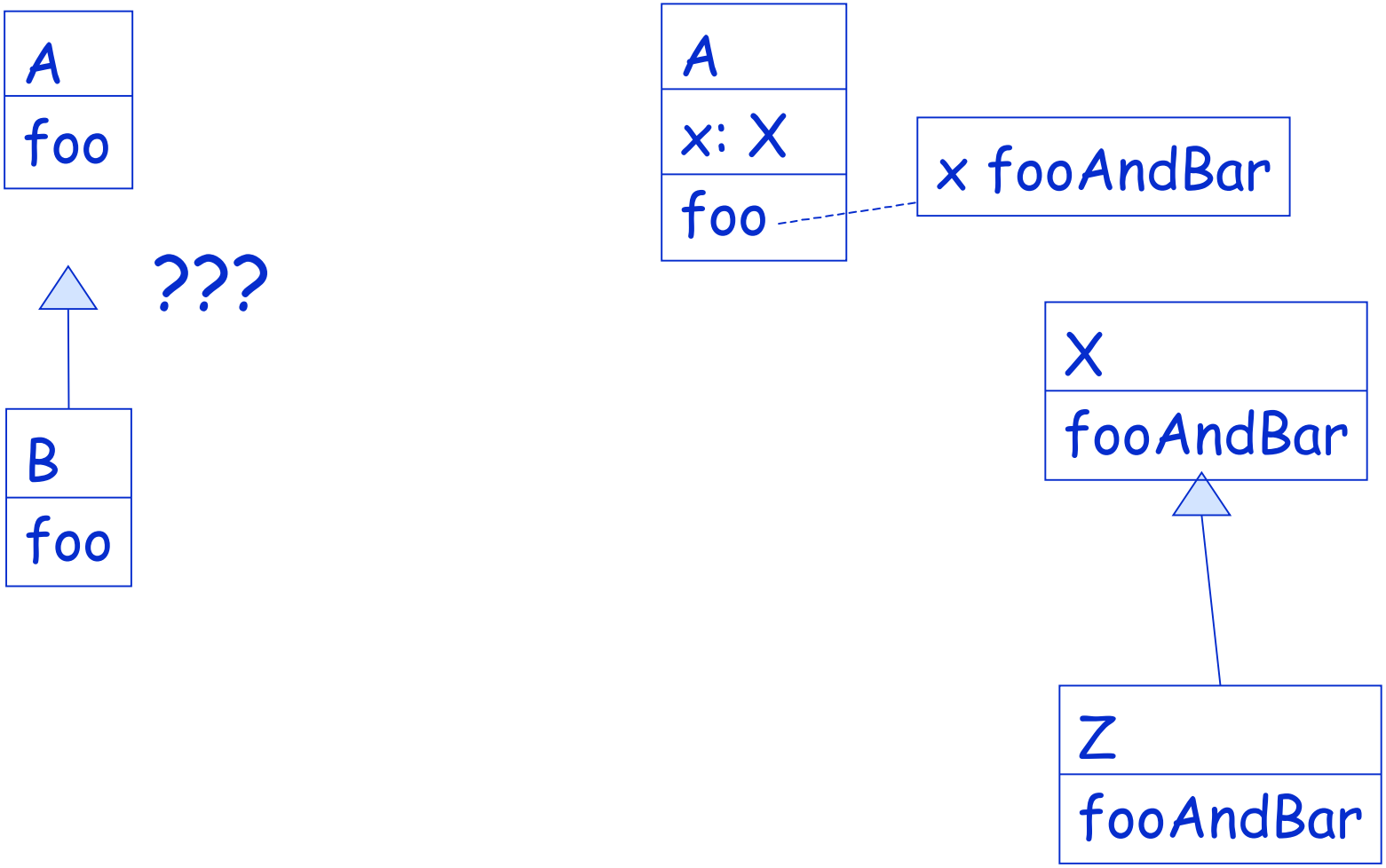
How to Choose?

- Favor subtyping
- When you are in a hurry, do what seems easiest.
- Clean up later, make sure classes use "is-a" relationship, not just "is-implemented-like".
- Is-a is a design decision, the compiler only enforces is-implemented-like!!!

Subclassing vs. Composition

- Subclassing is not a panacea
- Require class definition
- Require method definition
- Extension should be prepared in advance
- No run-time changes
- Composition: basis of Design patterns
  - Strategy,

The idea: Delegating to other objects



Composition Advantages

- Pros
  - Run-time change
  - Clear responsibility
  - No blob
  - Clear interaction protocol
- Cons
  - New class
  - Delegation
  - New classes

Example of Parametrization Advantages

```
DialectStream>>initializeST80ColorTable
  "Initialize the colors that characterize the ST80
  dialect"

  ST80ColorTable _ IdentityDictionary new.
  #((temporaryVariable blue italic)
     (methodArgument blue normal)
     ...
     (setOrReturn black bold)) do:
        [:aTriplet |
        ST80ColorTable at: aTriplet first put: aTriplet
  allButFirst]
```

- Color tables hardcoded in method

Still Require Subclassing & Compilation

```
DialectStream>>
  ST80ColorTable _ IdentityDictionary new.
  self description do:
        [:aTriplet |
        ST80ColorTable at: aTriplet first put: aTriplet
  allButFirst]

DialectStream>>description
   ^ #((temporaryVariable blue italic)
       (methodArgument blue normal)
        ...
       (setOrReturn black bold))
```

Composition-based Solution

```
DialectStream>>initializeST80ColorTableWith: anArray

  ST80ColorTable := IdentityDictionary new.
  anArray
        do: [:aTriplet | ST80ColorTable at: aTriplet first
                                    put: aTriplet allButFirst]
```

In a Client

```
DialectStream initializeST80ColorTableWith:
   #(#(#temporaryVariable #blue #normal) …
   #(#prefixKeyword #veryDarkGray #bold)
   #(#setOrReturn #red #bold) ).
  DialectStream initialize
```

Type Checking Verses Runtime Checking

A number of people believe that large programs can not be written in languages without typing, preferable strong type checking. They believe that without the compiler checking type usage programmers will make too many type usage errors. This will slow the development of programs and result in too many errors. Programmers using Smalltalk, Lisp, Perl, APL, Python or Ruby (to name a few) tend to believe that type usage slows program development. Mixing these two groups of people in newsgroups results in many flame wars. These flame wars are a waste of emotional energy. Try Smalltalk and see for yourself. You might find that for you type checking is very important. If so then you know it by experience rather than repeating what you were told in a course. You might find that you do just fine without type checking.

Behavior Up and State Down

- Define classes by behavior, not state
- Implement behavior with abstract state: if you need state do it indirectly via messages. Do not reference the state variables directly
- Identify message layers: implement class's behavior through a small set of kernel method

Example

```
Collection>>removeAll: aCollection
  aCollection do: [:each | self remove: each]
  ^ aCollection

Collection>>remove: oldObject
  self remove: oldObject ifAbsent: [self notFoundError]

Collection>>remove: anObject ifAbsent: anExceptionBlock
  self subclassResponsibility
```

Behavior-Defined Class

- When creating a new class, define its public protocol and specify its behavior without regard to data structure (such as instance variables, class variables, and so on).

- For example:
  Rectangle
- *Protocol:*
  area
  corners
  intersects:
  contains:
  perimeter
  width
  height
  insetBy:

Implement Behavior with Abstract State

- The public behavior of a class is identified but the actual implementation of this behavior is undefined.

- If implied state information is needed in order to complete the implementation details of behavior, identify the state by defining a message that returns that state instead of defining a variable.

- For example, use
```
Circle>>area
        ^self radius squared * self pi
```

- not
```
Circle>>area
        ^radius squared * pi.
```

Identify Message Layers/Encapsulate Concrete State

- How can methods be factored to make the class both efficient and simple to subclass?

- Identify a small subset of the abstract state and behavior methods which all other methods can rely on as kernel methods.

```
Circle>>radius
        ^???
Circle>>pi
        ^???
Circle>>center
        ^???
Circle>>diameter
        ^self radius * 2
Circle>>area
        ^self radius squared * self pi
```

Simple Coding Practices Promoting Design

Same Level of Abstraction

```
Controller>>controlActvity
        self controlInitialize.
        self controlLoop.
        self controlTerminate
```

About Methods

- Avoid long methods
- A method: one task
- Avoid Duplicated code
- Reuse Logic

Tell, Don't Ask!

```
MyWindow>>displayObject: aGrObject
  aGrObject displayOn: self
```

And not:

```
MyWindow>>displayObject: aGrObject

  aGrObjectisSquare ifTrue: [...]
  aGrObject isCircle ifTrue: [...]
  ...
```

Don't Check Results of Actions

```
MyStuff>>doit
  self countDays.
  dayCount status =~ 0
        ifTrue: []
         ifFalse: [dayCount Status = 1
                          ifTrue:[....]

 MyStuff>>doit
   [self countDays] on: Error do: [:ex| ...]
```

Good Signs of OO Thinking

- Short methods
- No dense methods
- No super-intelligent objects
- No manager objects
- Objects with clear responsibilities
  - State the purpose of the class in one sentence
- Not too many instance variables