Compositeパターンについて

・目的

Compositeパターンを使い目的は単一のオブジェクトとその集合のどちらも同じように扱えるようにするためのです。つまり、「単一のオブジェクト」と「オブジェクトの集合」を混ぜて、アクセス方法を同じにしてしまうパターンです。

・効果

このパターンを用いると、ディレクトリのような、ツリー構造を伴う再帰的なデータ構造を容易に表現することができます。Compositeパターンにおいて登場するオブジェクトは、「枝」と「葉」であり、これらは共通のインターフェイスを実装します。したがって、枝と葉を同様に扱えるというメリットがある。つまりこのパターンは、プリミティブなオブジェクトであっても、複合的に構成されたコンテナオブジェクトであっても、一様に扱うことができます。

・背景

コンピュータのプログラミングで、素人と達人の間ではびっくりするほどの生産性の差があるが、その差はかなりの部分が経験の違いからきている。達人は、さまざまな難局を、何度も何度も耐え忍んで乗り切ってきている。そのような達人たちが同じ問題に取り組んだ場合、典型的にはみな同じパターンの解決策にたどり着くのだが、これがデザインパターンである (GoF)。

それぞれのパターンは、プログラマの間で何度も繰り返し考え出されてきた。したがって、それは最善の解決策ではないかもしれないが、その種の問題に対するトレードオフも考慮した、典型的な解決策ではある。更に、コストがかかるかもしれない問題解決を実際に行う前の先行調査として大変役に立つ。パターンには名前がついていることが重要である。なぜなら、名前がついていることで問題や解決策を記述したり、会話の中で取り上げたりすることができるようになるからである。

・Compositeパターンの実装のコードという考え方

-サンプルコード

class MakeBatterTask < Task

def initialize

super('Make batter')

@sub\_tasks = []

add\_sub\_task(AddMilkTask.new)

add\_sub\_task(MixTask.new)

# add\_sub\_task(AddFreshCreamTask.new)

# etc ..

end

def add\_sub\_task(task)

@sub\_tasks << task

end

def remove\_sub\_task(task)

@sub\_tasks.delete(task)

end

def get\_time\_required

time=0.0

@sub\_tasks.each { |task| time += task.get\_time\_required }

time

end

end

・まとめ

一見複雑なオブジェクトたちを再帰的に整理し、変化に強い構造へ。

ベーシックな考え方であり、他のパターンに紛れて登場することも多い。

・注意

Compositeパターンを用いる際には、データ構造がきちんと木構造を保つようにしなければならない。親子関係が循環してしまった場合、無限ループに陥るからです。

・総括

　コンポジットパターンは、ディレクトリ構造のような再帰的な構造を解決することに適しています。

　再帰的とは、ディレクトリの下にさらにディレクトリを置くことができるように、自分自身と同じタイプの要素を含むことができる性質のことです。 このような構造では、要素の下にさらに要素を含むことができる複合要素と、末端となる単独要素の2つのタイプの要素から成り立っています。 ディレクトリ構造で言うと、ディレクトリが複合要素、ファイルが単独要素に当たります。 この複合要素と単独要素を組み合わせていくことで再帰的な構造が形成されていきます。

　再帰的な構造全体に対する処理を行いたい場合に、複合要素と単独要素を区別して実装してしまうと、処理の実装は非常に煩雑になってしまいますし、新しい要素を加えたいなどの構造に対する変更もコストが大きくなってしまいます。

　これに対しコンポジットパターンでは、すべての要素に共通の処理（オペレーション）を持つ抽象クラス（コンポーネント）を定義し、サブクラスとして複合要素（コンポジット）と単独要素（リーフ）を別々に用意しておきます。

　コンポジットクラスのオペレーションではリーフクラスのオペレーションをすべて代表して実行するように実装しておくことで、複数の要素の集まりを、まるで1つの要素であるかのように扱えます