Abstract and Concrete Data Types vs Object Capabilities

J. Noble, A. Potanin, T. Murray, M. S. Miller

Abstract The distinctions between the two forms of procedural data abstraction — abstract data types and objects — are well known. An abstract data type provides an opaque type declaration, and an implementation that manipulates the modules of the abstract type, while an object uses procedural abstraction to hide an individual implementation. The object-capability model has been proposed to enable object-oriented programs to be written securely, and has been adopted by a number of practical languages including JavaScript, E, and Newspeak. This chapter addresses the questions: how can we implement abstract data types in an object-capability language? and, how can we incorporate primitive concrete data types into a primarily object-oriented system?

0.1 Introduction

> *Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary.*
>
> On Understanding Data Abstraction, Revisited,
> William Cook [3].

William Cook's "On Understanding Data Abstraction, Revisited" [3] emphasises a dichotomy between abstract data types, on one hand, and objects on the other.

Based on facilities originating in Alphard [28] and CLU [10], Cook defines an Abstract Data Type (ADT) as consisting of "a public name, a hidden representation, and operations to create, combine and observe values of the abstraction". The identification of a "public name" emphasises the fact that ADTs are not first class — certainly ADTs are not first class in most subsequent modular programming languages [2, 11, 26, 27]. An ADT has a hidden

representation: this representation is not of the (non-first-class, singleton) ADT itself, but of the *instances* of the ADT — the values of the abstraction that are manipulated by the ADT's operations. ADTs encapsulate their implementations using type abstraction: all the instances of an ADT are instances of the same (concrete) type, and the language's (static) type system ensures that the details of the instance's implementations cannot be accessed outside the lexical scope of the ADT's definition. The exemplary ADTs are generally stacks, queues, lists, but crucially numbers, strings, and machine level values can also be modelled as ADTs.

In contrast, objects are essentially encapsulated individual components that use procedural abstraction to hide their own internal implementations [3]. "Pure" objects do not involve a hidden type, or indeed any form of type abstraction: rather an object is a self-referential record of procedures. Whereas ADTs are typically supported in statically typed languages (because they depend on type abstraction), objects are as common in dynamically typed languages as in statically typed languages.

According to Cook, ADTs and objects have complimentary strengths and weaknesses. Objects are organised around data, so it is easy to add a different representation of an existing interface, and have that implementation interoperate with every other implementation of that interface. On the other hand, it is easier to add a new operation to an ADT, but hard to change an ADT's representation.

A crucial difference, however, is that ADTs offer support for what Cook calls "complex operations": that is, operations that involve more than one instance. Complex operations may be low-level, such as arithmetic operations on two machine integers, or higher level operations, such as calculating the union of two or more sets, or a database style join of several indexed tables. The distinguishing factor is that these operations are complex in that implementations must "inspect multiple representations" i.e. representations of other instances. [3]. Complex operations are easy to support with ADTs: all the instances of the ADT are encapsulated together in the ADT, and the code in the ADT has full access to all the instance's representations. In contrast, pure object-oriented programming does not support complex operations: each object is individually encapsulated and only one instance's representation can be accessed at any time.

This is particularly the case in object-capbility systems — pure object-oriented systems designed for security. Following Butler Lampson [9], Miller [13] defines the key design constraint of an object-capability system: *"A direct access right to an object gives a subject the permission to invoke the behaviour of that object"*. A programming language or system that grants one object privileged access to the representation or implementation of another object does not meet this criterion.

This, then, is the first question addressed by this chapter (and the workshop paper that preceeded it [21]): how can we implement Abstract Data Types, in pure object-oriented languages with object capabilities, while still

permitting complex operations across multiple instances? The key design question is: how do we model the boundary between the protected outside interface of an ADT and the shared inside implementation, and how do we manage programs that cross that boundary? [5, 12, 14, 15, 17, 18]

Then, we take our discussion further and propose a way to support primitive/concrete data types in a fully object-oriented language in the later part of this chapter.

0.2 Mint as an Abstract Data Type

Let us consider the well known Mint/Purse (or Bank/Account) example [12, 13, 15] used in the object capability world. A Mint (a Bank) can create new Purses (Accounts) with arbitrary balances, while a purse knows its balance, can accept deposits from another purse, and can also sprout new empty purses. Figure 0.1 shows the interface of the Mint/Purse example as an Abstract Data Type. Here, **makePurse** creates a new purse with a given balance, **deposit** transfers funds into a destination purse from a source purse, and **balance** returns a purse's balance. We also have an auxiliary operation **sprout** that makes a purse with a zero balance.

1. **makePurse**(Number) ⟶ Purse
2. **deposit**(Purse, Number, Purse) ⟶ (Purse, Purse, Boolean)
3. **balance**(Purse) ⟶ Number
4. **sprout** ⟶ Purse

Fig. 0.1 Mint/Purse Abstract Data Type

Figure 0.2 shows how one can define the type's behaviour axiomatically. These axioms reduce the ADT to a normal form where each purse is just created by **makePurse** with its balance. The nature of the Mint/Purse design as an ADT is shown by the **deposit** method which must update the balances of both the source and destination purses. As we will see, this is a key difficulty when implementing the Mint/Purse system in a pure object-capability language, because such a language cannot permit methods to access the representation of more than one object [3].

This description of Mint/Purse as an ADT obscures a couple of important issues. The first of these is that some of the operations on the ADT are more critical than others: notably that **makePurse** operation "inflates the currency" [15], that is it increases the total amount of money in the system — i.e. the overall sum of all the balances of all the ADT's purses. This operation must be protected: it should only be invoked by components that, by design, should have the authority to create more money. The other operations can

1. **deposit**(**makePurse**(D), A, **makePurse**(S)) = **true** $\leadsto$
   (**makePurse**($D + A$), **makePurse**($S - A$), true) $(A > 0) \wedge (S >= A)$
2. **deposit**(**makePurse**(D), A, **makePurse**(S)) = **true** $\leadsto$
   (**makePurse**(D), **makePurse**(S), false) $(A <= 0) \vee (S < A)$
3. **sprout** $\leadsto$ **makePurse**(0)
4. **balance**(**makePurse**(N)) $\leadsto N$

Fig. 0.2 Mint/Purse Axioms

be called by general clients of the ADT to transfer funds between purses but not affect the total money in the system.

This is why there is the auxiliary **sprout** operation which also creates a new purse, but which does not create additional funds — even though the semantics are the same as **makePurse**(0). In an object-oriented system, particularly an object-capability system, this restriction can be enforced by ensuring that the **makePurse** operation is offered as a method on a distinguished object, and access to that object is carefully protected. Languages based on ADTs typically use other mechanisms (such as Eiffel's restricted imports, or Modula-3's multiple interfaces, C++'s friends) to similar effect. These approaches are rather less flexible than using a distinguished object, as typically they couple the ADT implementation to concrete client modules — on the other hand, the extra object adds complexity to an object-oriented system's design.

The second issue is that, in an open system, particularly in an open distributed system, programs (and their programmers) cannot assume that all the code is "well behaved". This is certainly the case for a payments system: the point of a payment system is to act as trusted third party that allows one client to pay another client, even though the clients may not trust each other; one client may not exist at the time another client is written. In that sense, the notion of an "open system" as a system is, at best, ill-defined: where new components or objects can join and leave a system dynamically, questions such as what is the boundary of the system, which components comprise the system at any given time, or what are the future configurations of the system are very difficult to answer.

The question then is: how best can we implement such an ADT in a pure object-oriented language, particularly one adopting an object-capability security model, within an "open world": where an ADT may have to interoperate with components that are not known in advance, and that cannot be trusted by the ADT?

0.3 Implementing the Mint

We now try and answer that question, considering a number of different designs to support ADTs in object-oriented languages. We will present various different Grace implementations of the "Mint and Purse" system ubiquitous in object-capability research [15] to illustrate different implementation patterns.

0.3.1 Sealer/Unsealer

Our first implementation is based on the "classic" Mint in E, making use of sealer/unsealer brand pairs [15]. The sealer/unsealer design encodes the Mint/Purse ADT into two separate kinds of objects, Mints and Purses (see figure 0.3). The Mint capability (i.e. the Mint object) must be kept secure by the owner of the whole system, as it can create funds. On the other hand, Purses can be communicated around the system: handing out a reference to a Purse risks only the funds that are deposited into that purse, or that may be deposited in the future.

```
type Mint = interface {
  purse(amount : Number) -> Purse
}

type Purse = interface {
   balance -> Number
   deposit(amount : Number, src : Purse) -> Boolean
   sprout -> Purse
}
```

Fig. 0.3 Mints and Purses

This design is based on brand pairs [16, 15]. Brand pairs are returned from the makeBrandPair method, which returns a pair of a sealer object and an unsealer object. The sealer object's seal method places its argument into an opaque sealed box: the object can be retrieved from the box only by the corresponding unsealer's unseal method. The sealer/unsealer pairs can be thought of as modelling public key encryption, where the sealer is the public key and unsealer the private key (see Figure 0.4).

We can implement these two types using two nested Grace classes, (see figure 0.5, which follows the nesting in the E implementation [15]). The outer class implements the Mint type, with its purse method implemented by the nested class purse. Thanks to the class nesting, this implementation is quite compact. The Mint class itself is straightforward, holding a brandPair that will

```
type Sealing = interface {
  makeBrandPair -> interface {
    sealer -> Sealer
    unsealer -> Unsealer
  }
}

type Sealer = interface { seal(o : Object) -> Box }

type Unsealer = interface { unseal(b : Box) -> Object }

type Box = interface { }
```

Fig. 0.4 Brand Pairs

be used to maintain the integrity of the ADT, i.e. the whole Mint and Purse system. Anyone with access to a mint can create a new purse with new funds simply by requesting the purse class. (Grace doesn't need a **new** keyword to create instances of classes — just the class name is enough.) There is a sprout method at the end of the purse class so that clients with access to a purse (but not the mint) can create new empty purses (but not purses with arbitrary balances).

The work is all done inside the purses. Each purse has a *per-instance private* variable balance, and a deposit method that, given an amount and a valid source purse which belongs to this Mint/Purse system (i.e. which represents an instance of this ADT) adjusts the balance of both purse objects to perform the deposit. The catch is that the deposit method, here on the destination purse, must also modify the balance of the source purse. In a system that directly supported ADTs (such as many class-based OO languages [3]) this is simple: the balance fields would be *per-class private* and the deposit method could just access them directly (see figure 0.6).

This is not possible in an object-capability language because objects are encapsulated individually. The brokenDeposit method could only work if each purses' balance field was publicly readable and writeable: but in that case, any client could do anything it wanted to any purse it could access. Rather, in this design, the decr and getDecr and deposit methods, and the sealer/unsealer brandPair, collaborate to implement deposit without exposing their implementation beyond the boundary of the ADT. First, the decr method can decrease a purse's balance: this method is annotated as confidential, that is, *per-instance* private. Second, the public getDecr wraps that method in a lambda expression "{ amt -> decr(amt) }" and then uses the brandPair to seal that lambda expression, putting it into an opaque box that offers no interface to any other object. Although getDecr is public, meaning that it can be called by any object that has a reference to a purse, an attacker does not gain any advantage by calling that method, because the result is sealed inside the

```
class mint −> Mint is public {
  def myMint = self
  def brandPair = sealing.makeBrandPair

  class purse(amount : Number) −> Purse {
    var balance := amount

    method decr(amt : Number) −> Boolean is confidential {
      if ((amt < 0) || (amt > balance)) then {
          return false }
      balance := balance − amt
      return true }

    method getDecr
      {brandPair.sealer.seal { amt −> decr(amt) } }

    method deposit(amt : Number, src : Purse) −> Boolean {
      if (amt < 0) then { return false }
      var srcDecr
      try { srcDecr := brandPair.unsealer.unseal(src.getDecr) }
        catch { _ −> return false }
      if (srcDecr.apply( amt )) then {
         balance := balance + amt
         return true }
      return false }

    method sprout { purse(0) }
  }
}
```

Fig. 0.5 Sealer/Unsealer based Mint

```
method brokenDeposit(amt : Number, src : Purse) −> Boolean
  { if ((amount >= 0) && (src.balance >= amount))
      then {
          src.balance := src.balance − amount
          balance := balance + amount
          return true
       } else {return false}
   }
```

Fig. 0.6 ADT Deposit Method (per-class private)

opaque box. Finally, the deposit method will use the matching unsealer from the *same* brand pair to unseal the box, and can then invoke the lambda expression to decrement the source purse. This remains secure because each instance of the mint class will have their own brand pair, and so can only unseal their own purses' boxes — the unseal method will throw an exception if it is passed a box that was sealed by a different brand pair.

0.3.2 Generalising the Sealer/Unsealer

The previous section's Mint/Purse design works well enough for, well, purses and mints, but sealing a single lambda expression only works when there is just one operation that needs to access two (or more) instances in the ADT. We can generalise the sealer-unsealer design by sealing a more capable object to represent the instances of the ADT.

In this design, we have an ExternalPurse that offers no public methods, and an InternalPurse that stores the ADT instance data, in this case the purse's balance (see figure 0.7).

```
type ExternalPurse = interface { }

type InternalPurse = interface {
  balance -> Number
  balance:= ( n : Number )
}
```

Fig. 0.7 External and Internal Purse Interfaces

Because the external purses are opaque, we need a different object to provide the ADT operations — effectively to reify the ADT as a whole. Rather than making requests to the ADT instance objects directly ("dst.deposit(amt, src)") we will pass the ADT instances to the object reifying the ADT, e.g.:

```
mybank.deposit(dstPurse, amt, srcPurse)
```

In fact, to deal with the difference in privilege between creating new purses containing new money, versus manipulating existing purses with existing money, this design needs two objects: an Issuer that presents the main interface of the ADT, and which can be publicly available, and a Mint that permits inflating the currency, and consequently must be kept protected (see figure 0.8).

These interfaces can be implemented with a generalisation of the basic mint design (see figure 0.9). Each mint again has a brand pair, and auxiliary (confidential) methods to seal and unseal an InternalPurse within an opaque

```
type Issuer = interface {
  balance(of : ExternalPurse) -> Number
  deposit(to : ExternalPurse,
          amount : Number,
          from : ExternalPurse ) -> Boolean
  sprout -> ExternalPurse
}

type Mint = interface {
  purse(amount : Number) -> ExternalPurse
  issuer -> Issuer
}
```

Fig. 0.8 Splitting the Issuer from the Mint

sealed box: these boxes will be used as the ExternalPurse objects. A new internal purse is just a simple object with a public balance field; an external purse is just an internal purse sealed into a box with the brand pair. Implementing the ADT operations is quite straightforward: any arguments representing accessible proxies for ADT instances (external purses) are unsealed, yielding the internal representations (internal purses) and then the operations implemented directly on the internal representations. An invariant of this system, of course, is that those internal representation objects are confined with the object reifying the whole ADT, and so can never be accessed outside it.

0.3.3 Hash table

A similar design can employ a hash table, rather than sealer/unsealer brand-pairs to map from external to internal representations (see figure 0.10). This has the advantage that the external versions of the ADT instances have to be the sealed boxes themselves, and can offer interfaces so that they can be used directly as the public interface of the ADT. This means we do not need to split the ADT object into two objects to distinguish between a public interface ("Issuer") and a private interface ("Mint").

The implementation of this design (in figure 0.11) is more straightforward than the sealer/unsealer design (figure 0.9). The mint class contains a map (here instances) from external to internal purses; we also have a couple of helper methods to check if an (external) purse is valid for this mint, and to get the internal purse corresponding to an external purse.

To actually make a new purse, the mint makes a pair of objects (one internal and one external purse) stores them into the instances map, and returns the external purse. As in the sealer/unsealer based design, here the Mint object reifying the ADT must still offer methods implementing the ADT operations.

```
class mint -> Mint {

  def myBrandPair = sealing.makeBrandPair

  method seal(protectedRep : InternalPurse) -> ExternalPurse
    is confidential { myBrandPair.sealer.seal(protectedRep) }

  method unseal(sealedBox : ExternalPurse) -> InternalPurse
    is confidential { myBrandPair.unsealer.unseal(sealedBox) }

  method purse(amount : Number) -> ExternalPurse {
     seal( object { var balance is public := amount } ) }

  def issuer is public = object {

    method sprout -> ExternalPurse { purse(0) }

    method balance(of : ExternalPurse) -> Number {
      return unseal(of).balance}

    method deposit(to : ExternalPurse,
                   amount : Number,
                   from : ExternalPurse) -> Boolean {
      var internalTo
      var internalFrom
      try {
          internalTo := unseal(to)       // throws if fails
          internalFrom := unseal(from)   // throws if fails
      } catch { _ -> return false }

      if ((amount >= 0) && (internalFrom.balance >= amount))
        then {
          internalFrom.balance := internalFrom.balance - amount
          internalTo.balance := internalTo.balance + amount
          return true
        } else {return false}
    }
  }
}
```

Fig. 0.9 Generalised Sealer/Unsealer based Mint

These operations are by the external purses to implement the ADT: they cannot generally be used by the ADT's clients as the reified ADT object (the mint) can inflate the currency by creating non-empty purses, so that capability must be kept confined.

The internal purse implementation is also straightforward. We could have used just objects holding a balance field, or even stored the balance directly in the map, but here we add some additional behaviour into the representation objects (see figure 0.12).

```
type ExternalPurse = interface {
  balance -> Number
  deposit(amount : Number, src : ExternalPurse) -> Boolean
  sprout -> ExternalPurse
}

type Mint = interface {
 purse(amount : Number) -> ExternalPurse
 deposit(to : ExternalPurse,
         amount : Number,
         from : ExternalPurse) -> Boolean
 balance(of : ExternalPurse) -> Number
 sprout -> ExternalPurse
}

type InternalPurse = interface {
  balance -> Number
  balance:= (Number) -> Done
  deposit(amount : Number, src : ExternalPurse) -> Boolean
}
```

Fig. 0.10 Interfaces for Hash Table based Mint

Finally, the externalPurse class implements the ADT instances — the public purses — as "curried object" proxies that delegate their behaviour back to the mint object that represents the whole ADT. Here we give the external purses their mint as a parameter: this would work equally well by nesting the external purse class within the mint (see figure 0.13).

0.4 Owners as Readers

In earlier work we have argued that an owners-as-readers discipline can provide an alternative formulation of ADTs [22]. Owners-as-readers depends on object ownership rather than type abstraction to encapsulate the implementations of the ADT instance [23]. In this model, all the instances are owned by an additional object that reifies the whole ADT, and the ownership type system ensures that they can only be manipulated within the scope of that object. Where owners-as-readers differs from other ownership disciplines is that other objects outside the ADT can hold references to the ADT instance objects, but those outside references appear opaque, and any requests on those objects from outside raise errors.

A range of ownership systems can be characterised as providing an owners-as-accessors discipline [20, 7, 25, 8, 4]: we have discussed these in more detail elsewhere [22]. Owners-as-readers systems clearly **do not** meet the key requirement of an object-capability system, precisely because owned objects

```
class mint -> Mint {

  def instances = collections.map[[ExternalPurse,InternalPurse]]

  method valid(prs : ExternalPurse) -> Boolean
    { instances.contains(prs) }

  method internal(prs : ExternalPurse) -> InternalPurse
    { instances.get(prs) }

  method purse(amount : Number) -> ExternalPurse {
    def ext = externalPurse(self)
    def int = internalPurse(amount)
    instances.put(ext, int)
    return ext
  }

  method deposit(to : ExternalPurse,
                 amount : Number,
                 from : ExternalPurse) -> Boolean {
    if ((valid(to)) && (valid(from))) then {
        return internal(to).deposit(amount, internal(from))}
    return false
  }
  method balance(prs : ExternalPurse) -> Number
    { internal(prs).balance }

  method sprout -> ExternalPurse {purse(0)}
}
```

Fig. 0.11 Hash table based Mint

```
class internalPurse(amount : Number) -> InternalPurse {
  var balance is public := amount
  method deposit(amount : Number, src : InternalPurse)
          -> Boolean
    { if ((amount >= 0) && (src.balance >= amount)) then {
        src.balance := src.balance - amount
        balance := balance + amount
        return true }
      return false
    }
}
```

Fig. 0.12 Internal Purse for Hash Table based Mint

```
class externalPurse(mint' : Mint) -> ExternalPurse {
  def mint = mint'
  method balance {mint.balance(self)}
  method sprout -> ExternalPurse { mint.sprout }
  method deposit(amount : Number, src : ExternalPurse)
     -> Boolean { return mint.deposit(self, amt, src)  }
}
```

Fig. 0.13 External Purse for Hash Table based Mint

are opaque outside their owners — although they would meet the following modified criterion: *"A direct access right to an object gives a subject the permission to invoke the behaviour of that object* ***from inside that object's owner****"*.

The resulting design is most similar to the sealer/unsealer version, because outside the mint ADT the internal purses are opaque. This means clients need to interact with the object reifying the ADT, and so we must split that object to separate the privileged capability (again, Mint) from the general capabilities to use the rest of the ADT operations (again, Issuer). On the other hand, we do not need an explicit split between internal and external purses (see figure 0.14).

```
type Mint = interface {
  purse(amount : Number) -> Purse
  issuer -> Issuer
}

type Issuer = interface {
  balance(of : Purse) -> Number
  deposit(to : Purse,
          amount : Number,
          from : Purse) -> Boolean
  sprout -> Purse
}

type Purse = interface {
   balance -> Number
   balance:= ( n : Number )
}
```

Fig. 0.14 Interfaces for Owners-as-Readers based Mint

Implementing this really should be straightforward by now. We make a purse class that only holds a balance: crucially that class is annotated is owned. The main ADT operations are defined inside the issuer object — the methods implementing these operations can just access the owned purse objects

directly because they are within the mint: the owners-as-readers constraint ensures that the purses cannot be accessed from outside the ADT's boundary (see figure 0.15).

```
class mint -> Mint {

  class purse(amount : Number) -> Purse is owned {
    var balance is public := amount
  }

  def issuer is public = object {

    method sprout -> Purse { purse(0) }

    method deposit(to : Purse is owned, amount : Number,
          from : Purse is owned) -> Boolean {
      if (
          (amount >= 0) && (from.balance >= amount))
        then {
          from.balance := from.balance - amount
          to.balance := to.balance + amount
          return true
        } else {return false}
    }

    method balance(of : Purse is owned) -> Number {
      return of.balance}
  }
}
```

Fig. 0.15 Owners-as-Readers based Mint

From the perspective of the owners-as-readers design, we can consider that both the sealer/unsealer or the map-bgased designs embody an ad-hoc form of ownership: in both cases there are internal capabilities — the internal purses — that must be confined within the ADT implementation, and the ownership — the control of the ADT's boundary — is embodied in the sealer/unsealer's brand-pair, or in the map from external to internal purses: here that ownership is supported directly in the programming language.

We can also speculate on whether there is an obvious way to provide a public ADT interface via the purses, rather than again requiring operations to be addressed to the object reifying the ADT (here the Issuer and the Mint. The answer is both yes and no: yes, because a language could e.g. distinguish between ADT-public and ADT-private operations on those instance objects, and no, because that takes us right back to ADT oriented languages with per-class access restrictions, that is, right away from the object-capability model.

0.5 Primitive Data Types

Object-oriented languages must also integrate another type of data item into their object models. Machine-level primitive types such as integers, floating point numbers, Booleans, may be provided directly by the underlying CPU; other types such as Strings or Symbols that be implemented directly by a virtual machine. Different languages tackle this problem in different ways:

- C++ & Java objects and primitives are in different universes. The langauges' static type systems ensure primmitives and objects cannot be mistaken for each other — although different types of conversions can be applied to "box" a primitive into an object, and "unbox" it as necessary.
- Smalltalk objects have a primitive part and an "object-oriented" part [6]. The primitive part stores e.g. the value of a number as a primitive double, while the object-oriented part holds the objects' variables and methods Objects' method's bodies can designate virtual machine primitive operations that should be invoked using data stored in the objects' primitive parts
- Self objects also have a primitive part and an object-oriented part (the same data model as Smalltalk) [24]. In Self, primitive code is invoked by syntactically differentiated messages (requests, method calls) rather than by designated method bodies.
- Newspeak objects again have primitive and object-oriented parts, but primitive code is invoked via a VM proxy object, rather than distinguished methods or messages [1].

0.5.1 External Primitives

Figure 0.16 shows the kind of code Newspeak uses to invoke operations on primitive objects (albeit expressed using Grace syntax). A request on integers ("1 + 2") resolves to one of these method definitions, resulting in a request such as ("vm.addInteger(1,2)") to the vm object that reifies the virtual machine.

```
class integer { // construction is magic
  method +(other : Number) { vm.addInteger(self, other) }
  method −(other : Number) { vm.subtractInteger(self, other) }
  method ∗(other : Number) { vm.multiplyInteger(self, other) }
  method /(other : Number) { vm.divideInteger(self, other) }
  method prefix−  { vm.negateInteger(self) }
}
```

Fig. 0.16 Newspeak Primitive Invocations via VM Proxy

The key here is to compare the code for integers in figure 0.16 with the code for the external purses in the mint implemented with a hashtable in figure 0.13. Both of these figures exhibit the same pattern: a method is called on an "instance object" — generally a method that takes more than one instance to implement a complex operation and the reciever and arguments to that message are passed in to a third party (the mint or vm proxy) which effectively retrieves the data and executes the operation.

This model certainly makes the route to invoke VM primitives clear — and, in accordance with Newspeak's pure object-object design, does so without any language support for primtive methods or messages. This model is still asymmetric, however: although the behaviour is moved out of the objects, the primitive data still remains in their primitive parts. This is most obvious when asking how instances of the integer class in figure 0.16 are created — the answer is: they have to be created by the VM or language implementation itself, either from literals in the source code or operations on other integers or other primitive types.

0.5.2 Object-as-Readers Primitives

Perhaps the cleanest model is to follow William Cook's distinction between objects and ADTs to its logical conclusion [3]. We can treat all low-level primitive types as (interlinked) sets of ADTs provided by the VMs — rather than objects. Individual instances of these ADTs offer no methods in their own rights, not even equality. To other code, they are opaque 'magic cookies' that can be stored in object's fields, stored in variables, passed as arguments, but that offer no behaviour themselves: the only way to operate on the instances is to pass them into the vm or another object reifying the ADT, as in the code in figure 0.17.

Behaviour for these ADTs is again provided by a vm object — or, perhaps, a set of singleton objects one per ADT. The methods e.g. on the vmIntegerADT class take and return those magic cookies, which cannot be manipulated any other way. All the language-level behaviour for primitives is then written as normal code that manipulates the magic cookies. In figure 0.17, the integer class can be written in completely normal code, again as in figure 0.17.

This design results in a clear interface to the VM ADTs, a clearer data model — as objects are either all primitive, or all language-level objects, but with nothing in between. There is also a clear interface to the VM support for the primitive types — the interface supported by objects like vmIntegerADT module object. Other information or behaviour can be added in to the wrapper classes — perhaps to add provenance infomation, or taint tracking — without affecting the interface to the underlying ADT.

This model need not necessarily be slower or more memory intensive than one based on primitive parts and inheritance. Raw CPUs, and high-

```
class integer(cookie' : VmInteger) -> Number  { // construction is normal
  def cookie is public = cookie'

  method +(other : Number) -> Number {
    integer(vmIntegerADT.addInteger(cookie, other.cookie))}
  method -(other : Number) -> Number {
    integer(vmIntegerADT.subtractInteger(cookie, other.cookie))}
  method *(other : Number) -> Number {
    integer(vmIntegerADT.multiplyInteger(cookie, other.cookie))}
  method /(other : Number) -> Number {
    integer(vmIntegerADT.divideInteger(cookie, other.cookie))}
  method prefix- -> Number {
    integer(vmIntegerADT.negateInteger(cookie)) }
}
```

Fig. 0.17 Objects-as-Readers Primitive Invocations

performance VMs both end up implementing primitive types by allocating implementation fields in the representation of the object. This is as true for a machine code implementation as it is for an implementaiton in a higher level VM or even a translation to a dynamic langauge such as Javascript.

0.5.3 Primitives for All Objects

The remaining issues is that in many object-oriented languages, all objects are to some extent primitive objects. Just as class Object in Smalltalk or Java provides language level methods that apply to all objects, so it also relies on a small number of primitive methods that also apply to all objects. In these object models, we can consider that — conceptually — each object has a primitive data part that can be thought of has holding its identity (at least), perhaps also its class, lock, type, etc. Again, as with other kinds of primitives, primitive behaviours can be invoked by an appropriate mechanism to access that primitive data part. Because every object has that primitive data, those primitives apply to all objects. Figure **??** shows how this could be implemented in the explicit-ADT style.

This design, though, gives object an identity-based hash, and identity-based equality whether they want it or not — e.g. "functional" objects (bascialy records) may not want this equality; a proxy may want both identity and hash code to be delegated to the object they are proxying.

Time, then, to go to one last remove. Figure 0.19 shows an abstractEquality trait that defines a suite of "left-handed" equality operators in terms of two abstract operations, isMe and hash [19]. Objects can either provide those operations directly, or if they do want identity semantics, they can additionally

```
class graceObject −> Object  {

  def cookie is public = vmObjectADT.newUniqueIdentity

  method ==(other : Object) −> Boolean
    { boolean( vmObjectADT.eq(cookie, other.cookie) ) }
  method hash −> Number
    { integer( vmObjectADT.hash(cookie) ) }
  method synchronized (block : Block[Done]) −> Done
    { with (vmObjectADT.lock(cookie).acquire) do (block) }
}
```

Fig. 0.18 Identity ADT at the Top of the Hierarchy.

inherit from the identityEquality trait shown in figure 0.20 which provides a ready-made identity-based implementation.

```
trait abstractEquality {
  method ==(other) { isMe(other) }
  method !=(other) {! (self == other) }
  method hash { identityHash }

  method isMe(other) is required { }
  method identityHash is required { }
}
```

Fig. 0.19 Abstract Equality Trait

```
trait identityEquality {
  use equality

  def cookie is public = vmObjectADT.newUniqueIdentity

  method ==(other : Object) −> Boolean
    { boolean( vmObjectADT.eq(cookie, other.cookie) ) }
  method hash −> Number
    { integer( vmObjectADT.hash(cookie) ) }
}
```

Fig. 0.20

0.6 Conclusion

In this chapter we have considered issues in designing and implementing abstract data types in purely object-oriented systems, and in object-capability systems in particular. We show how primitive instances can be treated as abstract data types in object-oriented systems, and extend that treatment to encompass even explicit object identity.

The first design we considered used sealer/unsealer brand pairs to encapsulate the ADT's shared state, but kept that state within the individual purse objects. The code that implemented the system is also primarily in the purses — a mint object primarily exists to provide a separate capability to inflate the currency.

Our second design also encapsulates the ADT implementation using sealer/unsealer pairs, but generalises the design, to split each logical purse into two different capabilities, that is, into two separate objects, one of which is accessible from outside the ADT, and the second accessible only from inside. In this design, the external purse objects are opaque, so in effect we also split the mint object representing the whole ADT into an unprivileged Issuer, and the privileged Mint.

Our third design retains the split between internal and external purses, but uses a hash table rather than sealer/unsealer brand pair to provide the encapsulation boundary. This has the advantage that the external purses do not have to be the sealed boxes, and so we can return to a more "object-oriented" style API, where clients interact with the purse objects directly, rather than via the issuer object; this means we no longer need to split the mint capability. The catch, of course, is that this is probably the most complex design that we consider in this chapter.

Our last design revisits our owners-as-readers encapsulation model, which tries to build in minimal support for ADTs in a dynamic, object-oriented setting. This is the smallest implementation, because owners-as-readers renders the purses opaque outside the ADT, and so the purse objects no longer need to be split in any way. On the other hand, because the purses are opaque to all the clients of the system, we again need an issuer offering the classic ADT-style interface.

Finallly, we show how primitive objects, and primitive operations across all objects can be encompassed within these designs.

We note that aliasing issues arise pervasively in these object-capability implementations. Wherever we have to split objects to divide public and private capabilities (i.e. those capabilities on the inside and outside of the ADT's boundaries) then there will be an implicit aliasing relationship between those objects. These designs also involve confinement or ownership relationships, implicitly or explicitly, in that the internal object-capabilities must not be accessible from outsides.

As with much object-capability research, we have once again tackled the mint/purse system. The abstract data type perspective can explain why this

example is so ubiquitous: because the mint/purse system is about as simple an abstract data type as you can get: the data held in each ADT instance is just a single natural number. We hypothesise that many of the examples used in object-capability systems may be better modelled as ADTs, rather than objects, and that much of the difficulty in implementing those examples in object-capability systems stems directly from this incompatibility in underlying model. We hope the object-capability designs that we have presented here, however, should be able to cope with a range of more complex abstract data types.

Acknowledgements

We thank the anonymous reviewers for their comments. This work was supported in part by a James Cook Fellowship and by the Royal Society of New Zealand Marsden Fund.

References

1. Bracha, G.: Newspeak programming language draft specification version 0.05. Tech. rep., Ministry of Truth (2009)
2. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 language definition. SIGPLAN Not. **27**(8), 15–42 (Aug 1992). https://doi.org/10.1145/142137.142141, http://doi.acm.org/10.1145/142137.142141
3. Cook, W.R.: On understanding data abstraction, revisited. In: OOPSLA Proceedings. pp. 557–572 (2009)
4. Dimoulas, C., Moore, S., Askarov, A., Chong, S.: Declarative policies for capability control. In: Proceedings of the 27th IEEE Computer Security Foundations Symposium (Jun 2014)
5. Drossopoulou, S., Noble, J., Miller, M.S.: Swapsies on the Internet. In: PLAS (2015)
6. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983)
7. Gordon, D., Noble, J.: Dynamic ownership in a dynamic language. In: DLS Proceedings. pp. 9–16 (2007)
8. Gruber, O., Boyer, F.: Ownership-based isolation for concurrent actors on multi-core machines. In: ECOOP. pp. 281–301 (2013)
9. Lampson, B.W.: Protection. Operating Systems Review **8**(1), 18–24 (Jan 1974)
10. Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.: Abstraction mechanisms in CLU. Comm. ACM **20**(8), 564–576 (Aug 1977)
11. MacQueen, D.: Modules for Standard ML. In: LISP and Functional Programming. pp. 198–207. ACM (1984). https://doi.org/10.1145/800055.802036, http://doi.acm.org/10.1145/800055.802036
12. Miller, M.S., Cutsem, T.V., Tulloh, B.: Distributed electronic rights in JavaScript. In: ESOP (2013)
13. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D. thesis, Baltimore, Maryland (2006)
14. Miller, M.S.: Secure Distributed Programming with Object-capabilities in JavaScript (Oct 2011), talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com

15. Miller, M.S., Morningstar, C., Frantz, B.: Capability-based financial instruments: From object to capabilities. In: Financial Cryptography. Springer (2000)
16. Morris Jr., J.H.: Protection in programming languages. CACM **16**(1) (1973)
17. Noble, J.: Iterators and encapsulation. In: TOOLS Europe (2000)
18. Noble, J., Biddle, R., Tempero, E., Potanin, A., Clarke, D.: Towards a Model of Encapsulation. In: Clarke, D. (ed.) IWACO Proceedings. No. 030 in UU-CS-2003, Utrecht University (Jul 2003)
19. Noble, J., Black, A.P., Bruce, K.B., Homer, M., Miller, M.S.: The left hand of equals. In: Onward! ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 224–237 (2016). https://doi.org/10.1145/2986012.2986031
20. Noble, J., Clarke, D., Potter, J.: Object ownership for dynamic alias protection. In: TOOLS Pacific 32 (1999)
21. Noble, J., Drossopoulou, S., Miller, M.S., Murray, T., Potanin, A.: Abstract data types in object-capability systems. In: IWACO Proceedings (2016)
22. Noble, J., Potanin, A.: On owners-as-accessors. In: IWACO Proceedings (2014)
23. Potanin, A., Damitio, M., Noble, J.: Are your incoming aliases really necessary? counting the cost of object ownership. In: International Conference on Software Engineering (ICSE) (2013)
24. Ungar, D., Smith, R.B.: SELF: the Power of Simplicity. Lisp and Symbolic Computation **4**(3) (Jun 1991)
25. Wernli, E., Maerki, P., Nierstrasz, O.: Ownership, filters and crossing handlers. In: Dynamic Language Symposium (DLS) (2012)
26. Whitaker, W.A.: Ada - the project: The DoD high order language working group. In: HOPL Preprints. pp. 299–331 (1993)
27. Wirth, N.: Programming in Modula-2. Springer Verlag (1985), isbn 0-387-15078-1
28. Wulf, W.A., London, R.L., Shaw, M.: An introduction to the construction and verification of Alphard programs. IEEE Trans. Softw. Eng. **SE-2**(4), 253–265 (1976)