Creational Invariants

Cees Pierik[1], Dave Clarke[2], and Frank S. de Boer[1,2,3]

[1] Institute of Information and Computing Sciences, Utrecht University, The Netherlands
[2] CWI, Amsterdam, The Netherlands
[3] LIACS, Leiden University, The Netherlands
cees@cs.uu.nl {dave,frb}@cwi.nl

Abstract. A characteristic property of the invariants underlying creational design patterns is that they quantify over all the objects (of a certain class). We examine such invariants, determining the constraints that they place on the environment. In addition, we analyze the degree to which (some) creational patterns contribute to the satisfaction of such constraints.

1 Introduction

Object invariants simplify the verification of object-oriented programs by enabling programmers to specify properties which hold in all stable phases of an object's lifetime. With invariants, such properties need only be expressed once. Object invariants may express constraints on the values stored in the fields of an object *and* those reachable from that object — both expressed from the perspective of a particular object.

Further constraints can be expressed by allowing invariants to quantify over all existing objects of a particular class. This kind of invariant generally states things about the relationship between objects of a class and each other, *e.g.,* that a certain property uniquely characterizes each object; their role within a data structure, *e.g.,* that a manager object holds references to all objects of a certain type; or about the number of instances created, *e.g.,* the unique instance created within the singleton pattern. Invariants involving quantification are the focus this paper.

Such invariants often reveal important design decisions made by software developers; they also appear within creational design patterns (flyweight, manager, singleton, etc) [1], which often exist to (informally) enforce such invariants. These design patterns often achieve their effect by providing (or presupposing) control over object creation for the classes of interest.

To establish object invariants of the form we are interested in, namely those quantifying over all instances of a class, certain constraints must be placed on the environment using the class in question. In this paper, we demonstrate how to compute these constraints. Our technique combines the recent work by Barnett and Naumann [2, 3] on object invariants over shared objects with the weakest precondition calculus for object creation developed by the first and third authors [4]. We illustrate our technique by applying it to creational design patterns.

2 Creational Patterns

A well-known creational pattern is the Singleton pattern [1]. An example implementation of this pattern in a Java-like language is listed in Fig. 1. The aim of this pattern is to ensure that a class has a unique instance and to provide a global point of access to it. In our example, this boils down to the following invariant:

$$(\forall o : \mathit{Singleton} \bullet o = \mathit{Singleton.unique}) \ . \qquad (I_{\text{Singleton}})$$

(Throughout this paper quantification ranges over allocated non-null objects). Note that the invariant reflects both aims of the pattern.

The Flyweight pattern [1] can be used to reduce the storage required to represent value objects by maintaining an object pool (of flyweights) in a central Flyweight Factory. The flyweights can

```
class Singleton {
   private static Singleton unique ;
   private Singleton() {}
   public static Singleton getInstance() {
      if (unique = null)
         unique := new Singleton();
      return unique ;
   }
}
```

Fig. 1. The Singleton pattern

be shared by several clients, and moreover, generally satisfy the constraint that each distinct value is represented by only one flyweight object.

To illustrate this pattern, we apply it to integer flyweights (more useful examples include fonts and glyphs). The code for this pattern is listed in Fig. 2. Integers (in the range 0–999) are represented by class *MyInt*. The class *IntFactory* globally creates and stores the *MyInt* objects.

```
class IntFactory {                        class MyInt {
   private MyInt[] ints ;                    private int x ;
   public IntFactory() {                     public MyInt(int x) {
      ints := new MyInt[1000] ;                 this.x = x ;
   }                                         }
   public MyInt getInt(int x) {              public boolean equals(MyInt i) {
      if (ints[x] = null)                       return i = this ;
         ints[x] := new MyInt(x) ;           }
      return ints[x];                     }
   }
}
```

Fig. 2. The Flyweight pattern

Based on the flyweight pattern's presupposition that there is at most one instance corresponding to each value object, the programmer can implement the *equals* method of *MyInt* by simply comparing object identities. The invariant required for this to work is:

$$(\forall i : \mathit{MyInt} \bullet i.x = \mathsf{this}.x \rightarrow i = \mathsf{this}) \ . \qquad (I_{\mathrm{MyInt}})$$

This invariant is restrictive and difficult to establish. The class *IntFactory* cooperates to simplify reasoning about the invariant (I_{MyInt}) by preserving the invariant:

$$(\forall i : \mathit{MyInt} \bullet \mathsf{this}.\mathit{ints}[i.x] = i) \wedge \mathsf{this}.\mathit{ints}.\mathsf{length} = 1000 \ . \qquad (I_{\mathrm{IntFactory}})$$

Both invariants depend on the cooperation of the environment of their defining class because they can be invalidated by the creation of a *MyInt* object in any class that has access to its constructor. Such invariants can only be established if the class can rely on certain assumptions regarding the behavior of its environment. We explain how these assumptions can be calculated in Sect. 4.

Other design patterns such as the Abstract Factory [1], the Factory [1], the Prototype [1], and the Manager [5] all impose (and demand) invariants on object creation. For this paper, however, we restrict our attention to the Singleton and Flyweight patterns.

3 Background and Related Work

A number of groups have been addressing invariants in object-oriented program verification. We work (implicitly) within the 'Boogie' methodology [6], along the lines of Barnett and Naumann's approach for reasoning about objects which share state [2, 3]. For now, we ignore inheritance and subtyping.

The object invariant for class C (denoted by Inv_C) is a formula over program expressions [4], often defined in terms of this (its receiver), which describes each allocated object of class C. Object invariants are usually expected to hold in all 'visible' states. This implies that the invariants must hold every time control leaves a method of a class [7]. While quite stringent, because invariants must hold even when control temporarily leaves an object in an outgoing call, being more flexible requires that scenarios in which such a call re-enters the object and wrongfully assumes that the invariant holds must be prevented.

More invariants can be allowed by relying on a state space which is extended to also express when each invariant is required to hold [6]. In this setting, every object is given an auxiliary specification field *inv* to record this information. This field signals in the specification of a method if the invariant of the object holds. The following system invariant (for each class C) relates this field to the object's invariant:

$$(\forall o : C \bullet o.inv \rightarrow Inv_C[o/\mathsf{this}])$$

The substitution $[o/\mathsf{this}]$ replaces all occurrences of this by o in the predicate Inv_C.

The *inv* field is initially set to false during object allocation, but it can be switched on and off by two special commands pack and unpack, which follow for an expression o of (static) type C:

$$\begin{aligned} \mathsf{pack}\ o &\equiv \mathsf{assert}\ \neg(o = \mathsf{null}) \wedge \neg o.inv\ \wedge\ (Inv_C[o/\mathsf{this}])\ ;\ o.inv := \mathsf{true}; \\ \mathsf{unpack}\ o &\equiv \mathsf{assert}\ \neg(o = \mathsf{null}) \wedge o.inv\ ;\ o.inv := \mathsf{false}\ ; \end{aligned}$$

These commands enable a discipline whereby each object is unpacked before its fields may be modified. This can be achieved by placing the additional precondition $\neg e.inv$ on all field assignments of the form $e.x := e'$.

Basic object invariants refer only to private fields and the keyword this. Leino and Müller [8] used ownership [9] in order to extend the range of invariants to objects beyond of the original object, to owned objects and objects with the same owner. Yet another, orthogonal, extension enables an invariant to be divided into a number of slices, one for each superclass, and permits at various phases of the verification only part of the invariant to hold [6].

Barnett and Naumann [2] extended the set of admissible invariants by allowing object invariants to mention fields of unowned objects. The classes of these objects are called *friend* classes, which reflects the fact that they should cooperate in order to maintain the invariant. The willingness to cooperate is checked by additional constraints on field assignment in the friend's class. Such constraints can be determined using a weakest precondition calculation, and they ultimately constrain the use of the shared state. Suppose that the invariant of class F mentions field x of class C. Now consider an assignment $e.x := e'$, where the static type of e is C. This assignment potentially disturbs the invariant of objects of class F. The following additional precondition on the assignment checks that either the friend is unpacked or its invariant is not invalidated by the assignment, where the operator $[e'/e.x]$ is the weakest precondition operation for the assignment $e.x := e'$ (defined elsewhere [4]):

$$(\forall o : F \bullet \neg o.inv \vee (Inv_F[o/\mathsf{this}][e'/e.x]))$$

The dependencies can be managed by means of explicit pivot, friend and dependency information [2]. We ignore these details for now as we will focus on object creation.

Barnett and Naumann use a semantical characterization of the admissible invariants that rules out every invariant that can be invalidated by object creation [3]. This condition prohibits invariants that quantify over objects that are not reachable from the point of view of the invariant's receiver. We will remove this restriction in order to analyze the invariants of the discussed creational patterns.

4 Invariants and Quantification

Barnett and Naumann's friendship methodology extends the set of admissible invariants, but a mere glance at the invariants of the creational patterns shows that more is needed. The desired extension is unrestricted quantification over all objects of a class. (We also require statements about static variables, but this can easily be incorporated.)

Note that the invariant of the Singleton pattern does not refer to objects of type Singleton via the keyword this. It can therefore also be associated with the class itself, rather than with an instance. We will refer to such invariants as *class* invariants. An advantage of class invariants is that they may already hold prior to the first creation of an instance of the class. In fact, the invariant of class Singleton already holds in the initial (empty) heap of every program.

The technology for object invariants can easily be adapted to class invariants. We will assume a static auxiliary variable $C.inv$ in each class C that will signal whether the class invariant holds. Initially, this class invariant bit is set to false. The static initializer of the class can pack the class after completing initialization, thus setting the bit to true. The class invariant of class C itself is written $ClassInv_C$.

We now move to our main problem: quantification. Invariants that quantify over the objects of some class are vulnerable to object creation. Whenever an invariant says that a property P holds for all objects of class C, then the invariant will be broken if P does not hold for an arbitrary freshly created object from class C.

We will assume that the execution of a statement $lhs := \mathsf{new}\ C(e_1, \ldots, e_n)$ proceeds in four steps as in Java. First, a new object of class C is allocated. We will imagine that is temporarily assigned to some fresh local variable u. Subsequently, the arguments $e_1, \ldots, e_n$ of the constructor method are evaluated. In the third step the constructor method is called on the new object u. Finally the new object is assigned to the left-hand side lhs.

We will focus here on the first step. The second step has no effect on any invariant because we assume that the arguments of the constructor method are side-effect free expressions. The call to the constructor method does not differ significantly from other method calls. The assignment to the left-hand side can be handled along the lines of the friendship methodology if the left-hand side is a heap location.

Suppose that the statement $lhs := \mathsf{new}\ C(e_1, \ldots, e_n)$ occurs somewhere in class D. Let F be a class with a class invariant $ClassInv_F$ that quantifies over the objects of class C. Then class D will have to cooperate in order to maintain the invariant of friend class F. One can verify if the allocation of the new object preserves the invariant by checking if the following formula is implied by the precondition of the creation statement.

$$\neg F.inv \vee ClassInv_F[\mathsf{new}_C/u]$$

Here $[\mathsf{new}_C/u]$ denotes the operation that computes the weakest precondition of the allocation of a new object of class C and its assignment to the fresh local variable u with regard to an arbitrary postcondition. A formal definition of this operation is given elsewhere [4].

A similar proof obligation has to be generated for object invariants. For each object invariant Inv_F that quantifies over C, we must prove that the precondition of the creation statement implies:

$$(\forall o : F \bullet \neg o.inv \vee Inv_F[o/\mathsf{this}][\mathsf{new}_C/u]) \ ,$$

where o is a logical variable that does not occur free in Inv_F.

The additional proof obligations for object creation ensure that the system invariant is maintained throughout a program even when invariants quantify over objects. A proof of this claim is beyond the scope of this paper. We believe, however, that the proof can be obtained by combining Naumann and Barnett's soundness proof [3] with our own results concerning the weakest precondition of object creation [10].

The approach suggested thus far suffers from the problem that it may expose an invariant to other classes in the proposed checks on creation statements. Following Barnett and Naumann [2], we can hide these invariants by allowing a class to declare in its interface an alternative constraint on the creation of objects of its class. We will call such constraints *creation guards*.

The declaration of a creation guard in class C takes the following form:

$$\textsf{guard creation by } P \ ;$$

P is a predicate which reveals the condition under which the environment is allowed to create objects of class C in order to preserve invariants that depend on all objects of class C. It may not refer to private fields of any class.

Let F be a class whose class/object invariant Inv_F quantifies over objects of class C. The creation guard P *protects* this invariant if we can prove that $Inv_F \wedge P \rightarrow Inv_F[\textsf{new}_C/u]$ holds, where u is a fresh local variable. Only invariants that are protected by the creation guard of class C are allowed to quantify over all objects of class C. The proof obligation for a creation statement $lhs := \textsf{new}\ C()$ in the environment now becomes $(\forall o : F \bullet \neg o.inv \vee P)$ for object invariants, and $\neg F.inv \vee P$ for class invariants.

The creation guards should not be seen as additional clauses of the precondition of the constructor method because the precondition describes a different state. The precondition of the constructor method must hold *after* the allocation of the new object, whereas the creation guard should hold *before* the allocation of the new object.

The above described technique gives classes the right to quantify over all objects of another class. It is, in general, unlikely that such invariants hold if there is no (implicit) design principle like the described creational patterns that guides the programmer(s). The technique presupposes in this sense that the involved classes cooperate to fulfill certain goals. Otherwise a class has no reason for restraining the creation of its objects by means of an update guard.

5 The Patterns Revisited

We now apply the techniques just described to the Singleton and Flyweight patterns.

The Singleton Pattern In order to choose the right update guard for the *Singleton* class, we first compute the weakest precondition for an arbitrary creation statement $u := \textsf{new}\ Singleton()$ with regard to the class invariant $I_{\text{Singleton}}$:

$$\begin{array}{ll}
(\forall o : Singleton \bullet o = unique)[\textsf{new}_{Singleton}/u] & (1)\\
\equiv (\forall o : Singleton \bullet ((o = unique)[\textsf{new}_{Singleton}/u])) & \\
\qquad \wedge (o = unique)[u/o][\textsf{new}_{Singleton}/u] & (2)\\
\equiv (\forall o : Singleton \bullet o = unique) \wedge (u = unique)[\textsf{new}_{Singleton}/u] & (3)\\
\equiv (\forall o : Singleton \bullet o = unique) \wedge \textsf{false} &
\end{array}$$

These steps can be explained intuitively as follows. (1) Both the old objects and the new object u must satisfy the property $o = unique$. (2) The validity of $o = unique$ for some old object o is not affected by the creation of a new object. (3) The new object u cannot be equal to the 'old' object *unique*. The resulting formula is equivalent to false. Therefore we choose false as the update guard on creation of *Singleton* objects in the Singleton pattern. It is not difficult to see that no weaker update guard P would protect the invariant.

$$\begin{array}{l}
ClassInv_{Singleton} \wedge P \rightarrow ClassInv_{Singleton}[\textsf{new}_{Singleton}/u]\\
\equiv (\forall o : Singleton \bullet (o = unique)) \wedge P \rightarrow (\forall o : Singleton \bullet (o = unique)) \wedge \textsf{false} \ .
\end{array}$$

An annotated version of the Singleton pattern is listed in Fig. 3. By means of a guard-creation-by clause, the class specifies that the environment may not create instances of *Singleton*.

The annotation of the *getInstance* method reveals a general pattern. A class (or object) is usually unpacked upon entry of the method and packed on exit. We display the precondition of the creation statement (preceded by the assert keyword) in the code for clarity. It is not difficult

```
class $Singleton$ {
  class invariant $(\forall o : Singleton \bullet o = unique)$ ;
  guard creation by false ;
  private static $Singleton\ unique$ ;
  ...
  requires $Singleton.inv$ ;
  ensures $Singleton.inv \wedge$ result $\neq$ null ;
  public $Singleton\ getInstance()$ {
    unpack class $Singleton$ ;
    if $(unique =$ null$)$ {
      assert $\neg Singleton.inv \wedge (\forall o : Singleton \bullet o = unique) \wedge unique =$ null ;
      $unique :=$ new $Singleton()$ ;
    }
    pack class $Singleton$ ;
    return $unique$ ;
  }
}
```

Fig. 3. The annotated Singleton pattern

to see that this precondition implies the proposed constraint on object creation for this example: $\neg Singleton.inv \vee$ false. The keyword private of the constructor method (see Fig. 1) prohibits creation of *Singleton* objects outside the *Singleton* class. Therefore no further constraints have to be checked, and we conclude that the pattern successfully establishes the class invariant.

The Flyweight Pattern The weakest precondition of the allocation of a new instance of class *MyInt* and its assignment to some fresh local variable u with respect to the object invariant of class *IntFactory* is:

$$(\forall i : MyInt \bullet \mathsf{this}.ints[i.x] = i) \wedge \mathsf{false} \wedge \mathsf{this}.ints.length = 1000 \ .$$

The computation is similar to the one above for the *Singleton* class invariant. The weakest precondition reveals that also in this case the fresh object does not satisfy the invariant. We must therefore again constrain creation of particular objects (*MyInt* in this pattern) by means of a guard-creation-by clause with the guard false. This guard is crucial for this pattern because the class *Myint* has a public constructor. The guard forbids creation of *MyInt* objects when the factory object is packed.

The *getInt* method reveals a potential pitfall of the implementation of the pattern, namely that there should be only one instance of the *IntFactory* object. The *getInt* method would presumably be annotated as follows:

```
requires $inv \wedge 0 \leq x < 1000$ ;
ensures $inv \wedge$ result$.x =$ old$(x)$ ;
public $MyInt\ getInt($int $x)$ {
  unpack this ;
  if $(ints[x] =$ null$)$ {
    assert $\neg inv \wedge (\forall i : MyInt \bullet ints[i.x] = i) \wedge ints.length = 1000$
      $\wedge\ 0 \leq x < 1000 \wedge x =$ old$(x) \wedge ints[x] =$ null ;
    $ints[x] :=$ new $MyInt(x)$ ;
  }
  pack this ;
  return $ints[x]$ ;
}
```

The potential pitfall is brought to light by the test on the creation of the *MyInt* object in the *getInt* method. We have to check that the precondition of the creation statement implies the formula $(\forall o : \mathit{IntFactory} \bullet \neg o.\mathit{inv} \vee \mathsf{false})$. Unfortunately, the precondition is too weak to establish this; it merely states $\neg\mathsf{this}.\mathit{inv}$, which implies nothing about other *IntFactory* instances. Achieving the desired invariant requires that all flyweight factories cooperate to achieve the uniqueness of flyweight instances. The easiest way of guaranteeing this cooperation is to ensure that there is only one flyweight factory, which can be done using the Singleton pattern.

Weaker Update Guards The above examples show that the update guard must often be false. Such a guard forbids the creation of objects of a particular class from within the environment. The reason for these strict constraints is that the present system requires that each invariant remains valid when objects are allocated within the environment. One could weaken this requirement by allowing invariants to restrict quantification to all *packed* instances of some class C. Recall that fresh objects are unpacked. They are usually packed for the first time after finishing their initialization in a constructor method. Thus the quantification would only involve initialized objects.

This modification can be achieved by allowing invariants to refer to the invariant bit *inv* of instances of class C. For example, the invariant of the *IntFactory* class would then become:

$$(\forall i : \mathit{MyInt} \bullet i.\mathit{inv} \rightarrow \mathit{ints}[i.x] = i) \wedge \mathit{ints}.\mathit{length} = 1000 \ \ .$$

The pack procedure needs a stronger precondition for this scenario to be sound. Consider a statement pack e where the type of e is C. If friend class F depends on field *inv* of class C, we would have the following additional clause in the precondition of this statement for object invariants:

$$(\forall o : F \bullet \neg o.\mathit{inv} \vee (\mathit{Inv}_F[o/\mathsf{this}][\mathsf{true}/e.\mathit{inv}])) \ \ .$$

The procedure unpack e needs the additional clause $(\forall o : F \bullet \neg o.\mathit{inv} \vee (\mathit{Inv}_F[o/\mathsf{this}][\mathsf{false}/e.\mathit{inv}]))$ in its precondition in these circumstances.

6 Discussion

This paper explained how to deal with unbounded quantification over objects within object/class invariants by showing which constraints are propagated. Creational design patterns often attempt to localize creation and initialization of objects. We analyze a couple of these patterns and determine the degree to which they achieve their desired effect and calculate any residual constraints they impose on their environment.

Several open questions concerning class/object invariants remain. Firstly, object-oriented verification would profit from techniques developed in the concurrency community. As noted by Barnett and Naumann [2], techniques required to establish invariants about shared state in object-oriented programs resemble the rely-guarantee method [11]. The constraints on the environment resemble the interference freedom test for concurrent processes with shared variables. Should we then focus on adapting the rely-guarantee method to the object-oriented setting? Are other techniques for reasoning about such processes applicable too? We believe that the introduction of *histories* [11] will be an important step towards stronger invariants and full information hiding.

Most of the work (including our own) on the verification of object-oriented programs has focussed on the flow of control or typical features (aliasing, object creation, subtyping) in object-oriented programs. What is lacking are compositional proof techniques. Before these can be systematically developed, the question of what compositionality actual means in terms of object-oriented programs and their invariants remains open.

Acknowledgements We thank the anonymous referees for their useful suggestions.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)

2. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: Proc. of MPC. (2004) Available from http://www.cs.stevens-tech.edu/∼naumann/publications/. To appear.
3. Naumann, D.A., Barnett, M.: Towards imperative modules: Reasoning about invariants and sharing of mutable state. In: Proc. of LICS. (2004) Available from http://www.cs.stevens-tech.edu/∼naumann/publications/. To appear.
4. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. In Najm, E., Nestmann, U., Stevens, P., eds.: Formal Methods for Open Object-Based Distributed Systems (FMOODS) VI. Volume 2884 of LNCS. (2003) 64–78
5. Martin, R.C., Riehle, D., Buschmann, F., eds.: Pattern Languages of Program Design 3. The Software Patterns Series. Addison-Wesley (1998)
6. Barnett, M., DeLine, R., Fändrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. In Eisenbach, S., Leavens, G.T., Müller, P., Poetzsch-Heffter, A., Poll, E., eds.: Formal Techniques for Java-like Programs. (2003) Available as Technical Report 408, Department of Computer Science, ETH Zürich.
7. Huizing, K., Kuiper, R.: Verification of object oriented programs using class invariants. In Maibaum, T.S.E., ed.: Proc. of FASE 2000. Volume 1783 of LNCS. (2000) 208–221
8. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Proc. of ECOOP. (2004) Available from http://research.microsoft.com/∼leino/papers.html. To appear.
9. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA Proceedings. (1998)
10. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. Technical Report UU-CS-2003-010, Institute of Information and Computing Sciences, Utrecht University, The Netherlands (2003) Available from http://www.cs.uu.nl/research/techreps/UU-CS-2003-010.html.
11. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification. Volume 54 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2001)