WIKIPEDIA

Abstract Document Pattern

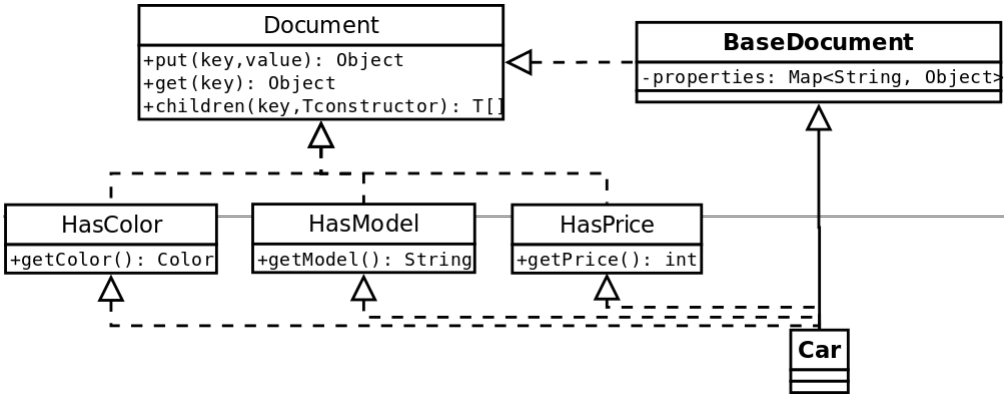
An object-oriented structural design pattern for organizing objects in loosely typed key-value stores and exposing the data using typed views. The purpose of the pattern is to achieve a high degree of flexibility between components in a strongly typed language where new properties can be added to the object-tree on the fly, without losing the support of type-safety. The pattern makes use of traits to separate different properties of a class into different interfaces.[1] The term "document" is inspired from document-oriented databases.

Contents

Definition

A document is an object that contains a number of properties. A property can for an example be a value like a number or a string, or it can be a list of other documents. Every property is referenced using a key.[2] When traversing the document tree, the user specifies a constructor to be used for creating the implementation class of the next level. The implementations are often a union of various traits that extend the Document interface, making it possible for them to handle setting and getting properties on their own.

Structure



The interface "Document" states that properties can be edited using the "put" method, read using the "get" method and sub-documents can be traversed using the "children" method. The "children" method requires a functional reference to a method that can produce a typed view of a child given a map of the data the child should have. The map should be a pointer to the original map so that changes in the view also affect the original document.

Implementations can inherit from multiple trait interfaces that describe different properties. Multiple implementations can even share the same map, the only restriction the pattern puts on the design of the implementation is that it must be stateless except for the properties inherited from "BaseDocument".

Pseudocode

```
interface Document
  put(key : String, value : Object) : Object
  get(key : String) : Object
  children(key : String, constructor : Map<String, Object> -> T) : T[]

abstract class BaseDocument : Document
  properties : Map<String, Object>

  constructor(properties : Map<String, Object>)
      this->properties := properties

  implement put(key : String, value : Object) : Object
      return this->properties->put(key, value)

  implement get(key : String) : Object
      return this->properties->get(key)

  implement children(key : String, constructor : Map<String, Object> -> T) : T[]
      var result := new T[]

      var children := this->properties->get(key) castTo Map<String, Object>[]
      foreach ( child in children )
          result[] := constructor->apply(child)

      return result
```

Usage

The Abstract Document Pattern allows the developer to store variables like configuration settings in an untyped tree structure and operate on the documents using typed views. New views or alternative implementations of views can be created without affecting the internal document structure. The advantage of this is a more loosely coupled system, but it also increases the risk of casting errors as the inherit type of a property is not always certain.

Example implementation

Document.java

```
public interface Document {
    Object put(String key, Object value);
    Object get(String key);
    <T> Stream<T> children(
            String key,
            Function<Map<String, Object>, T> constructor
    );
}
```

BaseDocument.java

```
public abstract class BaseDocument implements Document {
    private final Map<String, Object> entries;
    protected BaseDocument(Map<String, Object> entries) {
        this.entries = requireNonNull(entries);
    }
    @Override
    public final Object put(String key, Object value) {
        return entries.put(key, value);
    }
    @Override
    public final Object get(String key) {
        return entries.get(key);
    }
```

```
    @Override
    public final <T> Stream<T> children(
            String key,
            Function<Map<String, Object>, T> constructor) {
        final List<Map<String, Object>> children =
            (List<Map<String, Object>>) get(key);
        return children == null
                    ? Stream.empty()
                    : children.stream().map(constructor);
    }
}
```

Usage.java

```
Map<String, Object> source = ...;
Car car = new Car(source);
String model = car.getModel();
int price = car.getPrice();
List<Wheel> wheels = car.children("wheel", Wheel::new)
    .collect(Collections.toList());
```

References

1. Forslund, Emil (2016-01-15). "Age of Java: The Best of Both Worlds" (https://ageofjava.blogspot.com/2016/01/the-best-of-both-worlds.html). *Ageofjava.blogspot.com*. Retrieved 2016-01-23.
2. Fowler, Martin. "Dealing with Properties" (http://martinfowler.com/apsupp/properties.pdf) (PDF). Retrieved 2016-01-29.