# API Design Guidelines

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

# Outline

- Characteristics of a good API
- API design process
- API design guidelines

- Sources:
  - Stefan Nilsson: Thoughts on effective API design
    https://yourbasic.org/algorithms/your-basic-api/
  - Matt Gemmell: API Design,
    http://mattgemmell.com/api-design/
  - Jasmin Blanchette: The Little Manual of API Design
    http://people.mpi-inf.mpg.de/~jblanche/api-design.pdf

# Why is API design important to developers?

- If you program, you are an API designer
  - good software is modular
  - each module has an API
- Useful modules are reused
  - once it has users, it can’t be changed at will
- Following the API design principles improves code quality

# Characteristics of a Good API

# Easy to learn and memorize

- Consistent naming conventions and patterns
  - same name for the same concept
  - different names for different concepts
  - throughout the API
- A minimal API is easy to memorize
  - there is little to remember
- A consistent API is easy to memorize
  - you can reapply what you learned in one part of the API in a different part

# Easy to learn and memorize

- The semantics should be simple and clear
- Follow the principle of least surprise
  - If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature
- Don’t force boilerplate code on the user
  - a “hello world” sample shouldn’t be more than a few lines
- Convenience methods
  - they contribute to a bloated API
  - but they are accepted and encouraged if they are clearly documented and fit nicely with the API

# Leads to readable code

- Code is written once but is read over and over again
- Readable code is easier to document and to maintain
- It is less likely to contain bugs, since bugs are made more visible by the code's readability
- Readable code can be concise and verbose, it does not matter
- It is at the right abstraction level
  - it does not hide important things
  - doesn't force specifying irrelevant information

# Hard to misuse

- A good API makes it easier to write correct code than incorrect code
- It does not force the user to call methods in a specific order
- Users do not have to be aware of implicit side effects

# Easy to extend

- Libraries grow over time
  - new classes
  - new methods
  - new parameters
  - new enum values
- APIs should be designed with this in mind
- Also provide extensibility for users of the API

# Complete

- Ideally, an API should be complete and let users do everything they want
- In practice, this is never true
  - there will always be users who need something the API does not provide
- Therefore, it should be possible for users to extend and customize existing API
  - e.g. subclassing, overriding

# Well documented

- The public API of a library must be documented
  - the internal implementation doesn't need documentation, it can follow clean-code rules
- For each documented element:
  - What is it?
  - What does it do? (Not how it works!)
    - short sentence stating the purpose
    - a slightly longer explanation
    - and finally, all the details

# API Design Process

# 1. Know the requirements

- Collect the requirements
  - sometimes its easy: implement a standard
  - sometimes its hard: the requirements are unclear
- Ask as many people as possible
  - the most important is the target audience, the potential users
  - colleges, boss, etc.
- Sometimes you will receive solutions instead of requirements
  - be careful, better solutions may exist

# 2. Write use cases before you write any other code

- The wrong order of design is:
  - implement the functionality -> design API
- An API like this usually reflects the structure of the underlying code rather than meeting the users' needs
- The implementation should adapt the user not the other way around
- Before you start implementing the API write code snippets how you would like to use it
  - don't worry at this stage that it will be hard to implement
  - they will be good for samples and unit tests, too

# 3. Look for similar APIs in the same library

- Look if you can find a similar API
  - e.g. if you would like an XmlQuery, you can look at SqlQuery
  - the other library has probably evolved by a lot of feedback
- Mimic the similar API in your API
  - users will be comfortable with it already
  - you can adopt years of design decisions
- Of course, if the other API is bad, don't follow it
  - but not every API is that bad, there can be useful ideas
  - also if applications have to be ported from the old API, consider being backwards compatible and implement a superset of the old API

# 4. Define the API before you implement it

- Specify the API and its semantics before you implement it
- It is better for users if the API is straightforward and the implementation is tricky than the other way around
- As you implement the API or write unit tests you might find flaws or undefined parts in your API design
  - correct these in the design
  - but never let implementation details leak into the API
    - on-disk and on-the-wire formats, exceptions, tuning parameters
    - except for optimization options, but be careful with these

# 5. Have your peers review your API

- Ask feedback
  - potential users
  - colleges, boss
- Momentarily forget that it will be hard to implement
- Negative feedback is also useful
- Every opinion should be considered
- Keep the specification of the API short
  - 1 page is ideal
  - it is easier for people to read
  - it is easier for you to maintain

# 6. Write several examples against the API

- Write a few examples that use the API
- Start it early
  - even before you implement it
  - even before the specification
- Use the use-cases defined earlier
- It will also help if other people write examples and give you their feedback
- These examples will be good for documentation, samples and unit tests
- Keep writing examples as the API evolves
- Eat your own dog food
  - use your own API over an extended period of time on different types of tasks and projects to know if it really works as intended
  - meanwhile see if you can make improvements, simplify things

# 7. Prepare for extensions

- The API will be extended in two ways:
  - by the maintainers of the API
  - by the users of the API
    - through inheritance and polymorphism
- Service Provider Interface (SPI)
  - plug-in interface enabling multiple implementations
- Planning for extensibility requires going through realistic scenarios
  - for each class that can be overridden write at least 3 different subclasses (as examples or as public classes of the API) to ensure that it is powerful enough to support a wide range of requirements

# 8. Don’t publish internal APIs without review

- Sometimes APIs start as internal APIs
- Later they are published
- Make sure they are reviewed before publishing
  - e.g. misspellings

# 9. When in doubt, leave it out

- If you have doubts about a functionality
  - leave it out
  - or mark it as internal
  - you won't be able to please everyone
  - aim to displease everyone equally
- Wait for feedback from users
  - wait at least three independent users request something until you implement a new feature
- You can always add, but you can never remove
- Just say no
  - an API shouldn't encourage bad design decisions
- Expect to make mistakes
  - a few years of real-world use will flush them out
  - expect to evolve API

# 10. Don’t change it

- A software library needs to be backwards compatible
- It is OK to:
  - improve documentation
  - change the implementation
  - introduce new features
- But don’t change the API!
  - You will break other people’s code!
  - You need to get your API right at the very first attempt!
- Use semantic versioning: major.minor.patch
  - increments:
    - major: when an incompatible API change is made
    - minor: when a functionality is added in a backwards-compatible manner
    - patch: when backwards-compatible bug fixes are made

# API Design Guidelines

# G1. Choose self-explanatory names and signatures

- Pick names and signatures that are self-explanatory
- The names should read like prose
- The meaning of arguments should be clear at the call
  - beware of bool values
  - beware of lots of values of the same type after each other
  - consider using enums
  - consider using parameter objects
- Be consistent with the order of parameters
- Use the terminology of your audience
- Use meaningful names for parameters

# G2. Choose unambiguous names for related things

- Use consistent names
- Don’t use different names for the same thing
  - e.g. `Control-Widget`, `send-post`
- If similar concepts have to be distinguished, use different names for them

# G3. Beware of false consistency

- If you use a convention for something, then don't use this convention for something else
- Example:
  - prefix '`set`' is for setters
  - don't use this convention for methods other than setters

# G4. Avoid abbreviations

- Abbreviations are problematic because they have to be remembered what they mean and in what context
  - e.g. `GetWin` instead of `GetWindow`
- Exception: conventional abbreviations if they are obvious
  - `min`, `max`, `dir`, `prev`
  - just be consistent, and use the abbreviation everywhere
- Acronyms are OK, acronyms are not abbreviations
  - so use `XML` and not eXtensible Markup Language
  - also: `HTML`, `UML`, `IO`, etc.

# G5. Prefer specific names to general names

- Prefer specific names even if they seem to be too restrictive
- Once a name is taken, it cannot be used for something else
- Later if you would like to generalize you can pick general names
- Example: `GetLength` is better than `GetInt`

# G6. Use the local dialect

- APIs belong to a platform and a developer ecosystem
- Learn your target platform's conventions before coding
- Learn conventions for constructors, destructors, exceptions, method names, property names, memory management
- Use that terminology throughout your code
- Similarly, if you port an API to a new programming language, follow that language's conventions
  - the users of that language will be happy
  - the old users of the API won't mind

# G7. Don’t be a slave of an underlying API’s naming practices

- If you need to wrap an existing API, don’t hesitate to invent your own terminology
- The wrapped API may have poorly chosen names, or it clashes with the names of your library

# G8. Choose good defaults

- Choose good defaults so that the users won't have to copy and paste boilerplate code to get started
- Good defaults also make the API simple and predictable
- Name boolean options so that they are default to false
  - e.g. visible vs. hidden
- Document the defaults

# G9. Avoid making your APIs overly clever

- Don’t have a too clever API
- Avoid unexpected side effects
  - e.g. setText() automatically recognizes HTML text and also calls setTextFormat()

# G10. Consider performance consequences of API design decisions

- Bad decisions can limit performance
  - making a type mutable requires defensive copies
  - providing constructor instead of static factory
  - using implementation type instead of interface
- Do not warp API to gain performance
  - underlying performance issue will get fixed, but headaches will be with you forever
  - good design usually coincides with good performance

# G11. Pay attention to edge cases

- Edge cases are important, since other cases build on them
- Usually edge cases don't have to be handled separately, they are usually the neutral element of the operation
  - e.g. 1 for multiplication, 0 for addition, empty collection, etc.
- If you write extra code for edge cases, be suspicious
  - e.g. returning null instead of empty collection
- Write unit tests for the edge cases

# G12. Minimize mutability

- Classes should be immutable unless there's a good reason to do otherwise
  - advantages: simple, thread-safe, reusable
  - disadvantage: separate object for each value
- If mutable, keep state-space small, well-defined
  - make clear when it's legal to call which method
- Examples:
  - C#: DateTime
  - Java: Date

# G13. Design and document for inheritance or else prohibit it

- Inheritance violates encapsulation
  - Subclass sensitive to implementation details of superclass
- If you allow subclassing, document self-use
  - How do methods use one another?
- Conservative policy: all concrete classes final/sealed

# G14. Be careful when defining virtual APIs

- “Fragile base class problem”
- Difficult to get virtual base classes right
- Easy to break them between releases
- Mistakes:
  - too few virtual methods: cannot be reused
  - all methods are virtual: some methods are dangerous to reimplement
- Rules:
  - public methods should not be virtual
  - virtual methods should be protected
  - use template methods and don’t make them virtual

# G15. Strive for property-based APIs for GUI

- Don’t make users to initialize everything in constructors with long parameter list
  - only the required settings should be constructor parameters
- Use setters instead
  - choose appropriate default values so that the user only has to explicitly change what they need
  - the user doesn’t need to remember the order of the values
  - the code is more readable, since it is visible, which property has which value
  - the values of the properties can be retrieved by getters
  - use lazy initialization because the properties can be set in any order

# G16. Anticipate customization scenarios

- User interfaces should have sensible defaults to fit the majority
  - don’t give the user options
- However, APIs should be customizable to be flexible
  - if you publish properties for customization, also publish related properties
    - e.g. ForegroundColor, BackgroundColor
- Decide what options will serve 70% or so of the usage situations you can think of, and provide those options
- Better to have a simple API with few options than a complicated API with a lot of options

# G17. Avoid long parameter lists

- Three or fewer parameters is ideal
  - more and users will have to refer to docs
- Long lists of identically typed parameters harmful
  - programmers transpose parameters by mistake
  - programs still compile, run, but misbehave
- Two techniques for shortening parameter lists
  - break up method
  - create parameter object to hold parameters

# G18. Use convenience methods

- Implementing the API as a developer you may add convenience methods
- They are usually kept hidden from the public interface of the API
- Consider publishing these convenience methods
  - if they are convenient for you, they may prove to be convenient for others, too
- Example:
  - Opening a file for reading:
    - C#: reader = new StreamReader("file.txt")
    - Java: reader = new BufferedReader(new FileReader("file.txt"))
  - Reading the contents of a file:
    - C#: reader.ReadToEnd()
    - Java: new String(Files.readAllBytes(Path.get("file.txt")))

# G19. Get up and running in 3 lines

- The API should be able to used in 3 lines:
  - instantiation
  - basic configuration, setting properties
  - execution
- Anything more and the API has too much boilerplate code
  - boilerplate code spreads with copying (e.g. StackOverflow)

# G20. Magic is OK. Numbers aren't.

- It is possible that you need special values to implement the API
- It is also possible that you have to apply tricks and magic in the implementation to keep the API nice for the users
- But never publish magic with the API
- Wrap special values in meaningful named constants, enumerations, etc.

# G21. Fail fast – report errors as soon as possible after they occur

- Compile time is best - static typing, generics
- At runtime, first bad method invocation is best
  - check pre-conditions at the beginning of the methods
    - e.g. null checks, valid value ranges
  - methods should be failure-atomic
- Throw exceptions to indicate exceptional conditions
  - don’t force client to use exceptions for control flow
  - conversely, don’t fail silently

# G22. Favor unchecked exceptions

- Checked: client must take recovery action
- Unchecked: programming error
- Overuse of checked exceptions causes boilerplate
- Checked exceptions violate OCP
- Include failure-capture information in exceptions
  - allows diagnosis and repair or recovery
  - for unchecked exceptions: message suffices
  - for checked exceptions: provide accessors

# G23. Include resolution in the exception message

- Avoid exceptions with no messages
- Avoid leaking null-pointer exceptions
- Avoid swallowing exceptions
- Describe the problem in the exception message
- Provide meaningful description of the problem
- Provide resolution message: how to fix the problem
- Don’t include sensitive information in exceptions
  - exceptions are usually logged
- Example:
  - Unhandled Exception: System.InvalidOperationException: Service 'Services.Company' has zero application (non-infrastructure) endpoints. This might be because no configuration file was found for your application, or because no service element matching the service name could be found in the configuration file, or because no endpoints were defined in the service element.

# G24. Overload with care

- Avoid ambiguous overloads
  - multiple overloads applicable to same actuals
  - conservative: no two with same number of args
- Just because you can doesn't mean you should
  - often better to use a different name
- If you must provide ambiguous overloads, ensure same behavior for same arguments

# G25. Test the hell out of it

- Test your API
- Use the use-cases and samples written earlier
- Write unit tests and regression tests
- Write tests for the edge cases
- Don’t publish a buggy API
  - users won’t trust a buggy API
  - they may soon abandon it

# G26. Document your API

- An API without documentation is painful to use
  - e.g. a lot of open source libraries
- Documentation in comments is a useful way to do this
  - it should be sufficiently detailed but not too verbose
  - class: what an instance represents
  - method: contract between method and its client
    - pre-conditions, post-conditions, side-effects
  - parameter: indicate units, form, ownership
  - document default values
  - document state space
- Provide HTML and PDF documentation
  - HTML is easier to browse
  - PDF can be easier to read in order

# Summary

# Summary

- Characteristics of a good API
- API design process
- API design guidelines

- Sources:
  - Stefan Nilsson: Thoughts on effective API design
    https://yourbasic.org/algorithms/your-basic-api/
  - Matt Gemmell: API Design,
    http://mattgemmell.com/api-design/
  - Jasmin Blanchette: The Little Manual of API Design
    http://people.mpi-inf.mpg.de/~jblanche/api-design.pdf