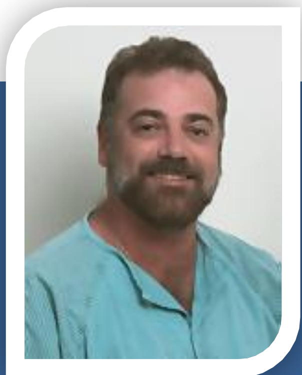
API Design Matters

Presented by
Danny Burakov

Communications of the ACM **52(5)**, pp. 46-56, May 2009



Michi Henning
Chief Scientist of ZeroC

- Prominent Australian software engineer
- 25+ years of experience
- Worked on ICE, CORBA, Unix kernel
- More info: *www.**triodia**.com*

API = Application Programming Interface

Bad APIs plague software engineering.

How do we get things right?

Why is API Design important **to you**?

- If you program, you are an API designer
  - Good code is modular; each module has an API
- Good APIs increase the pleasure and productivity of the developers who use them
- Thinking in terms of APIs improves code quality
- Designing a bad API can have a great cost

Good APIs are hard

- We recognize a good API when we use one
- Characteristics of a Good API:
  - Intuitive
    - Easy to learn
    - Easy to use (even without documentation)
  - Hard to misuse
    - Forces you to do the right thing
  - Easy to read and maintain code that uses it
  - Sufficiently powerful to satisfy requirements
  - Easy to evolve (to meet future requirements)
  - Well documented
  - Appropriate to audience

Why so many bad APIs?

- They're too easy to create.
  - APIs are provided once, but called many times
  - Minor design flaws get magnified
    - Problems show up at every point the API is called
  - Isolated flaws can interact with each other in surprisingly damaging ways
    - Lead to a lot of collateral damage

Example | `Select()` function

- .NET socket `Select()` function in C#

```
// API
public static void Select(List checkRead, List checkWrite,
                          List checkError, int microseconds);
```

- Typical use *(continued on next slide)*

```
// Server code
int timeout = ...;
ArrayList readList = ...; // Sockets to monitor for reading.
ArrayList writeList = ...; // Sockets to monitor for writing.
ArrayList errorList; // Sockets to monitor for errors.
```

Usage example

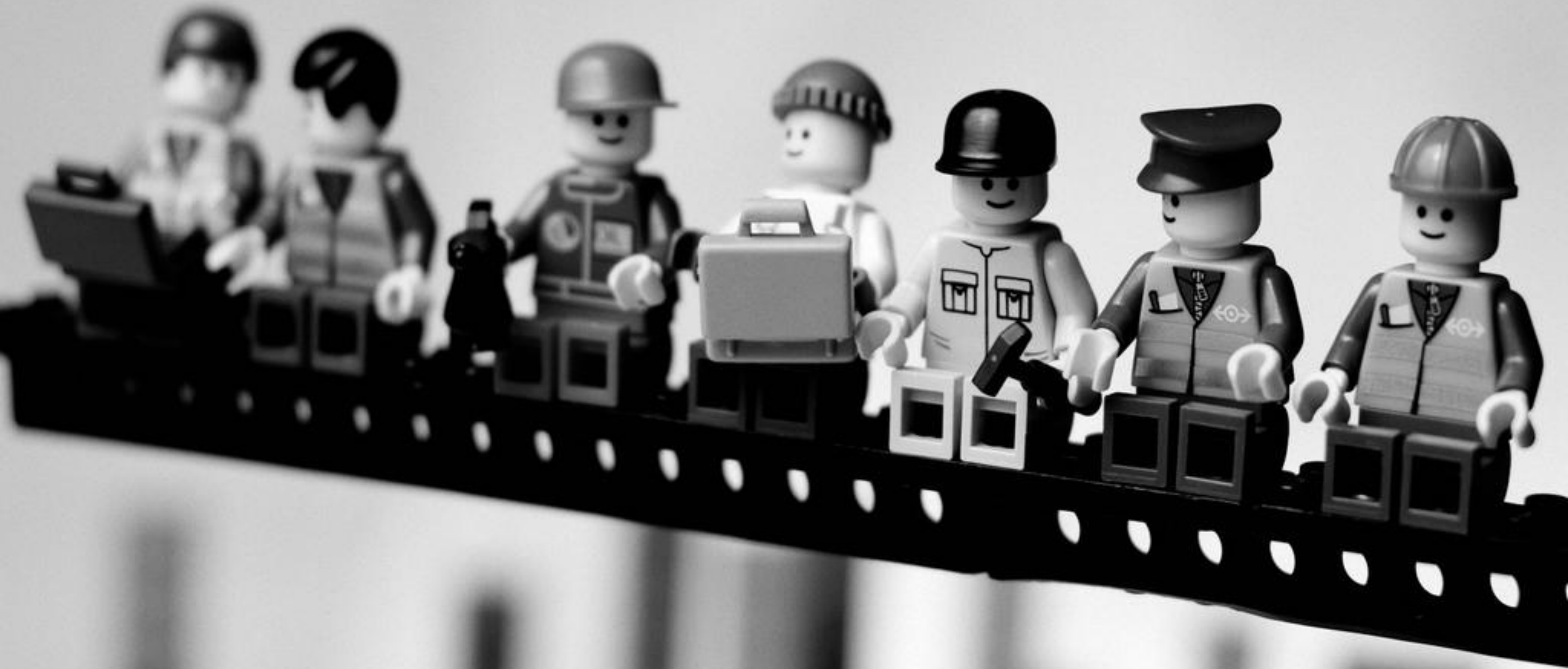
```
// Server code
while (!done) {

   SocketList readTmp  = readList.Clone();
   SocketList writeTmp = writeList.Clone();
   SocketList errorTmp = readList.Clone();

   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...
   }
}
```

The cost of poor APIs

Cause	Effect	Consequence
Require writing additional code	Makes programs larger, less readable and less efficient	
Difficult to understand and work with	Longer to write code	Increased development cost
Results in complex code	Higher likelihood for expensive, undetected bugs	Increased testing effort

- Cumulative cost easily runs to many *billions* of dollars

So, how to do better?



The 8 Guidelines
to always consider

1 Sufficient Functionality

- API must provide **sufficient** functionality for the caller to achieve its task.
- Insufficiency can go undetected
- Use a checklist of functionality

2 Smaller is Better

- API should be **minimal**.

  The fewer types, functions, and parameters an API uses –
  the easier it is to learn, remember, and use correctly

- Don't impose undue inconvenience on the caller

  Minimize non-fundamental 'convenience functions' –
  a function is worth adding only if it will be used frequently

- When in doubt - leave it out

  You can always add later to an API, but you can never remove

3 Understand the Context

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

Example

- Consider a string map (string pairs of key-value)
- *Lookup* method behavior if mapping is not set:
  - Throw a *VariableNotSet* exception
  - Return null
  - Return the empty string

4 General-purpose APIs should be "policy-free", Special-purpose APIs should be "policy-rich"

- APIs inevitably dictate policy
  - Dictates semantics, style
- Little known context – keep all options open
  - `Lookup()` should return null
- More known context – set more policy
  - Catches more compile-time errors
  - `Select()` fails this
- You cannot please everyone; make compromises
  - Displease everyone equally
  - Strategy design pattern is useful – caller-provided policies e.g. Comparator, Templates

5 Design from the perspective of the caller

- API is a **user** interface, just as much as GUI

- Example
  - `makeTV(false, true);`
  - `makeTV(Color, FlatScreen);`

- Let the customer write the function signature

- Design with needs of the caller in mind

  … even if it makes your job more complicated

6 Don’t “Pass the Buck”

- Don’t be afraid to set policy
  - A good API is clear about what it wants to achieve and what it doesn’t
  - "I should not pay for what I don't use"
- Don’t sacrifice usability on the altar of efficiency
  - It’s an illusion; caller does the dirty work instead of the API
  - `Select()` fails this…
- Is there anything I could reasonably do for the caller I am not doing?
  - If so, do I have valid reasons for not doing it?

7 Document **Before** You Implement

- Never forget: documentation is **part of the API**.
- Worst person to write documentation is the implementer, and worst time is after implementation
  - Implementer is mentally contaminated by the implementation
  - Tends to write what he or she has done
  - Too familiar with API, assumes some things are obvious
  - Misses important use cases
- Caller and implementer should iterate over the design
  - Neither caller nor implementation concerns are neglected
- The API should be tried out by someone unfamiliar with it
  - Check how much of the API can be understood without documentation

8 Good APIs are ergonomic

- Ergonomics are hard to pin-down
- Be Consistent
  - *(bad)* Example

```
char *strncpy(char *dst, char *src, size_t n);
void *bcopy  (void *src, void *dst, size_t n);
```

  - Use simple and uniform naming conventions for related tasks
  - Easier to use and memorize
  - Enables transference of learning
- Names matter – they should be largely self-explanatory
  - Good APIs read like prose

```
if (car.speed() > 2 * SPEED_LIMIT)
          speaker.generateAlert("Watch out for cops!");
```

  - Names are a good indication of how good your design is

API Change Requires Cultural Change

- We need to address the problem at its root
- Education
  - Recognition of the importance of the topic
- Career Path
  - Retain experienced programmers
  - Software designers should eat their own dog food
- External Controls – legislation, peer review
  - There are APIs whose correct functioning is of immense importance; any change in them incurs an enormous economic cost
  - Find the right balance between legislation and open peer review

Summary

- API is one of the most fundamental parts of programming
- Poorly designed APIs are as common as ever
- Guidelines for how to improve
- Look beyond the mere technical issues

Conclusions

- We lack a precise definition of a good API
- We need API design patterns
- It's impossible to please everyone
  - A good API is a subjective term
  - You have to know your audience
- We better start treating this issue more seriously
  - Serious mistakes in APIs can cause unprecedented damage
- **API Design truly matters** – we'd better realize it before we're left without choice

Thank you!

Questions, please

- How To Design A Good API and Why it Matters
  - Joshua Bloch, Google Tech Talks
  - *www.youtube.com/watch?v=aAb7hSCtvGw*
- API Design Wiki
  - *www.**apidesign**.org*

Additional Resources

Courtesy of Google