Ownership in Design Patterns

Master Thesis
Half-Time Presentation
Stefan Nägeli
21.12.05
Overview

- Idea and Motivation
- Explanation of Different Ownership Concepts
- Status Quo
- Creational: Abstract Factory
- Structural: Composite
- Behavioral: Visitor
- Remaining Work
Idea and Motivation

- Question of practical usage of ownership concepts
- Example typings of industrial applications have been done
- Patterns encapsulate core design ideas and are widely used
- Comparison between different ownership concepts
Ownership systems under review

- **The Universe type system**
  Müller

- **Ownership Types**
  Clarke, Potter, Noble

- **Ownership Domains**
  Aldrich, Chambers
The **Universe type system**

- Apply alias control to reference types by annotating type declarations with new keywords:
  - `rep` to point into the object's representation
  - `peer` to point to objects in the same context
  - `readonly` to point to objects in any context
Universe type system Example
The Ownership Types Concept

• Every class def. is parameterized with one or more owners, where the first param. identifies the owner of the object.

```java
class Stack<stackOwner, elementOwner> {...}
```

• Each object can access:
  - `this` and objects `this` owns
  - Its ancestors and objects they own (not trans.)
  - Globally accessible objects (in the root context)
Ownership Types Access Rights
Ownership Types Example
The Ownership Domains Concept

- Each object can declare several ownership contexts, so-called domains (public / private).
- Each object can declare links between locally declared contexts and foreign contexts.
- Each object can access:
  - other objects in the same domain (peer)
  - objects in declared domains (rep)
  - objects in a domain to which the owner domain is linked
  - objects in a public domain when access rights to the domain's declarator is granted
Currently writing the report and analyzing each pattern from the Design Patterns book in terms of

- Intent
- UML Diagram
- Ownership Discussion
- Universe type system
- Ownership Types
- Ownership Domains
- Conclusion
- Ease of Ownership Applicability
## Status Quo - Creational

<table>
<thead>
<tr>
<th></th>
<th>UTS</th>
<th>Ownership Types</th>
<th>Ownership Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Builder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factory Method</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Prototype</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Singleton</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

* ✗ = applying ownership enhances pattern and works fine
* ✗ = ownership typing leads to problems or has few benefits
* ✗ = ownership typing is not possible with the current system
### Status Quo - Structural

<table>
<thead>
<tr>
<th></th>
<th>UTS</th>
<th>Ownership Types</th>
<th>Ownership Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Bridge</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Composite</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Decorator</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Facade</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Flyweight</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Proxy</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

- ✗ = applying ownership enhances pattern and works fine
- ❌ = ownership typing leads to problems or has few benefits
- ❌ = ownership typing is not possible with the current system
## Status Quo - Behavioral

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UTS</td>
<td>Ownership Types</td>
<td>Ownership Domains</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **x** = applying ownership enhances pattern and works fine
- **x** = ownership typing leads to problems or has few benefits
- **x** = ownership typing is not possible with the current system
Creational Pattern Example

Abstract Factory
Abstract Factory - UML

**Abstract Factory**
- +createProductA()
- +createProductB()

**ConcreteFactory1**
- +createProductA()
- +createProductB()

**ConcreteFactory2**
- +createProductA()
- +createProductB()

**AbstractProductA**
- ProductA1
- ProductA2

**AbstractProductB**
- ProductB1
- ProductB2

**Client**
AF – Ownership Motivation

- It would be useful if clients could be the owners of the created products.
- Client could safely establish invariants over the product's state.
Abstract Factory - Problems

- The abstract factory is likely to be a singleton, serving multiple clients.
- Clients want to be the owner of the created products.

Product instances are not created in the correct ownership context.

*Delegation of object creation*
Abstract Factory – UTS Diagram

Garage
+createTire()

Car

Tire

Tire

Tire

Tire
AF – Domains Diagram

App

Garage
+createTire()

Car
+createDunlop()
+createMichelin()

Tire
tires

factories

productowners
Abstract Factory - Workarounds

• Introduce a call-back mechanism so that the concrete factory still determines the product's type, but the factory delegates the creation of the product back to the client.

• Use a static factory instead.

• Introduce ownership system extension.
public class FactoryA extends Factory {

    public pure readonly Product createProduct(readonly Owner o) {
        return o.createProductA();
    }
}

public abstract class Owner {

    public pure rep Product createProductA() {
        return new rep ProductA();
    }
    // ... other concrete products
}
Abstract Factory – Static Factory

```java
public class Factory {
    public static pure peer Product createProduct() {
        if(...) {
            return new peer ProductA();
        } else {
            return new peer ProductB();
        }
    }
}

public class Client {
    public void generateMyProduct() {
        rep Product = rep Factory.createProduct();
    }
}
```
Abstract Factory – New Modifiers

- Enable creation of objects in the caller's context.
- Introduce new keywords `callerrep`, `callerpeer` or pass the owner as parameter (like in Java generics).
Structural Patterns Example

Composite
**Composite - UML**

**Component**

- `+operation()`  
- `+add(in component : Component)`  
- `+remove(in component : Component)`  
- `+getChild() : Component`

**Leaf**

- `+operation()`

**Composite**

- `+operation()`  
- `+add(in component : Component)`  
- `+remove(in component : Component)`  
- `+getChild() : Component`

forall g in children {
g.operation();
}

**Client**
Composite - Ownership Motivation

- Flat ownership structure: the client is the owner of all elements in the composite structure.
  - The client can be sure that no external objects manipulate its data structure.

- Deep ownership structure: each parent is the owner of its children.
  - The composite may hold invariants over its children
  - The composite may cache properties of its children
Composite – Deep Ownership Problems

- Components cannot be shared.
- The parent becomes responsible for creating the children.
- The ownership structure reflects the data structure and cannot be changed after creation.
- Clients may not maintain direct refs. into the DS anymore (except readonly in case of the UTS)
Composite – Workarounds

- Ownership transfer is needed to support dynamic structures.
- New modifiers are needed to support delegation of object creation.
Behavioral Patterns Example

Visitor
Visitor – Ownership Motivation

- The visitor should be able to visit any data structure that eventually already makes use of ownership.
Visitor - Problems

- Depending on who is responsible for the data structure traversal, the visitor or the element calls `accept(Visitor)` on the next element. `accept(Visitor)` is a non-pure operation.

- Each element calls `visitConcreteElement(this)` on the supplied visitor instance, also a non-pure operation.

The visitor and the data element need full access rights to each other.
Visitor – Ownership Structure

Client

Element

Visitor

Element

Element

Element

accept

visit

query
Visitor - Problems

- If the data structure has a deep ownership structure, the visitor may not access the element with read/write access rights.
- Only the Ownership Domains concept allows the visitor to access encapsulated elements by declaring appropriate links.
- The Universe type system misses a feature to give certain objects or classes special access rights.
Visitor – Domains Structure
Remaining Work

- Review the remaining patterns:
  - Builder
  - State, Strategy
  - Command
  - Interpreter
  - Chain of Responsibility
  - Memento
- Investigate usage of patterns in conjunction with established frameworks, like Swing or AWT
- Review composite patterns in terms of ownership
Questions?

Comments?