Interface Definition

Guidelines and Recommendations

Author:
Radovan Semančík

Date:
January 2010

Version:
1.0

Abstract:
This document provides guidelines and recommendations for interface definitions. It describes how the interface definitions should be formed, what usually needs to be included, how to create good definition and what to avoid. It provides some examples in Java programming language for illustration.

nLight, s.r.o.
Vendelínska 109
Lozorno, Slovakia
www.nLight.eu

Table of Contents

1 Interface Definition Guidelines....1
1.1 Names....1
1.2 Contract....2
1.3 Operations....3
1.4 Concurrency and Synchronism....3
1.5 Atomicity, Transactions and Consistency....4
1.6 Limitations....5
1.7 Error Conditions....6
1.8 Dependencies....7
1.9 Preconditions....8
1.10 Form....8
1.11 Notes....11
2 Interface Definition Checklist....12
3 Copyright....13

1 Interface Definition Guidelines

Surprisingly not many people can work well with abstractions. It probably needs a special kind of thinking, somehow twisted mind, to be able to do it well. However, abstractions are the daily bread for software architects and designers, it is the air we breathe and the water we drink. You may expect that only one or two members of project team can naturally think abstractly, but the rest of the team will be quite lost. Therefore I find it very helpful to provide some guidelines how to define abstractions. These will help people that are not natural abstract thinkers to create and maintain a reasonably good interface definition.

Interfaces are everywhere in well-designed system. Interfaces are formed from abstract classes and other abstractions that programming languages provide. Interfaces are formed from network protocol messages, signals, file formats, XML tags and other means. No matter what is the actual mechanics of the interface, some aspects are always the same. These aspects are summarized in following guidelines. Please note that these are guidelines, not rules and definitely not a dogma. Feel free to adopt as much or as little of these as you find suitable. Also, these guidelines describe interface **definition**, not interface design. Interface design is a separate, much more difficult, much less mechanical and often very intuitive activity.

1.1 Names

Names are extremely important in the project. That applies to concepts, component names and especially to interface names. Interfaces should be abstract, should survive for a long time. However it must be expected that interfaces will change, albeit slowly. However the names, once they spread through the project team, are even more difficult to change. Therefore the name should at least a bit more abstract that the interface itself. The designers and programmers will deal with many interfaces during their day-to-day work, interfaces that were designed by other people, interfaces that use concepts that may be not well know to the one using the interface. Therefore name should also provide an information of interface purpose at the first sight. This helps a lot to maintain understandability of the project.

Names in the interface definition should be unambiguous and precise. Do not shorten the names too much, try to be descriptive in interface naming. Also do not make the names too long or people will use acronym instead of the full name. If you cannot avoid using a long name, spend some time to figure out a name that makes nice, pronounceable acronym. Ideally the acronym should form a word that is close to the purpose of the interface. Also note that it may be lesser evil to use unusual names for components and interfaces instead of heavily overloaded names. Therefore it may be better to name interface “EvilEye Network Protocol” than to use “Management Network Protocol” which is too generic.

If your system consists of several platforms or languages, consider using a convention for interface names. For example use “Java API” suffix for consumer-oriented interface in Java, “C SPI” for provider-oriented interface in C programming language, “XML Schema” for XSD-based data format definition, etc.

Following table provides **good** examples of interface names:

Name	***Description***
Reactive Rule Engine Java API	Client-oriented interface (Application Programming Interface, API) in Java language for rule engine that is designed to react to generic events.
Environmental Sensor Driver C SPI	Provider-oriented interface (Service Provider Interface, SPI) in C programming language that allows to plug-in drivers for various environmental sensors.
EvilEye Network Protocol	Network protocol definition describing a simple network management protocol for monitoring system components.
Photo Description XML Schema	Definition of XML dialect designed to describe a photograph, containing textual description, identification of author, copyright, technical meta-data, etc.

Following table provides **bad** examples of interface names:

Name	*Why is it bad?*	*Recommendation*
Management Network Protocol	The name is too generic. “Management” is heavyly overloaded term. The protocol could manage the network components, it may be system administration protocol or even a protocol to automate team management. The audience will easily make false assumptions about such interface.	Use something less generic, such as “System Component Management Network Protocol”. If that is too generic as well, you may bet on exotic names such as “EvilEye Protocol”.
FXCTLNC API	Unless “FXCTLNC” is a well-established acronym in the project, such interface name will not provide any information to the audience. It is also difficult to remember and could be easily confused with FXCTTNC and FXTTLCN acronyms that may happen to be used in the project.	If you must use acronyms, try to make the acronym shorter. If possible make the acronym pronounceable, ideally something that forms a word close to the interface purpose. E.g. “WANG”, “XTREM” or “GLUE”.
Simple Pattern-Based Sequential Event-Processing Module for Perl	Interface name contains too much implementation concepts. The name should be more abstract than the interface itself.	Better names include “Reactive Rule Engine Perl Interface”, “Event-processing Rule Engine for Perl” or even “eRule for Perl” for short.

Also make sure you provide a separate **namespace** for you interface definition. Interfaces are often globally visible and it is important to avoid naming conflicts. The dedicated namespace may mean to reserve a dedicated package name for each interface (Java, Perl, …) or a unique prefix for macros, data structures and functions (C), creating a dedicated URL (XML, WSDL, …), allocating an object identifier (LDAP, SNMP, …), etc. Most practical platforms have a mechanism for namespace allocation. The best results are achieved if you establish a project-wide convention how to allocate and use namespaces for each platform and apply these conventions to interfaces as well.

1.2 Contract

Interface definitions must **describe interface contract**, not the implementation. This is the most important guideline of all. The definition should abstract from all implementation detail that it could. Think about the most bizarre implementation of the interface. The interface definition must be able to work even for the craziest practical implementation.

What you are describing is contract between the users and providers of the interface. You should abstract most of the implementation details and describe only the **boundary** between the two parties. Try to think ahead and only promise what you can keep in a subsequent revisions of the interface.

> **GOOD:** Content of the object will be kept synchronized with the underlying storage.
>
> **BAD:** Content of the object will be refreshed if the file /foo/bar.db is modified. Modifications to the object will be immediately flushed to the /foo/bar.db file.

Bad example above is too specific, mentioning a file name of the database. The implementations may store the data only in memory or may use remote database. The bad example is also promising too much regarding immediate detection and flushing of changes. This may work well for a first release, but later you may find that you need to improve system performance. The promise of immediate synchronization will not allow you to add efficient caching to the implementation unless you change in interface definition. Good example allows reasonable degree of freedom to the implementation while still being good enough a definition for interface users.

For important constructs of the interface (such as classes) always try to define a **responsibility** of a construct. Document the reason for existence of such construct and what a user can gain from using it. Also describe relations between other major constructs of the interface.

GOOD:

```
/**
 * Representation of bank account.
 * <p>
 * The interface provides access to high-level operations on
 * checking account such as withdrawal, deposit and transfer
 * between accounts.
 * <p>
 * ...
 * @see CheckingAccount
 */
interface BankAccount { ... }
```

BAD:

```
/**
 * Representation of bank account.
 */
interface BankAccount { ... }
```

1.3 Operations

Describe **operations, parameters, return values, messages, protocol data units** or any other mechanics of the interface. Describe both syntax and semantics of the parameters. Define and describe data types and data formats. This is quite easy for local interfaces in modern programming languages, but it may be difficult for remote, multi-platform messaging interfaces. Do not forget to describe file formats and formats for binary or stream-based arguments. Especially take care when the argument is in generic format, such as XML. Appropriate schema should always be part of the interface.

GOOD:

```
/**
 *
 * Returns value of a property.
 * <p>
 * Returns value of an component's property defined by
 * the <code>name</code> parameter. The method will return
 * value of the property (as String) if the property exists
 * or null otherwise.
 *
 * @param name property name. Must not be null.
 * @return value of the property. May return null.
 */
public String readProperty(String name);
```

BAD:

```
public String readProperty(String name);
```

It is obvious what's wrong with the bad example. There is no description (no contract), the parameter is not defined and return value is not defined (except for type definitions, but that is not enough).

1.4 Concurrency and Synchronism

Describe **concurrency and synchronism** properties of the interface. Clearly state whether the operations can be invoked from multiple processes or threads, if more that one operation can be in progress, if they can share a resource and so on. Describe whether you interface is synchronous or asynchronous. Synchronous operations are waiting for the operation to complete. They are easier to use for non-realtime clients.

However they are difficult to use correctly in low-level code or real-time systems. Asynchronous interfaces just initiate operation and return immediately, the result of the operation will be delivered when it is completed, usually by a callback mechanism. They allow complex handling of events, especially in low-level code and real-time systems, however they are difficult to use correctly. Message-oriented interfaces are always asynchronous and the message sequencing may be complex. Add description of interaction sequences, for example using UML sequence diagrams. If the interface is a complex network protocol, it may also require description of states and state transitions of communication parties.

GOOD:

```
/**
 *
  * Sets value of a property.
  * <p>
  * Sets value of an component's property defined by
  * the <code>name</code> parameter to a new value defined
  * by the <code>value</code> parameter.
  * <p>
  * This method is synchronous. It will block until the property
  * value is stored in the backing storage.
  * <p>
  * This method is thread-safe, it may be concurrently called
  * from several threads and the consistency of the operation
  * will hold. However, there is no versioning or locking
  * support. In case of conflicting updates only one of the
  * values will be set in the property and it cannot be
  * determined which one it will be.
  *
  * @param name property name. Must not be null.
  * @param value new property value. Must not be null.
  */
public void writeProperty(String name, String value);
```

BAD:

```
public void writeProperty(String name, String value);
```

Please note that the good example also mentions limitations, such as a possibility that the call will block or the weak consistency guarantee of the method.

1.5 Atomicity, Transactions and Consistency

Describe **atomicity, transaction and consistency guarantees**. Describe whether the operations change data and state atomically and whether the failure of operation can leave data in half-updated state. Describe any promises of the interface regarding transactional behavior, especially if it is expected that the interface will be invoked in a context of a transaction. Describe consistency guarantees, especially for remote interfaces. Describe interface promises about maintaining any reservations, temporary locks or any other mechanism that is used for consistency.

GOOD:

```
/**
 * Representation of bank account.
  * <p>
  * The interface provides ...
  * <p>
  * Every operation on the back account is expected to be
  * atomic, maintaining bank account consistency. For that
```

```
 * reason it is required that all the methods of this
 * interface must be  invoked in a context of an existing
 * (JTA) transaction, unless  explicitly stated otherwise in
 * method description.
 * ...
 */
interface BankAccount { ...
  /**
   * ...
   * Precondition: Existing (JTA) transaction
   */
  void transferTo(BankAccount target,Amount amount);
  ...
}
```

BAD:

```
/**
 * Representation of bank account.
 * <p>
 * The interface provides ...
 * <p>
 * Operations on the back account are atomic.
 */
interface BankAccount { ... }
```

The bad example is too vague in a way how atomicity is supposed to be provided. Such definition may imply that the atomicity is provided by the implementation without any requirement from the user (which is seldom the case). Good example above makes the requirement clear.

1.6 Limitations

Describe **limitations and undefined behavior**. Include all the limitations you can think of. The clients should not rely on anything that is not defined in the interface. But make the situation easier for them by mentioning the dangers explicitly. Define the limitations, which means cases where it is known that the interface will fail. Also warn about undefined behavior, which means situation where it may work or may not. Problems that originate from limitations and undefined behavior are difficult to diagnose, therefore make sure that the client is aware of them and can avoid them. No interface is perfect and (because all abstractions are leaky) all the interfaces are somehow limited. Be suspicious about interface definitions that do not document limitations.

> **GOOD:** The `resize` method changes the size of the file to a new absolute value. If the file is already filled with data beyond the new absolute size, the behavior of the method is undefined. It may produce an error or it may silently proceed creating unexpected results.

Good definition provides a warning that the behavior is undefined and that unexpected things may happen. This leaves a degree of freedom for implementations that may not be able to maintain a record of how much data was actually written and therefore cannot detect illegal resize and cannot raise an error.

> **BAD:** The `resize` method changes the size of the file to a new absolute value. The file must not be filled with data beyond the new absolute size.

Such definition is bad because it does not describe what happens if someone actually tries to resize the file filled with data beyond the new absolute size. The user of the interface could (wrongly) assume that the resize method will raise an error in such situation.

> **BAD:** The `resize` method changes the size of the file to a new absolute value. If the file is already filled with data beyond the new absolute size an `InvalidArgumentException` is thrown.

This definition promises too much. Current implementation may be able to detect that the user is trying to resize an object while it is filled with data beyond the new absolute size. But if you would need to provide alternative implementation (e.g. mock implementation that works on plain files instead of strongly managed data objects), such an implementation may not have the ability to detect that the file is filled with data beyond the new absolute size and it cannot raise an error in such a situation. That would mean that you will not be able to keep the interface contract and you will be forced to change the method definition.

1.7 Error Conditions

Describe **error conditions and exceptional states**. Error indicators, such as error codes and exceptions, are integral part of interface definition. Make sure that the error conditions are well-described. Error indicators should contain both human-readable message and computer-readable error code.

An operation/sequence of operations triggered by the interface must not drive the system into an inconsistent state, no matter how ridiculous or incorrect the input parameters or operation sequence were. Possible input parameter/operation sequence inconsistencies should be signaled by error indicators (exceptions, error codes). Generic, catch-all error indicators meaning “Generic error”, "An error occurred" or "There was an exception" must not be the only error indicators in the interface. The messages that accompany error indicators should state where and why an error occurred and provide details necessary to quickly track a possible bug. The user should not be expected to wade through several megabytes of system log and examine hundreds of lines of stack trace including a dozen exceptions wrapped inside each other.

GOOD:

```
/**
 * Returns value of a property.
 * <p>
 * Returns value of an component's property defined by
 * the <code>name</code> parameter.
 *
 * @param name property name. Must not be null.
 * @return value of the property. Cannot return null.
 * @throws NoSuchPropertyException
 *                          if the property does not exist
 * @throws IllegalArgumentException
 *                          if a parameter is null
 */
public String readProperty(String name)
                               throws NoSuchPropertyException;

/**
 * Exception indicating missing property.
 * <p>
 * This exception indicates an attempt to work with a
 * property that does not exist.
 */
public class NoSuchPropertyException { ...}
```

BAD:

```
/**
 * Returns value of a property.
 * <p>
 * Returns value of an component's property defined by
 * the <code>name</code> parameter.
 */
```

```
public String readProperty(String name);
```

The bad example does not provide any definition of error conditions, therefore the used might assume that it cannot fail. However it is very unlikely that the code will work properly if the requested property does not exist or if the parameter is `null`. Good example above provides a better description. The error condition is described both in the method javadoc (specific error for the method) and in javadoc for exception class (generic description of the error class). The exception class takes place of computer understandable error code as it may be used in `try-catch` blocks. Please also note the description of `IllegalArgumentException` in `@throws` javadoc tag. `IllegalArgumentException` is a runtime exception and it is not included in the method's `throws` clause. However it is good to describe that such exception can be thrown from the method in the javadoc. Such approach will ease debugging and improve system understandability.

BAD:

```
/**
  * Returns value of a property.
  * <p>
  * Returns value of an component's property defined by
  * the <code>name</code> parameter.
  *
  * @param name property name. Must not be null.
  * @return value of the property. Cannot return null.
  * @throws PropertyOperationException
  *                          in case of any error
public String readProperty(String name);


public class PropertyOperationException
                                  extends RuntimeException { …}
```

The above example does define an error condition, however the error condition is way too generic. It provides no indication of what is the problem, therefore the use of such interface has no chance to react to the situation. There is no textual description for `PropertyOperationException`, most likely because there is not much to write about. Also the `PropertyOperationException` is a runtime exception, therefore the compiler will not warn that the exception is not checked. This will cause that the exception will probably not be checked at all and even trivial recoverable errors will propagate up the call stack and cause a critical failure of the entire system.

1.8 Dependencies

Describe **dependencies** of the interface. Interfaces may depend on other concepts, usually data type and formats, industry specifications and standards. Describe all the concepts that the interface depends on. The interface should be reviewed and maybe even updated once these concepts change. Make sure that you are describing dependencies of the interface, not dependencies of implementation.

GOOD:

```
/**
  * Manager of persona representations.
  * <p>
  * Implementations of this interface manage personas
  * in local storage as well as provide access to remote
  * persona representations. ...
  * <p>
  * Remote personas are located using URLs. The URLs must be
  * resolvable for the implementation to operate properly,
  * therefore only ”concrete” types of URLs will work.
```

```
 * Implementations of this interface depend on the URL
 * resolution code provided by the <code>URL</code> class.
 *
 * @see URL
 */
interface PersonaManager { ... }
```

BAD:

```
/**
 * Manager of persona representations.
 * <p>
 * Implementations of this interface manage personas
 * in local storage as well as provide access to remote
 * persona representations. ...
 */
interface PersonaManager { ... }
```

The bad example does not mention URL as one of the basic concepts used by the interface (assuming URL class is not defined in this interface). At least a simple paragraph and a javadoc `@see` tag should be used to indicate that this interface has a dependency on a concept from other interface.

1.9 Preconditions

Describe **preconditions**, the conditions that needs to be satisfied before the interface can be used. If some object should be acquired, the implementation should be initialized, specific configuration is needed or system must be in a specific state document all of that in the interface definition.

GOOD:

```
/**
 * Returns next token from a token stream.
 * <p>
 * ...
 * <p>
 * Precondition: The stream must be open.
 */
public String getNextToken(TokenStream stream);
```

1.10 Form

You should define a convention for a uniform form of interface definition in the entire project. The convention should more-or-less hold for all types of interfaces used in the system, regardless whether these are local Java APIs or network protocols. The uniform way of defining interface is good for understandability and efficiency. Architect, designer and programmer could evaluate the purpose and state of interface definition almost at a first glance.

Inspire yourself by the way how standard documents and RFCs (IETF Requests for Comments) are written. Especially if your interface definition is about to be used outside your team, outside your company or even released to public. The strictness of interface definition form should be proportional to importance of the interface. Intra-component interfaces could be allowed to be defined very informally, while interfaces between subsystem or public interfaces of your system should have a strict requirements for definition quality and completeness.

The best results are achieved if a native tool is used for the target platform. For example using a tool for structured documentation inside the code (javadoc, doxygen) will significantly improve interface definition quality. Firstly, the programmers like to look at the code instead of reading through the documents. If the documentation is in the code, it has a higher probability of catching programmer's eye. Secondly, if the

documentation is bundled with the code and the code is modified, it is natural to modify the documentation right away. Therefore it is likely that the textual definition and the code will be kept consistent. If the documentation is separated from the code, the one that updates the interface frequently forgets to update the documentation as well.

Each interface should have short page, document, file or a section in the source code comments that defines few basic facts about the interface. We will call this section an interface definition **header**. The header should contain following information:

- **Name:** Name of the interface. Should be a full name, not an acronym.
- **Responsible person:** Name of the person who is responsible for interface maintenance. The person who should be contacted if a problem with the interface is discovered or when an explanation is needed. The person that coordinates the updates of the interface.
- **Version:** Interface version, last revision from source code control system, product version that this interface is a part of or date of last interface modification. This is a good tool to evaluate the freshness of interface definition. Interface definitions have a tendency to diverge from the code and version numbers are one of the mechanisms that to judge how accurate the definition is.
- **Status:** Whether interface is public, internal, accessible only inside subsystem, inside component, whether it is experimental, ready for release, etc.
- **Platform:** Programming language or other definition of platform. Specifies if the interface is local Java API, network protocol, message format for MOM system, XML dialect, etc. Also include version of the platform if it is important or a language dialect/extension if it is non-standard.
- **Dependencies:** Enumerate the interfaces that this interface depends on. For example if this interface uses data types from other interface or it relies on a concept that is defined elsewhere. Please be careful to describe only the dependencies of interface, not the dependencies of implementations. Obvious dependencies (e.g. Java core libraries) are usually not mentioned.
- **Description:** Few lines that describe the purpose of the interface, it's reason for existence in the system. It may also describe basic idea, but the description should not be longer than a dozen sentences.
- **History:** Short history of major interface revisions. Description of dates and the most important changes. Just one sentence for each history entry.

You may also want to add following entries to interface definition header, however if you do make them optional. Too much data in the header usually means too much data that can be out of sync with reality.

- **Pre-conditions:** The conditions that must be satisfied for all interface implementations to work. This may include a need to initialize a subsystem, for a system to be on-line, requirement to invoke another interface before this interface can be used, etc. However, avoid describing pre-conditions of a specific interface implementation. The preconditions specified here should be a generic preconditions that apply to all interface implementations as a direct consequence of the interface design.
- **Post-conditions:** Conditions that will be satisfied after invocation of interface operations. These may be very useful for testing of interface implementations.
- **Concurrency:** Define the concurrency, atomicity, consistency, transactional or other interface limitations. For example specify whether the interface is thread-safe or not, if the operations can take part in transactions, etc. Be careful to describe the requirements of the interface, not implementations. This field should specify requirements for all implementations (e.g. thread-safety) or direct consequences of interface design (e.g. operations assume that a transaction exists before invoking any operation).

- **Idempotence:** Describe the state management properties and side effects of the interface. Does two invocations of the interface influence each other (not idempotent) or are independent (idempotent). Is there any difference if an operation is invoked once or many times? Does network conditions or a state of a remote node influence the result of interface operations?
- **Persistence:** Describe how long the effects of interface invocations persist. Do the effects disappear at the time the operation is finished? Or do the effects persist until the system is restarted? Are the results saved to a disk system and remains there for weeks? Or years?
- **Security:** What are the security properties of interface? Is is sensitive remote interface that needs establishment of keys and certificates to operate properly? Or is it a local interface that assumes interaction in a trusted environment?

In all the header be careful not to describe the properties of implementation, but only properties of the interface. Each field should specify requirements for all implementations (e.g. thread-safety, ability to keep state across system restarts) or direct consequences of interface design (e.g. interface designed to work with disk drives can assume long-term persistence).

Following table provides an example of interface definition header:

Name	System Configuration Java API
Responsible Person	Radovan Semančík
Version	1.1
Status	Public
Platform	Java SE 5
Dependencies	FooBar Common Classes for Java
Description	Provides easy-to-use access to system configuration. May be used by management tools to read and modify system configuration and to monitor system state.
History	1.1, September 2008: Minor bugfixes and adjustments 1.0, May 2008: Initial release
Pre-conditions	System started up to milestone CONFIG (may not be fully initialized yet)
Concurrency	Thread-safe
Idempotence	Idempotent except for configuration changes
Persistence	Configuration changes survive system restart
Security	No security, trusted environment assumed

Following table provides recommended interface definition form for some platforms:

Platform or Language	***Form***
Java	Provide interface definition in a form of Java source package. It should only contain `interface`, `abstract class` and `final class` constructs if possible. Each file should be well documented using *javadoc*. The class or interface should provide introductory text in javadoc in front of opening definition and then specific javadoc section in front of each method or field. The javadoc text should document the interface contract, including all limitations that cannot be expressed in Java code (e.g. whether the values for parameters and return statements can be `null` or cannot).
C	Provide interface definition in a form of C header file or several header files. The files should contain data structure, macro and function definitions, each of them well documented using doxygen-style comments.

SOAP	Provide complete and valid WSDL document including all necessary XML Schema types definitions. Use XML comments or appropriate documentation XML tags to describe the interface contract. Document especially the WSDL operations, but do not forget to document usual XML constructs as well (see XML below).
XML	Provide XML Schema Definition (XSD) file for the XML dialect. Use XML comments or appropriate documentation tages in XSD to document the structures. The XML Schema type definitions may et very complex, therefore document the meaning and purpose of data types. Use text in comments to document limitations that cannot be expresses in XML schema language.
Network Protocol	Use semi-formal protocol specification similar to standards, RFCs or other normative specifications. Define underlying protocol, protocol data units (PDUs), message sequences and state machines of communicating parties. Do not forget to define errors, timeouts and similar exceptional conditions. Sequence or timing diagrams showing the communication are usually very useful.
Message Exchange	For interfaces based on message exchange (MOM, IPC or similar) define the communication channel and message formats using any mechanism appropriate for communication channel. You usually do not need to specify underlying messaging mechanism in details (it is abstracted away by messaging channel), but the rest is the same as for the network protocol case.

1.11 Notes

Although this text describe interface definition and not interface design per se, there are is one design guideline that is worth mentioning: **keep interfaces minimal**. The interface is not perfect when there is nothing more to add. The interface is perfect when there is nothing more to remove. Every class, method, function, attribute or any other construct needs a reason to exist. If the reason is not obvious at the first sight, document the reason in the description. Try to provide only one way to do something. Do not try to provide operations that does two things in one call unless there are very strong reasons to do so (e.g. atomicity). And even if there is a very very good reason, document the reason and all the motivation in the interface description.

Catalog interfaces in your system, especially public, inter-subsystem and inter-component interfaces. Cataloging the interfaces gives you some idea how well is your system structured. Adding development status to the catalog will create a kind of project dashboard that can be used to track the progress of system design and re-design. The catalog will also allow junior architects and designers to quickly look for a functionality that the system provides, to quickly find appropriate interfaces for new or modified components. Adding live references to artifacts (files) that contain interface definition will significantly increase the value of interface catalog.

Track endpoints where the interfaces are exposed. Endpoints are URLs where web services can be accessed, network ports where daemons listen, RPC ports and names, references in a naming service, class instance handles and names, instance identifiers used in dependency injection frameworks, etc. Endpoints are not abstract. Endpoints represent specific implementation and therefore are not, strictly speaking, part of interface definition. However, each endpoint implements particular interface, therefore tracking endpoints, cataloging them and checking the development status of implementation behind them will give you very good idea about the state of project development (implementation).

This text is not intended to be used as is. It is not a guideline for interface definition that can used in a project in this very form. It is a rather to be used as inspiration to create your own interface definition guidelines. In practical situation you may find that changing or ignoring some of the guidelines will improve your results. Each project is different, therefore the guidelines has to adapt as well. Use this material with case and apply common sense whenever possible.

Provided examples are not expected to be complete or absolutely correct. The demonstration of “good” and “bad” may also not entirely fit into your project. Some project will find that what is recommended as “good” is too much work and going with the “bad” example actually improves the result. Once again, use this document only for inspiration, not as a dogma.

2 Interface Definition Checklist

Is interface name appropriate?

Isn't it too short and generic or too long and difficult to remember?

If it is long, is the acronym easy to remember, pronounceable or even meaningful?

Does the name include platform, language or any other major limitations of the interface?

Does it has a dedicated namespace according to the project conventions?

Does the definition contain enough summary information (header)?

Have you specified name of a responsible person (author or maintainer)?

Have you specified interface version?

Have you specified interface status (public, internal, experimental, ...)?

Have you specified platform (including language version)?

Have you listed interface dependencies on other interfaces?

Have you included a short description (no more than few lines)?

Have you updated interface history record (if this is a major revision)?

Would it help to include any additional information (pre-conditions, post-conditions, concurrency, Idempotence, persistence, security)?

Is there anything to remove?

Are there redundant classes in the interface? Classes that have similar responsibility or function?

Is there more than one way how to do something? Do we really need that?

If we need redundant constructs are the motives documented well enough?

Is interface definition precise enough?

Are the limitations documented? Is there a class or operation without any limitations?
(It is unlikely that there are no limitations, therefore it is likely that the limitations are just not documented.)

Are all the constructs defined (structs, macros, classes, interfaces, XML data types, …)?

Are all data types in operation parameters and return values defined or is appropriate reference to their definition provided?

Are all error indicators (exceptions, error codes) well defined in the interface?

For all the operations, is there at least one error condition defined?
(It is unlikely that operation will always succeed, therefore it is likely that the error is just not documented.)

Be especially careful about time-outs and network errors. Does the definition account for these?

3 Copyright