# Insight Software Consortium

# Policy on Backward Compatibility

## Draft



# Table of Contents



# 1. INTRODUCTION

## 1.1. Goal

Define a policy on backward compatibility to ensure that Insight Consortium Software:

1. Provides an effective level of backward compatibility to maintain its user base and also
2. Evolves to meet the needs of the image analysis community.

## 1.2. Scope

This policy covers software that is supported by the Insight Software Consortium. Some ISC supported software uses other software that this policy calls Third Party software. The policy only covers changes that affect application programming interface (API) backward compatibility. These API changes typically result in a compilation error. API changes do not usually cause changes performance or results produced by the software. Issues such as the addition of new classes, methods and member data will be addressed in another policy. This policy does not cover changes to the format and content of error,

warning and exception messages. This policy does not currently cover file format changes.

## 2. BACKGROUND

One of the major criticisms of open-source software is that new revisions are not compatible with old revisions. Breaking compatibility impedes the acceptance and utility of open-source software. On the other hand, strict backward compatibility polices can impede innovation in software. The tension between these two viewpoints is not easily resolved.

As projects mature and the customer base grows, backward compatibility becomes more important. Commercial hardware and software products call this customer base, the *installed base*. Commercial products usually have a know customer base consisting of those who have purchased or licensed the software. Also, commercial systems seldom expose internal API's. Open source projects rarely know the identities of their customers. And, since the source is open, customers have access to all public and protected classes, methods and data in the code. **For open source software, it is almost impossible to determine how the customer base is using the software.**

> *When a project hits a certain point in its life cycle, the unpleasant issue of backward compatibility begins to rear its ugly head. All of a sudden the changes introduced in a new release of the software have a dark side to them; they hold hidden possibilities that will break something one of your users depends on. This is true in both open and closed source projects, but in the open source world it seems that the community has spent less time worrying about it than in the closed source world. From "Preserving Backward Compatibility, http://www.onlamp.com/lpt/a/5626, Garrett Rooney.*

> *The Dark Side of Extreme Programming: The nightly test/build was so effective in empowering programmers to make changes, that API changes occurred too frequently without the necessary buy-in from the user community. From "Insight Insight", http://www.itk.org/CourseWare/Training/InsideInsight.pdf, Bill Lorensen*

Some argue that open source software should be used at your own risk. But even using open source software requires an investment in time, energy and funds. Also the reputation of the development community is at risk.

> *...consider your user base. If you have only a dozen highly technical users, jumping through hoops to maintain backward compatibility may be more trouble than it's worth. On the other hand, if you have hundreds or thousands of nontechnical users who cannot deal with manual upgrade steps, you need to spend a lot of time worrying about those kinds of issues. Otherwise, the first time you break compatibility you'll easily burn through all the goodwill you built up with your users by providing them with a useful program. It's remarkable how easily people forget the good things from a program as soon as they encounter the first real problem. From "Preserving Backward Compatibility, http://www.onlamp.com/lpt/a/5626, Garrett Rooney.*

These investments are made by customers that include developers, users and sponsors.

## 2.1. Definitions

### 2.1.1. Backward compatibility

From WikiPedia,

> In technology, especially computing, backward compatibility has several related but different meanings:
>
> - A *system* is backward compatible if it is compatible with earlier versions of itself, or sometimes earlier systems, especially systems it intends to supplant. That is, other systems or objects that interoperate with the old version of the system should continue to interoperate with the new version.
> - A program is backward compatible if it can share data with earlier versions of itself.
> - A library or platform is backward compatible if programs that interfaced with the old version continue to work with the new version as well.
> - Binary compatibility means that programs can work correctly with the new version of a library without requiring recompilation. Source compatibility requires recompilation but no changes to the source code.

### 2.1.2. Deprecated

There are many definitions of this term. Here are two slightly different definitions:

From http://whatis.techtarget.com:

> In dictionaries, deprecated is a term used to indicate a pronunciation or usage that is acknowledged but discouraged. In computer programming, a deprecated language entity is one that is tolerated or supported but not recommended.

From http://www.webopedia.com:

> Used typically in reference to a computer language to mean a command or statement in the language that is going to be made invalid or obsolete in future versions.

## 2.2. API Change Taxonomy

We define a taxonomy of API changes motivated by the taxonomy described in http://www.cs.ucsc.edu/~ejw/papers/kim-MSR2005.pdf. That paper looks at signature changes in eight open source C-language projects.

### 2.2.1. Class Name Addition

The addition of a new class is an API change that does not affect backward compatibility. Class name additions will be covered under another ISC policy.

### 2.2.2. Class Name Change

A class name change does affect backward compatibility.

### 2.2.3. Class Name Deletion

The removal of a class does affect backward compatibility.

#### 2.2.4. Method Name Addition

The addition of a new method does not affect backward compatibility. Method name additions will be covered under another ISC policy.

#### 2.2.5. Method Name Change

For private methods, change does not affect backward compatibility. For protected methods, change affects backward compatibility of derived classes. For public methods, change affects backward compatibility.

#### 2.2.6. Method Name Deletion

For private methods, deletion does not affect backward compatibility. For protected methods, deletion affects backward compatibility of derived classes. For public methods, change affects backward compatibility.

#### 2.2.7. Method Signature Change

A method's signature includes the type and order of its arguments and its return type. For private methods, a change in a method's signature does not affect backward compatibility. For protected methods, signature affects backward compatibility of derived classes. For public methods, signature affects backward compatibility.

#### 2.2.8. Method Access Modifier Change

Method access modifiers are public, protected or private. Moving a method from more restrictive to less restrictive access does not affect backward compatibility. Moving a method from less restrictive to more restrictive access does affect backward compatibility.

#### 2.2.9. Class Inheritance Change

Changing the inheritance of a class does affect backward compatibility.

#### 2.2.10. Member Data Addition

The addition of member data does not affect backward compatibility. Member data additions will be covered under another ISC policy.

#### 2.2.11. Member Data Change

Member data change includes the name, type and access modifier of the member data. For private member data, change does not affect backward compatibility. For protected member data, change affects backward compatibility of derived classes. For public member data, change affects backward compatibility. The change of the access modifier affects backward compatibility.

#### 2.2.12. Member Data Deletion

For private member data, deletion does not affect backward compatibility. For protected member data, deletion affects backward compatibility of derived classes. For public member data, deletion affects backward compatibility.

# 3. POLICY

The ISC will protect the users of ISC software from changes that affect backward compatibility. This section describes, for each change, the rationale for the policy and a recommended workaround to achieve backward compatibility. In general, once a class and its associated methods and data appear in an official release of the software, this policy places the burden of backward compatibility on the ISC developers and not the software users.

This policy discourages the use of a deprecated API, but tolerates the deprecated API. Whenever possible both compile time and run-time information will be used to announce the deprecated API.

## 3.1. General Principles

The ISC has a responsibility to ensure that released software conforms to software guidelines, respects intellectual property, compiles and runs on supported platforms. The ISC software process must support these principles.

It must always be difficult to change an existing API. Every change, no matter how small, must be questioned. The burden for change is on the ISC developers. The primary goal of this policy is to minimize API changes, but when necessary, those changes should never cause user code to fail to compile. Compilation errors are not able to report to a user how to correct the code in error. Documentation in user mailing lists or online forums like wiki's are not acceptable as the only venues for reporting how to achieve backward compatibility.

In general, API changes are only permitted if:

1. The compiler, if possible, can warn the user about the deprecated API. Some compilers show line numbers where the deprecated API is being used.
2. At run-time, deprecated API's report how to change code from the old API to the new API.
3. Documentation in the deprecated code clearly informs the user how to move the code from the old API to the new API.

## 3.2. Class Name Change

Once appearing in an official release, class name changes are not permitted.

*Rationale:* Changing the name of a class affects ISC software developers and outside developers of the software. The type of compilation error varies amongst compilers and these errors give no indication on how to fix the error. Name changes are sometimes needed because the name chosen by the developer does not meet the naming convention of the software. This sort of change must be detected before the software is released. Sometimes a name change is accompanied by other changes in the signature and behavior of a class. This change must be reviewed by the community.

*Workaround:* Create a derived class with the name of the old class. The derived class should be a subclass of the new class. This eliminates backward compatibility issues and minimizes maintenance.

*Issues:* Multiple names for the same object can be confusing to new users of the software. Additional burden is placed on the documentation.

*Deprecation :* At compile time, if possible, the compiler should announce that the class is deprecated. At run-time, the derived class should clearly announce in its constructor that the class has been replaced with the new class. The announcement should provide explicit instructions on how to change code to use the new class name. The source code should clearly describe how to use the new class.

### 3.3. Class Name Deletion

Once appearing in an official release, class name deletion is not permitted.

*Rationale:* User code that uses a deleted class will have a compilation error that gives no clues about how to repair the error. The ISC policy for introducing new classes defines a process for reviewing software before it is included in ISC software. This process will eliminate introduction of classes that should not be in the software.

*Workaround:* None.

*Issues:* When the ISC takes over a software package, the ISC may elect to remove some of the existing software. The removal of software will not be taken lightly and must undergo strict review.

Reasons for removal include:

- software that was never compiled or tested
- software that cannot be made portable
- software that violates intellectual property law
- software that does not have an ISC conforming license.

Sometimes during the process of refactoring, classes that are used internally may no longer be required. However, these classes may be used in user created classes and must remain in the software.

*Deprecation:* Classes that are removed should be placed in the DeprecatedCode directory. At compile time, if possible, the compiler should announce that the class is deprecated. The comments in the code should clearly describe why the class was deleted and how to find it in the DeprecatedCode directory. A header file containing empty methods should be created. At run-time, each method in this empty class should announce that the class has been deleted. Software in this directory should remain in the nightly build/test process. Building of deprecated software is optional for users.

### 3.4. Method Name Change

Once appearing in an official release, method name changes are not permitted.

*Rationale:* Changing the name of a method will cause a compilation error in user code that uses the old method name. The compiler has no way to notify the user how to use the new name. Released software has already been reviewed for inclusion in the software. The justification for method name changes is usually subjective and often arbitrary. If a method name is misspelled, that method can be removed after a deprecation waiting period.

*Workaround:* Create a method with the old name that invokes the new method.

*Issues:* Method names could be misspelled or may not conform to the software's naming conventions. Conformance to the software's naming conventions must be established before the first release containing the method. Misspelled names should be corrected.

*Deprecation:* At compile time, the compiler should warn, if possible, that the method is deprecated. At run-time, the method should clearly announce the name change and provide explicit instructions on how to change existing code that uses the old method. The source code for the deprecated method should describe how to use the new method and provide justification for changing the method name.

### 3.5. Method Name Deletion

Once appearing in an official release, protected and public methods cannot be deleted.

*Rationale:* Deleting a method name will cause a compilation error in user code. The compiler error can not give corrective action nor provide justification for the error. Consequently, methods cannot be deleted.

*Workaround: None.*

*Issues:*

*Deprecation:* At compile time, if possible, the compiler should warn that the method is deprecated. At run-time, the method should clearly announce that the method has been deleted and its use should be removed from user code. The source code should contain comments surrounding the deleted method that justifies its removal.

### 3.6. Method Signature Change

Once appearing in an official release, changes to a public or protected method 's signature are not permitted. Changes can be made to private methods since these are not accessible outside the class.

*Rationale:* Changes to a method's signature will cause compilation errors in user code that accesses the method. The compiler error cannot specifically contain corrective action.

*Workaround:* A new method should be added with the new signature. If only the method's parameters are changing, then both the old and new methods may be able to co-exist. If the return type of the new method is different from the return type of the old signature, then a new method name should be created. If the ISC supported compilers cannot compile the method with both signatures, then new signature must use a new method name.

*Issues:* None.

*Deprecation:* At compile time, if possible, the compiler should warn that the method with the old signature is deprecated. At run-time, the method should clearly announce that the method signature has been changed. The announcement should include a clear description on how to change the user code to avoid the deprecation announcement. The source code should contain comments surrounding the method that has the signature change. This code should justify the signature change.

### 3.7. Member Data Change

Once appearing in an official release, changes to protected and public member data is not permitted. Changing member data from protected or public to private is not allowed. Changing member data from public to protected or private is not allowed.

*Rationale:* Ideally, all member data in a class should be private. Access to the member data should always be through method calls. However, for efficiency and clarity,

sometimes member data that should be private is moved to the protected section of the class. This permits derived classes to access the member data directly. The ISC review process will restrict the access of member data to be private unless there is a compelling performance reason to open the access of the member data to derived classes (protected). The ISC recognizes that users may create classes that derive from existing ISC classes. Any change in the name for type of the member data will affect the backward compatibility of the user classes.

*Workaround:* None.

*Issues:* None.

*Deprecation:* None.

### 3.8. Member Data Deletion

Once appearing in an official release, protected and public member data deletion is not permitted.

*Rationale:* If protected or public data is removed, user code that uses that data will not compile. There is nothing in the compiler error message that can tell the user how to correct the problem. Ideally, all member data in a class should be private. Access to the member data should always be through methods. The ISC review process will restrict the access of member data to be private unless there is a compelling performance reason to open access to derived classes (protected). If the review process approves protected access to member data, then the ISC commits to never removing that member data. Member data should never have public access.

*Workaround:* None.

*Issues:* Keeping member data that is no longer used does cause some extra memory use in the class. Member data that is no longer used should be isolated in the header and marked with comments that the data is no longer used.

*Deprecation:* None.

## 4. BIBLIOGRAPHY