Encapsulation in Object-Oriented Programming: Comparison & Evaluation

W. Al-Ahmad
University of Sharjah, Department of Management Information Systems
P.O.Box 27272 Sharjah
walahmad@sharjah.ac.be

Abstract
This paper discusses the concept of encapsulation from object-oriented programming viewpoint. In addition to briefly reviewing issues relevant to the concept, I present an overview of the support that major object-oriented languages such as C++ [1], Eiffel [2], Smalltalk [3],and Java [4] offer to accommodate it. This support is explained, critically compared, and then evaluated to assess its adequacy. Finally, I propose some heuristics and guidelines with respect to an adequate support for the concept.

1. Introduction

Given appropriate formalisms for specifying the object representation and the object behavior, how should the integration of the representation and behavior be achieved in a way that will ease the maintenance problem of software? Current object-oriented programming languages (OOPLs for short) adopt the principle of encapsulation or information hiding to achieve this goal. In general, encapsulation means the ability to hide the implementation behind a single interface. Implementation may refer to an operation, a class, a subsystem, a component, or a complete application. In object-oriented programming (OOP), the basic idea behind encapsulation is to hide a great deal of details that are irrelevant for the clients of the class. In OOP, the focus of encapsulation is on hiding the representation of objects from the potential clients of a class. In this way, changes to the representation of objects only influence the implementation of the corresponding class. Hence, modifications to a class do not affect the users of the class, which is a major advantage of the encapsulation principle. In fact, in their interaction with objects of a given class, clients will only use the specification of the messages understood by those objects. In this way, both the representation of the objects and the details concerning the actual processing of messages by the receiving objects are hidden from the clients.

This paper is structured as follows: section 2 briefly explains the principles underlying the notion of encapsulation. Section 3 illustrates the concepts offered by the languages C++, Eiffel, Smalltalk, and Java to support encapsulation. Section 4 discusses encapsulation in the context of inheritance. Section 5 then evaluates the different approaches adopted by these languages and proposes an adequate support for encapsulation in OOPLs. Finally, section 6 introduces some recommendations and conclusions.

2. Techniques of Encapsulation

This section introduces concepts relevant to encapsulation in OOP. Actually, these concepts can be considered as a framework for encapsulation in OOP.

2.1. Specification versus Implementation

In OOP, the notion of encapsulation distinguishes between the specification of a class and its implementation. The specification of a class, or class interface, focuses on the services or messages the class has to offer to its clients. Typically, the specification of a message in the class interface consists of its signature and an informal description of its effect. The implementation of a class focuses on how the class will serve its clients. Basically the implementation introduces a proper method for responding to each of the messages offered by a class. It also includes the internal representation of the objects of the class. It is often said that the specification of a class focuses on the "what", whereas the implementation of a class focuses on the "how". Evidently, the specification of a class must somehow be related to its implementation. One way is to verify whether each message specified in the class interface is complemented with a proper method in the implementation of the class. There is a consensus within the object-oriented community about the need to separate the specification of a class from its implementation. This is the first requirement of encapsulation in OOP.

2.2. Assertions

The specification of a class plays the role of a contract. Sine contracts need to be clear and concise, it is useful to use the concept of programming by contract introduced by Meyer [2] to specify them. The use of assertions should be the second requirement of encapsulation in OOP. Here are some of the techniques used in the programming by contract approach.

2.2.1. Preconditions

Whenever a client of a class needs to send a message to some object, and the method used to implement that message is hidden, the client must somehow be informed of the information to supply. Evidently, part of that information is available in the signature of the message. In an untyped language like Smalltalk, the signature only informs about the number of arguments to be supplied. In typed languages like C++, Eiffel, and Java, the user is also informed about the types of arguments to be supplied. However, even in a typed language, the clients of a class need more information concerning the arguments. As an example, although the signature of a message may indicate an integer number for a particular argument, a client may be further restricted to supply only positive values for that argument.

In general, the specification of a message should include a section in which all conditions are specified that must be fulfilled when applying the message. These conditions are commonly referred to as preconditions; the section in which they are specified is typically called the require-clause in the specification.

2.2.2. Postconditions

Another equally important part in the specification of a message serves to describe the effect of applying the message. Postconditions establish assertions to be guaranteed upon completion of a method. Postconditions establish rights for the clients of a class, and at the same time duties for the implementers of a class. In particular, a client of a class, sending a message to one of its objects, is guaranteed the entire effect stated in the postconditions, provided all the preconditions of the message have been respected.

In general, the specification of a message should include a section in which all postconditions are specified. Such a section is typically called the effect-clause in the specification.

2.2.3. Class Invariants

Preconditions and postconditions specify semantics of individual messages offered by a class. Moreover, the objects of a class will be subjected to a number of general restrictions that apply to each of them at all stable times. These restrictions are commonly referred to as class invariants. The invariants of a class are specified in a general clause, which is part of the overall description of the class itself. Class invariants imply rights and duties for the clients and the implementers of the class. The implementer must guarantee that upon completion all the objects involved in the message satisfy their invariants. In this way, clients of a class can immediately forward other messages to these objects without a need to check whether they satisfy their invariants.

2.5. Access Rights

A class usually has a number of potential clients. In the first generations of OOPLs, all these clients were treated in a uniform way. In these languages, each client is granted access to the same set of messages offered by a class. Contemporary OOPLs tend to distinguish between the possible clients of a class. Some privileged clients are granted access to a more extended set of messages and maybe to the representation of the objects. Some less privileged clients are granted access to only a restricted portion of the messages offered by a class and maybe no access to the representation of the objects. In its most general form, this leads to formalisms in which the developer of a class can state which messages offered by a given class are available in methods associated with messages of other classes. None of the languages covered in section 3 offers such a general formalism. The specification of access rights is another important requirement of encapsulation in OOP. There is a consensus within the object-oriented community about restricting access to the object representation (attributes of a class).

3. Practice of Encapsulation

This section discusses in detail the concepts for supporting encapsulation in each of the OOPLs C++, Eiffel, Smalltalk, and Java in light of the concepts introduced in the previous section. Listing 1 shows a partial specification for the class of persons. Notice that only aspects related to the marital status of persons have been included. The specification is given in Java language. The pre/postconditions and class invariants will not be repeated in the other code listings.

```
/** A class of persons involving a spouse relationship.
  * invariants          Two married persons must be each other's spouse.
                        this.getSpouse().getSpouse() = this                  */
class PERSON {
    /** Register a marriage between this person and partner.
      * require     Effective Partner: partner ≠ 0
      *             Different Persons: partner ≠ this
      *             Unmarried Persons: (not IsMarried()) and (not partner→IsMarried())
      * effect      This person and partner become each other's spouse.
      *             (getSpouse() = partner) and (partner→getSpouse() = this)        */
  public void Marry (PERSON partner);

    /** Register a divorce between this person and its spouse.
      * require     Married Person: IsMarried()
      * effect      This person and its spouse are no longer married.
      *             (not IsMarried()) and (not old→getSpouse()→IsMarried())         */
  public void Divorce ( );

    /** Return a pointer to the spouse of this person.
      * result      The spouse of this person if married, a null pointer otherwise.  */
  public PERSON getSpouse ( );

    /** Check whether this person is married.
      * result      True if this person is married, false otherwise.
      *             getSpouse() ≠ 0                                                   */
  public boolean IsMarried ( );

    /** Register partner as the spouse of this person.
      * effect      Partner becomes the spouse of this person (partner may be null).
      *             getSpouse() = partner                                             */
  private void SetSpouse (PERSON partner);
}
```

Listing 1: Class Specification with Pre/Postconditions and class invariants.

Notice that preconditions, postconditions and class invariants have been worked out both informally and formally through the primitive inspector `getSpouse`. The formal specifications may be left out, sticking to informal descriptions of the semantics of the messages offered by a class. Evidently, one may prefer to use only informal descriptions of assertions.

3.1 Encapsulation in C++

In C++, the development of the class interface is separated from the implementation of the class. Yet, some aspects related to the representation of objects must be provided in the definition of the class. The implementation of each of the member functions offered by a class is principally deferred to the implementation of the class. Only for so-called inline functions, the C++ programmer is offered the choice whether their implementation is directly available in the definition of the class or in its implementation. In specifying access rights, the language basically distinguishes between a public section in the definition of a class and a private section. The public section introduces aspects available to all the clients of a class; the private section introduces aspects that are in principle only available to the implementers of the class. However, the notion of friends of a class is introduced to grant certain classes or certain member functions access to the private section of a class.

The definition of a class typically includes the signature of the member functions, and the representation of the characteristics to be retained for each of its objects. Encapsulation is primarily obtained through a separation of the class definition in a public interface and a private interface . Typically, the public interface will contain a specification of all the member functions applicable to the objects of the class. The private interface will then be restricted to a specification of the representation of the objects, together with a specification of a number of auxiliary functions. The basic aspects of the concepts offered by C++ in developing the definition of a class are illustrated in listing 2. The implementation, usually in separate file, is not shown.

```
class PERSON {
public:
    void Marry (PERSON *partner);
    void Divorce ( );
    PERSON* getSpouse ( ) const;
    bool IsMarried ( ) const;
private:
    void SetSpouse (PERSON *partner);
    PERSON *spouse;
}
```

Listing 2: Class Specification in C++.

The private part defines the data member for referring to the spouse of a person. This data member is only accessible to the implementer of the class, and its friends, as will be explained later. Similarly, the `SetSpouse` function can only be used in implementing more complicated functions offered by the class of persons. Indeed, contrary to the public functions `Marry` and `Divorce`, this function is considered unsafe because it only registers the relationship for one of the parties involved. As long as the link is not established from the other side, the application will be in an inconsistent state. This explains why the auxiliary function is part of the private interface of the class, thereby prohibiting ordinary clients of the class from invoking it.

In some cases, restricting access to only the public data members may hamper an efficient and elegant coding of other member functions involved in an application. For that purpose, C++ introduces the notion of friends of a class. Two levels of friendship must be distinguished at this point:

- Any function, including member functions of a given class `A`, can be granted access to the private interface of another class `B` by specifying this function as a friend of class `B`.
- Whenever a class `A` has a tight relationship with another class `B`, the entire class `A` can be qualified as a friend of class `B`. In that case, the implementer of class `A` is granted complete access to the private interface of the friend class `B`.

The notion of friends of a class is illustrated in listing 3 using the relationship between the class of cars and the class of persons. Notice, the relationship involving cars and their owners is assumed to be bi-directional. In this way, the example is similar to the bi-directional spouse-relationship involving persons, except for the fact that the relationship now involves objects of different classes.

```
class PERSON {
public:
    void Buy (CAR *car) {SetCar(car); car→SetOwner(this);}
    void Sell ( ) {ownedCar→SetOwner(0); SetOwner(0);}
    CAR* getCar ( ) const {return ownedCar;     }
private:
    void SetCar (CAR *car) {ownedCar = car;     }
    CAR *ownedCar;
}

class CAR {
    friend void PERSON::Buy (CAR *car);
    friend void PERSON::Sell ( );
public:
    PERSON* getOwner ( ) const {return owner;}
private:
    void SetOwner (PERSON *person) {     owner = person;}
    PERSON *owner;
}
```

Listing 3: Function Friends in C++.

The public member functions `Buy` and `Sell` are declared friends of the class of cars. Consequently these member functions have access to the auxiliary member function `SetOwner` and to the data member `owner` introduced in the private interface of the class of cars. Instead of declaring individual member functions of the class of cars as friends of the class of persons, the programmer can declare the entire class of cars a friend of the class of persons.

3.2. Encapsulation Eiffel

In Eiffel, the specification and the implementation of the operations applicable to the objects of a given class are developed together. Eiffel offers a tool, called `short`, which extracts the interface of a given class from its entire description. Furthermore, it is worth mentioning that Eiffel is one of the few programming languages ever developed, in which an attempt is made to specify the semantics of operations using preconditions, postconditions and class invariants. Unfortunately, the formalism is not powerful enough for expressing the more important properties of class interfaces being developed. Typically, specifications in Eiffel restrict themselves to the more obvious semantics of the operations involved.

Encapsulation in Eiffel is obtained by supplying additional information in the class interface. In specifying features offered by a class, one will designate the classes that may access those features. Basically, the list of classes that is granted access can be specified in three different ways. First, a given set of features can be exported to all possible classes using the `{ANY}` construct. Second, a set of features can be hidden from all possible classes using the `{NONE}` construct. Finally, access to a set of features can be restricted to a dedicated set of classes by listing their names between braces `{` and `}`. This mechanism is referred to as selective export: it resembles to some extent the notion of friends in C++. The basic aspects of the concepts offered by Eiffel for developing class interfaces are illustrated in listing 4.

```
class PERSON feature
    Marry (partner: PERSON) is
        require   Effective Partner: partner ≠ null;
                  Different Persons: partner ≠ Current;
                  Unmarried Persons: (not IsMarried)    -- and (not partner.IsMarried)
        do        SetSpouse(partner); partner.SetSpouse(Current)
        ensure    (spouse = partner) and (partner.spouse = Current)
        end; -- Marry
    Divorce is
        require   Married Person: IsMarried
        do        spouse.SetSpouse(Void);         SetSpouse(Void)
        ensure    (not IsMarried) –- and not (old spouse).IsMarried;
        end; -- Divorce
    spouse : PERSON;
    IsMarried : BOOLEAN is
        do        Result := spouse /= Void
        ensure    Result = (spouse /= Void)
        end; -- IsMarried
feature { }
    SetSpouse (partner: PERSON) is
        do        spouse := partner
        ensure    spouse = partner
        end; -- SetSpouse
-- invariant          IsMarried and then Current = spouse.spouse
end -- PERSON
```

Listing 4: Class Specification in Eiffel.

The first feature-clause does not explicitly list a set of classes to which its features are exported. These features are available to all potential clients of the class. An equivalent way of obtaining this kind of export status is to add `{ANY}` after the feature-clause. The feature `SetSpouse` is not exported to any other class; it can only be used within the implementation of the class of persons itself. An even more restrictive form of exporting features consists in exporting them to no class at all, not even to the class being defined. This can be achieved by adding `{NONE}` after the feature-clause. In this case, the features can only be applied to the current object and not to any other object of the given class. The feature `SetSpouse` has to be exported to the class of persons. Otherwise, it would be impossible to apply it to the partner of the person having received a message to marry, respectively to divorce. Finally, notice that exporting an attribute such as `spouse` corresponds to offering a function by means of which the current value of the associated characteristic can be retrieved. Consequently, assignment to the attributes of a class can only be performed within the class itself (read-only attribute). The selective export mechanism is illustrated in listing 5 in the context of the relationship between the class of persons and the class of cars. For reasons of simplicity, we assume that a person cannot own more than one car at the same time.

```
class PERSON feature
    Buy (theCar: CAR) is      ...     end;
    Sell is         ...     end; -- Sell
    car : CAR;
feature { PERSON, CAR }
    SetCar (theCar: CAR) is  ...     end;
end -- PERSON

class CAR feature
    owner : PERSON;
feature { CAR, PERSON }
    SetOwner (person: PERSON) is  ...     end;
 end -- CAR
```

Listing 5: Selective Export in Eiffel.

Here, access to the procedure `SetCar` is restricted to the class of cars and to the class of persons itself. These restrictions are imposed because these features may leave the system in an unsafe state, that is, the underlying relationship is only accomplished from one side. In the same way, the specification of the class of cars delimits access to the procedure `SetOwner` to the class of persons and to the class of cars. Because the features for linking cars to their owners, and vice versa, cannot be hidden from all possible clients of the classes involved, one cannot formulate a class invariant that requires the underlying relationship to be registered in both directions at all stable times. Indeed, a successful call to the exported feature `SetOwner` from within the class of cars, for example, would require both the postcondition of that routine and the class invariant to be satisfied. Clearly, the latter would not be satisfied because the relationship is only registered in one direction. This explains why the entire invariant-clause has been included as a comment.

3.3. Encapsulation in Smalltalk

In Smalltalk, the specification of the messages understood by the objects of a given class is not developed separate from their implementation. The environment offers a tool which extracts the specification of a class, referred to as the class protocol, from its definition. The protocol includes a specification of all the messages applicable to the objects of the given class. Because Smalltalk does not include any concepts for specifying the semantics of operations, the class protocol enumerates the signatures of the messages it offers. The support for encapsulation in Smalltalk is rather primitive. By definition, all the instance variables of a given class are hidden from all its clients. On the other hand, all the messages introduced in the class definition are available to all potential clients. The concepts offered by Smalltalk for developing class definitions are illustrated in listing 6.

```
PERSON class
InstanceVariableNames:
    ' spouse '

Instance Methods
    Marry: partner
        self SetSpouse: partner.
        partner SetSpouse: self
    Divorce
        spouse SetSpouse(nil).
        self SetSpouse(nil)
    IsMarried
        ↑(spouse notNil)
    GetSpouse
        ↑spouse
    SetSpouse: partner
        spouse := partner
```

Listing 6: Class Specification in Smalltalk.

Notice that it is impossible to restrict the scope of the primitive message `SetSpouse` to the class itself, such that only the implementer of the class can use them. In this way, ordinary clients of the class can easily violate the invariant stating that the spouse relationship must be registered in both directions.
Finally, notice that none of the instance variables of the class of persons is accessible to the clients of the class. In fact, only the instance variables of the person receiving a message (`self`) are available. In the method for responding to the message `Marry`, for example, it is impossible to access the instance variables of the additional argument `partner`. Again, this is strongly related to the policy of untyped variables in Smalltalk. From the signature of the message, one cannot conclude that `partner` will refer an object of the class of persons. Consequently, it would be unsafe to allow access to some of its instance variables. This immediately explains the need for the auxiliary message `SetSpouse` in the definition of the class of persons.

3.4. Encapsulation in Java

In Java, the specification of the class, or class interface, is integrated with its implementation in a single file. A typical programming environment for Java will offer a tool, called `javadoc`, to extract the interface of a class from its entire definition (see listing 1). In specifying access rights, the language offers the qualifiers `public`, `private` and `protected`. A public method or instance variable can be accessed by any client of the class; a private method or instance variable can only be accessed within the definition of the class itself. The meaning of unqualified methods and instance variables is explained in the context of the grouping of classes into packages. Finally, the meaning of protected methods and instance variables is related to inheritance. As for most OOPLs, Java does not offer any concepts supporting the specification of the methods applicable to the objects of a class. Principally, the interface of a class is restricted to a specification of the signature of the methods it offers.

The definition of each class in Java is principally stored in a separate file. The language offers the ability to group several classes into a package. Whereas the definition of a class corresponds to a file, a package will correspond to a directory in which the definitions of each of its classes are stored. Basically strongly related classes should be grouped into packages. As an example, one might introduce a package in which all classes representing the business logic of a banking system are grouped. This package would then include classes such as bank accounts, savings accounts and bankcards. Classes residing in the same package can be given some privileges in accessing instance variables and methods of other classes in that package. Such variables and methods will not be qualified public, private or protected.

4. Encapsulation and Inheritance

The previous section reviewed support for encapsulation in OOPLs in isolation from inheritance. Inheritance imposes its own requirements for encapsulation. This section briefly introduces such requirements and the support our OOPLs provide for it. No code examples are given due to the limits in space.

4.1. The C++ Approach

C++ offers three different types of derivations for controlling access to the member functions and data members inherited from the base class. In public derivation, both public and protected members of the base class are inherited to become public and protected members in the derived class, respectively. Private members are inaccessible at the level the derived class. In protected derivation, both the public members of the base class and its protected members are inherited to become protected members in the derived class. Consequently, ordinary clients of the derived class have no access to any of these members. In private derivation both the public and protected members of the base class members are inherited to become private members in the derived class. Consequently, neither ordinary clients of the derived class, nor classes subsequently derived from it have access to any of these members. Private derivation is default in C++.

A derived class is never granted access to the private interface of its base classes. In general, a derived class is only granted access to member functions and data members introduced in the public or protected section of its base class. It is already stated that direct access to the representation of the objects of a class must be restricted as much as possible. In view of this general principle, a base class should never grant derived classes direct access to its data members. In C++, this can be realized in an elegant way by introducing all the data members of a class in its private section.

4.2. The Eiffel Approach

An heir class inherits all the features offered by its parent class, regardless of whether or not they are exported by the parent class. Consequently, the software engineer responsible for the implementation of an heir class has direct access to the representation established at the level of the parent classes. As a consequence, instead of using primitive procedures for setting attributes introduced at the level of a parent class, it is possible to assign to the underlying variables directly in the implementation of routines at the level of heir classes. However, it is commonly accepted that the representation of objects of a class must be hidden as much as possible. It is definitely not wise to offer subclasses access to the representation established at the level of its superclasses.
Eiffel supports the concept of descendant hiding which is the ability for an heir class to hide a feature exported by one of its parents. The use of descendant hiding is problematic when combined with polymorphism, though. In this case the Liskov substitution principle is violated [5].

4.3. The SmallTalk Approach

A subclass in Smalltalk not only inherits all the messages understood by its superclass, the instance variables of the superclass are also inherited and they are directly accessible to the software engineer implementing the subclass. The rule of encapsulation easily generalizes towards inheritance: a client of a subclass has access to all the messages understood by the subclass itself, by the superclass of the subclass, by the superclass of the superclass of the subclass, etc. In other words, the public interface (messages) of the direct and indirect superclasses are inherited to become part of the public interface of the subclass; the private interface (instance variables) of the direct and indirect superclasses are inherited to become part of the private interface of the subclass.

4.4. The Java Approach

A subclass always has access to the public and protected instance methods and instance variables introduced by its superclass. In Java, as in C++, the principle of encapsulation can be realized in terms of private and protected instance variables and methods. In Java, a subclass cannot change the access right inherited from the superclass, i.e., the subclass cannot change a protected instance method to a public one. The access right of the inherited method or variable is preserved. A subclass can access a package access right of an inherited method only if the subclass resides in the same package as the superclass.

5. Evaluation of Object Encapsulation

Having explored the concepts offered by the OOPLs C++, Eiffel, Smalltalk, and Java to support encapsulation, I evaluate the notion of encapsulation as supported in these languages and propose an adequate support for encapsulation.

5.1. Interfaces versus Implementations

All current OOPLs somehow recognize the roles class interfaces and class implementations have in structuring a given application. Smalltalk, Eiffel and Java tend to integrate both descriptions. This approach is justified by the observation that implementing a given class is just providing additional information concerning its specification. The approach adopted in C++ more or less shifts the burden from integrating the interface of a class with its implementation towards the programmer. In particular, the programmer is forced to repeat the specification once more in developing the implementation of the class. Moreover, each time a class appeals to another class, the programmer is responsible for including the interface of that class. Managing header files in C++ is also a serious problem, especially in large end complex systems. Eiffel, Smalltalk and Java all impose the availability of a tool for extracting the specification of a class from its entire definition. In this respect, the approach adopted by these languages is clearly superior to that of C++. Therefore, an adequate support for encapsulation should allow the separation between class interface and class implementation. The class definition should combine the specification and implementation and a tool should be provided to extract the specification from the class definition.

5.2. Formal Versus Informal Specifications

Proper documentation of software systems has always been a serious problem. Most programmers experience major difficulties in producing meaningful comments concerning the software they develop. Often, documentation is therefore completely left out, resulting in software systems that are extremely difficult to maintain. One of the problems concerning the production of proper documentation concerns the lack of a proper notation which imposes some structure in the documentation to be produced. In this respect, the approach adopted by Eiffel is definitely a promising one. Regardless of the formalism, the documentation of a class must be structured in a number of sections, each section focusing on a particular aspect of the class or one of its components. In such formalisms, preconditions, postconditions and class invariants are crucial instruments. Additional sections may be considered in documenting the implementation of a class, such as loop variants and invariants, and representation invariants. The use of each of these elements in documenting classes in any object-oriented programming language is therefore recommended. In the current state of the art, a full formal specification of the messages understood by a class is still out of reach. Nevertheless, formal specifications can have a considerable influence on the reliability of software systems. First of all, it is relatively easy to verify aspects of formal specifications at run-time. In Eiffel, preconditions, postconditions and class invariants can be verified during the execution of a program. Needless to say that such a verification can reveal a great number of errors in an early stage of development. In the long run, one may hope for tools that are able to verify statically whether the specification of (parts of) a software system is consistent with its implementation. Therefore, an adequate support for encapsulation should allow the specification of a class formally using the programming by contract approach. However, even if the semantics of a class should be completely defined in a formal notation, informal descriptions can still be used along with the formal specifications for software engineers who were not trained to use formal specifications.

5.3. Partial Encapsulation

Some client classes indeed need privileges in accessing certain parts in the definition of other classes. A typical example is the case when two classes are related with bi-directional association. In Java privileges can only be granted to classes residing in the same package. This seems to introduce some conflicting design goals. Packages are principally introduced for grouping classes in a particular application domain. Privileged access to certain aspects of a class should not be tied to their grouping into packages. Indeed, this would imply that all classes implementing a bi-directional relation must reside in the same package. There is a general agreement that access to the representation of the objects of a class must not be exposed outside the scope of the class. In fact, it may be wise to restrict access to the internal representation to some of the more primitive messages offered by a class (accessor methods). More complicated messages will then be implemented in terms of these primitive messages. Therefore, encapsulation in OOPLs must answer the question which of the messages offered by a class can be used in the implementation of other messages. In this respect, the most flexible formalism enables the listing of all the messages that may use a particular message. Therefore, full support for encapsulation should completely encapsulate the attributes (representation) of the object and provide access methods to manipulate them.

5.4. Access Rights for Subclasses

In C++, access rights can only be strengthened using the notions of public derivations, protected derivations and private derivations. For example, an inherited public member can be redefined to become a protected or a private member in the derived class. The language further introduces the notion of exemption as a simple mechanism to preserve the inherited access right. In Eiffel, access rights can be changed at will. In particular, a feature hidden from the clients of a parent class can become accessible to the clients of an heir class, and a feature available to the clients of a parent class can be hidden from the clients of an heir class. In Smalltalk, the access rights concerning the messages inherited from the superclass cannot be changed at the level of the subclass. In particular, all the messages applicable to the objects of the superclass are equally applicable to the objects of the subclass. In Java, the access rights concerning the methods and instance variables inherited from a superclass cannot be changed. The need for hiding certain facilities from the clients of subclasses is merely due to the lack of expressiveness in performing specialization inheritance. Indeed, if software engineers can effectively specialize the behavior inherited from superclasses, the need for hiding some of the inherited messages from the clients of the specialized class would completely disappear. It is to be expected that the next generation of OOPLs will be equipped with more powerful formalisms for expressing relationships between messages of superclasses and subclasses. In literature, it widely accepted that access to the representation of the objects of a class must be restricted as much as possible. C++ and Java offer the ability to hide certain aspects in the definition of a superclass from all subclasses. It is rather remarkable to see that languages such as Smalltalk and Eiffel do not offer the ability to hide the representation of the objects of a superclass from its subclasses. Especially Eiffel is a language with a very strict encapsulation policy. Contrary to C++ and Java, attributes cannot be qualified public, in the sense to allow clients to modify them, in Eiffel or Smalltalk. Eiffel and Smalltalk are superior to Java and C++ in this regard. In view of this strong policy, one would have expected rules in these languages that would simply forbid a subclass to have direct access to attributes inherited from its superclass. Unfortunately, such a policy is not imposed in these languages.

A full support of encapsulation should allow the distinction between public and private interfaces of a class. The private interface should include the representation of the objects and some auxiliary methods. The distinction between subclasses and other classes in terms of access rights is not really necessary. There is no need for protected access rights. All clients can access superclass information via simple accessor methods. In terms of efficiency, this will cause a little bit of overhead comparable to dynamic dispatch of methods. I also do not see the benefits of the selective or friends access rights. The change of inherited access rights such as the descendant hiding mechanism should not be allowed.

5.5. Abstract Classes

Abstract classes are classes that contain common behavior which should be implemented at the level of subclasses. An abstract class must have at least one method without implementation. Abstract classes introduce another level of encapsulation. It does not only hide the implementation of so-called abstract methods, but also hide the different variations behind the abstract concept. In object-oriented software development, abstract classes play very important role and they proved to be very useful. C++, Eiffel, and Java fully support the notion of abstract classes. Smalltalk, on the other hand, does not provide any support for the concept. This is more or less in line with the philosophy of the language that, apart from some syntactical controls, all checking should be performed during execution of the given application. However, the notion of abstract classes can be simulated to some extent in the sense that one can postpone the implementation of certain messages to be understood by the instances of a given class to subclasses.

5.6. Adequate Support

The following table summarizes the features an object-oriented programming language should have to fully support encapsulation. The table also shows which of our selected languages provide support for the required features.

Feature / language support	**C++**	**Eiffel**	**SmallTalk**	**Java**
Separate specification & implementation	yes	yes	yes	yes
Formal specification of operations	no	partial	no	no
Private/public access rights	yes	partial	partial	yes
Abstract classes	yes	yes	no	yes
Accessor methods for manipulating attributes	no	partial	partial	no

6. Conclusions

This paper reviewed OOP support for a very important principle of software engineering, encapsulation. The support major OOPLs provide for this concept was compared and evaluated. While most OOPLs provide support for encapsulation, they differ in the way they do it and the level of encapsulation they provide. An OOPL must provide language constructs and mechanisms to restrict access to the representation of objects. There is no need to distinguish between different clients of a class regarding access rights to the features of the class. Abstract classes are useful and provide another level of encapsulation. Some formalism such as the contract paradigm is also required to specify the interface of a class. This will improve their robustness and reliability of class specifications.

References

[1] Stroustrup B., The C++ Programming Language, Addison-Wesley, Reading (Mass.), 1994.
[2] Meyer B., Object-Oriented Software Construction, 2nd Ed., Prentice-Hall, New Jersey, 1997.
[3] Goldberg A., Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading (Mass.), 1983.
[4] Flanagan D., Java in a Nutshell, 2nd Ed., O’Reilly & Associates, Cambridge, 1997.
[5] Alan Snyder, Encapsulation and Inheritance in Object-Oriented Programming Languages, Proceedings OOPSLA’86.