Design Pattern-Based Extension of Class Hierarchies to Support Runtime Invariant Checks

John Lasseter and John Cipriano

Fairfield University, Fairfield CT 06824, USA
jlasseter@fairfield.edu, johnmikecip@gmail.com

Abstract. We present a technique for automatically weaving structural invariant checks into an existing collection of classes. Using variations on existing design patterns, we use a concise specification to generate from this collection a new set of classes that implement the interfaces of the originals, but with the addition of user-specified class invariant checks. Our work is notable in the scarcity of assumptions made. Unlike previous design pattern approaches to this problem, our technique requires no modification of the original source code, relies only on single inheritance, and does not require that the attributes used in the checks be publicly visible. We are able to instrument a wide variety of class hierarchies, including those with pure interfaces, abstract classes and classes with type parameters. We have implemented the construction as an Eclipse plug-in for Java development.

1 Introduction

Several, if not most, mainstream languages include features to support object-oriented programming, yet most of these (C++, C#, Java, Python, etc.) lack any native language support for the specification and runtime checking of class invariants. While it is usually easy enough to implement the invariant predicates themselves, manual addition imposes further requirements in order to implement the operational requirements of invariant checking and to handle the interplay of invariant specification and inheritance. Class invariants are further troublesome in that they involve direct access to an object's attributes. This makes manual addition particularly unappealing, as the available choices are invasive with respect to the original interface and implementation (to which we may not have access), compromise encapsulation, and are error-prone if done manually.

This paper presents a lightweight, non-invasive technique for automatically extending a collection of class definitions with a corresponding collection of structural invariant checks. The invariants are given as a stand- alone specification, which is woven together with the original source files to produce a new collection of drop- in replacement classes that are behaviorally indistinguishable from the originals in the absence of invariant- related faults but will expose such faults in a way that the original classes do not. Each replacement is defined to be a subclass (indeed, a *subtype* [1]) of the original class whose functionality it extends, and it can thus be substituted in any context in which the original occurs. The generation is itself completely automatic, and the incorporation into a test harness or other program is nearly seamless. We focus here on the Java language, a choice that complicates the overall strategy in some ways while simplifying it in others.

2 Background and Related Work

A *class invariant* is a conjunction of predicates defined on the values of an object's individual attributes and on the relationships between them. It characterizes an object's "legal" states, giving the predicates that must hold if the object is to represent an instance of that abstraction. Usually, a class invariant is given in conjunction with the *contracts* for each publicly-visible method of a class, *i.e.*, the preconditions that must hold on arguments to each method call and the consequent guarantees that are made as postconditions upon the method's return. Unlike the contracts, however, a class invariant is a property concerning only an object's *data values*, even (especially) when those values are not publicly visible. An invariant must hold at every point between the object's observable actions, *i.e.* upon creation of any object that is an instance of this class and both before and after every publicly-visible method call [2,3]. At other points, including non-visible method calls, it need not hold, and runtime checks are disabled in this case. Further, since runtime invariant checks can impose a non-trivial performance penalty on a system, in general, it is desirable to have a mechanism for leaving the checks in place during testing, while removing them from a final, production system. Finally, there is an important interplay between the subtype relation (which determines when one object can safely be substituted in a context calling for another [1]) and class invariants: if B is a subtype of A (as well as a subclass) then the invariant for B must include all of the constraints in A's invariant [2,3].

Some languages offer native support for invariant checking, but for Java and other languages that lack this, including such checks is challenging. A common approach is to make use of the language's assertion mechanism, by including assertions of the invariant at the end of each constructor body and at the beginning and end of the body of each public method [2]. If the language's assertions mechanism is used, disabling the checking functionality after testing is usually quite easy. However, this approach carries the disadvantage of requiring the class designer to code not only the predicates themselves but also an explicit handling of the inheritance requirements and the full execution model, discussed above. Both of these tasks must be implemented for each invariant definition, in each class.

To avoid the implementation burden of the assertions approach, we can use a tool that generates the invariant checks from either specialized annotations of the source code [4,5,6] or reserved method signatures [7,8,9]. Essentially, such tools offer language extensions to resemble native support for invariant definitions. In comparison to assertion-based approaches, they eliminate the requirement of implementing the execution model, a clear advantage. As with the assertions approach, annotation approaches are invasive, in that they require modification of the original source code. More substantially, the approach generally requires the use of a specialized, nonstandard compiler, whose development may not keep up with that of the language[1].

Instead, we can view the addition of runtime invariant checking across a class *hierarchy* as a kind of cross-cutting concern, *i.e.* code that is defined across several classes and hence resists encapsulation. Under this view, it is natural to approach this problem as one of aspect-oriented programming (AOP) [11], in which we can use a tool such as AspectJ [12] to define the checks separately as aspects. The entry and exit points of each method become the join points, the point cuts are inferred from a class's method signatures, and the invariant check itself becomes the advice [13,14]. Unlike annotation-based approaches, aspect weaving can be done without the need for a non-standard compiler, either through source code transformation or byte code instrumentation [15]. However, the AOP approach also presents several difficulties. For example, Balzer *et al.* note that mainstream tools such as AspectJ lack a mechanism to enforce the requirement that the definition of a class's invariant include the invariant of its parent class [16]. It is possible to write invariant checking "advice" so that it correctly calls the parent class's invariant check, but this must be done manually (*e.g.* [13]). A similar problem occurs in implementing the correct disabling of checks on non-public calls. Lastly, because aspects cannot in general be prevented from changing an object's state, the weaving of additional aspects may compose poorly with the aspect that provides the invariant check [17,16,18]. It is possible that another aspect could break the class invariant, and since interleaving of multiple aspects is difficult to control, it is possible the two aspects could interleave in such a way as to make the invariant failure go undetected.

The work closest in spirit to our own is the design pattern approach of Gibbs, Malloy, and Power ([19,20]. Targeting development in the C++ language, they present a choice of two patterns for weaving a separate specification of invariant checks into a class hierarchy, based on the well known *decorator* and *visitor* patterns [21]. However, the decorator approach involves a fairly substantial refactoring of the original source code. Moreover, the authors note that this technique interacts poorly with the need to structure invariant checks across a full class hierarchy. The refactoring in this case is complex, and it requires the use of multiple inheritance to relate the decorated classes appropriately, making it unsuitable for languages such as Java, which support only single inheritance. Their alternative is an application of the visitor pattern, in which the invariant checks are implemented as the *visit* methods in a single *Visitor* class. This pattern usually requires that the classes on the "data side" implement an *accept* method, which is used to dispatch the appropriate *visit* method, but in their use of it, only the top of the class hierarchy is modified to be a subclass of an "invariant facilitator", which handles all *accept* implementations. However, successful

[1] For example, JML has not seen active development since version 1.4 of the Java language [10].

implementation of the *visit* methods rests on the assumption that all fields are either publicly visible or have their values readily available through the existence of accessor ("getter") methods. Unless the language simply lacks a mechanism to hide this representation (*e.g.* Python), such exposure is unlikely to be the case, as it violates encapsulation, permitting uncontrolled manipulation of an object's parts, either directly or through aliasing [2].

The central thesis of our work is that, under assumptions common to Java and other statically-typed OO languages, these limitations—source code modification, multiple inheritance, and public accessibility of fields—are unnecessary for a design-pattern approach. The remainder of the present paper shows how to relax them.

3 Weaving Invariant Checking from Specifications

Our approach draws from the Gibbs/Malloy/Power design pattern efforts and from ideas in AOP in the treatment of invariant specifications as a cross-cutting concern. We begin with an assumption that the class invariants are given in a single specification file, separate from classes that they document. Each constraint is a boolean- valued Java expression, with the invariant taken to be the conjunction of these expressions. We assume (though do not hope to enforce) that these expressions are free of side effects, and that the invariant given for a child class does not contradict any predicates in inherited invariants. Otherwise, the particulars of the specification format are unimportant. The current version of our tool uses JSON [22], but any format for semi-structured data will do.

We focus on the Java programming language, which means that we assume a statically-typed, object-oriented language, with introspective reflection capabilities, support for type parameters in class definitions, single inheritance (though implementation of multiple interfaces is possible), and a uniform model of virtual method dispatch. We make some simplifications of the full problem. Specifically, we work only with synchronization- free, single-threaded, non-*final* class definitions, and we consider only instance methods of a class that admit overriding, *i.e.*, non-*static*, non-*final*[2] method definitions. We do not consider anonymous inner class constructs nor the *lambda expressions* planned for Java 8 [23]. Finally, we assume a class's field visibility grants at least access through inheritance (*i.e. protected* accessbility or higher). This last is made purely for the sake of simplifying the technical presentation, since, as discussed in section 5, introspection makes it easy to handle variables of any accessibility.

3.1 An Inheritance-Based Approach

As a first effort, we will try an approach that leverages the mechanism of inheritance and the redefinition of inherited method signatures through subtyping

[2] The *final* keyword has two uses in Java: to declare single-assignment, read-only variables and to prohibit extension of classes or overriding of methods. The latter form is equivalent to the *sealed* keyword in C#, and it is this usage we avoid here.

```
public class $A'$<$T_{A'}$> extends $A$<$S_{A'}$> {
  private int $\delta$ = 0;
  public $A$($\overrightarrow{\tau_A\ x}$) {
    super($\overrightarrow{x}$);
    $\delta$ = $\delta$ + 1;
    $\phi_2$();
  }
  public $\tau_{f_A}$ $f_A$($\overrightarrow{\sigma_{f_A}\ y}$) {
    $\phi_1$();   $\tau_{f_A}$ $\chi$ = super.$f_A$($\overrightarrow{y}$);   $\phi_2$();
    return $\chi$;
  }
  private boolean $inv$() { return $\rho_A$; }
  private void $\phi_1$() {
    if ($\delta$ == 0 && !$inv$())
      $\langle$ handle invariant failure $\rangle$
    $\delta$ = $\delta$ + 1;
  }
  private void $\phi_2$() {
    $\delta$ = $\delta$ - 1;
    if ($\delta$ == 0 && !$inv$())
      $\langle$ handle invariant failure $\rangle$
  }
}
```

Fig. 1. Inheritance-based generation of invariant checks

polymorphism. The idea is to derive from a class and its invariant a subclass, in which we wrap the invariant in a new, non-public method (perhaps with additional error reporting features), similar to the "*repOK*" approach advocated by Liskov and Guttag [2]. To this new subclass, we also add methods ϕ_1 and ϕ_2 to handle the checking tasks at (respectively) method entry and exit points, and we use these to define constructors and overridden versions of every public method.

Let A be a class, with parametric type expression T_A defined on type parameters S_A, field declarations $\overrightarrow{\tau\ a}$, invariant ρ_A, constructor definition $A(\overrightarrow{\tau_A\ y})$ and public method $\tau_f\ f(\overrightarrow{\sigma_f\ z})$.

```
public class $A$<$T_A$> {
  $\overrightarrow{\tau\ a}$;
  public $A$($\overrightarrow{\tau_A\ x}$) { ... }
  public $\tau_{f_A}$ $f_A$($\overrightarrow{\sigma_{f_A}\ y}$) { ... }
}
```

We extend A with runtime checking of ρ_A by generating the subclass in Fig. 1, where $T_{A'}$ and $S_{A'}$ are identical to T_A and S_A (respectively), except perhaps for renaming of type parameters (*i.e.*, they are α-equivalent).

For each constructor in A', the body executes the "real" statements of the corresponding superclass constructor, followed by a check of ρ_A, whose execution is itself controlled by the ϕ_2 method. Likewise, the body of each public method f_A wraps a call to the superclass's version between checks of ρ_A, with execution controlled by the ϕ_1 and ϕ_2 methods. If f_A returns a value, then this value is captured in the overridden version in a "result" variable, χ. A method or constructor call is publicly-visible precisely when the call stack depth on a given A' object is 0, and this value is tracked by the additional integer-valued field δ. The ϕ_1 and ϕ_2 methods increment/decrement δ as appropriate, evaluating ρ_A only if $\delta = 0$.[3]

The inheritance-based approach suggests an easy mechanism for reusing code while adding the necessary invariant checks and capturing the distinction between publicly-visible and inner method calls. For the user, the burden consists of replacing constructor calls to A with the corresponding calls for A'. This may be an excessive requirement when A objects are used in production-level code, but in many settings where invariant checking is desirable, such constructor calls are limited to only a handful of sites. In the JUnit framework, for example, integration of A' objects into unit tests for A is likely quite simple, as object construction occurs mainly in the body of a single method, *setUp*.

Note the assumptions of uniform polymorphic dispatch and non-*final* declarations here. If a class cannot be extended (*e.g. String* and other objects in the *java.lang* package), then construction of a subclass that implements the invariant checks is obviously impossible. Similarly, a method whose dispatch is statically determined cannot be transparently overridden, and if declared *final*, it cannot be overridden at all. In many languages (notably, C# and C++) the default convention is *static* dispatch, with dynamic binding requiring an explicit *virtual* designation; in such cases, the inheritance construction is far less convenient and may be impossible without some refactoring of the original source code.

Unfortunately, our first attempt fails in two critical ways, which becomes apparent when we attempt to construct the invariant-checking extension across a hierarchy of class definitions. First of all, the inheritance hierarchy of a collection of objects requires a corresponding structure in the composition of invariant checks. This problem is very similar to the one encountered in the "decorator" approach of [20], but the multiple-inheritance solution given there is unavailable in a single-inheritance language such as Java. Consider a class B that is a subtype of A (written $B <: A$):

```
public class $B$<$T_B$> extends  $A$<$S_B$>{
   $\overrightarrow{\tau\ b}$
   public $B$($\overrightarrow{\tau_B\ y}$) {  ...  }
   public $\tau_{g_B}$ $g$($\overrightarrow{\sigma_{g_B}\ z}$) {  ...  }
}
```

Figure 2 depicts the problem[4]. The invariant for a B object, inv_B, must include the A invariant—*i.e.*, $inv_B = inv_A \wedge \rho_B$. However, a B' object cannot access the fields of its associated B object through inheritance and also reuse the functionality of the inv_A method. We might choose to have B' descend from A' instead,

[3] In the presence of concurrency, we would need a more sophisticated mechanism; keeping track of the call stack depth on an object for each thread, synchronizing all method calls on the object's monitor lock, and so on.

[4] There and throughout this paper, we write $[S/\tau]\,T$ to denote the substitution of type expression τ for the type parameter S in expression T, and use the shorthand $[S_1/\tau_1, S_2/\tau_2]\,T$ to denote the composition of type expressions $[S_1/\tau_1]\,[S_2/\tau_2]\,T$.

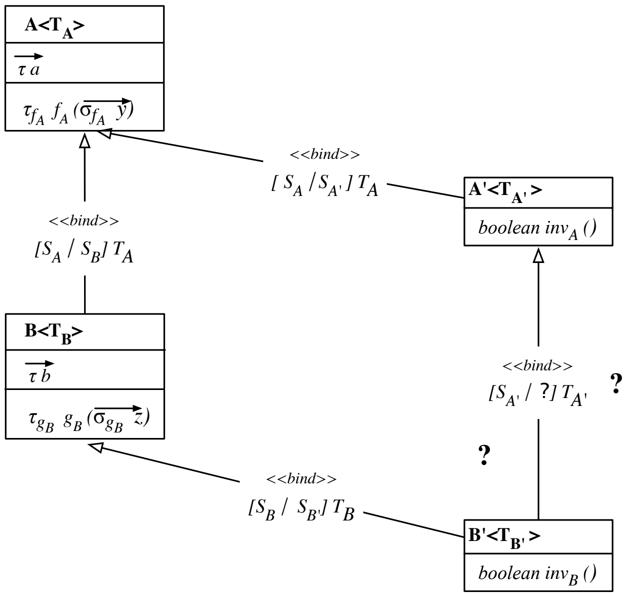

Fig. 2. Design flaw in the naive inheritance approach

but this only works if all fields in B are publicly accessible. As discussed above, this is unlikely to be the case.

The second, related failure is that inheritance does not facilitate a correct binding of the type parameters. Again, this is clear from Fig. 2. An instantiation of B supplies a type τ to the parameters S_B, which is used in turn to bind the parameters S_A with argument $[S_B/\tau, S_A/S_B]\, T_A$. When we instantiate B' instead, this same τ binds the parameters $S_{B'}$, with the resulting chain of arguments binding A's parameters S_A as $[S_{B'}/\tau, S_B/S_{B'}, S_A/S_B]\, T_A$. For correct use of the A' invariant check in this $B'\langle\tau\rangle$ object, we would need to bind the type parameter of A', $S_{A'}$, in the same way we do A's parameter, S_A; *i.e.* with argument $[S_{B'}/\tau, S_{A'}/S_{B'}, S_A/S_{A'}]\, T_A$, a binding that cannot be ensured, unless B' is a subclass of A'.

3.2 Exposing the Representation

Though unsuccessful on its own, we can use the inheritance approach of Section 3.1 as the basis for an auxiliary pattern, which we call an *exposure pattern.* The idea is to construct from the original hierarchy a corresponding set of classes that offers the interface of the original collection and in addition, a controlled exposure of each object's representation. The machinery for checking the invariants is factored into separate classes, as discussed in Section 3.3, below.

Consider a class definition

```
public class $A$<$T_A$> {
  $\tau_1$ $a_1$;   ...   $\tau_k$ $a_k$;
  public $A$($\overrightarrow{\tau_A\ y}$) {  ...  }
  public $\tau_{f_A}$ $f_A$($\overrightarrow{\sigma_{f_A}\ z}$) {  ...  }
}
```

We derive the *exposure* interface

```
public interface $IA_E$<$T_{A''}$> {
  $\tau_1$ $\gamma_{a_1}$();
    ...
  $\tau_m$ $\gamma_{a_m}$();
}
```

and *exposed* class

```
public class $A_E$<$T_{A'}$>  extends  $A$<$S_{A'}$>  implements $IA_E$<$S_{A'}$> {
  private int $\delta$ = 0;
  private void $\phi_1$() {...}
  private void $\phi_2$() {...}
  protected boolean $inv$($InvV$ $v$) {...}

  public  $A_E$($\overrightarrow{\tau_A\ y}$) {
    super($\overrightarrow{y}$);    $\delta$ = $\delta$ + 1;    $\phi_2$();
  }
  public  $\tau_{f_A}$  $f_A$($\overrightarrow{\sigma_{f_A}\ y}$) {
    $\phi_1$();    $\tau_{f_A}$ $\chi$ = super.$f_A$($\overrightarrow{y}$);  $\phi_2$();
    return $\chi$;
  }


  public $\tau_1$ $\gamma_{a_1}$(){ return $a_1$; }
    ...
  public $\tau_m$ $\gamma_{a_m}$(){ return $a_m$; }
}
```

where $T_{A'}$, $T_{A''}$ and $S_{A'}$, $S_{A''}$ are α-equivalent to T_A and S_A, as above. Note that the fields $a_1 \dots a_m$ include all of the original $a_1 \dots a_k$ and perhaps others, as discussed on page 171, below. The constructors and public methods in A_E are overridden in exactly the same manner as in the A' class of Section 3.1, and likewise the implementation of the $\phi_1()$ and $\phi_2()$ methods. The representation exposure happens through the $\gamma_{a_i}()$, a set of raw "getter" methods that expose each of the object's fields. In the presence of inheritance, the corresponding structure is realized not in the derived class but in the derived *interfaces*. Thus, for example,

```
public class $B$<$T_B$> extends  $A$<$S_B$>{
  $\tau_1$ $b_1$;   ...   $\tau_l$ $b_l$;
  public $B$($\overrightarrow{\tau_B\ y}$) {  ...  }
  public $\tau_{g_B}$ $f$($\overrightarrow{\sigma_{g_B}\ z}$) {  ...  }
}
```

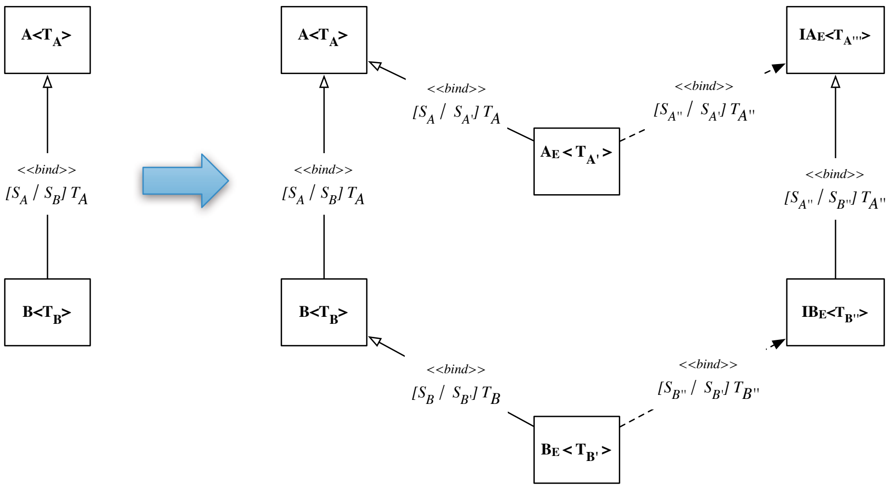

Fig. 3. Exposure pattern construction

gives rise to the interface and class definitions

```
public interface $IB_E$<$T_{B''}$>  extends  $IA_E$<$S_{B''}$> {
  $\tau_1$ $\gamma_{b_1}$();
    ...
  $\tau_n$ $\gamma_{b_n}$();
}
public class $B_E$<$T_{B'}$> extends $B$<$S_{B'}$> implements  $IB_E$<$S_{B'}$> { ... }
```

The construction is illustrated in Fig. 3.

Correctness. Since the type expressions in a class definition are copied to its exposed class and interface (perhaps with α-renaming of the parameters), it is easy to see that

Proposition 3.1. *For any type expression τ, an instance of a class A has type A<$T(\tau)$> if and only if A_E and IA_E have types A_E<$T(\tau)$> and IA_E<$T(\tau)$>, respectively.* $\square$

The construction of the accessor methods is less obvious. While we construct $\gamma_{a_i}()$ for each of the fields $\{a_1, \ldots, a_k\}$, we may need to construct others, as well, in case the invariant ρ_A makes reference to any inherited fields for which we have not already constructed an interface. This can happen in the case of an incomplete specification of the class hierarchy and invariants. The simplest way to handle this is to include in the interface a $\gamma_{a_i}()$ for each declared field in the corresponding A classes and also for each variable that occurs without explicit declaration in the the predicate ρ_A. However, we can leverage the inheritance of interfaces to eliminate redundant declarations (though not implementations, as discussed below).

To make the construction precise, we denote the *free variables* of the predicate ρ_A by $FV(\rho_A)$, *i.e.* those variables that occur in ρ_A without being explicitly declared in ρ_A. Conversely, the *bound* variables in a class A, $BV(A)$, are the instance fields declared in A. The following definition captures the notion of variables that are "free" in A through inheritance:

Definition 3.2. *Let P be a specification of a collection of classes and their associated invariants. For a class A,* the set of fields exposed through inheritance in *A, $\mathcal{I}(A)$, is defined by*

$$\mathcal{I}(A) = \begin{cases} \emptyset & \text{, if } A \text{ has no superclass specified in } P \\ \mathcal{I}(C) \cup BV(C) \cup FV(\rho_C) & \text{, if } A <: C \text{ and } C \text{ is specified in } P \end{cases}$$

We use this to define the necessary method signatures in each exposure interface.

Definition 3.3. *Given class A and invariant ρ_A, the body of IA_E consists of the the signatures*

$$IA_E = \{\tau_{a_i}\gamma_{a_i}();\ \mid a_i \in BV(A) \cup FV(\rho_A) \setminus \mathcal{I}(A)\}$$

where each τ_{a_i} is the declared type of a_i.

Definition 3.4. *For a field, τ_{a_i} a_i, either declared in or inherited by a class A, we say that a_i is* successfully exposed for A *if either*

- *there is an interface IA_E and subclass*
  `class` A_E `extends` A `implements` IA_E
  such that IA_E includes a method interface
  τ γ_{a_i}`();`
  and for every A_E object o, $o.\gamma_{a_i}()$`==` $o.a_i$
- *A is a subclass of C, and a_i is successfully exposed for C.*

Given A and ρ_A, the construction for IA_E in Definition 3.3 and the accompanying implementation A_E combine to give us the representation exposure we need for ρ_A. In particular,

Proposition 3.5. *If $x \in FV(\rho_A)$, then x is successfully exposed for A.* □

Space Requirements. The primary difference between the exposure pattern construction and the inheritance-based effort of Section 3.1 lies in the construction of the exposure interfaces, whose inheritance structure is congruent to that of the original collection of classes. Like the earlier attempt, however, the collection of exposed *classes* does not share this same relation, and as a consequence, both approaches are subject to some unfortunate redundancy consequences. In particular, we cannot reuse code between distinct exposed classes, even when the classes they expose are related by inheritance. For example, if a class A contains fields a_1 and a_2 and public method $f()$ then the exposed class A_E must override $f()$, and it must include exposure methods γ_{a_1} and γ_{a_2}, according to the interface IA_E. If $B <: A$ contains fields b_1, b_2, and method $g()$, then it must override not only $g()$ but also $f()$, with the body of the overridden $f()$ identical to that in A_E. Likewise, it must implement not only the γ_{b_1} and γ_{b_2} methods from the IB_E interface, but also γ_{a_1} and γ_{a_2}.

Happily, all of this is easily automated, and it is reasonable to suppose the space overhead manageable. Note first that, with the exception of classes at the top of a specified hierarchy, the size of the interface generated for a class is proportional to the number of fields in that class. Recalling Definitions 3.2 and 3.3, we can see that this is so because

Proposition 3.6. *Let C be a class included in a specification P. For every class $A <: C$, $FV(\rho_A) \setminus \mathcal{I}(A) = BV(A)$.*

In other words, only for classes specified at the top of an inheritance hierarchy will we ever need to generate additional γ declarations in the corresponding interfaces. In all other cases, the accessor interfaces for inherited fields are inherited from the corresponding parent interfaces. Hence, the space required to extend a collection of classes depends only on the size of each class and the depth of the inheritance relationship in the collection. Specifically, if we assume a bound of n new field and method definitions on each class and an inheritance depth of h, then the overall space growth is given by

$$\sum_{i=1}^{h}\left(\sum_{j=1}^{i} n\right) = \left(\sum_{i=1}^{h} i\right) n = \left(\frac{h(h+1)}{2}\right) n \in \mathcal{O}(h^2 n)$$

It is difficult to give a general characterization of either n or h, but there is reason to suspect that both are manageable values in practice. McConnell recommends a limit of 7 new method definitions in a class [24]. Shatnawi's study [25] finds no significant threshold value for h. Classes in the JDK's *java.** and *javax.** libraries implement anywhere from less than 10 to over 100 new methods, while the largest depth of any inheritance tree is 8.

3.3 Adding the Invariant Checks

As in Gibbs/Malloy/Power [20], we implement the runtime invariant checks themselves through an application of the *visitor pattern* [21], in which the methods implementing the invariant checks are aggregated into a single class (the "visitor"), with the appropriate method called from within the class being checked (the "acceptor"). Unlike their approach, however, our exposure pattern allows us to do this without modification of any part of the original source files, not even at the top of the inheritance hierarchy.

Suppose we have a class A<T_A>, with invariant ρ_A. From these, we generate the exposed class A_E<$T_{A'}$> and the exposure interface IA_E<$T_{A''}$>, as in Section

3.2. The specification of ρ_A and the access methods defined for IA_E are used to generate an invariant checking "visitor" class:

```
public class $InvV$ {
  public <$T_{A_I}$>  void  $v_A$($IA_E$<$S_{A_I}$> $obj$) {
    $\tau_1$ $a_1$  = obj.$\gamma_1$();
      ...
    $\tau_n$ $a_n$  = obj.$\gamma_n$();

    ⟨⟨ $compute$ $\rho_A$ $and$ $return$ $the$ $result$ ⟩⟩
  }
}
```

where T_{A_I} and S_{A_I} are equivalent to T_A and its parameters S_A, as above.

Runtime checking of ρ_A is invoked in the A_E methods through calls to that class's *inv* method, which serves as the "accept" method, handling dispatch of the appropriate invariant check:

```
public class $A_E$<$S_{A'}$> extends  $A$<$S_{A'}$>  implements $IA_E$<$S_{A'}$> {
  private int $\delta$ = 0;
  private void $\phi_1$() { ...}
  private void $\phi_2$() { ...}   // (as defined in Section 3.1)
  private boolean $inv$($InvV$ $v$) {
    $v.v_A$(this);
    return $v$.valid();
  }

    ...

}
```

Note that each $v_A()$ method in $InvV$ takes an argument of type IA_E and not A_E. This is necessary, because of the need to compose an invariant check with that of the object's superclass in each invariant method. For example, if we have $B <: A$, we define $v_B()$ as

```
  public <$T_B$>  void  $v_B$($IB_E$<$S_B$> $obj$) {
    $v_A$( ($IA_E$<$S_B$>) obj);
    ⟨⟨ $compute$ $\rho_B$, $as$ $above$ ⟩⟩
  }
```

Since A_E and B_E are not related by inheritance, it would not be possible to directly cast `obj` to its superclass's exposed version. Fortunately, the interface is all we need.

Finally, although we structure our solution here according to the traditional visitor pattern conventions, we do not really need the full generality of that pattern. In particular, it is unnecessary to support full double dispatch, as we only need one instance of $InvV$, and no $v_i()$ method will ever invoke a call back to the $inv()$ method of an object (not even indirectly, since the ϕ_1 and ϕ_2 methods in a class prevent a call to $inv()$ if one is already running). Our implementation of this approach as an Eclipse plugin instead drops the $InvV$ parameter from every *inv* method, relying instead on a single, static instance of the invariant visitor:

```
private boolean inv() {
  InvVv = InvV.getInstance();
  ...
}
```

4 Example: Unit Testing

Method contracts and class invariants are particularly useful in testing. In combination with test oracles, the use of runtime invariant and pre/post-conditions checks improves the exposure of faults as well as the diagnosability of faults when they are detected [26,27]. Our implementation as an Eclipse plug has proven useful in diagnosing invariant-related faults.

For example, a simple `List` interface provides an abstraction for the list data type. A standard way to implement this is with an underlying doubly-linked list, in which we keep a pair of "sentinel" head and tail nodes, with the "real" nodes in the list linked in between:

```
public abstract class AbstractList<T> implements List<T> {
    protected int size;
    ...
}

public class DLinkedList<T> extends AbstractList<T> implements List<T> {
    // inherited from AbstractList:  int size
    protected DNode<T> head,  tail;  ...
}
```

Among other predicates, the invariant for *DLinkedList* requires that $\forall n \neq tail$, $n.next.prev = n$.

This was given as part of a project for the first author's data structures course, and among the student submissions received was this implementation of *remove()*, in which the *cur.prev* pointer is not correctly updated:

```
public boolean remove(T v) {
    DNode<T> cur = head.next;
    while (cur != tail) {
        if (cur.data.equals(v)) {
            DNode<T> prev = cur.prev;  cur = cur.next;  prev.next = cur;
            size--;
            return true;
        } else
           cur = cur.next;
    }
    return false;
}
```

A JUnit test suite failed to uncover this fault, passing this and the tests for 12 other methods:

```
public void testRemove() {
    ls.add("a"); ls.add("b"); ls.add("c"); ls.add("d"); ls.add("a"); ls.add("d");
    int sz = ls.size();
    assertTrue(ls.remove("a"));    assertTrue(ls.size() == sz - 1);
    sz = ls.size();
    assertTrue(!ls.remove("**"));    assertTrue(ls.size() == sz);
}
```

From the original source code and a specification of invariants our tool generates the classes and interfaces

```
public interface IExposedAbstractList<T> {
    int _getSize();
}
public interface IExposedDLinkedList<T> extends IExposedAbstractList<T> {
    DNode<T> _getHead();
    DNode<T> _getTail();
}

public abstract class ExposedAbstractList<T>
                       extends AbstractList<T> implements IExposedAbstractList<T> { ... }

public class ExposedDLinkedList<T>
                       extends DLinkedList<T> implements IExposedDLinkedList<T> { ... }

public class RepOKVisitor {
    ...
    public <T> void visit(IExposedAbstractList<T> _inst) { ... }
    public <T> void visit(IExposedDLinkedList<T> _inst) { ... }
    ...
}
```

Objects in a JUnit test suite are constructed in the *setUp()* method, and a simple modification was all that was needed to cause *testRemove()* to fail appropriately:

```
protected void setUp() {
//    ls = new DLinkedList<String>();
    ls = new ExposedDLinkedList<String>();
}
```

5 Conclusion and Future Work

The design pattern given here provides a fairly seamless approach for adding correct runtime invariant checking to a class hierarchy, through the construction of drop-in replacements that can be removed as easily as inserted. In addition to the core material presented here, there are a number of extensions possible.

For example, the presentation in this paper relies on the assumption above that all fields in a class are accessible through inheritance. Happily, this is an easy if tedious limitation to overcome. If instead the field is declared with only intra-object or intra-class access (*e.g.* Java's "`private`"), we can use the introspective capabilities of the language to manufacture a locally-visible *get* method. To access a `private` field x, for example, our implementation generates a γ_x that handles the unwieldy details of Java introspection:

```
private τ _getX() {
  Class klass = this.getClass();   Field field = null;
  while (field == null) {
    try {
      field = klass.getDeclaredField("x");   field.setAccessible(true);
    } catch (NoSuchFieldException e) {
      klass = klass.getSuperclass();
    }
  }
  τ x = null;
  try {
    x = (τ) field.get(this);
  } catch (IllegalAccessException e) {   e.printStackTrace();   throw new Error();   }
  return x;
}
```

Other extensions, such as the inclusion of anonymous inner classes, concurrency, or *final* classes/methods, remain as open challenges.

Finally, the work described here incorporates only the invariant checks, rather than full contracts, and it would clearly be useful to extend our design pattern to support this. While we conjecture that our technique is easily extendable to this purpose, the invariant checks present the most interesting problems, owing to their need for attribute access and hierarchical definition. Philosophically, ordinary unit testing already performs at least the behavioral components of contract checking, *i.e.* the checks of pre and post-conditions. What unit testing cannot do is determine whether the invariant continues to hold, as it is often impossible to access an object's fields. The difference lies in the fact that both pre and post conditions are inherently extensional specifications. They impose requirements on method arguments and return values, but on the object itself, all constraints are made upon the abstraction of the object, not the concrete implementation. That implementation— whose consistency with the abstraction is the core assertion of a class invariant—is by definition opaque to an object's user.

Acknowledgments. The ideas in this paper began with an assignment in the first author's Spring 2010 Software Construction class, and the students there provided valuable feedback. Our thanks also to Prof. Peter Boothe of Manhattan College, for help in analyzing the inheritance and method complexity of the JDK.

References

1. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM TOPLAS 16(6), 1811–1841 (1994)
2. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley (2001)
3. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall (1997)
4. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 2002, pp. 234–245. ACM Press (2002)
5. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
6. Liu, H., Qin, L., Wang, J., Vemuri, N., Jia, X.: Static and dynamic contract verifiers for Java. In: Hamza, M.H. (ed.) SEA 2003, pp. 593–598. ACTA Press (2003)
7. Karaorman, M., Hölze, U., Bruno, J.: jContractor: A Java library to support design by contract. Technical Report TRCS98-31, University of California at Santa Barbara (1998)
8. Prasetya, W., Vos, T., Baars, A.: Trace-based reflexive testing of OO programs. Technical Report UU-CS-2007-037, Dept. of Information and Computing Sciences, Utrecht University (2007)
9. Prasetya, W., Vos, T., Baars, A.: Trace-based reflexive testing of OO programs with T2. In: STVV 2008, pp. 151–160. IEEE Press (2008)
10. Java Modeling Language (JML) Homepage, `http://www.eecs.ucf.edu/~leavens/JML/`

11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
12. The AspectJ programming guide, `http://www.eclipse.org/aspectj/doc/released/progguide/`
13. Briand, L.C., Dzidek, W., Labiche, Y.: Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In: ICSM 2005. IEEE Press (2005)
14. Skotiniotis, T., Lorenz, D.H.: Cona: Aspects for contracts and contracts for aspects. In: Vlissides, J.M., Schmidt, D.C. (eds.) OOPSLA 2004 Companion, pp. 196–197. ACM Press (2004)
15. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
16. Balzer, S., Eugster, P.T., Meyer, B.: Can Aspects Implement Contracts? In: Guelfi, N., Savidis, A. (eds.) RISE 2005. LNCS, vol. 3943, pp. 145–157. Springer, Heidelberg (2006)
17. Agostinho, S., Moreira, A., Guerreiro, P.: Contracts for aspect-oriented design. In: SPLAT 2008. ACM Press, New York (2008)
18. Klaeren, H., Pulvermüller, E., Rashid, A., Speck, A.: Aspect Composition Applying the Design by Contract Principle. In: Butler, G., Jarzabek, S. (eds.) GCSE 2000. LNCS, vol. 2177, pp. 57–69. Springer, Heidelberg (2001)
19. Gibbs, T.H., Malloy, B.A., Power, J.F.: Automated validation of class invariants in C++ applications. In: ASE 2002, pp. 205–214. IEEE Pr. (September 2002)
20. Malloy, B.A., Power, J.F.: Exploiting design patterns to automate validation of class invariants. Software Testing, Verification and Reliability 16(2), 71–95 (2006)
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
22. JSON Homepage, `http://www.json.org/`
23. Lambda expressions for the Java programming language, `http://jcp.org/aboutJava/communityprocess/edr/jsr335/index2.html`
24. McConnell, S.: Code Complete: A Practical Handbook of Software Construction. Microsoft Press (2004)
25. Shatnawi, R.: A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. IEEE Transactions on Software Engineering 36(2), 216–225 (2010)
26. Briand, L.C., Dzidek, W., Labiche, Y.: Investigating the use of analysis contracts to improve the testability of object-oriented code. Software—Practice and Experience 33(7), 637–672 (2003)
27. Le Traon, Y., Baudry, B., Jézéquel, J.M.: Design by contract to improve software vigilance. IEEE Transactions on Software Engineering 32(6), 571–586 (2006)