Nil and None considered Null and Void

Thomas Kühne (kuehne@isa.informatik.th-darmstadt.de)
Department of Computer Science, TU Darmstadt
Magdalenenstr. 11c, D-64289 Darmstadt

Abstract

Object references in object-oriented languages suffer from their pointer heritage. In this paper we discourage the use of a single *magic* value, shared among all reference types, to denote a non-referencing state. The advantages of *void values*, which include uniform treatment of void and non-void data, provision of default-behavior, termination of recursive definitions, and gracefully dealing with error situations are presented in pattern form. We then investigate the impact of the Void Value pattern for language design, extending the applications to implicit creation, abstract method execution, and simplified module usage. Finally we point out related mechanisms like pattern matching in functional programming languages and compare *void values* to *nullcases*.

1 Introduction

Objects are mutable [30]. Objects are frequently passed around and their state is modified by other objects. In order to achieve a natural form of communication and to maintain efficiency, often the notion of sharing is used. As a result, object variables are analogous to pointers in more traditional languages. Yet, object pointers may not point to arbitrary entities like at other pointers nor do they support pointer arithmetic (e.g., possible in C [15]). As a further difference, object pointer variables are typically polymorphic entities, i.e., they are allowed to point to an object of a particular class **and** all its subclasses. For these reasons they are typically referred to as *references* rather than *pointers*. Several object-oriented languages make references transparent to the programmer (Smalltalk [14], Eiffel [20], Self [28]). They appear as normal variables without extra dereferencing and address operators. Hence, pointer semantics only occurs implicitly.

However, the object reference still seems to carry an ancient relic, inherited from its pointer ancestry: Object references, just as pointers, possess a default magic value, known under various names as None (Simula [9] & Beta [19]), Nil (Smalltalk [14]), Null (C++ [27, 12]), and Void (Eiffel [20]). Even a language based on prototypes like Self uses a special `nil` value [28]. Nil is *magic* in two flavors. First, it is a polymorphic value common to all reference types. For instance, Eiffel accounts for this, by making Void of class `NONE` which inherits all classes by definition[1]. Second, Nil is used to account for references that do not reference any object. Hence, it could be considered an exception. It does not appear as such, since the exception is encoded in the reference's co-domain.

We question this convention by pointing out the disadvantages of a single untyped void value that does not support any behavior. Rather than treating the non-referencing state of references as an exception we propose to conceptually avoid it at all. We replace Nil with a real value called *void value*. Section 2 describes the usefulness of explicit void values in the form of a design pattern, as introduced in [13]. Section 3 discusses the impact of this pattern for language design. Section 4 points out related concepts and Section 5 concludes.

2 Design Pattern: Void Value

2.1 Intent

Raise Nil to a first-class value. This allows to treat void and non-void data uniformly, is a way to provide default behavior, facilitates the definition of recursive methods, and enables to deal with error situations more gracefully.

[1] While this repairs the situation, it appears to cure a symptom rather than the disease.

2.2 Motivation

Classes are often regarded as representing abstract data types (ADTs). Typically, the *empty* constructor of an ADT is translated to Nil. For instance, an empty `BINARY_TREE` is represented by Nil and a tree leaf is represented by setting both child attributes to Nil. It is tempting to identify *empty* constructors with the special pointer content Nil, since there are some reasons in favor of it:

- In analogy to *empty*, Nil does not support any view operations.
- Nil does not waste any storage.
- Most languages initialize references to Nil, i.e., provide the empty case by default.

However, such a design decision puts a great burden on the clients of the ADT. Before the client can apply operations on a tree, it must check whether it is void or not. Otherwise, invoking a method through a Nil reference would produce a runtime-exception or even -error. Typical (Eiffel) code[2] is:

```
Result:=  (tree /= Void) and then
          tree.has (v)
                    or
if tree /= Void then
   io.putstring (tree.out)
end
```

Code example 1: Robust behavior.

We can replace the above code with the underlined parts only, if we abandon Nil (Void) and replace it with an extra class (`VoidTree`) (see Figure 1).

Now we can define `VoidTree` to return `false` on a `has` message and to produce no output on `out`. In fact, we applied the general pattern to replace case analysis (testing for Nil) with dynamic binding (provision of an extra type constructor VoidValue). As a result many such `if` statements as above can be removed from client code.

An important special case of robust behavior is a Null iterator (also see Iterator [13]). Instead of

```
if set /= Void then
   loop over all set elements.
```

[2]All code examples in this paper have been extracted from a commercially available library; identifiers have been adapted for presentation, though

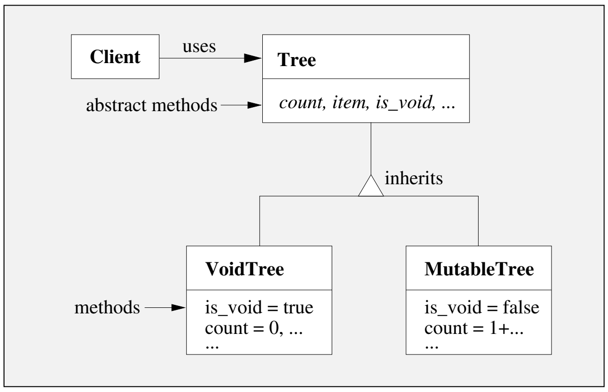

Figure 1: Empty tree as void value.

one should use internal iteration, e.g.,

```
set_iterator.do_all
```

Code example 2: Null iterator.

and let void sets do nothing on iteration.

Answering queries to void values with default values again leaves the underlined code part only.

```
if tree /= Void then
   max_plane := tree.max_plane
else
   max_plane := Default_max_plane
end
```

Code example 3: Default Behavior.

This obviously improves the treatment of trees from a client's perspective, but there is more to gain. Let us `count` the number of nodes in a tree:

```
Result:=1

if left /= void then
   Result:=Result+left.count
end

if right /= void then
   Result:=Result+right.count
end
```

Code example 4: Base case definition.

The recursive calls must be guarded with `if` statements when tree leafs are represented with Nil. If void tree values return 0 as their `count`, then the code becomes much more readable and better expresses the function's spirit:

```
Result:=1+left.count+right.count
```

The provision of base cases by void values is a special case of the general scheme to distribute cases. Complicated nesting, like

```
equal(me: ANY; you: ANY): ...
Result:=(me=Void and you=Void)
or else ((me/=Void and you/=Void)
and then me.is_equal(you))
```

Code example 5: Case distribution.

becomes `Result:=you.is_void` (in class `VoidAny`) and `Result:=is_equal(you)` (in class `AnyObject`) respectively. Note that `obj.is_void` is different to `obj = Void`. The former tests for voidness as well, but allows the representations for void objects to be changed. This is useful for multiple void values as well as for standard objects which just happen to be in an empty or void state.

It is worth mentioning, that we may provide void trees, -nodes, and -leafs independently with possibly differing behavior. See section 4 for an analogous example when this is useful. Also note that while void values are a natural concept for container classes their applicability extends to all types of classes. For instance, the `subordinates` method of a `VOID_EMPLOYEE` can return an empty list. Its `salary` method may create a `no-valid-employee` exception, return `0`, or a `VOID_SALARY` instance, depending on the desired semantics. All three possibilities may coexist in separate void values. Application parts can individually choose to inject the appropriate void value into server calculations.

2.3 Applicability

- *Uniformity.* Clients benefit from Void Value by uniformly treating void and non-void data. Inquiring information and invoking behavior can be done independently of the data's void state, without the eventual need for creating initial instances first.

- *Default behavior.* Void Values allow to specify default behavior. In contrast to Nil, a Void value may specify results and behavior for any of its interface methods. This is useful for providing reasonable behavior for uninitialized data and void result values.

- *Error handling.* Void Value can be used as an exceptional value [8] when most of the code should not deal with error situations. An exceptional value may either behave properly indicating an error situation (e.g., by describing the error when by default displayed to the user) or can be caught by error checking code at the application's top-level code.

- *Termination.* Use Void Value to relieve recursive definitions from testing the base case. A recursive call can then be made regardless of possibly void arguments. Accordingly, the definition for the base case can be given at the base values (i.e., void values) instead of one step before. Here, Void Value plays the role of a first-class terminator.

2.4 Structure

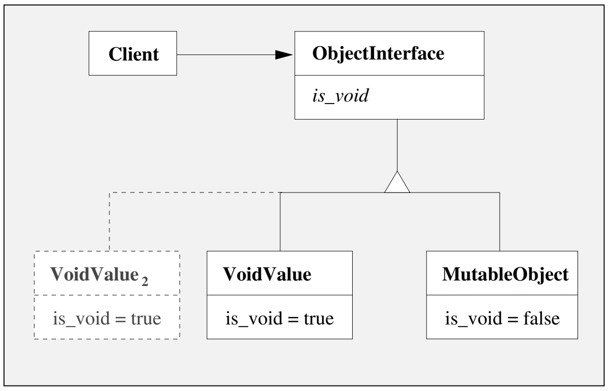

Figure 2: Void Value structure.

2.5 Participants

- **Client**
  - accesses both MutableObject and VoidValue through ObjectInterface.
- **ObjectInterface**
  - provides the common interface for MutableObject and VoidValue.
  - may provide common abstract[3] behavior to MutableObject and VoidValue.
- **MutableObject**
  - defines the standard object behavior.

[3] Concrete methods do not rely on state (attributes), but on an abstract interface, only.

- introduces attributes for object state.
- does not care for recursion base cases, default- and error behavior.

- **VoidValue**
  - replaces Nil.
  - defines base cases for recursion, default- and error behavior.

2.6 Collaborations

- Clients use the ObjectInterface to manipulate void, non-void, default, and error objects. If MutableObject extends the interface of ObjectInterface, clients may maintain specialized references to MutableObject, provided they are guaranteed to receive objects of type MutableObject only.

2.7 Consequences

- *Abstraction.* Void Value abstracts from the implementation detail to represent void or uninitialized data with Nil. Clients, therefore, are relieved of the need to treat data with Nil differently from data which does not use Nil for its representation.

- *Object-Orientation.* Checking for Nil, i.e., case analysis, is replaced with dynamic binding. This moves all case analysis from the program to a single distinction between normal *object* and void *value*. Object behavior must be defined at two places (object and void value). One the one hand, this allows to combine several void values with a single object. On the other hand, changing a single method might mean to change two class definitions (object and value class).

- *Efficiency.* It is more storage friendly to use void values, instead of full blown objects that simply have the state of being empty or uninitialized, but carry the overhead of unused attributes. Note, however, that void values are not as easily turned into objects again without support for "Implicit creation" (see sections 2.8 & 3).

  Depending on the implementation of dynamic binding a calling overhead may be incurred, compared to `if` statements. Note, however, although no call must be made for void trees, in the traditional solution of Code example 4, at least $\frac{arity^{depth}-1}{arity-1}$ non-void trees (inner nodes and non-void leafs) are unnecessarily checked for being void!

- *Immutability.* If void values should not need more storage than Nil, they have to be immutable. MutableObjects without attributes just claim code space for their methods once, and possibly a little storage for housekeeping mechanisms such as RTTI (Run Time Type Identification) and object identification.

- *Separation.* Program code relying on Void Values describes the standard case only. In analogy to language support for exception handling framework code or algorithm descriptions thus become easier to write and read (see Code examples). Handling of errors, denoted by void values, can be done in a few places at the application's top level. Lower level code can be designed to smoothly pass error values around, until they are identified by the top level.

- *Implicit initialization.* In order to avoid any occurrences of Nil, one must tie the creation of objects to the declaration of references. See section 2.8 for suggestions how to automatically achieve initialization to void values.

- *Initialization Errors.* Exceptions, caused by the application of operations on Nil, are dealt with more gracefully. Instead of triggering an exception or halting the system a possibly inappropriate behavior is executed, but users are still able to continue the application. The downside of this is that a missing initialization might go unnoticed, only to be discovered as unexpected application behavior very much later in the execution. According to the experiences of the UFO project [23], however, unnoticed initialization has not been found to be a problem in practice. Additionally, it is almost as easy to pass Nil around undetected. Error messages simply become more meaningful with void values [24]. If the void value is one among several alternatives for a type, its identity allows to better trace back its origin.

2.8 Implementation

- *Interface extension.* In analogy to the Composite pattern [13], MutableObject might extend the interface of ObjectInterface, in order to provide operations that make no sense to VoidValue. Each such extension, however, will minimize the applicability of VoidValue to play the role of an error value. Clients using the extended MutableObject interface, will not be able to transparently work on void values too. In this context, it is better to have the full interface at ObjectInterface and to define exception (e.g., error reporting) methods in VoidValue respectively.

- *Storage requirements.* Any behavior that goes to ObjectInterface should rely on abstract state only, i.e. "Implement behavior with abstract state" [2]. Any attributes will incur a space penalty on VoidValue.

- *Value Initialization.* If representation of a complex constant (requiring creation effort) is more important than preserving minimal space requirements, then a void value may calculate the constant information (possibly taking creation arguments) and store it in attributes. Accordingly, VoidValue is best implemented as a Singleton [13] in order to allow sharing of the constant data.

- *Multiple Inheritance.* In statically typed languages VoidValue needs to multiply inherit from its interface class and another void value class, if code reuse is desired between void value classes. One of the inheritance paths is used only for interface inheritance though, unless the interface classes implement common behavior for VoidValue and MutableObject. In the latter case the language should properly support repeated inheritance of code (e.g., as in Eiffel).

- *Reference Initialization.* The value to be gained from Void Value partly depends on how thoroughly we can eliminate any Nil values from program execution. To this end, one may replace standard references with *default-references*, akin to smart references [11].

  A DefaultReference (see Figure 3) is a value (e.g., non-reference type in C++, expanded type in Eiffel, part-object in Beta, etc.). Consequently, it is initialized by declaration, i.e., can not take the value Nil. After initialization it delegates any message to VoidValue by default. Hence, any usage of DefaultReference without prior initialization results in applications to VoidValue, instead of Nil. Ultimately, DefaultReference will be initialized (e.g., using a method called `make`) to reference a MutableObject. In the structure of Figure 3 DefaultReference plays the role of an ObjectProxy. Implementation issues, such as using overloading of the member access operator (`->`) in C++, are discussed in the implementation section of Proxy [13]. Note that using a DefaultReference involves a delegation overhead for each access to either VoidValue or MutableObject. This might be a price too high to pay just for gaining automatic initialization of references.

  Of course, we can achieve automatic initialization of MutableObject by directly declaring it to be a value, resulting in a much simpler structure and no delegation overhead. Yet, this is possible only if value semantics (e.g., no sharing, no efficient updates) is intended [18, 10].

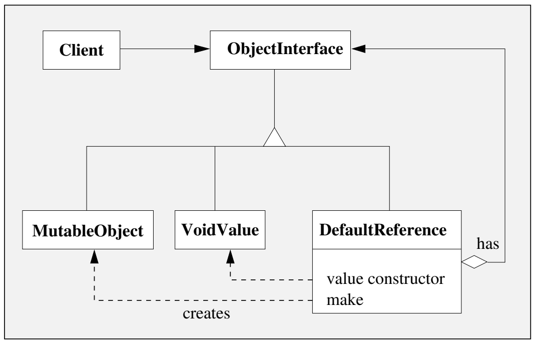

Figure 3: Automatic reference initialization.

- *Implicit creation.* Smalltalk allows to change the type of an object without affecting its identity. Ergo, a void value may change to an object and vice versa. If a void value can not deal with a message it may delegate it to its associated object and then `become:`[4] the object itself. Hence, there is no need for ex-

[4]Smalltalk's method name to replace objects.

plicitly creating objects anymore, since void values can be made responsible for this. In the absence of a `become` mechanism, implicit creation can still be achieved by "Reference Initialization" (see above).

2.9 Related Patterns

In the following we relate Void Value in terms of its nature, implementation and, collaboration potential to patterns to be found in [13].

2.9.1 Classification

Composite Both Composite and Void Value provide uniform access to components (value & object) of a structure (type). A VoidValue plays the role of a special immutable component leaf. As a difference, Void Value does not involve issues of child management.

Singleton A void value is a Singleton, in that it can be shared by all objects of a type. As it is immutable, there is no need to have multiple instances. (see section 3.4 for an exception).

Flyweight VoidValue can be regarded as a ConcreteFlyweight, as it holds intrinsic data and is shared among all clients that use it.

State When used as described in *Implicit creation* at section 2.8, VoidValue and MutableObject play the role of ConcreteStates, representing the void and non-void states of data.

2.9.2 Implementation

Proxy DefaultReference (see Figure 3) works as a Proxy since it keeps a reference to an object whose interface it shares.

Bridge DefaultReference behaves like a bridge, in that it shields the clients from the two "implementations" VoidValue and MutableObject.

2.9.3 Collaboration

Abstract Factory, Factory Method A void value can be used as a return value for not available products, e.g., unsupported widget types with regard to a specific window factory.

Iterator, Visitor, Chain of Responsibility All these patterns are based on iteration and Void Value can help to define their termination behavior, analogous to a base case in recursion.

Memento A Void Memento can be used as a default and reset state.

Command Void Command may stand for default or idle behavior.

State, Strategy Void State may represent the initial or an error state. While used like Void State, Void Strategy would allow Strategy-Context creation without a Strategy selection.

Template Method While *subclass responsibility* [14] normally requires at least one concrete subclass, Void Value can be used as a default subclass, allowing the execution of abstract methods. Of course, Void Value is just a special concrete subclass, so this collaboration is a lot more dramatic with language support for void values (see section 3).

3 Impact on language design

It is well known that one language's patterns are another language's features [13, 3]. Especially Void Value's heavy interaction with other patterns and the increase of safety to be gained, makes it a hot candidate for becoming a language feature.

We have seen that trying to achieve automatic initialization of references to void values involves some additional complexity (see Figure 3) and introduces both storage (for DefaultReference) and computation (for delegation) overhead (see *Reference Initialization* in section 2.8). Consequently, language semantics should not initialize references to Nil, but to their associated void values. As a result, it would become easier to force program execution to "stay in the game", i.e., stay within the domain of language semantics without creation of runtime aborts or implementation dependent effects [1]. With regard to the caveat of unnoticed errors, introduced by not stopping on not explicitly initialized data (see *Initialization Errors* in section 2.7), one may still provide no, or exception generating behavior, for void values, in order to force a brute but early detection of unintended void values.

Applications for Void Value that go beyond what most languages allow to emulate are:

3.1 Initialization by declaration

Even if it is possible to tie initialization (assigning useful default values) to creation (attaching an object to a reference), Nil values still exist "between" declaration and initialization. Void Value allows to transfer the concept of automatic initialization known for value types to reference variables, since even variables of abstract type can be attached to a proper void value. An abstract class definition thus not only provides an interface and template methods, but optionally also a void value description.

Beyond the implicit creation of void values, one may extend the concept to allow automatic initialization to any object. In analogy to the `creation` clause for methods in Eiffel, one may select a creation subclass to be used for implicit reference initialization. In contrast to void values, the thus created instances, could be used to capture and hold state right away.

3.2 Module support

Classes are sometimes used as modules. For instance, the `BASIC_ROUTINES` of Eiffel are simply a collection of routines. One Eiffel idiom of using a module is to declare a variable with expanded (value) type, which avoids the creation of a module instance. If modules are void values, then neither creation nor declaration as an expanded type is necessary to use the module. Void Value automatically works as an (stateless) Singleton for module access. Note that while we agree that in particular the methods of `BASIC_ROUTINES` should be distributed to their appropriate argument classes there are other applications for modules in object-oriented programming like namespaces, class clustering, and collection of true free functions.

3.3 Abstract method execution

If abstract classes provide void values then execution of their abstract methods becomes feasible. The void value plays the role of a concrete subclass, needed to fulfill *subclass responsibilities* [14]. Taking into account that void values are no real instances in that they can not hold state, their use as subclass surrogates seems to be of very limited utility. Yet, it may expand the scope of prototyping and testing of incomplete software. Also, Function-Objects [7, 16, 17] that do not need to keep any free variables or arguments internally (e.g., for currying) may be used without the need for a previous, formal creation of an object instance. In these cases, the instances are created only to allow execution of the Function-Object body.

3.4 Implicit creation

One key difference between *empty* constructors of ADT specifications and Nil, is that the former allows constructor application, while the latter does not. Instead of forcing the programmer to guarantee the creation of an object instance in advance, it appears very useful to allow implicit creation of object instances. Note that "Initialization by declaration" refers to the automatic attachment of references to void values, while "Implicit creation" refers to the automatic conversion of void values to objects. Whenever, a void value can not handle a message (e.g., state has to be accepted), it replaces itself with an appropriate object. Such an automatic conversion would make explicit creation calls superfluous:

```
if array.item(key) = Void then
  !! set.make
  input_array.put(set, key)
end
array.item(key).put(data)
```

Code example 6: Initialization.

In the presence of sharing the replacement of the above code with the underlined part is correct only if previous references to the void array member `set` will reach the freshly created object instead. The change of an object's type (value to object) is usually not supported by programming languages (Smalltalk being an exception (see *Implicit creation* in section 2.8)). Note, that implicit creation demands void values to possess identity. Otherwise, it is not clear which references to, e.g., void sets, should be affected.

"Initialization by declaration" and "Implicit creation" can work together to implement *Lazy Initialization*, which avoids exposing concrete state and initialization overhead, allows easy resetting, and provides a proper place for initialization code [2]. Auer's solution requires to test variables for Nil values, i.e., for being not yet initialized. Automatic initialization of references allows to apply messages to void values (instead of Nil), whereas automatic conversion to objects allows to implicitly create and provide results on demand. The latter step, may not even be necessary if the required results do not need to hold state. As long as no object functionality is needed void values suffice and do not create storage overhead for "empty" objects.

4 Related Work

Where is the place of void value in computer science? If we take Nil as a starting point in an evolution, we note that Nil is a value, i.e., immutable and lacking identity, supporting no operations (except identification of itself). If we add operations, we arrive at the Smalltalk model of Nil [14]. Smalltalk provides a class `UndefinedObject`, which has a sole instance accessible through `nil`. This is useful for catching calls to uninitialized references. We almost make a quantum leap, if we allow a special `nil` for each type, i.e., introduce void values. Now we can define "undefined object" behavior for each type individually, which allows us to specify much more specific and therefore useful behavior.

4.1 Procedural vs. type abstraction

Adding type and behavior to Nil, enables us to switch from *type abstraction* to *procedural abstraction*. Type abstraction is used in ADT specifications (observations share a hidden representation and cope with supplied constructors). Procedural abstraction is used in object-oriented programming (constructors define observations independently and cope with requested observations) [6]. An ADT observation (method) uses case analysis to distinguish between internal constructors, including Nil. In object-oriented programming the observations are given for each constructor separately and selection is done implicitly by dynamic binding. Hence, moving all behavior residing in case analysis (testing for Nil) to dedicated void types, corresponds to transforming ADT- into object-oriented style. Code example 5 clearly shows the increased readability of the object-oriented version, as it avoids nested case analysis. Most object-oriented languages disguise the difference between *type abstraction* and *procedural abstraction*, as they support both styles [6].

4.2 Functional Programming

Pattern matching in functional programming also separately defines behavior for distinct cases [4]. For instance, the definition of 0! can be given independently from the definition of $n!$, *where* $n > 0$. In order to dispatch in a similar way on values in object-oriented languages, we have to turn the values into (sub-) types, for the reason that dynamic binding works on types, instead of values. A void value is exactly a value turned into a type in order to allow implicit case analysis, like pattern matching in functional programming. Unlike pattern matching, Void Value does not break encapsulation by revealing constructors, i.e., representation [29, 5], unless clients are enabled to check for concrete void values. If access is allowed through the procedural object interface only, clients do not care for the actual constructors.

4.3 Checks pattern language

The utility of representing exceptions by values is demonstrated by the design patterns EXCEPTIONAL VALUE and MEANINGLESS BEHAVIOR [8]. Error- or quantification defying values, are not subject to be caught everywhere in the application. Instead, most methods are written without concern for possible failure. Therefore, exceptional values are smoothly passed through methods, to be caught by top level handlers. Void Value acts in exactly the same spirit as the above patterns and would fit well into the CHECKS pattern language [8].

4.4 Void values vs. nullcases

The UFO (United Functions and Objects) language also assigns each type its own distinct void value, but "... the special case of matching an empty list, or other null object, is handled by the `nullcase` construct or by explicitly testing with `is_null`." [23]. So, UFO uses dynamic binding in general, but pattern matching for nullcases. UFO avoids decreased readability of functions caused by (nested) case analysis by a special syntax, e.g.,

```
length: Int is
nullcase:  0
otherwise: 1 + self.tail.length
```

Figure 4: UFO function definition.

As this pattern matching decision concerns only one special constructor, the difference between void value binding and nullcase matching does not have the size of the fundamental dichotomy between type- and procedural abstraction. Yet, there are some notable differences:

As a consequence of the encapsulation breaking properties of pattern matching, function definitions in UFO are sensitive to the introduction and change of null values. For instance, if a set of functions is designed to always operate on non-null lists and later it is discovered that indeed null lists may also occur, then all function definitions must be changed to include the nullcase branch. In contrast, methods of an object do not need to be changed, since they operate on a non-void object per definition. The necessary addition here consists of adding one `VoidClass`.

Similarly to the introduction of null values, it may occur that a former null value should be transformed into a proper object, just as enumeration types are sometimes transformed into class types [25]. Consider a tree representing terminal leafs as void values. Now, we want to enhance the tree with a display method and therefore add attributes to the leafs, which store the display position[5]. In fact, the example describes the conversion of Void Value to Composite. A void class is easily converted into a standard class, but `nullcases` will not match object instances and must be repackaged into a class definition.

On the other hand, the introduction of a new function or the change of an existing one is more local with the UFO approach. Nullcase and the regular case are edited at one place (the function definition) whereas one may need to visit both void value and object class, in order to do the change. With both cases at one place it is also easier to see what the function does as a whole.

Nevertheless, the extraction of the nullcases into one class description produces an entity of its own right:

- The void value class acts as a compact documentation of base case-, default-, and error-properties of a particular type.
- When trying to understand a type, looking at the void class immediately communicates the set of methods that create exceptions or implement implicit creation or delegate to other definitions, etc.
- With all null properties concentrated at one place it is easier to check for consistency, e.g., to make sure that a terminal node does not answer *count* with zero, but *leaf* with true.
- The void value class provides a place for (private) initialization functions, needed and possibly *shared* by void behavior methods.
- Changing the void-behavior of a set of functions is regarded as providing new or changed void data, without changing the function definitions.
- It is possible to have multiple void values and to dynamically choose between them. With the nullcase approach, each time a new set of functions (that must repeat the regular case) must be provided.

An example for the utility of separately defining null- and regular cases is the addition of numbers (assuming a void number shall behave like zero). The void value method simply returns the argument (the number to be added), regardless whether the regular case would have used integer, or complex arithmetics. Void complex values, thus may inherit this method from void integer values.

In turn, one may want keep the regular behavior of a type, but vary the exception or termination behavior. A fold (reduction operation) on an empty list may result in an error- or default value object, depending on which was given as the initial argument for folding.

UFO's lacking ability to redefine null- and regular cases independently is likely to be removed in

[5] If we do not want to treat leafs as Flyweights.

the future [24]. However, inheriting null- and regular cases separately, implies the loss of locality of a function definition. As in the case of void values, it will not be possible anymore to look at the full definition at once. Finding the actual definition becomes harder if redefinition in a hierarchy of definitions is allowed[6].

Even if inheriting and overriding of null- and otherwise clauses is possible, still the problem of consistently changing the behavior of null values remains. Consider a list of books found during a library retrieval. An empty result at the end of a non-empty list displays nothing and provides no actions. If we want to change that behavior into printing a help message (e.g., suggesting to broaden the scope of search) we can provide the action (invoking the help system on clicking) in the same void value class definition. In other words, changes to void behavior affecting more than one function are still local.

Note that void values are not restricted to procedural abstraction only. Either by providing a *is_void* or by using RTTI it is possible to check arguments or attributes for being void. That is, as in UFO it is possible to use the "voidness" of arguments (i.e., inspect their constructor) in order to do optimizations, such as returning the receiver of an `append` message if the argument to be appended is void[7].

An argument in favor of nullcases is that they do not produce inheritance relationships (see *Multiple Inheritance* in section 2.8). Also, they are possibly too rare in order to justify extra void class definitions. In 17K lines of code for the UFO compiler (without standard libraries) there are only about 200 applications for nullcases [24]. However, other projects may yield different numbers, and there should be no must to provide void classes. Any missing void class or method could mean to inherit the exception behavior of a root void class, akin to `UndefinedBehavior` in Smalltalk.

In summary there are good reasons for both approaches and a decision is probably made on the prospect of which disadvantages will be most annoying. Note that sophisticated tool support makes most differences between Void Value and nullcase clauses disappear. After all, both approaches just use different organizations to describe the same semantics. On top of both approaches class- and method viewers could be defined, which will allow (viewer-) local changes to method- or void value definitions. Consequently, nullcases would support null value documentation and their consistent editing too. Still, in order to transform null values into real objects, more sophistication in tool support than just View-Editors will be required.

Assuming anything else equal, there is also the motivation to use just one instead of two concepts for dynamically looking up definitions. A void value is simply a hierarchy sibling of standard objects and other base-, default-, and error values.

5 Conclusion

We have shown authentic code examples, which can be simplified by lifting Nil to a first class value. The key aspects of the solution are to assign each type its own void value and to use dynamic binding rather than case analysis. Although void values are *values* as opposed to *objects*, i.e., immutable, they can play useful roles such as default- and error values. Actually, the lacking identity of a void *value* allows it to be shared by all clients, which causes no or only little storage overhead, compared to Nil.

We were able to successfully use "Default References" to automatically avoid Nil, since they treat an unattached status as a reference property, rather than as an exceptional reference value. Direct language support for void values would allow to get rid of the additional overhead aligned with DefaultReferences. Furthermore, implicit creation, abstract method execution, and simplified module usage will be enabled by proper language support.

Main argument against void values are:

1. *Accesses to Nil correspond to exceptions and are better handled by clients.* Different Clients want to react to an exception individually and thus should be in control of the action. There are, however, many accesses to Nil which do not correspond to exceptions (see Code examples 1-5), but can be given a natural semantics with respect to the corresponding data type. If clients really need to vary the behavior of

[6]Eiffel in particular, provides the *flat* format for classes, that could be used to resolve all inherited and redefined methods of void value classes.

[7]With procedural abstraction only, one is forced to add each element of the receiver to the argument in any case.

void values they can do so by choosing the appropriate kind from multiple void values. Finally, void values still can be identified by case analysis so any client may provide its own actions on receipt of a void value.

2. *Errors are hidden and thus harder to detect.* Void values potentially can be passed around in applications, stored and retrieved, without causing errors. Eventually, they surface as unexpected application behavior and then it can be hard to find their origin. The easy way out is not to use void values, falling back to early and brute discovery of initialization failures. Alternatively, one can check for void value occurrences explicitly with case analysis, in order to "pull the brakes" before other parts of the systems become "infected". Furthermore, instead of *testing* for runtime errors, specialized approaches for ensuring initialization could be used [26, 22]. We think, it is advantageous to additionally have the option to replace runtime aborts with MEANINGLESS BEHAVIOR [8]. Consequently the only effects of uninitialized references are incorrect output values, possibly identifiable as such, as opposed to runtime aborts.

3. *Void Value classes introduce additional inheritance relationships.* For instance, if code reuse is desired between void- or object implementations of a descendent interface and its parent interface implementations then multiple inheritance is required (see *Multiple Inheritance* in section 2.8). We do not expect this to be a problem in practice, but lack the experience, though. The problem is alleviated by the fact that one of the two paths realizes interface inheritance only. With respect to language design we feel that some languages will have no problem at all (e.g., Sather [21]) and those who do will expose their flaws at other circumstances as well.

4. *Behavior for void data is rare and should be treated with* `nullcases` *in methods.* UFO [23] uses pattern matching for null values. This gives "null value" a name, thus forbidding transparent addition, deletion or replacement of null values. If desired, void values can be used in pattern matching style too. The syntactic sugar of the `nullcase` and `otherwise` clauses is simply replaceable with a `if`-statement, testing for void values with `is_void`. As a result, with only one concept instead of two, we may also achieve UFO's benefits of locality of function definition change, etc.

Void values represent a fresh view on the definition of void behavior. A VoidObject class reifies the void state of references into a value. It should be regarded as a prototype of its object. It depends on the nature of implemented type (class), a concrete purpose (role), and the particular operation (method) whether exception-, void propagation-, or default behavior is adequate.

Although we presented Void Value in pattern form it is not a pattern in its strict sense. At least we did not discover it by "pattern mining" existing software. Be that as it may, Void Value's interconnection with many other design patterns suggests its importance in object-oriented programming.

Object oriented languages, thus, should not inherit an outdated pointer model. They deserve a more appropriate treatment of uninitialized references. We firmly believe other languages will follow UFO's example and implement Void Value the one or the other way.

6 Acknowledgments

We like to express our thanks to John Sargeant who provided lots of information about UFO. He and Thilo Kielmann improved the paper by many helpful comments. We also thank Markku Sakkinnen and the anonymous reviewers for useful suggestions. Thanks to Dirk Koschorek for advice on Smalltalk.

References

[1] Andrew W. Appel. A critique of standard ML. *Journal of Functional Programming*, 4(3):391–429, October 1993.

[2] Ken Auer. Reusability through self-encapsulation. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 505–516. Addison-Wesley, 1994.

[3] Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the interaction of object-

oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Purdue University, February 1996.

[4] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* C.A.R. Hoare Series. Prentice Hall International, 1988.

[5] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programing*, 3(2):171–190, 1994.

[6] William R. Cook. Object-oriented programming versus abstract data types. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 151–178. Springer, May 1990.

[7] James O. Coplien. *Advanced C^{++}: Programming Styles and Idioms.* Addison-Wesley, 1992.

[8] Ward Cunningham. The checks pattern language of information integrity. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 145–156. Addison-Wesley, 1994.

[9] O.-J. Dahl and K. Nygaard. Simula Begin. Technical report, Norsk Regnesentral (Norwegian Computing Center), Oslo/N, 1967.

[10] G. Eckert and M. Kempe. Modeling with objects and values: Issues and perspectives, August 1994.

[11] Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. Technical Report UCSC-CRL-92-27, University of California, Santa Cruz, CA 95064, USA, June 1992.

[12] Ellemtel Telecommunications Systems Lab., Box 1505, 125 25 Älvsjö, Sweden. *Programming in C++: Rules and Recommendations*, April 1992.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Addison-Wesley, 1994.

[14] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[15] B. W. Kernighan and D. M. Ritchie. The C programming language, 1978.

[16] Thomas Kühne. Parameterization versus inheritance. In C. Mingins and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 15*, pages 235–245, Prentice Hall, 1995.

[17] Konstantin Läufer. A framework for higher-order functions in C++. In *Proc. Conf. Object-Oriented Technologies (COOTS)*, Monterey, CA, June 1995.

[18] B. J. MacLennan. Values and objects in programming languages. *SIGPLAN Notices*, 17(12):70–79, December 1982.

[19] Ole L. Madsen, Kristen Nygaard, and Birger Möller-Pedersen. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley and ACM Press, 1993.

[20] B. Meyer. *Eiffel the language.* Prentice Hall, 1992.

[21] Stephan Murer, Stephen Omohundro, and Clemens Szypersky. Engineering a programming language: The type and class system of Sather. In Jurg Gutknecht, editor, *Programming Languages and System Architectures*, pages 208–227. Springer Verlag, Lecture Notes in Computer Science 782, November 1993.

[22] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. July 1994.

[23] John Sargeant. Uniting Functional and Object-Oriented Programming. In *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 1–26. First JSSST International Symposium, November 1993.

[24] John Sargeant. Personal communication, April 1996.

[25] John Sargeant, Steve Hooton, and Chris Kirkham. Ufo: Language evolution and consequences of state. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 48–62, April 1995.

[26] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *Tose*, SE-12(1):157–171, January 1986.

[27] Bjarne Stroustrup. *The C^{++} Programming Language.* Addison-Wesley, 1986.

[28] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87*, pages 227–242, December 1987.

[29] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, Munich, Germany, January 1987.

[30] Peter Wegner. Dimensions of object-based language design. *OOPSLA '87*, 22(12):168–182, December 1987.