Principles of Software Construction: Objects, Design, and Concurrency

API Design, Part I: Process and Naming

Charlie Garrod  Chris Timperley
Administrivia

- Homework 4c due next Thursday
- Reading assignment due next Tuesday
  - Effective Java, Items 6, 7, and 63
Review: libraries, frameworks both define APIs

```java
public MyWidget extends JContainer {
    public MyWidget(int param) {
        // setup internals, without rendering
        // render component on first view and resizing
        protected void paintComponent(Graphics g) {
            // draw a red box on his component
            Dimension d = getSize();
            g.setColor(Color.red);
            g.drawRect(0, 0, d.getWidth(), d.getHeight());
        }
    }
}
```

your code
The next two lectures: API design

• An API design process
• The key design principle: information hiding
• Concrete advice for user-centered design

Based heavily on "How to Design a Good API and Why it Matters" by Josh Bloch.
“Time for Change” (2002)

If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```java
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```
What does it print?

public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}

(a) 0.9  
(b) 0.90 
(c) It varies 
(d) None of the above
What does it print?

(a) 0.9
(b) 0.90
(c) It varies
(d) None of the above: 0.8999999999999999

Decimal values can't be represented exactly by float or double
Another look

```java
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```
How do you fix it?

// You could fix it this way...
import java.math.BigDecimal;
public class Change {
    public static void main(String args[]) {
        System.out.println(
            new BigDecimal("2.00").subtract(
                new BigDecimal("1.10")));
    }
}

// ...or you could fix it this way
public class Change {
    public static void main(String args[]) {
        System.out.println(200 - 110);
    }
}
The moral

• Avoid float and double where exact answers are required
  – For example, when dealing with money
• Use BigDecimal or long instead
2. “A Change is Gonna Come”

If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```java
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```
What does it print?

(a) 0.9
(b) 0.90
(c) 0.8999999999999999
(d) None of the above

import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
What does it print?

(a) 0.9
(b) 0.90
(c) 0.8999999999999999
(d) None of the above:

0.899999999999999911182158029987476766109466552734375

We used the wrong BigDecimal constructor
What’s going on here?

The spec says:

```java
public BigDecimal(double val)
```

Translates a double into a BigDecimal which is the exact decimal representation of the double's binary floating-point value.

```java
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```
How do you fix it?

```java
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal("2.00");
        BigDecimal cost = new BigDecimal("1.10");
        System.out.println(payment.subtract(cost));
    }
}
```

Prints `0.90`
The moral

• Use `new BigDecimal(String)`, not `new BigDecimal(double)`
• `BigDecimal.valueOf(double)` is better, but not perfect
  – Use it for non-constant values.
  – Uses canonical string representation to construct decimal
• For API designers
  – Make it easy to do the commonly correct thing
  – Make it hard to misuse
  – Make it possible to do exotic things
Fundamental Design Principle for Change:
Information Hiding

- Expose as few implementation detail as necessary
- Allows implementation to be changed at a later date

**Service** implementation

Hidden from service* provider

Hidden from service* client

Service* interface

Client environment

* service = object, subsystem, …
Why create a public API?
Good APIs can be a great asset!

• Distributed development among many teams
  – Incremental, non-linear software development
  – Facilitates communication

• Long-term buy-in from clients & customers
  – Users invest heavily: acquiring, writing, learning
  – Cost to stop using an API can be prohibitive
  – Successful public APIs capture users
Poor APIs can be a great liability!

- Lost productivity from your software developers
- Wasted customer support resources
- Lack of buy-in from clients & customers

Public APIs are forever

- Your code
- Your colleague
- Another colleague
- ...
Public APIs are forever

Diagram:
- Eclipse (IBM)
  - JDT Plugin (IBM)
  - CDT Plugin (IBM)
  - ...

...
Hyrum’s Law

With a sufficient number of users of an API, it does not matter what you promised in the contract, all observable behaviors of your interface will be depended upon by somebody.

† Named after Hyrum Wright, Software Engineer at Google
Today’s topic: API Design

Review: what is an API?

- Short for Application Programming Interface
- Component specification in terms of operations, inputs, & outputs
  - Defines a set of functionalities independent of implementation
- Allows implementation to vary without compromising clients
- Defines component boundaries in a programmatic system
- A public API is one designed for use by others
Exponential growth in the power of APIs

This list is approximate and incomplete, but it tells a story

’50s–’60s – Arithmetic. Entire library was 10-20 calls!

’70s – malloc, bsearch, qsort, rnd, I/O, system calls, formatting, early databases

’80s – GUIs, desktop publishing, relational databases

’90s – Networking, multithreading, 3D graphics

’00s – **Data structures(!),** higher-level abstractions, Web APIs: social media, cloud infrastructure

’10s – Machine learning, IOT, robotics, pretty much everything
What the dramatic growth in APIs has done for us

• Enabled code reuse on a grand scale
• Increased the level of abstraction dramatically
• A single programmer can quickly do things that would have taken months for a team
• What was previously impossible is now routine
• APIs have given us super-powers
Why is API design important?

• A good API is a joy to use; a bad API is a nightmare

• APIs can be among your greatest assets
  – Users invest heavily: acquiring, writing, learning
  – Cost to **stop** using an API can be prohibitive
  – Successful public APIs capture users

• APIs can also be among your greatest liabilities
  – Bad API can cause unending stream of support calls
  – Can inhibit ability to move forward

• **Public APIs are forever** – one chance to get it 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
• Useful modules tend to get reused
  – Good reusable modules are an asset
  – Once module has users, can’t change API at will
• Thinking in terms of APIs improves code quality
Characteristics of a good API

• Easy to learn
• Easy to use, even without documentation
• Hard to misuse
• Easy to read and maintain code that uses it
• Sufficiently powerful to satisfy requirements
• Easy to evolve
• Appropriate to audience
Outline

• The Process of API Design
• Naming
• Documentation
Gather requirements—skeptically

- Often you’ll get proposed solutions instead
  - Better solutions may exist
- Your job is to extract true requirements
  - Should take the form of use-cases
- Can be easier & more rewarding to build more general API

What they say: “We need new data structures and RPCs with the Version 2 attributes”
What they mean: “We need a new data format that accommodates evolution of attributes”
An often overlooked part of requirements gathering

• Ask yourself if the API **should** be designed
• Here are several good reasons **not** to design it
  – It’s superfluous
  – It’s impossible
  – It’s unethical
  – The requirements are too vague
• If any of these things are true, **now** is the time to raise red flag
• If the problem can’t be fixed, fail fast!
  – The longer you wait, the more costly the failure
Start with short spec – 1 page is ideal

• At this stage, agility trumps completeness
• Bounce spec off as many people as possible
  – Listen to their input and take it seriously
• If you keep the spec short, it’s easy to modify
• Flesh it out as you gain confidence
Sample early API draft

// A collection of elements (root of the collection hierarchy)
public interface Collection<E> {

    // Ensures that collection contains o
    boolean add(E o);

    // Removes an instance of o from collection, if present
    boolean remove(Object o);

    // Returns true iff collection contains o
    boolean contains(Object o);

    // Returns number of elements in collection
    int size();

    // Returns true if collection is empty
    boolean isEmpty();

    ... // Remainder omitted
}
Write to your API early and often

• Start *before* you’ve implemented the API
  – Saves you doing implementation you'll throw away

• Start *before* you’ve even specified it properly
  – Saves you from writing specs you'll throw away

• Continue writing to API as you flesh it out
  – Prevents nasty surprises right before you ship

• Code lives on as examples, unit tests
  – *Among the most important code you’ll ever write*
  – Forms the basis of *Design Fragments*
    [Fairbanks, Garlan, & Scherlis, OOPSLA ‘06, P. 75]
Try API on at least 3 use cases before release

• If you write one, it probably won’t support another
• If you write two, it will support more with difficulty
• If you write three, it will probably work fine
• Ideally, get different people to write the use cases
  – This will test documentation & give you different perspectives
• This is even more important for plug-in APIs
• Will Tracz calls this “The Rule of Threes”
  (Confessions of a Used Program Salesman, Addison-Wesley, 1995)
Maintain realistic expectations

• Most API designs are over-constrained
  – You won't be able to please everyone – *don’t try!*
  – Come up with a unified, coherent design that represents a compromise
  – It can be hard to decide which “requirements” are important

• Expect to make mistakes
  – Real-world use will flush them out
  – Expect to evolve API
Issue tracking

• Throughout process, maintain a list of design issues
  – Individual decisions such as what input format to accept
    • Write down all the options
    • Say which were ruled out and why
    • When you decide, say which was chosen and why

• Prevents wasting time on solved issues

• Provides rationale for the resulting API
  – Reminds its creators
  – Enlightens its users
Key design artifacts

1. Requirements document
2. Issues list
3. Use-case code

Maintain throughout design and retain when done

- They guide the design process
- When API is done, they’re the basis of the design rationale
  • Public explanation for design
  • For an example, see https://docs.oracle.com/javase/8/docs/technote guides/collections/designfaq.html
Disclaimer – one size does not fit all

• This process has worked for me
• Others developed similar processes independently
• But I’m sure there are other ways to do it
• The smaller the API, the less process you need
Puzzler: “Big Trouble”

```java
public static void main(String[] args) {
    BigInteger fiveThousand = new BigInteger("5000");
    BigInteger fiftyThousand = new BigInteger("50000");
    BigInteger fiveHundredThousand = new BigInteger("500000");

    BigInteger total = BigInteger.ZERO;
    total.add(fiveThousand);
    total.add(fiftyThousand);
    total.add(fiveHundredThousand);

    System.out.println(total);
}
```
What Does It Print?

public static void main(String [] args) {
    BigInteger fiveThousand = new BigInteger("5000");
    BigInteger fiftyThousand = new BigInteger("50000");
    BigInteger fiveHundredThousand = new BigInteger("500000");

    BigInteger total = BigInteger.ZERO;
    total.add(fiveThousand);
    total.add(fiftyThousand);
    total.add(fiveHundredThousand);

    System.out.println(total);
}
What Does It Print?

(a) 0
(b) 500000
(c) 555000
(d) It varies

BigInteger is immutable!
Another Look

```java
public static void main(String[] args) {
    BigInteger fiveThousand = new BigInteger("5000");
    BigInteger fiftyThousand = new BigInteger("50000");
    BigInteger fiveHundredThousand = new BigInteger("500000");

    BigInteger total = BigInteger.ZERO;
    total.add(fiveThousand);   // Ignores result
    total.add(fiftyThousand);  // Ignores result
    total.add(fiveHundredThousand);  // Ignores result

    System.out.println(total);
}
```
How do you fix it?

```java
public static void main(String[] args) {
    BigInteger fiveThousand = new BigInteger("5000");
    BigInteger fiftyThousand = new BigInteger("50000");
    BigInteger fiveHundredThousand = new BigInteger("500000");

    BigInteger total = BigInteger.ZERO;
    total = total.add(fiveThousand);
    total = total.add(fiftyThousand);
    total = total.add(fiveHundredThousand);

    System.out.println(total);
}
```

Prints 555000
The moral

• Names like add, subtract, negate suggest mutation
• Better names: plus, minus, negation
• Generally (and loosely) speaking:
  – Action verbs for mutation
  – Prepositions, linking verbs, nouns, or adjectives for pure functions
• **Names are important!**
Outline

• The Process of API Design
• Naming
• Documentation

How to name things:

the hardest problem in programming

@PeterHilton

http://hilton.org.uk/

https://hilton.org.uk/presentations/naming
Names Matter – API is a little language

*Naming is perhaps the single most important factor in API usability*

• Primary goals
  – Client code should read like prose ("easy to read")
  – Client code should mean what it says ("hard to misread")
  – Client code should flow naturally ("easy to write")

• To that end, names should:
  – be largely self-explanatory
  – leverage existing knowledge
  – interact harmoniously with language and each other
Deliberately meaningless names

In theory, \texttt{foo} is \textit{only} used as a placeholder name (because it doesn’t mean anything)
The easy part: typographical naming conventions

*The language specification demands that you follow these*

- Package or module – `org.junit.jupiter.api`, `com.google.common.collect`
- Class or Interface – `Stream`, `FutureTask`, `LinkedHashMap`, `HttpClient`
- Method or Field – `remove`, `groupingBy`, `getCrc`
- Parameter – `numerator`, `modulus`
- Constant Field – `MIN_VALUE`, `NEGATIVE_INFINITY`
- Type Parameter – `T`, `E`, `K`, `V`, `X`, `R`, `U`, `V`, `T1`, `T2`
How to choose names that are easy to read & write

• Choose key nouns carefully!
  – Related to finding good abstractions, which can be hard
  – If you can’t find a good name, it’s generally a bad sign

• If you get the key nouns right, other nouns, verbs, and prepositions tend to choose themselves

• Names can be literal or metaphorical
  – Literal names have literal associations
    • e.g., Matrix → inverse, determinant, eigenvalue, etc.
  – Metaphorical names enable reasoning by analogy
    • e.g., Publication, Subscriber → publish, subscribe, cancel, issue, issueNumber, circulation, etc.
Another way names drive development

• Names may remind you of another API
• Consider **copying** its vocabulary and structure
• People who know other API will have an easy time learning yours
• You may be able to develop it more quickly
• You may be able to use types from the other API
• You may even be able to share implementation
Names drive development, for better or worse

- Good names drive good development
- Bad names inhibit good development
- Bad names result in bad APIs unless you take action
- The API talks back to you. Listen!
Vocabulary consistency

• Use words consistently throughout your API
  – Never use the same word for multiple meanings
  – Never use multiple words for the same meaning
  – i.e., words should be isomorphic to meanings
Vocabulary consistency as it relates to scope

*APIs are actually little language extensions*

• The tighter the scope, the more important is consistency
  – Within APIs, consistency is critical
  – In related APIs on a platform, it’s highly desirable
  – Across the platform, it’s desirable
  – Between platforms, it’s nice-to-have

• **If forced to choose between local & platform consistency, choose local**

• But look to platform libraries for vocabulary
  – Ignoring obsolete and unpopular libraries

• Finally, look to similar APIs on other platforms for naming ideas
Avoid abbreviations except where customary

• Back in the day, storage was scarce & people abbreviated everything
  – Some continue to do this by force of habit or tradition

• Ideally, use complete words

• But sometimes, names just get too long
  – If you must abbreviate, do it tastefully
  – No excuse for cryptic abbreviations

• Of course you should use gcd, U rl, cos, m ba, etc.
Grammar is a part of naming too

• Nouns for classes
  – BigInteger, PriorityQueue
• Nouns or adjectives for interfaces
  – Collection, Comparable
• Nouns, linking verbs or prepositions for non-mutative methods
  – size, isEmpty, plus
• Action verbs for mutative methods
  – put, add, clear
• If you follow these, they quickly become second nature
Names should be regular – strive for symmetry

• If API has 2 verbs and 2 nouns, support all 4 combinations
  – Unless you have a very good reason not to
• Programmers will try to use all 4 combinations
  – They will get upset if the one they want is missing
• In other words, good APIs are generally orthogonal
Don’t mislead your user

• Names have implications
  – Learn them and uphold them in your APIs
• Don’t violate **the principle of least astonishment**
• Ignore this advice at your own peril
  – Can cause unending stream of subtle bugs

```java
public static boolean interrupted()
```

Tests whether the current thread has been interrupted. **The interrupted status of the thread is cleared by this method.**
Don’t lie to your user

• Name method for what it does, not what you wish it did
• If you can’t bring yourself to do this, fix the method!
• Again, ignore this at your own peril

```java
public long skip(long n) throws IOException
```

Skips over and discards n bytes of data from this input stream. The skip method may, for a variety of reasons, end up skipping over some smaller number of bytes, possibly 0. This may result from any of a number of conditions; reaching end of file before n bytes have been skipped is only one possibility. The actual number of bytes skipped is returned...
Good naming takes time, but it’s worth it

• Don’t be afraid to spend hours on it; I do.
  – And I still get the names wrong sometimes
• Discuss names with colleagues; it really helps.
Adopt better naming practices

• Start with *meaning* and *intention*.
• Use words with precise meanings.
• Prefer fewer words in names.
• No abbreviations in names (except id)
• Use code review to improve names.
• Read the code out loud to check that it *sounds* okay.
• Actually rename things.
Lecture summary

- APIs took off in the past thirty years and gave us super-powers
- Good APIs are a blessing; bad ones, a curse
- Following an API design process greatly improves API quality
- Naming is critical to API usability