# Percolation [*]

## Intent

Implement automatic checking of superclass assertions to support design-by-contract and Liskov Substitution Principle.

## Motivation

Assertions cannot be inherited in Ada 95, C++, Java, Objective-C, or Smalltalk. Only Eiffel supports assertion inheritance. For example, although we can make a C++ base class `invariant( )` function visible in a derived class, there is no built-in mechanism for checking base class assertions with the proper Boolean operators in a derived class. The Percolation pattern results in a straightforward structure that yields the same checking as built-in assertion inheritance.

Percolation is effective at revealing bugs that result from inheritance and dynamic binding. The likelihood of such bugs increases as class hierarchies are extended, patched, and reused. Some bugs that result from incorrect inheritance structures can be detected if base class assertions are checked in a derived class. Automatic checking of base class [**] assertions is only supported by Eiffel. Recoding base class assertions in derived classes would be time-consuming, error-prone, and inelegant.

The Percolation pattern allows base class invariants, preconditions, and postconditions to be checked in derived class functions without redundant code. It respects base class encapsulation by providing a fixed interface to base class assertions without direct access to base class instance variables. It allows loosely-coupled compile-time enable/disable of assertion checking.

## Applicability

Checking base class assertions in a derived class is useful when the derived class depends on correct and consistent use of base class features. This is especially important in hierarchies that export polymorphic functions. The percolation pattern provides effective built-in test in any class hierarchy designed to conform with design-by-contract [Meyer 97] or the LSP [Liskov+94] [Martin 96]. A class hierarchy that exports polymorphic functions and does not follow the LSP is almost certainly buggy. In hierarchies composed for convenience or naive reuse without regard for the LSP, the flattened assertions will probably be ambiguous or inconsistent. Percolation will not be useful in this case.

## Structure

Figure 1 shows the basic structure of percolation in C++. The figure shows three levels, but the pattern extends to any number of levels and multiple inheritance.

---

[*] Appeared as "The Percolation Pattern," *C++ Report*, 11( 5), May 1999. Excerpted with corrections from Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools* (Addison-Wesley, 1999). 

[**] C++ terms are used since the example is C++. The pattern applies all other languages however.

## Participants

The participants in this pattern are protected functions that implement the assertions for all preconditions, postconditions, and invariants in each class in a hierarchy.


## Collaboration

The scope of the percolation pattern is a class hierarchy. Classes outside the hierarchy need not be considered. When a class invariant is checked, all of its base class invariants should also be checked. Similarly, when the precondition or postcondition of an overridden function is checked, all of its base class precondition and postconditions should also be checked. Checking begins at the lowest level (a derived class) and works up the hierarchy, suggesting a percolation action.

The pattern requires that base class assertions are implemented as functions visible to derived class assertion functions. Every derived class assertion function calls its corresponding assertion function in its immediate base class. This results in the assertion checking being percolated, one level at a time, until a root class assertion is reached.

The requirements for base class checking can be made explicit by considering flattened assertions. A flattened assertion is obtained by using *and* or *or* to concatenate all corresponding assertions in the hierarchy. Flattened assertions can be described with the following syntax.

```
<derived.pre>     // Derived class local precondition
<flat.pre>        // Flattened precondition
<base.pre>        // Base class precondition
<derived.post>    // Derived class local postcondition
<flat.post>       // Flattened precondition
<base.post>       // Base class postcondition
<derived.inv>     // Derived class invariant
<base.inv>        // Base class invariant
<flat.inv>        // Flattened precondition
<a> ::= <b>       // <a> is composed of <b>
<op><assert>^     // The concatenation of all
                  corresponding base class assertions,
                  prefixed by op, || (or) or && (and).
```

Parentheses denote order of evaluation. A flattened invariant can implement the lower/stronger invariant rule in four ways.

```
<flat.inv> ::= <derived.inv> && <base.inv>^
   // There are non-trivial invariants for derived and
      base classes

<flat.inv> ::= TRUE &&<base.inv>^
   // The Derived class invariant is trivial

<flat.inv> ::= <derived.inv> && TRUE
   // All base class invariants are trivial

<flat.inv> ::= <derived.inv>
   // There is no base class
```

The same principle holds under multiple inheritance. For example, suppose invariants are contributed to a derived class by two base classes `BaseA` and `BaseB`. The flattened invariant is obtained by *anding* all base class invariants.

```
<flat.inv> ::= <derived.inv> && <baseA.inv>^ && <baseB.inv>^
```

Percolation for preconditions and postconditions depends on how inheritance has been used to compose the function. There are three cases:

- The highest level definition of a function (specialization.)
- The automatic inclusion of base class function in a derived class (extension.)
- The redefinition of a base class function in a derived class (overriding.)

*Percolation for Defining/Specializing Methods.* The first level in which a function is defined requires evaluation of its preconditions, postconditions, and its flattened invariant.

```
<flat.pre> ::= <derived.pre> && <flat.inv>
   // There is no base class precondition to inherit

<flat.post> ::= <derived.post> && <flat.inv>
   // There is no base class postcondition to inherit.
```

*Percolation for Extended Methods.* A base class function made available by inheritance must conform to its preconditions, postconditions, and the *derived class* flattened invariant. Although the function is inherited, it would be unacceptable for it to create a corrupt derived class state. This is accomplished by evaluating the flattened invariant of the derived class (which includes the base class' invariant.)

```
<flat.pre> ::= <base.pre> && <flat.inv>
   // The flattened base class precondition is inherited and
   // the flattened derived class invariant must hold

<flat.post> ::= <base.post> && <flat.inv>
   // The flattened base class postcondition is inherited and
   // the flattened derived class invariant must hold
```

*Percolation for Overridden Methods*. The flattened invariant must be *anded* with preconditions and postconditions for a base class function which is redefined in a derived class. Clearly, the derived class functions must conform to the flattened invariant. There are three cases for a flattened precondition of any overriding derived class function implementation. They implement the lower/weaker precondition rule.

```
<flat.pre> ::= (<derived.pre> ||<base.pre>^) && <flat.inv>
   // There are preconditions in derived and base classes

<flat.pre> ::= (TRUE ||<base.pre>^) && <flat.inv>
   // The derived class precondition is trivial

<flat.pre> ::= (<derived.pre> || TRUE) && <flat.inv>
   // The base class preconditions are trivial, or
   // there is no base class.
```

There are three cases for the flattened postcondition of an overriding derived class function implementation. They implement the lower/stronger postcondition rule.

```
<flat.post> ::= (<derived.post> && <base.post>^) && <flat.inv>
   // There are postconditions in derived and base classes

<flat.post> ::= (TRUE && <base.post>^) && <flat.inv>
   // The derived class postcondition is trivial

<flat.post> ::= (<derived.post> && TRUE)   && <flat.inv>
   // The base class postcondition are trivial
```

## Consequences

The favorable consequences of percolation include:

- Improved readability. Assertions coded in a function body can decrease readability. Percolation reduces code clutter since only four additional statements appear in each function body (call to the precondition, postcondition, and to the invariant at entry and exit.)

- Reduced time to gain confidence in derived class extensions and modifications by testing, since all of the base class' built-in test is automatically activated by testing the derived class.

- The general benefits of built-in test and design by contract.

The costs and limitations include:

- The increased effort to design, implement, and maintain this code.

- The size of the executable and run time will increase when assertions are enabled.

- The percolation pattern requires an understanding of the LSP and design-by-contract. Developers must apply these principles while coding the class hierarchy. Not only must the assertions represent a well-formed class contract, but the structure of the percolating assertions must conform to the syntax given above, for each general case of inheritance.

- The general limitations and costs of built-in test and design by contract.

The example implementation is unable to distinguish between self as client and external clients. Eiffel does, and suspends checking for self-messages. This is useful if, for example, a private function computes an intermediate result which is inconsistent with the invariant.

## Implementation

Percolation assertions must be hand-coded in most languages. Eiffel implements explicit support for percolation: base class assertions are automatically inherited and checked with derived class assertions. When a function (routine) is overridden (redeclared), preconditions must be given as `require else <predicate>`. This *ors* the derived class precondition with all corresponding base class preconditions. Similarly, postconditions in an overridden function must be stated by `ensure then <predicate>`, which *ands* the derived class postcondition with the corresponding base class postcondition. This applies for all levels of redeclaration.

The functions of each immediate base class must be visible to a derived class to implement percolation. However, default visibility is typically all or nothing. Care must be taken in coding the assertion functions to be sure that the immediate base class is the target of percolation.

The generic C++ example given here does not show how to exercise a finer degree of control over assertion actions. This is a simple programming problem and could be accomplished in many ways.

## Sample Code

A generic C++ implementation of Percolation follows. In the example, functions `foo` and `bar`are overridden, `fee` and `fum` are examples of the defining and extending cases. Placement of the assertions follows recommendations given earlier: invariant after constructor body, invariant and precondition before function body, invariant and postcondition after function body, invariant before the destructor body. Only a single constructor is shown -- the same structure would be implemented for all constructors. The basic elements of this implementation are:

- The standard `<assert.h>` is replaced with `<ASSERT.h>` to allow the assertion to identify itself.

- Each precondition, postcondition, and invariant is implemented as a protected member function. A protected member function makes the base class assertions visible to derived classes without exposing them to clients. These member functions are all type `bool` (returning *true* when the assertion holds, *false* when it is violated), `inline` (to increase performance and then allow the optimizer to remove the entire function when they are null), and `const` to insure there are no side effects. The assertion function body may be only implemented with `ASSERT` statements. When the assertion expansion is disabled, an empty inline function results. Empty inline functions are completely removed by the compiler's optimization.

- The class to which an `invariant( )` message is bound is assumed to be the class in which the message appears. This is necessary to evaluate the derived class' flattened invariant when a base class function is used by a derived class object.

- The application function must make two calls: first to the invariant function, then to the pre or post function. If the invariant call was packaged with the pre or post function, then base class invariants would be called twice (once when the invariant is percolated and a second time when the pre condition is percolated.)

```
// ASSERT -- macro+function to support extended C++ assertion checking
#include <iostream.h>
#ifndef NDEBUG
  #define ASSERT(ex,msg)  Assert(ex, msg, #ex, __FILE__, __LINE__)

   inline bool
   Assert( bool condition,
           const char * message,
           const char * expression,
           const char * filename,
           int linenumber )
   {
       if (! condition) {
           cout << flush;
           cerr << endl
                << message
                << expression
                << " In File " << filename
                << " at line " << linenumber
                << " violated. "
                << endl;
           exit(1);
       }
       return true;
   }

#else
  #define ASSERT(ex,msg) /* */
#endif


Class Base {
public:

    Base ( ) { /*ctor body */  invariant( );}

    virtual void foo( ) {
        invariant( );
        fooPre( );

        // foo body

        invariant( );
        fooPost( );
        return;
    };

    virtual void bar( ) {
        invariant( );
        barPre( );

        // bar body

        invariant( );
        barPost( );
        return;
    };

    virtual ~Base( ) {invariant( ); /*dtor body */ }

protected:
//  Base Class Assertions ***************************************//
    virtual inline bool invariant( ) const {
        ASSERT(    baseValue == requiredBaseValue, “Base Invariant”);
    };

    virtual inline bool fooPre( ) const {
        ASSERT(    fooValue == requiredfooValue>, “Base::foo Precondition”);
    };

    virtual inline bool fooPost( ) const {
        ASSERT(    fooValue == requiredfooValue, “Base::foo Postcondition”);
    };

    virtual inline bool barPre( ) const {
        ASSERT(    barValue == requiredbarValue, “Base::bar Precondition”);
    };

    virtual inline bool barPost( ) const {
        ASSERT(    barValue == requiredbarValue, “Base::bar Postcondition”);
    };
};
```

```
Class Derived1: public Base {
public:
    Derived1 ( ) {/*ctor body */ invariant( );}
    virtual void foo( )  {
        invariant( );
        fooPre( );
    // foo body
        invariant( );
        fooPost( );
        return;
    };
    virtual void bar( ) {
        invariant( );
        barPre( );
    // bar body
        invariant( );
        barPost( );
        return;
    };
    void fum( ) {          // Specialization -- new member function
        invariant( );
        fumPre( );
    // fum body
        invariant( );
        fumPost( );
        return;
    };
    virtual ~Derived1( ) {invariant( ); /*dtor body */ }

protected: // Derived1 Class Assertions ************************************//

    inline bool invariant( ) const {
        ASSERT(     (   d1_Value == required_d1_Value &&
                        Base::invariant( )
                    ),
                    "Derived1 Invariant"
        );
    };
    inline bool fooPre( )  const {
        ASSERT(     (   fooD1Value == requiredfooD1Value ||
                        Base::fooPre( )
                    ),
                    "Derived1::foo Precondition"
        );
    };

    inline bool fooPost( )  const {
        ASSERT(     (   fooD1Value == requiredfooD1Value ||
                        Base::fooPost( )
                    ),
                    "Derived1::foo Postcondition"
        );
    };

    inline bool barPre( )  const {
        ASSERT(     (   barD1Value == requiredbarD1Value ||
                        Base::barPre( )
                    ),
                    "Derived1::bar Precondition"
        );
    };

    inline bool barPost( )  const {
        ASSERT(     (   barD1Value == requiredbarD1Value ||
                        Base::barPost( )
                    ),
                    "Derived1::bar Postcondition"
        );
    };

    inline bool fumPre( )  const {
        // Superclass asserts arent appended to specialization
        ASSERT(     (   fumD1Value == requiredfumD1Value),
                    "Derived1::fum Precondition"
        );
    };
    inline bool fumPost( )  const {
        // Superclass asserts arent appended to specialization
        ASSERT(     (   fumD1Value == requiredfumD1Value),
                    "Derived1::fum Postcondition"
        );
    };
};
```

```
Class Derived2 : public Base {
public:
    Derived2 ( ) {/*ctor body */ invariant( );}
    virtual void foo( )  {
        invariant( );
        fooPre( );
        // foo body
        invariant( );
        fooPost( );
        return;
    };
    virtual void bar( )  {
        invariant( );
        barPre( );
        // bar body
        invariant( );
        barPost( );
        return;
    };
        void fee( )  {     // Specialization
        invariant( );
        feePre( );
        // fee body
        invariant( );
        feePost( );
        return;
    };
    virtual ~Derived2( ) {invariant( ); /*dtor body */ }

protected:  // Derived2 Class Assertions ****************************************//

    inline bool invariant( ) const {
        ASSERT(     (   d2_Value == required_d2_Value &&
                        Derived1::invariant( )
                    ),
                    "Derived2 Invariant"
        );
    };
    inline bool fooPre( )  const {
        ASSERT(     (   fooD2Value == requiredfooD2Value ||
                        Derived1::fooPre( )
                    ),
                    "Derived2::foo Precondition"
        );
    };
    inline bool fooPost( )  const {
        ASSERT(     (   fooD2Value == requiredfooD2Value &&
                        Derived1::fooPost( )
                    ),
                    "Derived2::foo Postcondition"
        );
    };
    inline bool barPre( )  const {
        ASSERT(     (   barD2Value == requiredbarD2Value ||
                        Derived1::barPre( )
                    ),
                    "Derived2::bar Precondition"
        );
    };
    inline bool barPost( )  const {
        ASSERT(     (   barD2Value == requiredbarD2Value &&
                        Derived1::barPost( )
                    ),
                    "Derived2::bar Postcondition"
        );
    }
    inline bool feePre( )  const {
        ASSERT(     (   feeD2Value == requiredfeeD2Value ||
                        Derived1::feePre( )
                    ),
                    "Derived1::fee Precondition"
        );
    };
    inline bool feePost( )  const {
        ASSERT(     (   feeD2Value == requiredfeeD2Value &&
                        Derived1::feePost( )
                    ),
                    "Derived2::fee Postcondition"
        );
    };
```

## Known Uses

Several research systems have been developed to explore automatic verification of type/subtype relationships. The Larch system for specification verification has been extended to C++ and Smalltalk [Cheon+94]. Sun et al. show how assertions can be used to express similar correctness constraints for superclasses and subclasses [Sun+95].

Eiffel implements automatic percolation checking [Meyer 97]. Percolation is supported in the C++ extensions proposed by Porat and Fertig [Porat+95]. Percolation is supported in the design-by-contract extensions to Smalltalk/V developed by Carrillo et al [Carrillo+96]. A weak form of invariant percolation is recommended practice for classes derived from Microsoft Foundation Classes: the message `Base::AssertValid( )` should be sent on entry to derived class member functions [Stout 97].

Percolation is a basic strategy in object-oriented programming. It is used for initialization Smalltalk and Objective-C: upon receiving a `new` message, the `super new` message is sent to the immediate base class, which in turn sends `super new` to its base class, an so on. Percolation is also used for dynamic binding in these languages. The search for a method begins in the class of the object which receives a message and continues up the class hierarchy until a method is found which matches the message selector.

A form of Percolation is used in the abstract data type verification techniques developed by Bernot [Bernot+91] and implemented in the LOFT system [Dauchy+93]. An abstract data type can be specified in terms of its own operations. These specification expressions can be recursively evaluated in a process called unfolding. The effect of unfolding a specification expression is much the same as the effect of the sequence of assertion functions that result when an assertion is percolated from a subclass to a superclass.

## Related Patterns

The Coherence idiom [Cline 95] provides C++ assertion checking, but does not offer a strategy for percolation.

## References

[Bernot+91] Giles Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal* 6(6): 387-405, November 1991.

[Carrillo 94] Manuela Carrillo-Castellón, Jesus Garcìa-Molina, and Ernesto Pimentel. Eiffel-like assertions and private methods in Smalltalk. In *International Conference onTechnology of Object-Oriented Languages and Systems (TOOLS 7, Europe'94)* Englewood Cliffs, New Jersey: Prentice Hall Inc., 1994.

[Cheon+94] Yoonsik Cheon and Gary T. Leavens. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology* 3(3):221-253, July 1994.

[Cline 95] Marshall P. Cline and Greg A. Lomow. *C++ FAQs.* Reading, Mass: Addison-Wesley Publishing Co., 1995.

[Dauchy+93] P. Dauchy, —C. Gaudel, and B. Marre. Using algebraic specifications in software testing: a case study on the software of an automatic subway. *Journal of Systems and Software* 21(3):229-244, June 1993.

[Liskov+94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811-1841. November 1994.

[Martin 96] Robert C. Martin. The Liskov substitution principle. *C++ Report*, 8(3):30-37, March 1996.

[Meyer 92b] Bertrand Meyer. *Eiffel: the language.* Second revised printing. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1992.

[Meyer 92] Bertrand Meyer. Applying 'design by contract'. *IEEE Computer* 25(10):40-51, October 1992.

[Meyer 97] Bertrand Meyer. *Object-oriented software construction.* Upper Saddle River, New Jersey: Prentice Hall PTR, 1997.

[Porat 95] Sara Porat and Paul Fertig. Class assertions in C++. *Journal of Object-Oriented Programming* 8(2): 30-37, May 1995.

[Stout 97] John W. Stout. Front-end bug smashing in C++ and MFC: pointer validation and tracing. *Visual C++ Developers Journal*. January 1997.

[Sun+95] W. Sun, Y. Ling, and C. Yu. Supporting inheritance using subclass assertions. *Information Systems* (Germany) 20(8):663-685, 1995.