Object Oriented Software Engineering (COMP2003)

Lecture 4: Object Relationships

Updated: 23rd March, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

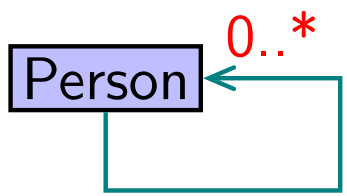
Outline

Objects

- One class can have many instances (objects).
  - A simple but very important concept!
- These objects can form complex structures among themselves. Consider this:

```
public class Person
{
    private Set<Person> friends;
    ...
}
```

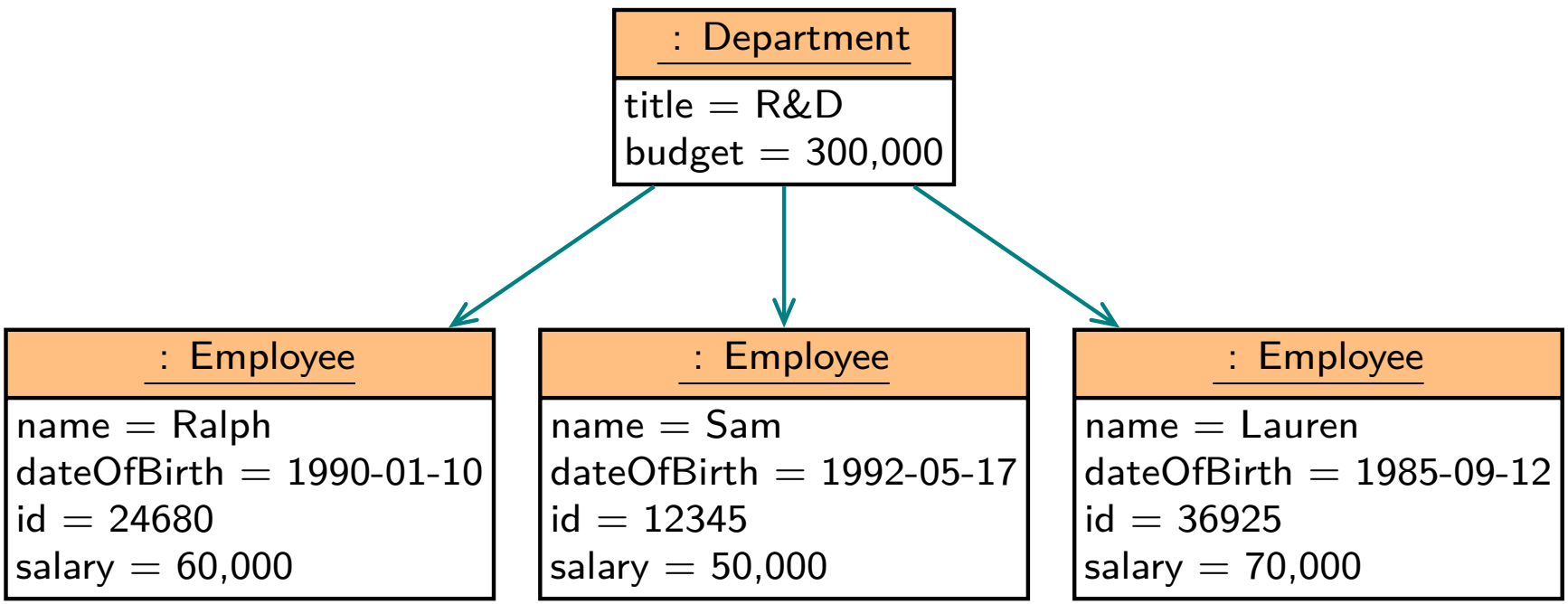
- The corresponding class diagram looks trivial:



- But the *object* structures could be very complex.
  - Imagine 100 `Person` objects, each with 10 `friends`.

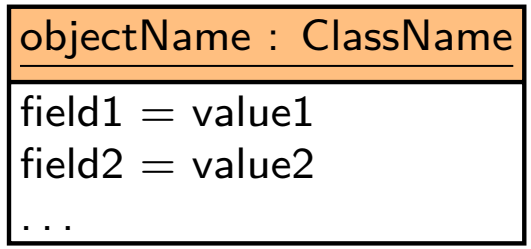
UML Object Diagrams

- We'll use *Object Diagrams* to represent object structure.
  - Rather, *examples* of object structure, since the actual objects are created at runtime.
- Say a `Department` has several `Employees`.
- We could have the following object structure:



UML Object Diagrams

- Each object is represented like this:



  - The object name is often not really needed.
- Objects associate and/or aggregate with other objects.
  - But only one-to-one – no multiplicity.
  - There is only *one* of each object.
- Objects *do not* inherit from other objects.

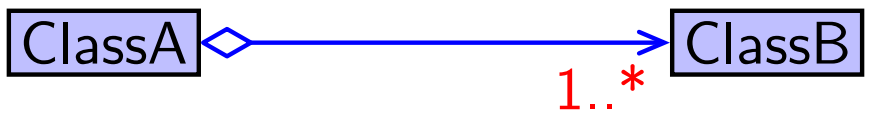
```
public class Employee extends Person { ... }
```

```
emp = new Employee(...);
```

  - Inheritance occurs between classes, not objects.
  - Only *one object* here (emp), with properties of both classes.

Recap: Aggregation vs Association

- Aggregation.
  - A "whole-part" relationship between two objects.
  - One object owns the other.
  - The owning ("whole") object has a field referring to the "part" object.



- Association (or "acquaintance"):
  - The two objects involved are equals. Neither one is the "whole" or "part".
  - As a result, associations can be circular (e.g. A refers to B, B refers to C, and C refers to A).

Accessing an Object

```
public class ShoppingCart                                    // Java
{
    private ClassB field;
    ...
    public ClassB getField()
    {
        return field;                          // This?
        return new ClassB(field);              // Or this?
    }
}
```

- How should you implement the above accessor?
- *Both* approaches are valid in different situations.
- Depends on the strength of the relationship between ClassA and ClassB.

Accessing an Object: Your Options

- How should you give access to an object field?
  1. *Don't!* (If you don't need to.)
  2. *Or* return a **reference**:

```
public Point getCentre() { return centrePoint; }
```

  3. *Or* return a **const reference** (C++ only):

```
const Point& Circle::getCentre() const {...} // C++
```

  4. *Or* return a **copy**:

```
public Point getCentre() {
    return centrePoint.clone();
}
```

  5. *Or* return a **read-only wrapper**:

```
public ReadOnlyPoint getCentre() {
    return new ReadOnlyPoint(centrePoint);
}
```

How do you Copy an Object?

- The copy constructor?

```
TheClass copy = new TheClass(original);
```

- Problem: you must know the exact class.
  - What if the original object is a *subtype* of `TheClass`?
  - This kills inheritance (just like raw objects).
- Solution: an overridable `clone()` method:

```
TheClass copy = original.clone();
```

  - This will handle subclasses.
  - Each subclass overrides `clone()` to return a copy of itself.
  - So, we can make a copy without knowing the exact class. (Polymorphism for the win.)

How do you Wrap an Object?

- Consider this:

```
public class ReadOnlyPoint // Wrapper class
{
    private Point p;
    public ReadOnlyPoint(Point p) { this.p = p; }
    public int getX() { return p.getX(); }
    public int getY() { return p.getY(); }
} // No mutators
```

- ReadOnlyPoint "wraps around" a Point.
- You can access a Point, but not modify it.
  - Point may have setters, but there's no way to call them given a ReadOnlyPoint.
- An alternative to making a copy of the Point. Why?
  - Copying can be slow, for objects that use lots of memory (not so much Point, but other cases).

Wrapping Containers

- In Java, you can get a read-only view of a list, set or map:

```
List<Thing> listView =
    Collections.unmodifiableList(myList);
```

(See the `Collections` class javadocs for more details.)

When Not to Copy/Wrap Objects

- Aggregation and association are not always so protected.
  - In Java, the `LinkedList` class does not copy/wrap objects for you.
  - In most languages, container structures don't copy things.
  - (C++ does, when you pass objects by value, it then doesn't have a choice.)
- Often the aggregation/association is *supposed* to be publicly accessible.
- Associations would be impossible to set up if accessors always returned copies.

Don't Copy/Wrap Immutable Objects

- Immutable objects pose no risk to encapsulation.
- It is totally unnecessary to copy/wrap them.

That Said...

- *Do* copy/wrap an object when it's:
  - Mutable, *and*
  - Part of the internal workings of another class.
- Usually applies to generic classes storing dates, addresses, etc. (if mutable).
- Usually applies to containers:

```
public class ShoppingCart
{
    private Set<Product> products;
    public Set<Product> getProducts()
    {
        return Collections.unmodifiableSet(products);
    }
} // *Don't* return 'products' directly.
```

The products set is part of the internal workings of ShoppingCart.

Object Communication

How do two objects communicate? Method calls.

- (Surprise! Hopefully you knew that.)

```
public class ClassA
{
    private ClassB field;

    public void methodOne()
    {
        int argument, value;
        ...
        value = field.methodTwo(argument);
        ...
    }
} // Spot the *two* communicating objects!
```

Intermediate Objects

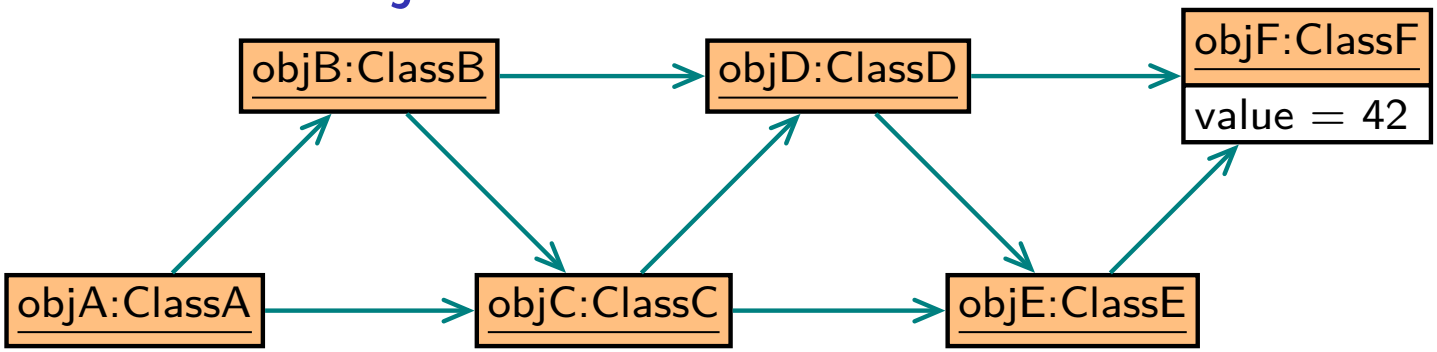
- Say object 1 wants to communicate with object 2.
- Say they must exhange complex information.
- You might do this using object 3. e.g.

```
s = new Search();
s.setSearchName("Bob");
s.setSearchEmail("@gmail.com");
s.setCaseSensitivity(false);
s.setPartialMatch(true);

entry = addressBook.search(s);
```

  - AddressBook can search by name, email, other fields, or a *combination* thereof. It can be case sensitive (or not), and partial (or complete).
  - Just one search() method will do – not hundreds.
  - We can pass complex information using a Search object.

Traversal of Objects



- Given objA, how do you obtain the value?
- Beyond the choice of path, there are two ideas:
  1. Retrieve a reference to each intermediate object, using it to get to the next one. e.g.

```
value = objA.getB().getD().getF().getValue();
```

  2. Have the intermediate objects find the value themselves:

```
value = objA.getValue();
```

(With similar code appearing inside `ClassA` – `ClassE`.)

Law (or Suggestion) of Demeter

- The "Law of Demeter" (LoD) is more advice than law.
  - Named for the "Demeter Project".
- LoD says that objects should only communicate if they are directly aggregated/associated.
  - Avoid chains of accessor calls (as on the previous slide).
  - Avoids coupling between distant classes.
- However, there is a tradeoff:
  - Intermediate classes can become very cluttered.
  - They can have many methods that simply delegate to another object.
- Always consider the LoD, but don't follow it blindly.

The Decorator Pattern

- Uses recursive aggregation.
- Creates a linked list of objects.
  - *Not* for storing a sequence of data.
  - For providing a set of add-on functionality.
- Used to avoid an unmanageable explosion of subclasses.
  - What if there are many *independent* ways to extend a class?
  - You'd need a subclass for every possible combination!
  - Or you could have one *decoration* class for each individual feature.

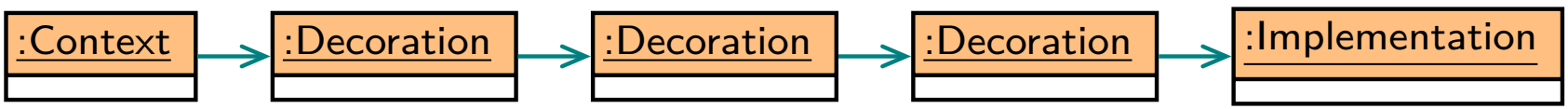
Decorator: Rationale

- Say you're writing a role-playing computer game.
- The player starts with a character from one of 10 "races":
  - Each race interacts with the game differently.
  - You write a `GameCharacter` class, and a subclass for each race.
- As the game progresses, the player's character can acquire up other special abilities and characteristics.
  - Stealth, healing powers, temporary invincibility, super-strength, etc.
- How would you implement these? Two naïve options:
  - You could put all this extra functionality in the `GameCharacter` superclass.
  - You could have extra subclasses.

Decorator: Rationale

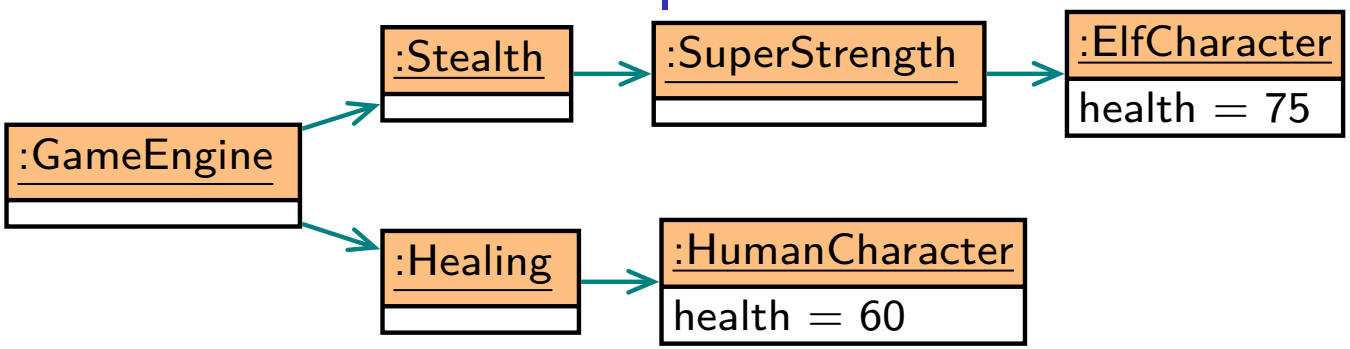
- What's wrong with everything-in-the-superclass?
  - You end up with a huge class, with many separate responsibilities.
  - Difficult to maintain.
- What's wrong with extra subclasses?
  - Consider the special abilities again: stealth, healing powers, temporary invincibility, super-strength, etc.
  - A character could acquire several of these abilities at once.
  - 10 races and 4 independent special abilities requires 160 (10×2^4) subclasses. That doubles for each additional ability.
  - Those classes are highly redundant.
- What's the alternative?
  - Have 14 subclasses of `GameCharacter`.
  - 10 for the races, and 4 for the special abilities.
  - The race classes will be self-sufficient.
  - The ability classes will be add-ons that wrap around an existing `GameCharacter` object.

Decorator: Object Structure



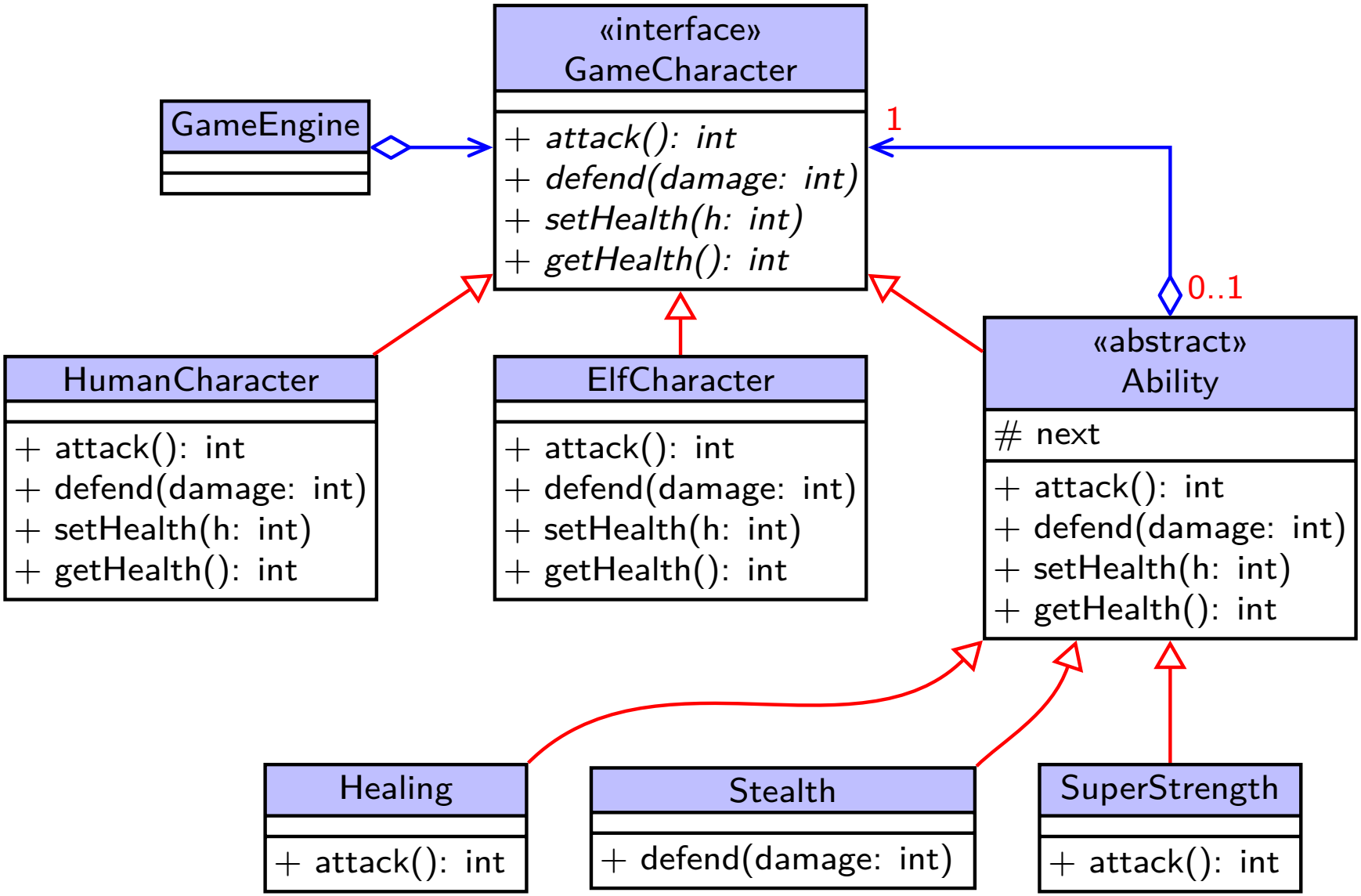
- The Decoration objects form a linked list.
- At the end, there is an object with a basic level of functionality.
- Each decoration object:
  - "Impersonates" the end object, by implementing the same interface.
  - Passes on method calls to the next object.
  - Makes certain tweaks when these method calls happen.

Decorator: Game Example



- In our game, we'll have several implementation classes: `HumanCharacter`, `ElfCharacter`, etc.
- They'll have these methods:
  - `attack()` returns the strength (int) of a character's attack.
  - `defend()` takes an int and reduces the character's health.
  - `setHealth()` and `getHealth()`.
- `Stealth`, `SuperStrength` and `Healing` have them too:
  - `SuperStrength.attack()` returns "`next.attack() * 2`"
    - (Doubles the damage dealt to another character.)
  - `Stealth.defend(x)` calls "`next.defend(x / 2)`".
    - (Halves the damage received by this character.)
  - `Healing.attack(x)` doesn't alter the attack, but increases *this* character's health at the same time.

Decorator: Game Example



Decorator: Game Example

- Here's the common interface for both base implementations *and* decorators.
- (The Decorator Pattern only works because we can pretend that a decoration *is* a `GameCharacter`, in this case.)

```
public interface GameCharacter
{
    int attack();
    void defend(int damage);
    void setHealth(int health);
    int getHealth();
}
```

Decorator: Game Example

```
public class ElfCharacter implements GameCharacter
{   // One base implementation
    private int health = 100;
    public ElfCharacter() {}

    @Override public int attack() { return 10; }
    @Override public void defend(int damage)
    {
        health -= damage;
        if(health <= 0) {...} // Bring out yer dead.
    }
    @Override public void setHealth(int health) {...}
    @Override public int getHealth() {...}
}
```

- Each base implementation would do this a bit differently.
- We could also have a common superclass *just* for the base implementations.

Decorator: Game Example

```
public abstract class Ability implements GameCharacter
{   // Decoration superclass
    protected GameCharacter next;
    public Ability(GameCharacter next) { this.next = next; }

    @Override public int attack() { return next.attack(); }

    @Override public void defend(int damage) {
        next.defend(damage);
    }
    @Override public void setHealth(int health) {
        next.setHealth(health);
    }
    @Override public int getHealth() {
        return next.getHealth();
    }
}   // Every method here just passes on to "next".
```

Decorator: Game Example

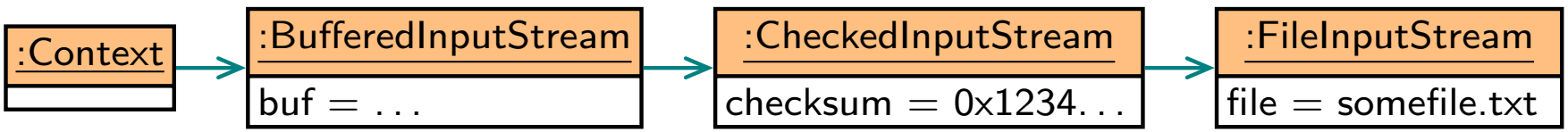
- The decoration superclass delegates everything.
- The decoration *subclasses* can then re-override certain methods and change the behaviour.

```
public class Stealth extends Ability
{   // Decoration
    public Stealth(GameCharacter next)
    {
        super(next);
    }

    @Override
    public void defend(int damage) // When defending, reduce
    {                              // damage by half.
        next.defend(damage / 2);
    }
}
```

Decorator: Java IO

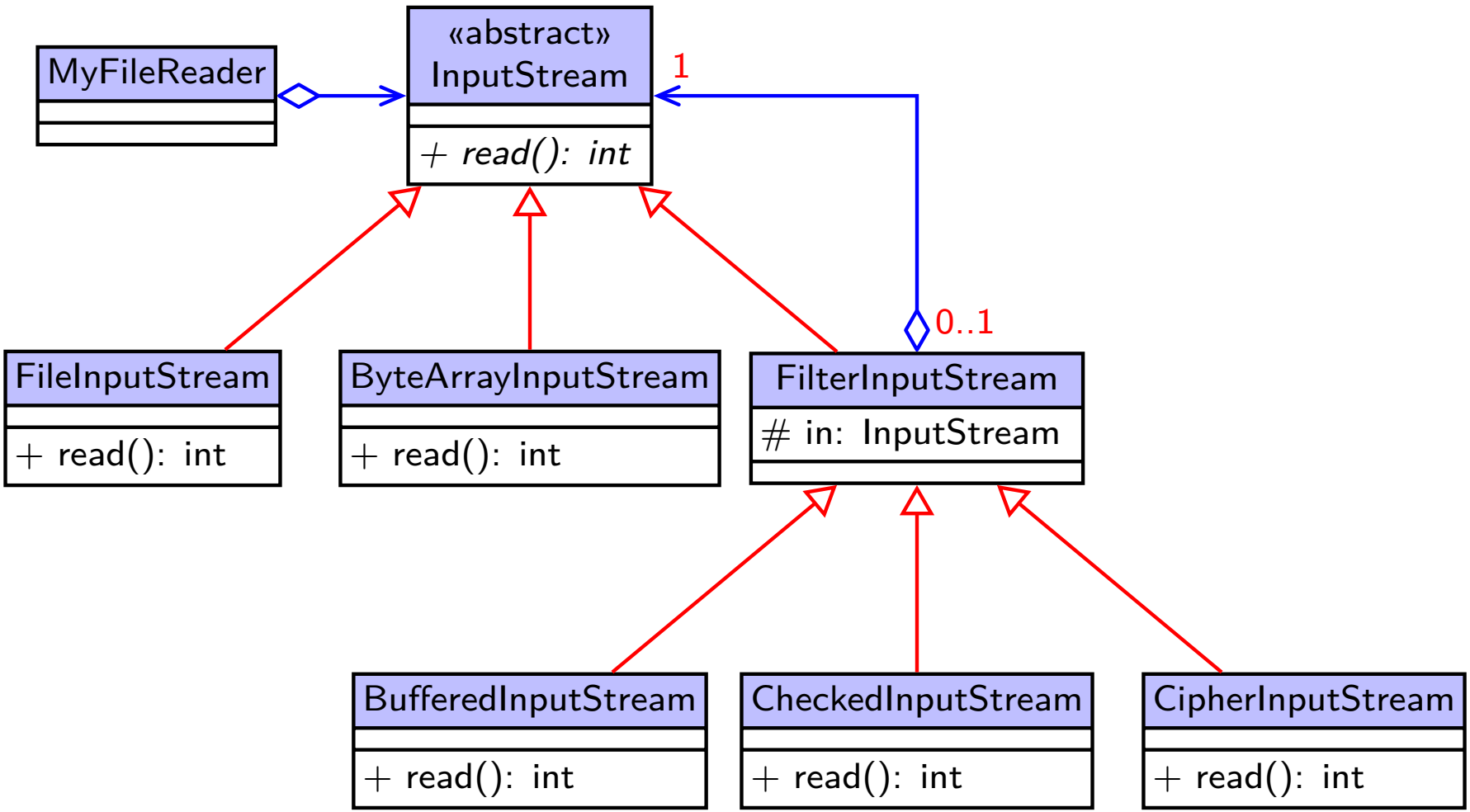
- Java's IO streams are based on the decorator pattern.



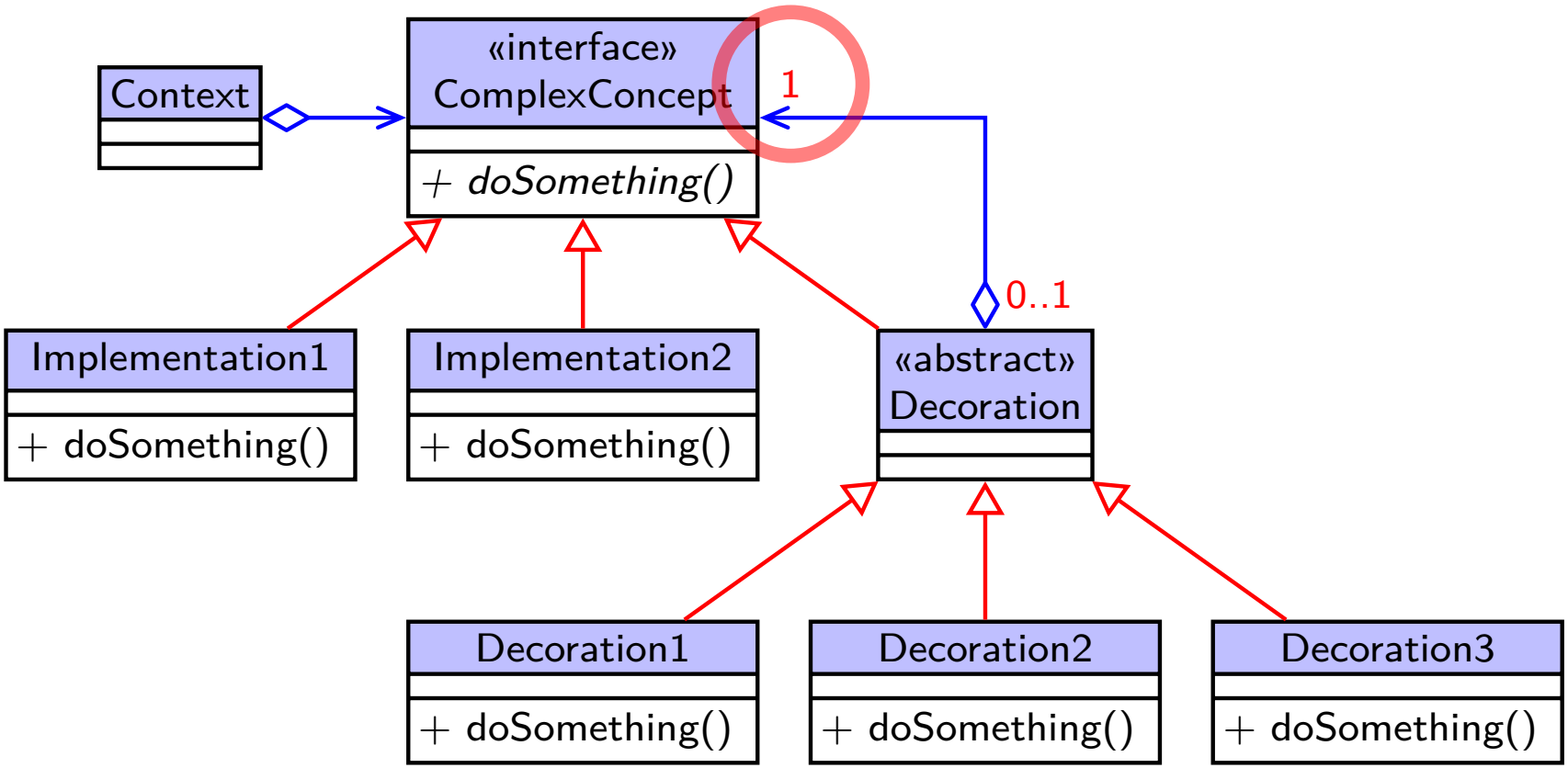
- Here, an instance of FileInputStream is the data source.
- The other objects are *decorating* it, adding functionality.
- Each of their classes implement the `InputStream` interface.
- So, they can all be mixed and matched:
  - FileInputStream could be replaced with ByteArrayInputStream.
  - The others could be replaced, or re-ordered, or added to.

Decorator: Java IO

- Some of the Java IO classes:



Decorator: General Class Structure



- A decoration owns *one* more element – either:
  - Another decoration, or
  - A base implementation (leaf).

The Composite Pattern

- *Also* uses recursive aggregation.
- Creates a complex hierarchy of objects; i.e. a tree.
  - *Not* for writing efficient algorithms.
  - For representing complex concepts.
- Used in various situations; e.g.:
  - Reading/processing HTML and XML documents.
  - Creating a graphical user interface.
  - Parsing source code (e.g. compilers and related tools).
- Sometimes a fixed set of aggregation relationships isn't flexible enough.
- (This is not related to the UML "composition" notation.)

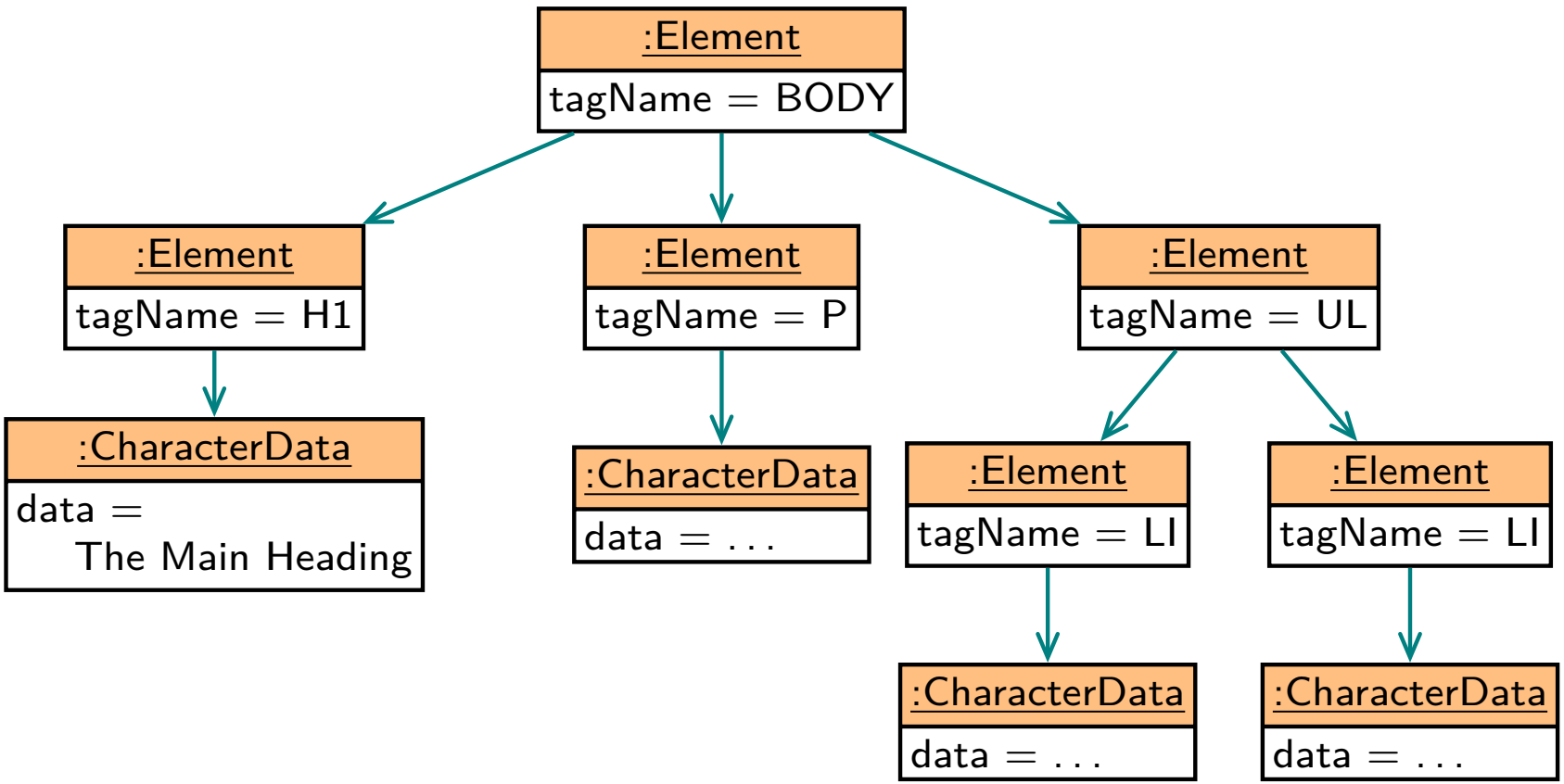
Composite Example: HTML "Document Object Model"

- Ever looked at HTML? It's hierarchical!
- HTML tags ("elements") divide up the text ("character data") into hierarchical sections. e.g.

```
<html>
    <head>
        <title>The Web Page</title>
    </head>
    <body>
        <h1>The Main Heading</h1>
        <p>A paragraph with <em>emphasis</em>.</p>
        <ul>
            <li>A list item.</li>
            <li>Another list item.</li>
        </ul>
    </body>
</html>
```

HTML as Objects

- When parsed, HTML is represented as a hierarchy of objects.
  - (This is called the Document Object Model (DOM).)
- Notice that `Element` objects contain both `Element` and `CharacterData` objects.

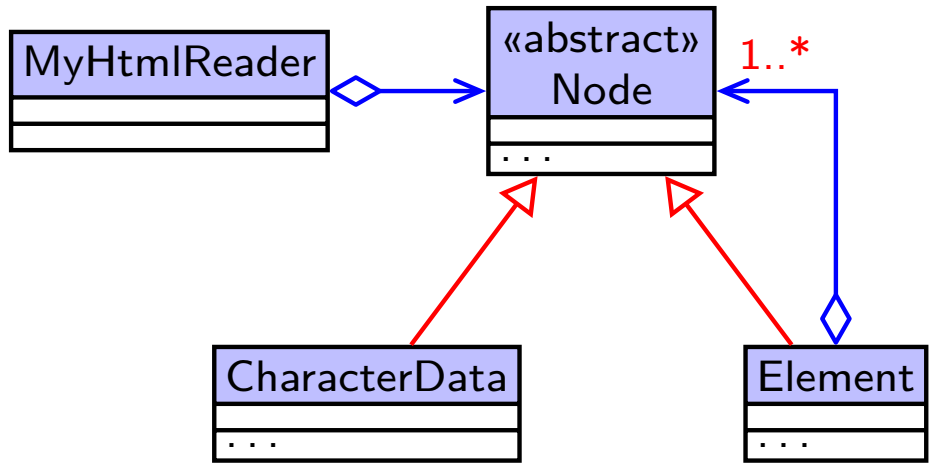


Composite Representation

- The Composite Pattern represents a tree of objects.
- An *arbitrary* tree – any size and shape.
  - An HTML document could be empty.
  - Or it could have 1,000,000 tags inside a tree structure a 1,000 levels deep.
  - We want the *same class structure* to handle all cases!
- Define two concrete classes:
  - A "leaf-node" class (call it something relevant).
  - A "composite-node" class.
- *And* a common interface (or abstract superclass).
  - This allows us to treat the leaf and composite nodes in the same way.
- The composite-node class aggregates its own superclass.
  - This allows composite nodes to have child nodes.
  - Child nodes themselves can be *either* type.

HTML as Classes

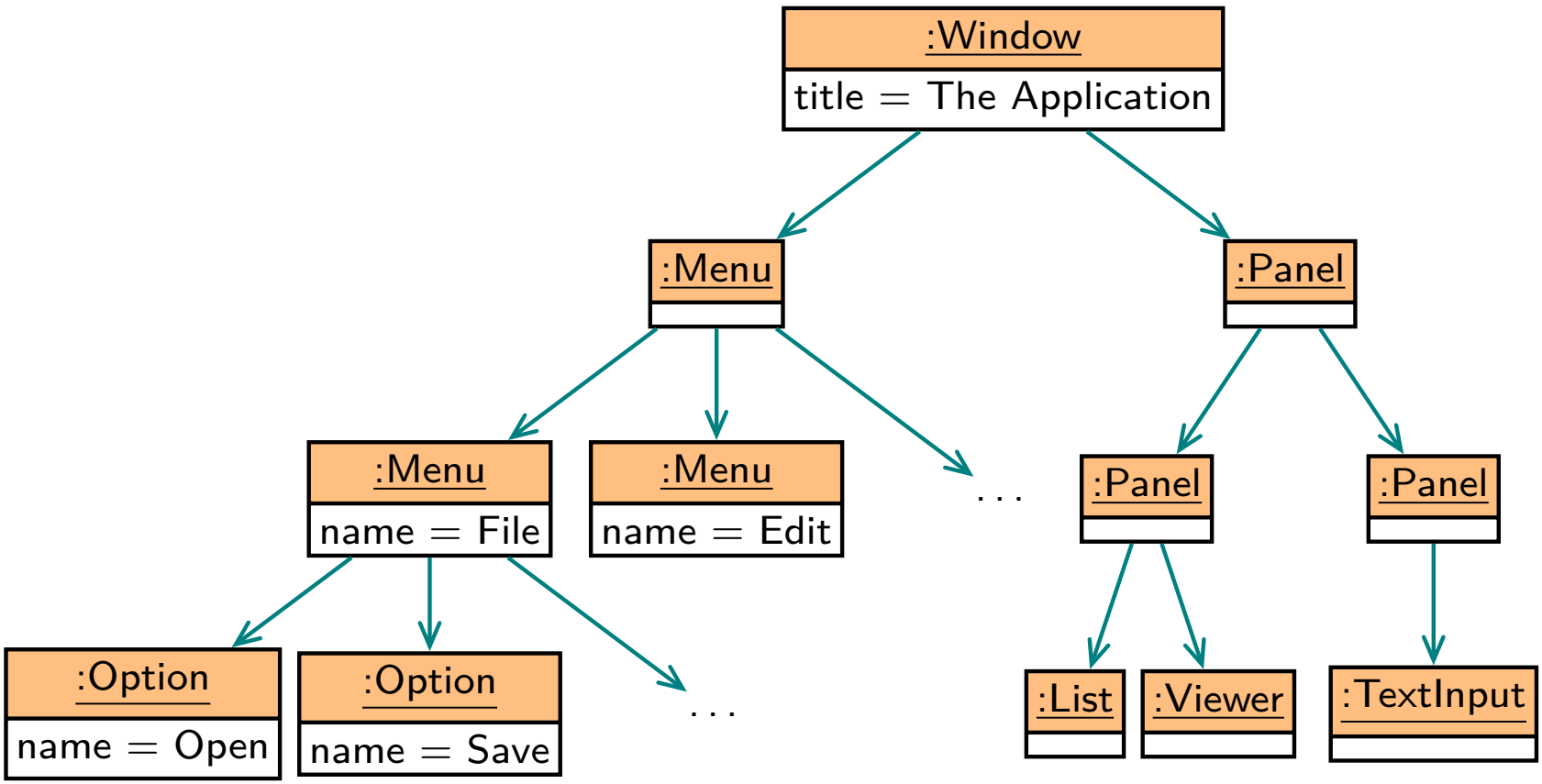
- In our HTML example:



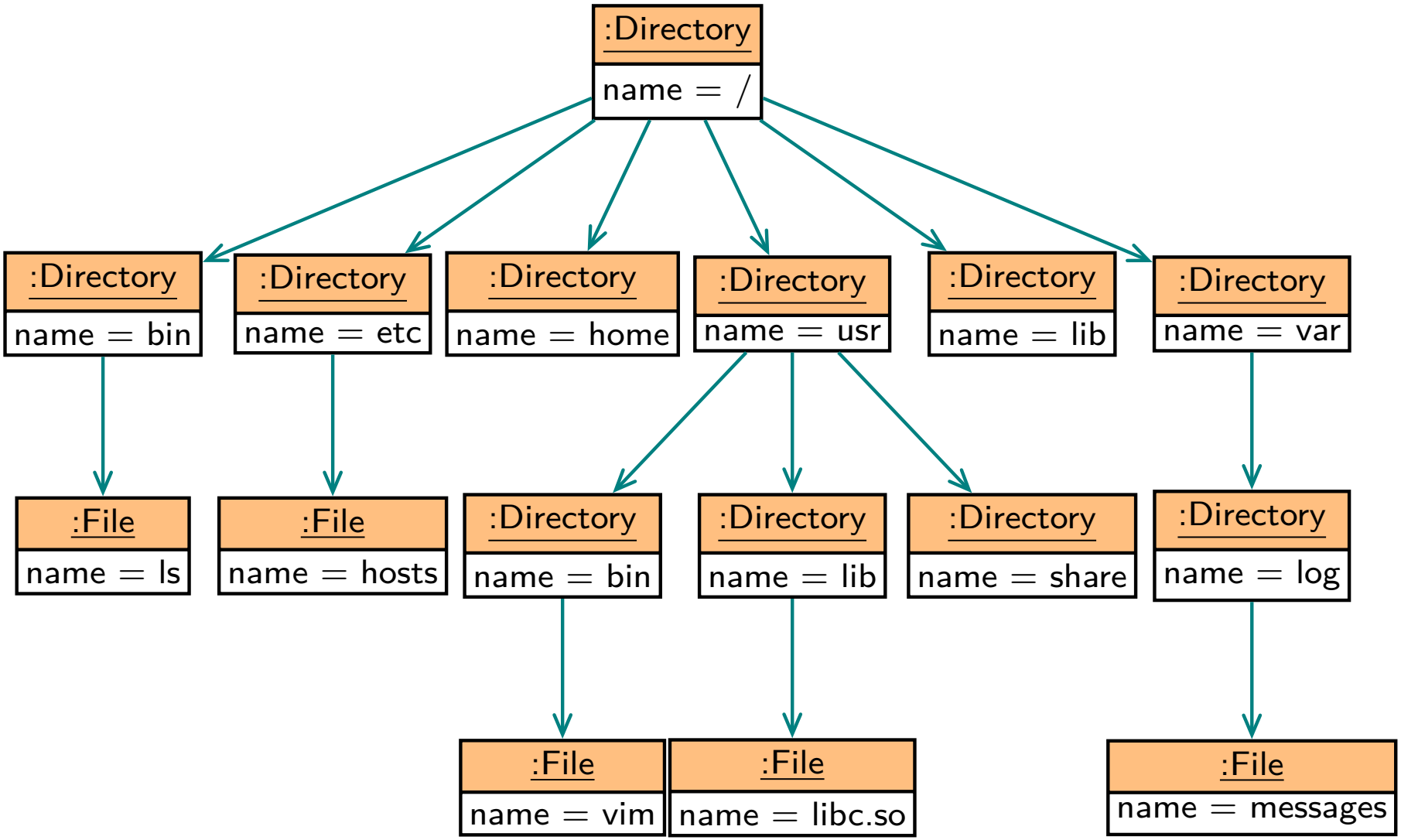
- Our concrete classes:
  - `CharacterData` represents a segment of actual text.
  - `Element` represents a pair of tags (e.g. ...).
- And the relationships?
  - Each `Element` can contain both text and other `Elements`.
  - Therefore, `Element` aggregates `Node`, which can be either.
  - 1..* multiplicity, since elements can own *many* nodes.

Composite Example 2: Graphical User Interface

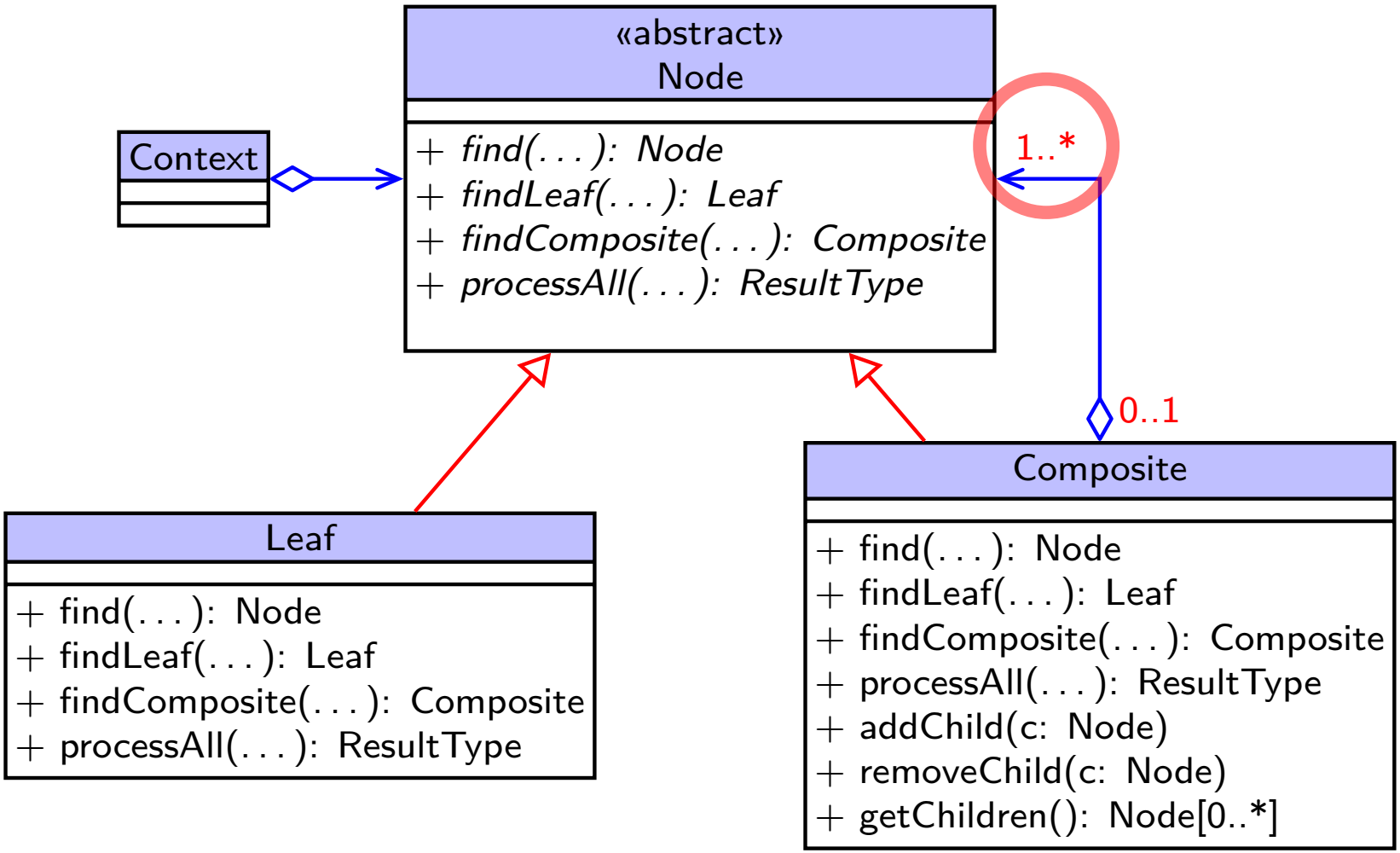
- GUIs are also broken up hierarchically.



Composite Example 3: Directory Hierarchies



The Composite Pattern: General Class Structure

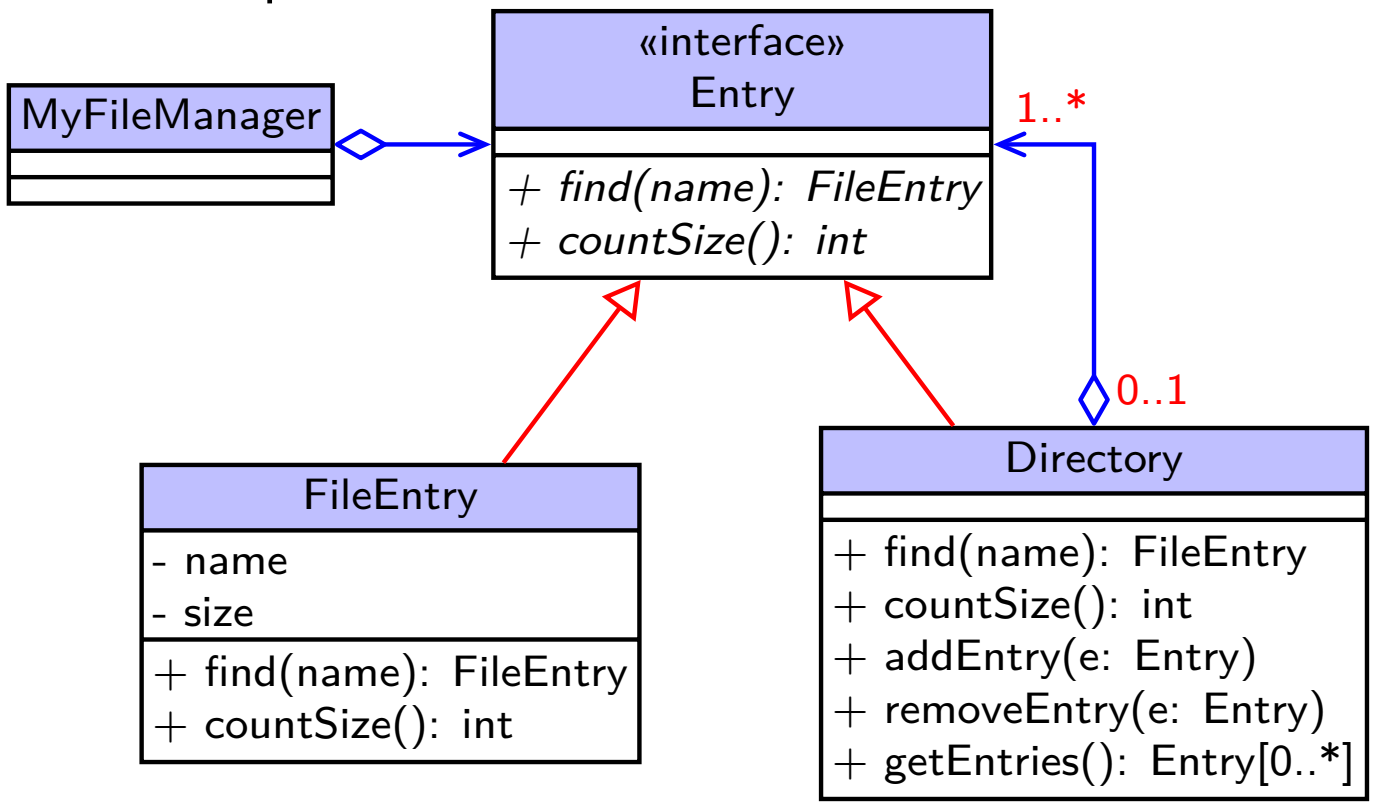


Composite Operations

- Composite structures are designed to perform operations on themselves; e.g.:
  - Find and return a particular node.
  - Calculate a result based on all the nodes.
- These happen recursively, in a node-by-node fashion.
  - Each node processes itself.
  - Leaf nodes typically just return a simple value.
  - Composite nodes typically recurse down to their child nodes.
- Composite nodes also need accessors/mutators for their children:
  - `getChildren()`, `addChild()`, `removeChild()`, etc.
  - When you need to modify the tree:
    - You execute a "find" operation to retrieve one of the composite nodes.
    - (*Or* maybe you also have your composite nodes in a Map.)
    - You call `addChild()`, etc. on the composite node.

Composite Operations: Example

- We can represent files and directories like this:



- We have two operations:
  - Finding a file within the directory structure.
  - Counting the total number of bytes used.
- These are implemented by both `FileEntry` and `Directory`.

Composite Operations: Example

- No surprises in the interface/superclass itself.

```
public interface Entry
{
    FileEntry find(String name);
    int countSize();
}
```

- However, these will be called from *two places*:
  - The context (`MyFileManager`) in this case.
  - The composite class (`Directory`).

Composite Operations: Example

```
public class FileEntry implements Entry
{
    private String name;
    private int size;
    ...
    @Override public FileEntry find(String name)
    {
        FileEntry found = null;
        if(this.name.equals(name)) { found = this; }
        return found; // Returns itself if the name matches.
    }                 // Returns null if not.

    @Override public int countSize()
    {
        return size;
    }
}
```

Composite Operations: Example

```
public class Directory implements Entry
{
    private List<Entry> children;
    ...
    @Override
    public FileEntry find(String name)
    {
        for(Entry child : children)
        {
            FileEntry found = child.find(name); // Recurse!
            if(found != null)
            {
                return found;
            }
        }
        return null;    // Multiple returns! Bad Dave...?
    } // continued...
```

Composite Operations: Example

```
    // continued...
    @Override
    public int countSize()
    {
        int size = 0;
        for(Entry child : children)
        {
            size += child.countSize(); // Recurse!
        }
        return size;
    }
}
```

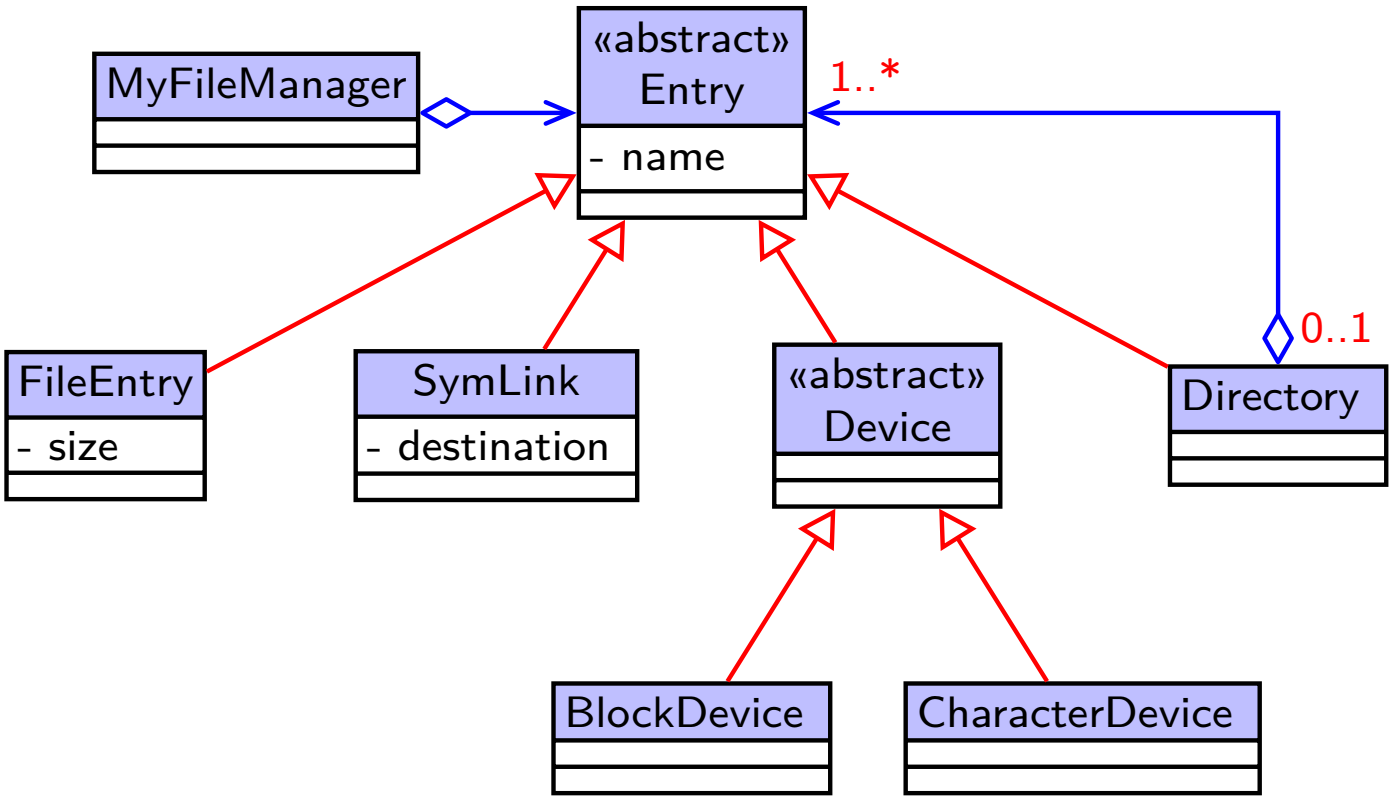
Composite Operations: Example

- Our `find()` implementation may be a bit naïve.
  - Notice the code above only finds *files*.
  - Notice it only finds *one* file.
  - What if we want to find *entries* – either a file or directory?
  - What if we want to find *all* matching entries at once?
  - I'll leave this consideration for the practical worksheet...

Composite Variations

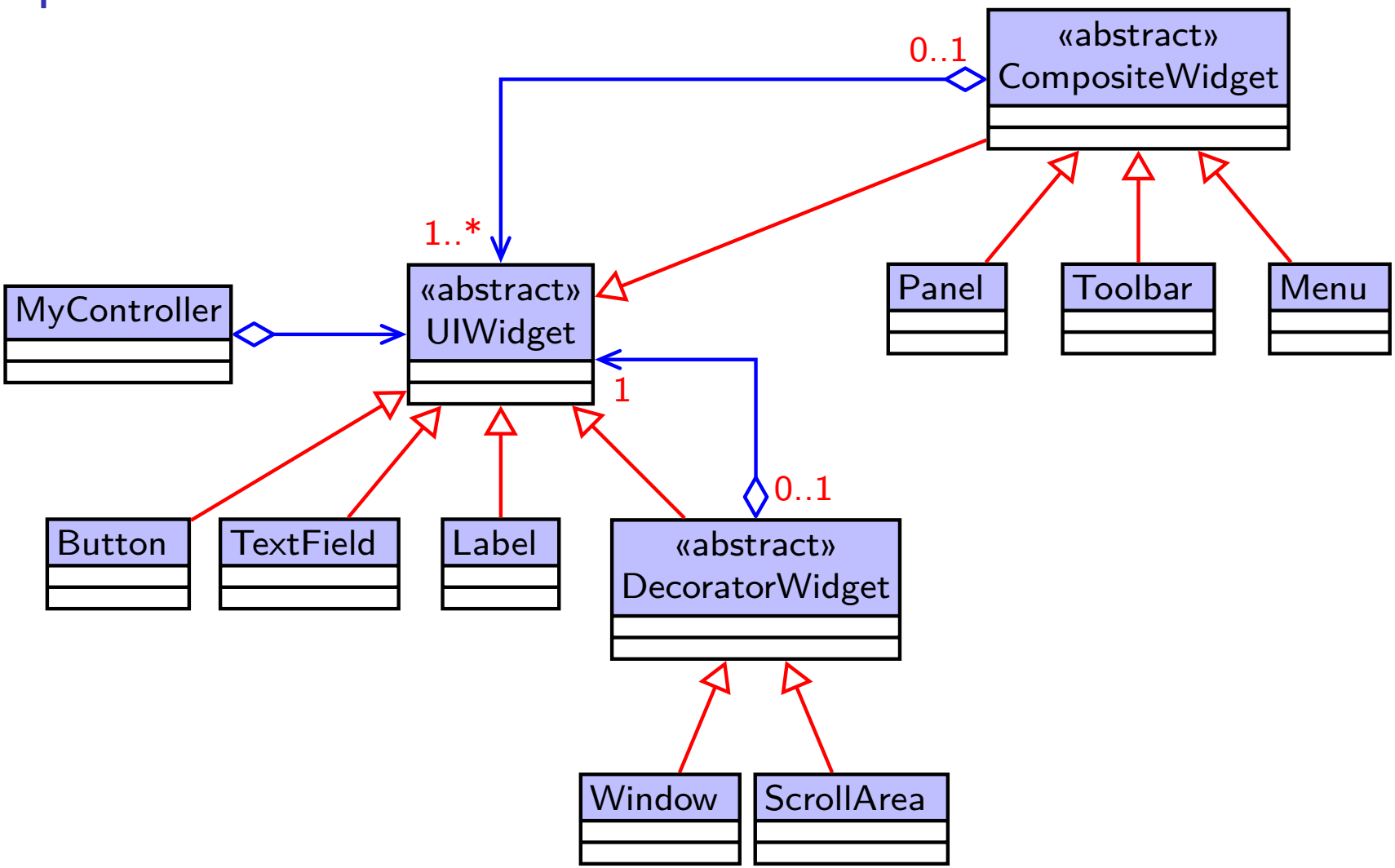
- So far we've only described the simplest version of the Composite Pattern.
- There are many possible variations.
- You could have multiple leaf node classes, and/or multiple composite node classes.
- Leaf/composite nodes could have a whole other inheritance hierarchy beneath them.
- You can have *decorator* nodes too!
  - An extra composite-like node that only has one child.
- The superclass could contain a template method.
  - With abstract hook methods.
  - The hook method implementation in the composite class would (probably) recursively call the template method on its child nodes.

Composite Variations



- Here, leaf node types include `FileEntry`, `SymLink`, and `Device` and its subclasses.

Composite Variations



- Not any particular GUI framework, but is based on common GUI concepts.

Decorator, Composite and MVC

- Decorator and Composite combine representation with operations.
  - Really, this is just an expression of OO philosophy. *But...*
  - Do they belong in the model?
    - Because they represent things.
  - Or in the controller?
    - Because they perform operations.
  - Or the view?
    - GUIs typically use the Composite Pattern.
- There are no rules, only better or worse choices for the situation at hand.
  - i.e. it depends what the decorator/composite is *for!*
- However, given the choice between the following...
  1. Model classes performing (simple) operations on themselves.
  2. Controller classes acting as a kind of model.

  ...The first one makes a lot more sense!