What goes wrong in APIs and how to fix it: API Design anti-patterns:

http://www.aleax.it/europ11_adap.pdf
When you do software...

...what do you focus on?

- functionality: "does a lot of useful stuff"
- correctness: "few if any serious bugs"
- performance: "what it does, it does fast"
- user interface: "looks & feels nice"
- security: "no easy target for hackers"
- elegance: "easy and smooth to maintain"

...what's missing from the above...?

BTW: all issues best achieved with agile, incremental development (except security)
This talk

- I'll tell you what I'm going to tell you
- then, I tell you
- finally, I tell you what I just told you

- now THIS is a classic Pattern, no auntie!-

- concrete examples: scarce, as I don't want to make (too many) more enemies
- except errors I made myself
- and stuff nobody could defend (Windows)
What's an API

"Application" (?) "Program Interface"

- a collection of functions, data types, protocols, events, &c, whereby any software system X can interact with a given software system Y
- Y can be: a library, framework, application, operating system, web site...
- Y's developers are typically responsible for designing and implementing the API
What's an Anti-pattern

- a category of counterproductive behaviors that are often, systematically repeated
- can be in: business processes, project management, design, programming, ...

an anti-pattern write-up should include:

- root causes (why did it seem a good idea at the time?)
- effects (why it's actually a bad idea)
- interactions (how it helps or hurts other patterns [or anti-patterns])
- remedies (suggestions for fixes)
API Design Antipatterns

- the very worst APIDA: not having any API
- the worst API's the one that's not there
- most frequent 1: "just didn't think of it"!
- the second worst: not having any design
- the API that "just happened"
- worse than most other "non-designs"
- Too many APIs spoil the broth
- "fear of commitment": to design→to choose
- inconsistency in APIs
- ...but wait, there's more...!
More kinds of APIDAs

Won't do these justice, either, but...:

"extremes": no balance between concerns

what language(s) to support?

excessive language dependence

excessive language INdependence

what about standard protocols/formats?

ignoring them blithely

rigidly, lavishly letting them "drive"

depbugging, error messages, documentation

performance-related APIDAs
The very worst APIDA

- the worst API is -- no API at all
- people just don’t think about APIs...!
- e.g. check stackoverflow: most common Qs
- spidering and scraping websites
- simulating keystroke & mouse gestures
- some of those Qs are about system testing
- most of them are about "missing APIs"
- the APIs may not actually be there
- ~ equivalent, they may be undocumented
Why "no API" is bad

- people DO need an API
- whether you supply it or not
- they're gonna "scrape" your UI
- or alleged-Ul;-)
- or, monkeypatch you (if no "hooks")
- useless extra load on your system
- rendering things that then get ignored
- makes their lives miserable
- every cosmetic change breaks their SW
- gives your "competitors" a nice entry!
What to do instead

- Offer an API! "Pick an API, any API"...
- Should be easy -- you ARE "in their shoes"
- Even a simple, weak one's better than none
- Document it! or, at least...:
  - to reduce workload, consider _examples_
  - may be easier than text to programmers
- Keep docs updated!
  - wrong docs can be worse than none
  - examples can be tested & should be
If you're an unwilling APIer

- Make life easy on yourself AND the users:
- Follow the yellow-brick road of the "path of least resistance": de facto standards
  - for web apps, REST & JSON
  - for Windows apps, COM
  - for Mac apps, "Applescript"
- ...wish there was ONE similarly broad answer for Linux apps too... ah well!–)
- doctest, for the my-only-docs-r-examples crowd (or as useful supplement to any doc)
The accidental API

- an "interface" that was never actually designed as such (also a "didn't think"...).
- designing an interface for proper program access is hard (though interesting) work
  (think of what you'd like to USE!)
- "but wait, we already have one!"...
  - often what the UI uses to the backend
  - sometimes the database schema
- "let's use that one -- look, ma, no work"!-)}
Why "no design" is bad API

- if you haven't designed what API you're exposing, specifically in order to expose it...
- ...then what you're exposing is not an API, but "internal implementation issues"!
- what happens when you want to change the implementation's details?
  - either you don't (→ forego improvements)
  - or you break your clients,
  - or you shoulder forever the burden of dual implementations (real one and API)...

What to do instead

THINK
...about your API
would you like to USE it...?

FORGET
...your implementation
definitely at least its details
THINK about your API!

"if I was an outside programmer, what would I want to be doing? And, how?"

"and why?" can't hurt, by the way;-)

don't just think; "walk a mile in their shoes" (e.g., your auxiliary scripts)

I dislike big-design-up-front, BUT...

...APIs are THE exception to this!

(well OK, security/privacy too;-)
Wtdi: Forget

- FORGET about your current implementation
- as an implementation, inevitably it's chock full of specific details, of course
- or, think about at least 2-3 alternative ones you might want to try in the future
- what's common to ALL?
- what changes w/every implementation is "an accident", irrelevant to your API
- what stays the same is "the substance", what really must be in the API
- the CONCEPTS your SW is about!
Wtdi: why?

- isn't this API design stuff a lot of work...?
- yes, some -- BUT...
  - the ROI on API design efforts is amazing!
    - not only does it enhance your API,
    - the insight it gives on your overall SW's design is hard to obtain otherwise
  +: a strong API helps you properly divvy up the system (as little as feasible "in the core", as much as can be, "outside"!)
- so you get better architecture too
A worst-case bad API...

"no API" / "expose the DB" worst case:
- accept SQL directly via URL or forms...
- (note the security/privacy connection...)
Too many cooks APIs

- zero APIs is too few, three is too many
- four is right out
- somewhere in-between, probably one (maybe two) is the sweet spot
- root cause #1: transition
  - esp. from a "not-really-designed" API
  - or, between technologies/platforms
- root cause #2: unwillingness to decide
  - cfr the "commitment issues" APIDAs
- root cause #3: org/project structure
Why "many APIs" is bad

- extra work to maintain all of them
- w/o real benefit to the user,
- w/o real benefit for refactoring either
- can often be confusing to the user (must learn them all and choose/pick one?)
- sometimes you can perform task A in one API, task B in another, but in neither can you do both needed tasks A and B
- of course, API transitioning/versioning is a very hard problem (no silver bullets...
What to do instead

"LAYERING" APIs is OK

-One lowest-level API -- exposing all the nuts and bolts of the system's logical architecture (NOT implementation!–)

-it's OK if it's user-unfriendly, hard to use, a little bit underdocumented, &c

-as long as it's full-power, high-performance, transparently debuggable

-because all OTHER APIs (one or more) are built entirely ON TOP OF the lowest-level one (no system internals involved)
API Transition: must plan!

- to err is human: you have a so-so API (or worse;-), have designed a better one, and want to transition all your users over to it
- "big-bang" transition (breaking all existing users) is right out: must take it in steps
- 1+ releases where using the old API still works -- but with clear, copious warnings
- tutorials & docs to help transitioning
- no new functionality in old API (motivate!)
- design to help transition? sometimes...
Fear of commitment

AKA, the "let's do both!" syndrome

...BUT OUR PRIMARY VENDOR CAN’T DELIVER, SO...

I WONDER WHAT’S ON TV TONIGHT.

...SHOULD WE RISK A LAWSUIT OR BUILD A PRODUCT THAT NOBODY ON EARTH WANTS?

DID HE ASK ME TO MAKE A CHOICE?

WILL IT BE A REQUEST FOR INFORMATION OR AN IMPractical SOLUTION?

LET’S DO BOTH!
The "Let's do both!" APIDA

- why is it bad?
  - to DESIGN is to DECIDE, i.e., to CHOOSE
  - oh boy, that's scary
    - am I going to be ACCOUNTABLE for it?
  - I am not worthy (to decide) -- do both
- management-structure / employee
  - empowerment problems (often, in firms)
- wishy-washy programmers (rare)
"fear to decide" example

HANDLE WINAPI CreateFile(
    __in    LPCTSTR lpFileName,
    __in    DWORD dwDesiredAccess,
    __in    DWORD dwShareMode,
    __in_opt LPSECURITY_ATTRIBUTES  
        lpSecurityAttributes,
    __in    DWORD dwCreationDisposition,
    __in    DWORD dwFlagsAndAttributes,
    __in_opt HANDLE hTemplateFile 
);  
    vs

int open(const char *pathname,
         int flags, mode_t mode);
gmpy as a bad example

```
import gmpy

x = gmpy.mpz(23)
y = x.whatever(45)

...or, equivalently...:

y = gmpy.whatever(23, 45)

we did fix in gmpy2 (not backwards comp;-)
```
To decide is human

Q to Ken Thompson:

"if you were to design Unix all over again from scratch, what would you change"?

Ken's A:

"I'd spell "creat" with a trailing E"

(better A: open is enough→no creat;-)

Perfection is not of this world

not an OK excuse for "not even trying"
What to do instead

- have the courage to choose
- choose to work in environments where failure (and honest acknowledgment of it, w/fixes) is not punished, but *encouraged*
  "fail -- but, fail fast!"
- AKA -- empowering environments
- you're human -- deal. You WON'T "get it right the first time". Be humble.
- Launch fast, and iterate
- "Rough consensus, and running code"!-)

[Image of a chalkboard with text written on it]
Inconsistency APIDAs

- argument ordering
  - foo(widget, value) vs bar(value, widget)
- lexical issues (under_score, MixedCase)
  - this_one(foo) vs TheOther(bar)
- nomina sunt consequentia rerum (verbs too)
  - RemoveThis/DeleteThat/EraseYonder/...
- plural vs singular: CommitTransaction vs RollbackTransactions (both w/1+ targets)
- SomeVeryDetailedSpecificName(x)/blah(y)
- acronyms: HttpConnect/HTTPSendQuery
Why inconsistency?

- too much Ralph Waldo Emerson?–)
- but that's against a _foolish_ consistency
- people, ideas change over time
- so do APIs
- maybe CommitTransaction used to take only one target, then grew to take 1+
- different people on the same project conceptualize (thus name) the same thing in slightly different (inconsistent) ways
What to do instead

- establish a "data dictionary" (not just "data": "verbs", too!), 1-1 mapping of words ↔ concepts in the SW system
- when a new concept arises, add it & the appropriate word in the DD _first_
  - before you name any API entry!
- cost: a little bit more work to coordinate
- advantages: not just to "external users" of the API -- like all coding conventions, once established, it _saves_ decision overhead!
In medio stat virtus

navigate very cautiously between pairs of "extreme" positions, esp. in underlying technology choices. For example:...

what prog. language(s) to support and how closely to adhere to their "style"s?
what protocols (esp. platform-standard or cross-platform standard ones) &c?

extremism is simpler, sharper, attractive...
but never works as well as balance, good taste, and moderation!–)
Proglanguage support

"sure we have an API... it's in BrainFork!"

whatever language(s) you've chosen to implement your SW system,

there's no good reason to foist it on everybody else who wants to use your system's API!

avoid language-specific data interchange formats in your API (for example, expose no Python "pickles"!!)

"you can program Fortran in any lang."

...but you shouldn't HAVE to!--)
Standard-protocol support

- don't invent Yet Another Data Format
- ain't JSON (or YAML) gud enuf 4 ya'?
- or CSV, FITS, HDF, netCDF, SAIF... (&c)?
- and Protocol Buffers, XML, ...
- you'd better have a darn good xcuse!-)

- if on the web, why not ReST? If not (yet)
  on the web, why not ReST _anyway_?
- on Windows, COM; on Mac, Applescript

- need more generality? RPC standards --
  CORBA, XMLRPC, ... -- why not those?
Debugging, errors, docs

- you make an API → somebody will (you hope!) be _developing_ with/on it
- they'll make mistakes; so will you
- good debugging support is a must
- open-source helps, does NOT suffice
- error msg "an error was encountered" (!)... 
- docs are hard to write, but precious
  - at least, provide COPIOUS examples!
  - and TEST them routinely (doctest)!
Performance issues

- A performance-incorrect API can kill performance in many, many ways, e.g.:
  - Excessive "make-work" in building / dismantling objects unnecessarily
  - Excessive "round trips" through lack of "batching" facilities
  - Improper support for threading/distrib.
  - No or inferior support for async use
    - E.g.: mandatory vs optional callbacks
    - Too-picky error-diag timing guarantees