

# API Design Matters

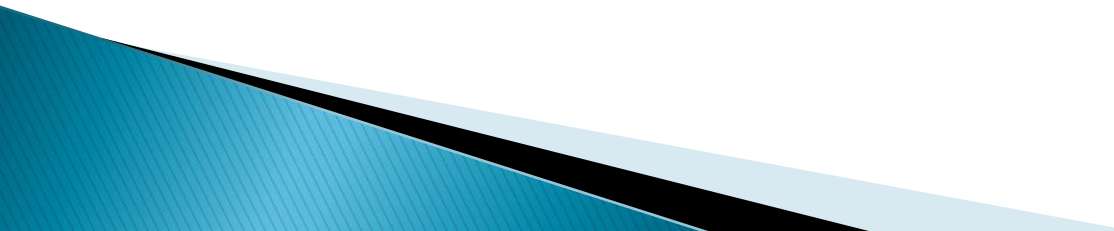
Why changing APIs might become a criminal offence



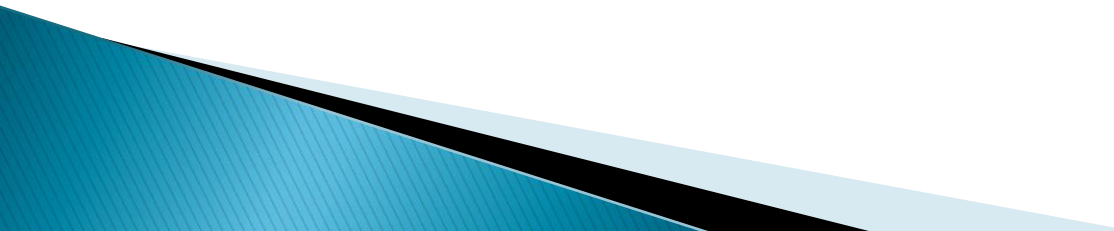
**ZeroC**

Michi Henning  
Chief Scientist, ZeroC, Inc.

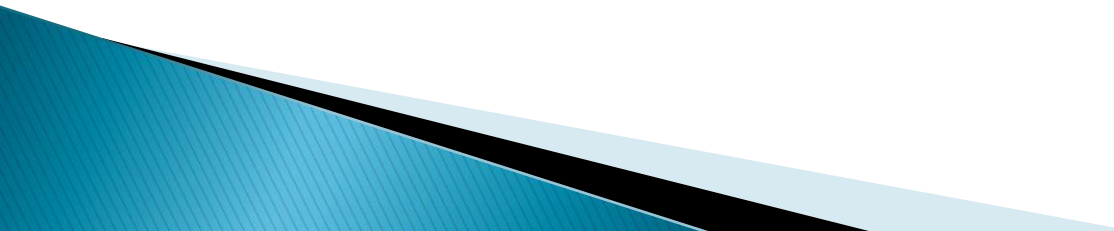
# The Problem

- ▶ I take on a programming task.
  - ▶ I estimate how long it will take.
  - ▶ It takes twice as long as my estimate.
  - ▶ Why?
  - ▶ Some @\$^^!!! API throws rocks in my path.
- 

# Good APIs

- ▶ are a joy to use
  - ▶ disappear from sight
  - ▶ are intuitive
  - ▶ are complete
  - ▶ can be memorized easily
  - ▶ are well documented
- 

# Bad APIs are Easy

- ▶ For every good API, there are dozens of bad ones.
  - ▶ Why? Because for every way to do it right, there are dozens of ways to do it wrong.
  - ▶ Minor glitches in design have surprising consequences and cause collateral damage.
  - ▶ Bad APIs are a major contributor to budget overruns.
- 

# Example: .NET Select()

```
public static void Select(  
    IList checkRead, IList checkWrite,  
    IList checkError, int microseconds);
```

- ▶ Method accepts three list of sockets to monitor, plus timeout.
- ▶ Call returns with lists of sockets that are ready or when timeout expires.

# Typical Use of select ()

```
int timeout = ...;
ArrayList readList = ...; // Monitor for reading.
ArrayList writeList = ...; // Monitor for writing.
ArrayList errorList = ...; // Monitor for errors.

while(!done) {
    // Monitor activity here...
}
```

# Typical Use of select ()

```
while(!done) {
    ArrayList readTmp = new ArrayList(readList);
    ArrayList writeTmp = new ArrayList(writeList);
    ArrayList errorTmp = new ArrayList(errorList);

    Select(readTmp, writeTmp, errorTmp, timeout);

    for(int i = 0; i < readTmp.Count; ++i) {
        // Deal with each socket that is ready for reading...
    }
    for(int i = 0; i < writeTmp.Count; ++i) {
        // Deal with each socket that is ready for writing...
    }
    for(int i = 0; i < errorTmp.Count; ++i) {
        // Deal with each socket that encountered an error...
    }
    if(readTmp.Count == 0 && writeTmp.Count == 0 &&
        errorTmp.Count == 0) {
        // No sockets are ready...
    }
}
```

# Problems with `select()`

- ▶ **`ArrayList`** is not type safe.
- ▶ **`select()`** overwrites its arguments.
  - If a server monitors 100 sockets, it copies 300 list elements in each iteration.
- ▶ Lists can contain duplicates.
  - What happens if the same socket appears more than once in a list?
- ▶ **`select()`** has void return value.
  - How to tell whether the call returned because of a timeout?
- ▶ Removing a socket from the lists requires linear time.

# Problems with `Select()`

- ▶ `Select()` does not scale:
  - Copying of lists is expensive.
  - Removal of sockets is expensive.
- ▶ How do I wait forever for a socket to become ready?
  - Zero timeout returns immediately.
  - Negative timeout returns immediately (.NET 2.0).
  - `Int.MaxValue` ( $2^{31}-1$ ) works out to a little over 35 minutes.
  - Timeout granularity of  $\mu\text{sec}$  is meaningless.

# A Drop-In Replacement

```
public static void
doSelect(IList checkRead, IList checkWrite,
         IList checkError, int microseconds)
{
    ArrayList readCopy; // Copies of the three
    ArrayList writeCopy; // parameters because
    ArrayList errorCopy; // Select() clobbers them.

    if (milliseconds <= 0) {
        // Simulate waiting forever.
    } else {
        // Deal with non-infinite timeouts
    }

    // Copy the three lists back into the
    // original parameters here...
}
```

# A Drop-In Replacement

```
if (milliseconds <= 0) {
    // Simulate waiting forever.
    do {
        // Make copy of the three lists here...

        Select(readCopy, writeCopy, errorCopy, Int32.MaxValue);
    } while (    (readCopy == null || readCopy.Count == 0)
                && (writeCopy == null || writeCopy.Count == 0)
                && (errorCopy == null || errorCopy.Count == 0));
} else {
    // Deal with non-infinite timeouts
    // ...
}
```

- ▶ Awkward test for loop termination because `Select()` has no return value and there are two ways indicate “no socket”: `nil` and zero-length list.
- ▶ Lists are clobbered by `Select()`, so need to be copied even when nothing happens.

# A Drop-In Replacement

```
if (milliseconds <= 0) {
    // Simulate waiting forever.
    // ...
} else {
    // Deal with non-infinite timeouts
    while ((milliseconds > Int32.MaxValue / 1000)
        && readCopy == null || readCopy.Count == 0)
        && writeCopy == null || writeCopy.Count == 0)
        && errorCopy == null || errorCopy.Count == 0)) {

        // Make a copy of the three lists here...

        Select(readCopy, writeCopy, errorCopy,
            (Int32.MaxValue / 1000) * 1000);
        milliseconds -= Int32.MaxValue / 1000;
    }

    if ((readCopy == null || readCopy.Count == 0)
        && (writeCopy == null || writeCopy.Count == 0)
        && (errorCopy == null || errorCopy.Count == 0)) {

        Select(checkRead, checkWrite, checkError,
            milliseconds * 1000);
    }
}
```

# A Drop-In Replacement

Copying the lists is awkward:

- ▶ **IList** is an interface.
- ▶ .NET does not have a **Clone** method on **Object**; cloneable objects must derive from **ICloneable**.
- ▶ **IList** does not derive from **ICloneable**.
- ▶ Copying the lists must be done element-by-element.

# A Drop-In Replacement

Overall length of the drop-in replacement:

- over 100 lines of code (with a few comments)

Why?

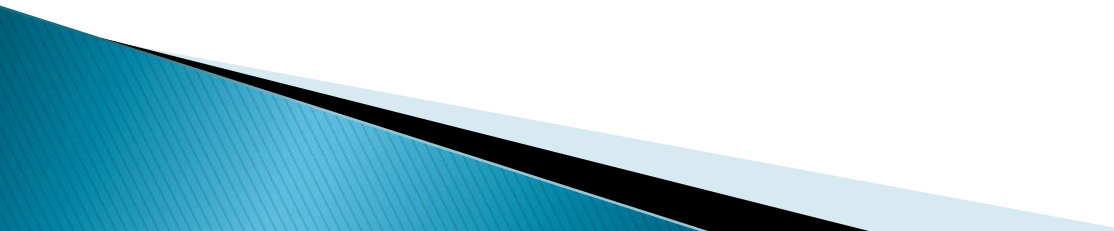
- **Select()** overwrites its arguments.
- **Select()** does not provide a return value to distinguish timeout return.
- **Select()** does not allow timeout > 35 minutes.
- **Select()** uses lists instead of (non-existent, in .NET) sets.

# A Better Version of `select()`


```
public static int  
    Select(ISet checkRead,  
          ISet checkWrite,  
          Timespan seconds,  
          out ISet readable,  
          out ISet writeable,  
          out ISet error);
```

- ▶ Return value indicates number of sockets that are readable.
- ▶ No separate error list on input.
- ▶ Arguments are not clobbered by the method.
- ▶ **Timespan** permits arbitrary-length timeouts.
- ▶ (Hypothetical) **ISet** disallows duplicates.

# The Cost of Bad APIs

- ▶ Difficult and error-prone to program with
  - ▶ Require additional code
  - ▶ Additional code makes programs larger, increases working set size and reduces cache hits.
  - ▶ Additional code is often inefficient due to extra data copies and wasted CPU cycles.
- 

# The *Real* Cost of Bad APIs

- ▶ Poor APIs are harder to understand and work with than good ones.
  - ▶ Programmers take longer to write code against a poor API.
  - ▶ More complex code means more bugs.
  - ▶ More complex code means more testing effort.
  - ▶ More complex code makes it more likely for bugs to go unnoticed.
  - ▶ Wrappers do *not* solve the problem!
- 

# API Design Guidelines

- ▶ Create sufficient APIs.
- ▶ Smaller is better: good APIs are minimal: they provide just enough, and no more.
  - Avoid Winnebago classes
- ▶ Be clear about how to do something
  - `wait()`, `waitpid()`, `wait3()`, `wait4()`, `waitid()`
  - At some point, extra complexity outweighs backward compatibility.

# API Design Guidelines

- ▶ APIs cannot be designed without an understanding of their context.

- ▶ 

```
class NVpairs {  
    public string lookup(string name);  
    // ...  
}
```

What should happen when a lookup does not find something? Empty string? Nil? Exception?

# API Design Guidelines

- ▶ General-purpose APIs are “policy-free”
- ▶ Specific-purpose APIs are “policy-rich”
- ▶ APIs dictate a particular programming style. If you know the style, do not be afraid to enforce it!
- ▶ If you do not know the style, leave it open as much as possible. (`select()` violates this.)
- ▶ The more “fascist” and API is, the safer it is.

# API Design Guidelines

- ▶ Design from the caller's perspective

```
makeTV(true, false);
```

This is not design from the caller's perspective. This is:

```
makeTV(Color, FlatScreen);
```

- ▶ Designing from the caller's perspective requires forethought:

```
void makeTV(bool isColor, bool isFlat);
```

versus:

```
enum ColorType { Color, BlackAndWhite };  
enum ScreenType { FlatScreen, CRT };  
void makeTV(ColorType color, ScreenType screen);
```

# API Design Guidelines

- ▶ Don't pass the buck!
  - “I can't decide whether the caller may want to do S or Y, so I'll provide both.”
  - Efficiency should *never* compromise API design.
- ▶ 

```
int select(int nfd, fd_set *readfds,  
          fd_set *writefds, fd_set *exceptfds,  
          struct timeval *timeout);
```
- ▶ **EINTR** is a bad idea.

# API Design Guidelines

- ▶ Document *before* implementation
- ▶ Write *complete* documentation
- ▶ Error behavior is as much part of the formal contract as non-error behavior
- ▶ `public class SocketException : Win32Exception { ... }`

```
public class Win32Exception {  
    public int NativeErrorCode { get; }  
}
```

This sucks!

- ▶ Document side effects: weak exception guarantee, strong exception guarantee?
- ▶ The worst person to write doc is the implementer
- ▶ The worst time to write doc is after implementation

# API Design Guidelines

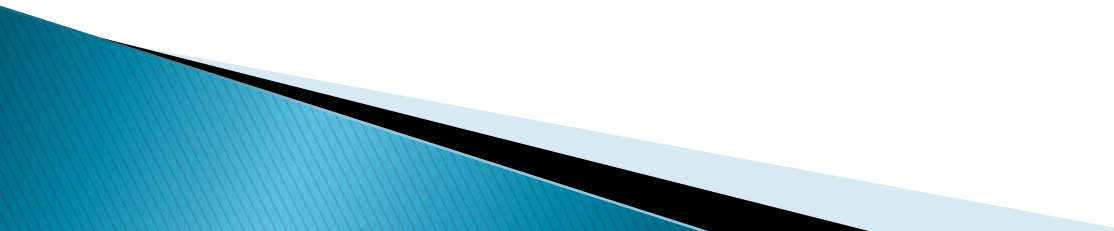
- ▶ Good APIs are ergonomic
  - Be consistent in
    - naming conventions
    - parameter ordering
  - Avoid adjacent same-typed parameters
  - Take advantage of transference
  - `read()`, `write()` versus `fgets()`, `fputs()`. But `fscanf()` and `fprintf()` are the odd ones out...

# Cultural Change

## ▶ Education

- Most university courses never teach API design
- Software development has changed from creation to integration
- We are designing more complex APIs than ever before
- Undergraduates *can* learn how to judge the quality of an API

# Career Path

- ▶ I am 49, and I write code.
  - ▶ I know no-one else who does.
  - ▶ Programmers past 40 don't "loose the edge".
  - ▶ There is no substitute for experience.
  - ▶ Be wary of promoting your best programmers to incompetence.
  - ▶ The best way to keep a designer sharp is to make him eat his own dog food.
- 

# External Controls

- ▶ Innocent minor changes to APIs can have drastic consequences.
- ▶ The world's economy depends on the correct functioning of many APIs:
  - libc, Win32, OpenSSL, UNIX system call interface, etc, etc.
- ▶ Building contractors create new concrete mixtures without approval. Why should companies be allowed to create new APIs?
- ▶ Why should companies with buggy APIs be exempt from indemnity and consequential damages claims?

# External Controls

- ▶ The first API flaw to wipe out more than 10% of the world's PCs *will* result in legislation, guaranteed.
  - ▶ Legislating something we barely understand is not likely to help.
  - ▶ Forcing mission-critical APIs to become open source *will* help.
  - ▶ Whatever we do, we had better do it soon, or events will take any choice away.
- 