ASPROTUNITY
www.asprotunity.com

Writing usable APIs in practice

ACCU 2012 Conference, Oxford, UK

Giovanni Asproni
gasproni@asprotunity.com
@gasproni

Summary

- API definition
- Two assumptions
- Why bother with usability
- Some techniques to improve usability

API

"Any well-defined interface that defines the service that one component, module, or application provides to other software elements"

From: "Sometimes You Need to See Through Walls — A Field Study of Application Programming Interfaces", Cleidson R. B. de Souza et al., http://www.ufpa.br/cdesouza/pub/p390-desouza.pdf

Package java.rmi

Provides the RMI package.

See:
Description

Interface Summary	
Remote	The `Remote` interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine.

Class Summary	
MarshalledObject<T>	A `MarshalledObject` contains a byte stream with the serialized representation of an object given to its constructor.
Naming	The `Naming` class provides methods for storing and obtaining references to remote objects in a remote object registry.
RMISecurityManager	A subclass of `SecurityManager` used by RMI applications that use downloaded code.

Exception Summary	
AccessException	An `AccessException` is thrown by certain methods of the `java.rmi.Naming` class (specifically `bind`, `rebind`, and `unbind`) and methods of the `java.rmi.activation.ActivationSystem` interface to indicate that the caller does not have permission to perform the action requested by the method call.
AlreadyBoundException	An `AlreadyBoundException` is thrown if an attempt is made to bind an object in the registry to a name that already has an associated binding.
ConnectException	A `ConnectException` is thrown if a connection is refused to the remote host for a remote method call.
ConnectIOException	A `ConnectIOException` is thrown if an `IOException` occurs while making a connection to the remote host for a remote method call.
MarshalException	A `MarshalException` is thrown if a `java.io.IOException` occurs while marshalling the remote call header, arguments or return value for a remote method call.
NoSuchObjectException	A `NoSuchObjectException` is thrown if an attempt is made to invoke a method on an object that no longer exists in the remote virtual machine.
NotBoundException	A `NotBoundException` is thrown if an attempt is made to lookup or unbind in the registry a name that has no associated binding.
RemoteException	A `RemoteException` is the common superclass for a number of communication-related exceptions that may occur during the execution of a remote method call.
RMISecurityException	**Deprecated.** *Use* `SecurityException` *instead.*
ServerError	A `ServerError` is thrown as a result of a remote method invocation when an `Error` is thrown while processing the invocation on the server, either while unmarshalling the arguments, executing the remote method itself, or marshalling the return value.
ServerException	A `ServerException` is thrown as a result of a remote method invocation when a `RemoteException` is thrown while processing the invocation on the server, either while unmarshalling the arguments or executing the remote method itself.
ServerRuntimeException	**Deprecated.** *no replacement*
StubNotFoundException	A `StubNotFoundException` is thrown if a valid stub class could not be found for a remote object when it is exported.
UnexpectedException	An `UnexpectedException` is thrown if the client of a remote method call receives, as a result of the call, a checked exception that is not among the checked exception types declared in the `throws` clause of the method in the remote interface.
UnknownHostException	An `UnknownHostException` is thrown if a `java.net.UnknownHostException` occurs while creating a connection to the remote host for a remote method call.
UnmarshalException	An `UnmarshalException` can be thrown while unmarshalling the parameters or results of a remote method call if any of the following conditions occur: if an exception occurs while unmarshalling the call header if the protocol for the return value is invalid if a `java.io.IOException` occurs unmarshalling parameters (on the server side) or the return value (on the client side).

Package java.rmi Description

Provides the RMI package. RMI is Remote Method Invocation. It is a mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine. Any object that can be invoked this way must implement the Remote interface. When such an object is invoked, its arguments are "marshalled" and sent from the local virtual machine to the remote one, where the arguments are "unmarshalled." When the method terminates, the results are marshalled from the remote machine and sent to the caller's virtual machine. If the method invocation results in an exception being thrown, the exception is indicated to caller.

Since:
JDK1.1

Overview Package Class Use Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES All Classes
Java™ Platform Standard Ed. 6

Public and private APIs

- In this talk we define:
  - Public APIs as APIs that are produced to be given to third parties
  - Private APIs as APIs that are created for internal project use

First assumption

Any non trivial software application involves writing one or more APIs

Second assumption

When we talk about **good code** we always mean **usable** code as well

When talking about good code...

- We always talk about principles we should apply
  - Single responsibility principle
  - Open closed principle
  - DRY (don't repeat yourself)
  - Principle of least astonishment
  - ...
- but it is not always clear how to apply them

We will talk about

- User's perspective
- Naming
- Give control to the caller
- Explicit context
- Error reporting
- Logging as a feature
- Organisation of modules and classes

Why bother (company's perspective)

- APIs can be among a company's greatest assets
  - Users invest heavily: buying, writing, learning
    - Cost to stop using an API can be prohibitive
- Can also be among company's greatest liabilities
  - Bad APIs result in unending stream of support calls

Adapted from: "How to Design a Good API and Why it Matters", Joshua Bloch, http://lcsd05.cs.tamu.edu/slides/keynote.pdf

Why bother (programmer's perspective)

- Fewer bugs to take care of
- Code of higher quality
- More productivity
- Less frustration when solving a problem
- Bad APIs result in unending stream of support calls

Why bother (Arjan van Leeuwen perspective)

- Fewer lines of code are better
- To promoting simple code with library design

Affordances

An affordance is a quality of an object, or an environment, that allows an individual to perform an action. For example, a knob affords twisting, and perhaps pushing, while a cord affords pulling.

From: http://en.wikipedia.org/wiki/Affordance

Affordances: processing file line by line in Java

```
BufferedReader reader;
try {
    reader = new BufferedReader(new FileReader("filename"));
    while (true) {
        String line = reader.readLine();
        if (line == null) {
            break;
        }
        processLine(line);
    }
}
catch (Exception exc) {
    // Do something here...
}
finally {
    if (reader != null) {
        reader.close();
    }
}
```

Affordances that are easier to see (almost Java)

```
File file = new File("filename"));
try {
    for (String line : file.readLines()) {
        processLine(line);
    }
}
finally {
    file.close()
}
```

And even easier (Python)

```
with open("filename") as infile:
    for line in infile.readlines():
        processLine(line)
```

Some important things for usability

- Abstraction level. The minimum and maximum levels of abstraction exposed by the API
- Working framework. The size of the conceptual chunk (developer working set) needed to work effectively
- Progressive evaluation. To what extent partially completed code can be executed to obtain feedback on code behaviour
- Penetrability. How the API facilitates exploration, analysis, and understanding of its components
- Consistency. How much of the rest of an API can be inferred once part of it is learned

Adapted from: "Measuring API Usability", Steven Clarke, http://drdobbs.com/windows/184405654

Conceptual integrity

I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.

Fred Brooks, "The Mythical Man Month"

"Make Interfaces Easy to Use Correctly and Hard to Use Incorrectly"

Scott Meyers, "97 Things Every Programmer Should Know"

Golden Rule of API Design

"It's not enough to write tests for an API you develop; you have to write unit tests for code that uses your API. When you follow this rule, you learn firsthand the hurdles that your users will have to overcome when they try to test their code independently."

Michael Feathers, "97 Things Every Programmer Should Know"

- User's perspective
- Naming
- Give control to the caller
- Explicit context
- Error reporting
- Logging as a feature
- Organisation of modules and classes

"Ask, 'What Would the User Do?' (You Are Not the User)"

Giles Colborne, "97 Things Every Programmer Should Know"

TDD

- It puts you in the shoes of an user
  - If writing a test is painful, the design may be wrong
- Tests will provide up to date documentation and examples of use

TDD helps with

- Abstraction level. It helps to limit the number of abstractions in mainline scenarios
- Working framework
- Progressive evaluation
- Penetrability. It provides examples on how the various components interact with each other
- Consistency. It is maintained by refactoring the code

Adapted from: "Measuring API Usability", Steven Clarke, http://drdobbs.com/windows/184405654

The file example again

```
File file = new File("filename"));
try {
    for (String line : file.readLines()) {
        processLine(line);
    }
}
finally {
    file.close()
}
```

And a version with more abstraction levels

```
BufferedReader reader;
try {
    reader = new BufferedReader(new FileReader("filename"));
    while (true) {
        String line = reader.readLine();
        if (line == null) {
            break;
        }
        processLine(line);
    }
}
catch (Exception exc) {
    // Do something here...
}
finally {
    if (reader != null) {
        reader.close();
    }
}
```

- User's perspective
- Naming
- Give control to the caller
- Explicit context
- Error reporting
- Logging as a feature
- Organisation of modules and classes

- Reserve the simplest and most intuitive names names for the entities used in the most common scenarios
- Pick one word per concept
- Don’t be cute!

- User's perspective
- Naming
- Give control to the caller
- Explicit context
- Error reporting
- Logging as a feature
- Organisation of modules and classes

What's wrong with these?

```
public interface Startable {

    Startable start()
        throws AlreadyStartedException;
}


public interface Stoppable {

    Stoppable stop()
        throws AlreadyStoppedException;
}
```

A better alternative

```
public interface Service {

    void start()
        throws AlreadyStartedException;

    void stop()
        throws AlreadyStoppedException;

    boolean isStarted();
}
```

- User's perspective
- Naming
- Give control to the caller
- Explicit context
- Error reporting
- Logging as a feature
- Organisation of modules and classes

Explicit context

- This about the assumptions on the external environment
- There are two kinds of context we are interested in
  - Deployment context
  - Runtime context

Deployment context

- Dependencies on other APIs
- Assumptions on deployment paths
- User permissions
- etc.

Runtime context

- Preconditions for calling methods (or functions) or instantiating classes
- Initialisation (and finalisation) steps
- etc.

Be careful with global state

- Using globals (yes, singletons are globals) can impose huge constraints in the testability and the usability of the API
- It will be difficult to use in a concurrent environment
- The setup of the tests can become extremely hard
- It can be difficult to use some functionality if it requires the setting of some magic variables with no clear link to the functions and classes used

- User's perspective
- Naming
- Give control to the caller
- Explicit context
- Error reporting
- Logging as a feature
- Organisation of modules and classes

Error reporting

- Error reporting code is important for usability
- Users need to know
  - How errors are reported
  - What is reported when
  - What they can do about them

Recovering from an error

- It is important to classify error in a way that makes recovery easy to do programmatically
  - Error codes
  - Exception classes
  - A mix of the above
- Text messages are usually not good enough

What is an error at one level....

- ...May not be an error at another one

- User's perspective
- Naming
- Give control to the caller
- Explicit context
- Error reporting
- Logging as a feature
- Organisation of modules and classes

Two types of logging

- **Support logging** (errors and info) is part of the user interface of the application. These messages are intended to be tracked by support staff, as well as perhaps system administrators and operators, to diagnose a failure or monitor the progress of the running system.
- **Diagnostic logging** (debug and trace) is infrastructure for programmers. These messages should not be turned on in production because they're intended to help the programmers understand what's going on inside the system they're developing.

From: "Growing Object Oriented Code Driven by Tests", Steve Freeman and Nat Pryce, Addison Wesley

```
Location location = tracker.getCurrentLocation();

for (Filter filter : filters) {

    filter.selectFor(location);

    if (logger.isInfoEnabled()) {
        logger.info("Filter " + filter.getName() + ", " + filter.getDate()
        + " selected for " + location.getName()
        + ", is current: " + tracker.isCurrent(location));

    }
        }

}
```

From: "Growing Object Oriented Code Driven by Tests",
Steve Freeman and Nat Pryce, Addison Wesley

```
Location location = tracker.getCurrentLocation();
for (Filter filter : filters) {
    filter.selectFor(location);
    support.notifyFiltering(tracker, location, filter);
}
```

From: "Growing Object Oriented Code Driven by Tests",
Steve Freeman and Nat Pryce, Addison Wesley

Give the programmer a choice

- The programmer may not be interested in the logs of the API. Give he a chance to turn them off
- But, if he is interested, give him a chance to get them and use them on his own terms

- User's perspective
- Naming
- Give control to the caller
- Explicit context
- Error reporting
- Logging as a feature
- Organisation of modules and classes

Start specific and small

- Don't try to overgeneralise your design from the beginning.
  - Solve a specific problem first and generalise later. Until you use your code, it is unlikely that you'll know in which direction you want to generalise it
- Provide one way only to do one thing. You can add more later if necessary (with an 80% case for everybody and the 20% one for whoever needs a finer grain of control)
- Start with the 80% case first
- It is always easier to remove constraints rather than to add them later
- YAGNI

Cohesiveness and coupling

- Make APIs highly cohesive and loosely coupled
  - Don't create a "constants" package or class
  - Don't create an "exceptions" package
  - Util/Manager/Helper classes are usually evil
  - Put together things that belong together
  - Separate things that don't belong together
  - Single responsibility is not one method per class

Bonus slide

- Order of function parameters
- A consistent ordering helps the user to predict what comes next and where to find what he's looking for.

From: http://www.codeproject.com/Articles/8707/API-Usability-Guidelines-to-improve-your-code-ease

A caveat

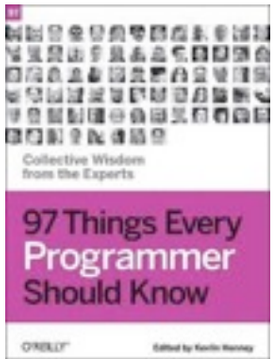
- Public APIs are more difficult to refactor. In fact, some errors may actually become features
- Techniques to refactor them usually involve some form of deprecation and versioning

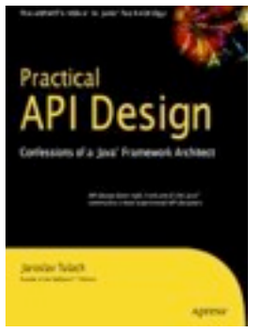
- User's perspective
- Naming
- Give control to the caller
- Explicit context
- Error reporting
- Logging as a feature
- Organisation of modules and classes

Links

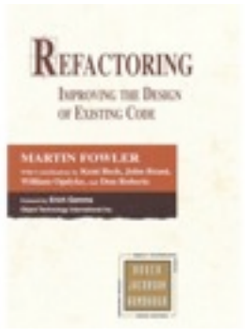
- http://www.apiusability.org
- "Sometimes You Need to See Through Walls — A Field Study of Application Programming Interfaces", Cleidson R. B. de Souza et al., http://www.ufpa.br/cdesouza/pub/p390-desouza.pdf
- "Measuring API Usability", Steven Clarke, http://drdobbs.com/windows/184405654
- http://en.wikipedia.org/wiki/Affordance
- "How to Design a Good API and Why it Matters", Joshua Bloch, http://lcsd05.cs.tamu.edu/slides/keynote.pdf
- "What Makes APIs Difficult to Use?", Minhaz Fahim Zibran, http://paper.ijcsns.org/07_book/200804/20080436.pdf
- http://www.codeproject.com/Articles/8707/API-Usability-Guidelines-to-improve-your-code-ease

Books

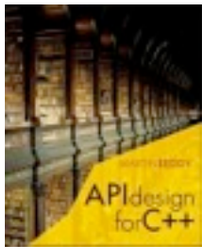

Collective Wisdom from the Experts
97 Things Every Programmer Should Know
O'REILLY


Practical
API Design
Confessions of a Java Framework Architect

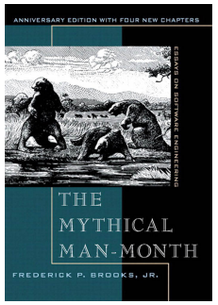

Growing Object-Oriented Software, Guided by Tests
Steve Freeman
Nat Pryce

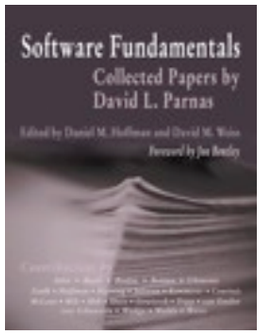

Refactoring
Improving the Design of Existing Code
Martin Fowler


Clean Code
A Handbook of Agile Software Craftsmanship


API design for C++


Simple and Usable


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.


Software Fundamentals
Collected Papers by David L. Parnas