Design Patterns

Software Architecture, Process, and Management

This lecture will introduce the notion of a *Design Pattern* and relate how they are used in the construction of large scale software.

First-Class Functions

- Allow us to parameterise not only with data values but with computation/evaluation as well

```
def map (f, xs):
  result_list = new List()
  for x in xs:
    result_list.add(f(x))
  return result_list
```

Higher-Order Functions

- Doing so additionally allows us to make use of “curried” function definitions to allow “partial instantiation”

```
def add x y:
    return x + y
increment = add 1
```

- Now we have a function for incrementing, not a huge win in definition but when used inline can be convenient/clear/concise

```
map (add 1) xs
```

Object-Oriented Code

- Objects similarly arose from patterns that programmers were already using in existing code
- Allowing encapsulation as well as parameterising code with other code

```
void perform_many(PerformerList performers){
    foreach (Performer performer : performers){
        performer.perform()
    }
}
```

Object Polymorphism

- This is allowing us to treat objects in the same way if the objects are of a similar enough kind
- Even though those objects are not of the exact same kind, they share enough similarity to be treated the same in some circumstances
- This is allowing yet more parameterising of our code and avoiding yet more duplication
- Even if that duplication is of a more abstract duplication of structure rather than exact lines

Modules and “Functors”

```
module Dictionary
    type dict k v = [(k,v)]

    empty_dict : dict
    empty_dict = []

    add : k -> v -> dict -> dict
    add k v d = (k,v) :: d

    lookup : k -> dict -> v
    lookup k [] = raise Error
    lookup k ((x,v)::rest) = if k == x then v else lookup k rest
end
```

Modules and "Functors"

```
module type DICTIONARY
    type dict
    val empty_dict : dict
    val add k v : k -> v -> dict -> dict
    val lookup : k -> dict -> v
end
module Dictionary : DICTIONARY
    type dict k v = [(k,v)]
    empty_dict : dict
    empty_dict = []

    add : k -> v -> dict -> dict
    add k v d = (k,v) :: d

    lookup : k -> dict -> v
    lookup k [] = raise Error
    lookup k ((x,v)::rest) = if k == x then v else lookup k rest
end
```

Modules and “Functors”

```
module WordCounter(DICTIONARY D)
    word_count : [ word ] -> D.dict word int
    word_count [] = D.empty_dict
    word_count (w :: rest) =
    let d = word_count rest in
    add w (lookup w d) d
end
```

- “Functor” is a terrible word for this
- This has not been a hugely popular module system
- It is not clear why
- Part of the reason might be that figuring out ahead of time which parts of a module to parameterise is difficult

Design Patterns

- A design pattern is a standardised solution to a problem commonly encountered during object-oriented software development
- A pattern is *not* a piece of *reusable code*, but an overall approach that has proven to be useful in several different systems already
- You will not find an executable library of design patterns
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.

Contents of a Design Pattern

- Design patterns usually include:
  1. A pattern name
  2. A statement of the problem solved by the pattern and/or the situations in which it is appropriate
  3. A description of the solution
  4. A list of advantages
  5. A list of liabilities

Template Method:

Problem Solution Advantages Liabilities

- You have two or more methods which have a similar structure or spine, but differ in some small details
- Common examples:
  - Formatting output, as above, is a common example
  - Operations over complex data structures, such as matrices sometimes have similar opportunities to apply Template Method

Template Method:

Problem Solution Advantages Liabilities

- As above:
  1. The template method: which has the common structure of the similar methods
  2. The interface: which specifies the types of the delegated behaviour
  3. Instantiations: each original similar method implements the interface defining the behaviour specific to their case

Template Method:

Problem Solution Advantages Liabilities

- Alternatively:
  1. The common behaviour can be implemented as an abstract base class
  2. The interface is specified as abstract methods in the abstract base class
  3. The instantiations are concrete classes which derive from the ABC
- This has the advantage that “hook” operations can be defined
  - lsquo;Hook” operations are those that **may** be overridden, but need not be

Template Method:

Problem Solution Advantages Liabilities

- A fundamental technique for code reuse and removing unnecessary duplication

Template Method:

Problem Solution Advantages Liabilities

- It is easy to apply this pattern prematurely, expecting many similar operations and unnecessarily factoring out a lot of plausibly adaptable behaviour
  - This can lead to obscuring the original template method
- It can make the code less adaptable because changes in the template method are enforced upon all consumers
- Some complain this leads to inverted logic, sometimes referred to as *"Hollywood Logic"* because it encompases: *"Don't call us, we'll call you"*

Composite:

Problem Solution Advantages Liabilities

- There is a situation in which an object can be a primitive or a container of such objects which may themselves be primitives or containers
- Two obvious examples:
  - Graphics libraries in which a widget may be a primitive such as a text label, or a container such as a dialog which contains buttons each of which contains a primitive text label
  - Abstract syntax trees in which an expression may be a primitive such as a variable or integer literal, or an application expression consisting of smaller expressions
- The user wishes to ignore the differences
- Surrounding code would get complex if it were always conditional on whether an object was a group or a primitive

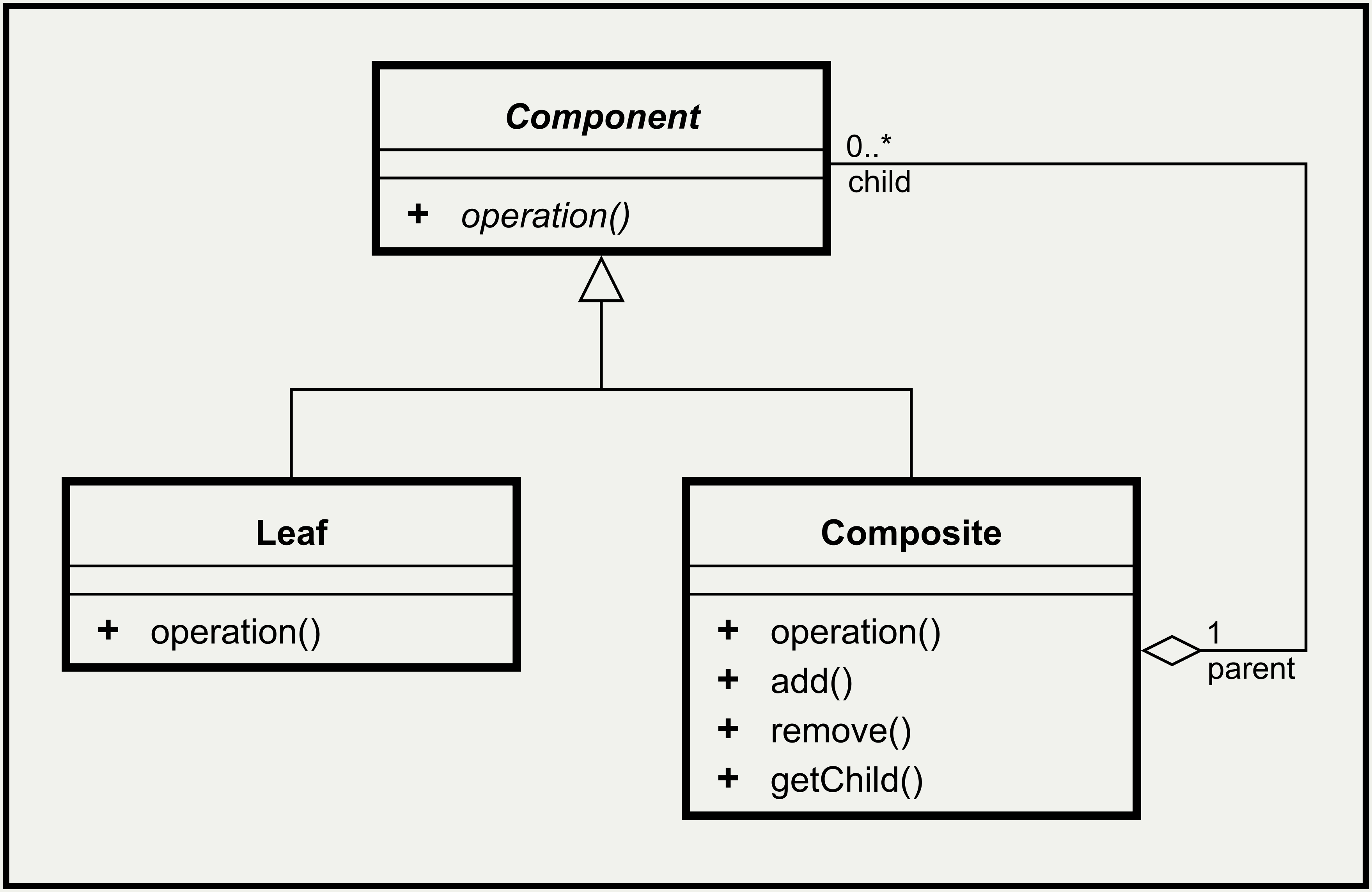
Composite:

Problem Solution Advantages Liabilities

- Three classes:
  1. Component: The shared interface between both primitives and containers
  2. Leaf: A primitive, implemented directly
     - The text label is actually drawn to the screen
     - A variable expression returns its name as the list of names mentioned in the expression, an integer literal the empty list
  3. Composite/container: Implements the shared interface generally by calling the same method on its children objects

Composite:

Problem Solution Advantages Liabilities



Composite:

Problem Solution Advantages Liabilities

- Simple support for arbitrarily complex hierarchies
- Clients can be simple as they do not need to know about composition
- New Composite and Leaf classes can be introduced without changing Component

Composite:

Problem Solution Advantages Liabilities

- Hard for client to predict/restrict what components might be encountered
- Hard to test that client works for all components
- Often need to define operations on Components that make sense only for some Component types, e.g. Composites

Visitor:

Problem Solution Advantages Liabilities

- Elements of an object structure respond to some interrogation differently: for example consider an object hierarchy representing a formatted document
- You have multiple operations to perform, suppose you wish to search the document, spell-check the document, count the words, extract the citations etc.
- The problem here is that these operations will be spread across the different node-types in the document structure, leading to a system that is difficult to maintain/change
- It could be an improvement to have the operations separate from the node-classes

Visitor:

Problem Solution Advantages Liabilities

- We can define an interface for a visitor class which defines a method to visit each kind of node in the document object hierarchy
- Each operation implements the visitor interface
- To visit a node of unknown kind, the visitor simply calls the `accept` method on the node
- The node implements the `accept` method by calling the visitor's specific method for its own node kind

Visitor:

Problem Solution Advantages Liabilities

1. Adding new operations is easier, since it requires that you instantiate the *Visitor* interface
2. Thus the related operations are not spread out over the node-classes but instead grouped together in one *Visitor* class
3. The nodes do not even need to have a common base class (other than that they implement *accept*
4. If the node hierarchy is exported as a library, users who are defining their own document operations do not need to modify the nodes in question

Visitor:

Problem Solution Advantages Liabilities

1. Can break encapsulation:
   - Since the visitors must rely on the exported interface of the concrete element nodes, that interface must give enough to allow the visitors to do their job

Design Pattern Examples

- Creational Patterns:
  - E.g. Abstract Factory, Factory Method
- Structural Patterns:
  - Composite
  - Proxy
- Behavioral Patterns:
  - E.g. Command, Visitor
- These are from Gamma et al. (1995) (a.k.a. the Gang of Four book), but there are many other pattern collections

Patterns Compiler

- Why do we not write a "Patterns Compiler"?
- This question is sometimes phrased:
  - "Are patterns missing language features?" or
  - "Are patterns a language smell?"
- Patterns generally imply some kind of duplication
- Which is almost explicitly a violation of the *DRY* principle

Design Patterns vs Language Constructs

A small list of *some* of the design patterns which have recognised language features which largely obviates the need for the pattern:

Pattern	Language Feature
SingletonPattern	MetaClasses
VisitorPattern	GenericFunctions or MultipleDispatch
FactoryPattern	MetaClasses, closures
IteratorPattern	Anonymous/higher order functions
InterpreterPattern	Macros (extending the language)

Design Patterns vs Language Constructs

A small list of *some* of the design patterns which have recognised language features which largely obviates the need for the pattern:

Pattern	Language Feature
RunAndReturnSuccessor	TailCallOptimization
HandleBodyPattern	Delegation, Macros, MetaClasses
Abstract-Factory, Flyweight, Factory-Method, State, Proxy, Chain-of-Responsibility	FirstClass types (Norvig)
FacadePattern	Modules (Norvig)

Yes, Patterns are a language smell

- Patterns are a sign that the language is missing some feature
- But, so what? If we do not know how to solve that particular problem, then we will not have a language feature any time soon
- Or we might not know how that particular feature can be integrated into the language in question
- In the meantime, a pattern is the best we can do
- The fact that we have now moved on to larger "design patterns" is evidence for the fact that code re-use in the small is mostly a solved problem
- It is also evidence that code re-use in the large is still mostly an unsolved problem