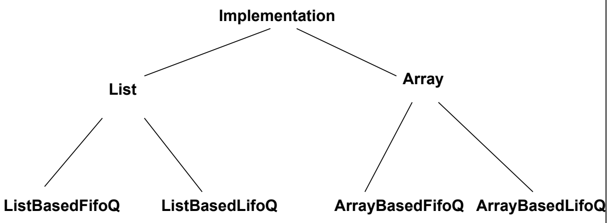
T3-3
Designing a Queue

The Problem

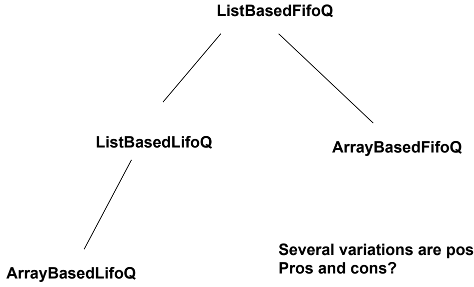
- We want to design and implement a type called queue:
  - The requirements state that it must use at least two distinct implementations namely the linked list and the array
  - any queue must understand the procedures *enqueue* and *dequeue*, plus a few utility ones such as *size*, *includes*, etc.
  - two policies must be available: FIFO and LIFO
  - we envision other implementations in the future…
  - we are concerned with performance AND with ease of evolution AND with variability
    - » C++ is our current target platform
    - » We want to avoid code redundancies! (WHY??)

Missing Interfaces



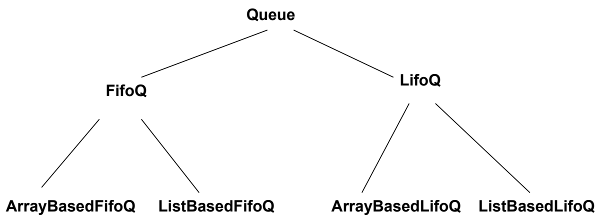
Let's start with "diagrammatic overdesign…" without using UML!
In this first case, we violate the "stable interface" principle…

Minimalist



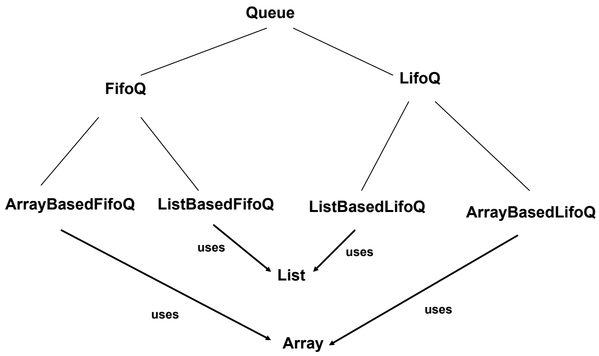
Several variations are possible here!
Pros and cons?

Duplicated Implementations

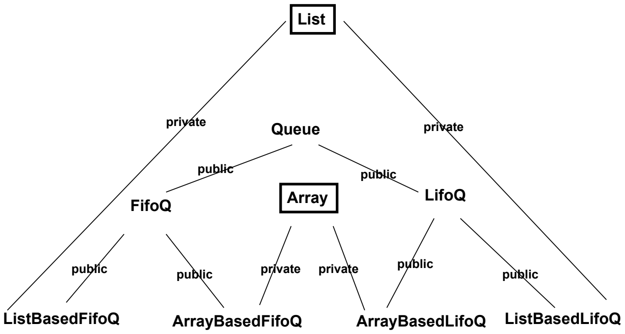


Do you see the problem?

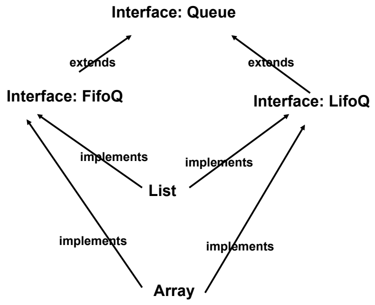
Using Delegation



C++ Mixin



JAVA Interfaces and Implementations



Is there a problem?

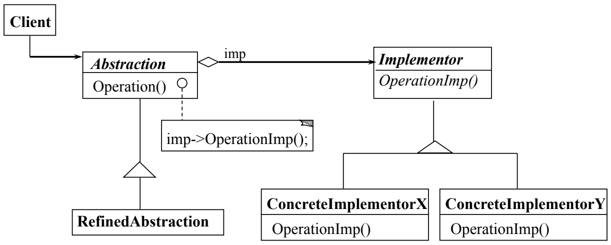
Design Choices

- Relationship between siblings
  - Instead of LIFO and FIFO queues, think of sets and bags:
    - » Set as parent, Bag as parent, Siblings, Independent?
- Subtyping
  - Do we want to transparently use one for the other?
- Implementation classes as parents?
- Implementation duplication
  - If we have interface classes, will the implementations be duplicated?
- Bottom line:
  - Can we agree on a solution without knowing the requirements?
    - » Performance may or may not be an issue…
  - Even if we agree on one solution, the picture leaves lots of room for good and bad implementations…
  - Should we attempt to capture a *space* of solutions?
    - » This means understanding variability, i.e., the 'degrees of freedom' of the system.
- Now let's look at the code!

Things to Look For

- The main program:
  - main() shows we are using subtyping in testing
  - Compiler restrictions?
    - » Don't pass a new in a parameter
  - The output_invalid and its bug: do you see it?
- Using 2 hierarchies:
  - Why virtuals in the implementation root class?
- Queue class:
  - The mystruct protected variable: code that is oblivious of the actual implementation in the subclasses
    - » How does it work in the subclasses?
  - The use of virtual: why not enqueue?
  - The costs of size, enqueue, and dequeue: each is different…

Bridge Structure



Bridge Consequences

- implementation is separated from abstraction
  - allows for run-time configuration of implementation
  - no compile-time dependencies on implementation
    - » change in implementation doesn't require recompilation
  - Abstraction-Implementation bridge forms a *layer* that isolates the rest of the system from the underlying implementation
  - implementation and abstraction can evolve independently
  - clients are shielded from implementation details

Bridge Implementation

- if there is only a single implementor there is no need for the abstract implementor
- choosing an implementor
  - at the time of constructing the abstraction by passing a parameter to the abstraction constructor
  - after abstraction is created, chose an implementation depending on conditions, e.g. linked list for small collections and hash table for large
  - delegate to a factory object
- multiple inheritance option
  - inherit publicly from Abstraction and privately from a ConcreteImplementor
    - » statically binds abstraction to implementation
    - » not a true Bridge implementation
    - » similar in structure to Adapter (Class)

Discussion of Structural Patterns (1)

- look very similar, but what distinguishes them are their intents
- Adapter and Bridge both use indirection but for different reasons
  - Adapter to match an interface a client expects to the one an adaptee provides, and bridge to provide a client access to different implementations transparently
  - Bridge provides stability to clients in presence of implementation evolution

Discussion of Structural Patterns (2)

- Composite, Decorator
  - composite and decorator both use recursive composition but for different reasons: composite for bringing apparent uniformity to a family of arbitrarily complex structures, and decorator for adding responsibilities to an object in an open-ended way
  - decorator uses object composition to
    - » avoid explosion in number of classes resulting from using subclassing to add responsibilities
    - » allow for dynamically adding responsibilities