Principles of Software Construction: Objects, Design, and Concurrency

API Design, Part II: Principles

Charlie Garrod **Chris Timperley**

Carnegie Mellon University
School of Computer Science

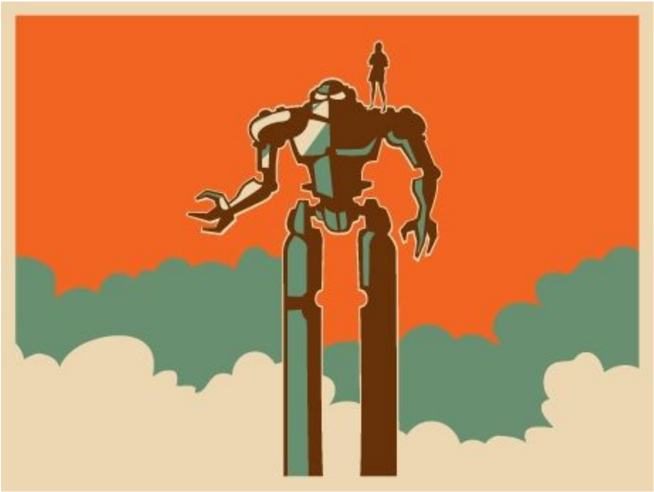
institute for SOFTWARE RESEARCH

Administrivia

- Homework 4c due on Thursday
  - Up to 75% of points lost on Homework 4a can be recovered by submitting revised design documents.
- Homework 5 coming soon!
  - Team sign-up sheet will be posted to Piazza later this week.
- Midterm exam in class next Thursday (31st October)
  - Review session details will be posted soon.

Key concepts from Thursday...

- APIs took off in the past thirty years, and gave programmers super-powers
- Good APIs are a blessing; bad ones, a curse
- Using a design process greatly improves API quality



Characteristics of a Good API

Review

Characteristics of a Good API

Review

- Easy to learn
- Easy to use, even if you take away the documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

Key concepts from Thursday...

- Naming is critical to API usability

Tips for better names

Review

Tips for better names

Review

- Start with *meaning* and *intention*
- Avoid ambiguous terms with multiple meanings
- Avoid terms with the same meaning -- use one term
  - `remove` and `delete`
- Prefer shorter names
- Avoid acronyms and abbreviations
- Read the code to make sure that it *sounds* okay
- Ask for feedback on your names!
- ...

https://www.youtube.com/watch?v=SctS56YQ6fg

Outline

- General principles (8)
- Class design (5)
- Method design (9)
- Exception design (4)
- Documentation

1. API Should Do One Thing and Do it Well

- Functionality should be easy to explain
  - **If it's hard to name, that's generally a bad sign**
  - Be amenable to splitting and merging modules

Good: `Font`, `Set`, `PrivateKey`, `Lock`,
`ThreadFactory`, `TimeUnit`, `Future<T>`

Bad: `DynAnyFactoryOperations`,
`_BindingIteratorImplBase`,
`ENCODING_CDR_ENCAPS`, `OMGVMCID`

https://assets.victorinox.com/mediahub/33790/640Wx560H/SAK_1_4713__S1.jpg

What not to do

```
public abstract class Calendar implements
Serializable, Cloneable, Comparable<Calendar>
```

The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week. An instant in time can be represented by a millisecond value that is an offset from the Epoch, January 1, 1970 00:00:00.000 GMT (Gregorian).

What not to do, continued

Like other locale-sensitive classes, Calendar provides a class method, getInstance, for getting a generally useful object of this type. Calendar's getInstance method returns a Calendar object whose calendar fields have been initialized with the current date and time:

```
Calendar rightNow = Calendar.getInstance();
```

A Calendar object can produce all the calendar field values needed to implement the date-time formatting for a particular language and calendar style (for example, Japanese-Gregorian, Japanese-Traditional). Calendar defines the range of values returned by certain calendar fields, as well as their meaning. For example, the first month of the calendar system has value MONTH == JANUARY for all calendars. Other values are defined by the concrete subclass, such as ERA. See individual field documentation and subclass documentation for details.

etc., etc., etc., etc., etc., etc., etc., etc.

What is a `Calendar` instance? What does it do?

- **I have no clue!!!**
- The confusion, bugs, and pain caused by this class are incalculable
- Thankfully it’s obsolete as of Java 8; use `java.time`
- Inexplicably, it’s not deprecated, even as of Java 10
- If you working on an API and you see a class description that looks like this, run screaming!

2. API should be as small as possible but no smaller

"Everything should be made as simple as possible, but not simpler." – Einstein

- API must satisfy its requirements
  - Beyond that, more is not necessarily better
  - But smaller APIs sometimes solve more problems!
  - **Generalizing an API can make it smaller**
- **When in doubt, leave it out**
  - Functionality, classes, methods, parameters, etc.
  - **You can always add, but you can never remove**

Conceptual weight (a.k.a. conceptual surface area)

- **Conceptual weight** more important than "physical size"
- **The number and difficulty of new concepts in API**
- Examples where growth add little conceptual weight:
  - Adding overload that behaves consistently with existing methods
  - Adding new static methods to a utility class
  - Adding `arccos` when you already have `sin`, `cos`, and `arcsin`
- Look for a **high *power-to-weight ratio***
  - In other words, look for API that lets you do a lot with a little

Example: generalizing an API can make it smaller

Subrange operations on `Vector` *– legacy List implementation*

```
public class Vector {
    public int indexOf(Object elem, int index);
    public int lastIndexOf(Object elem, int index);
    ...
}
```

- Not very powerful
  - Supports only search operation, and only over certain ranges
- Hard to use without documentation
  - What are the semantics of `index`?
  - I don't remember, and it isn't obvious.

Example: generalizing an API can make it smaller

Subrange operations on `List`

```
public interface List<T> {
    List<T> subList(int fromIndex, int toIndex);
    ...
}
```

- Extremely powerful!
  - **Supports *all* List operations on *all* subranges**
  - Returned list is a *view* of receiver, not a copy
- Easy to use even without documentation

3. Don't make users do anything library could do for them

APIs should exist to serve their users and not vice-versa

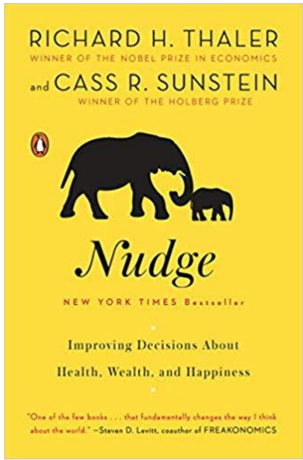
- Reduce need for **boilerplate code**
  - Generally done via cut-and-paste
  - Ugly, annoying, and error-prone

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// DOM code to write an XML document to a specified output stream.
static final void writeDoc(Document doc, OutputStream out) throws IOException {
  try {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
                        doc.getDoctype().getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out));
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can't happen!
  }
}
```

4. Make it easy to do what's common and preferable, possible to do what's less-common

- People tend to take the easy way out
  - Make sure it's what they want and should do
- If it's hard to do what they want, they'll get upset
  - They'll have to go to the documentation
- **If it's easier to do something wrong, dangerous, or expensive, that's exactly what users will do**
- Don't worry too much about truly rare stuff
  - It's OK if your API doesn't handle it, at least in the first release



5. Monitor complexity constantly

- Train yourself to run "complexity meter" in the background
- When it starts climbing, look for ways to simplify

6. Implementation should not impact API

- Natural human tendency to design what you know how to implement – fight it!
  - Design for the user; then figure out how to implement
- Implementation constraints may change; API won't
  - When this happens, API becomes unexplainable
- When platform and domain are at odds, choose domain
  - e.g., chess algebraic notation vs. zero-based indexing

7. APIs should coexist peacefully with platform

- Do what is customary
  - Obey standard naming conventions
  - Avoid obsolete parameter and return types
  - Mimic patterns in core APIs and language
- Take advantage of API-friendly features
  - varargs, enums, iterables, `try`-with-resources, default methods, etc.
- Don’t Transliterate APIs
  - Common in the early days (Corba, JGL, etc.)
  - Still happens

8. Consider the performance consequences of API design decisions

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil.** Yet we should not pass up our opportunities in that critical 3%."

Donald Knuth

8. Consider the performance consequences of API design decisions

- **Bad API decisions can limit performance forever**
  - Making type inappropriately mutable (or immutable!)
  - Providing public constructor instead of static factory
  - Using implementation type instead of interface
- **But 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

8. Consider the performance consequences of API design decisions

- `Component.getSize()` returns `Dimension`
- `Dimension` is mutable
- Each `getSize` call must allocate `Dimension`
- Causes millions of needless object allocations!

getSize

`public Dimension getSize()`

Returns the size of this component in the form of a `Dimension` object. The `height` field of the `Dimension` object contains this component's height, and the `width` field of the `Dimension` object contains this component's width.

Returns:

`a Dimension object that indicates the size of this component`

Since:

`JDK1.1`

See Also:

`setSize(int, int)`

Outline

- General principles (8)
- Class design (5)
- Method design (9)
- Exception design (4)
- Documentation

1. Don’t expose a new type that lacks meaningful contractual refinements on an existing supertype

- Just use the existing type
- Reduces conceptual surface area
- Increases flexibility
- Resist the urge to expose type just because it’s there

2. Minimize Mutability

- Parameters should be immutable
  - Eliminates need for defensive copying
- 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

Bad: `Date`, `Calendar`, `Thread.interrupt`

Good: `BigInteger`, `Pattern`, `Matcher`

3. Minimize accessibility of everything

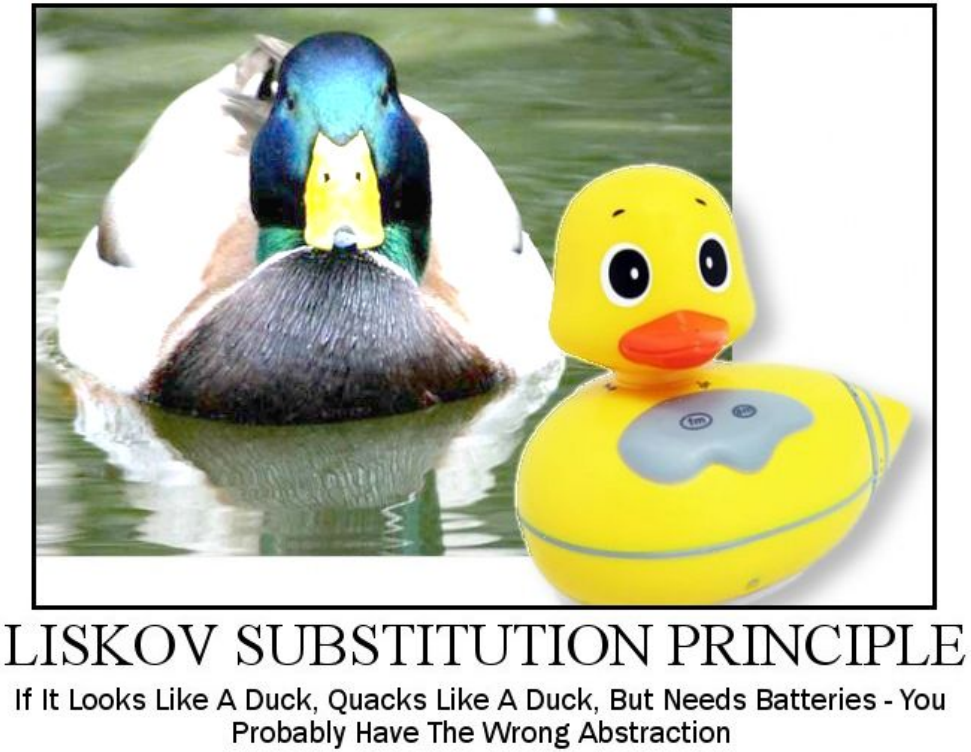
- Make classes, members as private as possible
  - If it's at least package-private, it's not a part of the API
- Public classes should have no public fields (with the exception of constants)
- Maximizes *information hiding* [Parnas72]
- Minimizes *coupling*
  - Allows components to be, understood, used, built, tested, debugged, and optimized independently

4. Subclass only when an is-a relationship exists

- Subclassing implies substitutability (Liskov)
  - Makes it possible to pass an instance of subclass wherever superclass is called for
  - And signals user that it's OK to do this
- If not is-a but you subclass anyway, all hell breaks loose
  - Bad: `java.util.Properties`, `java.util.Stack`
- Never subclass just to reuse implementation
- Ask yourself "Is every Foo really a Bar?"
  - If you can't answer yes with a straight face, don't subclass!

4. Subclass only when an is-a relationship exists

- Ask yourself “Is every Foo really a Bar?”
  - If you can’t answer yes with a straight face, don’t subclass!



https://miro.medium.com/proxy/0*r3w5f6R_xSk-f4PH.png

5. Design & document for inheritance or else prohibit it

- Inheritance violates encapsulation (Snyder, ’86)
  - Subclasses are sensitive to implementation details of superclass
- **If you allow subclassing, document *self-use***
  - How do methods use one another?
- Conservative policy: all concrete classes uninheritable
- See *Effective Java* Item 19 for details

Bad: Many concrete classes in J2SE libraries

Good: **`AbstractSet`**, **`AbstractMap`**

Outline

- The Process of API Design
- General Principles (8)
- Class Design (5)
- Method Design (9)
- Exception Design (4)

1. “Fail Fast” – prevent failure, or fail quickly, predictably, and informatively

- API should make it impossible to do what’s wrong
  - Fail at compile time or sooner
- Misuse that’s statically detectable is second best
  - Fail at build time, with proper tooling
- Misuse leading to prompt runtime failure is third best
  - Fail when first erroneous call is made
  - Method should be *failure-atomic*

Misuse that's statically detectable (and fails promptly at runtime if it eludes static analysis)

```
// The WRONG way to require one or more arguments!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Need at least 1 arg");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

API that makes it impossible to do what's wrong

```
// The right way to require one or more arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

Won't compile if you try to invoke with no arguments

- No validity check necessary
- Works great with for-each loop

1. “Fail Fast” – prevent failure, or fail quickly, predictably, and informatively

- Misuse that can lie undetected is nightmarish!
  - Fail at an undetermined place and time in the future

https://pbs.twimg.com/profile_images/632821853627678720/zPKK7jql_400x400.png

API that fails at an unknown time and place (***)

Sweet dreams…

```
// A Properties instance maps strings to strings
public class Properties extends Hashtable {
    public Object put(Object key, Object value);

    // Throws ClassCastException if this properties
    // contains any keys or values that are not strings
    public void save(OutputStream out, String comments);
}
```

2. Handle boundary conditions (edge cases, corner cases) gracefully

- Client should not have to write extra code
  - These cases should just work
- e.g., Return zero-length arrays, collections; not `null`

```
package java.awt.image;

public interface BufferedImageOp {
    // Returns the rendering hints for this operation,
    // or null if no hints have been set.
    public RenderingHints getRenderingHints();
}
```

- If client *must* treat boundary cases differently, use `Optional<T>`, checked exception, or some such

3. Use appropriate parameter and return types

- Favor interface types over classes for input
  - Provides flexibility, performance
- Use most specific reasonable input parameter type
  - Moves error from runtime to compile time
- **Don't use `String` if a better type exists**
  - Strings are cumbersome, error-prone, and slow
- Don't use floating point for monetary values
  - Binary floating point causes inexact results!
- Use `double` (64 bits) rather than `float` (32 bits)
  - Unless you **know** (via benchmarking) that you need the performance, and you can tolerate the low precision

4. Use consistent parameter ordering across methods

- Especially important if parameter types identical

```
#include <string.h>
char *strncpy(char *dst, char *src, size_t n);
void  bcopy  (void *src, void *dst, size_t n);
```

- Also important if parameter types “overlap,” e.g., (`int`, `long`) can hurt you if you pass two `int` values

`java.util.collections` – first parameter always collection to be modified or queried

`java.util.concurrent` – time always specified as `long delay, TimeUnit unit`

5. Avoid long parameter lists

- **Three or fewer parameters is ideal**
  - More and users will have to refer to docs
- Long lists of identically typed params are very harmful
  - Programmers transpose parameters by mistake
  - Programs still compile, run, but misbehave!
- Techniques for shortening parameter lists
  - Break up method
  - Create helper class to hold several parameters
    - Often they're otherwise useful, e.g., `Duration`
  - Use builder pattern

```
// Eleven (!) parameters including four consecutive ints
HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName,
      DWORD dwStyle, int x, int y, int nWidth, int nHeight,
      HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
      LPVOID lpParam);
```

6. Avoid return values that demand exceptional processing

- Return zero-length array or empty collection, not **`null`**

```
package java.awt.image;

public interface BufferedImageOp {
  // Returns the rendering hints for this operation,
  // or null if no hints have been set.
  public RenderingHints getRenderingHints();
}
```

7. Do not overspecify the behavior of methods

- Don't specify internal details
  - It's not always obvious what's an internal detail
- All tuning parameters are suspect
  - Let client specify intended use, not internal detail
  - Good: intended size; Bad: number of buckets in table
  - Good: intended concurrency level; Bad: number of shards
- Do not let internal details "leak" into spec
  - e.g., by propagating inappropriate exceptions
- Do not specify hash functions!
  - You lose the flexibility to improve them

8. Provide programmatic access to all data available in string form

- Otherwise, clients will be forced to parse strings
  - Painful for clients
  - Error prone
  - **Worst of all, it turns string format into de facto API**

8. Provide programmatic access to all data available in string form

Java got this wrong for exception stack traces…

```
public class Throwable {
  public void printStackTrace(PrintStream s);
  public StackTraceElement[] getStackTrace(); // Since 1.4
}

public final class StackTraceElement {
  public String getFileName();
  public int getLineNumber();
  public String getClassName();
  public String getMethodName();
  public boolean isNativeMethod();
}
```

9. Overload with care

- Avoid *ambiguous overloadings*
  - Multiple overloadings applicable to same actuals
- Just because you can doesn't mean you should
  - **Often better to use a different name**
  - But overloadings that really do the same thing for different types are a good thing; they reduce conceptual weight
    - Especially true for primitive types and arrays in Java
- If you must provide ambiguous overloadings, ensure same behavior for same arguments

```
// Bad – ambiguous overloading with different behaviors
public TreeSet(Collection<E> c); // Ignores order
public TreeSet(SortedSet<E> s);  // Respects order
```

Outline

- General principles
- Class design (8)
- Method design (5)
- Exception design (9)
- Documentation (4)

1. Throw exceptions to indicate exceptional conditions

- Don’t force client to use exceptions for control flow

```
private byte[] a = new byte[CHUNK_SIZE];
void processBuffer(ByteBuffer buf) {
  try {
    while (true) {
      buf.get(a);
      processBytes(a, CHUNK_SIZE);
    }
  } catch (BufferUnderflowException e) {
    int remaining = buf.remaining();
    buf.get(a, 0, remaining);
    processBytes(a, remaining);
  }
}
```

- Conversely, don’t fail silently

```
void threadGroup.enumerate(Thread[] list);
```

2. Favor unchecked exceptions

- Checked – client must take recovery action
- Unchecked – generally a programming error
- Overuse of checked exceptions causes boilerplate

```
try {
    Foo f = (Foo) super.clone();
    ....
} catch (CloneNotSupportedException e) {
    // This can't happen, since we’re Cloneable
    throw new AssertionError();
}
```

3. Favor the reuse of existing exception types

- Especially `IllegalArgumentException` and `IllegalStateException`
- Makes APIs easier to learn and use
- Extend existing exceptions if you need extra methods
  - e.g., IllegalNameArgumentException, getName()

4. Include failure-capture information in exceptions

- e.g., `IndexOutOfBoundsException` should include index and, ideally bound(s) of access
  - In early releases, it didn't; now it includes index
  - Added to detail message for arrays ca. JDK 1.1
  - `public IndexOutOfBoundsException(int index)` added in Java 9(!)
- Eases diagnosis and repair or recovery
- For unchecked exceptions, message suffices
- For checked exceptions, provide accessors too

Outline

- General principles
- Class design
- Method design
- Exception design
- Documentation

API documentation is critical

Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.

– D. L. Parnas, *1994*
In Brooks's *The Mythical Man Month, Anniversary Edition*

Document religiously

- Document **every** class, interface, method, constructor, parameter, and exception
  – Class: what an instance represents
  – Method: contract between method and its client
    - Preconditions, postconditions, side-effects
  – Parameter: indicate units, form, ownership
- Document thread safety
- If class is mutable, document state space
- If API spans packages, JavaDoc is *not* sufficient
  – Remember the collections framework?

API Design Summary

- A good API is a blessing; a bad one a curse
- API Design is hard
  - Accept the fact that we all make mistakes
  - But do your best to avoid them
- This talk and the last covered some heuristics of the craft
  - Don't adhere to them slavishly, but...
  - Don't violate them without good reason