Ownership in Design Patterns

Stefan Nägeli

Master Thesis

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

March 14, 2006

Supervised by:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group
inf Informatik Computer Science

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Ownership is the concept of structuring the object store into different contexts by enabling objects to be the owner of other objects. All objects with the same owner are said to be in the same context. Restricting aliasing of objects in other contexts enables local reasoning about code correctness and simplifies comprehension and maintenance of program code. So far, different ownership type systems have been proposed and proven to be sound. While many of them look very promising on small examples, the question of practical usage for large and complicated applications remains unanswered.

One approach to answer the question of practical usage of ownership type systems is to review the concept of ownership when applied to design patterns. Design patterns are of great importance and widely used in practice. They structure core ideas of an application's design and we therefore believe that revealing problems and showing the benefits of ownership in connection with design patterns is essential for an evaluation of practical usage.

After a short introduction to three major ownership type systems, this thesis identifies beneficial ownership structures for all design patterns, covered in *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95], and discusses feasibility under each reviewed ownership type system. After discussing all patterns separately, possible pattern combinations are reviewed in terms of ownership through a small Java GUI application and the Swing GUI toolkit.

We show that the concept of ownership succeeds to enhance many pattern implementations. However, the reviewed ownership type systems still lack the necessary flexibility to successfully tackle all posed design scenarios. As a result, the main problems are identified and listed.

Contents

1 Introduction

Object-oriented software construction is very powerful and popular due to its natural representation of data and operations. Since object-oriented software construction is so widely used, it is essential that its concepts are continuously enhanced with the goal to make writing reliable software as easy as possible.

Aliasing is known as the state where two or more object variables point to the same object. Aliasing is thus a direct consequence of the concept of object identity and enables the sharing of objects. In most scenarios, aliasing is intended in order not to copy object structures when passed from one object to another. Unfortunately, aliasing can also occur unintentionally, leading to incorrectness, security holes, mistaken assumptions or side effects. We call the situation of unintended aliasing either *capturing* or *leaking*. Capturing occurs when objects, provided as a parameter, are stored permanently by the receiver object. This usually happens in constructors or mutator method calls. As the method caller still holds a reference to the captured object, it may by-pass the capturer's interface, which can lead to problems as shown in [Mül05]. Leaking occurs when references, supposed to be internal, are passed to the outside. Thus, external objects that call the leaking method gain access to the receiver's internal structure and can by-pass its interface. Leaking often happens by mistake and can lead to security holes, as occurred in an early release of the JDK, version 1.1 [AC04]. Moreover, potential referencing and modification between any objects in the system makes it hard to reason about a program's correctness, i.e. its compliance with the specification. In order to cope with the posed problems in presence of aliasing, several suggestions regarding the alias structure have been made. [HLW$^+$92]

One promising solution that enables us to control aliasing is to define and enforce clear ownership relations in an application's object space. Each object has at most one owner and all objects with the same owner are located in the same *ownership context* hereafter referred to as *context*. All objects without an owner are located in the so-called *root* context. Aliasing between contexts is restricted. If an object X is declared as the owner of object Y, all objects that do not have X as their owner are not allowed to modify Y.

A simple but strict aliasing policy proposed in [CPN98] is called *owner-as-dominator* and requires all reference-chains to an object in the context C to go through C's owner. Hence, the owner can control all references to its context and define *how* the objects should be accessed. Figure 1.1 illustrates this property considering the example of a bar. The barkeeper is the owner of all bottles, i.e. only he can access them. If customers want a drink, they must call the barkeeper to mix one, using the liquor bottles. The bottles do not change their owner as the barkeeper does not sell whole bottles. In the case of beer bottles, the owner-as-dominator property would not

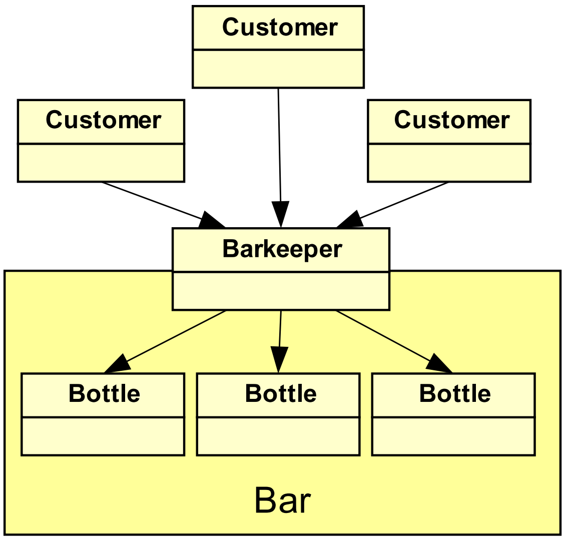


Figure 1.1: Owner-As-Dominator

be fulfilled as beer bottles are passed out to the customers, enabling them to directly access the bottle.

Although the owner-as-dominator property is simple and powerful, enforcing a clear object structure and encapsulation is too restrictive since it does not allow program idioms like iterators. Efficient iterators traverse data structures by holding a reference to the internal node that is currently being visited. However, as the data structure is the owner of its representation, i.e. all internal nodes, direct references from the iterator are prevented. Thus, an iterator implementation satisfying the owner-as-dominator property fails.

Ownership annotation syntax and correctness of an ownership structure are defined and ensured by *ownership type systems*. A correct ownership structure guarantees compliance of all aliases with the system's aliasing policy. No encapsulation breaching occurs, and unintended aliasing, such as capturing or representation exposure through leaking is prevented. This thesis reviews three major ownership type systems, namely the Universe type system [DM05] developed by P. Müller, Ownership Types [CPN98, BLS03], originally proposed by D. G. Clarke, J. M. Potter, and J. Noble, and Ownership Domains [AC04] from J. Aldrich and C. Chambers. In order to still allow programming idioms like the iterator design, all three type systems provide a concept on how to relax the too strict owner-as-dominator policy. Each concept is explained in detail in Part I.

The prime example to illustrate the benefits of ownership is a linked list implementation. By declaring the list object as the owner of its internal nodes, potential node modifications by external objects are forbidden. Regarding this example, the ownership approach looks promising. Present research is applying ownership type systems to real world applications with the conclusion that it is possible to apply ownership to an application of reasonable size, albeit some restructuring of the application is necessary [Häc05].

The question remains if all common programming idioms can by typed using an ownership type system. Hence, the goal of this Master thesis is to apply ownership type systems to design patterns covered in [GHJV95]. Design patterns express core ideas and best-practices on how to structure an application's design. They are widely used in practice and therefore essential for an applicability evaluation.

The thesis is structured into three parts:

Part I explains the reviewed ownership type systems in detail and gives an example typing of the well known linked list with an iterator scenario. The part's main purpose is to familiarize the reader with the different ownership type systems.

Part II builds the core of the thesis, a review of all design patterns in terms of ownership. Each pattern discussion points out the desired ownership structure and its feasibility under each reviewed ownership type system. Like in [GHJV95], the patterns are divided into three groups: creational, structural and behavioral patterns, where each group constitutes a separate chapter. Chapter 8 summarizes the results and contains a list of the main problems encountered during this research.

Part III describes our experiences in applying the concept of ownership to a combination of multiple patterns. Chapter 9 reviews ownership feasibility for an example application which has a Java Swing graphical user interface and its core design reveals a high pattern density. Chapter 10 extends our research on ownership to the Java Swing GUI toolkit, providing possible ownership structures and discussing potential impacts on applications, such as the one from Chapter 9. Chapter 11 concludes the research and provides a feature list for a flexible ownership system, capable to successfully type all patterns. Chapter 12 mentions related work in the area of alias control.

Part I

Ownership Type Systems

2 The Universe Type System

2.1 Concept

The Software Component Technology Group at ETH Zürich has developed the Universe type system and implemented it in the Multi Java compiler [Mul] and the JML Tools [JML]. JML stands for Java Modeling Language and is a behavioral interface specification language, used to specify the behavior of Java modules and establish software contracts [Mey92, LC05].

As explained in the introduction, the owner-as-dominator property is too restrictive since programming patterns such as iterators cannot be implemented. The Universe type system therefore weakens the owner-as-dominator property by allowing references to any object, but references into a foreign ownership context may not be used for updates. Only the owner and objects in the same context can thus modify each other and object encapsulation is still guaranteed. We call this property *owner-as-modifier*.

A big difference between the owner-as-dominator and the owner-as-modifier property is that the owner-as-dominator property restricts *where* object references may point to whereas the owner-as-modifier property restricts *how* an object reference can be used.

As shown in [BDM$^+$04, LM04, Mül01], the owner-as-modifier property enables modular verification of functional correctness properties and can handle programming patterns that cannot be implemented under the owner-as-dominator scheme.

In general, the Universe type system introduces contexts to structure the object heap, where all objects with the same owner reside in the same context. Objects without owner are located in the *root* context.

2.2 Universe Modifiers

The Universe type system extends each reference type with an ownership modifier. Ownership typing can be checked statically and any inconsistencies will be reported to the programmer during the type check.

Definition 2.1. Each reference type has one of the following modifiers:

- `peer` denoting a reference to an object located in the same context as `this`
- `rep` denoting a reference to an object residing in the context `this` owns.
- `readonly` denoting a reference pointing to an object in any context.

Annotation of the code is straight-forward by just extending all type declarations with one of the three modifiers.

2.3 Purity

Definition 2.2. We call a method *pure* if it does not modify existing objects.

The declaration of pure methods is done by simply adding the keyword *pure* in the method signature.

Rule 2.1. Pure methods must comply with the following properties:

- They may only call pure constructors and methods
- They must not update fields.
- They may only have `readonly` parameters.

Knowing which methods are pure is not only crucial when checking if the `readonly` modifier has been violated but also for formal specification using JML since only pure methods can be called when checking preconditions, postconditions or invariants. Otherwise, running a program with specification checking enabled would lead to a different run-time behavior than running the same program without specification checks.

2.4 Readonly References

The introduction of a `readonly` modifier requires to check that no object reference marked as `readonly` is used to modify the object's state. To ensure this, we need the code to be compliant with the following rules:

Rule 2.2. Fields may not be updated using a `readonly` reference in a reference chain.

Rule 2.3. `readonly` references may not be used in a reference chain to call non-pure methods.

2.5 Object Creation

Rule 2.4. All objects need an owner when created and therefore only peer and `rep` types are allowed for `new`-expressions

A new object can never be the owner of an already existing object since every object needs an owner upon creation and ownership transfer is not permitted. As a consequence, constructors cannot have a `rep` modifier in their signature.

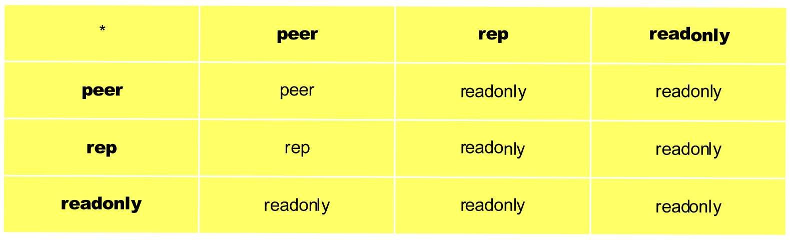
Definition 2.3. We call a constructor pure if it complies with the same restrictions as constructors and pure methods, but:

1. instance fields of `this` can be initialized
2. parameters must be declared as `peer` or `readonly`.

2.6 Type Combinator

This section describes how we can determine the *ownership type*, here simply referred to as *type*, of field accesses, method parameters and results. As these scenarios involve two types, namely the type of the target and the formal type of the declaration, a type combinator function is needed to determine the resulting type.

Figure 2.1 outlines the type combinator function. If the first argument is `this` the type combinator is not used. Whenever a `readonly` reference is part of the reference chain, the resulting type is `readonly` again. If the second argument is of type `rep` (meaning the reference chain will point into another context), the resulting type is `readonly` since no object is allowed to access another object's representation.



*	**peer**	**rep**	**readonly**
peer	peer	readonly	readonly
rep	rep	readonly	readonly
readonly	readonly	readonly	readonly

Figure 2.1: UTS Type Combinator Function. The first argument is indicated in the first column, the second argument in the first row.

2.7 Method Calls

The ownership type of a return value or method parameter is relative to the receiver of the method call. The resulting type is determined using the type combinator function.

2.8 Arrays

Arrays need two ownership modifiers, the first one for specifying the type of the array object, the second one for the elements of the array. Consequently, all objects contained in an array have the same ownership modifier. This does however not imply that all objects in the array are located in the same ownership context since a `readonly` reference can point to any context.

2.9 Static Methods

Static methods have no receiver which raises the question how to interpret the ownership modifier. The Universe type system treats the caller of the static method as the receiver object and all ownership modifiers are accordingly interpreted. As a consequence, the ownership modifier `rep` is not allowed in static method declarations.

2.10 Static Fields

The Universe type system treats the accessor of the field as the receiver object. This has the implication that all static fields must be `readonly` because they can be accessed from every context and therefore cannot be `rep` or `peer`. Treating all static fields as `readonly` is sound but too restrictive. In order to make a code transition into the Universe type system easier, all static fields are currently annotated implicitly as `peer` and a warning is emitted during the check.

2.11 Subtyping

Subtyping in the Universe type system follows the subtyping rules in Java, meaning that two `rep`, `peer` or `readonly` types are subtypes if the corresponding Java classes are subtypes of each other. Additionally, we introduce the following rule:

Rule 2.5. `rep` and `peer` types are subtypes of the corresponding `readonly` type.

The above subtype relationship holds as `rep` and `peer` types are more specific than the `readonly` type. The introduction of subtypes leads to the question of casts. Like in Java, casts are allowed in the Universe type system but since casts cannot be checked statically, runtime checks have to be introduced when performing an ownership type cast [Sch04].

2.12 Example

The following example shows the usage of the Universe type system in an implementation of a linked list together with an iterator. Figure 2.2 shows the object structure and the ownership contexts. The class `LinkedList` consists of double-linked nodes, representing the internal data structure where each node has a pointer to its data element. The class `LinkedList` has a `rep` reference `first`, pointing to the first node in the list. The iterator is declared with two references, `list`, pointing to the linked list over which iteration should be performed and `current`, pointing to the actual node. Since all nodes are located in the context owned by the list, the iterator may only have a `readonly` reference to the nodes. The actual data elements of the list are not contained in the linked list's context since the list is not the owner of its *content*.

```
class Node {
       peer Node prev, next;
       readonly Object elem;
       pure Node() {}
}

class LinkedList {
       rep Node first;
       pure LinkedList() {
              first = new rep Node();
       }

       //returns an iterator instance for the list as peer
       pure peer Iterator getIterator() {
              return new peer Iterator(this);
       }
}

class Iterator {
       peer LinkedList list;
       readonly Node current;

       //@invariant current.owner == list;
       pure Iterator(peer LinkedList l) {
              list = l;
              current = l.first;
       }

       readonly Object getNext() {
              readonly Object element = current.elem;
              current = current.next;
              return element;
       }

       pure boolean hasMoreElements() {
              return (current != null);
       }
}
```

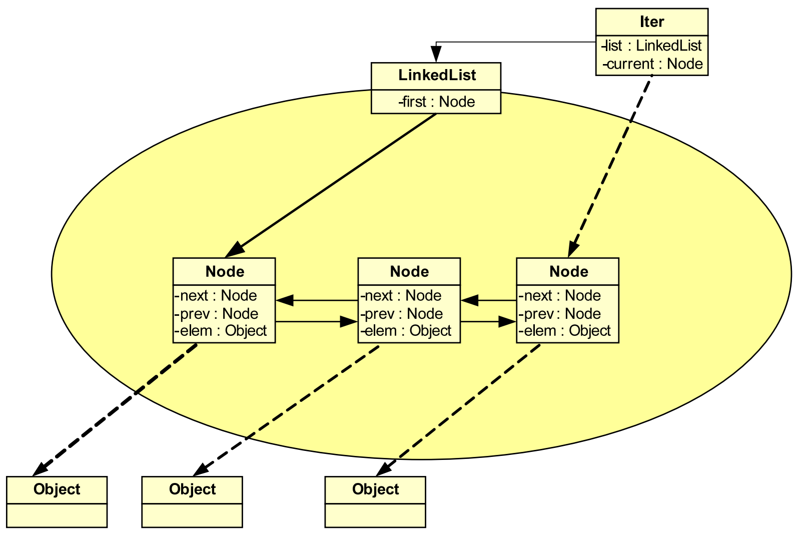


Figure 2.2: Linked List with Universes

3 Ownership Types

The idea to introduce ownership types for flexible alias protection was outlined in [CPN98], but the proposed concept was too strict to support constructs like iterators. Multiple suggestions on how to improve the system followed. We would like to take a look at the concept(-extension) proposed by C. Boyapati, B. Liskov and L. Shrira.

In [BLS03] they argue that the right way to relax the owner-as-dominator property is to allow inner classes to access the declarator's representation because a class and its inner classes can be reasoned about as one module.

3.1 Object Encapsulation

The Ownership Types concept aims to simplify the reasoning about a program's behavior and correctness by introducing different ownership types for *object encapsulation*. All objects with the same ownership type binding are encapsulated in the same context. This enables the programmer to reason about each context locally. The behavior of objects in a different context does not need to be considered.

Dependency is defined as in [BLS03]:

> ”An object x depends on subobject y if x calls methods of y and furthermore these calls expose mutable behavior of y in a way that affects the invariants of x.”

In the case of the linked list example we can clearly say that a list object is dependent on the nodes since the structure of the nodes can invalidate a list's invariants, e.g. that the list becomes cyclic. However, a list is not dependent on the objects it stores. An object should own all its dependent objects.

As shown in Figure 3.1, an ownership structure can be visualized as a tree with objects as nodes where each node denotes the owner of its children. Ownership relations are defined as follows:

Definition 3.1 (Ownership Relation)**.** $y \preceq x$ holds if either y and x denotes the same object or y is a descendant of x in the ownership tree.

Rule 3.1 (Access Rights)**.** Every object can access:

1. `this` and objects `this` owns.
2. its ancestors and objects they own (directly, not transitively).
3. globally accessible objects.

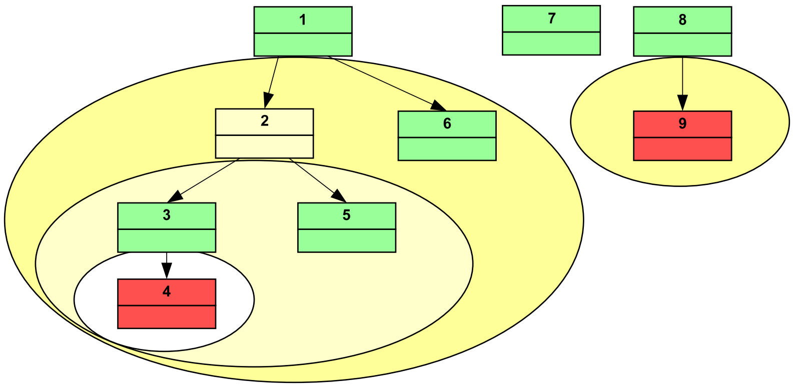


Figure 3.1: In this example we illustrate the access scope of object 2. Object 2 can access all objects it owns (3,5), its ancestors and objects they own (1,6), globally accessible objects (7,8). Access to objects 4 and 9 is forbidden.

3.2 Type Declaration

The Ownership Types concept works with owner parameterization, following the Java generics approach. Each class declaration may have one or more ownership type parameters (i.e. $T < x_1, x_2, ..., x_n >$). The first parameter represents the actual owner of the instantiated class and the other parameters are used to propagate ownership information. Objects without owner belong to the so-called `world` context and are globally accessible. An ownership type binding may either be `this`, `world` or any other object. With the proposed parameterization where multiple parameters can be passed, parameter constraints have to be defined in order to ensure that encapsulation can be guaranteed and is also correct in the presence of subtyping.

Rule 3.2. For a type $T < x_1, x_2, ..., x_n >$ with method $m < x_{n+1}, ..., x_k > (...)\{...\}$ the following constraints must hold:

1. (Subtype Constraint) $x_1 \preceq x_i \forall i \in \{1..n\}$
2. (Method Constraint) $x_1 \preceq x_i \forall i \in \{1..k\}$

An example and a graphical illustration of the resulting problems when neglecting the above constraints is given in Figure 3.2. In the presence of subtyping, the first parameter, specifying the owner, of the supertype must equal the first parameter of the subtype.

The Ownership Types concept relaxes the owner-as-dominator property by allowing inner classes to access the representation of the declarator class. This is not allowed for an inner class by default. If the inner class is instantiated with the owner type `C.this`, where C represents the declarator class, it is possible for the inner class to access the hidden representation.

Unfortunately, the treatment of static methods and fields has not been specified explicitly in [BLS03]. The following two sections describe how static fields and methods are interpreted in terms of ownership throughout this thesis.

Static Fields

When accessing a static field we have the problem that no receiver object exists and consequently no ownership parameters have been specified. Since it is possible to access static fields from any context we have to ensure that the field is in the root context in order to guarantee correct object encapsulation.

Rule 3.3. All static fields must have `world` as owner.

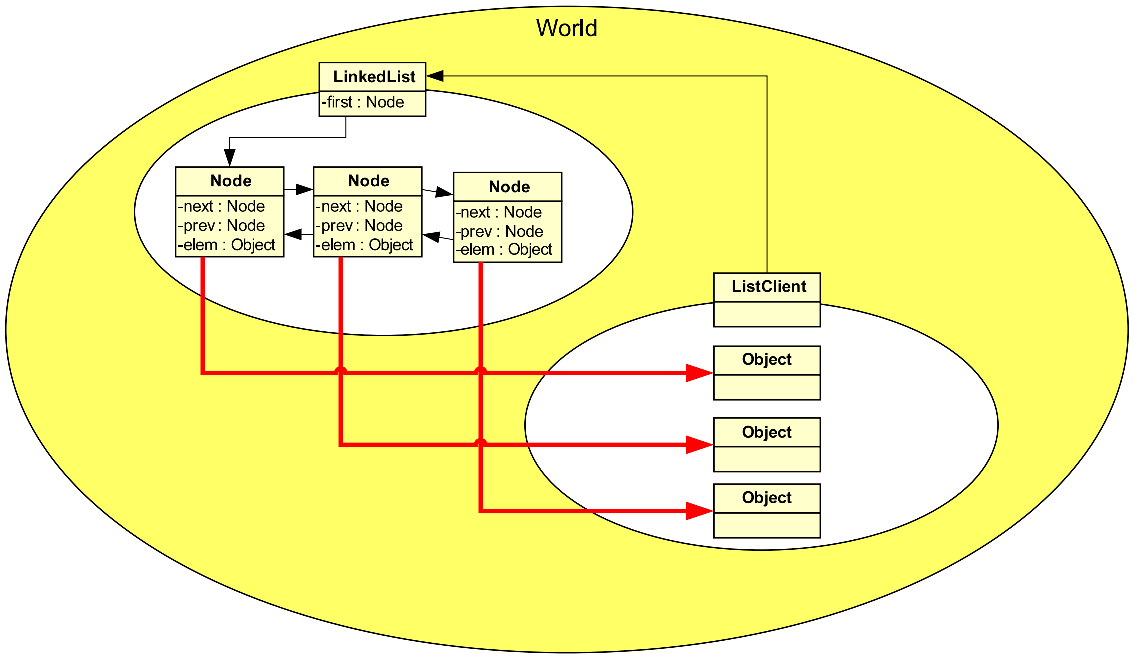


Figure 3.2: Example where constraint 3.2 does not hold. `ListClient` instantiates `LinkedList` with ownership parameters `<world, this>`, resulting in the situation where the `LinkedList` belongs to the `world` context and the actual list objects (`Object`) belong to the `ListClient`'s context. Since $world \not\preceq this$, the instantiation is illegal and the resulting context access violation is represented as red arrows.

Static Methods

Analogously, we have the problem that upon a static method call, no receiver object exists and due to this, no ownership parameters have been specified. We suggest using the same approach as the Universe type system does: treating the method caller as the receiver. This has the consequence that the caller object must now comply with the method constraint rule 3.2.

The following code illustrates our proposition:

```
class A<Aowner> {
       public static F<world> field;

       public static void foo<fooParameter>() {...}
}

class F<Fowner> {...}

class Client<clientOwner> {

       public void test() {

              /* Static fields can be accessed from every context
               * since they reside in the world context.
               */
              A.field = new F<world>();

              /* In static method calls the caller is treated as the receiver
               * and must therefore comply with the method constraint rule.
               */
              A.foo<world>();
       }
}
```

Instead of $Aowner \preceq fooParameter$, the constraint $clientOwner \preceq fooParameter$ must hold. This is the case because $clientOwner \preceq world$.

3.3 Effect Clauses

For the same reasons the Universe type system makes use of JML to additionally provide specifications, Ownership Types introduces so called *Effect Clauses*. These clauses enable the programmer to specify a read and write set of a method, naming all the objects which will be read and written respectively during method execution. If we consider for example a method that defines a write set $write(w_1, w_2, ..., w_n)$ and a read set $read(r1, r2, ..., r_m)$, the method can write all objects x for which $x \preceq w_i$ where $w_i \in \{w_1..w_n\}$ holds and it can read all objects y for which $y \preceq r_i$ or $y \preceq w_i$ where $r_i \in \{r_1..r_m\}$ and $w_i \in \{w_1..w_n\}$ holds. Hence, objects enlisted in the write clause can be read and written.

In presence of subtyping, effect clauses must extend already declared clauses in the supertype. Due to this, it is sometimes hard to specify correct effect clauses because we can only name the supertype elements since we do not know at this point of time on what concrete subtypes the declared class will operate. We can illustrate this considering a linked list: during implementation, effect clauses can only contain the type `Object` since we are not yet aware on what concrete subtypes the linked list will operate. In order to be able to deal with such cases, the system lets us include `world` in the effect clauses.

As with any other ownership system, many type declarations can be defaulted and therefore left out [BLR04, BR01].

3.4 Example

Once again we illustrate a concrete typing of a linked list, using the system presented. The iterator may have access to the list's context because `Iterator` is implemented as an inner class of `LinkedList`. The actual elements contained in the list belong to `world` since only the nodes belong to the list's representation. As shown in Figure 3.3, an iterator may either be instantiated in the client's context (`Iterator`) or in the `world` space (`Iterator2`). The mechanism of instantiating inner classes in the `world` context enables us to model wrappers, meaning that we can create globally accessible objects that define an interface for manipulating encapsulated data structures.

```
public class Node<nodeOwner, elementOwner> {
        Node<nodeOwner, elementOwner> prev, next;
        Object<elementOwner> elem;
}


public class LinkedList<listOwner, elementOwner> {
        Node<this, elementOwner> first;

        public LinkedList() {
                first = new Node<this, elementOwner>();
        }

        //returns an iterator instance for the list
        Iterator<iteratorOwner, elementOwner> getIterator<iteratorOwner>()
                where (iteratorOwner <= elementOwner)
        {
                return new Iterator<iteratorOwner, elementOwner>();
        }

        /* The iterator is implemented as an inner class of LinkedList
         * so that the iterator is able to access the list's representation.
         */
        class Iterator<iteratorOwner, elementOwner>
        {

                Node<LinkedList.this, elementOwner> current; //current position

                Iterator() {
                        current = LinkedList.this.first;
                }

                Object<elementOwner> getNext() {
                        Object<elementOwner> element = current.elem;
                        current = current.next;
                        return element;
                }

                boolean hasMoreElements() {
                        return (current != null);
                }
        }
}


public class ListClient<clientOwner>{

        public void test() {
                LinkedList<this, world> list = new LinkedList<this, world>();

                //illegal since world !<= this:
                //LinkedList<world, this> list2 = new LinkedList<world, this>();

                Iterator<this, world> iterator = list.getIterator();

                //Unencapsulated iterator that can be freely passed around since it belongs to world:
                Iterator<world, world> iterator2 = list.getIterator();
        }

}
```

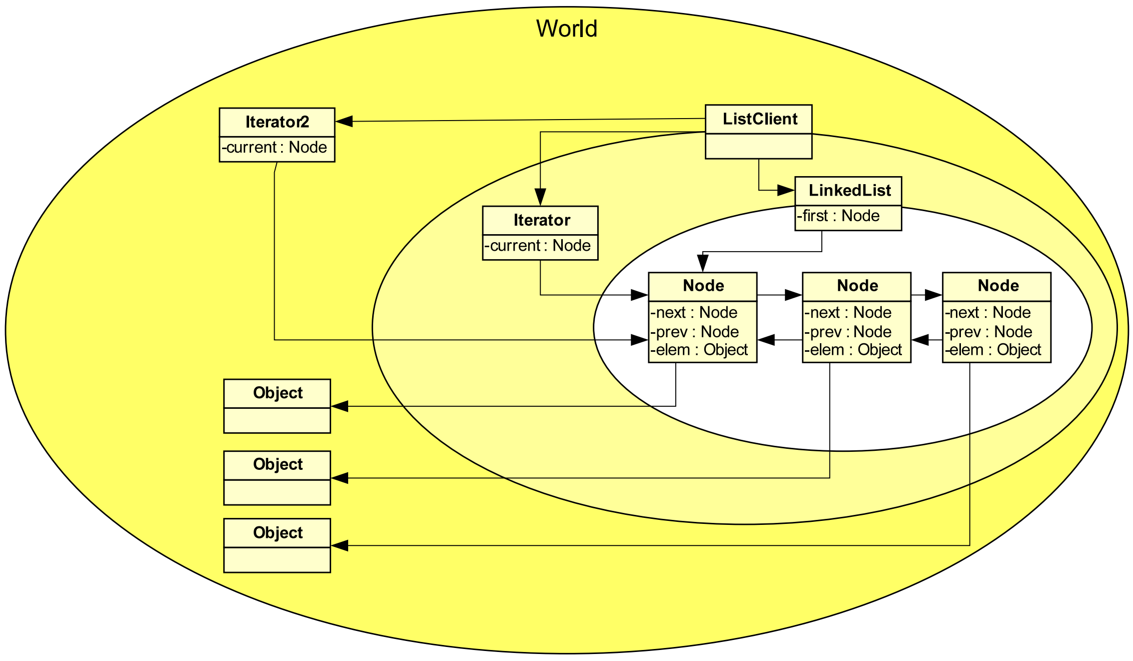


Figure 3.3: Visualization of the example

4 Ownership Domains

J. Aldrich and C. Chambers proposed a concept called Ownership Domains, another approach to alias control. In [AC04] they argue that previous ownership type system proposals are too strict and the implementation of many common programming idioms is not possible. Therefore, they developed the idea to separate the object store into different *domains* where visibility and aliasing between domains is restricted. Their idea is to separate aliasing policy from mechanism. In most ownership type systems each object has exactly one ownership context, holding all owned objects. The aliasing policy is fixed, i.e. the same for all objects in a system. In contrast, ownership domains provides more flexibility by allowing each class to declare multiple domains, enabling alias control between objects with the same owner. The aliasing policy can be defined by the declaration of *links* between domains. A link enables all objects in the source domain to reference objects in the target domain. Ownership Domains defines two special domains, namely `shared` as the top-level domain, and `owner` denoting an object's owner domain. If necessary, the *owner-as-dominator* property can still be ensured.

The terms *domain* and *ownership context* are used as synonyms throughout this thesis.

4.1 Concept

The concept of Ownership Domains is implemented in AliasJava, a special Java extension that provides alias control capabilities. Each object can declare domains in which newly created objects reside. As stated in [AC04] we introduce the following access rules and policies:

Rule 4.1 (Alias permissions)**.** The following rules define an object's access rights:

1. An object has permission to access objects in the *same* domain.
2. An object has permission to access objects in the domains that it *declares*.
3. Domains can be *linked*, allowing each object in the first domain to access objects in the second domain.
4. Domains can be declared *public*, denoting that access rights to an object implies access rights to the object's public domains.

Rule 4.2 (Intransitivity)**.** Access rights are *not* transitive.

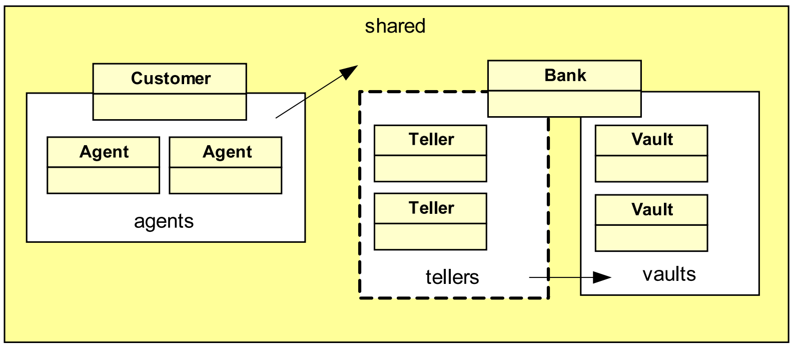


Figure 4.1: Conceptual view of domains as presented in [AC04]. Dashed rectangles represent public domains whereas solid rectangles are private domains. Arrows represent a link between domains. The `shared` domain is the top-level domain and by linking `agents` with `shared` we ensure that all agents gain access to the `Bank` object and all objects in the public domains it declares, namely the `tellers`. Note that only the tellers have access to the vaults and not the `Customer` / `Agent`.

Rule 4.3 (Object Creation)**.** An object O can only create objects in

- domains declared by O
- the `owner` domain of O
- the `shared` domain

Aliasing permission through a declared link does not imply object creation rights.

Rule 4.4 (Link Constraints)**.** Links can only be declared under the following conditions:

1. Each link declaration must include a locally-declared domain
2. An object can only link a local domain to an external domain d if the `this` object has permission to access d.
3. An object can only link an external domain d to a local domain if d has permission to access the `owner` domain.

Figure 4.1 provides a conceptual view of the concept. Since behavior of the ownership system in presence of static methods and fields has not been specified explicitly, the following two sections explain all assumptions on the system's behavior.

Static Fields

Assigning a domain to a static field is difficult because domain parameters have not been specified in a static context and domains are declared per object, not per class. We must therefore enforce that all static fields must be located in the root context `shared`. All domains, holding objects that need access to static fields, have to be linked to the `shared` domain.

Static Methods

Domain access rights are checked with respect to the static method caller. Moreover, static methods may only manipulate objects in the `shared` domain or a domain passed as a method domain parameter (m<domainParameter>(...){...}). Domains declared in the class or specified with a class domain parameter may not be used since no receiver object exists in a static context.

4.2 Parameterized Types

Domain parameterization is necessary for situations where different instances of a class act in different ownership domains which are specified in a parameter upon class instantiation. This is similar to the presented Ownership Types concept. The main difference between the two systems is that in Ownership Types the parameters have object bindings denoting the owner, whereas in Ownership Domains parameters have domain bindings. For example, a class `List` has to act as a container for instances of any type, in any domain. Therefore the list needs to be parameterized, not only with the type of objects it stores, but also with the ownership domain in which the elements reside. Parameterization is implemented by leveraging Java generics to use ownership information in addition to the type specification. In this case, a generic parameter is not only specified by the type but also by its associated domain. A special parameter annotation is *lent* which indicates that the parameter reference is only temporary and may thus not be captured by the receiver. Only *unique* or *owned* objects can be passed as *lent* and the receiver gains full access to the supplied object within method scope.

4.3 Example

This section illustrates how a linked list can be implemented together with an iterator using Ownership Domains. As outlined in Figure 4.2, the `LinkedList` class declares two domains: `owned`, containing the internal representation of the list and `iters`, containing the iterator. List traversal rights are guaranteed by introducing a link from the `iters` domain to the `owned` domain. List usage is shown by introducing a client class named `ListClient` which spans a domain called `state`, containing the linked list together with the data objects that are inserted into the list. Once again the data objects do not belong to the internal `owned` domain of the `LinkedList` class. The data objects belong to the client and only the nodes need to be shielded from outside. Since `ListClient` needs access to the iterator the `iters` domain is declared `public` so that the client may get an alias of the iterator.

Due to some problems we encountered with the current Ownership Domains command line tool, the example list below does not make use of Java generics and the stored elements are of type `Object`. Nevertheless, the example makes use of other presented features like domain parameterization, assumes-clauses, etc.

```
class Node<elements>
	assumes owner -> elements.owner {

	owner Node<elements> prev, next;
	elements Object elem;

}

class LinkedList<elements>
	assumes owner -> elements.owner {

	//domains
	domain owned;
	public domain iters;

	//links
	link owned -> elements.owner;
	link iters -> elements.owner;
	link iters -> owned;

	owned Node<elements> first;

	LinkedList() {
		first = new Node<elements>();
	}
```

```
       iters Iterator<elements> getIterator() {
              return new ListIterator<elements, owned>(first);
       }

       //other methods omitted...

}

interface Iterator<elements> {
       elements Object next();
       boolean hasMoreElements();
}

class ListIterator<elements, list> implements Iterator<elements>
       assumes list -> elements.owner {

       list Node<elements> current;

       ListIterator(list Node<elements> head) {
              current = head;
       }

       public elements Object next() {
              elements Object element = current.elem;
              current = current.next;
              return element;
       }

       public boolean hasMoreElements() {
              return (current != null);
       }
}

class ListClient {

       //domains
       domain state;

       public void test() {

              /* Each object resides in a single domain from instantiation to garbage-
               * collection. Consequently, the variable list must be final in order
               * to ensure that the domain list.iters cannot change at run-time.
               */
              final state LinkedList<state> list = new LinkedList<state>();
              list.iters Iterator<state> myIterator = list.getIterator();
       }
}
```

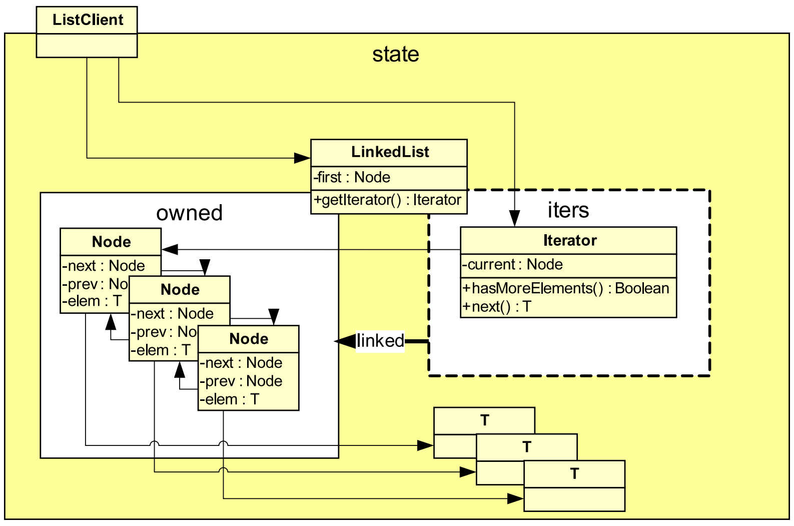


Figure 4.2: Illustration of the example. Dashed domain lines denote public domains.

Part II

Design Patterns

This part contains the classification and review of design patterns, covered in [GHJV95]. The goal is to examine design patterns with respect to ownership typing and it is assumed that the reader already has knowledge and experience using design patterns. We want to analyze every pattern in the context of each reviewed ownership type system, make comparisons between the systems and point out advantages, potential flaws and problems that come along with an ownership typing in general, and specifically when using one of the three concepts. Conclusions and possible system enhancement propositions will round up the review.

Each pattern review is made according to the following structure:

- **Intent** overview and recapitulation of the pattern's motivation and main usage scenario.
- **UML Diagram** overview of the pattern's structure.
- **Ownership Discussion** discusses if applying ownership to the pattern makes sense and if so, pointing out what the desired ownership structure is.
- **Universe type system** discusses ownership typings with the Universe type system.
- **Ownership Types** discusses ownership typings with Ownership Types.
- **Ownership Domains** discusses ownership typings with Ownership Domains.
- **Conclusion** summing up generally encountered problems and some of their solutions and providing a comparison between the different ownership systems.

5 Creational Patterns

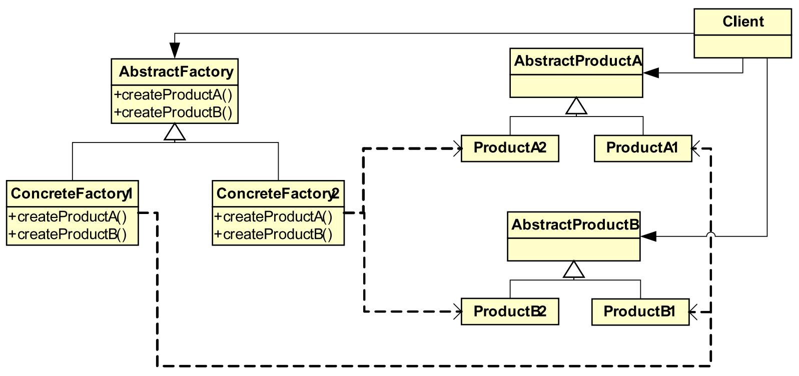
5.1 Abstract Factory

Intent

> ”Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”[1]

The prime example of an abstract factory is a widget factory that is responsible for creating different widgets for different platforms. A GUI application, able to run on multiple platforms, has to remain independent of platform-specific widget implementations. Therefore, the application logic only uses abstract widget classes and the actual widget creation is delegated to the factory.

UML Diagram



Ownership Discussion

Whether or not ownership is useful in connection with the abstract factory pattern depends on the benefits gained by a product encapsulation by the client. The factory itself is usually implemented as a singleton (5.5), serving multiple clients. Thus, the factory should be freely accessible by all clients, no matter in what ownership context they reside. Also, it is hard to make any assumptions

[1] [GHJV95] page 87

about the client's owner since a factory can have many clients with different owners. We will therefore investigate the benefits from ownership if each client could propagate himself as the owner of its products. Product encapsulation could enable the client to safely establish assumptions or invariants about the product's state, knowing that its products cannot be manipulated by objects outside its context.

We can therefore conclude that an ideal ownership structure for the abstract factory pattern would make the factory global accessible (from any ownership context) while each client would be the owner of its generated products.

We illustrate the desired ownership typing on the situation where we have an abstract product `Tire` that is extended by several concrete products (`Dunlop`, `Michelin`). Furthermore, we have the product owner `Car`, which is dependent on the product's state (is enough pressure in the tires?) and therefore the owner of all created products. The abstract factory (`Garage`) is an instance of either `MichelinGarage` supplying `Michelin` tires or `DunlopGarage` and it is responsible for creating all products.

Universe type system

Applying ownership to the abstract factory pattern is not easy as we face the situation where a class is instantiated by an object which will not be the owner. In our example we would like the car to be the owner of its tires, ensuring that no other object can manipulate the car's state by modifying the tires. Therefore, `Car` should have a `rep` reference to all its `Tire` objects, as illustrated in Figure 5.1.

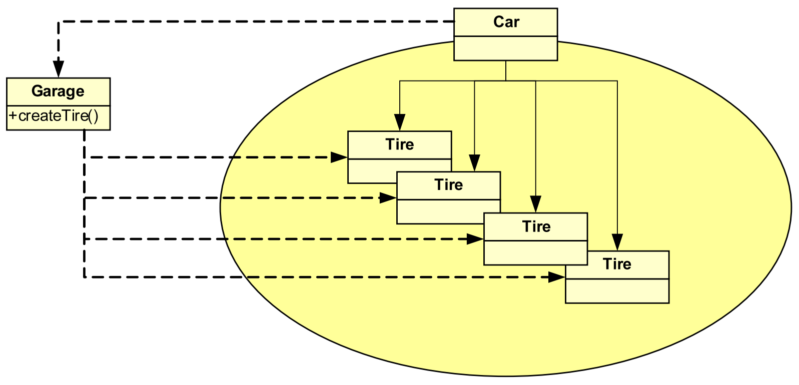


Figure 5.1: Illustration of the abstract factory pattern with the Universe type system. Dashed lines represent `readonly` references.

The problem here is that according to the object creation rule 2.4, only `peer` and `rep` types are allowed for `new`-expressions. However, since `Garage` creates the `Tire` objects, they must either reside in a context spanned by `Garage`, meaning that `Car` cannot be the owner of its tires, or that the garage must be in the same context as the tire objects. The first case is not satisfactory because we need to ensure that a car has full access rights to its tires. In the latter case, the garage would be owned by the car, requiring each car to have its own garage. This solution is also not feasible, considering that many abstract factories are implemented as singletons (5.5).

We recognize that patterns, where the creation of an object is delegated to other classes, are not satisfyingly supported by the system.

Solutions

Creation by callbacks One possible workaround for this problem is to use a callback mechanism: the product owner still works on abstract products, delivered by an abstract factory, and remains hereby independent of whatever concrete product is in place. Additionally, it is also the creator of the concrete product, whereas the question *which* concrete product has to be created is still determined by the underlying concrete factory.

When needing a new `Tire` instance the `Car` simply calls the `Garage`'s `createTire()` method. Depending on the used concrete factory, `createTire()` will perform a callback on the car in order to get its concrete instance (e.g. `createMichelin()`). If the products' constructors are `pure`, object creation can be solved in this way. The remaining problem with this approach is the readonly reference between the factory and the products, preventing us from performing further initialization on the created object in the factory's scope.

A big disadvantage of the callback approach is that it results in a programming overhead (or even code duplication) because we need to implement `createConcreteProduct()` methods for all concrete products in the product owner. If we have several different product owner classes, moving all creational methods up in an abstract superclass is advised so that they only have to be implemented once.

The greatest disadvantage is that due to the callback we introduce a class dependency from the abstract factory to the product owner, and from the product owner to concrete products. Total independence of the concrete products cannot be guaranteed anymore and partial deployment (where the application is only deployed with a subset of all available concrete factories) is not possible. Since we need to introduce for each concrete product a creation-method in the product owner, we loose a great portion of flexibility and independence compared with the original pattern proposition.

We present an example implementation in the following listing:

```
public abstract class Owner {

       public pure rep Tire createMichelin() {
              return new rep Michelin();
       }
}

public class Car extends Owner {
       private rep Tire firsttire;

       public void init(readonly Garage g) {

              /* createProduct() returns a readonly reference to the result. But,
               * since we are the creator of the resulting rep object, we can do a downcast here.
               * If createProduct() is called with a parameter different from this, this downcast
               * would fail.
               */
              firsttire = (rep Tire) g.createProduct(this);
       }
}

public abstract class Garage {
       public abstract pure readonly Tire createProduct(readonly Owner owner);
}

public class MichelinGarage extends Garage{
       public pure readonly Tire createProduct(readonly Owner o) {
              return o.createMichelin();
       }
}

public class Test {
       public void test() {
              rep Garage g = new rep MichelinGarage();
              rep Car car = new rep Car();
              car.init(g);

       }
}
public abstract class Tire {...}
public class Michelin extends Tire{...}
```

Using a factory instead of an abstract factory Depending on the scenario, we might be able or willing to modify our design that the factory is no longer abstract and extended by several concrete factories. There is only one concrete factory, still returning an abstract product in the `createProduct()` method, but deciding within this method what concrete product shall be instantiated. In this situation, the `createProduct()` method could be implemented as static. As explained in Chapter 2, static method calls are always evaluated in the context of the caller. This means in our case: if the factory would offer a static `createProduct()`-method, the object would be created in the context of the caller (the product owner).

The decision what concrete products will be created is still done by the factory. The problem is that the factory is no longer abstract and future product extensions can not be implemented by further subclassing but by modifying the factories `createProduct()`-method.

The other disadvantage of this pattern is that due to the static `createProduct` method we are not able to have multiple factory instances, supplying different types of products.

By using this approach we also loose a great deal of the proposed original design because the main intent of the pattern is to be able to dynamically introduce new products by subclassing. All existing code remains hereby unchanged and it is possible to only deploy or load a subset of the available products. All these obligations cannot be fulfilled by the proposed solution.

The following code snippet shows an example implementation of the proposed solution.

```java
public class Car {
	private rep Tire firsttire;

	public void init() {
		firsttire = rep Garage.createProduct();
		//...
	}
}

public class Garage {
	public static pure peer Tire createProduct() {
		if(...)
			return new peer Michelin();
		else if(...)

		//...
	}
}

public abstract class Tire {...}
public class Michelin extends Tire{...}

public class Test {
	public void test() {
		rep Car car = new rep Car();
		car.init();
	}
}
```

Introducing new universe modifiers Another suggestion would be to modify the Universe type system in a way to support new kinds of ownership types. In our case where object creation is delegated to an object in a totally different context, we need the ability of either ownership transfer or to create an object in the callers context. With this feature at hand we would be able to implement the abstract factory in the originally proposed way without loosing any aspect of the pattern.

We propose to introduce the following new universe modifiers:

- *callerrep*, indicating a `rep` ownership in the caller's perspective
- *callerpeer*, indicating a `peer` relation in the caller's perspective

Both new universe modifiers may only be used in conjunction with method parameter- or return-types and local variables.

Ownership Types

Once again we want to shield all tires on a car from the outside, this time using the Ownership Types concept. We therefore need to declare each car as the owner of all its tires. Unfortunately we face the same problem as we had when typing the abstract factory pattern with the Universe type system: The abstract factory needs the right to create products (`Tire`), although it is not located in the car's context. Allocating garage in the `world` context while passing the `Car` instance as the second ownership parameter so that the created tires can be put into the car's context is forbidden by the subtype constraint 3.2. Figure 5.2 illustrates the ownership scenario.

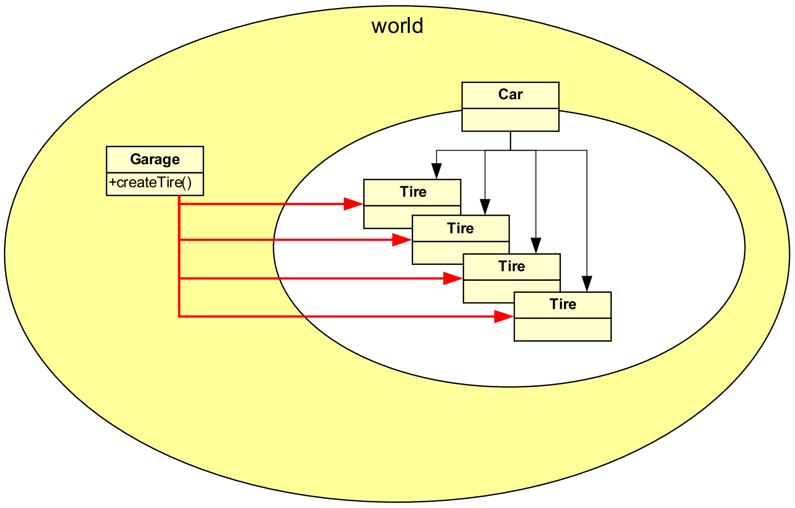


Figure 5.2: Illustration of the abstract factory pattern with Ownership Types applied. Accessing the `Car`'s context is not allowed for `Garage` (marked as red arrows) unless it is implemented as an inner class of `Car` which is not desirable, or the `createProduct()`-method is `static` and therefore evaluated in the context of `Car`.

The only way to let `Garage` create products, owned by a car, is to implement it as an inner class of `Car`. This is not realistic because of the many `Car` instances compared to presumably one `Garage` instance. The other way around where the Car is implemented as an inner class of `Garage` is also not desirable because we do not want `Garage` to be the owner of the tires.

It is also not possible to follow the callback approach because method constraints prevent us to write a method (`createProduct()`) returning an object which is not an ownership subtype of the classes owner. The following code illustrates the problem:

```
abstract class Garage<GarageOwner> {
      public abstract Tire<ProductOwner> createProduct<ProductOwner>();
}
```

Since `GarageOwner` would be instantiated with `world` and `ProductOwner` with `Car` and $world \not\preceq Car$, we cannot offer an abstract factory, delivering products that should reside in another encapsulated context.

Solutions

Using a factory instead of an abstract factory If we are able to use a factory offering a static `createProduct()`-method instead of an abstract factory, ownership typing can be done without further problems. Object creation and possible initialization of the product can be done within the static method because it is evaluated in the context of the caller, which is in our case a `Car` instance. Implementation of this approach is trivial and therefore omitted. All previously discussed risks and problems of this approach (5.1) remain.

Ownership Domains

When typing the abstract factory pattern with Ownership Domains, similar problems with object creation emerge. According to rule 4.3, an object can only instantiate objects in either the domain it resides (`owner`), the global (`shared`) domain or in all domains it declares, which leaves us with the following possibilities concerning our example domain structure:

1. Each product owner declares a domain in which both the products and the abstract factory instance resides. This is not desirable since we do not want a `Garage` instance for each `Car`.

2. An abstract factory could instantiate global (shared) objects. But in this case, all `Tires` would be located in the `shared` domain and we would therefore give up encapsulation of the `Tires`, resulting in the same situation as when there is no ownership in place at all.

3. The abstract factory spans a domain for all products. We would have `Garage` as the owner of the `Tires` instead of `Car`, which is also not desirable since, in contrary to the `Car` class, `Garage` is not dependent on the tires' state.

As we can see, following the object creation rules, the Ownership Domains approach does also not lead to a promising result with a straight-forward typing.

Solutions

Callback-Approach We would like `Car` to span a private context to hold the `Tire` instances. In addition, we need to ensure that the garage has access rights to the `tires` domain of each car in order to delegate product creation to the garage.

We suggest introducing a `factories` domain to hold all kinds of factories and linking the `factories` domain with each `tires` domain. Figure 5.3 illustrates our proposition. The declaration of the `factories` domain is essential as we only want to give the factories access rights to the products. The `factories` domain should also be declared public so that the `Car` can access it when declaring the link between `factories` and `tires`.

The problem with the described scenario is that even though access rights between the `factories` and the `tires` domain are guaranteed, the garage can, nevertheless, not instantiate any objects because access rights do not include object creation rights.

One option here is to use the callback mechanism previously mentioned in Section 5.1. Unlike in the Universe type system, we can even initialize the created objects within the factory's method because by linking `factories` with `tires` we gain full access rights to the `Tire` instances. All other previously discussed disadvantages of this approach remain. Most importantly:

- Due to the callback we introduce a dependency between the factory and the client.
- Adding new products leads to additional `createProductX()` methods in the owner.

The following code illustrates the callback approach:

```
abstract class Tire {}
class Dunlop extends Tire {}
class Michelin extends Tire {}

class Car<factories> assumes factories -> owner {
	domain tires;
	link factories -> tires;

	tires Tire tire;

	void init(factories Garage garage) {
		tire = garage.createProduct<tires, factories>(this);
	}

	//creation methods for each concrete product
	tires Michelin createMichelin() {
		return new Michelin();
	}
	tires Dunlop createDunlop() {
		return new Dunlop();
	}
}

abstract class Garage {
	abstract tires Tire createProduct<tires, factories>(Car<factories> car);
}

class MichelinGarage extends Garage {

	tires Tire createProduct<tires, factories>(Car<factories> car) {
		tires Tire tire = car.createMichelin();
		//more initialization code here...
		return tire;
	}
}

class DunlopGarage extends Garage {

	tires Tire createProduct<tires, factories>(Car<factories> car) {
		tires Tire tire = car.createDunlop();
		//more initialization code here...
		return tire;
	}
}

class Client {
	public domain factories;
	public domain productowners;

	public void test() {
		//create the product factory
		factories Garage garage = new MichelinGarage();

		//create the product owner
		productowners Car<factories> car = new Car<factories>();
		car.init (garage);
	}

	public static void main(String[] args) {
		Client c = new Client();
		c.test();
	}
}
```

Using a factory instead of an abstract factory Once again, the better workaround than using the callback-approach is to use a static method to create the product instead. Evaluation of a static method in the caller's context enables us to create a new product (`Tire`) in the scope of `Car` and perform further initialization on the `Tire` instance. Unfortunately, new products cannot be easily added by further subtyping but only by modifying the existing `createProduct()`-method. Furthermore, it is not possible anymore to have multiple instances of the factory in the same application where each factory supplies a different product.

```
class Car {
	domain tires;

	tires Tire firsttire;

	public void init() {
		firsttire = Garage.createProduct<tires>();
	}
}

class Garage {
	static boolean michelin;

	public static void setProduct(boolean michelin) {
```

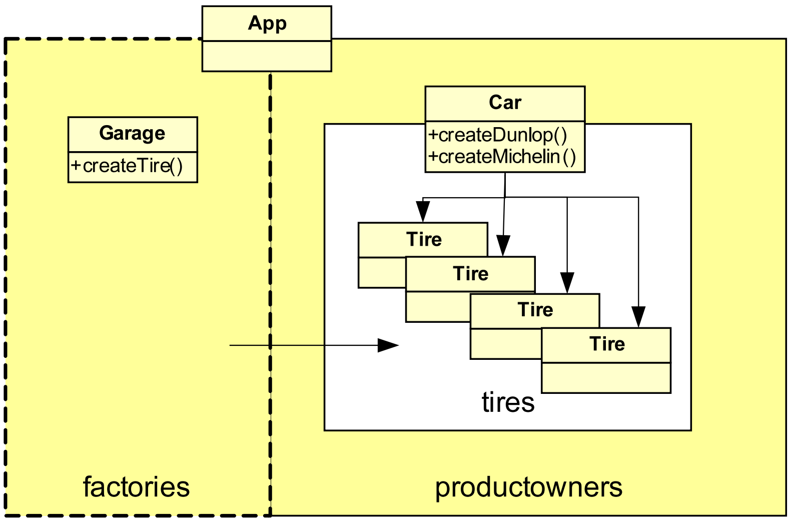


Figure 5.3: Illustration of the callback-approach with ownership domains in place. Public domains have dashed lines.

```
            this.michelin = michelin;
      }

      public static tires Tire createProduct<tires>() {
            if(michelin)
                  return new Michelin();
            else
                  return new Dunlop();
      }
}

abstract class Tire{...}
class Michelin extends Tire{...}
class Dunlop extends Tire{...}
```

System Extensions Other possible solutions to the object creation problem in foreign domains would require system extensions. One could think of introducing *creational links* from one domain to another, allowing objects in the first domain to access *and* create objects in the second domain. Another possible approach is to introduce new domain modifiers in order to be able to declare domains in which object creation is possible from outside. Detailed definitions and impacts on the existing system are the subjects of further research.

Conclusion

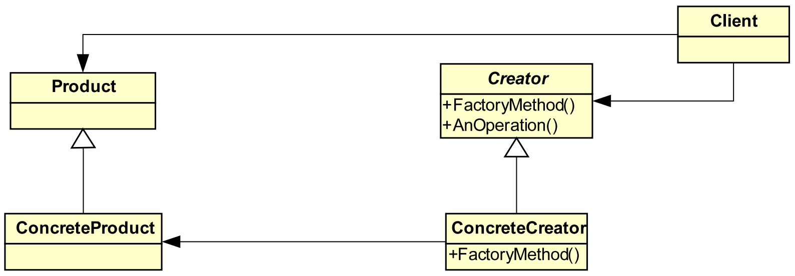
We can conclude that with existing ownership systems, typing of the abstract factory pattern is hard due to the delegation of object creation. All proposed solutions end up either with change requirements for the existing ownership system (the introduction of new ownership modifiers respectively new domain or link modifiers) or with slight modifications to the original pattern which, as a consequence, results in loosing some of the pattern's design aspects.

5.2 Factory Method

Intent

> ”Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.”[2]

UML Diagram



Ownership Discussion

Ownership typing of the factory method pattern is strongly related to the typing of the abstract factory pattern (5.1) since most abstract factories make use of the factory method pattern in order to manufacture the products. As in the abstract factory pattern, the client only operates on abstract products, not knowing which concrete products the factory method returns. Each different product has a concrete creator class which subclasses the abstract creator. The concrete product therefore depends on what concrete creator class is instantiated.

We would like to apply the same ownership typing as in the abstract factory pattern where the manufactured products are owned by or at least located in the same context as the client. Likewise, the creator is located in another context as there may be only one creator serving products for multiple different clients which may be located in different contexts.

The used ownership system has to cope with the omnipresent problem in presence of creational patterns: object creation is delegated and therefore the class that creates the product is not the desired owner. The optimal ownership system thus needs to provide a mechanism to transfer ownership of the created product to the client. Alternatively, it should support the creation of objects in the client’s (the factory method caller’s) context. Ownership type system extensions and possible workarounds for this problem have already been proposed in the abstract factory section (5.1) and we will skip a detailed discussion.

Conclusion

Due to the strong relation of the factory method pattern with the abstract factory pattern (in terms of ownership typing), detected problems and proposed workarounds or ownership type system extensions are the same. Once again we deal with the scenario where ownership typing is not satisfyingly applicable to *delegation of object creation* with the reviewed ownership systems.

5.3 Prototype

Intent

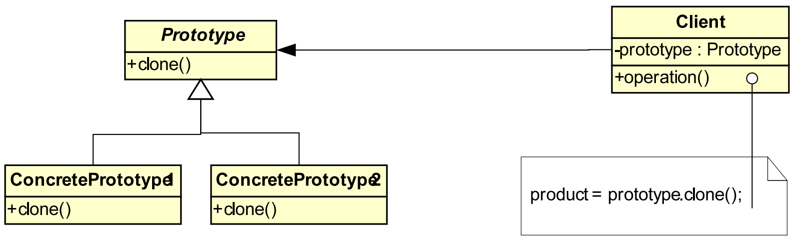
> ”Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.”[3]

[2][GHJV95] page 107
[3][GHJV95] page 117

The prototype pattern has some similarities with previously mentioned creational patterns like the abstract factory or the factory method in a way that the client code always operates on abstract products, not knowing the underlying concrete product. The big difference to other creational patterns is that building a parallel factory class hierarchy is not necessary. The client has a reference to a prototype instance and creates new products by cloning the prototype instead of calling the factory method. This design has several advantages over the factory based approach:

1. We need fewer classes because the factory class hierarchy can be omitted.
2. By using object composition instead of subtyping we might be able to specify new products by varying field values. As a consequence, we need fewer classes.
3. We can define new products at run-time by registering new prototypes.

UML Diagram



Ownership Discussion

When applying ownership typing to the prototype pattern we want to achieve that all created products are owned by the client. This will enable the client to safely establish assumptions over the product's state. An implementation where the client is dependent on its products will be safer. Furthermore, it will become easier to reason about the implementation's correctness since product manipulations by objects outside the context will not be possible anymore.

The prototype can be in any context because the client is only configured with it, not knowing on what concrete prototype it is operating. Therefore, we deal with the situation where an object can be located in any context but its clones should reside in the `clone()`-method caller's context. This is somehow similar to using a factory method to create objects as once again, the products are not created in the context we want them to be in. Trivially, having the prototype, the client and its products in the same context depicts no problem, but there would not be any benefits from ownership. In the following discussion we will use the term *prototype* for the original instance that will be cloned in order to get new products. The terms *product* and *clone* are used as synonyms and refer to the created clone of the prototype.

The problem is that although the prototype is located in a different context, we want its clone to be in a context owned by the client. The used ownership system therefore needs to support cloning of the prototype into the client's context or the transfer of ownership to the client after the prototype has been cloned.

Since none of the reviewed systems supports this feature, we will discuss a generally applicable workaround for this problem in the following and review its concrete implementation in the according sections.

The problem with clones

With all three reviewed ownership systems we have the following problem: an object X in context A can only clone itself into context B if either A = B or X is the owner of B. Ownership systems with support for a global context also offer the possibility to create global objects. But, in connection with cloning this does not help because we would like the clones to reside in a context owned by the caller of `clone()`.

An obvious workaround for the clone problem is to modify `clone()` to take an instance of the same class as parameter and copy its fields to `this` instead of returning a new clone of `this`. As a consequence, the new object will not be allocated in the `clone()` method anymore, but at the caller, who will then call the new clone method in order to set all fields correctly. We would rather call the new method `copy()` than `clone()` and show the approach in the following code snippet with Universe type system annotations:

```
class Client {
	readonly Cloneable c;

	public void test() {


		/* Original clone
		 *
		 * The problem is that we can only get a readonly reference to the clone
		 * when having a readonly reference to Cloneable.
		 */
		readonly Cloneable clone1 = c.clone();


		/* Cloning workaround that allows ownership context transfer
		 *
		 * First, the new object is created in a context owned by the client.
		 * Afterwards, all fields are copied from the original Cloneable instance.
		 */
		rep Cloneable clone2 = new rep Cloneable();
		clone2.copy(c);
	}

}

class Cloneable {
	int a;
	double b;

	public pure Cloneable() {}

	public pure Cloneable(int a, double b) {
		this.a = a;
		this.b = b;
	}

	public pure readonly Cloneable clone() {
		Cloneable c = new peer Cloneable(a, b);
		return c;
	}

	public void copy(readonly Cloneable c) {
		this.a = c.a;
		this.b = c.b;
	}
}
```

The clone problem and the proposition to use copy as a workaround instead has already been discussed in [Häc05].

So far, we managed to find an eligible workaround for ownership systems that do not support ownership transfer or allocation of objects in a method-caller's context. Unfortunately, we notice that the proposed approach cannot be applied to the prototype pattern in a straight-forward way as the client operates on abstract products. The client only has a reference of the abstract type `Prototype`, not knowing on what concrete products it operates. The prototype is instantiated outside the client's scope and can be changed at run-time by just registering a new prototype.

Due to the fact that the client is unaware of the concrete product type it cannot instantiate a new product and then call `copy()` on it, providing the prototype instance.

We will illustrate possible workarounds and solutions for this problem in the context of each reviewed system.

Universe type system

Having the prototype instance in a different context than the client leads to a readonly reference from the client to the prototype. Registering a new prototype at the client can be done by calling the `register(readonly Prototype)` method which sets the `readonly` prototype field in the client. In any operation where new products are needed, the given prototype is cloned. Refer to Figure 5.4 for an illustration.

Solutions

The copy approach combined with reflection Instead of cloning new products at the client, we will first create a new product instance of the same dynamic type as the current prototype. Then we call the `copy()` method on the newly created product to update all field values of the product to the ones currently set in the prototype instance. The dynamic type is determined using Java's reflection mechanism and the client code remains unaware of all concrete products by just operating on the abstract `Prototype` class. Like in [Häc05], we suggest to implement the `copy()` method in the following way:

1. `readonly` fields should be copied by reference
2. `rep` and `peer` fields should be copied recursively into the target universe. Hence, they need to implement the `copy()` method as well.

Introducing new universe modifiers As already proposed earlier, with the introduction of new universe modifiers that would allow us to transfer created objects between contexts or simply support the allocation of an object in a method's caller context would simplify the prototype pattern implementation. We could then implement a `clone()` method which is allowed to create the clone in the caller's context. With the proposed system enhancement, implementation of the prototype pattern would be trouble free.

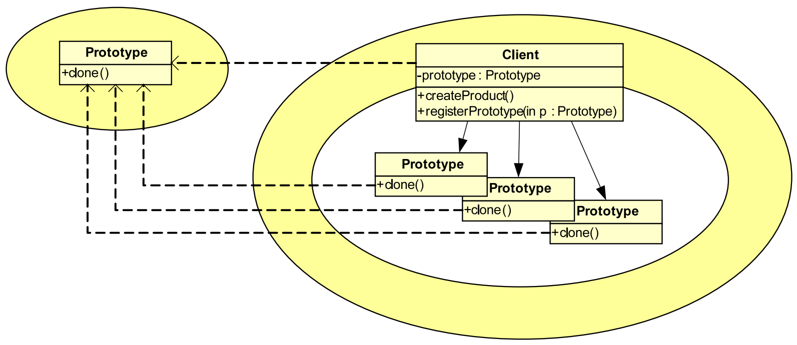


Figure 5.4: Universe type system structure of the prototype pattern. `readonly` references are dashed arrows.

Example Implementation

```java
class Client {
	readonly Prototype p;
	public void createProduct() {
		try {
			rep Prototype product;
			rep Class pClass = rep Class.forName(p.getClass().getName());
			product = (rep Prototype) pClass.newInstance();
				product.copy(p);

		} catch (ClassNotFoundException e0) {
		} catch (IllegalAccessException e1) {
		} catch (InstantiationException e2) {
		}
	}
}

abstract class Prototype {
	abstract void copy(readonly Prototype p);
}

class A extends Prototype {
	int a;

	void copy(readonly Prototype p) {
		if (p instanceof A) {
			readonly A orig = (readonly A) p;
			this.a = orig.a;
		}
	}
}

class B extends Prototype {
	int b;

	void copy(readonly Prototype p) {
		if (p instanceof B) {
			readonly B orig = (readonly B) p;
			this.b = orig.b;
		}
	}
}
```

Ownership Types

When trying to apply Ownership Types to the prototype pattern we immediately recognize that once again the original `clone()` approach does not work because the external `Prototype` object may not create instances in the context owned by `Client`. Therefore, we try to apply the previously proposed `copy()` methodology, i.e. the client instantiates a new product using reflection and then calls `copy()` on the new object, providing the prototype instance as parameter. This approach works fine for flat prototype ownership structures where the prototype either consists of only one object or of an object structure where all objects have the same owner. However, as soon as the prototype encapsulates some of its fields, cloning the prototype object(-structure) is not possible anymore since the client cannot read the prototype's fields in order to copy them. We recognize that when dealing with a deep prototype ownership structure, neither cloning nor copying is allowed and the prototype pattern cannot be implemented. Figure 5.5 illustrates this situation.

Solutions

In order to be able to implement the prototype pattern using Ownership Types we must ensure the following conditions:

- The prototype object(-structure) must be accessible by the client according to the object encapsulation rule.
- The prototype object(-structure) must have a flat ownership structure, i.e. all objects have the same owner.

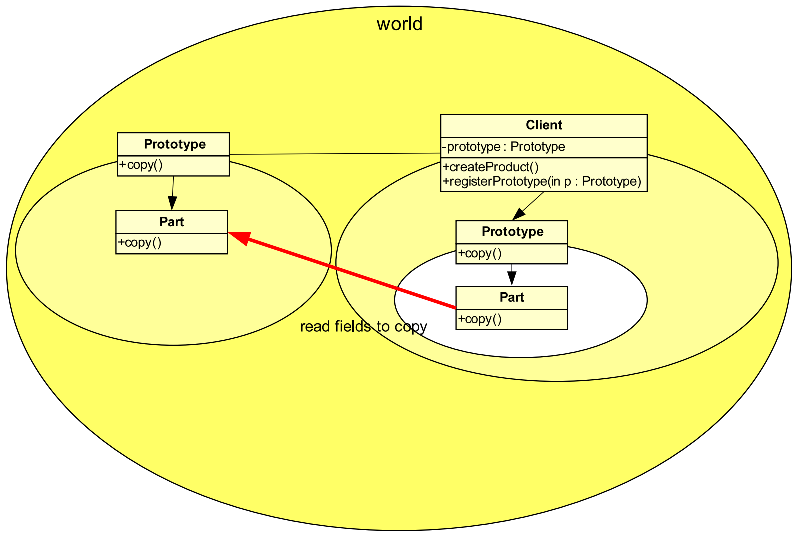


Figure 5.5: Ownership Types structure of the prototype pattern. A deep copy of the `Prototype` is not possible because the `Part` fields cannot be read from the context owned by `Product`.

Under these circumstances, we can implement the prototype pattern in the same way we proposed implementing it in conjunction with the Universe type system: first, the client uses reflection to create a new product. Second, on the created product the `copy()` method is called with the prototype instance as parameter in order to adopt all fields (recursively).

Introducing readonly references: A useful system extension to Ownership Types would be to support readonly-references between any context. This would enable us to also clone nested ownership structures with the copy-approach.

Example Implementation

```
class Client<clientOwner, prototypeOwner> {
       Prototype<prototypeOwner> prototype;

       public void operation() {
              //It is assumed that Java reflection is supported and accordingly annotated
              Class<this> pClass = Class.forName<this>(prototype.getClass().getName());
              Prototype<this> product = (Prototype<this>) pClass.newInstance<this>();
              product.copy<prototypeOwner>(prototype);

              //...
       }

       public void registerPrototype(Prototype<prototypeOwner> prototype) {
              this.prototype = prototype;
       }

       //The Client and Prototype can be instantiated like this:
       public void test() {
              Client<world, world> client = new Client<world, world>();
              Prototype<world> prototype = new ConcretePrototype<world>();

              client.registerPrototype(prototype);
       }
}

class Prototype<owner> {
        /* The Part field has the same owner as Prototype. The ownership
         * structure is therefore not nested and can be copied.
         */
        Part<owner> field;

        void copy<origOwner>(Prototype<origOwner> prototype)
                where (owner <= origOwner) {

                field = new Part<owner>();
                field.copy<origOwner>(prototype.field);
        }
}

class Part<owner> {
        //fields

        void copy<origOwner>(Part<origOwner> part)
                where (owner <= origOwner) {
                //copy fields...
        }
}
```

Ownership Domains

We propose defining two domains, one holding the prototype instance and the other holding all clients that generate products by cloning the prototype. Once again, we face cloning problems with ownership. Even when the prototype domain is linked to the client domain, we only gain access rights that do not include object creation rights. Therefore, we have to follow the copy approach proposed in previous discussions. When declaring the `prototypes` and `clients` domains we need to make sure that the clients have access right to the prototype. Hence, we establish a link from `clients` to `prototypes`. With the link in place, all clients can access the prototype instance in order to read its fields in the `copy()` method. Unfortunately, access rights guarantee the right to modify the prototype and we can not statically ensure that no client changes the original prototype instance. Having readonly links between domains would allow us to define even more secure architectures.

Solutions

Copy approach with reflection Like in the Universe type system, we can implement the prototype pattern while encapsulating the client's products by enabling the clients to access the prototype instance. Instead of creating new products by cloning the prototype, we create a new prototype instance with reflection and copy the prototype's fields into the product by calling `copy(Prototype p)` on the product.

A discussion of this approach is done best by differentiating several situations:

1. The prototype as a single object If we only need to copy the fields of a single object instead of a whole object structure, one just needs to ensure that the clone's context has access rights to the prototype instance in order to be able to read its fields in the `copy()` method.

2. The prototype as an object structure with only public domains If the prototype consists of an object structure which needs to be copied recursively, but all contained objects reside in public domains, access rights to the prototype instance automatically implies access rights to its representation and therefore no additional linking is required. A major flaw of this solution is that no alias control for the prototype's representation and the cloned products is in place. We can only control aliases of the prototype by restricting access rights to the `prototypes` domain and by allocating the clones in a private domain spanned by the client. Figure 5.6 illustrates this proposition.

3. The prototype as an object structure with private domains With the current system, it is not possible to clone a prototype structure that encapsulates its representation in private domains. The reason for this is that the client may not establish a link to the prototype's representation due to linking restrictions. Additionally, the prototype instance may also not link the client's (clone's) context with its representation since all links are declared statically but it is not known at compile time how many products will be cloned from the prototype. Hence, we need a workaround in order to clone a deep prototype ownership structure.

4. The prototype as an object structure with public domains, the clones as an object structure with private domains In order to be able to copy the prototype structure, all domains should be public. However, the clone's representation could still be encapsulated in private domains as it must not be accessed from outside. We therefore suggest to define a `Prototype` interface, and for each product, two classes that implement the interface. The first class (`PublicPrototype`) only spans public domains for its representation and can therefore be fully accessed from outside. The second class (`PrivatePrototype`) has exactly the same implementation as `PublicPrototype` with the only difference that all declared domains are private. The original prototype is an instance of `PublicPrototype` while each clone is of type `PrivatePrototype`. This setup ensures that the original prototype can be accessed by the clones. Moreover, the clones have full representation encapsulation.

The drawback of this approach is that always two versions of the prototype's implementation need to be synchronized. One can cope with this problem if the `PublicPrototype` only declares its fields respectively the object structure and only the `PrivatePrototype` implements the actual business logic. This leads to far less code duplication and the code gets more maintainable. Figure 5.7 illustrates this proposition.

Introducing different link types With the introduction of new link types, implementation of the prototype pattern could be even safer and more true to original. We propose the following link types:

- `readonly link` — enables objects in the source context to read objects in the destination context.
- `link` — enables objects in the source context to read and modify objects in the destination context.
- `creational link` — enables objects in the source context to read, modify and create objects in the destination context.

With the proposed links in place, the copy approach could be implemented in a safer way by just allowing the clients to *read* the prototype instance. Creational links would allow us to implement product-creation by *cloning* the prototype instance, as suggested in the original pattern. A detailed examination of all the implications to the system is subject of further research.

Ownership Diagram

Example Implementation

The following code shows an example implementation where the original prototype of a product (`PublicPrototype`) only spans public domains for its representation in order that the clients can access its structure to copy it. We achieve representation encapsulation for the products by using `PrivatePrototype` as the type of the products. The `prototypes` domain is passed as a class domain parameter to each new product (and its representation) to ensure that we can link each declared private domain on the product side to `prototypes`. Linking the product domain to `prototypes` lets the product access the prototype instance with all its sub-elements.

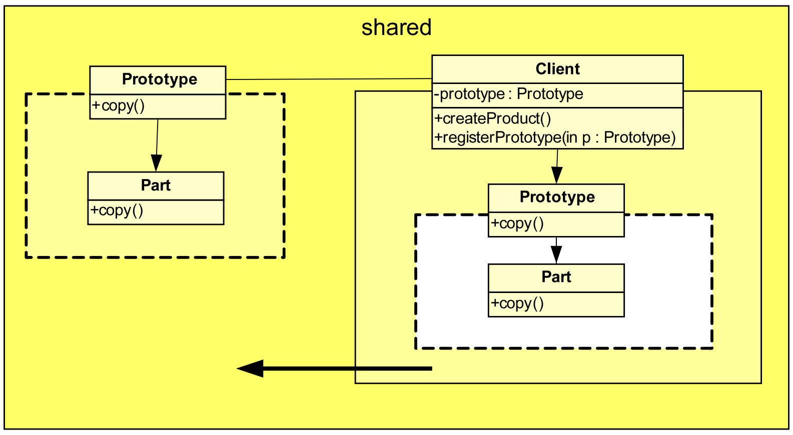


Figure 5.6: Illustration of an ownership typing where only the generated products are shielded from outside. The `Prototype` itself spans a public domain in order that its fields can be read from outside during the `copy()` method.

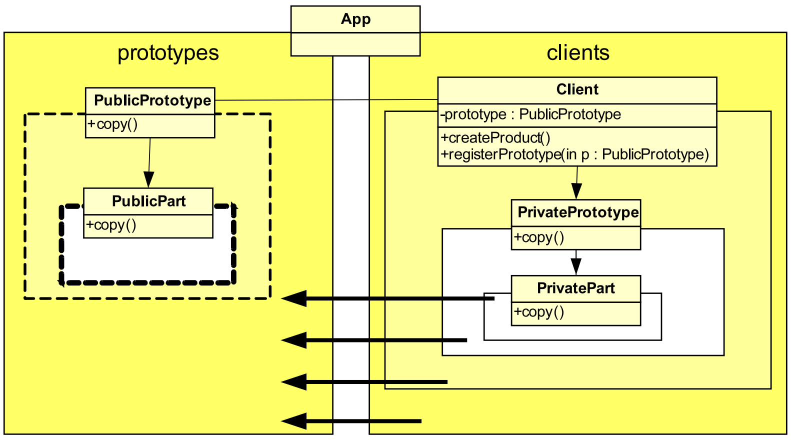


Figure 5.7: The `PublicPrototype` class has public domains in order that the clients have access to copy the structure. All products (clones of `PublicPrototype`) are of type `PrivatePrototype` which has private domains and therefore the representation is shielded.

```
class App {
	domain prototypes;
	domain clients;
	link clients -> prototypes;

	public void test() {
		prototypes Prototype<prototypes> prototype = new PublicPrototype<prototypes>();
		clients Client<prototypes> client = new Client<prototypes>();
		client.registerPrototype(prototype);
		client.operate();
	}
}

class Client<prototypes> assumes owner -> prototypes {
	domain owned;
	link owned -> prototypes;

	prototypes Prototype<prototypes> currentPrototype;

	public void registerPrototype(prototypes Prototype<prototypes> p) {
		currentPrototype = p;
	}

	public void operate() {
		String classname = currentPrototype.getClass().getName();

		//It is assumed that the used API has been annotated accordingly
		owned Prototype<prototypes> product =
			(owned Prototype<prototypes>) Class.
				forName(classname).newInstance<owned, prototypes>();

		product.copy(currentPrototype);
	}
}

abstract class Prototype<prototypes> {
	public abstract void copy(prototypes Prototype<prototypes> p);
}


class PublicPrototype<prototypes> extends Prototype<prototypes> {
	public domain rep;

	rep Part<prototypes> part;

	public void copy(prototypes Prototype<prototypes> p) {
		final prototypes PublicPrototype<prototypes> pp = (PublicPrototype<prototypes>) p;
		part = new PublicPart<prototypes>();
		part.copy<pp.rep>(pp.part);
	}
}

class PrivatePrototype<prototypes> extends Prototype<prototypes> assumes owner -> prototypes {
	domain rep;
	link rep -> prototypes;

	rep Part<prototypes> part;

	public void copy(prototypes Prototype<prototypes> p) {
		/* The original Prototype instance is of type PublicPrototype so
		 * that its representation can be accessed in the copy method.
		 */
		final prototypes PublicPrototype<prototypes> pp = (PublicPrototype<prototypes>) p;
		part = new PrivatePart<prototypes>();
		part.copy<pp.rep>(pp.part);
	}
}

interface Part<prototypes> {
       public void copy<origDomain>(final origDomain Part<prototypes> p);
}

class PublicPart<prototypes> implements Part<prototypes> {
       public domain rep;

       public void copy<origDomain>(final origDomain Part<prototypes> p) {
              //copy the fields
       }
}

class PrivatePart<prototypes> implements Part<prototypes> assumes owner -> prototypes {
       domain rep;
       link rep -> prototypes;

       public void copy<origDomain>(final origDomain Part<prototypes> p) {
              //copy the fields
       }
}
```

Conclusion

Once again we have to cope with the problem that, under the reviewed ownership systems, objects outside a context cannot create objects within that context (unless they are the owner of the context). Nevertheless, by using a copy mechanism instead of the originally proposed cloning we can decently handle this problem. A not very elegant aspect of our solution is that we need to make use of Java's reflection capabilities in the client code in order to get a new prototype object for which the dynamic type is not known at compile-time. While comparing the different ownership type systems, we observe that the Universe type system offers the safest implementation as references to the prototype are readonly. Thus, we can ensure that the prototype instance cannot be modified while creating new products. The Ownership Types system is too restrictive concerning alias rights. We neither have the possibility of readonly references through context boundaries nor can we establish access rights between different contexts. An implementation using Ownership Domains is possible with the copy approach, but we need some major workarounds to copy an encapsulated object structure.

5.4 Builder

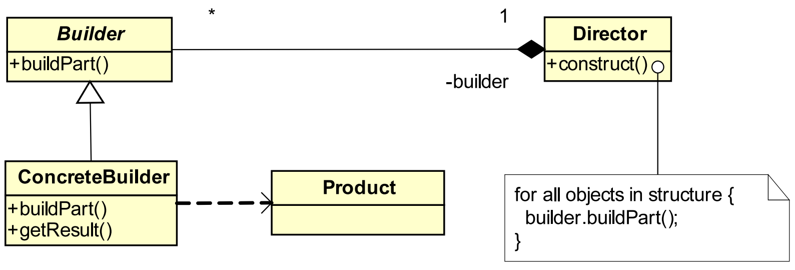
Intent

> ”Separate the construction of a complex object from its representation so that the same construction process can create different representations.”[4]

While the abstract factory (5.1) focuses on building families of products, the builder is responsible for creating complex objects. Hereby, the construction is separated from the representation: the director has knowledge of the construction procedure, but delegates the actual instantiation of the representation to the configured concrete builder. As the director only operates on an abstract `Builder` interface it does not know about the underlying concrete builder and thus the product's representation can easily be exchanged by configuring the director with a new concrete builder. New representations for a given abstraction can easily be added by further subclassing of the abstract `Builder` class.

[4][GHJV95] page 97

UML Diagram



Ownership Discussion

The main obligation of the builder pattern is to create a product for the client. Thus, there are no differences to other creational patterns like the abstract factory (5.1), factory method (5.2), or prototype (5.3). We notice that this leads to the same desired ownership structure as already discussed in the abstract factory discussion: it would be very beneficial if the client could be the owner of the manufactured object in order to be able to safely establish invariants over the product's state.

Reviewing the builder pattern's interaction diagram in Figure 5.8 we notice the following:

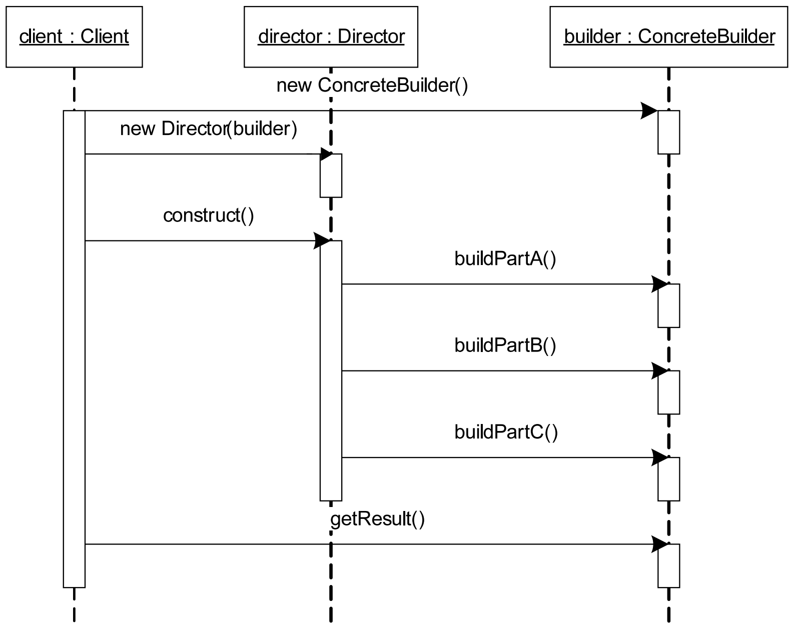


Figure 5.8: Builder Pattern Interaction Diagram

1. The client creates a director.
2. The client creates a concrete builder and configures the director with it.
3. The client requests a new product by calling `construct()` on the director.
4. The director builds the product according to the construction process by calling `build-PartX()` on the builder object. The whole product gets hereby assembled in the builder's scope but under the supervision of the director.
5. The client requests the finished product from the builder.

The interaction has several consequences on the possible ownership structures: Read/write access rights must be established between the director and the builder to trigger the manufacture process. After the manufacturing has finished, the client must be able to access the product and, desirably, also be the owner of the product. A good ownership scenario is therefore to declare the client as the owner of the builder, the director, and all created products (Figure 5.9). If the application scenario requires a sharing of the director and its configured builder, the used ownership system needs to provide support for shared contexts in order to allow all clients to access the shared director. Such a scenario additionally requires a mechanism for ownership transfer in order to transfer ownership of the created product to the client after the build process has finished.

Universe type system

The given interaction and the desired ownership structure impose that the director and the builder reside in the same context. In order to trigger the product's construction, the client must either be in the same context, or the owner of the director and the builder. Considering that the client should have normal read/write access rights to the created product, we recognize that two possible ownership structures exist:

Client, director, builder, and product are peer This is the same situation as when no ownership typing is in place at all and also does not correspond to the desired ownership structure. We will thus skip a further discussion of this ownership structure.

Client is the direct owner of director, builder, and product This is an interesting structure because it corresponds to the desired ownership structure on the one hand and fulfills all access requirements of the pattern on the other hand. The product is created by the builder as `peer` and thus resides in the client's context while the director is free to alias the configured concrete builder. Another desired benefit of such an encapsulation is that all objects that are involved in the pattern's implementation are located in the client's context and therefore safe from modifications from outside. Due to the encapsulated director and builder, sharing these objects between multiple clients is not possible anymore. However, in contrast to the abstract factory, which is likely to be implemented as singleton (5.5), the builder is usually created together with the director by each client separately. The main aspect of the builder pattern is to outsource the manufacture logic from the client and, in addition, separate the abstraction from the internal representation. Hence, the Universe type system is able to provide a feasible solution to type our desired ownership structure, assumed that neither directors nor builders need to be shared.

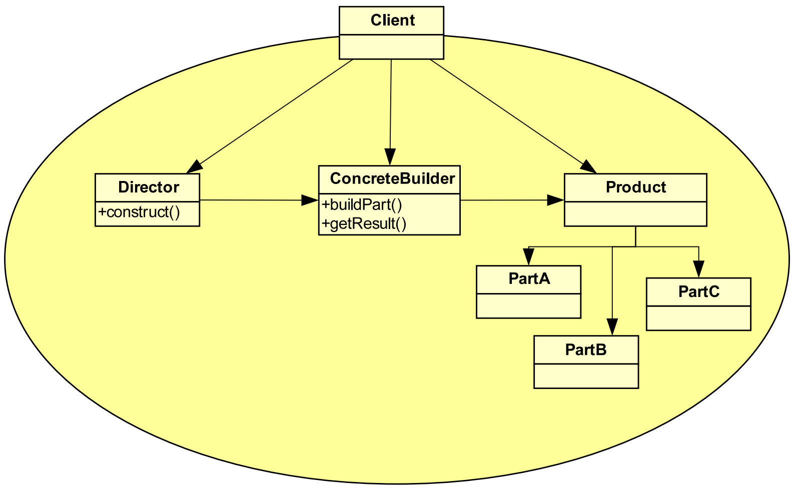


Figure 5.9: Illustration of the desired ownership structure of the builder pattern.

Example Implementation

An example implementation with the desired ownership typing is omitted. The client allocates the director and the needed concrete builder as `rep` and calls `construct()` on the director. The director then creates step by step the desired product by calling `createPartX()` on the builder which allocates the specified part as `peer`, also in the client's context. After the construction process ended, the client can safely retrieve the desired product by calling `getResult()` on the builder, returning the product as `rep`.

Ownership Types

Investigating feasibility of the desired ownership structure with Ownership Types leads to the same results as with the Universe type system: the client should be declared as the owner of the created product, the director, and the builder instance, resulting in the same, already discussed, benefits.

Example Implementation

An implementation of the desired ownership structure is trivial: the client just passes itself (`this`) to the director and the builder as the owner, upon creation. Products are also created with client as owner and the builder does therefore not require additional ownership parameters.

Ownership Domains

An implementation of the builder pattern with the desired ownership structure using Ownership Domains is also well supported. We can apply the same ownership structure as discussed before: the client simply declares a private context holding the director, the concrete builder and the generated products, which trivially leads to the same benefits and drawbacks already mentioned.

Moreover, due to the link concept the director and the concrete builder do not necessarily have to be in the same ownership context. We only have to ensure that the director can freely alias the builder to create parts of the complete product and that the client can freely alias the director to trigger the building process. Only the builder still needs to be located in the client's context. This enables us to share a director between multiple clients. However, each director can only be configured with one builder and thus sharing only the director has few benefits while sharing the director with a concrete builder is still not possible.

Thus, we suggest using the previously discussed ownership structure where the client encapsulates all objects involved in the pattern.

Example Implementation

When implementing the proposed ownership structure the client simply declares a private context `products` to hold the director, builder, and product instances.

Conclusion

We conclude that the concept of ownership helps us providing a correct pattern implementation. First of all, with the client encapsulating all involved objects we can reason about the pattern's correctness by only considering the client's ownership context since we know that modifications or even references from outside are not possible anymore. Second, with the client encapsulating all manufactured products, it can safely establish invariants over the product's state, enabling the definition of software contracts between involved objects. The only drawback of not being able to share the director and the configured builder anymore is negligible as usually a builder construct belongs to a client and is not implemented as a singleton.

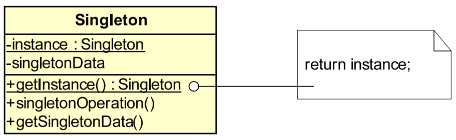
Comparing the three ownership systems reveals no further differences. All ownership systems achieve to model the desired ownership structure without problems.

5.5 Singleton

Intent

"Ensure a class only has one instance, and provide a global point of access to it."[5]

UML Diagram



Ownership Discussion

When typing the singleton pattern we have to make sure that the single instance of the class is globally accessible. The aim in this case is not to provide an ownership typing as strict as possible. We need to try to make one instance write accessible by potentially every object. The used ownership system therefore needs to provide support for a *global* or *shared context*.

Universe type system

When trying to provide an ownership typing using the Universe type system we recognize that the system is missing a global context which is read/write accessible by everyone. Using a readonly reference, an object can potentially reference any other object, but only pure methods can be invoked on that object.

Solutions

No explicit owner, read rights for everyone If all users of the singleton instance only need read access rights, ownership typing is straight forward:

```
class Singleton {
	private static readonly Singleton instance = null;

	/* In order to guarantee that only one instance of the class exists,
	 * the constructor has to be private.
	 */
	private Singleton() {}

	public static readonly Singleton getInstance() {
		if(instance==null) instance = new peer Singleton();

		return instance;
	}
}
```

With this implementation, all users will have a `readonly` reference to the singleton instance. The first caller of `getInstance()` determines the context in which the singleton instance will be located. Unfortunately, the first call to allocate the instance is not done explicitly and therefore nobody knows the singleton's context. If we would know the singleton's context, at least the owner and all `peer` objects would have access rights to the singleton. Refer to Figure 5.10 for an illustration.

Explicit owner, all others have read rights Following the proposition from [Häc05] we could also implement the pattern as follows:

[5] [GHJV95] page 127

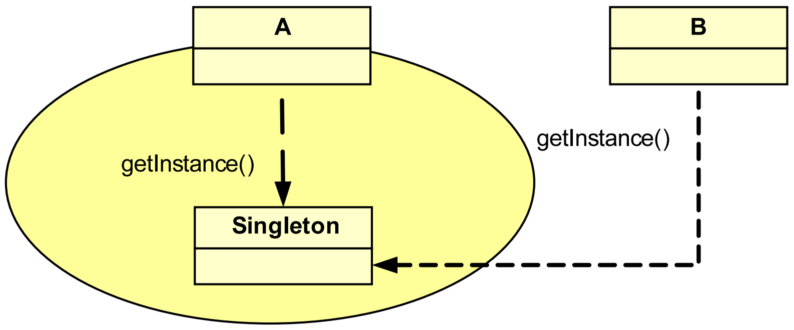


Figure 5.10: No caller is the explicit owner of the singleton. The singleton instance gets allocated during the first `getInstance()` call and each client has a `readonly` reference to the singleton.

```
class Singleton {
	private static readonly Singleton instance = null;

	private Singleton() {}

	public static pure readonly Singleton getInstance() {
		return instance;
	}
	public static peer Singleton initialize() {
		if (instance != null) {
			throw new RuntimeException("The singleton has already been initialized!");
		}
		instance = new peer Singleton();
		return (peer Singleton) instance;
	}
}
```

With the above implementation of the singleton pattern, we have an explicit owner or peer object of the singleton (the object who calls the `initialize()` method). All other objects can still have a readonly reference to the singleton instance. Refer to Figure 5.11 for an illustration.

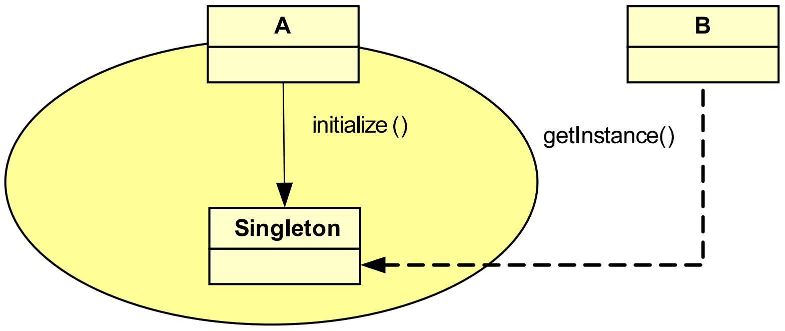


Figure 5.11: The first caller needs to call `rep Singleton.initialize()` in order to get a `rep` or `peer` reference to the singleton. All other clients may have a `readonly` reference to the singleton instance.

When multiple different objects in different universes need read/write access to the singleton instance we need to extend the existing system in the following way:

Introduction of a global read/write space It might be helpful to extend the Universe type system to support a global space to which every object, no matter in what context it resides, has read/write rights. The idea of a global context was primary introduced in [Häc04]. Following this approach we will introduce a new ownership modifier with the keyword *global*. All objects created with the `global` modifier are located in a global context. A `global` reference guarantees read and write rights to the referenced object. Therefore, `global` is also a subtype of `readonly`. The extended type combinator table with the new `global` type is presented in Figure 5.12.

*	**peer**	**rep**	**readonly**	**global**
peer	peer	readonly	readonly	global
rep	rep	readonly	readonly	global
readonly	readonly	readonly	readonly	readonly
global	global	readonly	readonly	global

Figure 5.12: Universe Type System Extended Type Combinator

Using the `global` ownership modifier, the new singleton implementation looks like this:

```
class Singleton {
      private static global Singleton instance;

      private Singleton() {}

      public static global Singleton getInstance() {
            if(instance == null {
                  instance = new global Singleton();
            }
            return instance;
      }
}
```

With the proposed system extensions, ownership typing of the singleton pattern is problem free and we can comply with the pattern's main intent: ensure a globally accessible single instance of a class. A detailed examination of all the implications to the system is subject of further research.

Ownership Types

Due to the notion of a global context, called `world`, the implementation of a singleton with Ownership Types is trivial and an illustration can be found in Figure 5.13.

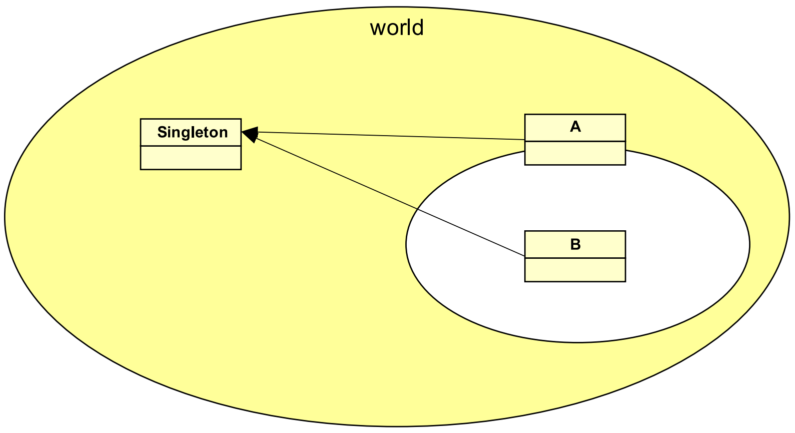


Figure 5.13: The `Singleton` instance is located in the global context `world` and therefore all objects may access the `Singleton`.

Solution

A singleton implementation with Ownership Types follows:

```
class Singleton<singletonOwner> {
      private static Singleton<world> instance;

      private Singleton<singletonOwner>{}

      public static Singleton<world> getInstance() {
            if(instance == null) {
                  instance = new Singleton<world>();
            }

            return instance;
      }
}
```

Ownership Domains

Using Ownership Domains, an implementation of the singleton pattern is straight-foreward. A simple way to implement the pattern is to allocate the singleton instance in the `shared` domain and establish access rights from each context holding singleton clients to `shared`. Figure 5.14 illustrates our proposition.

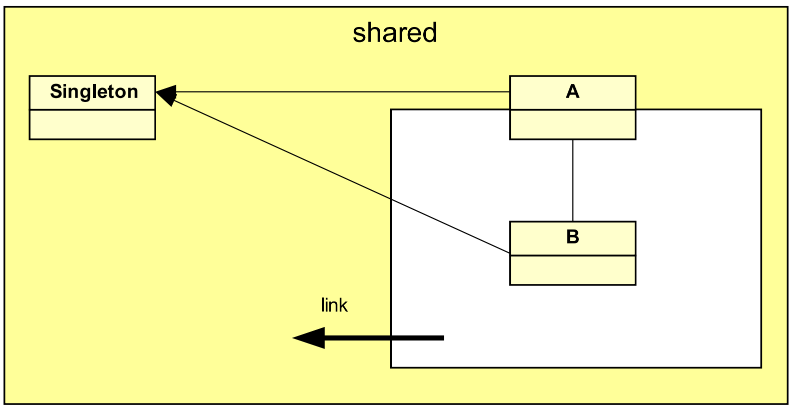


Figure 5.14: In order to allow object `B` to access the singleton, object `A` needs to link its context to the `shared` domain.

Solution

```
class Singleton {
      private static shared Singleton instance;

      private Singleton() {}

      public static shared Singleton getInstance() {
            if(instance == null) {
                  instance = new Singleton();
            }
            return instance;
      }
}
```

Conclusion

An implementation of the singleton pattern requires support for global contexts. As in Ownership Types each object can automatically access all global objects, implementing the singleton pattern poses no problems. Ownership Domains also provides support for a global context called `shared`, but we have to link all contexts containing singleton-clients to the `shared` domain manually. As long as having only readonly references to the singleton instance (or only one object needs read/write access) is enough, a typing with the Universe type system is feasible. The situation where the singleton has multiple clients (in different contexts) and more than one of them needs read/write access to the singleton is not supported. In order to support that situation we would need to extend the system as proposed.

6 Structural Patterns

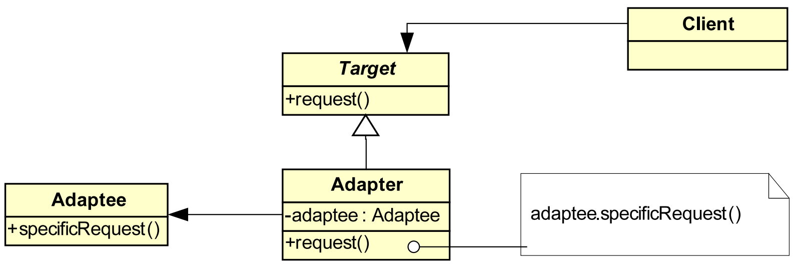
6.1 Adapter

Intent

> "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."[1]

The adapter basically converts calls from the client interface to the adaptee interface. There are two versions of an adapter structure, the *class adapter* adapts one interface to another by using multiple inheritance (`Adapter` inherits from `Target` and `Adaptee`) and the *object adapter* makes use of object composition (`Adapter` has a reference to `Adaptee`). Due to the fact that Java does not support multiple inheritance, we will only review the object adapter in our discussion.

UML Diagram



Ownership Discussion

In order to evaluate if the pattern's implementation benefits from an ownership structure that propagates the adapter as the owner of the adaptee we have to distinguish the following two cases:

1. The *simple adapter* is just responsible for mapping incompatible interfaces by simply adjusting and forwarding all calls to the adaptee.

2. The *smart adapter* does not only forward calls but may also cache certain properties of the adaptee or provide an access counting mechanism and features alike.

[1] [GHJV95] page 139

In case of a simple adapter, applying ownership makes only limited sense as there is no need to shield clients from the adaptee and the adapter instance can therefore be located in the same context as the adaptee. This results in the same situation as when no ownership is in place at all.

In case of a smart adapter, declaring the adapter as the owner of the adaptee is reasonable. The adaptee is shielded from being aliased by objects outside the adapter's context and we gain certainty that possible cached properties of the adaptee are not modified without the adapter noticing. The adapter can safely establish assumptions or invariants over the adaptee's state. If the adaptee is created before the smart adapter instance, the used ownership system needs to provide ownership transfer so that the adapter can take over ownership of the adaptee. It is also to mention that an adaptee can only be encapsulated by the adapter if all clients use the adapter. When some clients access the adaptee directly, appropriate aliasing rights have to be ensured or the adapter must not encapsulate the adaptee.

Because applying ownership to the adapter pattern only makes limited sense or is trivial, we will skip a detailed discussion for each ownership system.

By using the adapter pattern in conjunction with Ownership Types or Ownership Domains, one can make encapsulated objects accessible by providing an interface through a global adapter. In case of Ownership Types we could implement the Adapter as an inner class of the adaptee and specify a globally accessible interface for the encapsulated object. By allocating the adapter in the `world` context, all objects can gain access to the adaptee by using the defined interface. In case of Ownership Domains, we can declare a domain `adapters` holding the adapter and then link `adapters` with the private domain holding the adaptee. Linking all contexts in which clients reside with the `adapters` domain provides access rights for clients to the adaptee by using the adapter.

Conclusion

Applying ownership to the adapter pattern is trivial, and depending on the adapter's implementation, not always meaningful because it makes limited sense to shield the client from the adaptee when they have incompatible interfaces anyway. In case of a smart adapter, ownership can still help to make caching or access counting mechanism safer.

6.2 Bridge

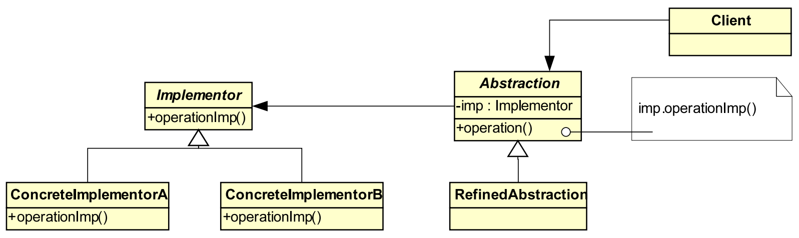
Intent

> "Decouple an abstraction from its implementation so that the two can vary independently."[2]

Usually, when an abstraction has several implementations, they are accommodated using inheritance. The problem is that the abstraction may also be subsequently redefined by subclassing. Accommodating the abstractions and implementations in one class hierarchy leads to a large and complicated structure and it will become difficult to modify, extend, and reuse the abstractions and implementations independently. The bridge pattern thus separates the abstraction hierarchy from the implementation hierarchy and combines the two hierarchies by composition.

[2][GHJV95] page 151

UML Diagram



Ownership Discussion

When implementing the bridge pattern, we suggest using as much object encapsulation as possible. This means that, if possible, the abstraction should be allocated in a context owned by the client in order to enable local reasoning. Furthermore, the implementation of an abstraction should be completely hidden from the client. The client must not be dependent on a concrete implementation anyway. We can enforce this property by using the concept of ownership: if we declare the abstraction as the owner of the underlying implementation we shield the implementation from the client. With the suggested ownership structure we can statically guarantee that the client only operates on a certain abstraction, not worrying about implementation details.

The decision which concrete implementation is used can be delegated to an abstract factory. For the sake of simplicity, it is assumed that the used implementation is specified in the abstraction's constructor or simply defaulted and maybe switched later by calling a method on the abstraction.

Universe type system

Typing the suggested ownership scenario with the Universe type system is trivial. If a single client encapsulates the abstraction, `Client` has a `rep` reference to `Abstraction`. Additionally, `Abstraction` has a `rep` reference to the used `Implementor`, as shown in Figure 6.1.

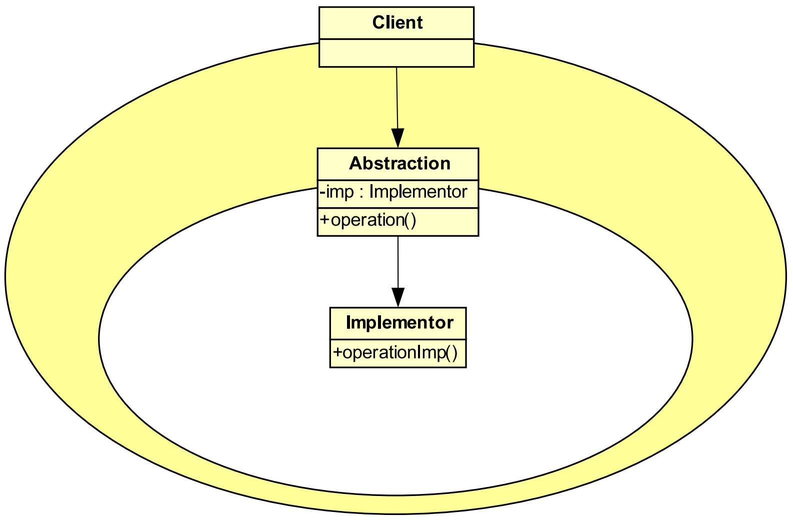


Figure 6.1: An implementation of the bridge pattern can benefit most from a deep ownership structure. The client encapsulates the abstraction in order to safely establish invariants over the abstraction's state. Each abstraction owns its implementation, hiding it from the client and ensuring total independence.

Example Implementation

An example where the abstraction refines windows of a window system, available for different platforms, follows.

```
class Client {

       //The client encapsulates the used abstraction in its context.
       rep Dialog dialog;

       public void test() {
              dialog = new rep Dialog(true);
              dialog.drawButton();
       }
}

abstract class Window {

       //The abstraction encapsulates the used implementation in its context.
       rep WindowImp imp;

       public Window(boolean mac) {
              if(mac)
                     imp = new rep MacWindowImp();
              else
                     //...
       }

       public void drawRect() {
              imp.drawLine();
              //...
       }
}

class Dialog extends Window {

       public Dialog(boolean mac) {
              super(mac);
       }

       public void drawButton() {
              drawRect();
              //...
       }
}

abstract class WindowImp {
       public abstract void drawLine();
}

class MacWindowImp extends WindowImp {

       public void drawLine() {
              //draw a line using the mac graphics lib
       }
}
```

Ownership Types

Typing the desired ownership structure with Ownership Types poses no problems. Once again, we would like to encapsulate the abstraction in the context of the client while the implementation is encapsulated in the abstraction's context. This can easily be done by passing both times `this` as the ownership parameter when the clients instantiates the abstraction, respectively the abstraction instantiates the concrete implementation.

Ownership Domains

Implementation of the bridge pattern using Ownership Domains follows the same approach as the other two ownership systems. The abstraction is kept in a private domain `abstractions` of `Client` and the implementation is allocated in a private domain `implementation`, declared by the abstraction.

Conclusion

By using ownership in conjunction with the bridge pattern some of its key properties can be emphasized. If the client has the possibility to own the abstraction, it can safely establish invariants over it. Ownership allows us to perfectly hide the implementation from the client and an ownership typing under all three systems shows no conceptual differences.

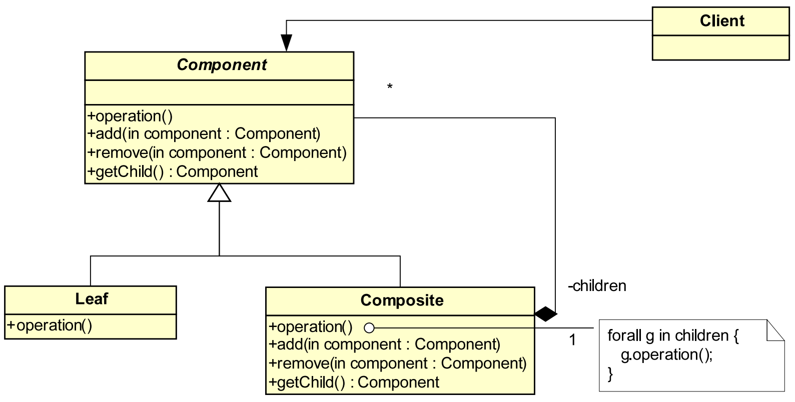
6.3 Composite

Intent

> ”Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.”[3]

The composite pattern is useful in situations where one has to deal with part-whole hierarchies in which each element should be treated the same. The client should not care if it is operating on a composition or an individual object. Therefore, the client only manipulates the structure through a component interface which is implemented by both, the primitive and the container object.

UML Diagram



Ownership Discussion

An investigation of the composite pattern regarding ownership quickly leads to the idea that each composite could encapsulate its elements. We would like to perform a detailed discussion, pointing out assets and drawbacks of the proposed ownership structure.

Whether a composite should be the owner of its components and restrict other elements of the data structure or even the client from accessing them, greatly depends on the desired implementation aspects.

When the composite holds invariants over its children With component encapsulation in place, the composite can safely establish invariants over its elements. This might be especially

[3][GHJV95] page 163

useful, and will lead to a safer implementation of the pattern, when the data structure is double linked. When each child maintains a reference to its parent, the data structure must fulfill the invariant that all children have the composite as their parent, which in turn has them as children. Thanks to an encapsulation of the children we can statically ensure that no object outside the composite's context can manipulate the children and probably invalidate our invariant. Depending on the application, one could think of many other invariants that the composite may define over the state of its children. Hence, ownership helps in these situations to make the implementation of the pattern safer.

When the composite has a cache Some applications require fast data structures. Thus, an implementation of the pattern makes use of caching mechanisms in order to speed up data access. One such scenario would be a graphic application where a graphical object (the composite) can consist out of several different other graphical objects and simple primitives (lines, etc). In this case it would be very useful if the composite always caches its current bounding box. By encapsulating the children, we could statically ensure that no objects outside the composite's context could modify the children and invalidate the cache without notifying the composite first.

When components need to be shared Sometimes, the application needs to share components in the data structure, for example to reduce storage requirements. In a shared component scenario object encapsulation does not work anymore because a component cannot have more than one parent and therefore also more than one owner. Each parent needs to be the owner of its children so that full access rights (including creational rights) to the children exist. If the used ownership system supports shared contexts, one might still have the possibility to enable sharing of components *and* have alias control in place.

When components might move in the data structure Dynamic data structures where components might move (i.e. in a file system where files and folders can be moved) together with a deep ownership structure require support for ownership transfer. As ownership transfer is not supported in the reviewed systems, an encapsulated composite structure needs to be static.

When already existing components need to be inserted In most scenarios, components are created by the client and added to the data structure later on. In a deep ownership scenario, this means that the components change their ownership context during life-time and the used ownership system thus has to support ownership transfer. Due to the fact that none of the three type systems supports this feature, an ownership typing of this scenario fails.

If, due to the application scenario, an encapsulation of the children of a composite is too restrictive, one can easily encapsulate the whole data structure in the client's context. Elements in the data structure can still freely reference each other but no other objects than the client or the components can manipulate the data. Thus, there is still a great benefit in using ownership. For most application scenarios, a deep ownership structure is likely to be too restrictive and an encapsulation of the whole composite structure will be the best choice.

When the composite pattern is implemented with a deep ownership structure and there is no support for ownership transfer, the `Component`'s and `Composite`'s interfaces need to be adjusted. The composite interface needs to act like a facade (6.5), offering methods for all child manipulations. This is because the children may not be freely aliased anymore and passing a read/write reference to the client is restricted. Also, the creation of a new child needs to be delegated to the composite and cannot be done by the client and passed to the composite by calling `add(Component c)`.

Another problem arises when deleting a child. Depending on the used ownership system, the encapsulated objects may not be referenced at all and therefore `delete(Component c)` needs to be changed too. One possible workaround for this problem is to introduce value type identifiers for all elements in the data structure so that the client can trigger a deletion of a child by passing the identifier instead of the object reference to the composite. Depending on the application scenario, other workarounds may suit.

Universe type system

Implementation of the composite pattern with the Universe type system is feasible. The `Composite` simply maintains `rep` references to its children and the Composite's interface has to be adjusted in the following way:

- `readonly Component add(readonly Component parent, int type)` — The composite is responsible for adding a new child. A `readonly` reference to the newly created child may be passed back to the caller. The parent has to be supplied with a `readonly` reference because `add()` is a non-pure method and cannot be called directly on a component in the structure. Moreover, a value type identifier, specifying the concrete component to be instantiated, needs to be provided.
- `void remove(readonly Component c)` — The `remove()` signature stays the same as in the original pattern proposition but the component to remove can only be passed as a `readonly` reference which can be downcasted to `rep` by the composite.
- `readonly Component getChild(int i)` — The composite may still return references to its children but they must be `readonly`.
- All child manipulations must be offered in the `Composite`'s signature because `Composite` must act as a facade for its children. This means that a client has to call the child update method on the root component, providing the object to update as `readonly`. The operation will then be forwarded to the component's parent who then can downcast the `readonly` reference to `rep` and perform the update.

Thanks to the concept of `readonly` references, encapsulation of the composite's children can be done with minor changes to the composite's interface, as illustrated in Figure 6.2.

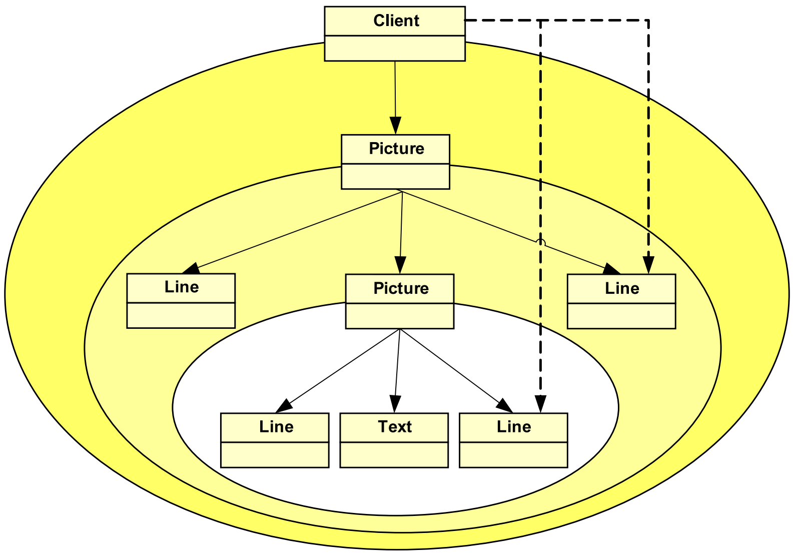


Figure 6.2: Universe type system: example of a composite pattern in a graphics application. `rep` references have solid lines, `readonly` references dashed lines. The client is the owner of the data structure and each composite owns its elements. The client is still able to access a composite's children but only through `readonly` references.

Example Implementation

We would like to illustrate ownership typing on the example of a graphics application where a picture is a composite that may contain components like another picture, a line, or a textbox. For fast responses to an intersect method that checks if two elements overlap, each component caches its bounding box. Using ownership, we can be sure that no objects inside the data structure can be modified without notifying the parent first. Hence, situations where a parent stores invalid bounding box values because of directly updated children may not occur anymore.

```java
import java.util.ArrayList;

class Picture extends DesignElement {
	private rep ArrayList elements;

	public Picture() {
		super();
		elements = new rep ArrayList();
	}

	public readonly DesignElement add(readonly DesignElement parent, int type) {

		rep DesignElement child = null;
		if(parent == this) {
			switch(type) {
				case DesignElement.LINE:
					child = new rep Line();
					break;

				case DesignElement.TEXTBOX:
					child = new rep Textbox();

				case DesignElement.PICTURE:
					child = new rep Picture();
					break;

				default:
					throw new RuntimeException("specified type not supported!");
			}
			elements.add(child);
			//adjust bounding box!
			return child;
		}
		else {
			for(int i=0; i < elements.size(); i++) {
				child = (rep DesignElement) elements.get(i);
				readonly DesignElement addedChild = child.add(parent, type);
				if(addedChild != null) {
					//adjust bounding box!
					return addedChild;
				}
			}
			return null;
		}
	}

	public boolean remove(readonly DesignElement element) {
		boolean removed = elements.remove(element);

		if(removed) {
			//adjust bounding box!
			return true;
		}
		else {
			//element to remove is not a direct child of this
			for(int i=0; i < elements.size(); i++) {
				rep DesignElement child = (rep DesignElement)elements.get(i);
				boolean removedAtChild = child.remove(element);
```

```
                                if(removedAtChild) {
                                        //adjust bounding box!
                                        return true;
                                }
                        }
                        return false;
                }
        }

        public pure readonly DesignElement getElement(int i) {
                rep DesignElement child = (rep DesignElement) elements.get(i);
                return child;
        }

        public void resize(readonly DesignElement element, double height, double width) {
                /* adjust bounding box and propagate operation to subtree
                 * where element is located if (element != this)
                 */
        }

        public void move(readonly DesignElement element, double horizontal, double vertical) {
                /* adjust bounding box and propagate operation to subtree
                 * where element is located if (element != this)
                 */
        }
}

abstract class DesignElement {
        final static int LINE = 0;
        final static int TEXTBOX = 2;
        final static int PICTURE = 3;

        //Each element caches its boundingbox (including all children).
        private rep BoundingBox bb;

        public DesignElement() {
                bb = new rep BoundingBox();
                //initialize bounding box
        }

        public pure boolean intersects(readonly DesignElement element) {
                /* Without ownership, each DesignElement could easily manipulate
                 * another element's bounding box!
                 *
                 * Check intersection with other element, if it does:
                 */
                return true;
        }

        public readonly DesignElement add(readonly DesignElement parent, int type) {
                //default implementation:
                return null;
        }

        public boolean remove(readonly DesignElement c) {
                //default implementation:
                return false;
        }

        public pure readonly DesignElement getElement(int i) {
                //default implementation:
                return null;
        }

        public void resize(readonly DesignElement element, double height, double width) {
                //adjust bounding box if (element == this)
        }
        public void move(readonly DesignElement element, double horizontal, double vertical) {
                //adjust bounding box if (element == this)
        }
}

class Line extends DesignElement {} // represents a leaf component

class Textbox extends DesignElement {} // represents a leaf component

class BoundingBox {
        public double x;
        public double y;
        public double width;
        public double height;
}
```

The example shows that adding a new `DesignElement` is rather artificial and an ownership transfer concept could greatly simplify the implementation. Furthermore, operations on the data structure are very inefficient. They are triggered on the root node and then forwarded to all elements in the tree until the correct element is found.

Ownership Types

With Ownership Types the desired encapsulation can also be easily achieved. The problem we have to cope with is that unlike in the Universe type system, no readonly references between context boundaries are allowed. This has the consequence that all child operations need different signatures:

- `int add(int parent, int type)` must specify the object type to create and the parent at which the child will be inserted. Optionally, the id of the created object can be returned.
- `void remove(int componentID)` must be called with an id as value type instead of passing the component to remove by reference.
- `int getChild(int parentID, int ith)` can only return the child's id instead of a reference to the child.

Due to the absence of readonly references the client cannot have a reference to a component in the data structure and all signatures in the `Composite` class dealing with child operations need to be changed. The main flaw of the introduction of value-type identifiers for the elements is that we have to reinvent references and ensure their uniqueness. Hence, the introduction of readonly references would greatly simplify an implementation. If it is not sufficient for the client to obtain object IDs and the application scenario requires it to have references to components inside the structure, deep ownership typing with Ownership Types fails. The proposed ownership structure is outlined in Figure 6.3.

Ownership Domains

Achieving the desired object encapsulation is straight-forward when using Ownership Domains: the composite simply declares a private domain, holding all children. Optionally, the client can also declare the whole data structure in a private domain, if it is the only accessor. This ownership structure has the same benefits and flaws as with Ownership Types: by encapsulating the children of a composite we restrict the client from aliasing them (not even using readonly reference). All signatures of the child manipulation methods must be adapted accordingly. Figure 6.4 illustrates the proposition.

When declaring the element's `children` domain as `public`, access rights to the parent would imply access rights to the children. We assume that these access rights are transitive, which results in the same ownership scenario as when having a flat ownership structure where all components are owned by the client.

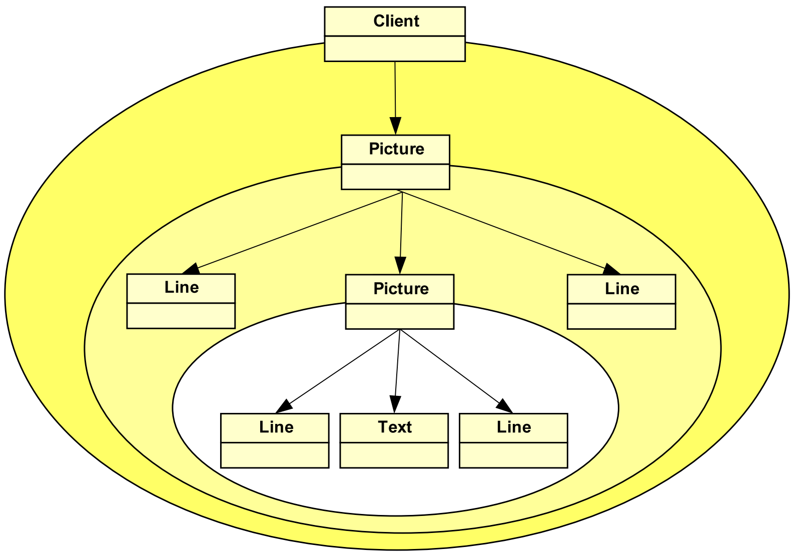


Figure 6.3: Ownership Types structure of the composite pattern. In contrast to the Universe type system, the client may not alias objects inside the structure.

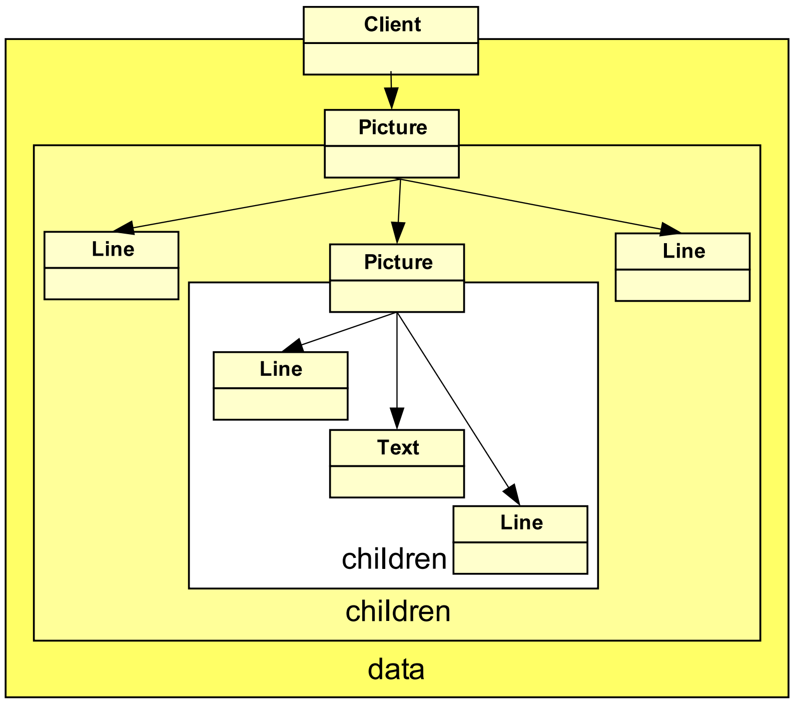


Figure 6.4: Illustration of the ownership structure with Ownership Domains. All declared domains are private so that the composite can safely establish invariants over the state of its children.

Conclusion

We can conclude that the concept of ownership can greatly enhance the pattern and emphasize certain design aspects. The missing support for readonly references in Ownership Types and Ownership Domains leads to inconvenient signatures of the child manipulation methods. Some applications will require the client to have (at least a readonly) reference to a component in the structure. Certain situations do not only require readonly references but also ownership transfer and support for shared contexts.

Applying deep ownership with one of the three reviewed systems fails when:

- dealing with dynamic structures where components can move or need to be integrated after creation.
- clients need to have a read/write reference to a component within the structure.
- components need to be shared.

Out of the three reviewed systems, the Universe type system offers the most eligible solution due to the concept of readonly references. For most scenarios where a deep ownership structure is too strict and therefore not an option, we suggest to encapsulate the whole composite structure in the client's context which still allows local reasoning about the code.

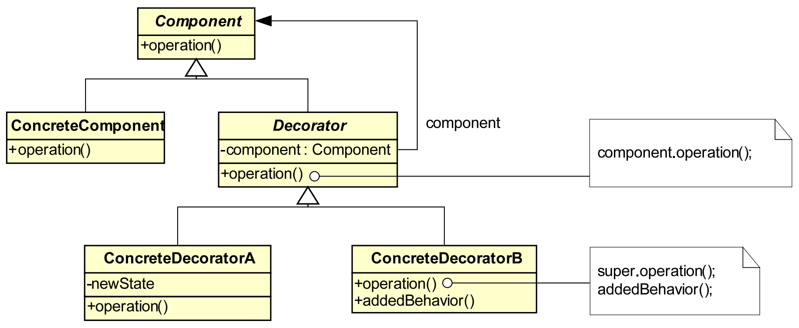
6.4 Decorator

Intent

> ”Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.”[4]

The decorator encloses the object to decorate transparently by forwarding requests to the original object while implementing new functionality. A decorator can decorate another decorator and thus an object can be attached with arbitrary responsibilities at run-time. The client remains unaware if it is operating on an original component or a decorated.

UML Diagram



Ownership Discussion

Investigating the pattern's structure reveals the idea that a decorator could be the owner of its component. This section discusses benefits gained by such an ownership structure, but also points out flaws and implementation problems of this proposition.

By specifying each decorator as the owner of its component, we could benefit from alias control to the component. Hereby, the decorator could safely establish invariants over the decoree. We

[4] [GHJV95] page 175

can, for example, think of a window system where each window can be decorated with a scrollbar or an additional border. In an implementation where the decorator (i.e. the scrollbar-decorator) is the owner of its decoree (the window), the decorator could safely establish assumptions over the window's state, like cache its size, etc.

In [GHJV95] the authors emphasize that a common problem of the decorator pattern is that it leads to a lot of little objects that all look alike. Each additionally attached responsibility results in an additional object, making it hard to learn and debug the system. One of the primary goals of the ownership concept is to make reasoning about code correctness much easier. Applying an ownership structure to the decorator pattern could improve the pattern's design by attenuating the mentioned problem. Dealing with huge and complicated object structures gets a lot more convenient when ownership typing is present.

Figure 6.5 shows the desired ownership structure as an example of a window system where each window can be additionally decorated with a scrollbar or a border.

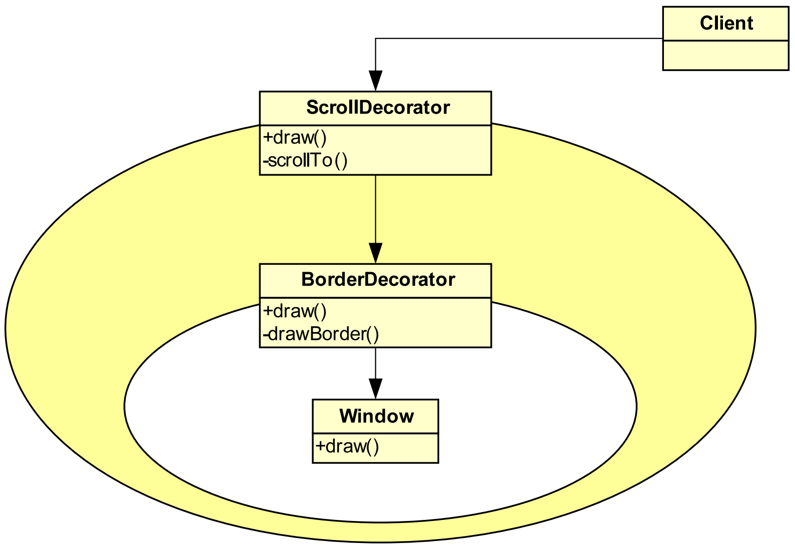


Figure 6.5: Desired ownership structure in a window system that uses the decorator pattern to dynamically enhance windows with a border or scrollbar behavior.

Although the mentioned benefits from an ownership structure in conjunction with the decorator pattern are evident, several major implementation problems and design flaws arise.

An obvious reason against an ownership structure in the decorator pattern is that it is very likely that the decorated component is shared between multiple clients with different decorations, at the cost of having no alias control for the decorator to the decoree. One key benefit of the pattern is that multiple clients can use the same component but each client can additionally decorate the component to suit himself. With an ownership structure, sharing components is not possible anymore, unless all clients use the same component with the same decorations.

The key aspect of the decorator pattern is that responsibilities can be attached and detached at runtime. The client just creates the new decorator object and passes the component to be decorated to the constructor of the decorator. Responsibilities can always be detached by operating directly on the component, instead of using the enclosing decorator object. An optimal ownership system would therefore support ownership transfer to dynamically assign responsibilities and adjust ownership of the decoree. Unfortunately, this dynamic behavior cannot be modeled with the ownership systems under review as none of them supports ownership transfer. In case where the decorator owns the decoree, the reviewed ownership systems require us to allocate the decorator *before* the decoree since the owner must be specified upon class instantiation. Detaching responsibilities is also not possible as the decoree would loose its owner, resulting in garbage collection of the owned object, or if the decoree is still referenced, no object in the system has write access rights to the component anymore. Thus, all reviewed systems fail to decorate an existing object while encapsulating it. We recognize that we deal with the same problem as in the composite pattern (6.3) with dynamic data structures.

Even if the responsibilities of a component are clear before its instantiation and stay unchanged until the component's disposal, applying ownership is not possible. Current ownership systems require us to first instantiate the owner which then instantiates all objects it owns. Regarding the decorator pattern, this means that all decorators have to consecutively create each other first until the decorated component gets created. This implies that the information what decorations must be applied to which object needs to be passed along the decorator's constructor chain and therefore a decorator needs to be aware of other existing decorations. This is clearly not the pattern's intent and all main benefits from the pattern's design are lost, i.e. dynamically configure objects with independent and freely combinable responsibilities.

Applying ownership to the decorator pattern might have its benefits but in many cases is not feasible. Above all, the reviewed ownership systems cannot provide a correct implementation of the pattern due to the absence of ownership transfer. We will therefore skip the discussion for each reviewed ownership system.

Conclusion

In conclusion we can say that certain benefits, like attenuating the problem of having many little objects or establishing safe invariants over the decoree's state, can be gained by applying ownership to the decorator pattern. However, in an environment were multiple clients need access to the same component but with different decorations, encapsulation is too strict. Currently, the main problem is that none of the reviewed systems can cope with the pattern's behavior where components need to be encapsulated after they are created, as ownership transfer is not supported.

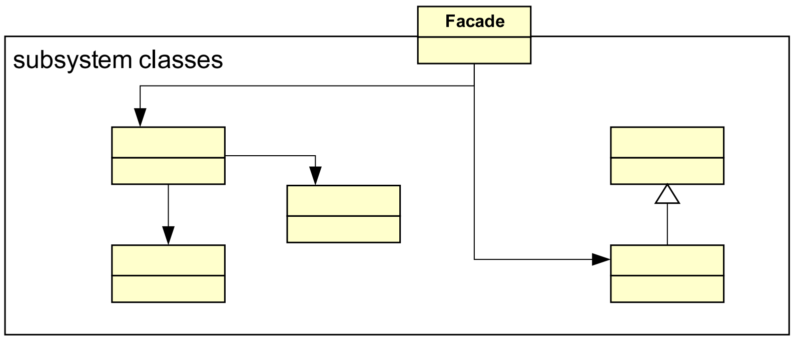
6.5 Facade

Intent

> ”Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.”[5]

Usually, larger programs are structured into collaborating subsystems in order to reduce complexity. The goal of a facade is to provide a higher-level interface to more general facilities and hereby hide lower-level details of the system. Clients that use the facade can be shielded from implementation details and therefore become independent of a concrete implementation. Note that the facade only provides an additional interface to a set of subsystems. It does not prevent clients to directly use the subsystems, if they really need to.

UML Diagram



[5][GHJV95] page 185

Ownership Discussion

The concept of ownership is pretty similar to the facade design pattern in a way that we want clients to use the owner of a context (the facade) instead of referencing the objects in the owner's context (the subsystem). A big difference between the proposed ownership concepts and the facade pattern is that ownership restricts aliasing of objects in the owner's context whereas in the facade pattern, using and referencing the subsystem is still possible. Basically, we can refer to the owner of an object structure as the facade of that structure.

Usually, we can divide a subsystem into public and private classes. Public classes (such as the facade class) can be used without limitations by any client whereas private classes only hold data and perform operations for other classes within the subsystem. Java supports this concept by the introduction of packages and modifiers. Nevertheless, ownership can provide additional help by establishing alias control in the subsystem in order to avoid representation leaking.

We would like to discuss ownership, based on two possible ownership structures:

1. When the facade encapsulates the whole subsystem One goal of defining a facade to a subsystem and let clients use the facade is to make the clients implementation-independent. If all clients only know the facade interface, the subsystem can easily be refactored or multiple implementations of the same subsystem can be offered, without affecting the clients. In such situations we propose declaring the facade as the owner of the subsystem in order to statically ensure that no uncontrolled aliasing of objects inside the subsystem is possible. Figure 6.7 outlines our proposition. The problem with declaring the facade as the owner of its subsystem is that there are situations where some clients still need direct access to parts of the subsystem. Depending on the ownership system, those clients may not have the necessary access rights anymore. With ownership, the facade needs to provide an interface that satisfies all clients, even those that need low-level access to some classes of the subsystem. Hence, the facade interface might possibly end up being too large and complicated for most of the clients and the main benefit of the pattern, providing a higher-level interface that is easier to use, is lost. We therefore propose the introduction of multiple interfaces to a facade reaching from very narrow to wide. For each client, an interface is defined that perfectly satisfies its needs. The `Facade` class implements all interfaces. Thus, although the facade has a very wide interface suiting all potential clients, each client uses its own interface when referring to the facade. Figure 6.6 illustrates this approach.

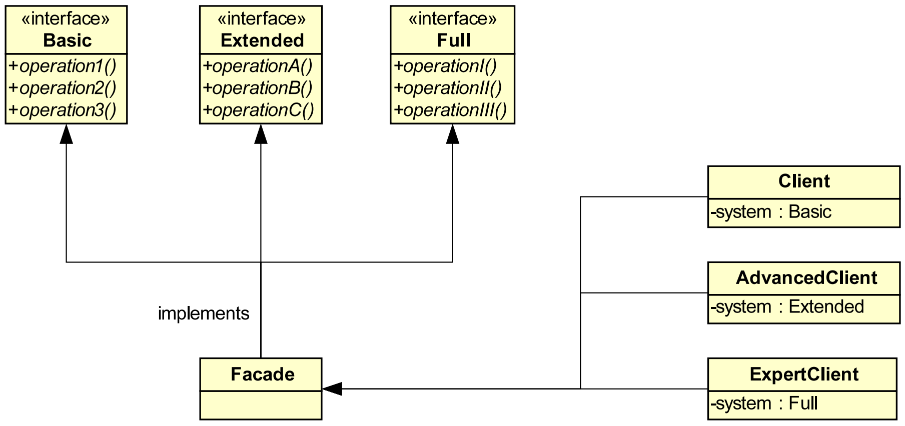


Figure 6.6: Multiple interfaces to the facade are defined (`Basic`, `Extended`, `Full`). `Facade` implements all defined interfaces and each client declares the facade object with the interface type most suited for its needs.

2. When only parts of the subsystem are encapsulated If encapsulation of the whole subsystem by the facade is not an option in general, applying ownership only to some parts of the system can still be. For example, the public classes of the subsystem might still own their private classes. Thus, most clients are able to perform their tasks over the high-level facade interface, clients that need access to certain parts of the subsystem still get it and, above all, representation exposure of the private parts in the subsystem is prevented. Figure 6.8 outlines the concept.

In all reviewed ownership system, the owner must always be created before the owned objects. This implies that when the facade owns the subsystem, it must trigger the instantiation process of the whole subsystem. Of course, the same applies if only private parts of the subsystem are owned. In that case, the public (exposed) parts are responsible for creating their appropriate private parts.

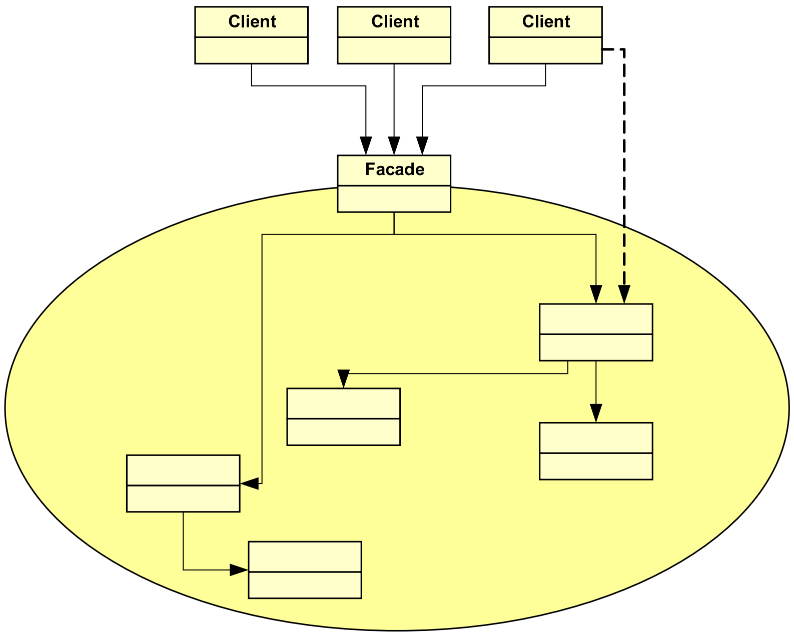


Figure 6.7: Illustration of an ownership structure for the facade pattern where the facade object is the owner of the whole subsystem. If supported by the system, `readonly` references are represented as dashed arrows.

Universe type system

An implementation of the desired ownership structure with the Universe type system is straightforward. When the facade object encapsulates the whole subsystem, it just declares all references to the subsystem as `rep` while all components of the subsystem have `peer` references to each other. The clients may still have `readonly` references to objects within the subsystem, but this is fine since no potential assumptions of the facade about the subsystem's state is violated. When only parts of the subsystem are encapsulated, objects residing in different parts may only hold `readonly` references to each other.

Ownership Types

Ownership typing with Ownership Types can easily be done: the facade just passes itself as the owner to all directly owned objects upon instantiation. In case where only private parts are owned by their public parts, the public parts pass themselves to the private parts as owner. In contrast to the Universe type system, referencing encapsulated parts is not possible anymore, not even readonly.

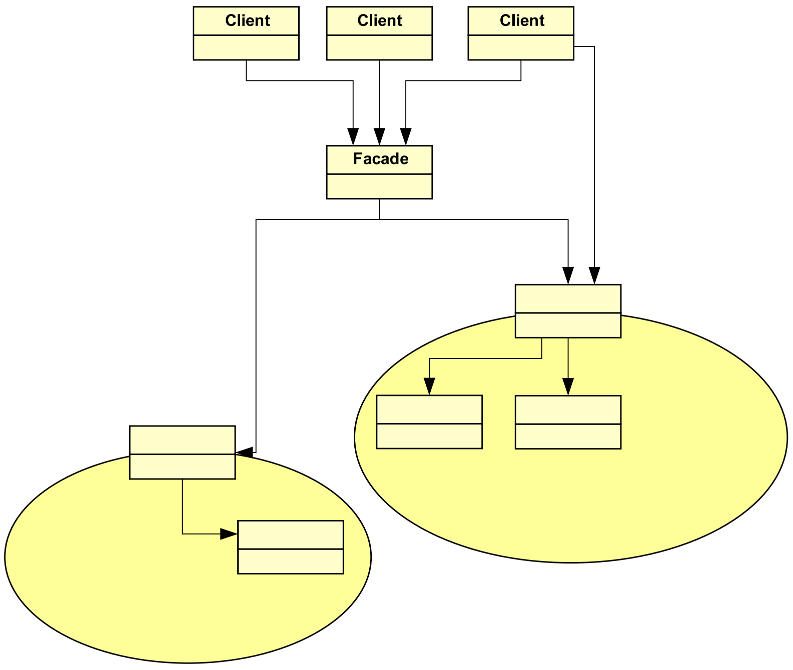


Figure 6.8: Illustration of an ownership structure for the facade pattern where only the private classes in a subsystem are encapsulated by their owner, the public class. Accessing public classes of the subsystem is still possible for those clients who need it.

Ownership Domains

Typing of the facade pattern with Ownership Domains can be done without problems. Thanks to the concept of public domains and links, interesting new ownership typings are possible.

public domain, private domain We can, for example, divide the subsystem's classes into a public domain, holding all public classes and a private domain, holding the private ones (Figure 6.9). In doing so, we can guarantee that all clients that have access to the facade, also have access to the public parts, but still shield the private parts from being aliased from outside.

private domains with links Another idea would be to declare the subsystem in two private domains, one holding all components that might still be accessed from outside and the other domain holding all components that may not be used from outside the subsystem. All domains holding clients that not only need access to the facade but also to some of the public classes can then be linked into the subsystem, if the linking rules permit it (Figure 6.10).

Conclusion

We can conclude that by using ownership in conjunction with the facade pattern we can fully hide the subsystem from the clients and make them totally implementation independent. The facade can safely establish assumptions about the subsystem's state. As a drawback, the facade must provide an interface, suitable for all clients. Unfortunately, this can result in a very wide interface. One can cope with this problem by either defining multiple interface-levels or only use ownership to encapsulate some private parts of the subsystem instead of the whole subsystem. Comparing the different ownership systems clearly shows that Ownership Domains is the most flexible one, giving the programmer the possibility to fully encapsulate the private parts of the subsystem while still allowing certain clients to access the public parts.

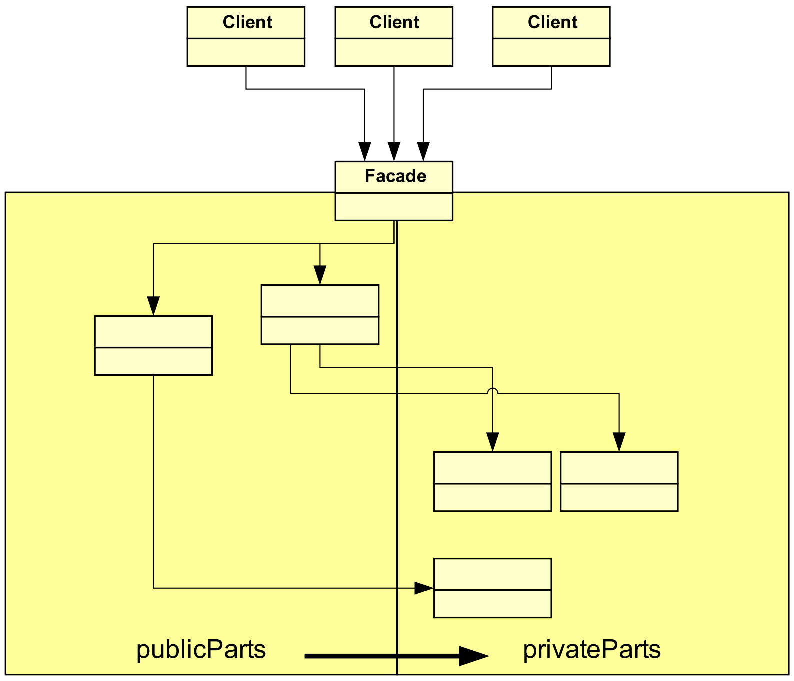


Figure 6.9: Ownership typing of the facade pattern where the domain, holding all public accessible classes, is declared as `public`. This allows all objects that have access to the facade to also access its public parts.

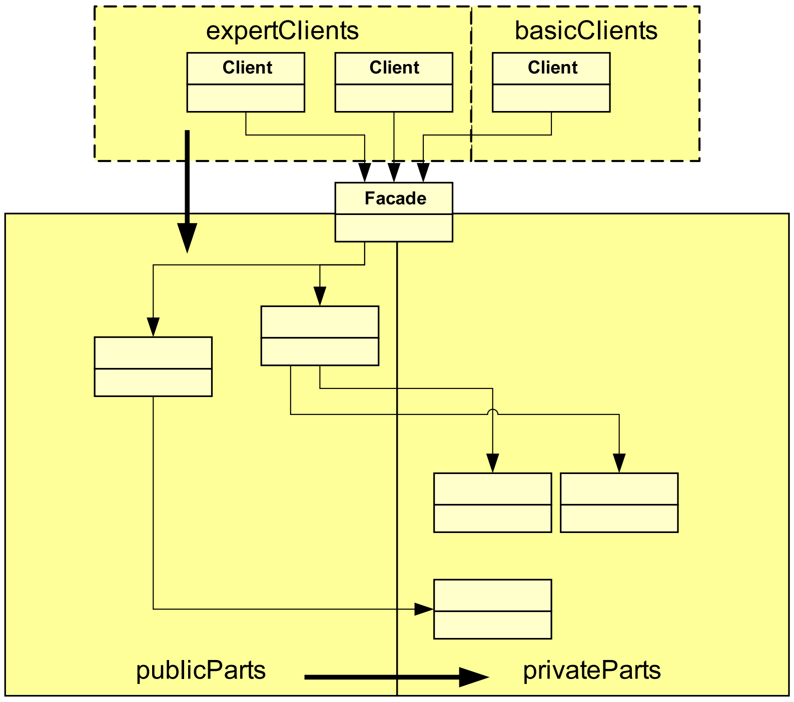


Figure 6.10: Ownership Domains typing of the facade pattern where the subsystem is defined in two domains, one holding all parts that must be accessible from outside, the other holds private parts of the subsystem. All clients that need direct access to some parts of the subsystem are located in the `expertClients` domain which is linked by the facade to the `publicParts` domain. The facade needs access to `expertClients` in order to establish the link.

6.6 Flyweight

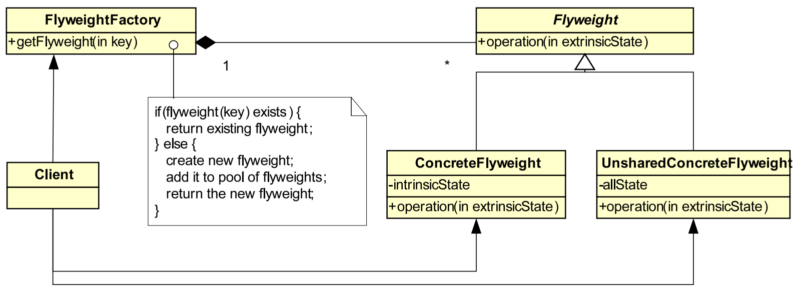
Intent

"Use sharing to support large numbers of fine-grained objects efficiently."[6]

Some applications would greatly benefit from using objects throughout their design, but this would result in way too many objects and memory usage. The idea behind the flyweight pattern is to have a set of predefined objects which will be used over and over again by the application through sharing. The prime example is the design of a text editor where each character can be represented with an object. Since all characters in the alphabet are reused over and over again in a text it would make sense to just maintain one object per character.

It is essential that all shared objects only have an *intrinsic* state which means that the state of the object does not depend on the object's context. The context-dependent state of an object is called *extrinsic* and has to be passed to the object each time a context-dependent operation is executed. For instance, the text editor will pass the format to the character object when calling `draw()`, since the same character(-object) can appear in different contexts with different formats. It is therefore crucial to distinguish between an object's intrinsic and extrinsic state.

UML Diagram


FlyweightFactory
+getFlyweight(in key)
Flyweight
+operation(in extrinsicState)
1
*
if(flyweight(key) exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
Client
ConcreteFlyweight
-intrinsicState
+operation(in extrinsicState)
UnsharedConcreteFlyweight
-allState
+operation(in extrinsicState)

Ownership Discussion

An investigation of the flyweight pattern reveals some risks: as a result of sharing, the implementation has to prevent any flyweight modification because a state change impacts all clients. A careful removal of the extrinsic state is necessary so that the object can be used by any client at the same time. Still, a change of the intrinsic state can result in a wrong implementation and would impact all clients. Of course, carefully chosen Java access modifiers will help dramatically to reduce the problem of an unauthorized state change (only the flyweight factory should be able to setup and configure a flyweight) but an ownership concept that supports readonly references could still enhance the pattern's implementation and ease a correctness evaluation.

The usage of ownership in conjunction with the flyweight pattern can result in a much safer implementation of the pattern: if all flyweights would be owned by the flyweight factory and only readonly references to the flyweights could be passed out, it could be statically ensured that no client is able to change a shared flyweight. The flyweight factory could safely establish invariants over the flyweight's state without worrying about any modifications from a client.

Although we have a factory concept, in terms of ownership the flyweight pattern is different from an abstract factory (5.1) or factory method (5.2) pattern because the products (the flyweights) should remain in the factories context and only readonly references to the products must be passed out to the clients.

Problems arise when dealing with unshared flyweights. These instances can also be allocated by the factory but since they are unshared they contain not only the intrinsic- but also the extrinsic-state and must therefore perform state changes according to the object's context. The client is not

[6][GHJV95] page 195

able to manipulate the unshared flyweight's state as it only maintains a readonly reference. We recognize that for unshared flyweights we face the same problems as with creational patterns, like the abstract factory (5.1). Thus, the flyweight factory should only be responsible for maintaining shared flyweights. Unshared flyweights must be allocated by the client or flyweight ownership must be transferred to the client after creation, if the system supports this feature.

Universe type system

The Universe type system supports the desired ownership structure very well due to its concept of readonly references. The flyweight factory allocates all flyweights, keeping a `rep` reference. The clients can request a new flyweight instance at the factory by calling `getFlyweight(int key)` which returns the requested flyweight through a `readonly` reference. Due to the fact that the client and the flyweight reside in a different ownership context, the flyweight instance only has readonly access to the supplied extrinsic state, as shown in Figure 6.11. Anyway, this poses no problem since the flyweight only has to read the extrinsic state without having the need or permission to modify it. For the problems discussed in connection with unshared flyweights, proposed workarounds (the client must allocate the unshared flyweight) or ownership system extensions (like the allocation of an object in the caller's context) should be applied.

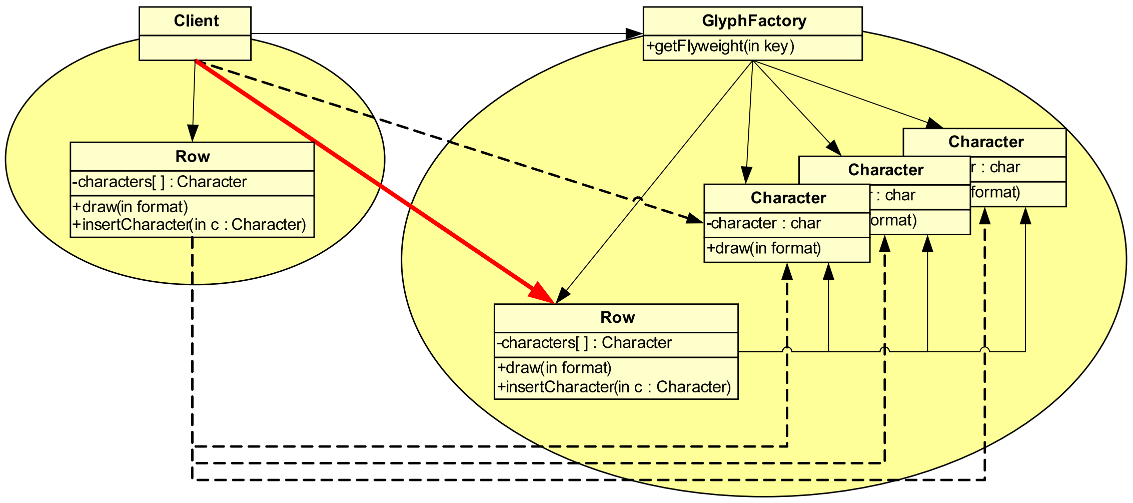


Figure 6.11: Ownership structure in the flyweight pattern. Dashed lines represent `readonly` references. This figure shows two possible allocations of the unshared `Row` class. If the client allocates its unshared instances as `rep`, it can call non-pure methods like `insertCharacter()` on the row. On the other hand, if the unshared flyweights are allocated in the factory's context, we have to deal with the problem that the client may now only call pure methods and cannot perform the necessary state change on the object anymore. The red arrow marks the reference that is readonly but should grant write access rights.

Example Implementation

As outlined in Figure 6.11, the following example implementation illustrates ownership in connection with the flyweight pattern. The implementation follows the example application of a text editor, as in [GHJV95]. All characters are shared flyweight objects and each row is an unshared flyweight, containing the characters. The flyweight factory creates all characters of the alphabet upon initialization so that `getGlyph()` can be pure and `readonly` references to characters can be returned immediately. `draw()` has to be pure as `Client` or `Row` invoke it through a `readonly` reference. The problem is that `draw()` is an I/O operation that updates the window's state when the character is visualized and may therefore not be pure by default. Hence, we changed the behavior of `draw()` in a way that it does not paint the character directly on the screen but just returns a graphics object (image) of the character with the correct style setting. The actual painting is invoked by the client.

This is somehow similar to the Java Swing painting process as all characters visualize themselves using a platform independent *graphics* object. The main difference to Swing is that the graphics object cannot be passed as a parameter but must be instantiated by the shared flyweight instance because pure methods may only have `readonly` parameters.

```
class App {
        peer GlyphFactory factory;

        public void test() {
                factory = new peer GlyphFactory();
                readonly Row r1 = (readonly Row) factory.getGlyph(GlyphFactory.Row);
                readonly Character a = (readonly Character) factory.getGlyph(GlyphFactory.A);

                //The following statement fails because inserting a character is not pure
                //r1.insertCharacter(a);

                /* If the unshared objects are directly allocated by the client everything
                 * works fine:
                 */
                rep Row r2 = new rep Row();
                r2.insertCharacter(a);
                rep Format f = new rep Format();
                f.bold = true;
                readonly Image rowImage = r2.draw(f);
                //send rowImage to Window in order to draw the row
        }
}


class GlyphFactory {
        //keys
        public final static int A = 0;
        public final static int Z = 25;
        //...

        public final static int OFFSET = 97;
        public final static int Row = 1000;
        //...

        private rep peer Glyph[] glyphs;

        public GlyphFactory() {
                this.glyphs = new rep peer Glyph[26];

                //initialize the glyphs pool
                for(int i = GlyphFactory.A; i <= GlyphFactory.Z; i++) {
                        rep Glyph g = new rep Character(i + GlyphFactory.OFFSET);
                        glyphs[i] = g;
                }
        }

        /* If all flyweights are generated upon factory creation, getGlyph() is pure
         * and can be invoked by clients over a readonly reference.
         */
        public readonly Glyph getGlyph(int key) {

                if((GlyphFactory.A <= key) && (key <= GlyphFactory.Z)) {
                        return glyphs[key];

                } else if(key == GlyphFactory.Row){
                        /* Unfortunately, with the current system also unshared flyweights
                         * like Row are passed as readonly to the client since the factory
                         * allocates the instance.
                         */
                        return new rep Row();

                } else {
                        return null;
                }
        }
}


//abstract flyweight
abstract class Glyph {
        public abstract pure readonly Image draw(readonly Format f);
}


//shared flyweight
class Character extends Glyph {
        //intrinsic state
        private int asciiCode;

        public pure Character(int asciiCode) {
                this.asciiCode = asciiCode;
        }

        public pure readonly Image draw(readonly Format f) {
                if (f.bold) {
                        //return an image of the character with a bold format
                        //...
                        return new Image();
                } else {
                        //return an image of the character with a normal format
                        //...
                        return new Image();
                }
        }
}


//unshared flyweight
class Row extends Glyph {
        //state
        rep java.util.ArrayList children;

        public pure Row() {
                this.children = new rep java.util.ArrayList();
        }

        public pure readonly Image draw(readonly Format f) {
                Image rowVisualization = new Image();

                for(int i = 0; i < children.size(); i++) {
                        readonly Glyph c = (Glyph) children.get(i);
                        readonly Image character = c.draw(f);
                        //append Image character to row visualization
                        //...
                }
                return rowVisualization;
        }

        public void insertCharacter(readonly Character c) {
                children.add(c);
        }
}

class Format {
        boolean bold;
```

```
}

/*
 * Image represents the graphical visualization of a character with
 * a specific format. The Image instance can be passed to the Window
 * for visualization.
 */
class Image {
       public pure Image() {}
}
```

Ownership Types

With Ownership Types we are not able to model the desired ownership structure due to the lack of readonly types. If the factory would propagate itself as the owner of the created flyweights, no client would be able to access them anymore, unless the client classes would be implemented as inner classes of the factory which is neither likely nor desired. In order to make the flyweights accessible by any client they must reside in the global context `world`. But in this case we would give up any alias control for the flyweights.

Unless readonly types are introduced, the Ownership Types concept clearly fails to enhance the flyweight pattern implementation.

Ownership Domains

With Ownership Domains we do not achieve to implement the desired ownership structure either because we lack the concept of readonly references. In contrast to Ownership Types, one is not forced to allocate all flyweight instances in the global context anymore since either a public domain `flyweights` could be declared at the flyweight factory, holding all flyweight objects, or all domains holding factory clients could be linked to the factory's private domain `flyweights` in which all flyweights will be allocated. We will call the union of all domains holding a factory client from now on `clients`, as shown in Figure 6.12.

Both implementation possibilities do not accomplish our main goal: enhancing the pattern's implementation in a way that we can ensure that no shared flyweight can be modified by any client.

However, with both implementation possibilities we could at least statically ensure that only objects that have access rights to the factory can manipulate the flyweights. By this, we forbid at least that flyweight-references cannot be freely passed around anymore and we only need to care about possible manipulations by the factory users. This is still an advantage compared to an implementation with no ownership at all and we therefore suggest using this approach when implementing the flyweight pattern.

Since all clients have read/write access to the flyweight instances, unshared flyweights could also be allocated by the factory and each client still has the necessary access rights to perform the required state change operations on their instances.

Unfortunately, due to the absence of readonly references we have to deal with another problem: for shared flyweights the extrinsic state has to be supplied by the client but the `flyweights` domain has no access rights to the clients domain in order to read the supplied state-object. We have to make sure that `flyweights` has access to `clients` by establishing all necessary links (if the application structure permits this linking at all). If the extrinsic state can be expressed by value types, this problem does not exist.

Possible System Extension To enhance a pattern implementation with Ownership Domains we propose to extend the system to also support *readonly domains* that behave exactly like `public` domains but only allow readonly references to objects located in the domain.

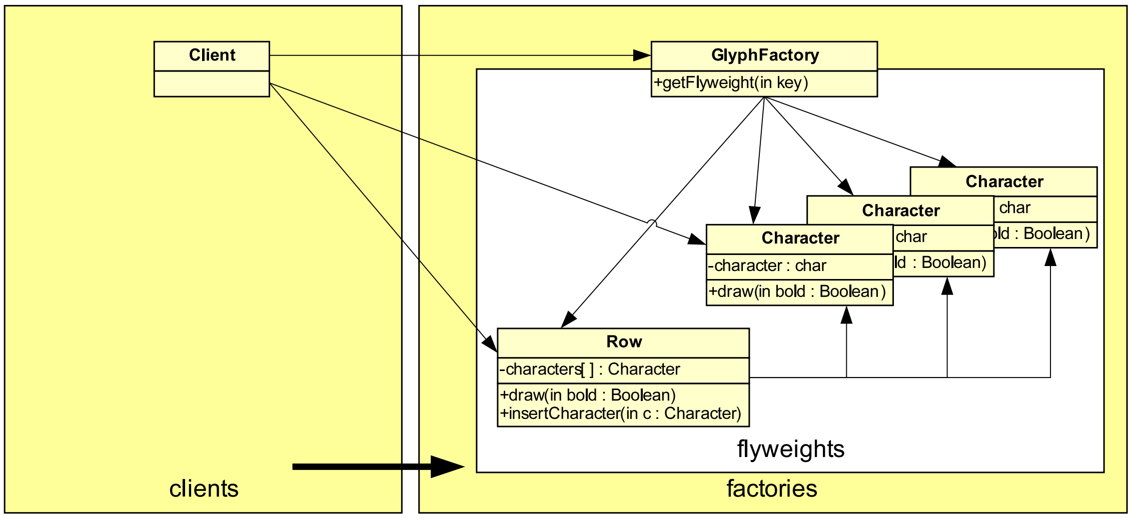


Figure 6.12: Illustration of a possible ownership structure in the flyweight pattern. Public domains have dashed lines whereas private domains have solid lines. `Row` instances are unshared flyweights whereas `Character` instances are shared. The `clients` domain is linked to the `factories` domain so that the clients gain access rights to the factory and its generated flyweights. For the sake of simplicity, the extrinsic state is expressed by only one value type `bold`. If the extrinsic state would be expressed by an object and allocated at the `clients` domain, one would have to make sure that `flyweights` has access rights to `clients` in order to read the object that represents the extrinsic state.

Example Implementation

The following example illustrates ownership typing with Ownership Domains in connection with the previously mentioned text editor example.

```
class App {
       domain clients;
       domain factories;
       link clients -> factories;

       factories GlyphFactory factory;

       public void test(){
              factory = new GlyphFactory();
              clients Client<factories> c = new Client<factories>(factory);
              c.addRowAndWrite();
       }
}


class Client<factories> {
       final factories GlyphFactory factory;

       public Client(factories GlyphFactory factory) {
              this.factory = factory;
       }

       public void addRowAndWrite() {
              factory.flyweights Row row = (Row) factory.getGlyph(GlyphFactory.Row);
              factory.flyweights Character a = (Character) factory.getGlyph(GlyphFactory.A);
              row.insertCharacter(a);
       }

}
```

```
class GlyphFactory {
       public domain flyweights;

       //keys
       public final static int A = 0;
       public final static int Z = 25;
       //...

       public final static int OFFSET = 97;
       public final static int Row = 1000;
       //...

       private flyweights Glyph[flyweights] glyphs;

       public GlyphFactory() {
              glyphs = new Glyph[26];

              //initialize the glyphs pool
              flyweights Glyph g;

              for(int i = GlyphFactory.A; i <= GlyphFactory.Z; i++) {
                     g = new Character(i + GlyphFactory.OFFSET);
                     glyphs[i] = g;
              }

       }

       public flyweights Glyph getGlyph(int key) {

              if((GlyphFactory.A <= key) && (key <= GlyphFactory.Z)) {
                     return glyphs[key];
              }
              else if(key == GlyphFactory.Row) {
                     return new Row();
              } else {
                     return null;
              }
       }
}

//abstract flyweight
abstract class Glyph {
       public abstract void draw(boolean bold);
}

//shared flyweight
class Character extends Glyph {
       private int asciiCode; //intrinsic state

       public Character(int asciiCode) {
              this.asciiCode = asciiCode;
       }

       public void draw(boolean bold) {
              if (bold) {
                     //draw bold character
              } else {
                     //draw normal character
              }
       }
}

//unshared flyweight
class Row extends Glyph {
       owner Glyph[owner] children;
       int index=0;
```

```
        public Row() {
                children = new Glyph[256];
        }

        public void draw(boolean bold) {
                for(int i = 0; i < children.length; i++) {
                        owner Glyph c = children[i];
                        c.draw(bold);
                }
        }

        public void insertCharacter(owner Character c) {
                children[index++] = c;
        }

        //...
}
```

Conclusion

Ownership facilitates reasoning about the correctness of a flyweight implementation because shared flyweights cannot be manipulated by clients anymore. Only the Universe type system is able to model the desired ownership structure as the other two systems lack the concept of readonly references. With Ownership Domains we cannot guarantee that no client manipulates the flyweights but we can at least have alias control for the flyweight instances.

6.7 Proxy

Intent

”Provide a surrogate or placeholder for another object to control access to it.”[7]

In practice, there are many reasons to introduce a level of indirection by calling a proxy object first. We can distinguish the following common situations where the proxy pattern is applicable:

1. **Remote proxy** — In distributed systems a remote object, residing in a different address space, is usually represented as a proxy in the local address space. All messages to the remote object are sent to the proxy which then forwards the message to the remote object.

2. **Virtual proxy** — In some cases, instantiation and initialization of an object is expensive and should therefore be delayed until the object is really needed. In such a situation the application should just create a proxy as a placeholder, instead of the expensive object. Only when the expensive object is really needed it gets created by the proxy which then forwards all requests to it.

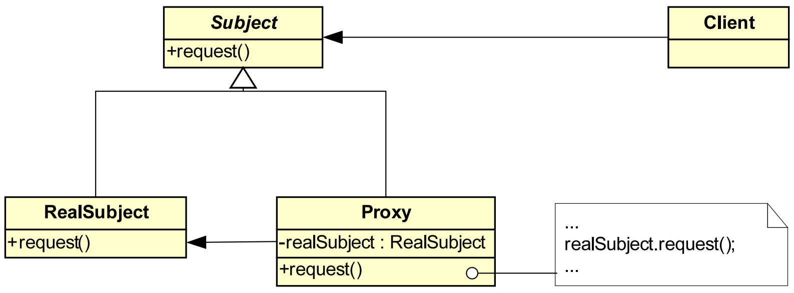
3. **Protection proxy** — A proxy can also be a good choice to establish access control for an object. When a client object wants to access the subject it first needs to contact the proxy, which then decides whether or not to grant access.

4. **Smart reference** — By encapsulating the original object we introduce a level of indirection, making housekeeping tasks like reference counting or caching possible.

In all cases the proxy implements the interface defined for a certain subject and can therefore substitute the real subject. The proxy keeps a reference to the real subject in order to forward all requests while optionally performing additional tasks.

[7] [GHJV95] page 207

UML Diagram



Ownership Discussion

We will discuss ownership in the proxy pattern based on the four possible scenarios where a proxy is likely to be used. The question is whether or not it would be useful if the proxy would own the subject. All benefits, risk, and implementation problems will be discussed in the following.

Remote proxy In case of a remote proxy in a distributed scenario, introducing ownership does not make sense since the subject is not even located in the same address space (likely not even on the same machine). In this scenario, the proxy serves primary as the communication facility, providing the same interface as the remote object but taking the responsibility of sending the messages and marshaling the arguments and return values. Due to the different address spaces, uncontrolled aliasing of the subject is not possible and there is no need to introduce ownership.

Virtual proxy In case of a virtual proxy, total encapsulation of the subject by the proxy becomes an interesting idea. The main goal of the virtual proxy is to provide an on-demand subject creation and initialization because these operations may be very expensive. It might be useful for the proxy to store and cache certain additional information about the subject so that the proxy can handle as many requests as possible without creating the subject. For instance, in a text editor a proxy may serve as a placeholder for an image because loading images into the layout is an expensive operation. Consequently, it makes sense that the proxy stores information about the image (like its extent, title, etc.) in order to handle some image queries from the system without having to load the actual image. In general, a virtual proxy caches some information about its subject. The use of ownership in connection with a virtual proxy can make an implementation safer. By encapsulating the subject in the proxy's context we can restrict aliasing of the subject from outside. Thanks to this ownership structure the proxy can safely establish assumptions about the subject's state and build up a cache of certain subject properties. This will allow the proxy to handle some requests by on its own, instead of forwarding everything to the subject.

Protection proxy In case of a protection proxy, ownership will help us to make the implementation a lot safer. The goal of the protection proxy is to control access to the subject. By declaring the proxy the owner of the subject, we can eliminate the possibility of representation leaking. Hence, it is not possible for clients to obtain a direct reference to the subject, which enables them to bypass the proxy's access control. Depending on the implementation and the used ownership system it is still possible for clients to gain a readonly reference to the subject.

Smart reference If the proxy is used to establish smart references to a subject, the main goal is to maintain information about the subject's state, like counting references or providing a locking mechanism to the subject. Again, this information is very sensitive and an implementation has to make sure that *all* clients will use the proxy as their access point to the subject. Otherwise the gathered information about the subject will be wrong and could result in a faulty behavior of the application. Introducing ownership in this scenario is desired. With the proxy as the owner of the subject it is statically ensured that no uncontrolled aliasing of the subject occurs. It is to point out that the used ownership system should provide support for readonly reference suppression since even the leaking of readonly references can lead to a faulty implementation.

Universe type system

As shown in Figure 6.13, when using the Universe type system the proxy is responsible for creating and maintaining a `rep` reference to the subject. Due to the subject's encapsulation in the proxy's context, all clients may only access the subject through the proxy or through a direct `readonly` reference, making any direct-modification of the subject impossible.

The proposed ownership structure enables us to safely implement a virtual proxy, since no objects outside the proxy's context can modify the subject and hereby invalidate any cached subject properties. Depending on the scenario, the Universe type system can also make an implementation of a proxy, that offers smart references to the subject, safer. Ownership ensures that clients can only manipulate the subject by calling the proxy first, making the proxy the central access point to the subject. This design makes implementations of locking mechanisms a lot easier. Of course, direct `readonly` references to the subject are still possible but such accesses do usually not need to be synchronized. If smart references are used for reference counting it gets more difficult as the proxy must also keep track of all `readonly` references to the subject. In this case, the programmer has to make sure that the proxy does not return a subject reference to the clients in order to keep track of the correct number of references.

In a scenario of a protection proxy, it strongly depends on the situation whether the implementation can benefit from the Universe type system or not. On the one hand, ownership ensures that the proxy is the central access point for subject updates. On the other hand, the Universe type system still allows direct `readonly` references to the subject. The programmer must ensure that no subject reference is transfered to the clients. The Universe type system can therefore only help to ensure access control for subject updates.

We can conclude that ownership can greatly improve the design of a protection proxy if access control is only necessary for subject updates. Situations where also read access must be controlled, the Universe type system does not help us to improve our pattern implementation.

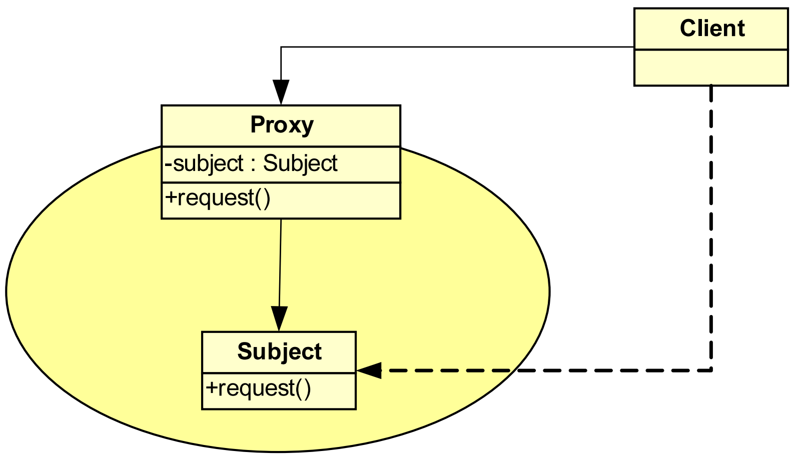


Figure 6.13: Illustration of the proxy pattern in connection with the Universe type system. Dashed lines represent `readonly` references. The proxy is the owner of the subject. The client may still directly access the subject but only through a `readonly` reference.

Ownership Types

The aliasing policy in Ownership Types can make an implementation of the proxy pattern even safer, as shown in Figure 6.14. Once again, an implementation of a virtual proxy greatly benefits from ownership, making it easy to cache certain subject properties. An implementation of a proxy that provides smart references can also benefit from Ownership Types as locking and reference counting can be fully implemented at the proxy.

Due to the lack of readonly references, Ownership Types makes it also safer to implement a protection proxy since the encapsulated subject may not be accessed from any client, not even through a readonly reference.

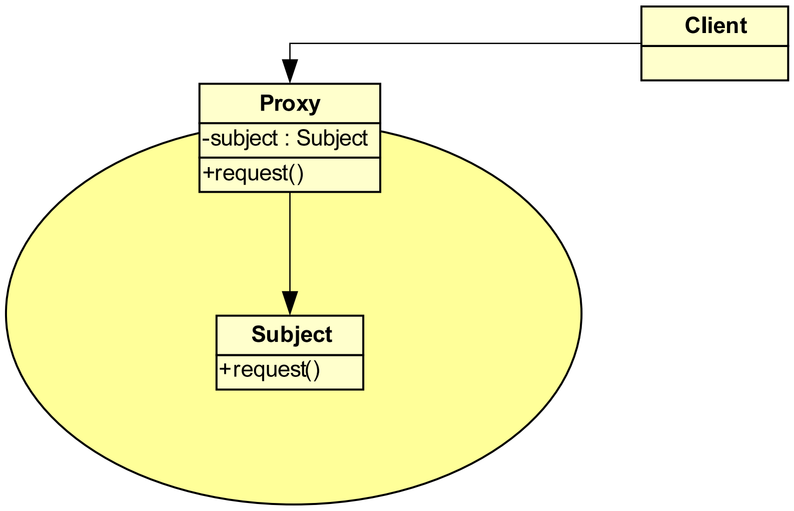


Figure 6.14: The proxy encapsulates the subject by propagating himself as the owner upon creation. The client may only access the proxy, making it the global access point to the subject.

Ownership Domains

Ownership typing of the proxy pattern using Ownership Domains leads to the same benefits as already mentioned in the Ownership Types section. As outlined in Figure 6.15, the proxy spans a private domain holding the subject and hereby prevents direct references.

Conclusion

The concept of ownership can greatly improve an implementation of the proxy pattern. By encapsulating the subject, the proxy can safely establish assumptions over the subject's state, cache certain subject properties, control access or provide a locking mechanism for the subject. The ownership system enforces the proxy to be the single access point to the subject which is one of the pattern's main obligations. In contrast to the facade pattern (6.5), the proxy does not aim to provide a simple interface to an underlying subsystem while still allowing some clients to use the subsystem. The proxy aims to act as a placeholder or substitute for the subject by encapsulating it.

When comparing the reviewed ownership systems we can conclude that all systems achieve to enhance the pattern's implementation. It is to point out though that due to the inability to suppress the leaking of readonly references, the Universe type system sometimes fails to provide an aliasing policy strict enough. For instance, this is the case when read access control or reference counting is desired.

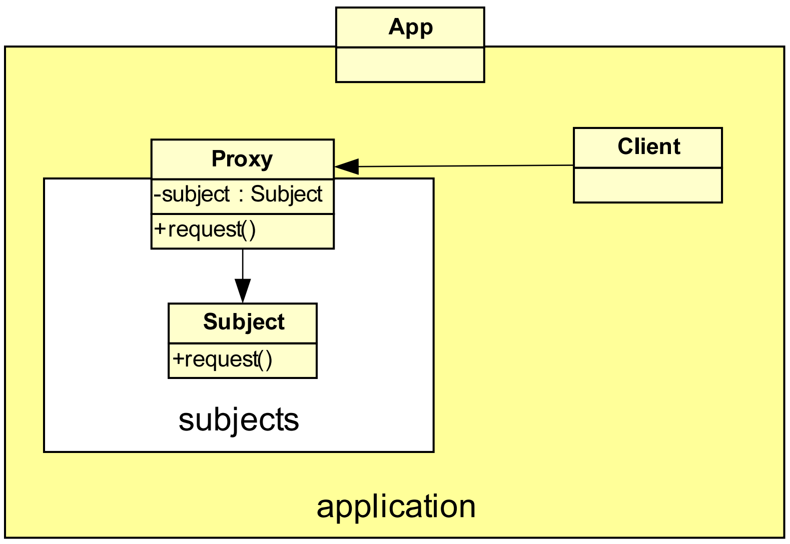


Figure 6.15: Example ownership structure of a proxy using Ownership Domains. The proxy spans a private domain `subjects` that holds the subject instance, making it impossible for the clients to directly access the subject. The proxy remains the single access point to the subject.

7 Behavioral Patterns

7.1 Iterator

Intent

> ”Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.”[1]

The iterator pattern lets clients access elements of an aggregate without forcing the aggregate to expose its internal representation. Furthermore, many different iterators may be defined and instantiated for one single aggregate, allowing clients to use different aggregate traversal methods, element filtering or multiple concurrent traversals.

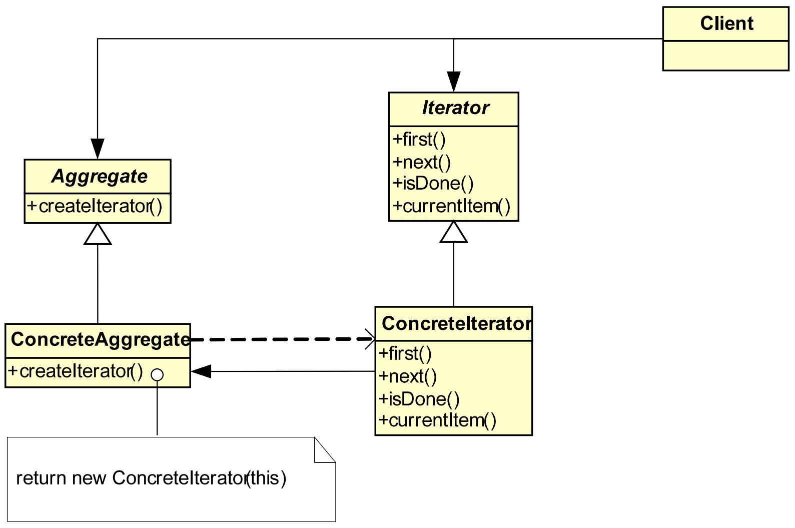
The client object does not have to care about a concrete aggregate implementation or the concrete type of the iterator. It just operates on an abstract iterator interface and the aggregate is responsible for instantiating the corresponding iterator class. `createIterator()` is an example of a factory method (5.2).

The obligations of an iterator are:

- support multiple traversal methods.
- simplify the aggregate’s interface since traversal behavior is implemented in the iterator and not in the aggregate.
- support concurrent traversals of a single aggregate.

[1] [GHJV95] page 257

UML Diagram



An iterator can even define a smaller interface than outlined the UML diagram. For example the `java.util.Iterator<T>` interface just consists of the three methods

- `boolean hasNext()`
- `T next()`
- `void remove()`

where even the `remove()` operation is optional. Thus, an Iterator interface can basically define only two operations (`hasNext()`and `next()`). The new `next()` operation merges `next()` and `currentItem()` from the iterator proposition in [GHJV95]. If the iterator also offers methods to modify the data structure (e.g. the iterator implements `remove()`), we speak of a *modifying iterator*. Otherwise, it is a *readonly iterator*. Modifying iterators that offer multiple concurrent traversals need to be synchronized.

Ownership Discussion

The iterator pattern is of great importance. Although an ownership type system needs to ensure that no representation exposure occurs, the iterator instance should still be able to traverse the aggregate's representation. The examples in Part I show how the different ownership systems deal with an iterator scenario.

In the following ownership discussion we want to review different iterator designs compiled in [Nob00]. Figure 7.1 gives a structured overview of the different possible iterator designs. In contrast to the discussion in the paper, the following sections discuss each iterator design with a focus on the desired ownership structure and feasibility under each reviewed system.

External Iterator

The external iterator provides the iterator's standard interface and acts as an independent client of the aggregate. An external iterator has no references into the aggregate's representation and only uses the public interface of the aggregate. Therefore, the iterator does not expose the aggregate's representation, but only stores the number of already traversed elements. Upon a `currentItem()` call the iterator requests the i-th item of the data structure. From a design perspective, the external iterator is well designed because it provides sequential access while not breaching the

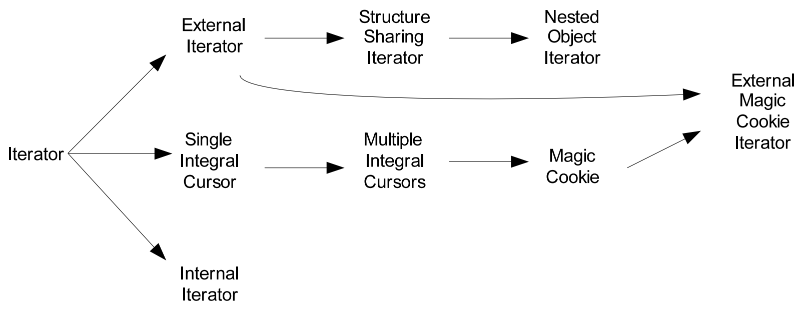


Figure 7.1: Different iterator designs as presented in [Nob00]

aggregate's representation encapsulation. The main disadvantage is that iteration is slow because only linear-time traversal is guaranteed.

From an ownership perspective the implementation of the external iterator is easy since only the aggregate's public interface is used. The aggregate can safely own its representation and does not need to allow any other objects to access the internal structure. We therefore suggest full encapsulation of the aggregate's internal structure while allocating the iterator instance in the same context as the aggregate. Ownership typing helps to improve the external iterator's implementation as the system ensures that no representation leaking of the aggregate occurs. Thus, it is statically guaranteed that the iterator behaves as an external instance, not knowing anything about the aggregate's internal structure. As long as the iterator has full access rights to the data structure's public interface, an implementation of both, a readonly or a modifying iterator is possible.

All ownership systems under review provide the desired behavior while Ownership Types and Ownership Domains guarantee that not even readonly references to the aggregate's representation can exist.

Structure Sharing Iterator

The structure sharing iterator offers the same interface as the external iterator but it directly accesses the internal structure of the aggregate in order to avoid the access penalty of the external iterator. A great disadvantage of the structure sharing iterator is that, as the name implies, it shares the structure with the aggregate and thus breaches the aggregate's encapsulation.

Ownership also helps in case of a structure sharing iterator. The Universe type system ensures that only `readonly` references are passed out to the iterator, allowing the iterator to still traverse the structure, but not to modify it. Hence, a modifying iterator cannot perform data structure manipulations directly, but needs to use the aggregate's public interface to perform the modifications. In order to be able to call non-pure methods on the aggregate's interface, the `Iterator` instance needs to be located in the same context as the aggregate.

The implementation of a structure sharing iterator with Ownership Types is not possible since aliasing of the aggregate's structure is forbidden for external objects, unless the iterator is implemented as an inner class of the aggregate (see nested object iterator).

Ownership Domains provide an eligible way to implement a structure sharing iterator: we declare a special `iterators` domain, holding only instances of `Iterator` and link this domain to the aggregate's representation domain. Thereby, it is ensured that the aggregate's structure is fully encapsulated and only the iterators have the necessary access permissions. Since the iterator gains read/write access rights to the aggregate's internal structure, an implementation of a modifying iterator that directly performs updates on the data structure is possible.

In case of a structure sharing iterator, the concept of ownership helps to greatly improve an implementation. The disadvantage of representation exposure does not exist anymore as ownership statically guarantees that no representation leaking occurs.

Nested Object Iterator

The nested object iterator behaves like a structure sharing iterator with the difference that the iterator is implemented as an inner class of the aggregate. This has the advantage that the iterator has full access to the aggregate's internal representation, without breaching encapsulation. The nested object iterator is well designed in terms of encapsulation, interface design and performance.

Still, the concept of ownership helps to improve an implementation by statically ensuring that the aggregate's implementation does not unintentionally expose its internal structure.

Due to its inner class implementation, the nested object iterator has full access to the aggregate's structure when typing with Ownership Types. Moreover, this allows an implementation of a modifying iterator.

Single Integral Cursor

The single integral cursor appeared in early versions of the Eiffel libraries. The idea was to integrate the iterator's behavior directly into the aggregate. Thus, the aggregate's interface is expanded with iterator functionality, offering `void start()` (to start the traversal), `boolean hasNext()` and `T next()` methods. From a design perspective, the idea is interesting since the object that really offers the data also offers the traversal through it. Additionally, encapsulation is guaranteed. The major drawback of this approach is that only a single client can traverse the aggregate.

The use of ownership additionally improves the implementation by ensuring that no representation leaking of the aggregate occurs. This iterator concept works fine for all reviewed ownership systems and an implementation is trivial for a readonly or a modifying iterator.

Multiple Integral Cursors

The multiple integral cursors iterator follows the design ideas from the single integral cursor, but allows multiple clients to iterate over the same data structure. The idea is that clients get a key upon an iteration request which they can provide on each iteration call. This key is used by the list to distinguish the different clients. Consequently, the aggregate's interface is expanded by the following operations:

- `int openCursor()` — allocates a new cursor
- `boolean hasNext(int cursor)` — indicates if the aggregate has more elements to traverse
- `T next(int cursor)` — returns the next element
- `void closeCursor(int cursor)` — releases the cursor

From an ownership perspective there exists no difference to a single integral cursor. Due to the fact that the cursor key is of value type, no aliasing problems can occur. The aggregate simply declares itself as the owner of its internal structure in order to avoid representation leaking.

Magic Cookie

In the multiple integral cursors' design the aggregate has to internally store each cursor's state and provide a client visible key. The idea of the magic cookie is to extract the integral cursor into an object that encapsulates the iterator's state. When a client requests a cursor, a magic cookie is returned and for all iterator operations that cookie is passed back to the aggregate. Hereby, the client knows nothing about the cookie's state.

The following code is taken from [Nob00] and illustrates an example implementation:

```
class MagicCookieList extends LinkedList {

      //list methods deleted

      Cookie cookie() {
```

```
                return new Cookie();
        }

        boolean hasNext(Cookie cookie) {
                return cookie.ilink != null;
        }

        Object next(Cookie cookie) {
                Object rv = cookie.ilink.value;
                cookie.ilink = cookie.ilink.next;
                return rv;
        }

        class Cookie {
                private Link ilink;

                Cookie() {
                        ilink = link;
                }
        }
```

In terms of ownership we have to ensure that the aggregate's internal representation is fully encapsulated and no internal references are passed out to clients. We also need to make sure that the cookie is accessible by the client since the client is maintaining the cookie reference.

The Universe type system copes very well with the desired structure: the aggregate allocates the cookie in its context (`rep`) and only a `readonly` reference to the magic cookie is passed out to the client. This has two advantages over an implementation without ownership: first, representation encapsulation of the aggregate is statically ensured and second, since the client only maintains a `readonly` reference to the magic cookie, we have absolute safety that only the aggregate manipulates the cookie. Figure 7.2 illustrates the desired ownership structure.

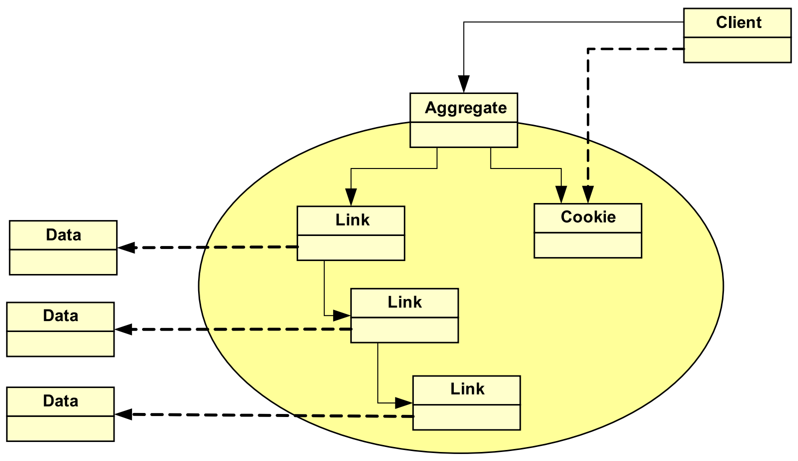


Figure 7.2: Ownership structure of the Magic Cookie Iterator. The aggregate creates the cookie object in its context in order to let the client only have a `readonly` reference to the cookie.

An implementation of the desired ownership structure with Ownership Types poses problems: the aggregate cannot allocate the magic cookie in its ownership context because this would disallow the client to hold a reference to the cookie (unless the client is implemented as an inner class of the aggregate, which is definitely not the case and not desirable). Hence, the cookie needs to be allocated in the same context as the client and the aggregate. This solution offers encapsulation safety for the aggregate but does not prevent the client from manipulating the cookie.

When using Ownership Domains, we propose the following ownership structure: the aggregate declares two domains, one holding the aggregate's internal representation in order to avoid representation leaking, the other domain holds the generated magic cookies. The client's owner domain can now be linked to the magic cookie domain to enable the clients to maintain a reference to their magic cookie. This ownership structure has the advantage that it guarantees that only the client domain can actually access the magic cookie, which prevents uncontrolled aliasing of the magic cookie. A magic cookie iterator can be modifying or readonly, but with multiple clients where some perform updates on the data structure, concurrency problems may emerge.

Clearly, the Universe type system offers the most elegant solution for an implementation of a magic cookie iterator because only read rights are granted to external objects like the client.

External Magic Cookie Iterator

A disadvantage of the internal cursors is that they have a different interface from the external iterators. If the magic cookie implements the iterator's interface, usage of the iterator would be the same as for an external iterator. The following code is taken from [Nob00] and illustrates an implementation:

```
class CookieIterator extends Cookie implements Iterator {
	public boolean hasNext() {
		return MagicCookieList.this.hasNext(this);
	}

	public Object next() {
		return MagicCookieList.this.next(this);
	}
}
```

In terms of usage, the client operates on the magic cookie, but the cookie still forwards all requests to the aggregate, supplying `this`.

This slightly changed scenario has an impact on the proposed ownership structure in conjunction with the Universe type system. Since the client only maintains a `readonly` reference to the cookie, it is not allowed to call non-pure methods on the cookie, such as `next()`. As a consequence, the aggregate needs to allocate the magic cookie as a peer object and the client has to reside in the same context as the aggregate and the cookie. This is somewhat restricting compared to other iterator designs and definitely not a good solution when using ownership. Furthermore, if the cookie implements a modifying iterator, the data structure cannot be manipulated directly by the cookie because the cookie only holds `readonly` references into the data structure. All modifications need to be delegated to the aggregate's public interface.

For Ownership Types and Ownership Domains there are no differences to the original magic cookie typing. Ownership Types requires the aggregate and the client to be either in the same context as the cookie, or to allocate the cookie in the shared context. Ownership Domains already offered a read/write reference to the cookie and thus the interface change of the cookie does not matter.

Internal Iterator

The design of an internal iterator has already been mentioned in [GHJV95]: the client supplies the iterator with an operation which will then be executed on each element of the aggregate. In this scenario, the client defines and allocates a new object which encapsulates the desired operation. This operation is then passed to the aggregate and applied to each element. An internal iterator is therefore much more tightly encapsulated within its aggregate than the external iterator. The disadvantage of this approach is that the client looses control over the iteration.

The following code is adapted from [Nob00] and illustrates an example implementation:

```
class LinkedListInternal extends LinkedList {
	public void run(Operation op) {
		Link l = link;
		while(l != null) {
			op.execute(l.value);
			l = l.next;
		}
```

```
        }
}

class Client {
        public void test() {
                LinkedListInternal list = new LinkedListInternal();
                list.run(new Operation() {
                        public void execute(Object o) {
                                System.out.println(o);
                        }
                });
        }
}
```

Feasibility of an internal iterator with ownership greatly depends on the ownership structure. In case of the Universe type system, we need to distinguish two different situations.

All operations are pure The `execute()` method in `Operation` and the `run()` method in the aggregate can be declared as pure. In this case, the `Operation` instance and the data objects of the aggregate can reside in any context since only `readonly` references are needed to perform the operation.

Some operations are non-pure The aggregate, all data objects, and the `Operation` instance must be peer to each other. The aggregate still encapsulates its internal structure by having `rep` references to `Link` objects. If these requirements cannot be met, an implementation of an internal iterator fails. Figure 7.3 illustrates the desired ownership structure.

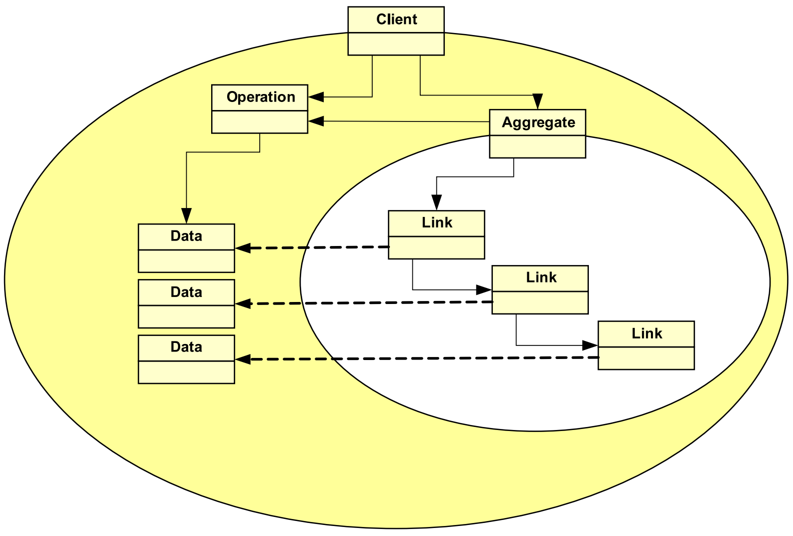


Figure 7.3: Desired ownership structure of an internal iterator. The aggregate, the operation and the data objects are located in the same ownership context and therefore also non-pure operations are supported. In case of a non-pure operation, the `Aggregate` instance simply downcasts the `readonly` reference to the data object to `peer`.

In case of Ownership Types or Ownership Domains, one has to ensure the following properties: the aggregate needs access rights to the `Operation` instance in order to execute the supplied operation. In addition, the `Operation` instance requires access rights to the data objects. When using Ownership Types, this can be achieved if all data objects and the `Operation` instance are ancestors of `Aggregate` in the ownership tree and $Operation \preceq DataElement$ holds. When using Ownership Domains, this can be attained by linking the domains accordingly.

The table in Figure 7.4 is adapted from [Nob00] to include an ownership review for each system.

	Simultaneous traversals	Efficient traversals	Aliases breaching encapsulation	UTS compliance	Ownership Types compliance	Ownership Domains compliance
External Iterator	Yes	No	No	+	+	+
Structure Sharing Iterator	Yes	Yes	Yes	+	-	o
Nested Object Iterator	Yes	Yes	-	+	o	o
Single Integral Cursor	No	Yes	No	+	+	+
Multiple Integral Cursors	Yes	Yes	No	+	+	+
Magic Cookie	Yes	Yes	-	+	o	o
External Magic Cookie Iterator	Yes	Yes	-	-	o	o
Internal Iterator	Nested	Yes	No	o	o	+

Figure 7.4: Overview of different iterator designs.

Conclusion

Ownership is very useful in conjunction with the iterator pattern. All ownership systems achieve an enhancement of the pattern's implementation by ensuring that no representation leaking of the aggregate's internal structure occurs. The concept of ownership makes an implementation of a structure sharing iterator possible without having the disadvantage of an encapsulation breach. In the case of a magic cookie iterator the implementation can gain additional safety from ownership by having alias control or readonly references to the cookie. A comparison of the three different ownership systems shows that due to readonly references, the Universe type system sometimes offers the most eligible solution, e.g. in case of a structure sharing iterator, nested object iterator, etc. In other situations, the Ownership Domains concept provides the best solutions due to the links concept that enables inter-context read/write access rights e.g. internal iterator, external magic cookie iterator, etc.

7.2 Observer

Intent

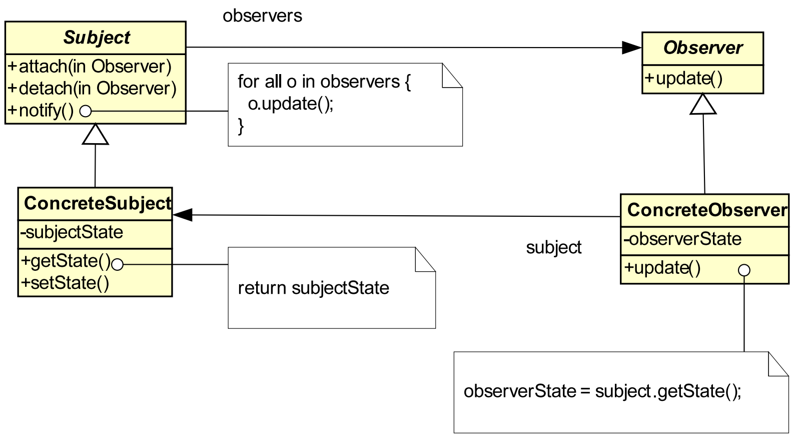
> ”Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”[2]

A primary motivation for using an observer pattern is to maintain consistency between related objects. An application with a graphical user interface for example should ensure consistency between the view and the underlying data. Since many views can represent the same data, we have a one-to-many dependency between the data and the different views.

The key objects in the pattern are the *subject* and the *observer*. The subject notifies the observer when it changes state so that the observer can adjust its state. This mechanism is also known as *publish-subscribe* and it enables broadcast communication between objects.

[2] [GHJV95] page 293

UML Diagram



Ownership Discussion

When analyzing the observer pattern in terms of ownership, we recognize that the subject and the observer are tightly coupled and both objects need to reference each other through a read/write reference. The prime example of an observer pattern is an application with a graphical user interface. The subject corresponds to the data and is therefore located in the application's domain layer, while the observers correspond to different views for the data and are thus located in the presentation layer. In this scenario, the subject and the observers are created independently from each other and later on, the observer is attached to the subject by calling `attach(Observer)`.

In another likely scenario, the observer pattern is combined with a composite pattern, where each parent registers itself as the observer of all children in order to reflect possible child state changes. The application in Chapter 9 implements such a scenario.

There are several possible ownership scenarios for the observer pattern, which we will discuss in the following.

The subject is the owner of its observers We can clearly say that this ownership structure is not desired and offers no additional guarantees in terms of safety. The subject does not know anything about its observers and is therefore not dependent on their state. The subject just maintains a list of all its observers in order to notify them about its state changes. No assumptions about the observers are made. Therefore, there is no need for the subject to control aliasing of the observers and an encapsulation makes no sense.

The observer is the owner of the subject We know that the observer is dependent on the subject's state since it is interested in all state changes of the subject. However, the observer just relies on being notified by the subject when it changes state. The observer is not concerned about other objects aliasing or modifying the subject's state as long as it is notified correctly. Hence, an encapsulation of the subject by the observer is neither necessary nor beneficial as it would only restrict other objects in interacting with the subject. In addition, a subject can have multiple observers but only one owner, which would pose an ownership structure problem anyway. The only special scenario where such a situation can occur is when the observer pattern is implemented together with a composite pattern (6.3) with a deep ownership structure. All parents observe their children in order to get updated on child state changes.

The subject is not a single object Assuming that not only the state of a single object, in the role of the subject, matters but all state changes of an object structure need to be propagated to the observers, it is worthwhile to encapsulate the (subject-)structure by declaring the root object as the owner. The proposed encapsulation has the benefit that it is statically ensured that all modification requests for the structure are passed through the root object, which can then notify all observers.

The subject and the observer reside in different ownership context This is the most realistic scenario since the subject and the observer are likely to be created in different contexts. The problem we face is the following: due to the different ownership contexts, aliasing between the subject and the observer is restricted. Considering the interaction diagram of the pattern, we notice that the observer attaches itself to the subject and hereby modifies the subject's internal state, hence `attach(Observer)` requires a read/write reference. Likewise, when the subject calls `update()` on each registered observer, `update()` will not be pure because the observer is adjusting its state according to the new subject's state. We can conclude that the ownership type system has to provide a mechanism to allow read/write references between the observer and the subject, although they might be located in different ownership contexts. Hence, the system must either allow the declaration of *friendly objects* which can freely reference each other, no matter in what context they reside, or provide *shared contexts* that allow references from a defined set of other objects in the system.

The subject and the observer reside in the same ownership context In this ownership scenario, a correct implementation of the observer pattern is trivial and requires no further discussion.

We can conclude that alias control is not a primary concern of the observer pattern. It is rather interesting if an implementation of the observer pattern is possible at all, assuming that one of the three reviewed ownership system is used.

It is to mention that a design alternative to the original proposition exists, where a so-called *change manager* is introduced to manage all events. The change manager mediates between the subjects and the observers and is usually implemented as a singleton. This scenario has no major differences to the original design, except that additional ownership aspects of the mediator (7.4) and singleton (5.5) patterns have to be considered. We will therefore skip a detailed discussion of the change manager scenario.

Universe type system

A feasibility evaluation of the presented scenarios in conjunction with the Universe type system clearly reveals certain incapabilities:

Situations where the subject and its observers are located in different contexts only allow readonly referencing. As shown in Figure 7.5, this is not enough as both objects are to call non-pure methods on each other. Moreover, ownership typing of situations where either the subject or the observer is the direct owner of the other fails since the Universe type system only provides `readonly` references from owned objects to their owner, as Figure 7.6 illustrates.

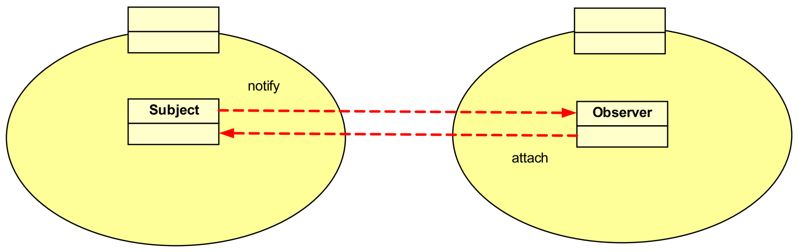


Figure 7.5: Universe type system ownership structure of the observer pattern where the subject and the observers reside in different ownership contexts. `readonly` references are represented as dashed arrows. The red color indicates references that are readonly but actually need to be read/write in order to perform the required calls.

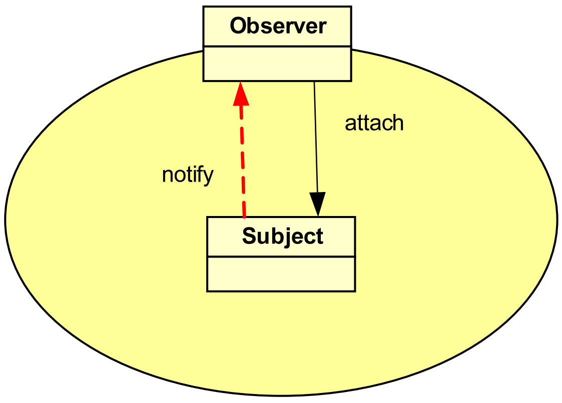


Figure 7.6: Universe type system ownership structure of the observer pattern where the observer is the owner of its subject. Dashed arrows represent `readonly` references whereas solid arrows represent `rep` references. Although the observer can attach itself to the subject, notification of a state change is not possible. Hence, the `readonly` reference from the subject to the observer is colored red.

Ownership Types

An implementation of the observer pattern with Ownership Types can lead to the same problems as using the Universe type system. If the subject and its observers are located in totally different ownership contexts, referencing each other is prohibited. However, as an advantage compared to the Universe type system, Ownership Types allow objects to access its ancestors and objects they own (see rule 3.1). So in case of a subject being the owner of its observers or an observer being the owner of the subject, bidirectional referencing is possible. Nevertheless, in a very likely ownership scenario the subject and the observers reside in totally different application (and ownership) contexts. An implementation under this scenario fails as well. Implementing the subject as an inner class of the observer or the other way around is probably not desired since the two classes are not related closely enough as that an inner class implementation is justified.

Solutions

An implementation of the observer pattern is only possible if the observer and the subject are located in the very same ownership context or, as illustrated in Figure 7.7, either object is the direct owner of the other.

Ownership Domains

The Ownership Domains concept offers the best implementation possibilities for the observer pattern because only with Ownership Domains a correct implementation of the pattern is (theoretically) possible under all ownership structures. The situation where the subject and the observers reside in the same context poses no problems. If the subject is the owner of its observers, one simply needs to link the context holding the observers with the subject's owner context in order to allow the owned objects to reference their owner. The same applies if the observer is the owner of the subject.

In contrast to the other two ownership concepts, it is even possible to implement the observer pattern if the subject and the observers reside in totally different contexts (see Figure 7.8). One just needs to establish bidirectional links between the different contexts. Of course, this only works if the linking constraints allow such a link-scenario (rule 4.4). The problem is that although it is theoretically possible to establish such a link structure, this will require linking all parent domains of the subject and the observers with each other, resulting in a generally weak encapsulation.

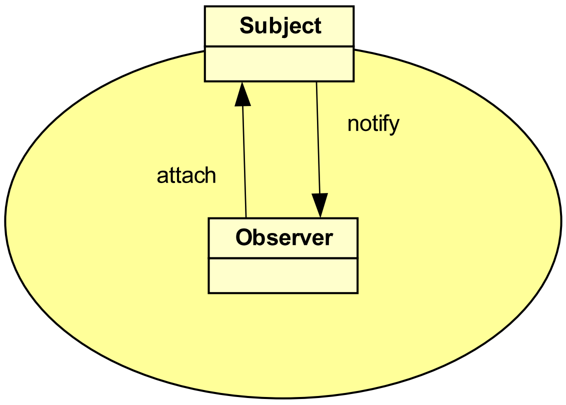


Figure 7.7: Ownership Types structure of the observer pattern when the subject is the owner of its observers. In contrast to the Universe type system, owned objects can have read/write references to their ancestors and objects they own.

Solutions

The observer pattern can be implemented under all possible ownership scenarios. One just needs to ensure that bidirectional referencing between the context holding the subject and all contexts holding observers is allowed. This can be achieved by establishing links accordingly.

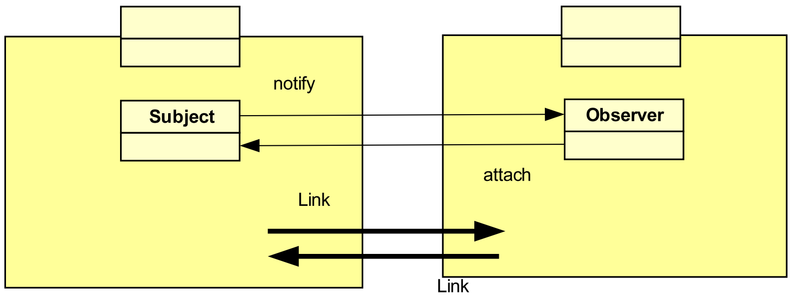


Figure 7.8: Ownership structure of the observer pattern with Ownership Domains when the subject and the observers are located in different ownership contexts. Bold arrows represent links. If the linking restrictions permit the bidirectional linking of the involved contexts, a correct implementation of the pattern can be accomplished.

Conclusion

Even though the concept of ownership improves a pattern implementation in case of the subject being an object structure instead of a single object, alias control is not a primary concern of the pattern.

The subject does not know anything about its observers and has therefore no need to restrict aliasing. The observer does only rely on getting correct notifications when a subject changes state. No further assumptions about the subject's state are made, hence there is no need for alias control of the subject either.

The crucial question is if a pattern implementation is even possible with an already existing ownership structure. Depending on the used ownership concept and existing ownership structure, a correct implementation can be done. The most restricting ownership concept in conjunction with the observer pattern is the Universe type system. An implementation of the pattern is only possible if the subject and all observers are located in the very same ownership context. Ownership

Types, in addition, does allow an implementation of the pattern when either involved object is the owner of the others. The most flexible concept is Ownership Domains that even allows a correct pattern implementation when the subject and the observers reside in totally different ownership contexts (with the condition that linking constraints allow a bidirectional linking of the involved contexts).

7.3 Visitor

Intent

> ”Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”[3]

A common scenario is that the application maintains a data structure and performs certain operations over the data. For example, a compiler parses the source code and builds an abstract syntax tree (the data structure). Then, the compiler performs several checks and operations over the abstract syntax tree, such as semantic checks, code generation, etc. In many scenarios it makes sense to keep the data structure separate from the operations to:

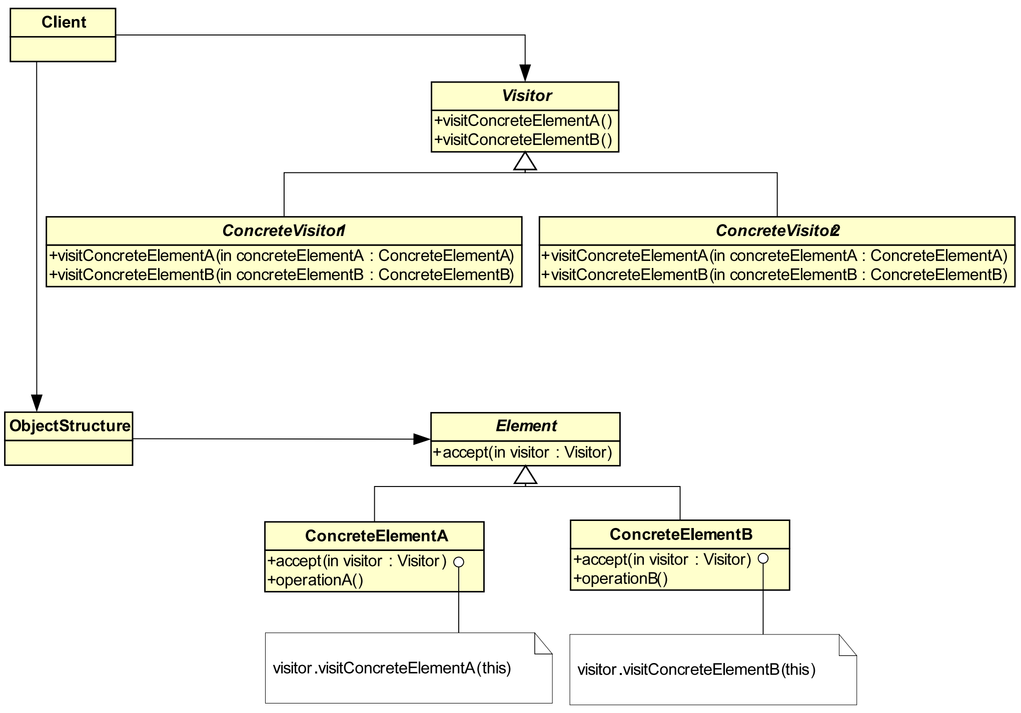
- not ”pollute” the data with several totally unrelated operations that will only make reading the code confusing.

- not require a recompilation of the existing data classes when a new operation is added.

The visitor pattern therefore separates the data classes from operations, allowing to easily add new operations without changing the existing data classes. A new operation just extends the abstract `Visitor` class and implements for each concrete element the appropriate behavior. When the data classes (subtypes of the abstract `Element` class) are likely to change, it is probably better to let the data classes define the operations as a data class change requires all visitors to adjust their interface, which might be costly. The main design flaw of the visitor pattern is that encapsulation of the data objects is broken since a visitor usually needs access to the data object’s internal structure in order to do its job. An advantage of the visitor compared to the iterator (7.1) is that the visitor can also traverse object structures of unrelated types, i.e. not all data elements need the same supertype. The only requirement for traversable types is the implementation of the `accept(Visitor)` method. The question of who is responsible for traversing the data structure cannot be answered easily as there are multiple strategies. On the one hand, the data structure can determine the traversal strategy if each element forwards the `accept()` call to its children in the desired order. Another possibility would be to use a separate iterator to determine the traversal method. But also the visitor could be responsible for traversing the data structure. This is an especially interesting option if different operations need different traversal methods.

[3] [GHJV95] page 331

UML Diagram


Client
Visitor
+visitConcreteElementA()
+visitConcreteElementB()
ConcreteVisitor1
+visitConcreteElementA(in concreteElementA : ConcreteElementA)
+visitConcreteElementB(in concreteElementB : ConcreteElementB)
ConcreteVisitor2
+visitConcreteElementA(in concreteElementA : ConcreteElementA)
+visitConcreteElementB(in concreteElementB : ConcreteElementB)
ObjectStructure
Element
+accept(in visitor : Visitor)
ConcreteElementA
+accept(in visitor : Visitor)
+operationA()
ConcreteElementB
+accept(in visitor : Visitor)
+operationB()
visitor.visitConcreteElementA(this)
visitor.visitConcreteElementB(this)

Ownership Discussion

Before discussing appropriate ownership structures for the visitor pattern we need to consult the interaction diagram presented in Figure 7.9 in order to understand all operations and references that occur during a visit.

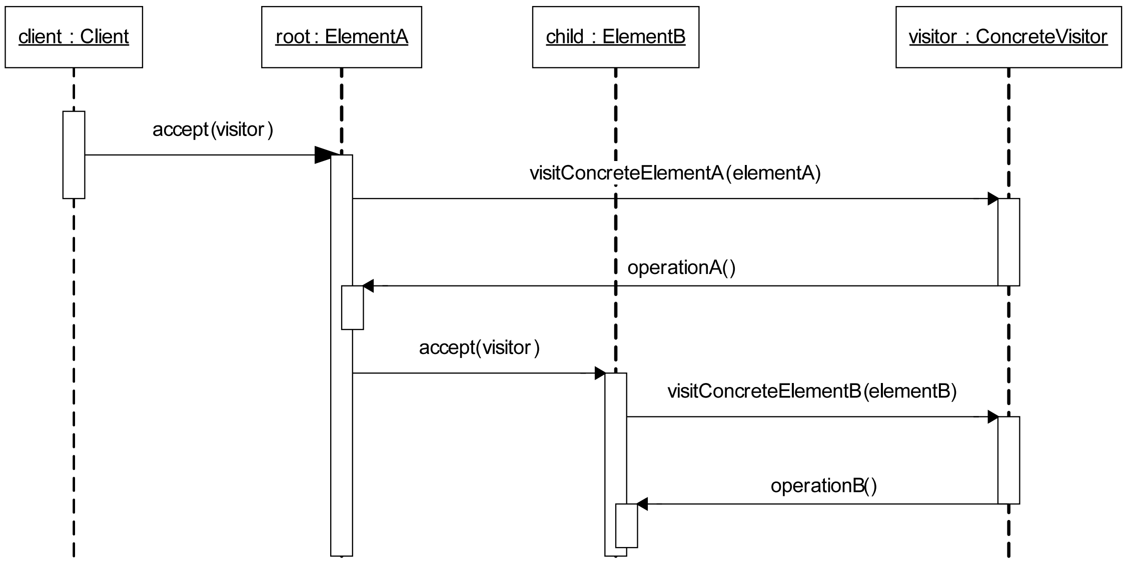


Figure 7.9: Interaction diagram of the visitor pattern where the traversal order is determined in the `accept()` method.

1. The client first calls `accept(visitor)` on the root node of the data structure, providing the concrete visitor that performs the desired operation.

2. The data elements then call `visitor.visitConcreteElement(this)` passing themselves to the visitor, where `ConcreteElement` refers to the concrete type of the data element involved.

3. During a `visitConcreteElement()` call, the visitor performs several queries and method calls on the data object.

4. The traversal order is either determined by the visitor, the object structure, or a separate iterator instance. In Figure 7.9, traversal order is specified in the object structure.

Hence, each data element holds a reference to the visitor instance in order to call the `visitConcreteElement()` method which is, depending on the operation, likely to be non-pure. Furthermore, the visitor holds a reference to the data element in order to query the element's state.

Since the data elements need to expose their internal data so that the visitor can correctly work, it is desired that the visitor is, together with the data structure, encapsulated in the client's context. An implementation greatly benefits from such an encapsulation as one does not need to worry about external objects, potentially modifying the exposed data structure, anymore.

From an encapsulation point of view, it is not necessary that the data element encapsulates the visitor in its context as it is not dependent on the visitor's state. However, both objects call non-pure methods on each other and the visitor thus needs to reside in the same context as the object that is being visited. Since it is likely that the traversed data structure is implemented as a composite pattern (6.3) that already makes use of a deep ownership structure (where each parent node owns its children), the used ownership system needs to support a mechanism to transfer ownership or otherwise the typing fails. With ownership transfer in place, the data elements pass ownership of the visitor instance on to the next element being visited. Hence, the visitor always remains in the context of the currently visited object. Alternatively, the used ownership system could also support a mechanism to grant special access rights for objects, i.e. the visitor instance is privileged to traverse the encapsulated data structure.

Universe type system

We would like to divide our discussion into the following two possible ownership scenarios:

The data structure and the visitors are located in a single ownership context Implementation of the desired ownership structure, where the client is the owner of all data elements and the visitor instances, reveals no problems. As shown in Figure 7.10, the client simply allocates the data structure and the visitor as `rep`.

The data structure has a deep ownership structure As already pointed out in the general ownership discussion, this scenario requires a mechanism to transfer ownership of the visitor instance along the object structure, unless all operations are pure. Due to the lack of ownership transfer, a typing of this scenario is likely to fail. Considering all other discussed restrictions that result out of a deep ownership structure in a composite pattern (6.3) we can conclude that using deep ownership in connection with a visitor pattern should be avoided when using the Universe type system.

Example Implementation

An implementation of the desired ownership structure is straight-forward and needs no further explanations. The client simply allocates the data structure and each `Visitor` instance as `rep` in its context and calls `accept(visitor)` on the root element in order to start the visit.

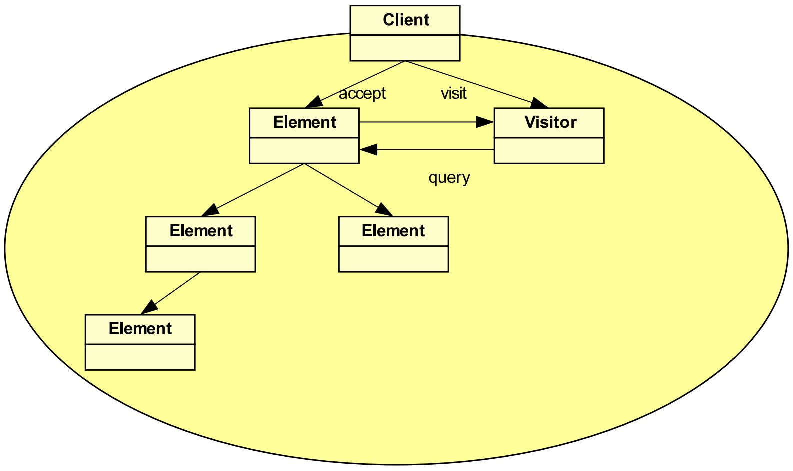


Figure 7.10: Ownership structure of the visitor pattern. The client encapsulates all data elements and the visitor instance in its context, allowing all involved elements to freely reference each other, while restricting aliasing and modifications from outside.

Ownership Types

Implementation of a flat ownership structure, where the client is the owner of the visitors and all the elements in the data structure, can easily be realized with Ownership Types. As all involved elements are siblings in the ownership tree, access rights between the visitor and each data element are granted. Figure 7.10 outlines the scenario.

However, like in the Universe type system, if the data structure is a composite with deep ownership we have access right problems. The client is the owner of the visitor and an ancestor of each data element in the ownership tree. Therefore, access rights from each data element to the visitor is granted. But, as each child node in the data structure is encapsulated in the parent's context, the visitor is not allowed to query the data element in order to perform its operation. The only way to work around this problem is to implement the visitor as an inner class of the data element. However, due to the fact that the data structure can be built from several different element types, which do not have to be type-related at all, this approach does also not work. Like the Universe type system, Ownership Types lack support for an ownership transfer mechanism and an implementation of the visitor pattern, able to traverse a deep ownership data structure, fails.

Ownership Domains

As in the other two ownership systems, an implementation of a flat ownership structure where the client is the owner of all involved objects is easy to do: the client simply spans a context, holding the data elements and the visitor instances, which ensures access rights between the objects.

In contrast to the other ownership systems, even an implementation of the visitor pattern, where the composite has a deep ownership structure is possible: the client declares two contexts, one holding the visitor instances and the other holding the root element of the data structure. In addition, each node spans a context, holding all its children and establishes a link from the visitor context to the children's context and vice versa. Figure 7.11 and the following section illustrates this scenario.

For the sake of simplicity, we suggest using only a flat ownership structure in conjunction with the visitor pattern.

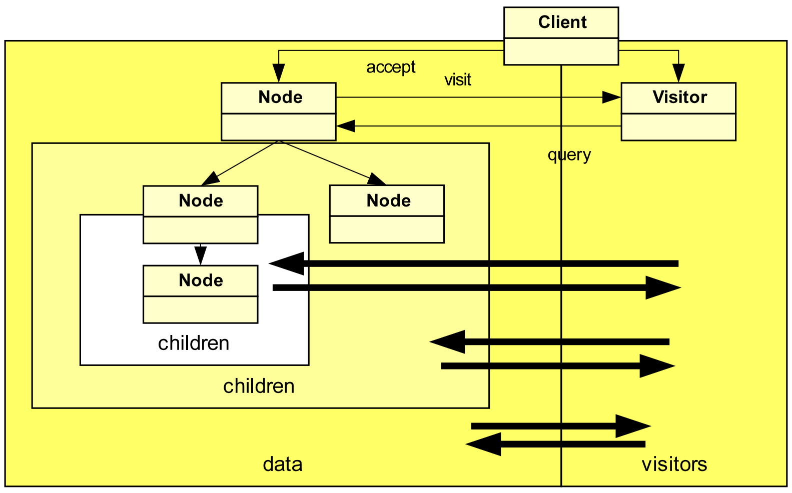


Figure 7.11: Deep ownership structure for the visitor pattern. Bold arrows represent links, solid rectangles represent private domains.

Example Implementation

The following code shows an example implementation of the visitor pattern with deep ownership structure. As the client does not have access rights to encapsulated nodes in the data structure, the example uses the previously proposed workaround for deep ownership in the composite pattern (6.3).

```
class Client {
       domain data;
       domain visitors;
       link data -> visitors;
       link visitors -> data;

       public void test() {

              //build up data structure
              final data Node<visitors> root = new Assignment<visitors>("=");

              int rootID = root.getID();
              root.attachChild(Node.VARIABLE, rootID, true, "a");
              int opID = root.attachChild(Node.OPERATION, rootID, false, "+");
              root.attachChild(Node.VARIABLE, opID, true, "b");
              root.attachChild(Node.VARIABLE, opID, false, "c");

              //create a visitor to visit the data structure
              visitors Visitor v = new Printing();
              v = null;
              root.accept(v);
       }

       public static void main(String[] args) {
              Client client = new Client();
              client.test();
       }

}

abstract class Visitor {
       public abstract void visitAssignment(lent Assignment<owner> a);
       public abstract void visitVariable(lent Variable<owner> v);
       public abstract void visitOperation(lent Operation<owner> o);
}
```

```
class Printing extends Visitor {

       public void visitAssignment(lent Assignment<owner> a) {
              System.out.println("visiting assignment: " + a.getName());
       }

       public void visitVariable(lent Variable<owner> v) {
              System.out.println("visiting variable: " + v.getName());
       }

       public void visitOperation(lent Operation<owner> o) {
              System.out.println("visiting operation: " + o.getName());
       }
}

abstract class Node<visitors>
       assumes visitors -> owner, owner -> visitors {

       final static int ASSIGNMENT = 0;
       final static int VARIABLE = 1;
       final static int OPERATION = 2;

       static int NextID=1;

       domain children;
       link visitors -> children;
       link children -> visitors;

       private shared String name;
       children Node<visitors> left;
       children Node<visitors> right;
       int id;

       public Node(shared String name) {
              this.name = name;
              id = NextID++;
       }

       public int getID() {
              return id;
       }

       public shared String getName() {
              return name;
       }

       public void accept(visitors Visitor v) {
              if(left != null) left.accept(v);
              if(right != null) right.accept(v);
       }

       private children Node<visitors> createNode(int type) {
              children Node<visitors> n;

              switch(type) {
                     case ASSIGNMENT:
                            n = new Assignment<visitors>(name);
                            break;

                     case VARIABLE:
                            n = new Variable<visitors>(name);
                            break;

                     case OPERATION:
                            n = new Operation<visitors>(name);
                            break;
```

```
                        default:
                                throw new RuntimeException("unsupported type");
                }
                return n;
        }

        public int attachChild(int type, int parentID, boolean attachLeft, String name) {
                int cID;

                if(parentID != id) {
                        //propagate operation to children
                        if(this.left!=null) {
                                cID = this.left.attachChild(type, parentID, attachLeft, name);
                                if(cID != 0) return cID;
                        }
                        if(this.right!=null) {
                                cID = this.right.attachChild(type, parentID, attachLeft, name);
                                if(cID != 0) return cID;
                        }
                        return 0;

                } else {
                        children Node<visitors> n = createNode(type);

                        if(attachLeft)
                                this.left = n;
                        else
                                this.right = n;

                        return n.getID();
                }
        }
}

class Assignment<visitors> extends Node<visitors> {

        public Assignment(String name) {
                super(name);
        }

        public void accept(visitors Visitor v) {
                v.visitAssignment(this);
                super.accept(v);
        }

}

class Variable<visitors> extends Node<visitors> {

        public Variable(String name) {
                super(name);
        }

        public void accept(visitors Visitor v) {
                v.visitVariable(this);
                super.accept(v);
        }
}

class Operation<visitors> extends Node<visitors> {

        public Operation(String name) {
                super(name);
        }

        public void accept(visitors Visitor v) {
            v.visitOperation(this);
            super.accept(v);
      }
}
```

Conclusion

The concept of ownership helps to improve a visitor pattern's implementation by restricting aliasing of the data structure elements from outside the client's context. The necessary representation exposure of the data elements, in order to allow the visitor to perform the required operations, is limited to the client's context and one does not have to worry about unwanted data modification by external objects.

Using deep ownership in the object structure requires either a mechanism to transfer ownership of the visitor along the data structure, or granted aliasing rights between all contexts holding data elements and the visitor's context, as shown with Ownership Domains.

7.4 Mediator

Intent

> ”Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.”[4]

Since object-oriented systems consist of an arbitrary number of objects and distribute their behavior among these objects, the resulting object structures tend to be strongly interconnected in order to fulfill their task. While assigning clear responsibilities to a single object facilitates reuse, a lot of interconnections tend to reduce the reuse-potential again since each (re-)used object requires the presence of all dependents.

The mediator pattern provides a mechanism to decouple direct connections between objects by introducing a mediator object that encapsulates the collective behavior and coordinates how the different objects, so-called colleagues, interact with each other.

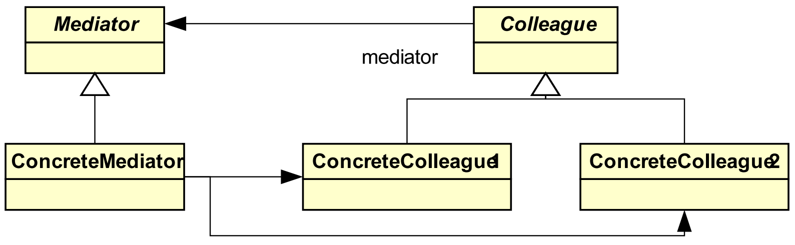
Instead of having each object communicate with all others, every object communicates only with the mediator and therefore commits only to the mediator's interface. This mechanism greatly reduces the number of interconnections in a system, making it easier to understand and facilitates reuse.

A well-known usage scenario of the mediator pattern is a dialog box in an application that has a graphical user interface. A dialog box usually consists of several dependent widgets. A state change in a widget can lead to multiple state changes in other widgets. Every widget is somehow dependent from the others, which results in many interconnections between the widget objects. In order to reduce the number of interconnections, all widgets only communicate with the dialog. When a widget changes state, the dialog is notified and all dependent widgets are adjusted by the dialog.

Another example of a mediator pattern is the *change manager* scenario, mentioned in the observer pattern (7.2) discussion: all event sources and event sinks register at the change manager instance, which plays the role of the mediator and handles the routing of events in the system.

[4][GHJV95] page 273

UML Diagram



Ownership Discussion

The mediator pattern enforces an architectural constraint: each involved object (a colleague) should only communicate with other colleagues through the mediator object. As a consequence, no direct references between colleagues must exist. By using the concept of ownership, we can express the pattern's architectural obligations with an ownership structure: each colleague should be able to reference the mediator object, but not other colleagues. The mediator should be able to reference all colleagues.

In the following discussion, we would like to distinguish two possible scenarios:

Low-level mediation This is the case when all involved objects are created in the same application scope and highly collaborate to provide a *unit of functionality.* In terms of ownership, we would like to declare the mediator as the owner of all colleagues, while each colleague is located in a separate ownership context declared by the mediator. Due to the different contexts, direct references between the colleagues are forbidden and only the mediator has the possibility to reference all colleagues. Hence, we can use the ownership system to statically prove a correct implementation of the pattern.

High-level mediation This is the case when mediation takes place on a higher-level logic between totally different objects, not tightly coupled in terms of functionality or creation order. For instance, in a layered architecture, presentation layer objects should only communicate with domain layer objects through a controller. The controller instance acts in the role of the mediator and establishes loose coupling between the two application layers. Loose coupling of application layers enables the possibility to exchange the implementation of a layer.

The two possible mediation scenarios do not differ greatly in terms of ownership. In both scenarios the used ownership system must provide the ability to let objects declare multiple contexts and to establish read/write references from an owned object to its owner. Another observation is that *a colleague* is not necessarily a single object, but rather a group of objects that should be decoupled from another group of objects by enforcing communication over a mediator instance. Hence, the term *colleague* perfectly matches the definition of an ownership context.

The only difference between the two scenarios is that in the second scenario, mediation takes place between many, in terms of functionality and creation order totally unrelated objects. Therefore, the used ownership system might require a mechanism of ownership transfer in order to reallocate created objects to the correct colleague context.

We will discuss feasibility of the desired structures under each ownership system.

Universe type system

The attempt to implement the desired ownership structure with the Universe type system leads to several problems. An owned object can only have a `readonly` reference to its owner, but the mediator needs a read/write reference to each colleague and vice versa. This implies that all objects should reside in the same context. However, each object in a context can freely reference all other objects in the same context, as outlined in Figure 7.12. The Universe type system does therefore not achieve to express the architectural obligations from the mediator pattern. Furthermore, a potential reallocation of objects is not supported due to the lack of ownership transfer.

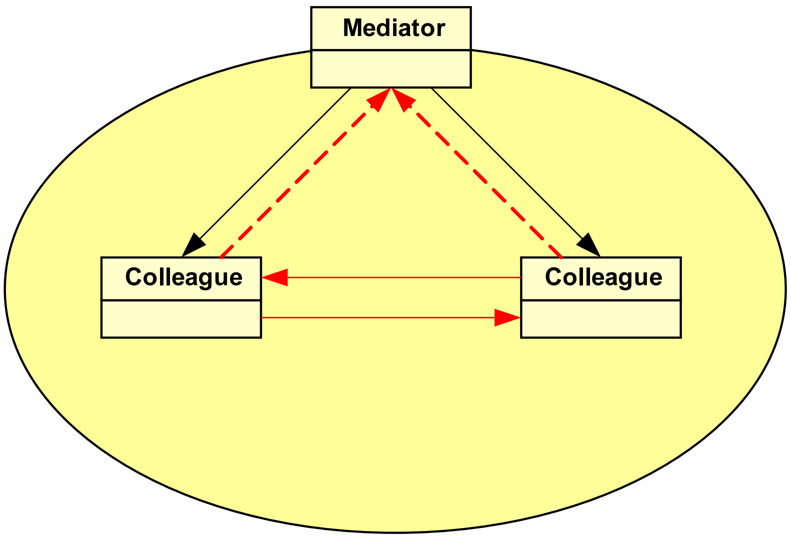


Figure 7.12: Universe type system ownership structure of the mediator pattern. `readonly` references are represented with dashed arrows. Problematic references that do not comply with the pattern's obligation are colored red. We notice two problems: with the mediator as the owner of the colleagues, the colleagues may only maintain `readonly` references to `Mediator`. On the other hand, an object does only have one ownership context in which all objects may freely reference each other. Therefore, references between colleagues are not forbidden.

Ownership Types

In contrast to the Universe type system, Ownership types allows all owned object to reference their owner. This enables us to declare the mediator as the owner of the colleagues and to establish references between the mediator and the colleagues and vice versa. However, if all colleagues are instantiated and owned by the mediator, they will reside in the same ownership context and may therefore freely reference each other.

A useful property of Ownership Types is its concept of *role separation*:

> ”Two different ownership types appearing in the same context are not compatible, regardless of the ensuing bindings.”[5]

If we consider another scenario where a client is responsible for instantiating all involved objects, the concept of role separation can still enhance a pattern implementation. The mediator instance is configured with all colleagues, but each colleague has, although residing in the same context, a different ownership type. Through this scenario, the programmer has static safety that the mediator does not confuse different colleagues or, in the worst case, enable colleagues to reference each other. Figure 7.13 illustrates the scenario from the mediator's point of view.

Unfortunately, role confusion can still happen in the client's scope as the declaration of multiple ownership contexts is not supported.

Example Implementation

The following example code illustrates the concept of role separation and shows how Ownership Types can enhance a mediator pattern implementation.

[5] [CPN98] page 13

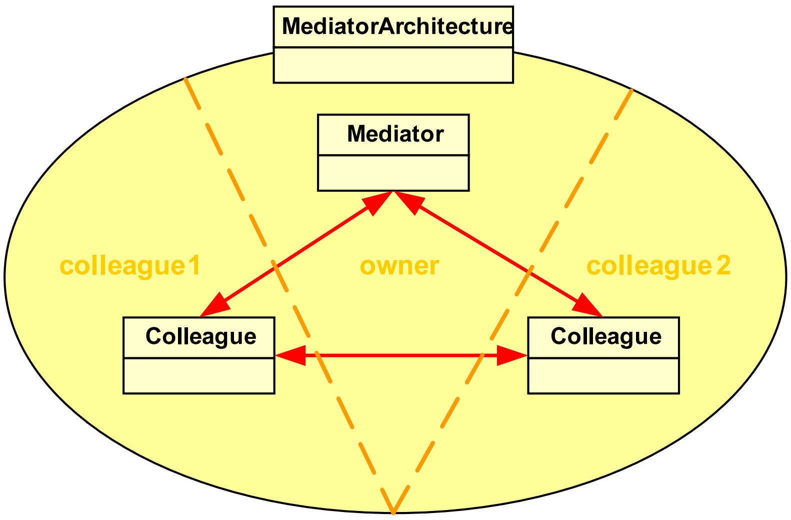


Figure 7.13: Ownership Types structure of the mediator pattern. References that do not comply with the pattern's obligations are colored red. The dashed lines through the ownership context symbolize the mediator's assumption that the colleagues have different ownership types, although residing in the very same context.

```
class MediatorArchitecure<owner> {

       Colleague<this> c1;
       Colleague<this> c2;
       Mediator<this, this, this> mediator;

       /* Setting up the mediator architecture is done by allocating all objects
        * with owner = this.
        */
       public void setupArchitecture() {
              c1 = new Colleague<this>();
              c2 = new Colleague<this>();
              mediator = new Mediator<this, this, this>(c1, c2);
       }

       public void confuseRoles() {
              /* Due to the missing support for different ownership context the following wrong
               * statement still executes:
               */
              c1 = c2;
       }

       public void enableDirectReferences() {
              /* Due to the missing support for different ownership context the following wrong
               * statement still executes:
               */
              c1.forbiddenReference(c2);
       }
}

class Mediator<owner, firstColleague, secondColleague> {

       Colleague<firstColleague> first;
       Colleague<secondColleague> second;

       public Mediator(Colleague<firstColleague> c1, Colleague<secondColleague> c2) {
              first = c1;
              second = c2;
```

```
	}

	public void confuseRoles() {
		/* The following statement is detected as wrong since the mediator assumes that the
		 * colleagues reside in a different ownership context:
		 */
		first = second;
	}

	public void enableDirectReferences() {
		/* The following statement is detected as wrong since the method declaration requires
		 * the parameter to be in the same ownership context, but the mediator assumes
		 * c1 to be in a different context than c2:
		 */
		first.forbiddenReference(second);
	}
}


class Colleague<owner> {

	public void forbiddenReference(Colleague<owner> anotherColleague) {
		/* A potential call to this method is forbidden by the pattern's constraint that
		 * colleagues should not directly communicate with each other.
		 */
	}
}
```

Ownership Domains

The only concept that achieves to express all architectural constraints from the mediator pattern in an ownership structure is Ownership Domains. Not only bidirectional references between an owner and its owned objects can be established by linking the owned context with the owner context, an object can also declare multiple ownership contexts and define access rights between them.

A possible scenario, shown in Figure 7.14, is a client that allocates the mediator and all colleagues in separately declared contexts and establishes the necessary links between the contexts. Each colleague context should be linked with the mediator context and the mediator context should be linked with all colleague contexts.

As another scenario, shown in Figure 7.15, the mediator is in charge of creating all colleagues in separate contexts. In order to let the colleagues also access the mediator, each context is linked with the mediator's owner context.

Compared to the other concepts, Ownership Domains offers the best flexibility to express architectural constraints, which was also one of the main goals of the system's designers.

Example Implementation

The following example implementation of the first scenario, where a client allocates the architecture, is adapted from [AC04] and presented in Figure 7.14.

```
class MediatorArchitecture {
	domain colleague1, colleague2, colleague3;
	domain mediator;

	link colleague1 -> mediator, colleague2 -> mediator, colleague3 -> mediator;
	link mediator -> colleague1, mediator -> colleague2, mediator -> colleague3;
}
```

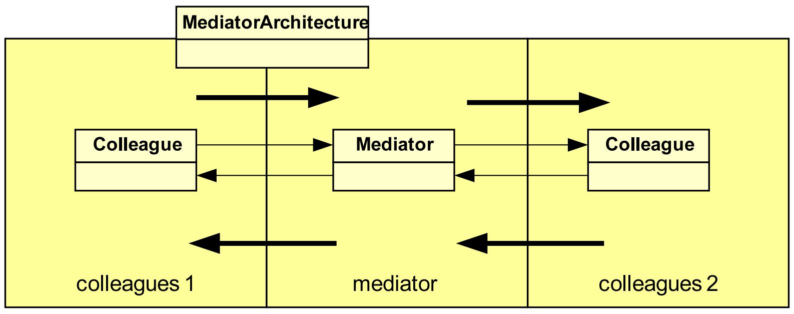


Figure 7.14: Ownership Domains structure of the mediator pattern where a client allocates the mediator architecture. Bold arrows represent context links and solid rectangles represent private contexts. The mediator and all colleagues are located in separate domains and linked together, according to the pattern's obligations.

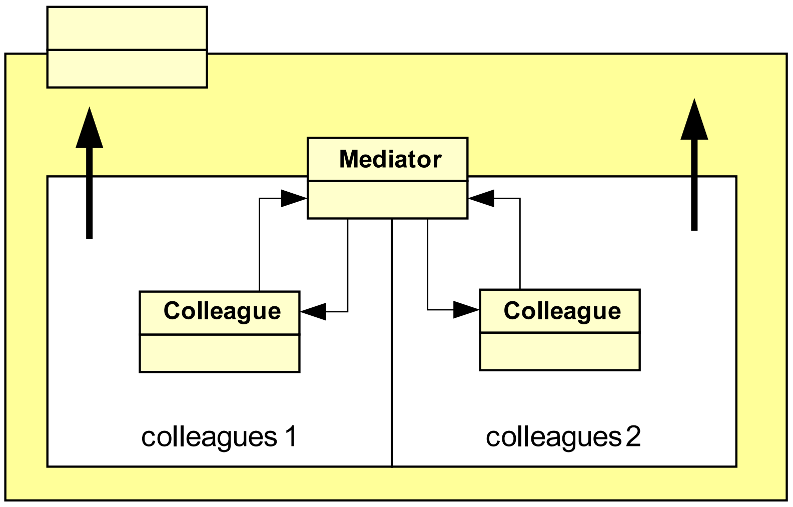


Figure 7.15: Ownership Domains structure of the mediator pattern where the mediator is the owner of all colleagues, residing in different contexts. Bold arrows illustrate links, solid rectangles private contexts. All declared colleague domains are linked with the mediator's owner domain in order to allow `Colleague` objects to access the `Mediator` instance.

Conclusion

Ownership systems can be really useful to express architectural constraints, like the mediator pattern imposes. By modeling these constraints with an ownership structure, correctness of the pattern can be statically checked and ensured. Only Ownership Domains achieves to express a satisfying ownership structure. The Universe type system lacks the ability to establish bidirectional read/write references between an owner and its owned objects, and in both systems, i.e. the Universe type system and Ownership Types, an owner cannot declare multiple ownership contexts and control aliasing between these contexts. However, Ownership Types' concept of role separation ensures at least a correct behavior of the mediator class.

7.5 State

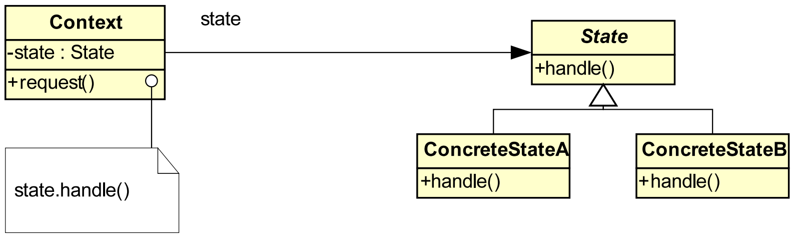
Intent

> "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class."[6]

In an application, some objects may change their behavior at run-time when a state change occurs. For instance, a `TCPConnection` object will respond differently to a `send()` operation, depending on the connection's state (established, closed). Different behaviors for different internal states lead to large multi-part conditional statements in the operations' implementation, which is hard to read and maintain. The state pattern therefore refactors the state-dependent operations into a `State` object and represents different object states by different concrete subclasses of `State`. The client only deals with the `Context` instance, that forwards all state-specific operations to the `State` object.

Since the term *context* always refers to an ownership context and this pattern declares a `Context` class, we will consequently write *ownership context* when discussing ownership or `Context` when referring to the pattern's class in order to avoid confusion.

UML Diagram



Ownership Discussion

When reviewing the state pattern we notice that `Context` and `State` are tightly coupled and that they should act as a single instance in front of clients. The concept of ownership can help us emphasize this property: `Context` should be the owner of `State`. Implementation details should be hidden from the client and therefore no direct references to the `State` object should be passed to the client. Ownership can help us to statically verify this property, a great benefit for the pattern's implementation.

Depending on the concrete implementation of the pattern, several requirements for the used ownership system arise. If the client needs to configure the `Context` instance with a state object directly, the ownership system needs the ability to transfer ownership of the `State` instance from the client to `Context`. However, this implementation scenario can be avoided by defining an

[6] [GHJV95] page 305

implicit initial state and after all, the client should not have the responsibility to allocate State objects since this is not a concern of the client at all.

In another implementation scenario, the State objects are implemented as shared flyweights (6.6) and therefore an encapsulation by a single Context instance does not work anymore. Each Context should rather have full access rights to the shared states. The used ownership system should provide a mechanism to share certain objects by providing alias rights for a carefully chosen group of other objects, not necessarily residing in the same ownership context of course.

Some operations require the Context to be passed along in order to adjust the Context's state. A read/write reference from the state to the Context might therefore be necessary.

Universe type system

When implementing the desired ownership structure with the Universe type system, we suggest the following approach: Context creates the initial ConcreteState as rep and forwards all state dependent requests to the state object. Upon a state change, Context allocates the new state object as rep and updates its state variable. However, as illustrated in Figure 7.16, an encapsulated state may only have a readonly reference to its owner, namely the Context. Hence, operations that modify the Context cannot be implemented correctly and moreover, even though the sequent state can be allocated by the current state, only the Context can actually configure itself with a new state. Due to the lack of a shared ownership context, a typing of the scenario where the different state objects are shared flyweights also fails, unless only pure operations are called on the state object.

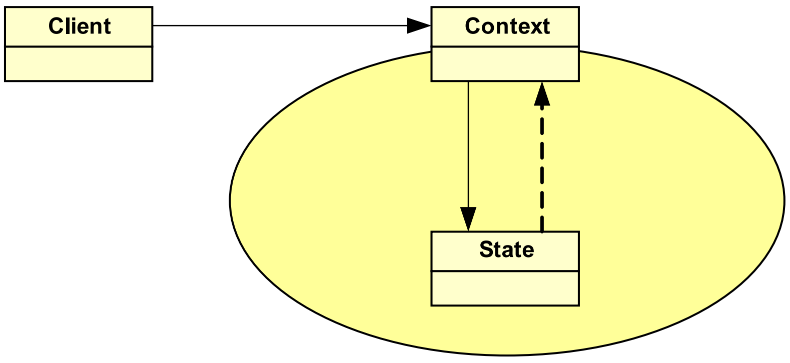


Figure 7.16: Ownership structure of the state pattern. Dashed arrows represent readonly references. The State object may only maintain a readonly reference to Context which leads to problems if a state dependent operation needs to modify the surrounding context.

Example Implementation

```
class Connection {
        rep State state;

        public pure Connection() {
                state = new rep Unconnected();
        }

        public void connect() {
                state = state.connect();
        }

        public pure readonly String getStatus() {
                return state.getStatus();
        }
}
```

```
abstract class State {
        protected String s;

        public pure readonly String getStatus() {
                return s;
        }

        public abstract State connect();
}

class Unconnected extends State {

        public pure Unconnected() {
                s = "not connected";
        }

        public State connect() {
                return new peer Connected();
        }
}

class Connected extends State {

        public pure Connected() {
                s = "connected";
        }

        public State connect() {
                System.err.println("already connected");
                return this;
        }
}

class Client {

        public void test() {
                rep Connection conn = new rep Connection();
                System.out.println(conn.getStatus());
                conn.connect();
                System.out.println(conn.getStatus());
        }

        public static void main(String[] args) {
                Client c = new Client();
                c.test();
        }
}
```

Ownership Types

An implementation of the desired ownership structure with Ownership Types is straight-forward and without problems: `Context` just passes itself as the owner to the `State` object upon instantiation. In contrast to the Universe type system, the `State` instance may also perform updates on the context, allowing us to assign state change responsibilities to `State` or `Context`.

The scenario where the state objects are shared is not fully supported: on the one hand, Ownership Types provides a global ownership context and each object can access directly owned objects of its ancestors. Depending on the established ownership structure, we can say that there is the possibility to share certain objects. On the other hand, we must also point out that the concept of sharing is not optimal (yet) and that one could think of shared ownership contexts only accessible by objects from a certain type (such as `Context`).

Nevertheless, Ownership Types is capable to provide a decent ownership typing for the state pattern.

Ownership Domains

When applying ownership to the state pattern using Ownership Domains, `Context` simply spans an ownership context `states` holding the `State` instance. Additionally, to allow `State` to access `Context`, we can declare an up-link from the `states` ownership context to the ownership context, holding the `Context` instance. Figure 7.17 outlines the proposed structure.

Ownership Domains is even able to provide support for shared state objects by following the implementation approach from the flyweight pattern (6.6). All shared `State` instances are allocated and owned by a factory. One simply has to ensure access rights from all `Context` objects to the ownership context, holding the shared states, through links.

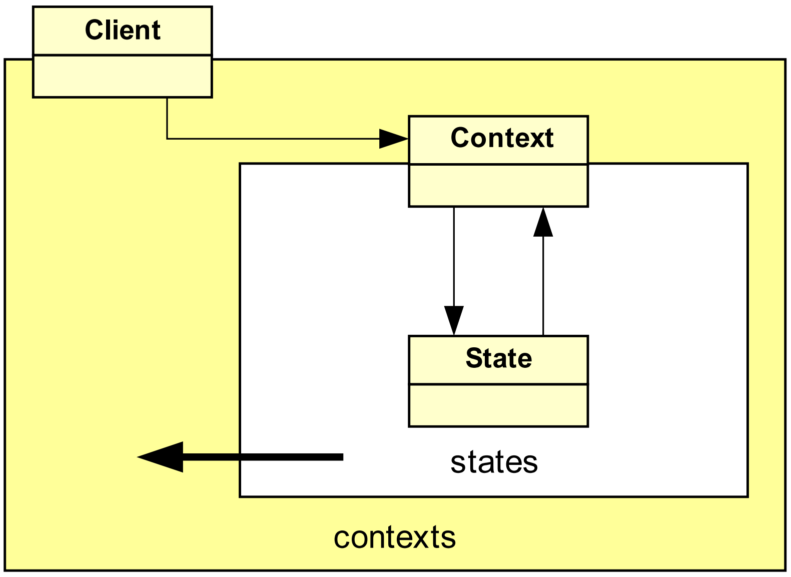


Figure 7.17: Ownership structure with Ownership Domains. Bold arrows represent links between contexts while solid rectangles represent private contexts.

Example Implementation

```
class Connection {
       domain states;
       link states -> owner; //uplink so that state can modify context

       states State state;

       public Connection() {
              state = new Unconnected();
       }

       public void connect() {
              state.connect(this);
       }

       public shared String getStatus() {
              return state.getStatus();
       }
}

abstract class State {
       protected shared String s;

       public shared String getStatus() {
```

```
		return s;
	}

	public abstract void connect(final lent Connection c);
}

class Unconnected extends State {

	public Unconnected() {
		s = "not connected";
	}

	public void connect(final lent Connection c) {
		c.states State s = new Connected();
		c.state = s;
	}
}

class Connected extends State {

	public Connected() {
		s = "connected";
	}

	public void connect(final lent Connection c) {
		//do nothing
	}
}

class Client {
	domain connections;

	public void test() {
		connections Connection conn = new Connection();
		conn.connect();
	}

	public static void main(String[] args) {
		shared Client c = new Client();
		c.test();
	}
}
```

Conclusion

The concept of ownership greatly enhances an implementation of the state pattern by ensuring that the implementation is fully hidden from the client. The client just operates on the `Context` object, not being able to receive a direct reference to the `State` object. A comparison of the different ownership systems shows that the Universe type system is too strict as it does not allow modifications of the `Context` (the owner) by the different states (owned objects). Furthermore, the sharing of `State` objects is not an option due to lack of shared ownership contexts. A pattern implementation in conjunction with Ownership Domains or Ownership Types works fine. Even the sharing of `State` objects can be implemented, easily when using Ownership Domains, strongly depending on the situation when using Ownership Types.

7.6 Strategy

Intent

> ”Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.”[7]

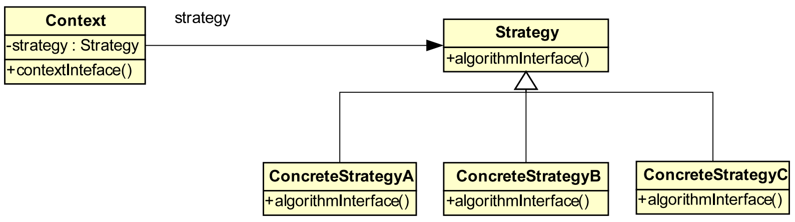
[7] [GHJV95] page 315

The strategy pattern should be used when several related classes only differ in their behavior and we want to shield the client from algorithmic details. Inheritance can help us factor out the common behavior of all concrete strategies into the abstract `Strategy` class. Furthermore, the strategy pattern helps to eliminate conditional statements by encapsulating each conditional block in a separate subclass.

The `Context` instance is configured with a concrete strategy by either itself, a client of the context, or an abstract factory (5.1). However, `Context` only operates on the abstract strategy class, remaining implementation independent from any concrete strategy.

A problem of the strategy pattern might be the increased number of objects in the system. One can counter this by using the flyweight pattern (6.6) in order to share all strategy objects that are stateless. Strategy objects may also be optional and if `Context` is not configured with a certain strategy it could use a default routine.

UML Diagram



Ownership Discussion

Similar to the state pattern (7.5), ownership could be used to make the `Context` and the configured strategy act as a single instance. `Context` decides, based on its state, which strategy to use and a client of the context is totally shielded from any algorithmic details. In case of an abstract factory that allocates the strategy object, the ownership system has to provide a mechanism to transfer ownership of the allocated `Strategy` to the `Context` instance. An encapsulation of `Strategy` by `Context` only makes sense if the strategy has a state. Stateless strategies may easily be shared, as in the flyweight pattern (6.6) and in this case, the ownership system has to ensure that all `Context` instances have alias rights to the shared `Strategy` instances. Operations defined in the concrete strategy are likely to be non-pure and therefore full access rights have to be provided. Obviously, having static strategy methods would guarantee the necessary access rights, but due to the used polymorphism, the methods cannot be declared static. Hence, the ownership system must provide a notion of shared ownership contexts in order to let all `Context` instances access the shared strategies.

Since the desired ownership structure does not differ from the state pattern's ownership structure, all system specific implementation issues remain and we will omit a detailed discussion for each system under review.

Conclusion

An encapsulation of the `Strategy` object by the `Context` can help us emphasize the pattern's aspect that both, the `Context` and the strategy, should act as a single unit and by this shield the client from algorithmic details. If the strategy is stateless, strategy objects can easily be shared and the ownership system has to provide a notion of shared ownership contexts. As already discussed in the state pattern's discussion (7.5), the Universe type system does currently not support shared contexts while in Ownership Types it strongly depends on the ownership structure. Ownership Domains supports this feature by linking all contexts, that require access to the shared context.

If the algorithm has a state, alias control should be applied and each individual strategy instance should thus be owned by its `Context` instance.

7.7 Interpreter

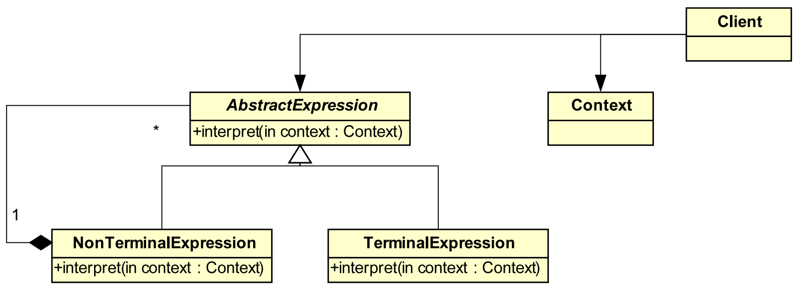
Intent

> "Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language."[8]

Defining grammars and interpreting sentences in the specified language is quite a common scenario in applications. The interpreter pattern defines how to build an object model for a particular grammar and how to interpret the resulting abstract syntax trees. Basically, each grammar consists of terminal- and nonterminal-expressions where each expression is represented by its own class, providing an `interpret(Context c)` method. The context is usually a `String` containing the input and information on what has already been interpreted. Every node in the abstract syntax tree modifies the context upon interpretation in order to store the state of the interpreter.

The interpreter pattern makes it easy to create and extend simple grammars but complex grammars are hard to maintain, due to the complicated resulting class structure. The client, or a parser, is responsible for creating the abstract syntax tree which is implemented as a composite pattern (6.3). Usually, interpretation is done by the visitor pattern (7.3) which enables multiple interpretations of the same tree. Because terminals are immutable, they can be shared easily and it makes sense to use the flyweight pattern (6.6) when many occurrences of terminal symbols exist.

UML Diagram


Client
AbstractExpression
+interpret(in context : Context)
Context
*
1
NonTerminalExpression
+interpret(in context : Context)
TerminalExpression
+interpret(in context : Context)

Ownership Discussion

When discussing ownership in the interpreter pattern, one has to consider the use of other patterns (composite, visitor and flyweight) and their ownership benefits and drawbacks. We therefore distinguish several possible implementation and ownership scenarios:

The abstract syntax tree makes use of a deep ownership structure The resulting benefits of a deep ownership structure are negligible (no need for caching and only a few meaningful invariants) compared to the added complexity and requirements for the used ownership system. The use of deep ownership in the composite pattern is likely to require ownership transfer in order to intuitively build up the data structure and provide support for possible tree transformations for optimization reasons. If a visitor is used to perform the interpretation of the syntax tree, access rights between the visitor instance and each node are needed. In a deep ownership structure, this will also lead to the requirement of ownership transfer.

The ownership system has to make sure that read/write access permissions to the `Context` instance are established from each node. This can either be done by passing ownership of `Context` along the composite structure upon interpretation, or to allocate `Context` in a shared ownership context, accessible by all nodes. A graphical illustration of this ownership structure is presented in Figure 7.18.

[8] [GHJV95] page 243

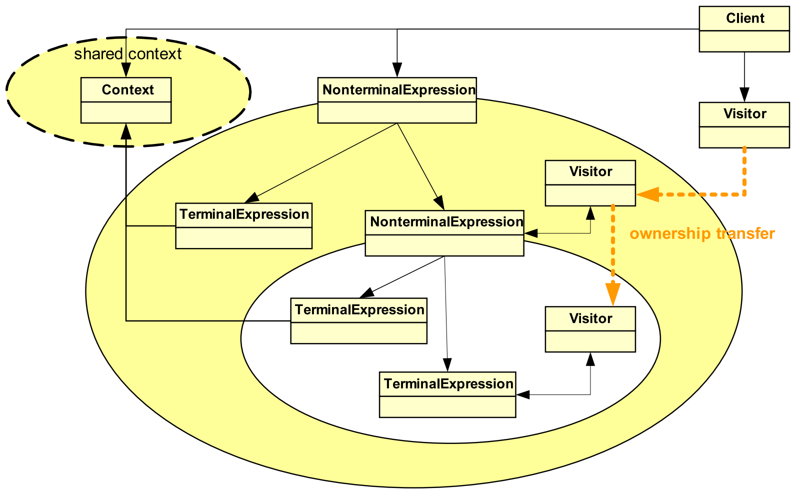


Figure 7.18: Graphical illustration of the interpreter pattern with a deep ownership structure. Notice the *shared ownership context* that provides access rights to the `Context` instance from all tree nodes. The visitor instance needs to traverse all elements in the tree and therefore gets reallocated to the different ownership contexts using an ownership transfer mechanism.

We can conclude that deep ownership is not a desired ownership structure, considering the resulting complexity and ownership system requirements.

Terminals are shared and owned by the factory Usually, when sentences with many terminals are parsed, it makes sense to use the flyweight pattern (6.6) for the terminal symbols in order to reduce the number of objects in the system. This clearly poses the question if the desired ownership structure of the flyweight pattern, where the factory is the owner of all shared flyweights (6.6), could improve an implementation of the interpreter pattern.

As discussed in the flyweight pattern discussion, an ownership system with support for read-only references can greatly enhance the pattern's implementation by ensuring that the shared instances do not get modified after creation. In the interpreter scenario however, the shared `TerminalExpression`s need to be referencable by `NonTerminalExpression`s and by the `Visitor` instance in a read/write manner. Therefore, providing only readonly references to the terminals is not an option. Furthermore, all terminals need the right to modify the `Context` instance upon interpretation.

We recognize that this ownership scenario requires support for shared contexts: on the one hand it must be ensured that the `Context` instance is accessible from all nodes in the abstract syntax tree. On the other hand, access rights from the visitor instance and all nodes to the shared terminals must be granted. A graphical illustration of this ownership structure is presented in Figure 7.19.

We can conclude that the flyweight pattern can be used in conjunction with the interpreter pattern to reduce the number of objects for an abstract syntax tree. However, applying the proposed flyweight ownership structure does not work if only readonly references are provided to the shared instances. Since many objects need read/write references to the terminals anyway, an encapsulation simply does not make sense.

The client encapsulates all involved objects Obviously, previously proposed ownership structures for a single pattern cannot be applied offhand to collaborations of multiple patterns. Due to the strong interconnection of all involved objects, we propose to encapsulate all objects in the same, namely the client's, context. Thereby, necessary reference rights between nodes within the abstract syntax tree or between nodes and the `Context` instance are established and no further requirements for the used ownership system, such as a mechanism to transfer ownership or a no-

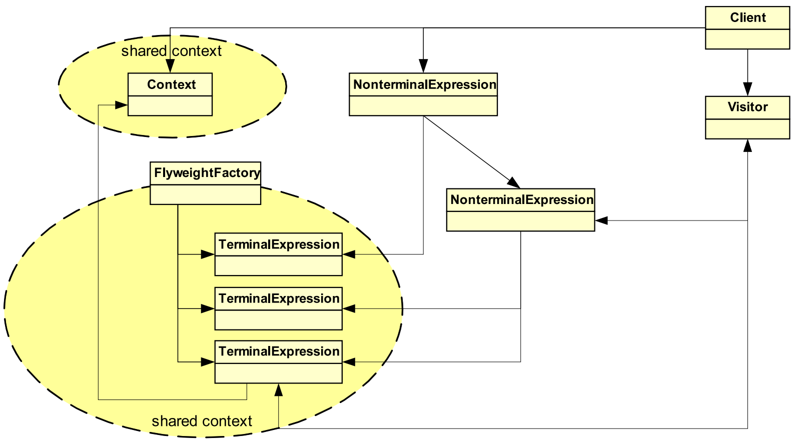


Figure 7.19: Graphical illustration of the interpreter pattern that shares the terminal flyweight instances. Notice the *shared ownership contexts* holding the `Context` instance and all shared flyweights.

tion of shared ownership contexts, arise. An encapsulation of all involved objects has the benefit that one can reason about an interpreter implementation locally, without considering the rest of the application as one does not have to worry about modifications from outside. A graphical illustration of this ownership structure is presented in Figure 7.20.

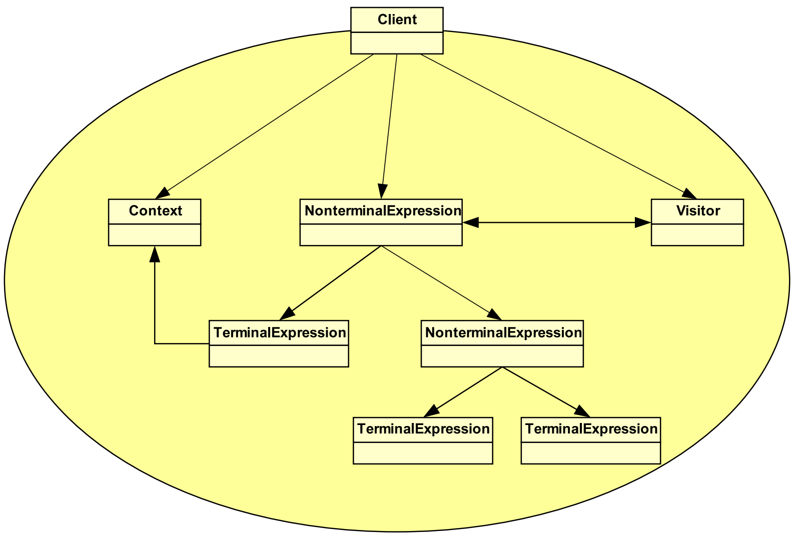


Figure 7.20: Graphical illustration of the interpreter pattern where the client encapsulates everything. This is the desired ownership structure.

Universe type system

Considering the requirements imposed on the ownership system by each possible ownership scenario, we recognize that the Universe type system insufficiently supports a deep ownership structure for the abstract syntax tree due to the lack of ownership transfer. Also an implementation of the second ownership scenario is not possible since the system has no support for shared contexts.

Only the third scenario where all objects are encapsulated in the client's ownership context works fine and this is also the desired ownership structure.

Ownership Types

Regarding the different proposed ownership scenarios, Ownership Types fails to provide a correct typing of the first scenario with a deep ownership structure, due to lack of an ownership transfer mechanism as well. However, since all ancestor objects with their directly owned objects are accessible, each node could still access the `Context` object or the `Visitor` instance. The problem is that neither the visitor has access to the encapsulated tree nodes, nor is an intuitive creation or transformation of the tree possible.

The typing of the second scenario fails because Ownership Types has no real support for shared ownership contexts, but only looser aliasing restrictions, compared to the other systems.

An ownership typing of the third, desired, scenario works fine and needs no further explanation.

Ownership Domains

As already pointed out in the composite pattern discussion (6.3), only Ownership Domains is able to provide a typing for a deep ownership structure in the composite pattern, together with a visitor. All ownership contexts are just recursively, bidirectionally linked with the ownership context that holds the visitor instance. Access rights to the `Context` instance can be ensured from all nodes in the tree. Only an intuitive creation and transformation of the tree is not possible since also Ownership Domains provides no support for ownership transfer.

The second scenario is also implementable with Ownership Domains as links can be established from the ownership contexts holding the visitor and the tree nodes to the ownership context holding the shared terminals. However, as already explained in the general ownership discussion, such an encapsulation makes no sense and one should prefer the third proposed ownership structure where the client encapsulates everything.

The third scenario is well supported and does not need further explanation.

Conclusion

Ownership does not achieve to greatly enhance the interpreter pattern's implementation. The only gained benefit from the desired structure, where the client is the owner of all involved objects, is that it enables local reasoning about the pattern's implementation. Interestingly we also find out that as soon as multiple patterns are combined, the desired ownership structure can greatly change and the ownership propositions for a single pattern may not be feasible anymore. Due to the triviality of the desired ownership structure, there are no greater differences between the reviewed systems.

7.8 Memento

Intent

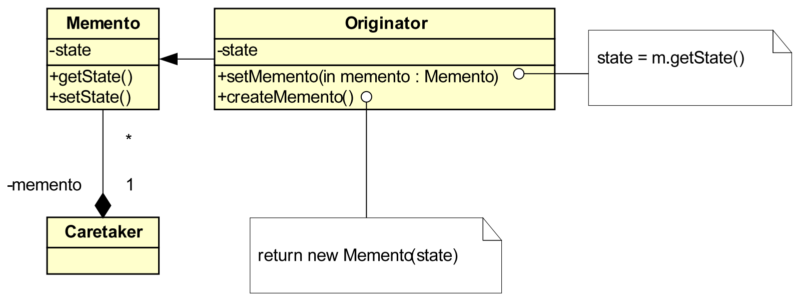
> ”Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.”[9]

Occasionally, objects need to save their current state in order to be able to perform a possible rollback or undo operation later on. The problem with externalizing an object's state is that objects normally encapsulate their behavior, making it impossible for other objects to read and store the state. The memento pattern solves this problem by introducing a `Memento` class that stores another object's, the so-called *originator*'s, state. Memento objects are created by the

[9] [GHJV95] page 283

originator and the originator can write its state to a given memento. The memento is responsible for only allowing the originator to access its stored information, making it desirably impossible for other objects to modify the memento. After creation, mementos can be passed to so-called *caretakers* until they are needed by their originator again for a state restore. It is to say that the memento pattern can be quite expensive since possibly large amounts of data may be copied.

UML Diagram



Ownership Discussion

Java's concept of inner classes perfectly fulfills the pattern's obligations. First, an inner class is associated with its surrounding class for lifetime and, second, has full access to the outer class' properties and private methods. This allows the memento, implemented as an inner class of the originator, to access the originator's representation while providing a narrow interface, that cannot be exploited, by other objects. All methods and fields of the memento can be private since they only need to be consulted by the originator. Let us take a look at the suggested implementation:

```java
package memento;

import java.util.ArrayList;

public class Originator {
       private String name;
       private ArrayList<String> collection;

       public Originator(String name, ArrayList<String> collection) {
              this.setName(name);
              this.setCollection(collection);
       }

       public void addItem(String item) {
              collection.add(item);
       }

       public void setMemento(Memento memento) {
              memento.getState();
       }

       /* Return the created memento based on the current object state to
        * the caller, the caretaker.
        */
       public Memento createMemento() {
              Memento memento = new Memento();
              memento.setState();
              return memento;
       }

       /* The memento is implemented as an inner class of the originator.
        * In that way, the memento has full access rights to the originator's
        * state while maintaining a very narrow interface to all foreign objects.
        */
        class Memento {
                private String mementoName;
                private ArrayList<String> mementoCollection;

                //writes the originator's state into the memento
                private void setState() {
                        //Strings are immutable and can therefore be copied directly:
                        mementoName = Originator.this.getName();

                        /* Since the collection is not immutable we have to allocate a new ArrayList.
                         * If the elements in the collection are not immutable, one has to provide
                         * a deep clone for the memento.
                         */
                        mementoCollection = new ArrayList<String>(Originator.this.getCollection());
                }

                /* Restores the originator's state by cloning(!) the memento's internals
                 * so that multiple rollbacks can performed without setting the
                 * memento's state again.
                 */
                private void getState() {
                        Originator.this.setName(mementoName);
                        Originator.this.setCollection((ArrayList<String>)mementoCollection.clone());
                }
        }

        /* All getter and setter-methods are private so that the object
         * does not expose its state:
         */
        private ArrayList<String> getCollection() {
                return collection;
        }

        private void setCollection(ArrayList<String> collection) {
                this.collection = collection;
        }

        private String getName() {
                return name;
        }

        private void setName(String name) {
                this.name = name;
        }
}
```

We notice a deficiency: the originator provides a method `setMemento(Memento)` taking a memento instance from the caretaker in order to perform a rollback. The rollback operation forwards the request to the supplied memento by calling `getState()` which sets the properties of the originator according to the stored values in the memento. Because an inner class, like the memento, has exactly one outer class for life-time, the memento instance automatically sets all properties of its associated originator. This means that, implicitly, `originator.setMemento(memento)` requires `originator` to be the associated outer class of `memento`. Otherwise, another originator's state, the actual associated outer class of `memento`, is set. In other words, the receiver of the `setMemento(Memento)` call does not matter, because the supplied memento already determines the operation's target. Such a behavior can be very misleading and may result in bugs that are very hard to track.

The concept of ownership can help us to verify this precondition either statically, or by raising a run-time exception upon a wrong ownership type cast.

If the originator is the owner of its created mementos, the originator remains the only object that can call non-pure methods on the memento. Hence, no other object can call a foreign memento's `getState()` method to perform a state rollback on a foreign originator.

When implementing the memento pattern, it must be ensured that the memento's fields do not leak. Considering the example implementation above we notice that the memento's `getState()` method sets a clone of the collection on the originator. If the memento would not clone its fields upon a rollback request, the originator would automatically modify the memento's fields, preventing subsequent rollbacks. By declaring the memento owner of its fields, leaking is detected and reported by the ownership type system.

The optimal ownership system thus provides readonly references for the caretaker, while the originator encapsulates all its mementos in its ownership context. All mementos should encapsulate all fields in their context in order to prevent leaking.

We can thus conclude that an implementation of the memento pattern using inner classes is very safe and true to original. Nevertheless, ownership enhances the pattern implementation by ensuring that no memento/originator confusion happens and the memento's fields do not leak.

Universe type system

The Universe type system prevents a confusion of memento/originator-pairs by raising a type-cast exception:

```
public class Originator {

	public void setMemento(readonly Memento memento) {
		((rep Memento) memento).getState();
	}

	//other methods omitted
}
```

Since `setMemento()` is a public method called by the caretaker, method parameters may not be `rep` and we declare the memento as `readonly`. In the method body, a downcast from `readonly` to `rep` is made and if the supplied memento is not owned by the originator, a run-time exception is thrown.

Ownership Types

An encapsulation of the memento by its originator is of course possible with Ownership Types. However, the caretaker may not reference the owned memento anymore. The only exception to this rule is when the caretaker is also implemented as an inner class of the originator. If such an inner class implementation is desired, a confusion of memento/originator-pairs is also prevented.

Ownership Domains

Ownership Domains even provides static verification that no confusion of memento/originator-pairs happens. Unfortunately, we experienced problems with the checker tool in the code listed below. Nevertheless, an implementation proposition is given in the following.

```
class Originator<caretakers> assumes caretakers -> owner {
	domain mementos;
	link mementos -> owner;
	link caretakers -> mementos;

	shared String name;

	public void setMemento(mementos Memento memento) {
		memento.getState();
	}

	public mementos Memento createMemento() {
		mementos Memento memento = new Memento();
		memento.setState();
		return memento;
	}
```

```
        private shared String getName() {
                return name;
        }

        private void setName(shared String name) {
                this.name = name;
        }

        class Memento {
                shared String mementoName;

                private void setState() {
                        mementoName = getName();
                }

                private void getState() {
                        setName(mementoName);
                }
        }
}

class Caretaker {
        domain originators;

        public void testMemento() {
                final originators Originator<owner> o = new Originator<owner>();
                o.mementos Originator.Memento memento = o.createMemento();
                o.setMemento(memento);

                final originators Originator<owner> o2 = new Originator<owner>();
                //o2.setMemento(memento);    //static check fails since o.mementos != o2.mementos!
        }

        public static void main(String[] args) {
                Test t = new Test();
                t.testMemento();
        }
}
```

Conclusion

The memento pattern can be very nicely implemented using Java's concept of inner classes. Nevertheless, ownership still provides added value by ensuring that no confusion of memento/originator-pairs exist. Moreover, leaking of the memento's fields can prevent multiple subsequent rollbacks on the same memento. Thus, an ownership type system can help to detect and report the occurrence of leaking.

Comparing the different systems, we recognize that the Universe type system provides an excellent implementation due to its concept of readonly references. Due to the encapsulation of the memento, neither the caretaker nor any other object outside the originator's context is able to modify the memento. The disadvantage of the Universe type system is that a potential memento confusion is only detected at run-time.

Ownership Types is too restricting since, if not implemented as an inner class, the caretaker may not maintain a reference to the memento it should take care of.

Ownership Domains can statically ensure the absence of a memento confusion, but due to the lack of readonly references, the caretaker can freely reference mementos and thus a carefully designed interface of the memento class is necessary.

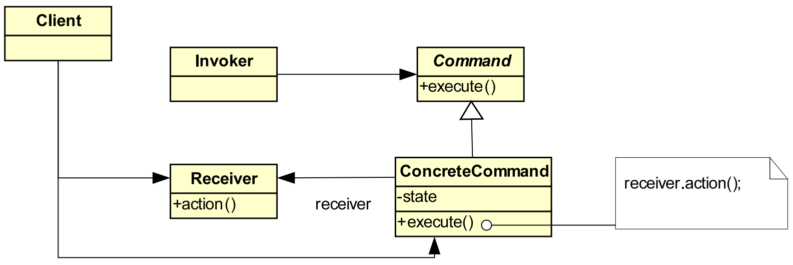
7.9 Command

Intent

> ”Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.”[10]

One of the main goals of the command pattern is to decouple the object that invokes an operation from the one that actually has the knowledge how to execute it. Thereby, the concrete action that needs to be carried out can be configured dynamically and changed at run-time. Furthermore, command objects can be shared which is especially useful when certain commands may be triggered by several user interface elements like menu items, buttons, etc. Some major benefit gained by the introduction of command objects is the possibility to support undoing, redoing and chaining operations easily.

UML Diagram



Ownership Discussion

When considering the interaction diagram of the command pattern in Figure 7.21 we notice that

- the client is responsible for creating the command object and configure it with a receiver.
- the invoker maintains a reference to the command and is the caller of `execute()`.
- the command object performs its operations on the configured receiver.

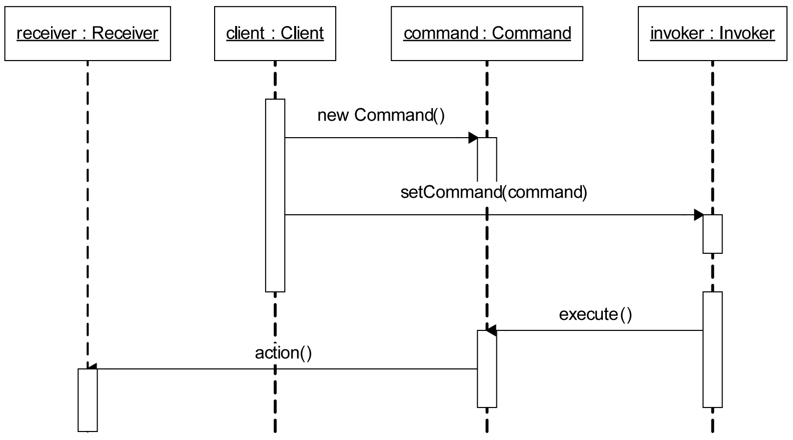


Figure 7.21: Interaction Diagram of the Command Pattern

[10] [GHJV95] page 233

This setup has some implications on the possible ownership structure: the invoker needs to maintain a read/write reference to the command and the command needs to maintain a read/write reference to the receiver. In some situations, command objects may be shared between different invokers. Optimally, ownership can help us enforcing a clear interaction structure. For instance, by encapsulating the command object in the invoker's context, we can ensure that commands are only triggered by dedicated invokers and not by other objects that snatch a reference. Such an ownership scenario requires support for ownership transfer though, because the client is responsible for creating the command objects and configure the invokers with them.

When commands are shared, the ownership system needs to provide a shared context containing all commands, ideally only accessible by the invokers. Furthermore, the ownership system needs to provide the necessary alias rights between an invoker, a command and a receiver.

Let us consider the standard pattern usage scenario, where the client is the application itself, invokers are very likely to be GUI elements and receivers are likely to be the domain layer controller, or any objects from the domain layer itself. Consequently, an implementation of the command pattern implies alias rights between different objects, belonging to different application layers.

We will discuss possible ownership structures and how well the reviewed ownership systems cope with this situation in the according sections. For a more detailed discussion of the command pattern in conjunction with Swing events refer to Chapter 9.

Universe type system

In case of the Universe type system we mainly have two possible setups that satisfy the above mentioned scenario:

All involved objects are peer to each other This is the standard case and the same as when no ownership is in place at all. It is therefore not worth further discussing.

The client is the owner of all other involved objects As shown in Figure 7.22, with such an ownership scenario, we would at least have the benefit that we do not have to consider objects outside the client's context when reasoning about the command pattern's implementation. On the other hand, *client* is presumably a global object and such an encapsulation is not that beneficial since there might only be few to no objects outside the client's context anyway. The problem is that this scenario is the only ownership structure that satisfies the required access rights. With the desired encapsulation of the command object by its invoker, the command instance may only invoke pure methods on the receiver, which is not enough. Considering that invoker, command and receiver belong to different application layers, having to declare these objects peer to each other results in a very flat ownership structure and therefore impacts the whole application's ownership design. Applying an ownership structure to the command pattern, using the Universe type system, does not only result in few to no benefits, it may even be possible that with an existing ownership structure, implementing the command pattern is hard or even impossible due to the aliasing restrictions imposed by the ownership system. The Universe type system clearly lacks support for shared contexts and ownership transfer in order to fully support, and enhance, the command pattern.

Ownership Types

The relaxed aliasing restrictions of Ownership Types compared to the Universe type system enables some interesting scenarios. Because an encapsulated object may reference its ancestors and objects they own directly in a read/write manner, the invoker (e.g. a menu item) can be embedded in a deep ownership structure in the presentation layer, while still having access rights to a global command object, allocated by the client. The remaining problem is that command objects need access rights to their receivers, usually domain layer objects. Assuming that the domain layer is also structured with deep ownership, necessary alias rights might not be granted to the command, as shown in Figure 7.23. As a workaround, one could implement the command as an inner class of the

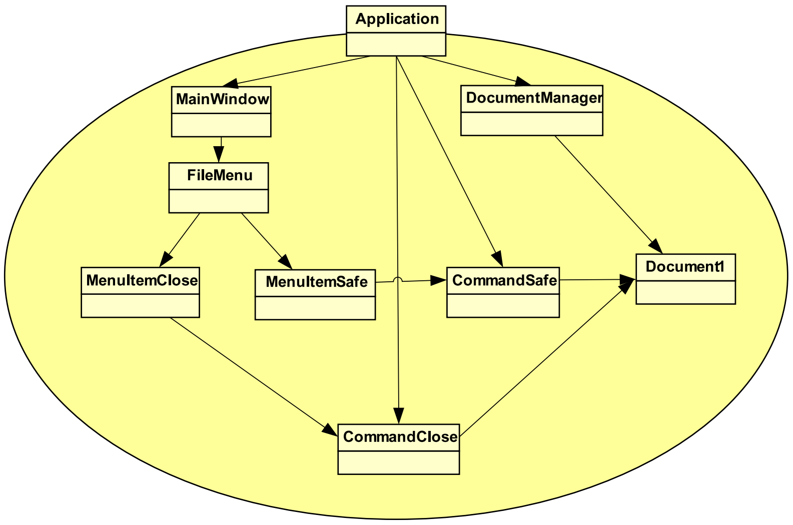


Figure 7.22: Universe type system ownership structure of the command pattern at the example of a document editor. Because alias rights between invoker, command and receiver need to be guaranteed, the application needs to have a flat ownership structure, even though it would be advantageous if `DocumentManager` could be declared as the owner of all documents.

receiver. This will solve the problem, as long as each receiver has its own commands. Otherwise, such a concept would lead to code duplication. Another more elegant way an application can deal with missing access rights of commands to the domain objects is to implement a controller class that acts as a facade for the domain layer. All business logic operations are passed through the controller which then performs the necessary delegations. The controller is allocated by the application and therefore remains in the same context as the command objects. Chapter 9 provides an example of such a scenario.

Ownership Domains

The Ownership Domains system is, due to its flexible linking concept, the only reviewed ownership system that is able to provide the necessary alias rights between the involved objects, but still allows them to be in different ownership contexts. In the GUI application scenario, the client allocates all available commands in a `commands` context and establishes the necessary access rights from all contexts holding invokers to `commands` and from the `commands` context to all contexts holding receivers. This enables us to build a deep and beneficial ownership structure for the whole application while still being able to implement the command pattern. Figure 7.24 illustrates the ownership structure. Command objects may be successfully shared, but unfortunately, the link restrictions require linking all parent contexts of a receiver to `commands`. Thus, the ideal design where only dedicated invokers can access commands cannot be guaranteed.

Large applications with very deep ownership structures are of course still hard to type due to the great number of necessary links between all contexts, which results in an overall weak encapsulation.

Conclusion

Once again we deal with the major problem that with the introduction of ownership, patterns that express collaborations between totally different objects, which are likely to belong to different application layers, are hard to type. Although ownership can help to enforce a clear interaction structure, requirements for an optimal ownership system are high: it must provide mechanisms like ownership transfer and shared contexts where fine-grained alias rights can be defined.

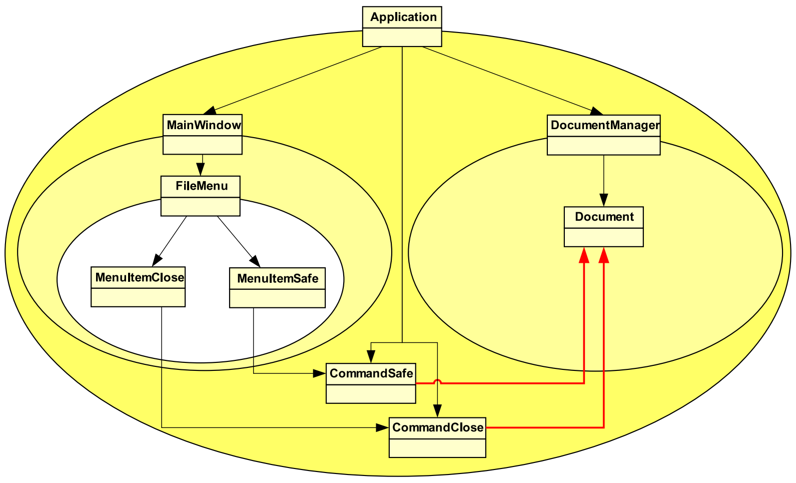


Figure 7.23: Ownership Types structure of the command pattern, considering as example a document editor. In contrast to the Universe type system, invokers, commands and receivers may be hierarchically structured. However, in order to enable a command to access `Document`, the command must be implemented as an inner class of `Document` or `DocumentManager` must act as the controller for its documents by forwarding all business logic operations. The critical references are colored red in the figure.

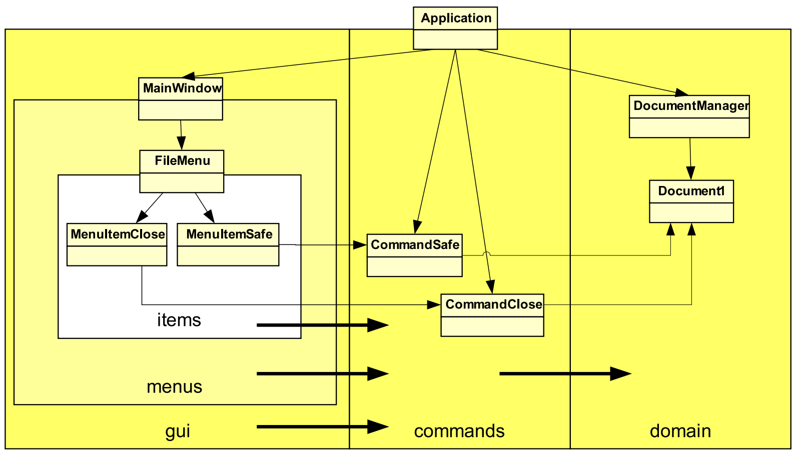


Figure 7.24: Ownership Domains structure of the command pattern at the example of a document editor. Private contexts are represented as solid rectangles, links are bold arrows. We are able to introduce a deep ownership structure while still guaranteeing the necessary alias rights through well chosen links.

Ownership in conjunction with the reviewed systems is more a question of feasibility when an ownership structure is already in place than what the resulting benefits from ownership are.

It has been shown that Ownership Domains are, thanks to the linking concept, able to deal with deep ownership structures while Ownership Types require commands to be implemented as inner classes of their according receivers or the introduction of a controller class. The Universe type system is the strictest system which requires invokers, commands and receivers to be located in the very same ownership context, resulting in a generally flat ownership structure for the whole application.

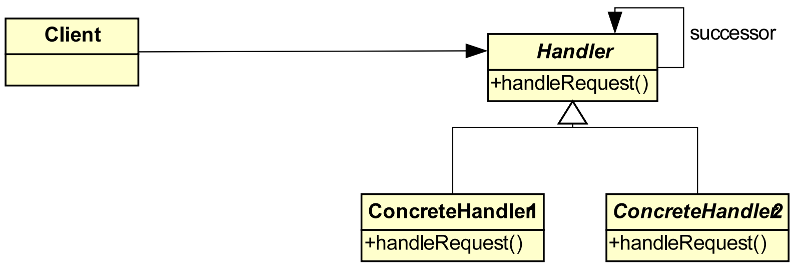
7.10 Chain of Responsibility

Intent

> ”Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.”[11]

An application may sometimes issue a request to handle but we do not want to specify the receiver explicitly or more than one object may be able to handle the request. The chain of responsibility pattern provides a solution for such a situation by defining a `Handler` interface that offers a `handleRequest()` method together with a successor field, pointing to the next possible handler. Hereby, a chain of possible handlers can be built up dynamically while it remains unknown to the request issuer what object, if any at all, actually processes the request. We say that the request has an implicit receiver and since neither the receiver nor the sender has knowledge of each other, the pattern provides a mechanism of reduced coupling between objects. The prime example of a chain of responsibility pattern is the context-sensitive help mechanism in most applications. A user may request help for any user interface object but because not all objects will have a help dialog defined, the help request gets forwarded from a very specific interface object (i.e. a text field) to a more general one (i.e. the dialog window) until some help information is found.

UML Diagram



Ownership Discussion

An investigation on the chain of responsibility pattern in terms of ownership shows that alias control is not an issue. None of the involved objects makes any assumptions on another object’s state. Quite the contrary, due to the reduced coupling of all involved objects, responsibilities may be reassigned dynamically at the cost of having no guarantee that a request will be handled.

Since ownership does not help to improve the pattern’s implementation, the probing question is if the chain of responsibility pattern can be implemented at all, assuming that an ownership structure is already in place. Answering this question in general is hard because it greatly depends on the application scenario. We can still investigate the common usage scenario where requests are routed between similar objects in the same application layer, likely along a composite structure

[11] [GHJV95] page 223

towards the root element. The prime example of the context sensitive help for GUI elements is such a scenario since the user interface elements are hierarchically ordered in a composite structure. Considering the fact that requests are usually routed from specific handlers to more general ones, they are likely to be passed up in the object structure to ancestor ownership contexts.

For the further discussion we will distinguish three different request routing scenarios.

1. Requests are passed along the chain in the same ownership context to peer objects.
2. Requests are passed to the current owner in an ownership structure.
3. Requests are passed to an object residing in a totally different ownership context.

Universe type system

The first case where a chain of responsibility is established between objects in the same ownership context is trivial and well supported by the system and therefore needs no further discussion.

The second case, where requests are passed up an ownership structure to the current owner object, is illustrated in Figure 7.25 and leads to problems as the Universe type system only provides `readonly` references to parents.

The third case is obviously not feasible with the Universe type system due to the `readonly` references to foreign contexts.

A chain of responsibility can therefore only be established between peer objects, constraining the application's overall ownership structure. Unless of course the `handleRequest()` method is pure which is very unlikely, considering the fact that all I/O operations are not pure.

Hence, in the prime example of a context sensitive help system all involved GUI elements need to be located in the same ownership context, making it impossible to introduce a deep ownership structure for GUI elements.

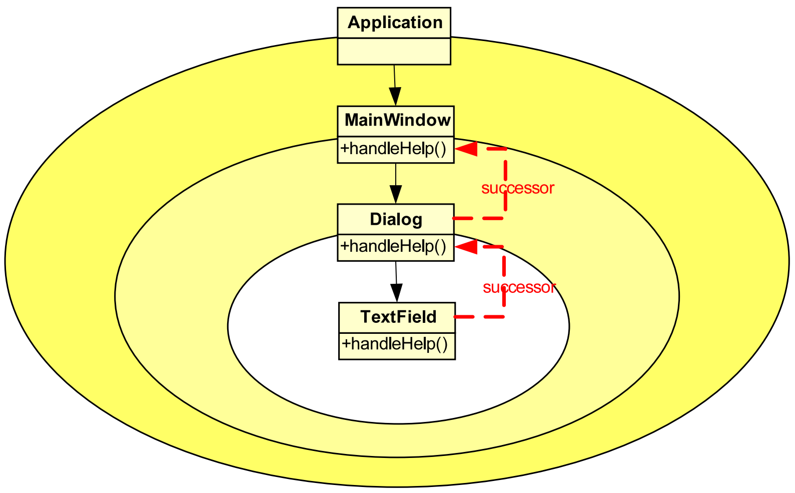


Figure 7.25: Universe type system: Chain of responsibility pattern in case of a context sensitive help system. `readonly` references are dashed arrows whereas problematic references are colored red. Only `readonly` references to the next handler object are provided, requiring the `handle-Help()` method to be pure.

Ownership Types

Since Ownership Types allow read/write references from an object to its owner, not only the first but also the second scenario, shown in Figure 7.26, is possible to implement. The only case where an implementation of the chain of responsibility pattern fails is when the application scenario requires us to establish a chain between foreign objects, residing in totally different ownership contexts. At least the context sensitive help example is well supported, allowing the introduction of a deep ownership structure in the GUI element hierarchy.

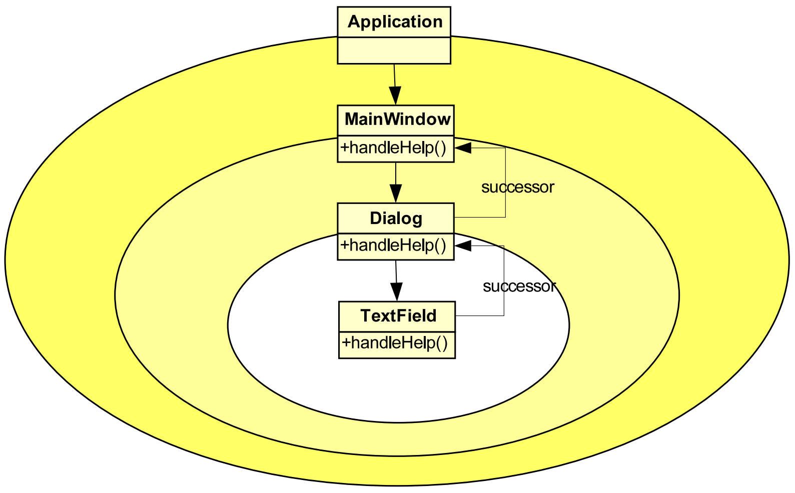


Figure 7.26: Ownership Types: Chain of responsibility pattern in case of a context sensitive help system. In contrast to the Universe type system, Ownership Types supports read/write references to the owner.

Ownership Domains

The only reviewed ownership type system that can possibly cope with all three request routing scenarios is Ownership Domains. In the case where requests need to be routed to the current owner object, one simply needs to introduce an up-link to the parent's context. In addition, even if references between totally different contexts need to be maintained by defining appropriate links, the desired chain of responsibility can be configured, always under the assumption that the link constraints 4.4 permit such a linking. Figure 7.27 illustrates this situation.

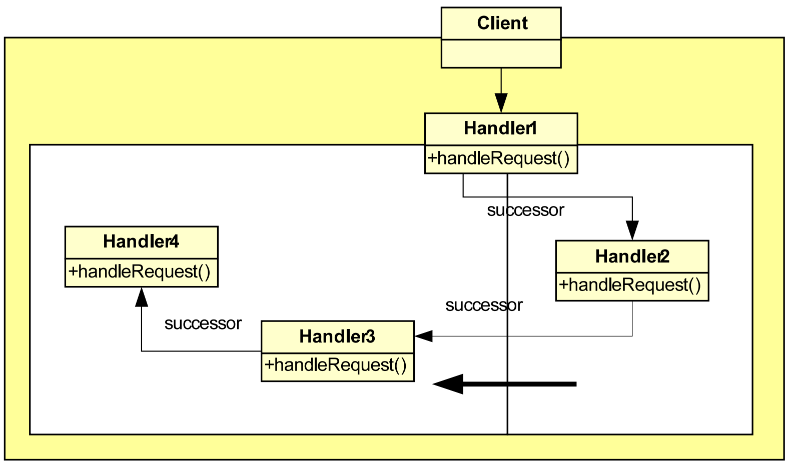


Figure 7.27: Ownership Domains: Chain of responsibility pattern where possible handlers are located in different ownership contexts. Private contexts are represented as solid rectangles, links as bold arrows.

Conclusion

The concept of ownership does not help to improve the chain of responsibility pattern's implementation. Due to the fact that all involved objects are independent from each other, not making any assumptions about another object's state, there is no need for alias control. Quite the contrary, an already existing deep ownership structure can make an implementation of the pattern hard or even impossible since references between objects, residing in different ownership contexts, might be needed. It turns out that the Universe type system is too strict by only allowing a chain of responsibility between peer objects. Ownership Types is even able to cope with the situation where a request gets routed upwards in the ownership hierarchy to the current parent. Only with Ownership Domains a chain of responsibility can possibly cross an ownership context's boundary, if the necessary links can be declared. It is to say though that being able to declare such a link is likely to result in the need to declare additional links first, ending up in an overall weak encapsulation.

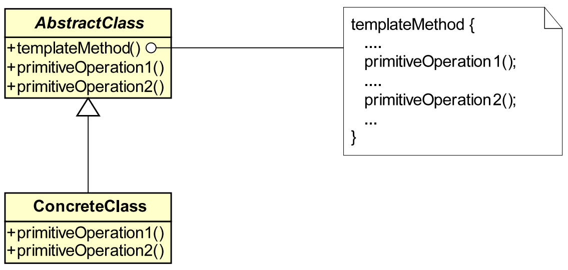
7.11 Template Method

Intent

> ”Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.”[12]

The template method pattern is often applied to existing code in order to make it more general. The idea is to declare the skeleton of an algorithm in an abstract super class. Thereby situation-specific operations are defined as abstract methods which can be implemented by concrete subclasses later on. We distinguish between so-called *hook*-operations that define a default behavior but *can* be overridden by a subclass and *abstract* operations which *must* be defined in the subclass.

UML Diagram



Ownership Discussion

A reflection on the template method pattern clearly reveals that ownership is irrelevant. The pattern's obligation is to reuse an algorithm for several different situations by moving the general part out to the abstract superclass and leave the concrete operations to be defined in the subclasses. The client just instantiates the desired scenario-specific subclass and calls `templateMethod()`. Since the pattern only consists of one object it does not make sense to discuss ownership.

Conclusion

Having only one involved object in the pattern (the concrete class that implements the template method) ownership is not needed.

[12][GHJV95] page 325

8 Summary

8.1 Pattern Overview

This section provides an overview of all examined design patterns. The patterns are structured into three tables, pointing out if an ownership typing can be applied successfully (yes), if it strongly depends on the usage scenario (depends), or if ownership cannot be applied (no). Possible restrictions or problems are marked with footnotes.

	UTS	**Ownership Types**	**Ownership Domains**
Abstract Factory	no[1]	no[1]	no[1]
Builder	yes	yes	yes
Factory Method	no[1]	no[1]	no[1]
Prototype	yes	depends[1]	depends[1]
Singleton	no[2]	yes	yes

[1] Request for ownership transfer or object creation in a foreign context .

[2] Request for a global context .

	UTS	Ownership Types	Ownership Domains
Adapter	yes	yes	yes
Bridge	yes	yes	yes
Composite	depends[1]	depends[1,3]	depends[1,3]
Decorator	no[1,2]	no[1,2]	no[1,2]
Facade	yes	yes	yes
Flyweight	yes	depends[3]	depends[3]
Proxy	yes[4]	yes	yes

[1] Request for ownership transfer or object creation in a foreign context .

[2] Encapsulation is not beneficial .

[3] Request for readonly references .

[4] Request for readonly references suppression .

	UTS	Ownership Types	Ownership Domains
Chain of Resp.	no[2,4]	depends[2]	depends[2]
Command	no[4]	yes	yes
Interpreter	depends[1]	depends[1]	depends[1]
Mediator	no[5]	no[5]	yes
Memento	yes	no[3]	depends[3]
Observer	no[4]	no[4]	yes
State	depends[4]	yes	yes
Strategy	depends[4]	yes	yes
Templ. Method	no[2]	no[2]	no[2]
Visitor	depends[1]	depends[1]	yes

[1] Request for ownership transfer.

[2] Encapsulation is not beneficial.

[3] Request for readonly references .

[4] Request for a "friend"-concept or r/w reference to owner.

[5] Request for multiple contexts.

8.2 Main Problems Encountered

We sum up the main problems encountered when applying ownership to design patterns.

Ownership Transfer

An investigation of each design pattern revealed the fact that it is often not sufficient for an object to remain in a single ownership context its entire life. None of the reviewed systems provides a mechanism for ownership transfer and therefore the only workaround for a situation that requires a transfer mechanism is to declare fewer ownership contexts so that all collaborating objects reside in the same context. This is not desired since the result is a much weaker encapsulation structure.

The need to change the ownership context can emerge from the following situations:

1. Operations that traverse (deep) object structures (visitor 7.3).
2. Situations where the object creation order is possibly not in accordance with the ownership structure (adapter 6.1, proxy 6.7, composite 6.3, command (7.9)).

3. Object creation is delegated to facilities in a different context (abstract factory 5.1, factory method 5.2, prototype 5.3).

Aliasing Exceptions

The second problem we encountered is that ownership systems declare a strict aliasing policy but often do not provide a mechanism to deal with aliasing exceptions. An *exception* can range from having the notion of a *global* context to which all objects have access, over *shared* contexts for each class, to the authorization of single objects to access a foreign context.

The following situations revealed the problem:

1. Many designs express collaborations between tightly coupled objects in terms of usage, but not in terms of ownership (observer 7.2, command 7.9, chain of responsibility 7.10).

2. Many designs encourage sharing objects, but an encapsulation usually makes sharing impossible (builder 5.4, composite 6.3, strategy 7.6).

3. Global objects require the concept of a global context (singleton 5.5).

4. Bidirectional references between objects in a direct ownership relation require alias rights for owned objects to their owner (strategy 7.6, state 7.5, observer 7.2).

Of course, there are great differences regarding the possibility to declare exceptions when comparing the three ownership systems.

The Universe type system currently supports none of the situations above since all references to foreign contexts are strictly readonly.

Ownership Types provides the ability to declare aliasing exceptions through its inner class concept. But, it has shown that under almost all circumstances one would not want to implement one class as an inner class of another one, just to gain alias rights. Usually, the collaborating objects are, in terms of functionality, not as tightly coupled that an inner class implementation is justified. Hence, Ownership Types' inner class exception mechanism is rather insufficient for the first two situations. Nevertheless, it provides the notion of a global context and all owned objects can establish read/write references to their owner.

Ownership Domains is the only reviewed system that has the ability to declare aliasing exceptions through its link concept and is therefore theoretically able to deal with all situations. It is to say though that when coping with an exception, the ownership structure might easily become complicated and generally weakened up in terms of encapsulation due to the linking constraints.

Multiple Contexts

The third problem encountered is that in some designs, it turns out to be very useful if the ownership system has support for multiple ownership contexts. Such a property is especially useful when expressing architectural constraints like application layers or the mediator pattern (7.4). In these situations, ownership can help to statically verify a correct implementation of the design by preventing illegal aliasing between objects in different layers or direct references between colleagues in case of the mediator pattern.

Only Ownership Domains provides full support for this feature by allowing objects to declare multiple ownership contexts. In case of the Universe type system, all objects with the same owner automatically have the ability to freely alias each other and the programmer has no possibility to prevent certain aliases. Ownership Types has a concept called *role separation* which ensures that objects with a different ownership type are assumed to be in different contexts, although both types might have the same binding. This allows some alias control between objects in the same context. However, full alias control for objects with the same owner can only be provided if the system has support for multiple contexts.

Readonly References

The forth generally encountered problem is about the missing support for readonly references. The concept of readonly references has shown to be very useful in many situations. Especially an implementation of the flyweight (6.6) and the memento (7.8) pattern greatly benefits from readonly references since the system could statically ensure that no shared flyweight gets modified by a client object and that the caretaker has no possibility to modify the memento object.

Out of the three ownership systems, only the Universe type system supports readonly references and we believe that also the other two systems could greatly benefit from a readonly references support.

It is to point out that the Universe type system does not suppress leaking of readonly references since every object may reference all other objects through readonly references. This is in contrast to Ownership Types and Ownership Domains where it can be ensured that no aliasing at all may occur. We have shown that there are situations, e.g. when implementing a proxy (6.7) or sometimes a facade (6.5), where even the existence of readonly references should be prohibited. Providing support to suppress readonly reference in case of the Universe type system is easy: one could think of a new ownership modifier, e.g. `rep_strict`, that always resolves to *forbidden* when the type combinator is applied.

Part III

Case Study

9 An Accounting Application

9.1 Intent

After having reviewed feasibility of ownership in conjunction with each design pattern separately, we would like to review possible ownership structures for a demo application. The application under review heavily makes use of patterns to express the core design. For the sake of simplicity, the demo application is kept very small and simple, serving as a simple accounting tool to transfer amounts between different accounts. The full source code for the application can be found on the provided CD-ROM.

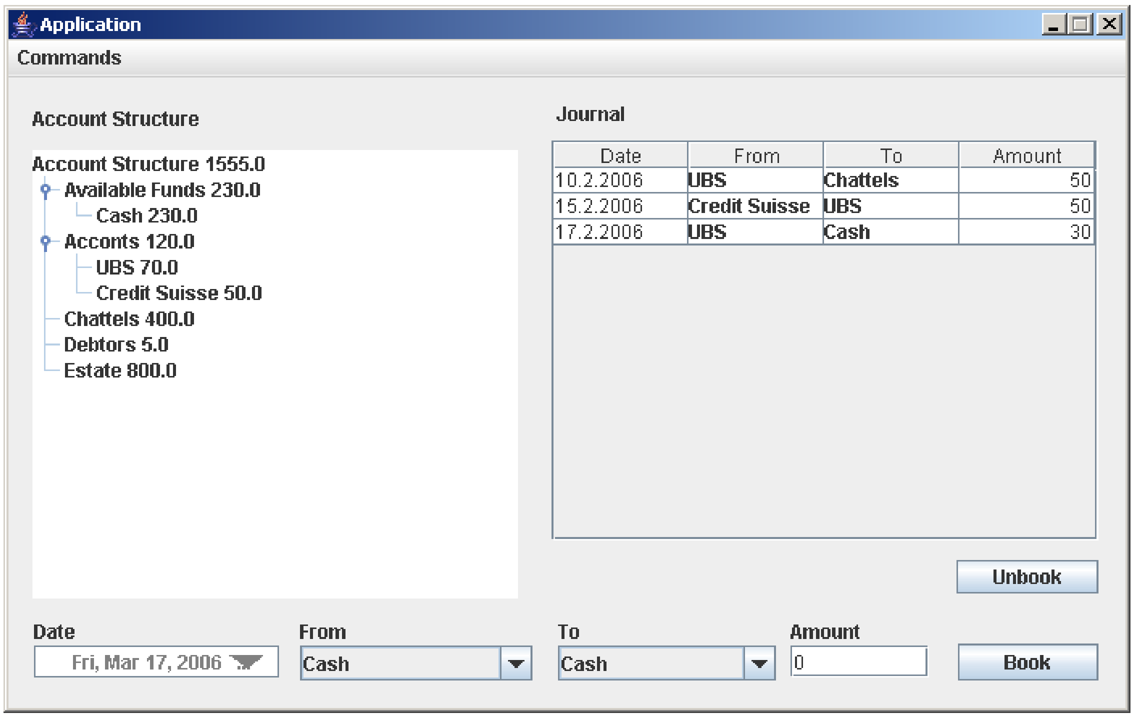
9.2 Application Logic and Overview

The business logic is modeled by only four domain classes: `Group`, `Account`, `Booking`, `Journal`. The main idea of the application is to maintain an account structure which consists of groups and accounts where a group can contain further groups or accounts. Each account has a name and a balance. Amounts can be transferred between different accounts by entering a booking. Bookings contain a date, the source account, the target account and an amount. The journal keeps track of all bookings in the system. The application's user interface consists of a single main window containing a tree control illustrating the account structure together with a table widget that visualizes the journal. New bookings can be entered into the system by selecting the date from the datepicker, the source and target account, and specifying the amount to be transferred.

9.3 Core Design

A good design separates different application concerns into multiple, layered components where each component is potentially replaceable by a different implementation. A very basic and simple approach separates the presentation from the business logic, namely the domain layer. Our accounting application follows this approach by segmenting the application's classes into different packages. To emphasize the pattern based design, not only the domain classes are tied together in the `domain` package, but also all interfaces and classes of each pattern implementation are packaged separately.

The main purpose of this demo application is to discuss possible benefits and problems when applying the concept of ownership to application designs that make heavy use of patterns. There-



Account Structure

Account Structure 1555.0
- Available Funds 230.0
  - Cash 230.0
- Acconts 120.0
  - UBS 70.0
  - Credit Suisse 50.0
- Chattels 400.0
- Debtors 5.0
- Estate 800.0

Journal

Date	From	To	Amount
10.2.2006	UBS	Chattels	50
15.2.2006	Credit Suisse	UBS	50
17.2.2006	UBS	Cash	30

Date	From	To	Amount
Fri, Mar 17, 2006	Cash	Cash	0

Figure 9.1: Screenshot of the accounting application's main window.

fore, the accounting application's design makes use of six important patterns: flyweight (6.6), composite (6.3), command (7.9), visitor (7.3), observer (7.2), and decorator (6.4).

Flyweight All used widgets are configured with a user-defined renderer that takes care of displaying the domain object to be visualized: each account is displayed, depending on the widget, either only with its name or with its name together with its current balance. The render objects are referred to as flyweights since they only contain one method (`Component get*Component()`) which is consulted upon painting the widget.

Composite For a convenient handling of the account structure, both `Group` and `Account` extend the same superclass `StructureItem` that defines the common properties and corresponds to the `Component` class in the Composite pattern description. `Group` hereby corresponds to the `Composite` class, being able to contain further `StructureItems`. The second obvious composite pattern in the application is the composition of the GUI elements. Swing widgets extend the abstract base class `JComponent` by either acting as a container for further `JComponents`, like the `JPanel` subclass or by acting as a leaf component such as `JButton`.

Command `Booking` acts as a command, implementing the `Command` interface. The interface defines two methods `execute()` and `undo()` where `execute()` transfers the specified amount from the source account to the target account and `undo()` performs the inverted transaction. With this mechanism, it is always possible to cancel already executed bookings from the journal.

Visitor The `PersistencyManager` class is responsible for loading the account structure. Since some widgets only need the `Account` instances and not the groups, an instance of `AccountCollectVisitor` traverses the account structure and collects all `Account` instances. One could think of further useful operations that could be provided by defining additional visitors (i.e. subclassing the abstract `Visitor` class).

Observer The observer pattern is used to ensure coherent balances throughout the accounting structure. Each group instance must have a balance that corresponds to the sum of all children's balances. Consequently, balance changes in an account affect all parent groups up to the root group. In order to ensure a consistent balance structure, each group observes its children's balances for changes. If such a change occurs the group automatically adjusts its own balance.

Decorator As explained in the previous section, the application makes use of a `JTree` component to display the account structures. Hence, it is obvious that all group and account objects need to be displayable by a tree control which requires all items to implement the `TreeNode` interface. Letting the domain objects implement the `TreeNode` interface directly would result in a mixed up architecture where domain objects are "polluted" with presentation specific code. We therefore decided to build a decorator class to add presentation specific responsibilities to the abstract class `StructureItem` dynamically. `ItemTreeDecorator` extends `StructureItem` and additionally implements the `TreeNode` interface to act as a displayable node in a `JTree`.

9.4 Desired Ownership

To emphasize the layered architecture and to gain the compile-time safety that the desired layering is not compromised by illegal references, we provide a possible ownership typing and discuss its up- and downsides.

Separation of the presentation layer from the domain layer is done by allocating GUI related objects in a different ownership context than domain objects and hereby preventing uncontrolled aliasing. We therefore suggest the following ownership structure: The `Application` class, implementing the main method, spans the application's root ownership context and allocates an instance of `MainWindow`, spanning the presentation layer, and an instance of `Controller`, declaring the domain layer ownership context. This initial and very simple structure ensures a correct layering: it separates the presentation from the domain objects and by this restricts aliasing between layers. Moreover, it enforces the property that all operations on the underlying model are passed through the controller, the owner of the domain layer. The single responsibility for the presentation layer is to listen to user requests and display the domain objects. All domain layer specific operations are ensured to be implemented in the appropriate domain classes.

This strict layering has of course several impacts on the implementation. We assume that the used ownership system supports readonly references. Having such references from the `JComponent` Swing models to the displayed objects prevents us from implementing content modifying operations directly in the model. Thus, the models have to pass all write operations over the controller to the domain layer. The second property an optimal ownership system has to ensure is that all `ActionListener` objects, configured to handle widget events, have the possibility to access the controller in a read/write manner to process domain operations. Usually, the window class instantiates `ActionListener` as an anonymous innerclass, passing it directly to its widget. Note that the widgets, and thus also the `ActionListener` objects, are owned by the window while the controller is peer to the window instance and the system therefore has to ensure that, although residing in different contexts, both objects have the possibility to unconditionally reference each other.

Ownership can not only ensure the design property of a strict layer architecture but also help to ensure immutability of certain objects. In the context of our accounting application, each booking consists, as previously mentioned, of a date having the type `GregorianCalendar`, the source and target `Account`, and the amount specified as `double`. Once a booking is entered into the system it can only be executed or canceled, but properties of an existing booking should not be modified. Accordingly, the `Booking` class does not allow modifications of its properties. The problem is that even by not providing setter methods for its properties, other objects are still able to mutate an existing booking by retrieving the `GregorianCalendar` object and modifying its internal date. `GregorianCalendar` objects are either instantiated by the `JDatePicker` bean or by the `PersistencyManager` class and we therefore suggest that both classes encapsulate the date upon instantiation in their ownership contexts, making it impossible for other objects to alter it after creation. Other objects like `Booking` instances or the journal table model should only be allowed to maintain readonly references to the dates. Refer to Figure 9.2 for a complete overview of the proposed ownership structure.

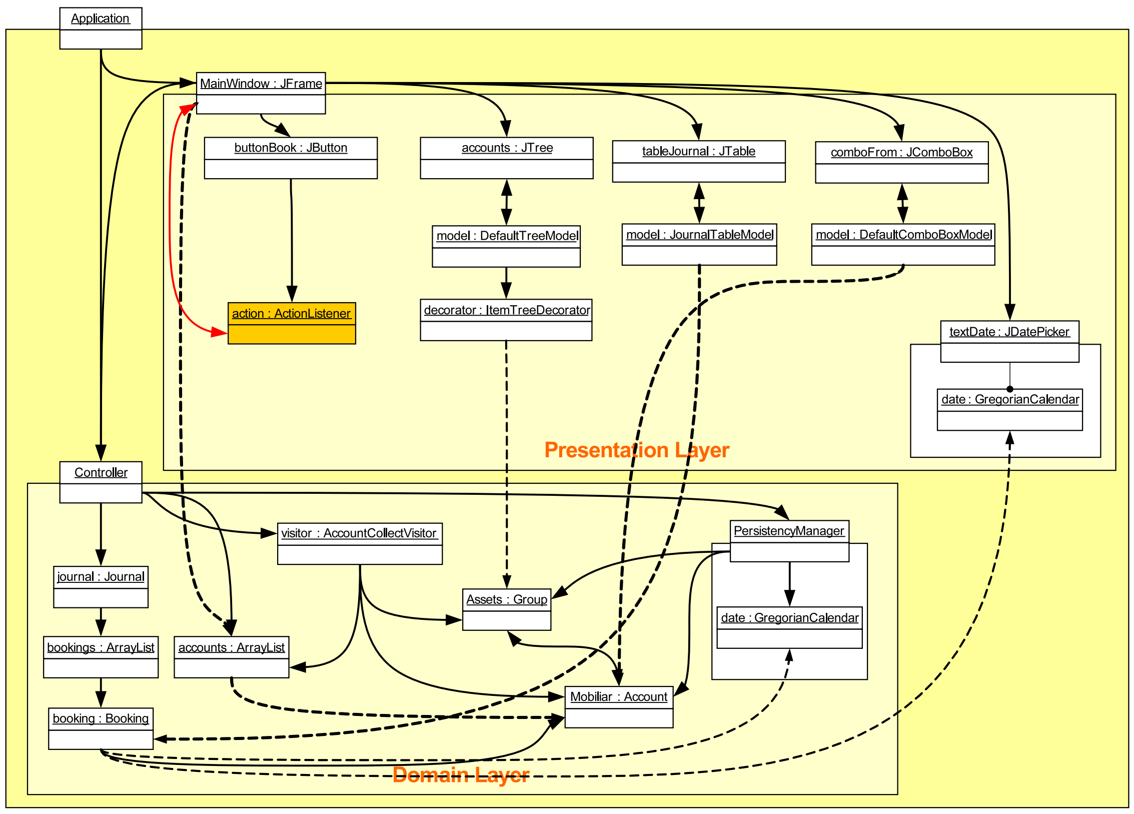


Figure 9.2: Object and ownership structure overview of the accounting application. Dashed arrows represent readonly references, solid arrows are read/write references. Rectangles mark ownership contexts. Notice the red arrow from the `ActionListener` instance to the `MainWindow`: it is crucial for an ownership system to allow such a bidirectional read/write referencing in order to be able to process domain operations triggered by GUI elements.

Regarding ownership in conjunction with the different pattern implementations, we notice flat ownership structures where all involved objects are peer to each other. The composite pattern expressing the account structure has a flat ownership structure due to the highly interlinked domain layer. This flat ownership structure allows visitors, as well as bookings representing a command, to directly reference accounts. Additionally, the implementation of the observer pattern, where each group object is the observer of all contained items, is straight-forward. This is somehow disillusioning regarding the possible beneficial ownership structures for each pattern in the according discussions. On the other hand, it is to mention that the implemented patterns are not only hard to improve with an ownership structure but also very likely to require system extensions. In an ideal ownership system one could also implement a deep ownership account structure where each group is the owner of its children. The system would just have to ensure that:

- The account or group is transferred to the correct ownership context after creation by the persistency manager or controller.
- The visitor instance is still allowed to traverse the account structure.

- A booking is allowed to access its referenced accounts in a read/write manner to transfer amounts.
- A composite child is allowed to send amount changed events to its parent in order to retain consistency.

By reviewing the referencing requirements on the domain layer, it becomes obvious that such a deep ownership structure is not necessary with regard to the application's size. However, with an increasing number of domain classes and operations, structuring the domain layer in different ownership contexts, as mentioned above, might become an evident need.

Of course, ownership should not stop at the application level as also frameworks like the used Java Swing API could benefit from ownership. We will discuss chances and risks of a Swing ownership typing in the next chapter.

9.5 Feasibility and Conclusion

We can conclude that the concept of ownership can improve an application's design by enforcing architectural constraints and make an implementation less error-prone. Due to object encapsulation, the programmer gains static safety that objects belonging to a certain layer do not get illegally modified by objects from another layer. Thanks to readonly references, immutability of certain objects can be guaranteed. In the context of the accounting application, this property has especially shown useful when dealing with `GregorianCalendar` instances.

Reviewing feasibility of the proposed ownership structure in conjunction with the presented ownership systems shows that no system is able to provide a correct typing. The Universe type system lacks the ability of read/write references to the parent object, required to pass modification requests from `ActionListener` objects to the controller. Ownership Types and Ownership Domains lack the ability to provide readonly references. It is to say though that with Ownership Domains one could still think of the possibility to only provide domain layer alias rights to the Swing models by allocating them in separate domains and declare appropriate links. However, this would result in weaker architectural constraints since the models would then be allowed to perform updates on the domain objects directly, bypassing the controller. To sum up: all reviewed ownership systems need additional features to provide an implementation for the desired ownership structure.

10 The Java Swing Framework

10.1 Intent

After investigating benefits and problems of ownership at the example of a small accounting application that has a Swing graphical user interface, we would like to extend our review to the Swing GUI toolkit in this chapter.

The Java Swing framework was chosen due to its wide usage in Java applications that provide a graphical user interface and due to the fact that its design heavily relies on patterns [Gam99]. The second reason why the Swing framework was chosen for a review is that ownership systems are fairly invasive, meaning that once an application is initially typed with ownership, it soon becomes necessary to also annotate at least the interfaces of the used libraries. Hence, it is worthwhile to also investigate possible ownership structures for the main Swing classes used in our example accounting application, presented in the previous Chapter 9.

In the following discussion, concepts and diagrams rely heavily on information from [ELW98].

10.2 Reviewed Swing Classes

Already the initial release of the Swing 1.0 libraries contained nearly 250 classes and 80 interfaces and has been extended since. Due to its size and complexity, it is therefore hard and beyond this thesis' scope to perform a complete discussion of the framework. Instead, we will focus on the most important classes and provide an example discussion based on a widely used widget, the combobox.

Before going into details we take a look at Swing's core design: the MVC architecture. MVC expresses the idea of breaking GUI components into three elements, the model, the view, and the controller. Quickly explained: the view is responsible for displaying the widget on the screen and listening to user events which will then be passed to the controller. The controller decides which events require modifications of the widget's underlying data model and, if necessary, updates the model. Since the view is responsible for visualizing the model's data, the view queries the model upon painting and gets notified by the model when changes occur.

Swing makes use of a simplified version of the MVC design, the so-called *model-delegate*. The idea is to express the view and controller's functionality in a single element, namely the delegate. Each Swing component has thus an assigned model together with a UI delegate that is responsible for painting the component in the current look and feel of the application and handling all user events. Since Swing is built on top of the AWT (Abstract Window Toolkit) framework, event handling in Swing is done in collaboration with AWT. Figure 10.1 illustrates the concept of the model-delegate design.

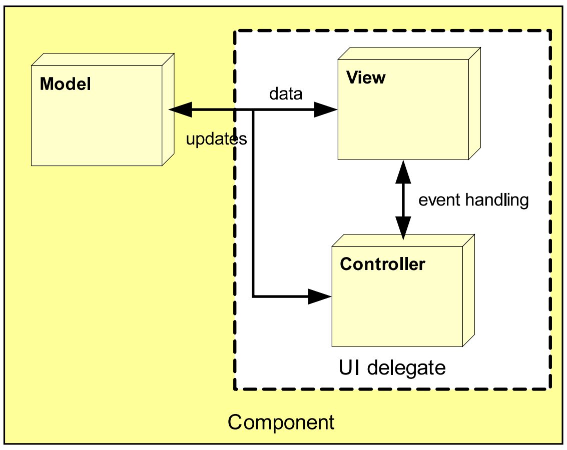


Figure 10.1: Illustration of the Swing's *model-delegate* design. In contrast to the originally proposed MVC design, the model-delegate unites the controller and the view in a single element called the UI delegate.

After this short introduction to the main design concept of Swing widgets, we list the main Swing classes involved in the accounting application worth discussing. Since conceptually the widgets do not differ greatly, we chose to discuss the combobox related classes. In the next section, collaborations and possible ownership between these classes are explained.

JComponent

`JComponent` is the abstract super class for all concrete Swing widgets and therefore an important candidate when discussing possible ownership structures. `JComponent` inherits from `java.awt.Container` and `java.awt.Component` which clarifies the fact that Swing is built on top of AWT, and has the consequence that each component acts as a container, opening the possibility to have an associated layout manager and host further components. However, one should be aware that not all Swing components are intended to be a container. `JComponent` declares important fields such as the UI delegate reference, mentioned above, background and foreground color, the used font, its size, and many more.

JComboBox

The `JComboBox` is a concrete widget that extends `JComponent`. Since different widgets may have totally different requirements to their underlying model, the model property is defined here. Furthermore, a `JComboBox` instance may be configured with a `ComboBoxEditor`, a component used for editing the combobox, and a cell renderer. A user defined cell renderer should implement the `ListCellRenderer` interface which only declares a single method, `getListCellRendererComponent()`, responsible for returning an AWT component to be displayed for each list entry. The renderer object hereby acts as a flyweight and follows the strategy pattern approach.

ComboBoxModel

Each combobox has its own unique model assigned, keeping track of the data objects to be displayed in the selection and remembering the selected entry. The Swing framework provides a `DefaultComboBoxModel` class that internally maintains a `java.util.Vector` for the data and a field to store the currently selected object. The accounting application makes use of this class as no special behavior of the model is needed. `JComboBox` registers itself as an observer at the model instance in order to get updated on model change events. Hence, the combobox is bidirectionally linked with its model.

BasicComboBoxUI

This class represents the UI delegate for a combobox and is responsible for visualizing the widget on the screen. All paint requests from the combobox are passed to this class that does the actual painting. Since the Swing platform supports multiple look and feels, BasicComboBoxUI is extended by a separate UI class for each installed theme (e.g. `MetalComboBoxUI`). In order to correctly perform a paint request, this class registers itself as an observer at the model, listening for any model changes, and at its associated `JComboBox` instance to listen for user events. Furthermore, the UI delegate maintains a reference to the configured `ListCellRenderer` instance to retrieve the AWT components for each displayed list entry.

LayoutManager

As previously mentioned, each `JComponent` also potentially serves as a container for further widgets and may therefore have an associated layout manager. The layout manager holds references to all widgets in its associated container. Upon a layout request from the container, it performs positioning and resizing of its elements.

10.3 Ownership Discussion

In order to discuss possible ownership structures for the Swing framework it is worthwhile reviewing interactions between the mainly involved objects first. We would like to point the reader to Figure 10.2 for an object diagram, illustrating the most important references and method calls.

Heavyweight vs. Lightweight

Swing distinguishes two kinds of widgets: lightweight and heavyweight. Most of the provided widgets are lightweight components. Their characteristic is to be able to draw themselves. Heavyweight components, like all widgets provided by the AWT toolkit, have a platform specific peer component and use the platform's routines to draw themselves. Only a few top level widgets are heavyweight components in Swing (`JWindow`, `JFrame`, `JDialog`, and `JApplet`). The window object shown in the diagram has such an associated platform dependent peer since its type is `JFrame`. Separating the abstraction (`JFrame`) from the actual platform implementation (`ComponentPeer`) corresponds to the bridge pattern (6.2) and we suggest using the previously proposed ownership structure: declaring the abstraction as the owner of its concrete implementation. Hence, the window object encapsulates its associated peer.

Swing's Composite Structure

Each window can have one or more `JPanel` instances, serving as a container for additional widgets. Each `JPanel` may be configured with a layout manager that takes care of positioning and resizing the widgets. All widgets extend the abstract class `JComponent` and therefore the hierarchical widget structure corresponds to the composite pattern (6.3). Having the possibility to use different layout managers corresponds to a strategy pattern (7.6). As already discussed in the corresponding pattern discussion, an interesting and promising design is to use a deep ownership structure for the composite. Each container, like the `content` object in the diagram, is the owner of its associated widgets and its layout manager. Thus the container is hiding any implementation details, like the used layout, from objects outside the very same context. Such a deep ownership structure can guarantee certain properties that must hold: first, it is assured that each widget does exactly belong to one panel. Second, we have certainty that each panel has exactly one layout manager and other layout managers cannot interfere.

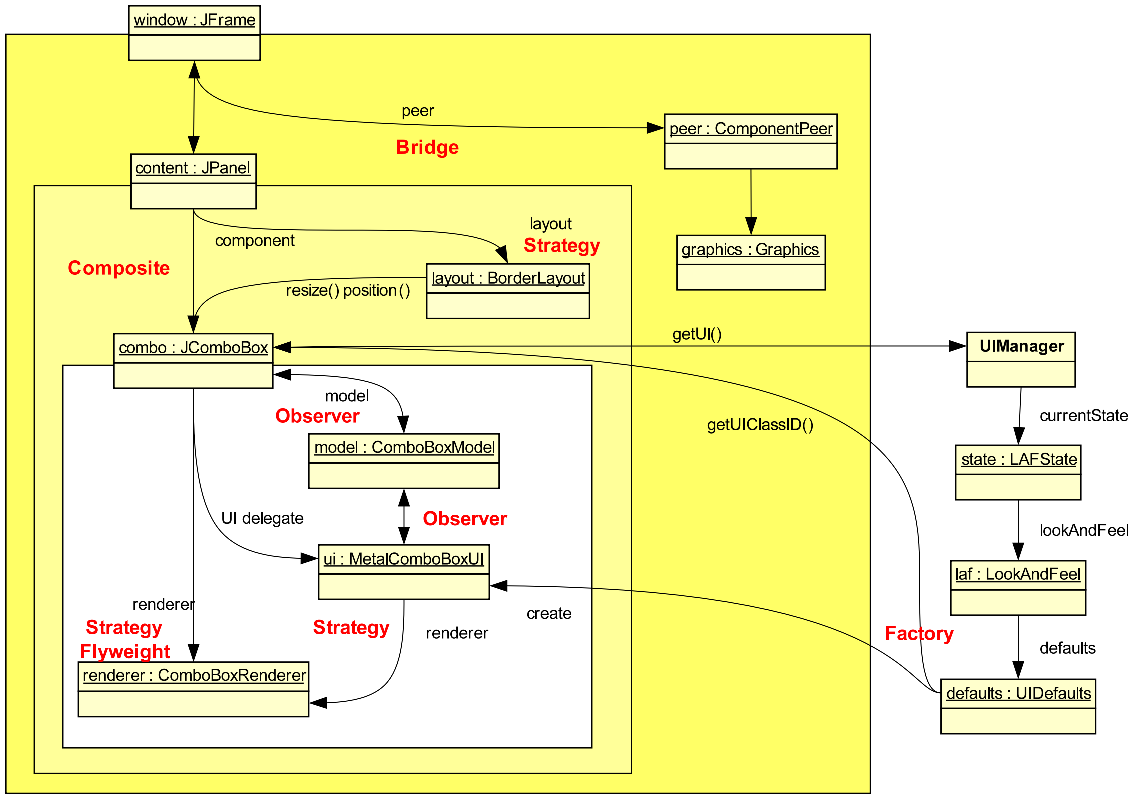


Figure 10.2: Object diagram of the main Swing objects together with the most important references and method calls. Used design patterns are labeled in red font. Desired ownership contexts are drawn as solid rectangles.

The problem is that, once more, a deep ownership structure in the composite pattern is too restrictive under most systems. After having setup the window's layout and its deep ownership structure, there are still certain objects that need access to encapsulated widgets in order to respond to user events and adjust the GUI, namely classes that implement the `ActionListener` interface. Widgets can be configured with `ActionListener` objects which handle user events that may trigger business operations and adjust other widget instances. These action objects do not only need access to potentially all (encapsulated) widgets, but also to the controller in order to process business operations. Assuming that the used ownership system provides a mechanism to establish the necessary access rights for all action objects, a deep ownership typing for Swing's composite structure works fine and ensures crucial design properties. If the used ownership system cannot meet these requirements, a typing with a deep ownership structure fails.

Model-View-Controller

The second observation from the diagram is that it might be useful to declare a widget as the owner of all its associated objects. Hence, a `JComponent` instance owns its model, UI delegate, possibly configured renderer, selection model, etc. This encapsulation allows a programmer to reason about the component as a whole, only considering involved and therefore owned, objects. Furthermore, it can be ensured that each widget has its own unique Swing model.

Such an ownership structure has many benefits, as pointed out above, but of course also greatly impacts the application's design and requires certain features from the used ownership system.

Since the application's custom window class is responsible for instantiating all Swing objects, it becomes clear that a deep ownership structure can only be built if the ownership system allows creation of objects outside the target context and provides the ability to transfer objects into the desired context after creation. In our case, the custom window instance would create all containers, widgets and customized widget objects like the desired model, renderer etc. and afterwards transfers ownership of these objects to the desired owner when setting the properties.

All combobox related objects are heavily interconnected by observer patterns and we require read/write references from an owned object (the model) to its owner (the `JComboBox` instance).

Moreover, it is to point out that the ownership system needs the notion of a global space, allowing all objects, or at least a carefully defined group of friendly objects to access objects contained in the global context. We recognize the need for such a mechanism when investigating the creation process of a UI delegate. As already pointed out, UI delegates follow the approach of a strategy pattern by painting the associated control in the configured look and feel of the application. When a widget (e.g. the `JComboBox`) gets instantiated, it calls the static method `getUI()` on the `UIManager` class. The `UIManager` is configured with a `LAFState` object (look and feel state) that has a property referring to an instance of the `LookAndFeel` class. Each `LookAndFeel` instance has a `defaults` property pointing to an instance of `UIDefaults`. The whole look and feel mechanism acts as an abstract factory and the `UIDefaults` object is responsible for creating the correct UI delegate. This is done by performing a callback on the passed-in caller widget to ask its `UIClassID`, a string to determine the widget type. Using Java's reflection mechanism, the correct UI delegate is created and returned to the original widget that triggered the request. For more information about Swing's look and feel and related topics refer to [ELW98].

Following the model-delegate methodology of Swing, widget clients are requested to directly operate on the models when changing or invalidating the data to trigger a repaint of the widget. Like in an encapsulated composite structure, the used ownership system thus needs the ability to grant special access rights for `ActionListener` instances so that they can change the widgets' models when an event occurs. The GUI does not only have to adjust itself upon locally triggered user events, but also upon domain layer changes, that may be triggered by system events or concurrent users. Thus, action objects do not only exist in the presentation layer, but also in the domain layer and the used ownership system must ensure appropriate access rights for *all* action objects.

Paint Process

Another interesting action worth looking at in terms of ownership is the paint mechanism of Swing components. As already mentioned, only the top level components have native peer components. All other widgets are lightweight components that can draw themselves based on a platform independent graphics library. We will examine how the delegation of the painting process happens. Figure 10.3 shows the interaction diagram of the process.

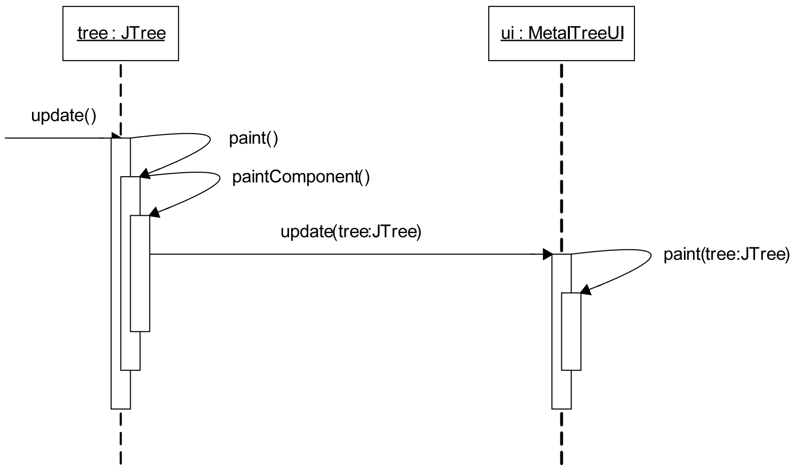


Figure 10.3: Painting delegation in Swing

1. When a Swing component is asked to update itself, it executes its local `paint()` method.
2. After doing some additional work, `paint()` will then call the `paintComponent()` method.
3. `paintComponent()` delegates the actual painting, by calling `update()` on the UI delegate.
4. `update()` calls the local `paint()` method on the UI delegate, which performs the actual painting of the widget in the correct look and feel.

Usually, the painting process does not get triggered manually. All painting requests get queued and scheduled by an instance of `RepaintManager`, which will then trigger the actual painting process by calling `paintImmediately()` on the `JComponent` instance. In order to start the painting process, lightweight components need a platform independent `Graphics` object to paint themselves. Retrieving an instance of such a `Graphics` object can be done by calling `getGraphics()` on `JComponent`. The method `getGraphics()` recursively calls its parent component until a heavyweight component is found that gets a `Graphics` object from its associated native peer object. The `Graphics` instance is then passed down the component hierarchy and each component draws itself and its border before passing the object on to its children during the painting process.
We recognize that from an ownership perspective we again need the ability of ownership transfer to correctly pass the `Graphics` instance on to the component's children. After all, this is necessary if the widget layout has a deep ownership structure.

Other properties

Other `JComponent` fields worth discussing are the widget's fore- and back-ground color, its font, location, size, cursor, bounds, etc. The main commonality of all these properties is their value type semantic and their flyweight nature. Although all the fields have associated objects to capture the field's state, these instances are either shared by all components (shared flyweights) or strictly copied and are thus not passed to the outside of the component.

Shared properties In the case of a component's color (`java.awt.Color`), cursor (`java.awt.-Cursor`), or font (`java.awt.Font`), we observe that although the programmer can create its own objects and set them directly on the widget, in the standard case the `UIManager` is requested to deliver appropriate instances, based on the set `UIDefaults` instance. Hence, color, cursor, and font objects are shared flyweights where the `UIManager` acts as the flyweight factory that delivers all instances. As proposed in the flyweight pattern's ownership discussion (6.6), the Swing component should therefore only maintain a readonly reference to these fields, ensuring that shared color, cursor or font instances do not get accidentally modified by a certain component.

Unshared properties In the case of a component's location, size, insets or bounds we deal with unshared properties for which each Swing component has its own unique property object. An investigation on how these properties are retrieved or set on the component reveals that the Swing programmers consequently prevented any capturing or leaking of these objects. We therefore propose to declare the component as the owner of these fields in order to let the ownership system statically ensure that potential leaking or capturing does not occur.

A special case is the component's border property. On the one hand, we observe that the `Border` instance leaks and gets captured by the component's getter and setter methods and that the `UIManager` provides methods to retrieve certain predefined border objects, all evidences for a shared flyweight property. On the other hand we notice that most border objects are directly created by the `setBorder()`-method's caller, sometimes the concretly used border is even defined as an inner class of the caller. After all, the `JComponent` instance needs read/write reference rights to the border object in order to call the non-pure `paintBorder()` method during a paint request. Thus, the ownership system needs to provide the notion of a shared context so that all widgets can access the global border instances in a read/write manner upon a paint request. Additionally, the ownership system should provide the ability to transfer custom border objects into the widget's ownership context when setting the property.

10.4 Feasibility and Conclusion

We can conclude that the proposed deep ownership structure in the Swing framework results in major design benefits, but also leads to high requirements for the used ownership system.

First, the system needs to provide an ownership transfer mechanism to move allocated objects like the UI delegate or the Swing model instances to the widget's ownership context or to move the `Graphics` instance along the composite structure during the painting process. Second, the system should allow owned objects to access their owner in a read/write manner in order to be able to implement the observer pattern between the widget instance (the owner) and its model (the owned object). Third, the system should provide a notion of a global context in order to allow references from all widgets to the global look and feel objects. Above all, the system needs to grant special aliasing rights for action objects in order to still be able to reflect domain layer changes and correctly respond to user events.

If an encapsulation of the model instance is not an option, all other objects associated with a widget, such as the UI delegate, the renderer, etc. should not be encapsulated either. This is due to the highly interconnected object structure of a Swing component: the UI delegate needs to reference the renderer and registers itself as an observer on the model.

10.5 Impacts on the accounting application

With the proposed Swing ownership structure in mind, we will review the accounting application from Chapter 9 again. Figure 10.4 visualizes the simplified object structure of the accounting application with the proposed Swing ownership structure.

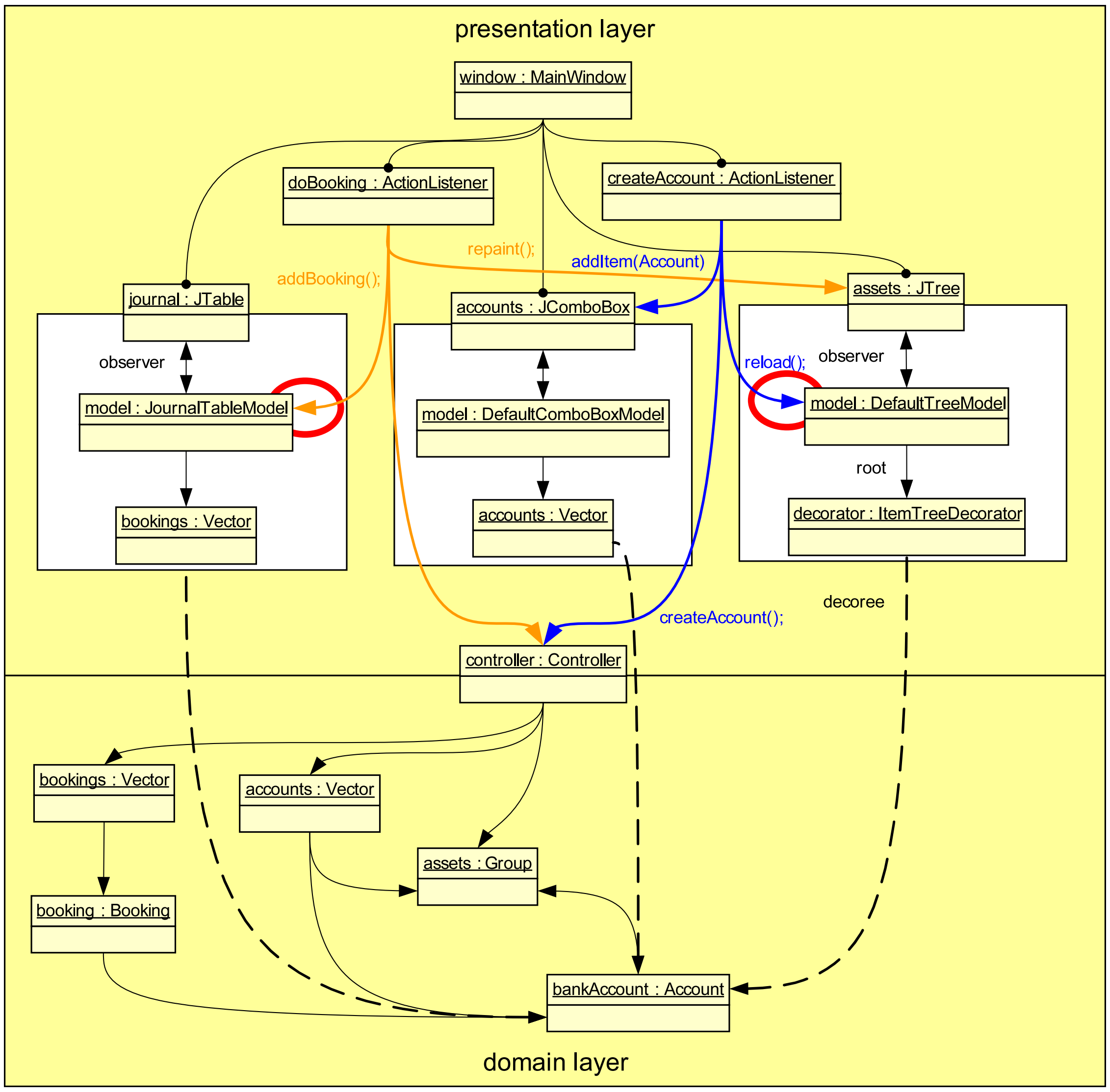


Figure 10.4: Simplified object structure of the accounting application. Readonly references are represented as dashed arrows whereas read/write references are solid arrows. Solid rectangles represent ownership contexts.

Notice the orange colored arrows which illustrate the most important method calls performed when executing a new booking. All bookings are stored in the journal and visualized by the `JTable` component. Following the methodology of Swing, new table entries (i.e. upon entering a new booking) are added directly on the model instance which will then notify the `JTable` instance through the established observer pattern. As already pointed out, an implementation of the proposed ownership structure in Swing requires special access rights for the `ActionListener` instances.

The second example where we notice this requirement in conjunction with the accounting application is when creating a new account, represented as blue colored arrows in 10.4. The creation of a new account results in a different account structure and therefore, the model of the `JTree` component changes which requires a repaint of the control. In order to inform the tree control about such a change, the action object has to trigger a `reload()` directly on the model.

To sum up: the proposed Swing ownership structure is, if supported by the ownership system, feasible in the context of the accounting application and would lead to a much clearer aliasing structure and control flow with the result that reasoning about code correctness is simplified.

11 Conclusion and Future Work

The concept of ownership succeeds to enhance many pattern implementations in general. Having the possibility of alias control enables local reasoning about code correctness for each pattern implementation separately. Furthermore, some pattern design properties can be statically proved to be correct. Object encapsulation succeeds to emphasize certain pattern aspects and the concept of readonly references enables a safe sharing of objects.

However, the ownership systems reviewed in this thesis still lack the necessary flexibility to successfully tackle all posed design scenarios. A list of the main problems has been presented in Section 8.2.

Although similar in concept, practical usage of the three reviewed ownership systems has shown some differences. In comparison, and the writer's personal impression, the Universe type system currently provides the best tool support. All Universe type system examples have been verified with the available command line tool. Although we experienced some checker problems and crashes in conjunction with Java inner classes, the tool was of great help to verify typing propositions. Advantages of the Universe type system are clearly its intuitive semantics, little annotation overhead, and the support for readonly references.

The relaxed aliasing restrictions, such as the ability to reference any ownership ancestor and its directly owned objects, enables Ownership Types to successfully cope with many encountered design scenarios. However, aliasing permissions of an inner class to access its declarator has not shown very useful apart from the presented iterator idiom. An inner class implementation is not desired or justified in most situations, even with highly collaborating classes. Ownership Types remains a theoretical concept as long as there is no tool support to enable practical usage.

At first glance, the Ownership Domains concept looks very intuitive and its concept is quickly understood. In our experience however, providing a correct ownership annotation can be a very tedious task, even in small and fairly simple programs. Moreover, the delivered tools are not feature complete yet and important aliasing permissions have not been checked correctly. It is to point out though that an ownership typing at design pattern level is presumably too subtle for Ownership Domains. We believe that the system is especially useful when providing a coarse-grained ownership structure on an application level to ensure architectural constraints.

In order to support most software engineering best-practices, it might become necessary for all systems to give up static type checking in favor of being able to type more designs. One could think of ownership systems that statically ensure certain properties; other properties need optional run-time checks that rely on unit tests with a high code-coverage ratio.

Based on the detected problems in this thesis, an ownership system, capable to type all patterns, provides the following features:

- Alias control for owned objects.
- Readonly references with the possibility to suppress any aliasing.
- Ownership transfer, support for delegation of object-creation.
- The possibility to declare multiple ownership contexts for a single object.
- "Friend" contexts that either allow references from all objects, objects of the same type, or explicitly authorized instances.

Future work in this area may include the verification of the encountered problems in other designs and application scenarios, as well as a feasibility check of the requested features. In the end, ownership system design will turn out to be a tightrope walk between simplicity and capability.

In order to be used in practice, an ownership type system should

- define little annotation overhead
- provide intuitive semantics
- be capable to type common application designs

Most likely, already a simple mechanism for ownership transfer will solve many problems encountered today.

12 Related Work

In this chapter we will briefly discuss related work that has been done in the area of alias control, especially looking at application case studies.

12.1 Practical Usage Studies

Since ownership theory was proven to be sound and looked very promising on small examples (e.g. ownership typing of a data structure), the probing question was if ownership could also be easily applied to larger, industrial applications. Several Master theses tried to answer this question. T. Hächler proposed in his Master thesis an ownership typing for an existing industrial application [Häc05]. G. Cele and S. Stureborg applied Ownership Types to several demo applications and examined how the concept of Ownership Types perform in practice with respect to size, complexity and interconnection of objects [CS05].

12.2 Miscellaneous

The concept of ownership and corresponding type systems are heavily researched and besides the three presented systems, many other propositions on beneficial aliasing policies, ownership transfer, and alias control in general have been developed. We present three interesting research projects in the following.

Confined Types

The concept of so-called confined types[VB01] introduces statically checkable inexpensive syntactic constraints with the aim to restrict uncontrolled alias spreading. The idea is to annotate class or interface definitions with the keyword *confined* in order to restrict aliasing of these types from outside the protection domain. Java packages are adopted as protection domains to take advantage of access modifiers. The advantage of confined types is that only little annotation overhead is needed and everything is statically checkable. The problem is its modularity and granularity. Only whole classes and interfaces can be declared as confined and the granularity is on a package level. This is too coarse-grained for many application scenarios, especially for ownership typing of design patterns. Still, there are interesting usage scenarios for confined types. Clarke, Richmond and Noble presented in [CRN03] an object confinement discipline based on confined types in order to statically check if the Enterprise JavaBeans (EJB) framework's architectural integrity constraints hold. Basically, confined types are used to ensure that no access to internal objects implementing a bean's functionality occurs from outside. All bean clients have to use the bean's interface so that transaction management and security is preserved. Confined types can help in this application scenario to prevent representation leaking of the Enterprise JavaBean.

External Uniqueness

Dave Clarke and Tobias Wrigstad argue in [CW03, CW02] that a property called *external uniqueness* is enough in conjunction with Ownership Types to support ownership transfer of aggregates. The originally proposed Ownership Types concept from Clarke, Potter, and Noble [CPN98] offers so-called deep ownership that satisfies the owner-as-dominator property. Let us consider an ownership graph where nodes represent objects and edges represent references. The owner-as-dominator property guarantees that every access into a data structure must go through the owner. External uniqueness means that there only exists *one* external references to an object, while there may still be arbitrary internal aliases. An externally unique reference corresponds to a dominating edge in the ownership graph. Unique references from the owner to its children need to be annotated with the keyword `unique` and the type system must ensure that uniqueness always holds. If there are no references from inside an aggregate to objects outside, a safe *movement* or *borrowing* of the whole aggregate within the ownership tree is possible, if only it is guaranteed that the reference to the aggregate stays unique. Therefore, the source reference must be nullified when the structure is moved (or borrowed) to the destination's unique reference.

Encapsulation and Aggregation in Eiffel

Stuart Kent and Ian Maung introduced in [KM95] an ownership system for Eiffel. Basically, they introduced new keywords such as `private` and `protected` to describe the owner of objects attached to entities (attributes, local variables, formal arguments, `Current` and `Result`). Each entity has a proprietor which is either `Void`, in case of a `public` entity (default), `private` if the entity's declarator is the proprietor, or `protected` if the proprietor of the declarating object should also be the proprietor of the entity. The authors claim that the proprietorship of an entity cannot be determined statically in their system and therefore illegal assignments are caught at run-time. Another disadvantage is that run-time checks result in an execution overhead. The authors therefore suggest to only enable ownership checking during system tests and not in the delivered system.

Bibliography

[AC04] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 2004.

[BDM+04] M. Barnett, R. DeLine, M.Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6), 2004. www.jot.fm.

[BLR04] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2004.

[BLS03] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, pages 213–223. ACM Press, 2003.

[BR01] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[CPN98] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 18–22 1998. ACM Press.

[CRN03] D. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: deployment-time confinement checking. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 374–387, New York, NY, USA, 2003. ACM Press.

[CS05] G. Cele and S. Stureborg. Ownership types in practice, 2005.

[CW02] D. Clarke and T. Wrigstad. External uniqueness, 2002.

[CW03] D. Clarke and T. Wrigstad. External uniqueness is unique enough, 2003.

[DM05] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005. To appear.

[ELW98] R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly, 1998.

[Gam99] E. Gamma. Swinging on the bleeding edge. http://www.old.netobjectdays.org/pdf/99/jit/gamma.pdf, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Häc04] T. Hächler. Static fields in the universe type system. 2004.

[Häc05] T. Hächler. Applying the universe type system to an industrial application, 2005.

[HLW+92] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[JML] The Java Modeling Language. http://www.cs.iastate.edu/~leavens/JML/.

[KM95] S. Kent and I. Maung. Encapsulation and aggregation. In *TOOLS Pacific*, volume 18, 1995.

[LC05] G. Leavens and Y. Cheon. Design by contract with JML. http://www.jmlspecs.org, 2005.

[LM04] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[Mey92] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, November 1992.

[Mul] The MultiJava Project. http://multijava.sourceforge.net/.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[Mül05] P. Müller. Konzepte objektorientierter programmierung. http://www.sct.ethz.ch/teaching/ws2005/koop/, 2005.

[Nob00] J. Noble. Iterators and encapsulation. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 431, Washington, DC, USA, 2000. IEEE Computer Society.

[Sch04] D. Schregenberger. Runtime checks for the universe type system. 2004.

[VB01] J. Vitek and B. Bokowski. Confined types in Java. *Software-Practice and Experience*, 31(6):507–532, 2001.