Principles of Software Construction: Objects, Design, and Concurrency

API Design - part deux

Michael Hilton Bogdan Vasilescu

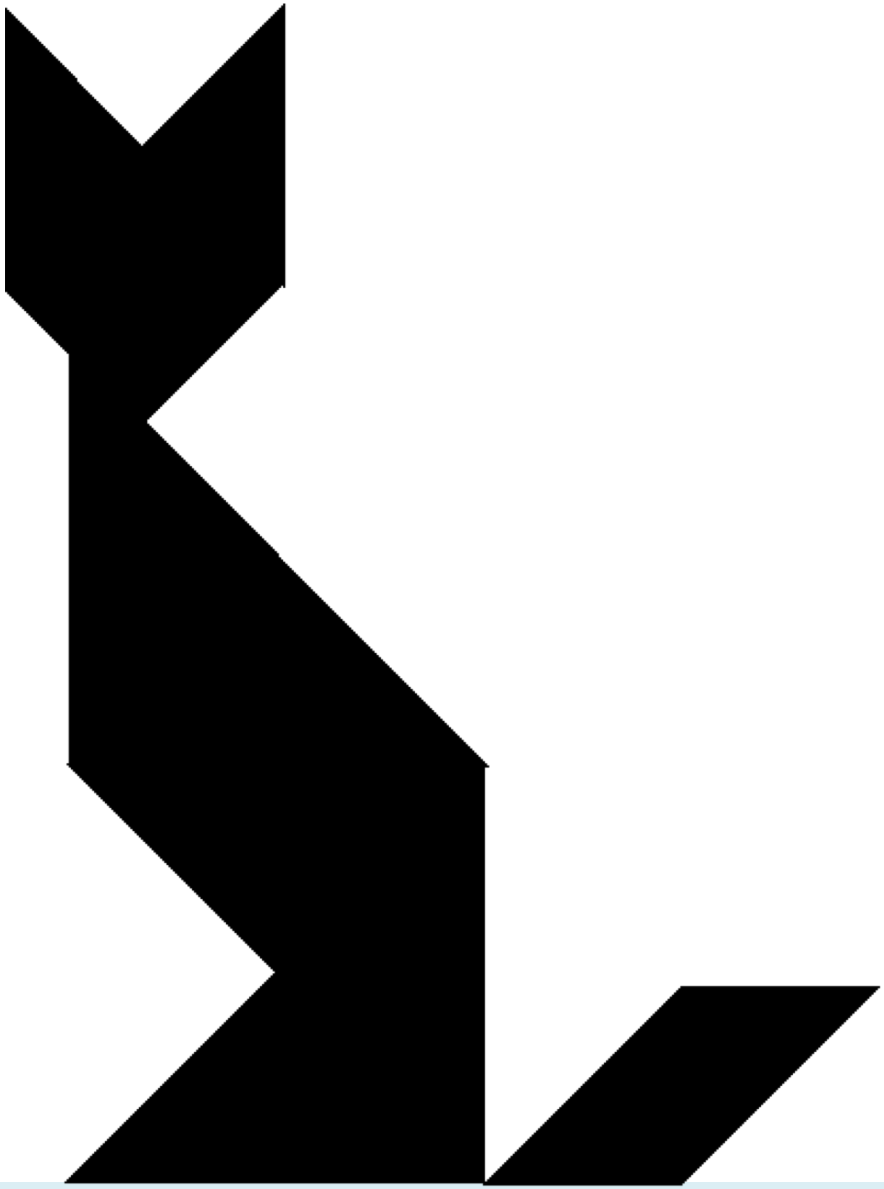
Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

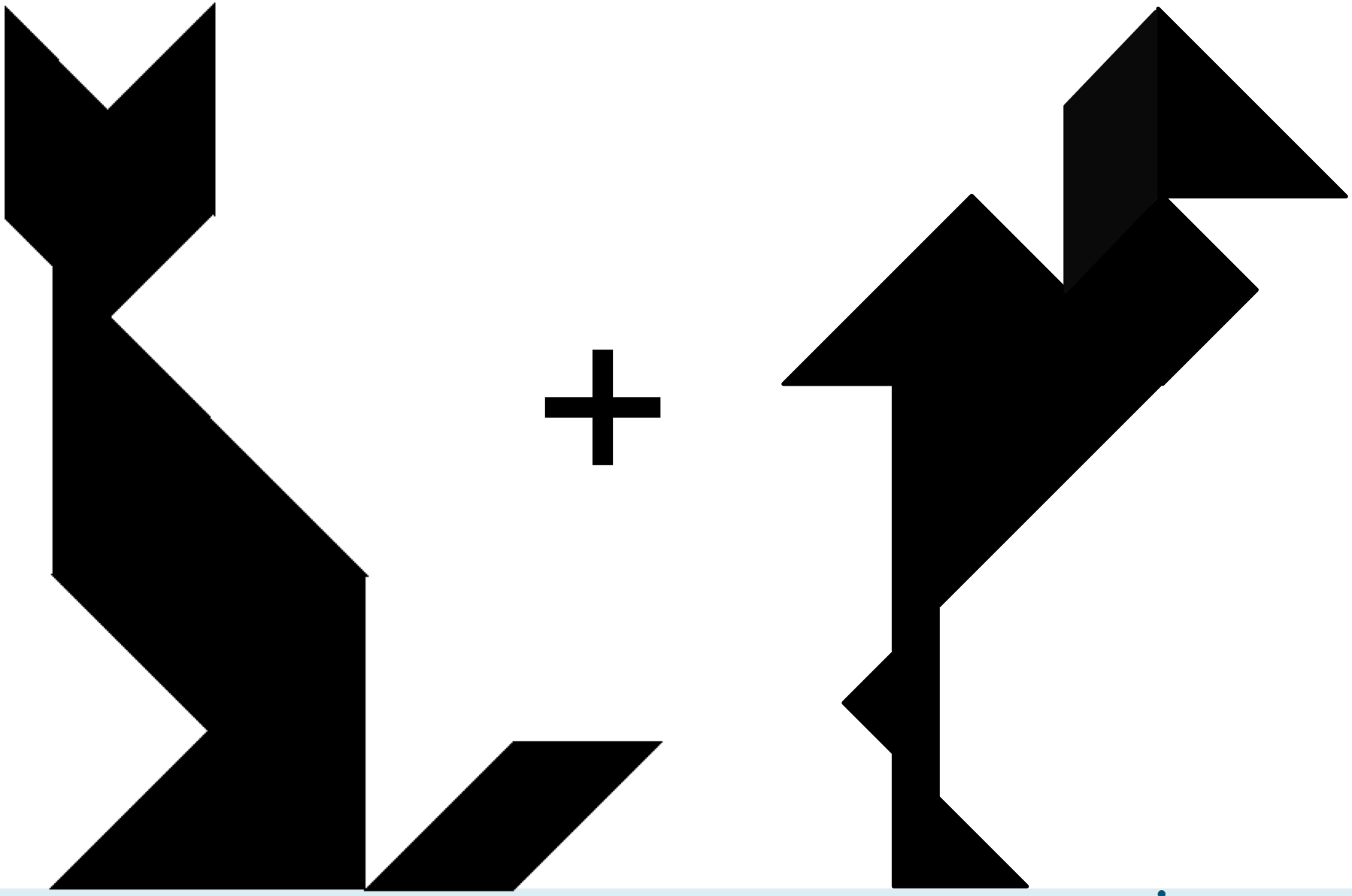
Administrivia

- Homework 4c due Monday March 25
- Midterm 2 Thursday March 28th
  - Midterm Review March 27 ??? pm in **NSH 3305**
    - Piazza Poll active

The use vs. reuse dilemma

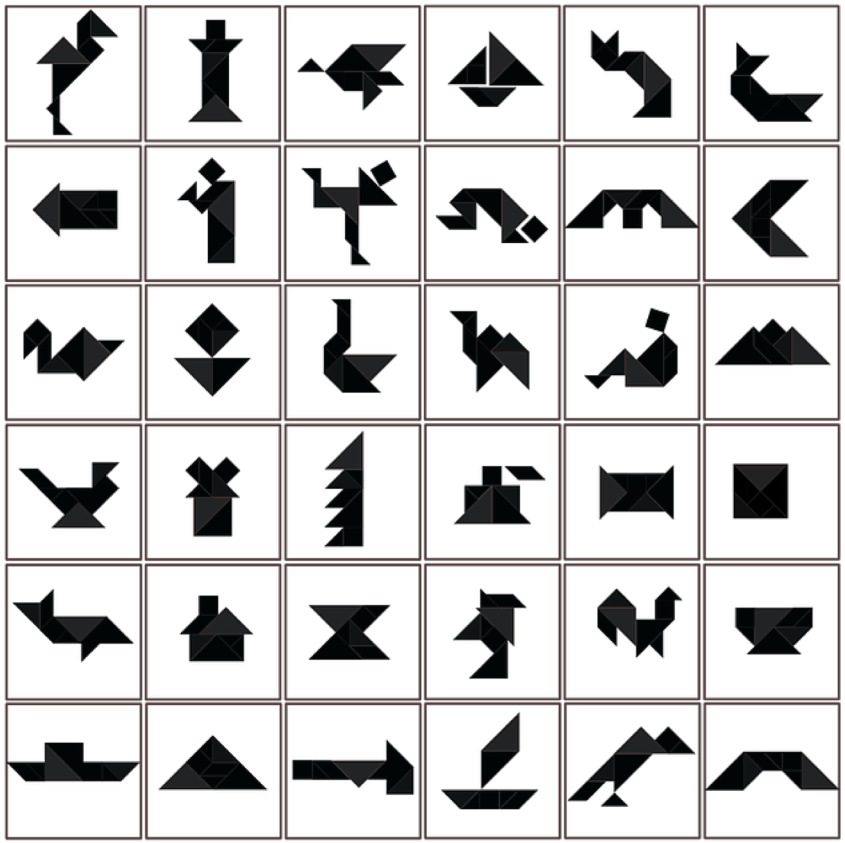


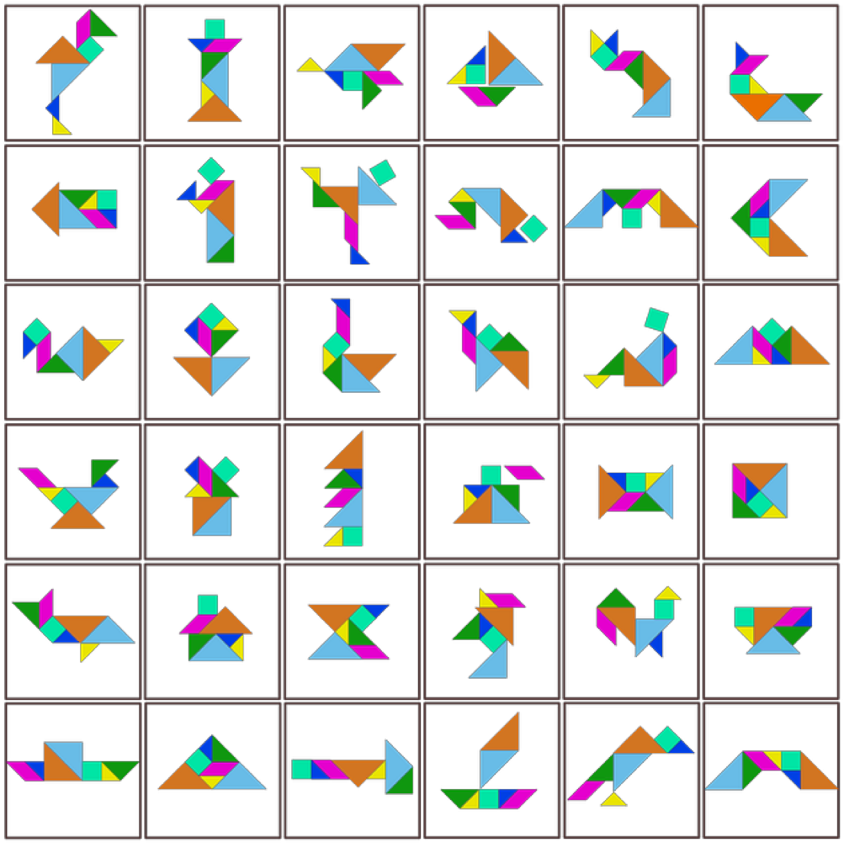
The use vs. reuse dilemma



The use vs. reuse dilemma

- One modularization: tangrams





The use vs. reuse dilemma

- Large rich components are very useful, but rarely fit a specific need
- Small or extremely generic components often fit a specific need, but provide little benefit

"Maximizing reuse minimizes use."

Clemens Szyperski

Intro to Java

Git, CI

UML

GUIs

Static Analysis

Performance

GUIs

More Git

Design

Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Unit Testing, Design Patterns	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Testing Subsystems	
	Design for Reuse at Scale: Frameworks and APIs	

This week: 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.

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

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/technotes/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”


```
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);
}
```

(a) 0
(b) 500000
(c) 555000
(d) Other

What Does It Print?

(a) `0`

(b) `500000`

(c) `555000`

(d) It varies

`BigInteger` is immutable!

Another Look

```
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?

```
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

- Blame the API designer
  - (In fairness, this was my first OO API,1996)
- 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

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, `foo` is *only* used as a placeholder name (because it doesn't mean anything)

@PeterHilton • 19

foo

Pull requests Issues Marketplace Explore

Repositories	51K
Code	97M
Commits	9M
Issues	593K
Wikis	72K
Users	5K

Languages

PHP	25,861,022
C	9,846,338
JavaScript	9,095,923
Python	5,342,260
HTML	4,637,208
Ruby	4,521,704
C++	4,138,224

Showing 99,397,762 available code results

Sort: Best match

kr/sodium – bare-message.na
Showing the top match Last indexed on Sep 14, 2016

```
1  .foo
```

ishanagrawal/glucial – deferred.ejs EJS
Showing the top match Last indexed on Sep 14, 2016

```
1  <%= foo %>
```

jillesvangurp/jsonj – test_malformed_2.json JSON
Showing the top match Last indexed on Sep 14, 2016

```
1  foo
```

bublik/medicina – foo.txt Text
Showing the top match Last indexed on Sep 14, 2016

```
1  foo
```

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`, `Url`, `cos`, `mba`, 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

```
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

```
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.

Interlude

On the Naturalness of Software

Abram Hindle, Earl T. Barr, Zhendong Su
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
{ajhindle,barr,su}@cs.ucdavis.edu

Mark Gabel
Dept. of Computer Science
The University of Texas at Dallas
Richardson, TX 75080 USA
mark.gabel@utdallas.edu

Premkumar Devanbu
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
devanbu@cs.ucdavis.edu

***Abstract*—Natural languages like English are rich, complex, and powerful. The highly creative and graceful use of languages like English and Tamil, by masters like Shakespeare and Avvaiyar, can certainly delight and inspire. But in practice, given cognitive constraints and the exigencies of daily life, most human utterances are far simpler and much more repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension.**

We begin with the conjecture that most software is also natural, in the sense that it is created by humans at work,

efforts in the 1960s. In the '70s and '80s, the field was re-animated with ideas from logic and formal semantics, which still proved too cumbersome to perform practical tasks at scale. Both these approaches essentially dealt with NLP from first principles—addressing *language*, in all its rich theoretical glory, rather than examining corpora of actual *utterances*, *i.e.*, what people actually write or say. In the 1980s, a fundamental shift to *corpus-based, statistically rigorous* methods occurred. The availability of large, on-line corpora of natural language text, including "aligned" text with translations in multiple languages,[1] along with the computational muscle (CPU speed,

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