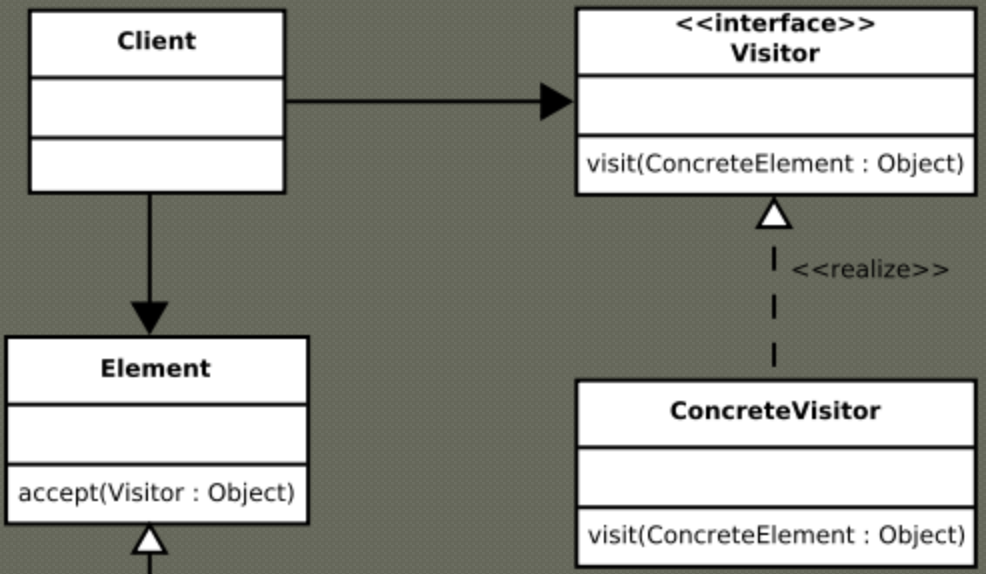
Understanding the Extended Visitor Pattern

Because OOPs is way more fun!

The Standard Visitor Pattern

- You want to decouple the object logic from the object (= decouple the data structure from the algorithms to process it)…
- You now have two things…
  - Host: The Object (data structure)
  - Visitor: The Object Logic (algorithm)

Wikipedia Says…



VISITOR

HOST

MULTIPLE CONCRETE HOSTS

ONE VISIT METHOD PER CONCRETE HOST:
-visitConcreteHost1(host)
-visitConcreteHost2(host)
-visitConcreteHost3(host)
-etc.

Limitations of the Normal Visitor Pattern

- With just the Visitor Pattern, each concrete Visitorm must contain a method (case) corresponding to each concrete host.
  - If you have three concrete objects, Truck, Car, and Bus, all visitors must have (read: the top-level Visitor interface has) a method corresponding to each concrete host, i.e. truckCase(), carCase() and busCase(). If you want to make the Visitor Move, you need to implement all of these methods. Adding another type of Visitor, perhaps Stop, would require implementing N concrete Visitor methods (cases) where N is the number of different types of objects to process over (hosts).

Limitations of the Normal Visitor Pattern, *cont.*

The problem arises if you want to add more hosts. In the standard pattern, the number of host is considered invariant, so it is unsurprising that adding more hosts causes problems.

To add one more host requires the addition of one more method (case) to the abstract Visitor interface. This means that one more concrete case must be added to *all existing visitors*.

Traditional Work-arounds

To circumvent the problem of invariant number of hosts, a traditional work-around is to use an abstract top-level Visitor *class*.

- This class implements all cases with no-op behavior.
- Concrete Visitors, override only the methods they need.
- Thus to add another host, is to add another no-op case to the abstract Visitor superclass.
- No modification of existing concrete Visitors is required.

<u>Limitations:</u>

- Cannot add or subtract hosts dynamically.
- All visitors must have the same default behavior for any cases they don't explicitly override.

The Extended Visitor Pattern

(from Nguyen and Wong, "Design Patterns for Self-balancing Trees", OOPSLA 2002)

Instead of multiple cases, one for each host, replace them all with a single, parameterized case, "caseAt(index)":

- Each host calls the case with a unique index value, i.e $host_i$ calls caseAt(i) on the visitor.
- Internally, a visitor can implement caseAt(index) any way it wishes.
  - One way is for the Visitor to hold a dictionary of lambdas, keyed to the index value. The return value is simply the execution of the lambda associated with the given index.
  - If a lambda for a given index is not found, a default lambda is run, giving consistent, well-mannered and predictable default behavior.

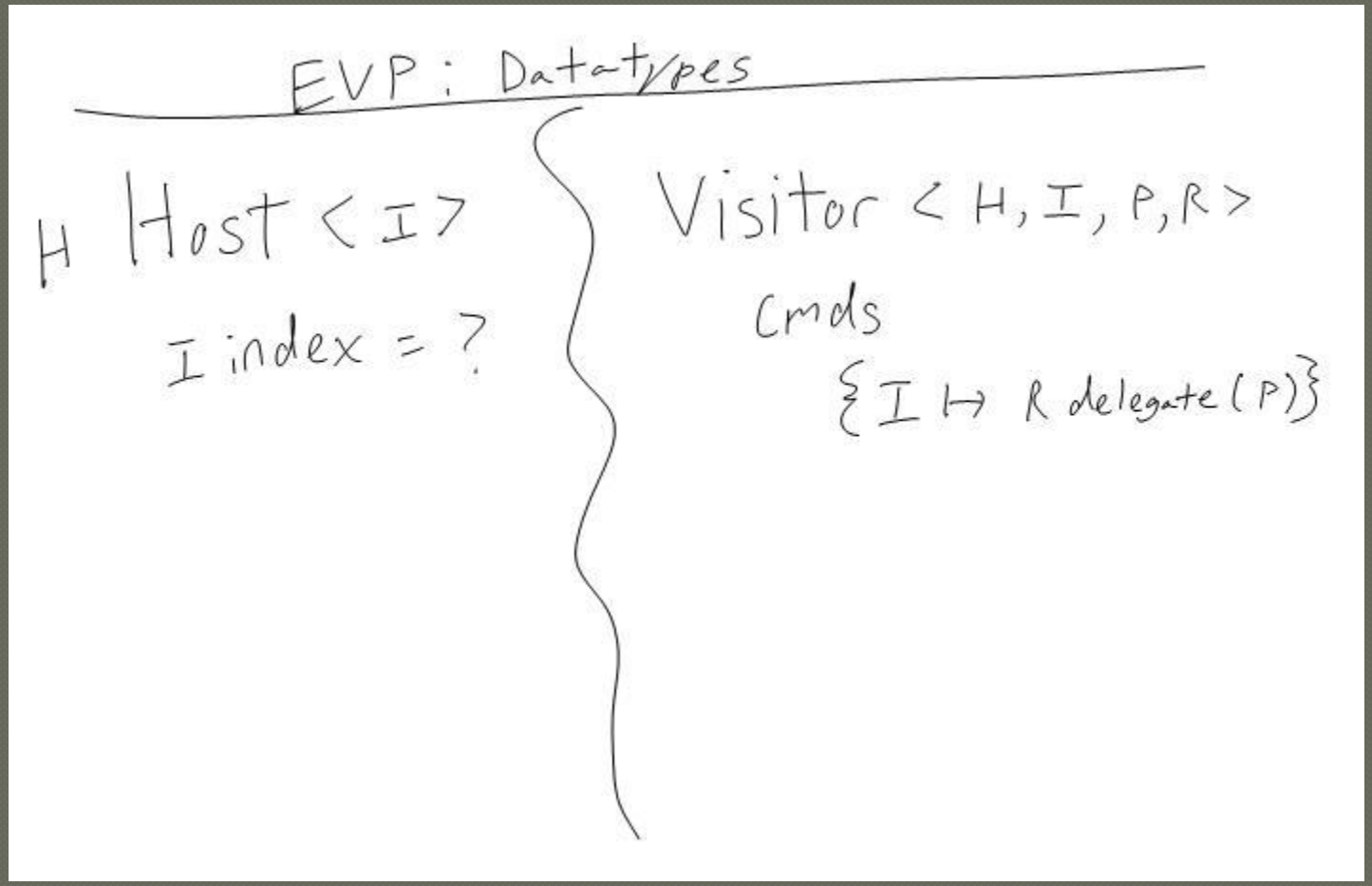
Advantages of EVP

- Dynamic addition and subtraction of hosts.
  - Adding or removing of hosts is simply a matter of adding or removing lambdas associated with that host's index value.
- Dynamic modification of behavior
  - Changing the associated lambda for a host's index will immediately change that visitor's behavior for that host.
- Variable default behavior
  - Default behavior is definable at the concrete Visitor level by setting individual default lambdas, so each visitor can have different default behaviors.
- (C# only) The ability to define hosts by generic types.
  - With run-time generics, the host's type information is usable as an index value, e.g. Packet<T> can use T as an index value and thus Visitors can be defined to handle a dynamically changing number of T types.

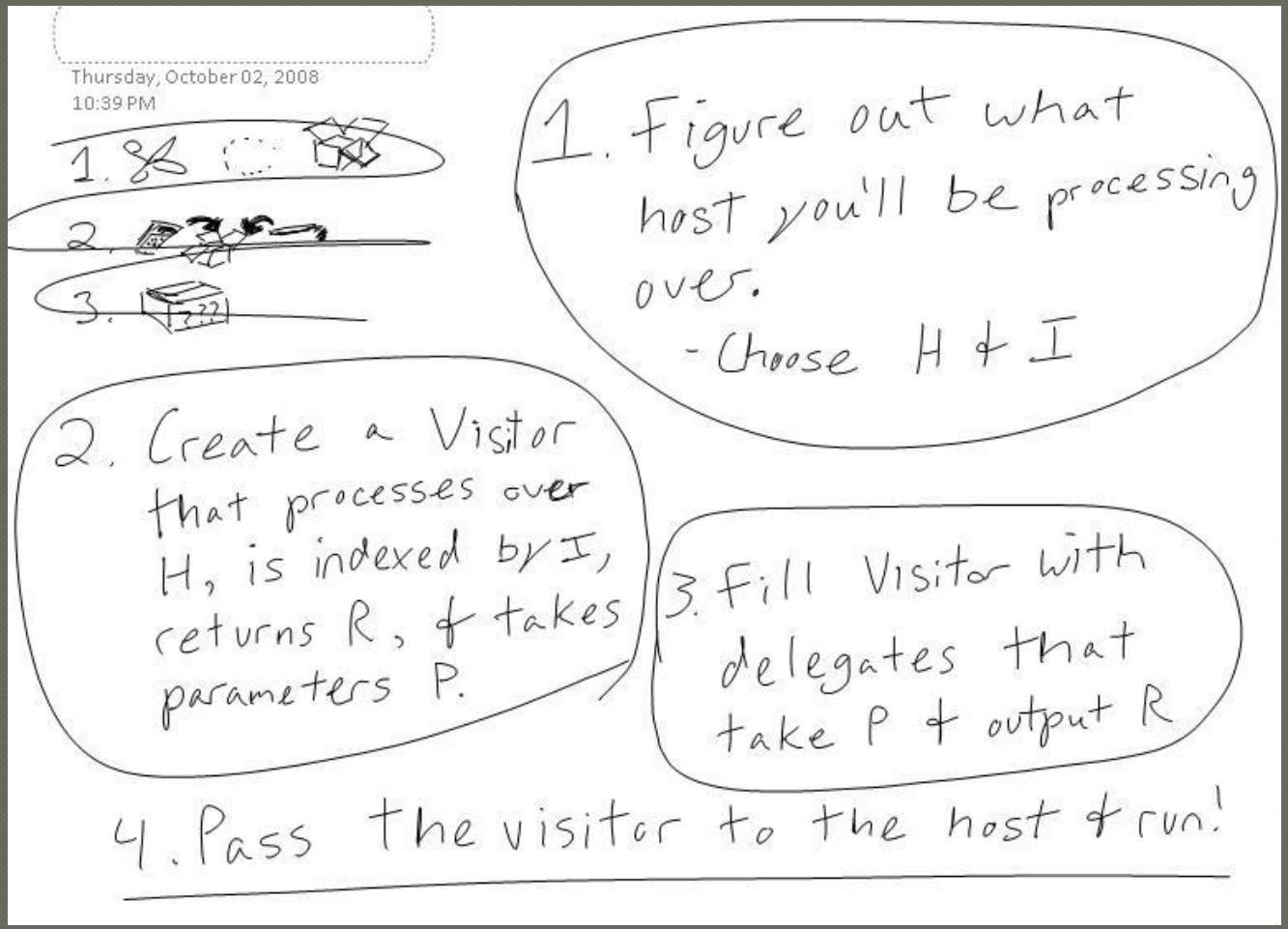
All That Jazz

- With the EVP, we now have more options for building, choosing, and swapping out algorithms that operate on the Host.
- The EVP design pattern is characterized by 4 generic types:
  - H = the superclass type for all concrete hosts
  - I = the type of the index value being used
  - P = the type of the vararg input parameter to the execution (acceptance) of a visitor.
  - R = the return type of a visitor execution.

Datatypes


EVP : Datatypes
H Host < I >
I index = ?
Visitor < H, I, P, R >
Cmds
{ I ↦ R delegate (P) }

Usage


Thursday, October 02, 2008
10:39 PM
1.
2.
3.
1. Figure out what host you'll be processing over.
- Choose H & I
2. Create a Visitor that processes over H, is indexed by I, returns R, & takes parameters P.
3. Fill Visitor with delegates that take P & output R
4. Pass the visitor to the host & run!

Conclusion

- Hopefully you now know the motivation for the changes in the Visitor Pattern that lead up to the Extended Visitor Pattern.
- Now we just need one MoveVisitor for any type of object.

- If these slides weren't clear enough, just let me know!