# SWEN 262

## Engineering of Software Subsystems

*Composite Pattern*

# Network Administration

1. A *network* comprises network elements of several types.
    a. Computers
    b. Network-attached File Systems
    c. Routers
    d. Subnets
        i. Subnets also contain network elements, including other subnets.
2. The administration console must display cumulative information about the network or individual network elements including:
    a. Temperature
    b. Energy consumption
    c. Usage
    d. Available storage
    e. Status of diagnostic checks

Q: How would you go about implementing these requirements?

# Conditionals/instanceof

```
public double getUsage(Object element) {
  double usage = 0;
  if(element instanceof Network) {
    Network network = (Network)element;
    List<Object> elements =
      network.getNetworkElements();
    for(Object element : elements) {
      usage += getUsage(element);
    }
  } else if(element instanceof FileServer) {
    FileServer server = (FileServer)element;
    usage += server.getUsage();
  } // and so on...

  return usage;
}
```

Use `instanceof` and casting to check for network elements that provide the data that you are looking for, e.g. usage data (whatever that means).

Q: What are some of the drawbacks to this approach?

We have already discussed that a long set of conditionals is a code smell, but using `instanceof` and casting is also a code smell that often indicates a missed opportunity for polymorphism.

Adding new kinds of network elements would also require breaking OCP; the conditional would need to be modified to handle the new types.

Let's take a look at another solution...

# A Component Interface

Begin by defining an `interface` to represent a ***component*** in the network. This interface will be implemented by each of the different kinds of network elements.

```
public interface NetworkElement {
  public double getTemperature();
  public double getUsage();
  public double getBandwidth();
  public double getStorage();
  public Status getStatus();
}
```

Next, create a ***concrete component*** for one of the network elements, e.g. a `FileServer`.

```
public class FileServer implements NetworkElement {
  public double getTemperature() {
    // return current temperature
  }

  public double getUsage() {
    // return current usage
  }
  // and so on...
}
```

We call these individual concrete components ***leaves***. Every leaf ***is a component***.

# A Composite Element

```
public class Network implements NetworkElement {

  private List<NetworkElement> elements =
    new ArrayList<>();

  public void add(NetworkElement element) {
    elements.add(element);
  }

  public void remove(NetworkElement element) {
    elements.remove(element);
  }

  public double getUsage() {
    double usage = 0;
    for(NetworkElement element : elements) {
      usage += element.getUsage();
    }
    return usage;
  }
  // and so on...
}
```

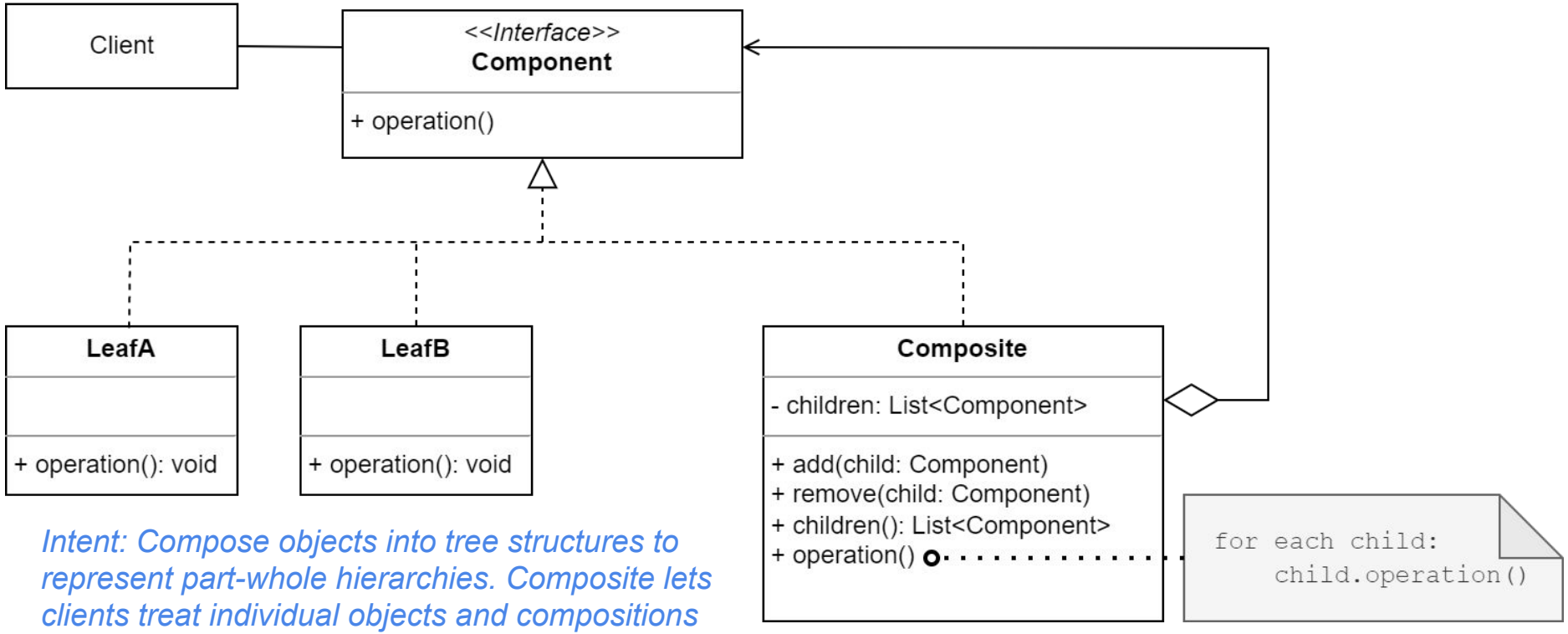
Next, create the `Network` class, which by virtue of implementing the `NetworkElement` interface, ***is a component***. It must implement all of the same methods, and can be treated the same as any other `NetworkElement` (polymorphism!).

The major difference is that `Network` is also a ***composite*** of `NetworkElements`, and so it will need to maintain a collection of ***children***.

This will require methods to ***manage*** child components.

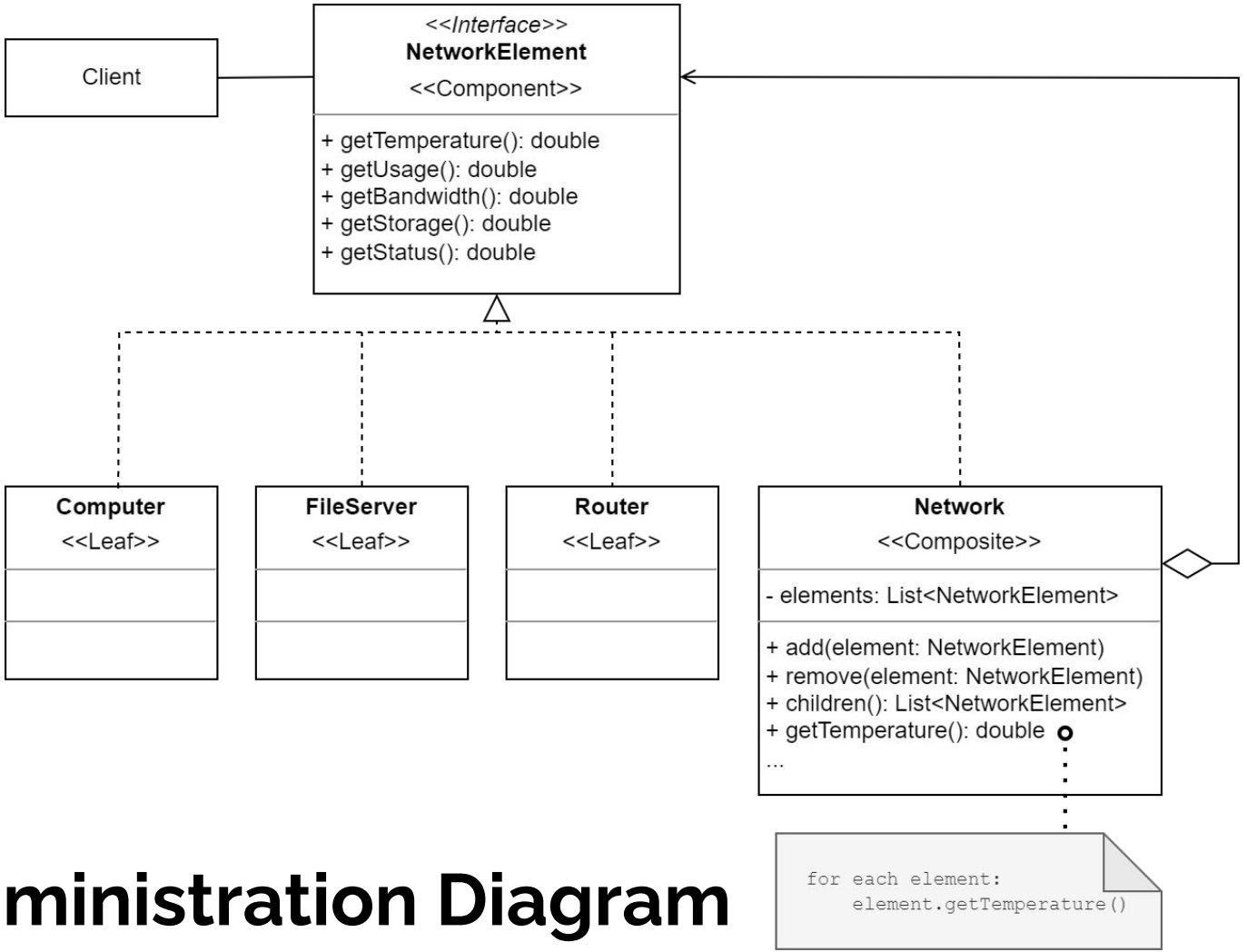
Its `NetworkElement` methods may return a value, collect information from the child components, or a combination of the two. This will be seamlessly transparent to the caller because the `Network` can be treated like any other ***component***.

# GoF Composite Structure Diagram




*Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.*

*(Structural)*




# Network Administration Diagram

# GoF Pattern Card

Sorry about the eye chart, but this is a lot of information to pack into one slide!

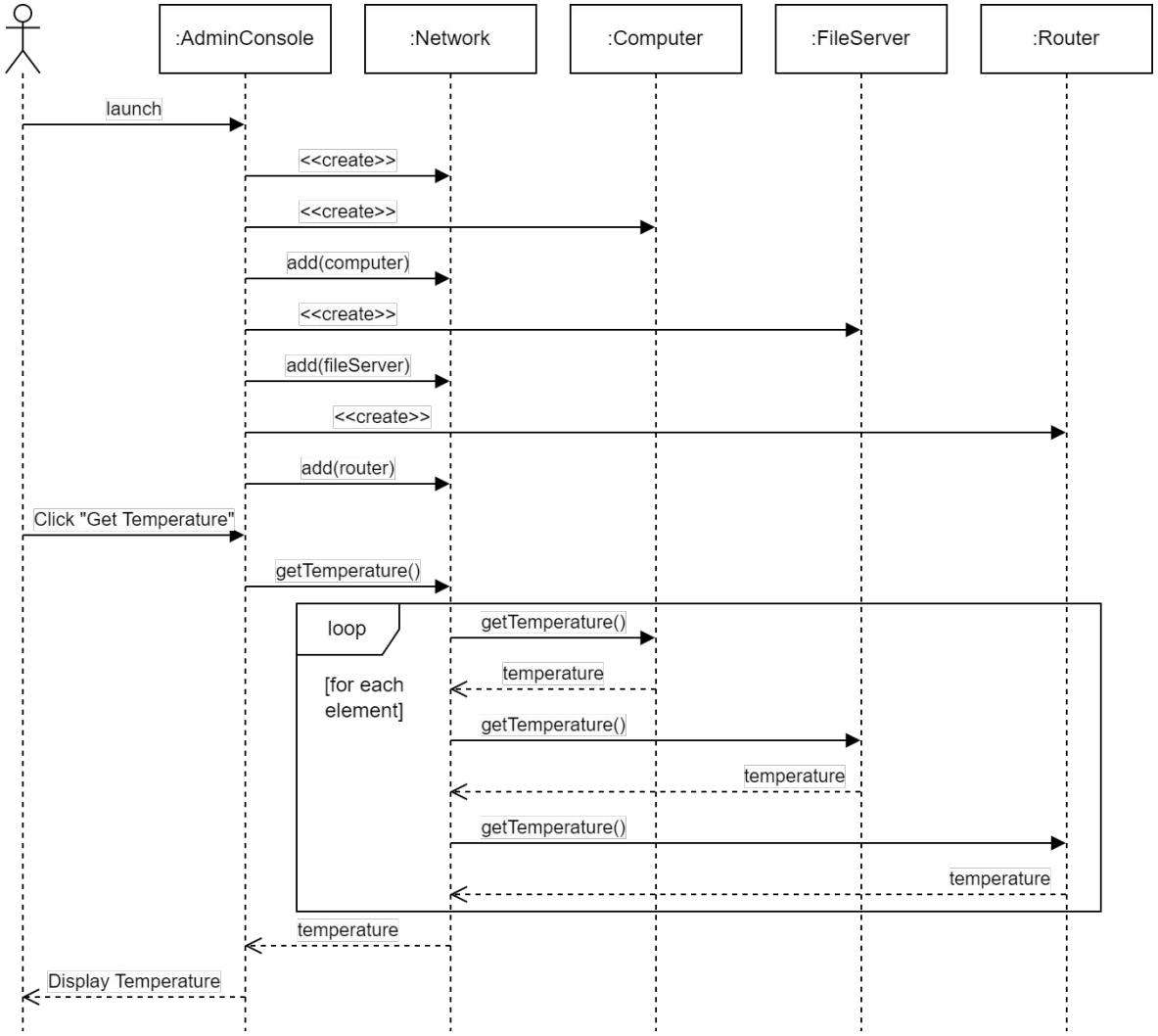
Note that each participant has *at least* 2-3 sentences of description.

Also note that each ***leaf*** is documented ***separately*** - they are ***not*** combined into a single row.

In your documentation it is OK for a card to span multiple pages for readability.

| Name: *Network Administration Subsystem* | | GoF Pattern: *Composite* |
|---|---|---|
| **Participants** | | |
| Class | Role in Pattern | Participant's Contribution in the context of the application |
| *Admin Console* | *Client* | *The user interface for system administrators. The UI is used to collect and display information about network connected devices at the network, subnet, and individual device level.* |
| *NetworkElement* | *Component* | *Defines the interface and operations that all network elements must support. This includes methods for collecting temperature, usage, bandwidth, storage, and the status of diagnostics.* |
| *FileServer* | *Leaf* | *Represents a network connected file server. Provides information about the file server including available storage and the status of diagnostics.* |
| *Computer* | *Leaf* | *Represents a network connected personal computer. Provides information about temperature, usage, and diagnostics.* |
| *Router* | *Leaf* | *Represents a network connected router. Provides information about temperature, available bandwidth, and status of diagnostics.* |
| *Network* | *Composite* | *Represents a network. A network may include any number of subnets, each of which is represented as a network. In addition, any elements connected to the network will be contained within. Most operations on the network aggregate information from connected devices.* |
| Deviations from the standard pattern: *Methods for managing children are not defined in the component interface, and so the Network is distinct from other components.* | | |
| Requirements being covered: *1. A network comprises file servers, computers, routers, and subnets that may be nested to an arbitrary depth. 2. Information including temperature, usage, bandwidth, storage, and diagnostics can be collected.* | | |

# Sequence Diagram: Get Temperature



# Composite

There are several *consequences* to implementing the composite pattern:

- *Defines a class hierarchy consisting of leaves and composites - a tree structure.*
- *Everything is a component! Clients can treat individual objects (leaves) and composites exactly the same.*
- *Makes it easier to add new kinds of components by implementing the correct interface.*
- *Can make the design overly general by forcing dissimilar objects to implement the same high level interface.*