Class Invariants: The end of the road?

(Position Paper)

Matthew Parkinson

University of Cambridge

Matthew.Parkinson@cl.cam.ac.uk

Introduction Since Hoare's seminal paper on data abstraction [5], the class invariant has been the foundation for verifying object-oriented programs. Experience has shown that there are two complications in scaling class invariants to real programs: (1) invariants need to depend on multiple objects; and (2) invariants need to be temporarily broken owing to call-backs. There are several proposals in the literature, which extend class invariants to partially address these two problems. The time seems to be right to pose the following (deliberately provocative!) question: "Is the class invariant the correct foundation for verifying object-oriented programs?"

The basic unit of a Java program is a class, but interesting programs use more than one class. They are decomposed into aggregate[1] structures, containing many inter-related classes collaborating in some function of the system. The aggregate structure is the key concept in any object-oriented program. Hence our verification method needs to describe invariants of these aggregate structures.

The class invariant can only reason about a single object. Using ownership based methodologies [1–4, 7, 8] we can extend the class invariant to some aggregate structures, and allow an object's invariant to depend on objects it owns completely. However, in more complex examples the ownership is less clear cut. Consider two collaborating classes: neither owns the other and each has an invariant depending on the state of the other. Any update to one object will potentially invalidate the invariant of the other object. So how can we update this co-dependent structure? Ideas such as peer invariants [6], friends and update guards [3], and history properties [7], have been used to extend the idea of a class invariant, so that it can depend, soundly, on other objects. But is the complexity of these proposals a sign that the class invariant is not the correct foundation?

Our position is to take a step back and consider a more general foundation. Our approach uses predicates [9, 10] to simply specify the properties of aggregate structures. A class invariant is then just a particular (useful!) predicate.

Subject/Observer The subject/observer pattern, given in Figure 1, exhibits many of the difficulties in reasoning with class invariants. We would like to specify an invariant for the Observer that **this**.sub.val = **this**.cache. However, this invariant does not always hold, because there is a time between when update is called on a Subject, and notify is called on the Observer where the invariant is not satisfied.

Instead of trying to write a property of the individual classes, let us consider a property of the aggregate structure. A single Subject object will have many Observer objects. We expect that, if we update the Subject object, then all the Observer objects will be notified and their status suitably updated. The aggregate structure

```
class Subject {
  List obs; int val;
  Subject()
  { obs = new List(); }
  void register(Observer o)
  { this.obs.add(o);
    o.notify(); }
  void update(int n) {
    this.val = n;
    foreach(Observer o:obs)
      o.notify();
  }
  int get() { return this.val; }
}
```

```
class Observer {
  Subject sub;
  int cache;
  Observer(Subject s) {
    this.sub = s;
    s.register(this);
  }
  void notify()
  { this.cache = s.get(); }
  int val()
  { return cache; }
}
```

Figure 1. Source code for subject/observer pattern

can be specified with the following predicate definition:

$$SubObs(s, O, v) \stackrel{\mathbf{def}}{=} Sub(s, O, v) \wedge \forall o \in O.Obs(o, s, v)$$

Here s is the Subject, O is a set (list) of Observers and v is the current value of the Subject. In the definition, $Sub(s, O, v)$ represents a Subject object s, that has O Observers, and current value, v; and $Obs(o, s, v)$ represents an Observer object, with Subject s that had value v last time it was notified. The ownership properties are captured directly by using separation logic (see [9, 10] for more details).[2] We give the definitions of the predicates in Figure 2.

The $SubObs$ predicate can be seen as the invariant of the aggregate structure. Accordingly, it should hold on the entry and exit of every public method of the aggregate (this is just a generalization of a class invariant). So the "aggregate invariant" should hold on the entry and exit of the two constructors, the update method of the Subject, and the val method of the Observer. The other methods (register, notify and get) are internal to the aggregate structure. We present the specifications of the methods and constructors in Figure 2.

When verifying the Subject methods, we use the definition of $SubObs$ and Sub predicates, and verifying the Observer methods we can use both $SubObs$ and Obs definitions. Hence, the Subject is independent of the Observer, and vice-versa, but they are both dependent on the aggregate structure to which they belong, hence our reasoning remains modular.

We present an example verification of the constructor of the Observer:

$\{SubObs(\mathsf{s}, O, v) * \mathsf{this.sub} \mapsto \_ * \mathsf{this.cache} \mapsto \_\}$

[1] Here we mean *aggregate* in its most general sense to capture additionally the UML meanings of *association* and *composition*.

[2] *Separation logic in a footnote.* Separation logic is an extension to Hoare logic that allows reasoning about heap data-structures. It has two new connectives: $P * Q$ means the state can be split into two disjoint parts, one satisfying P and the other Q; $x.f \mapsto y$ means the object x has a field f containing y; and $\circledast_{i \in \{i_1, \ldots, i_n\}}.P(i)$ means $P(i_1) * \ldots * P(i_n)$.

Predicates	Method	Pre-condition	Post-condition
$SubObs(s, O, v)$	s=Subject()	emp	$SubObs(\mathsf{s}, \emptyset, \_)$
$\stackrel{\mathbf{def}}{=} Sub(s, O, v) * \circledast_{o \in O} Obs(o, s, v)$	s.register(o)	$Sub(\mathsf{s}, O, v) * Obs(\mathsf{o}, \mathsf{s}, \_)$	$Sub(\mathsf{s}, \mathsf{o} :: O, v) * Obs(\mathsf{o}, \mathsf{s}, v)$
$Sub(s, O, v)$	s.update(n)	$SubObs(\mathsf{s}, O, v)$	$SubObs(\mathsf{s}, O, \mathsf{n})$
$\stackrel{\mathbf{def}}{=} \exists l.\ s.\mathsf{val} \mapsto v * s.\mathsf{obs} \mapsto l * list(l, O)$	ret=s.get()	$Sub(\mathsf{s}, O, v)$	$Sub(\mathsf{s}, O, v) \wedge \mathsf{ret}{=}v$
$Obs(o, s, v)$	o=Observer(s)	$SubObs(\mathsf{s}, O, v)$	$SubObs(\mathsf{s}, \mathsf{o} :: O, v)$
$\stackrel{\mathbf{def}}{=} o.\mathsf{cache} \mapsto v * o.\mathsf{sub} \mapsto s$	o.notify()	$Sub(s, O, v) * Obs(\mathsf{o}, s, \_)$	$Sub(s, O, v) * Obs(\mathsf{o}, s, v)$
	ret=o.val()	$SubObs(s, O, v) \wedge \mathsf{o} \in O$	$SubObs(s, O, v) \wedge \mathsf{ret}{=}v$

where $o \in (o' :: O') \stackrel{\mathbf{def}}{=} o = o' \vee o \in O'$ and $o \in \emptyset \stackrel{\mathbf{def}}{=} false$

Figure 2. Specification of subject/observer pattern

```
    this.sub = s;
{SubObs(s, O, v) * this.sub↦s * this.cache↦_}
{SubObs(s, O, v) * Obs(this, s, _)}
{Sub(s, O, v) * (⊛_{o∈O} Obs(o, s, v)) * Obs(this, s, _)}
    s.register(this);
{Sub(s, this::O, v) * (⊛_{o∈O} Obs(o, s, v)) * Obs(this, s, v)}
{Sub(s, this::O, v) * (⊛_{o∈(this::O)} Obs(o, s, v))}
{SubObs(s, this::O, v)}
```

Interestingly, the Observer's constructor causes problems for class invariant based verification, because it calls another class's method, which in turn calls back into the Observer. This complicated calling pattern is forbidden in the class invariant approach, and requires additional machinery [1]. Simply by using predicates over aggregates we avoid such constraints.

Conclusion We have demonstrated a straightforward proof of the subject/observer pattern. We have not invented new methodology or ownership types. We have simply considered a property of an aggregate structure. These properties, we claim, are the key to verifying object-oriented programs, and should not be shoehorned into class invariants. Class invariants have taken us a long way, but properties of aggregate structures should now form the foundation of verification.

Acknowledgments We thank Gavin Bierman, Sophia Drossopoulou, and Peter O'Hearn for encouraging me to write this position paper.

References

[1] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\sharp$ programming system: An overview. In *Proceedings of CASSIS*, pages 49–69, 2005.

[3] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.

[4] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 2005.

[5] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[6] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[7] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *Proceedings of ESOP*, LNCS, 2007.

[8] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[9] M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. UCAM-CL-TR-654.

[10] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.