# SWEN 262

*Engineering of Software Subsystems*

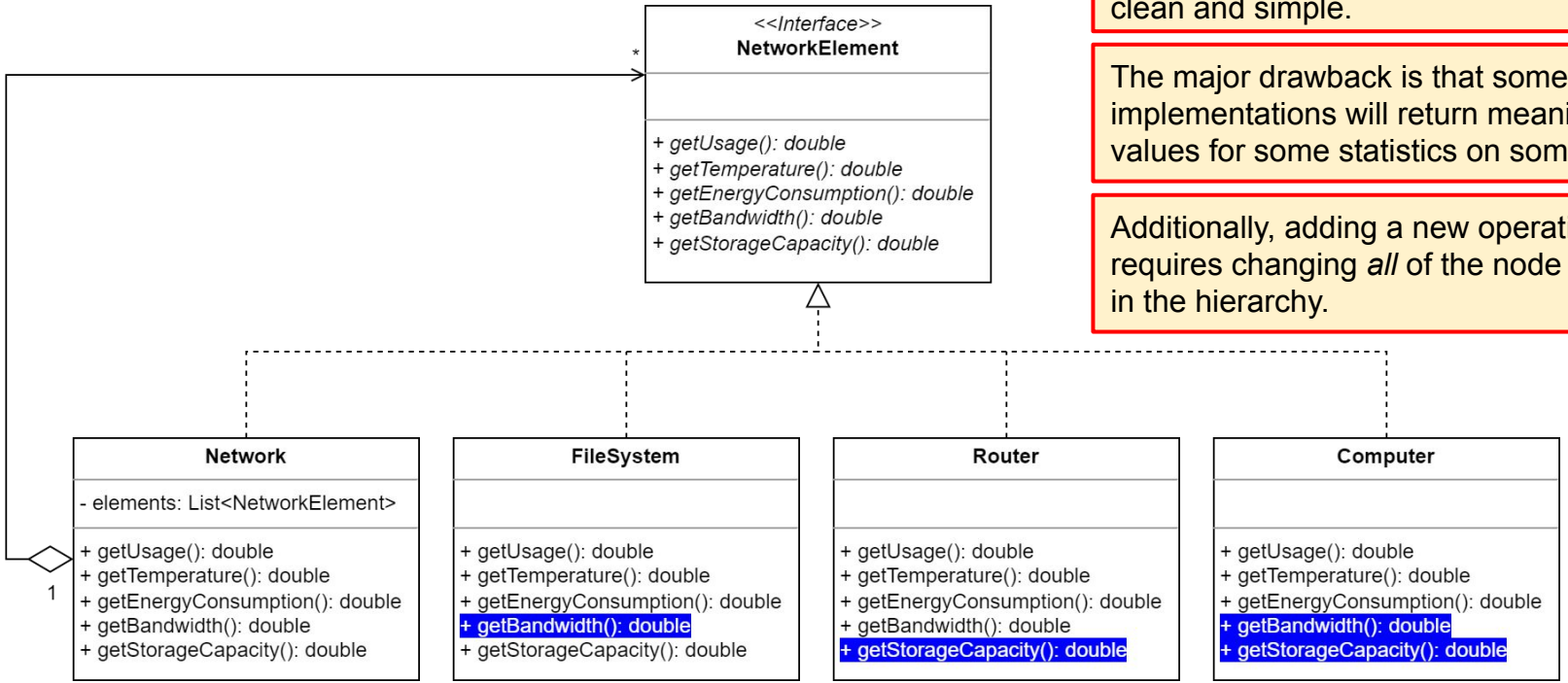
# Composite Design Challenges

- Recall one of the major design challenges with the Composite pattern: *leaf-specific operations*.
  - Not all leaves in the hierarchy will need the same methods.
- The Network Administrator Console example.
  - Some statistics make sense for all network elements:
    - usage
    - temperature
    - energy consumption
  - Some make sense for some, but not others:
    - storage capacity (file systems)
    - bandwidth (subnets, routers)

Q: What is the appropriate solution to this problem?

# Leaf as Superset

Here the *NetworkElement* interface defines all possible operations.




Therefore, each implementing class must provide implementations for every method, whether or not the specific statistic is available for the element type.

The major advantage here is that every node in the tree implements the same interface, making code to gather statistics clean and simple.

The major drawback is that some concrete implementations will return meaningless values for some statistics on some nodes.

Additionally, adding a new operation requires changing *all* of the node subclasses in the hierarchy.

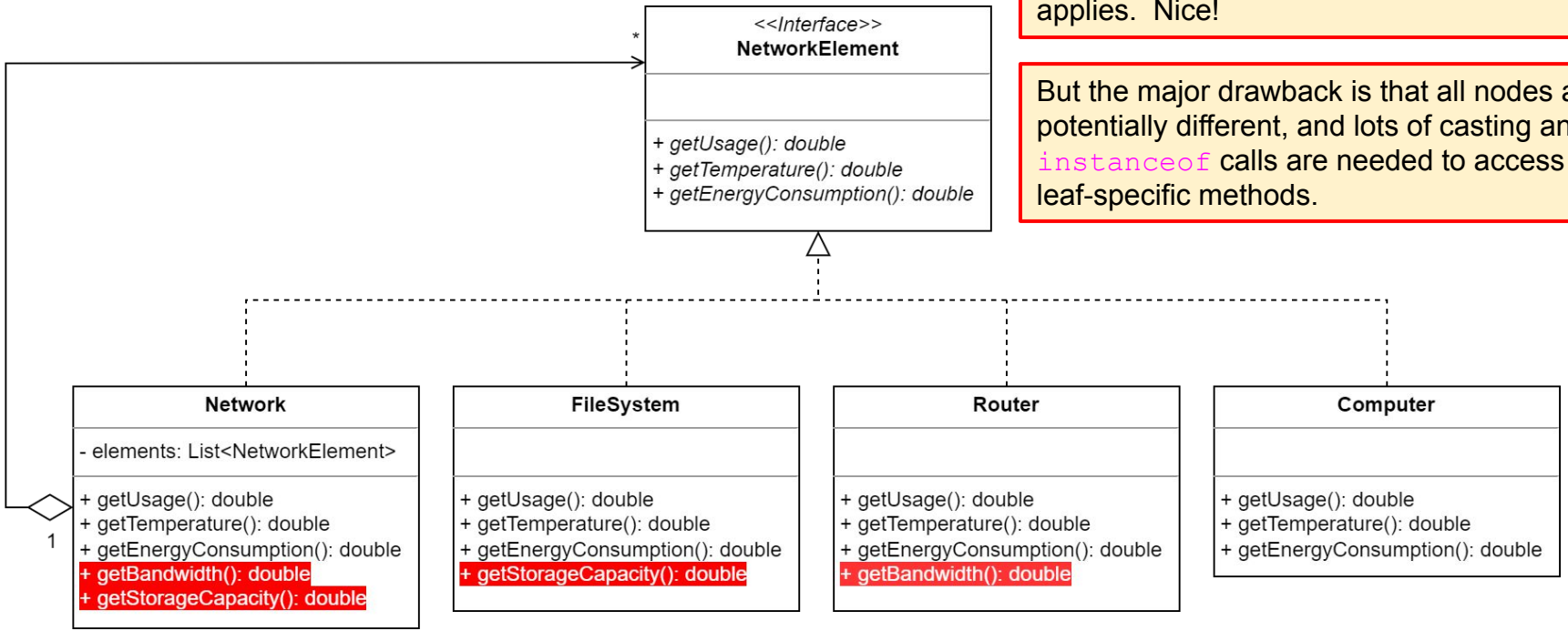
# Different Leaves

Here the *NetworkElement* interface defines only *common* operations. Each implementing class adds other methods as appropriate.

The major advantage here is that every node in the tree implements *only* the methods that make sense for that specific type of node.

Adding a new operation also only requires changing the nodes to which that operation applies. Nice!

But the major drawback is that all nodes are potentially different, and lots of casting and `instanceof` calls are needed to access leaf-specific methods.



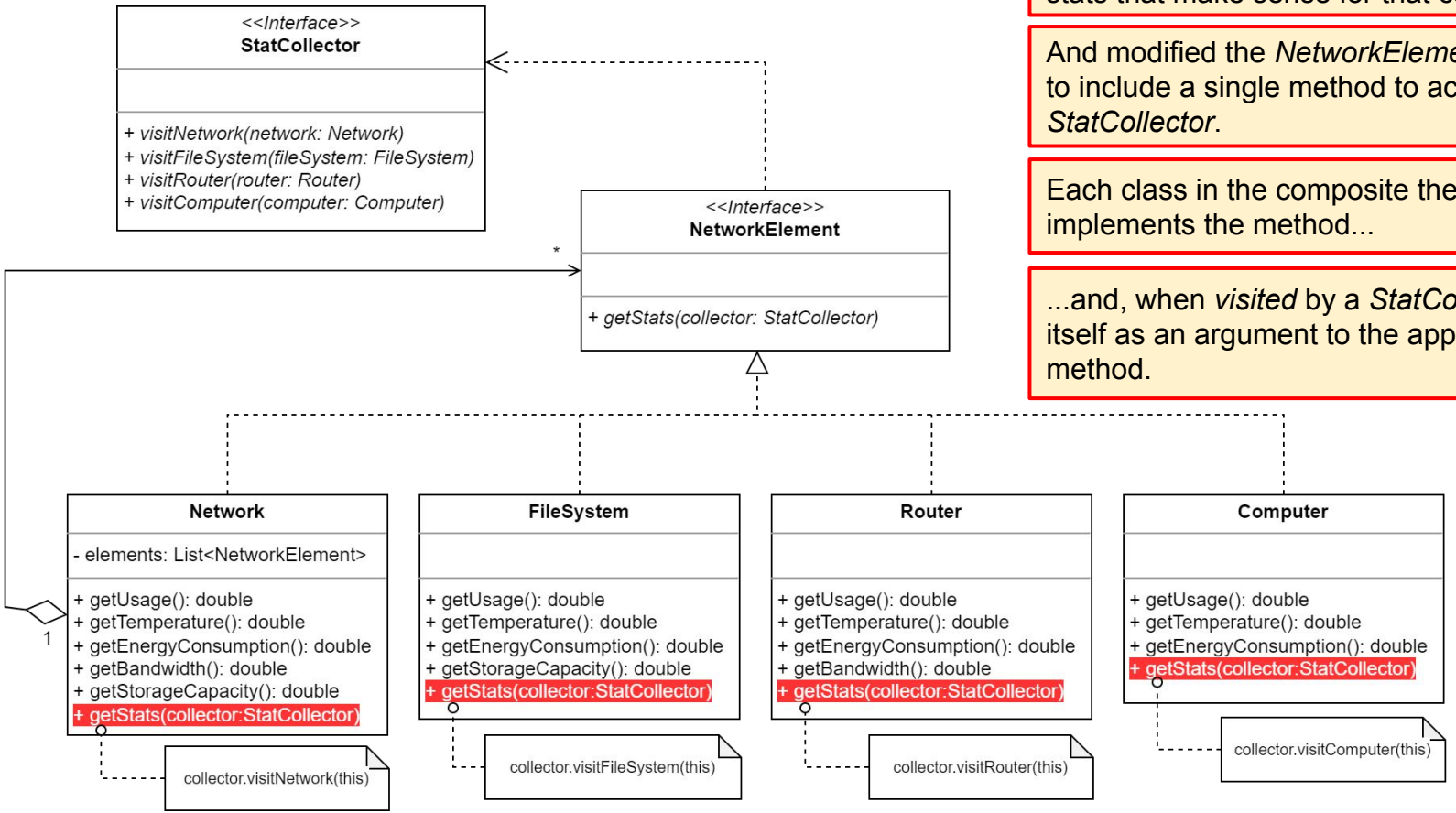
# Stat Collector Class

What if instead, we defined a *StatCollector* class with a method that, given a specific kind of network component, collects only the stats that make sense for that component?

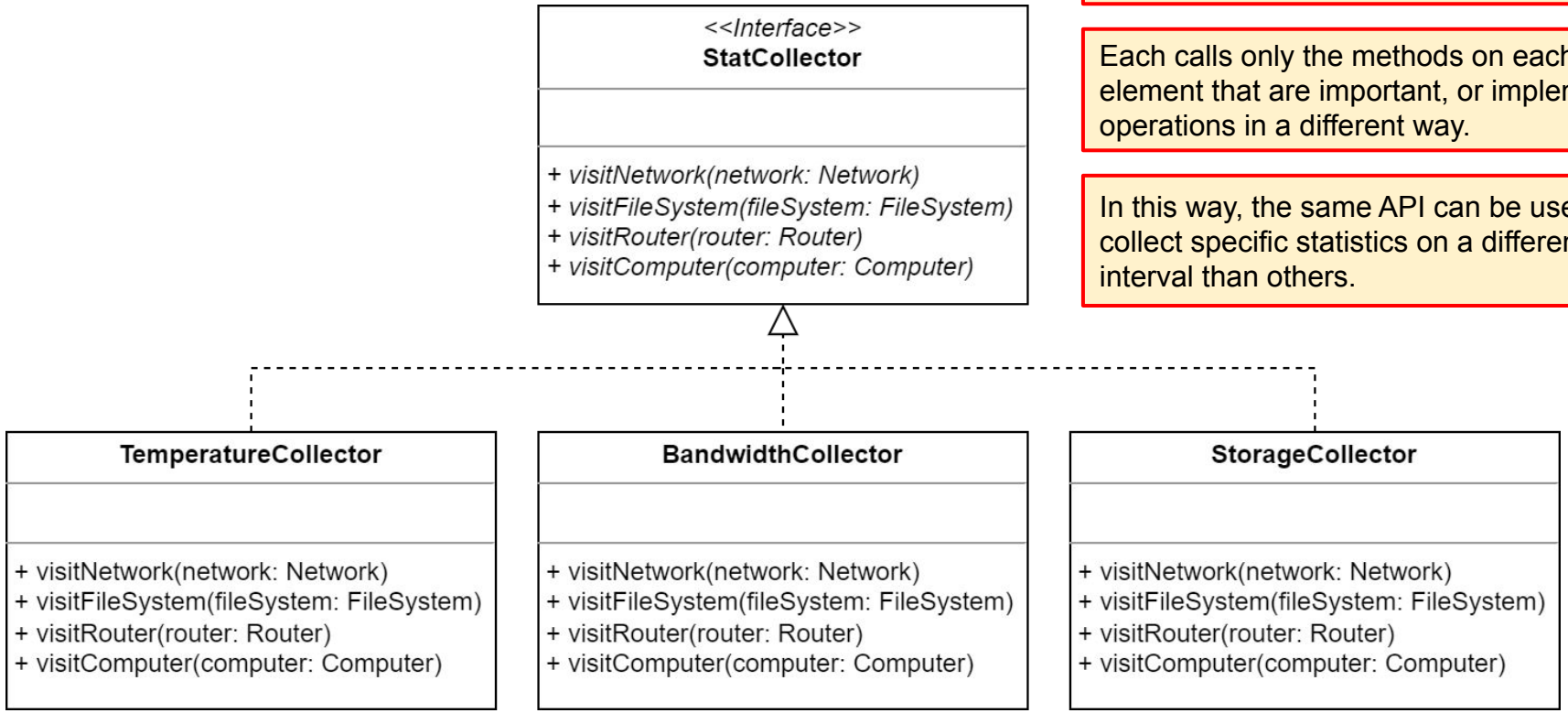
And modified the *NetworkElement* interface to include a single method to accept a *StatCollector*.

Each class in the composite then implements the method...

...and, when *visited* by a *StatCollector*, pass itself as an argument to the appropriate method.



# Stat Collector Class

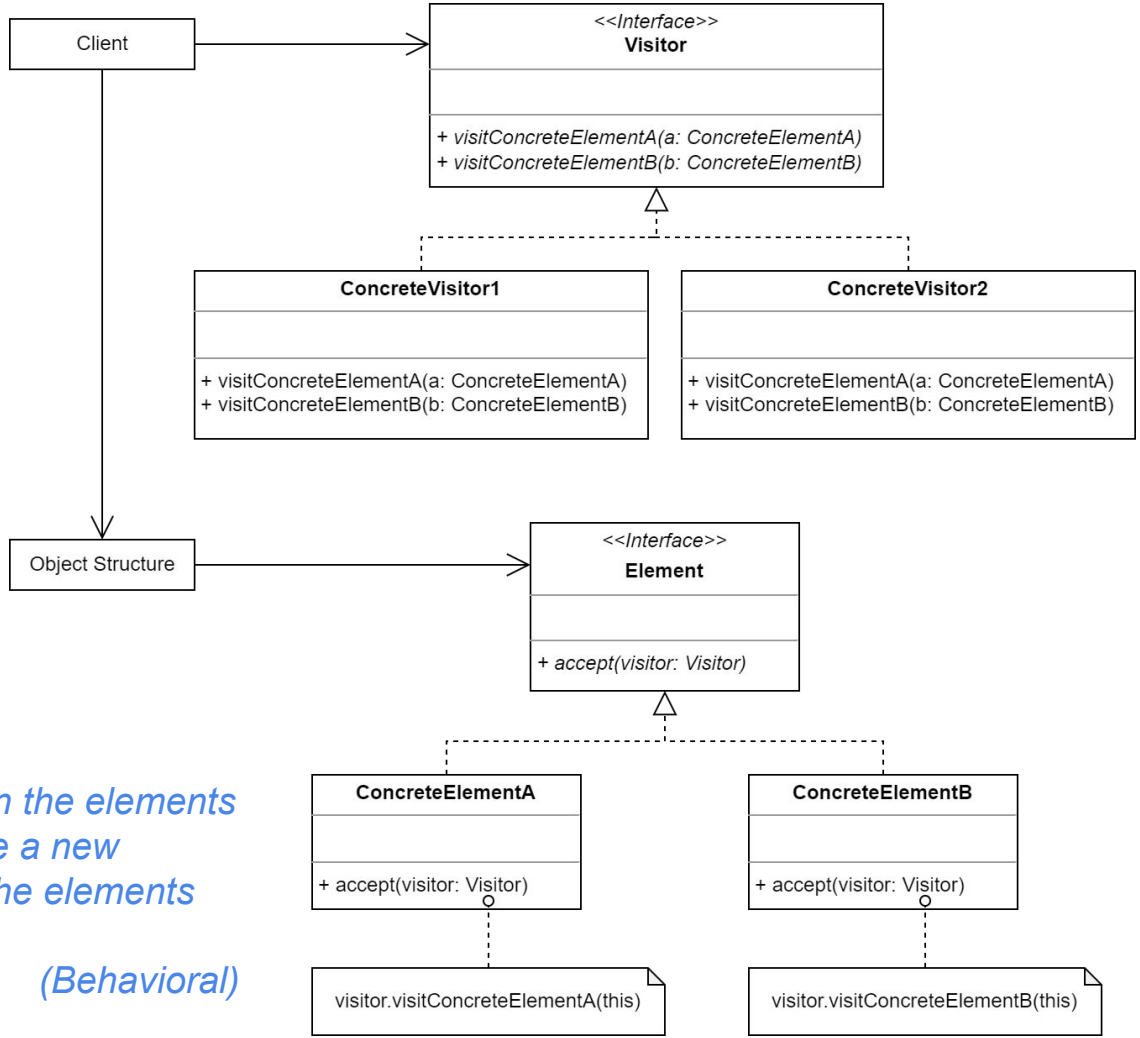



It's also possible to create subclasses to collect specific statistics, e.g. a *temperature collector*, *bandwidth collector*, *storage collector*, etc.

Each calls only the methods on each element that are important, or implements operations in a different way.

In this way, the same API can be used to collect specific statistics on a different interval than others.

# Visitor




## Intent

*Represent the operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.*

*(Behavioral)*

# How Does Visitor Work?

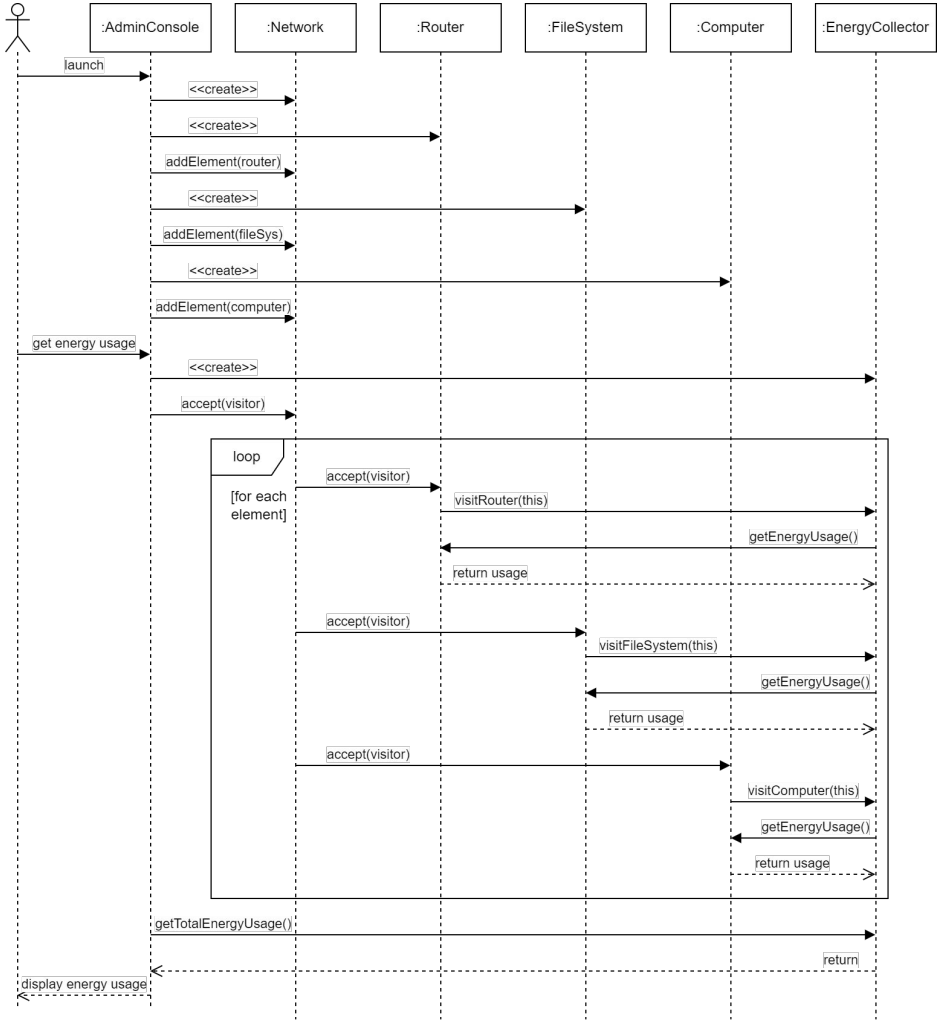
- The *object structure* is not necessarily a data structure (like a tree). It can be any class structure. Each class that makes up part of the object structure is an *Element*.
- The *Visitor* interface defines a method for each concrete *Element* in the object structure.
  - For example `visitFileSystem(FileSystem fs)`, `visitNetwork(Network n)`, etc.
- The *Element* interface defines a visit method that accepts any *Visitor.*
  - For example visit(StatCollector sc).
- An *operation* is some task that needs to be performed on the *Elements* in the object structure.
  - For example, collecting network statistics.
- A concrete implementation of the *Visitor* interface *is* an operation.
  - Each implementation is a new operation.
  - New operations are added by adding new implementations.

In order for Visitor to work, some object in the system must be able to iterate through the elements of the object structure.

In this example, the Network iterates over the elements that have been added to it and calls "`visit(StatCollector)`" on each.

Each element calls the corresponding "visit" method on the `StatCollector`, and passes *itself* in as the argument e.g. the `FileSystem` calls "`visitFileSystem(this)`".

This enables the `StatCollector` to call any method defined by that object - no `instanceof` or casting is necessary.



# Visitor

- The primary purpose of Visitor is performing an operation on the elements in an object structure (e.g. a tree).
- The operation is typically performed on the entire structure.
    - A concrete Visitor *visits* each element to perform the operation on the element, e.g. by traversing through the nodes in a tree.
- The pattern allows a *separation of concerns* for the maintenance of the *object structure* and performing operations on the elements of that structure.

# Visitor

As with most design choices, Visitor is not without its potential drawbacks.

- Adding new concrete element classes is difficult.
  - All visitors must change to add a new method to visit the new concrete element.
  - It may be a better choice to put new functionality in the structure instead.

| Frequency of... | | Adding New Classes is | |
|---|---|---|---|
| | | Rare | Frequent |
| Adding New Operations is... | Rare | Either | Structure |
| | Frequent | Visitor | Hard |

- Visitor also assumes that the element interfaces include sufficient access to perform operations.
  - If not, encapsulation may need to be “broken,” e.g. using *friends* or *package access*.

# Consequences

- *Adding new operations is easy by adding a new visitor.*
- *A visitor gathers related operations into a single object rather than spreading them over the object structure.*
  - *Conversely, visitors keep unrelated operations separate.*
- *Visitors can visit objects that do not have a common parent class.*
- *Visitors can accumulate state as they traverse the object structure.*
- *Adding new concrete element classes is harder as all of the visitor classes must change.*
- *Encapsulation may need to be broken.*
- *We “bend” the “behaviors follow data” principle in exchange for making new operations easier to add to a complex object structure.*