Principles of Software Construction: Objects, Design, and Concurrency

API Design

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 5-7pm in **NSH 3305**

QUIZ

Required reading from Effective Java, Items 51, 60, 62, and 64

(take out a piece of paper)

In less than 3 minutes

- Write your Andrew ID
- Based on the reading, fix the following code:

```
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

1. What is your Andrew ID?

- +2 points for any answer

Fix the code snippet

```
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

Becomes

```
Set<Son> sonSet = new LinkedHashSet<>();
```

Why?

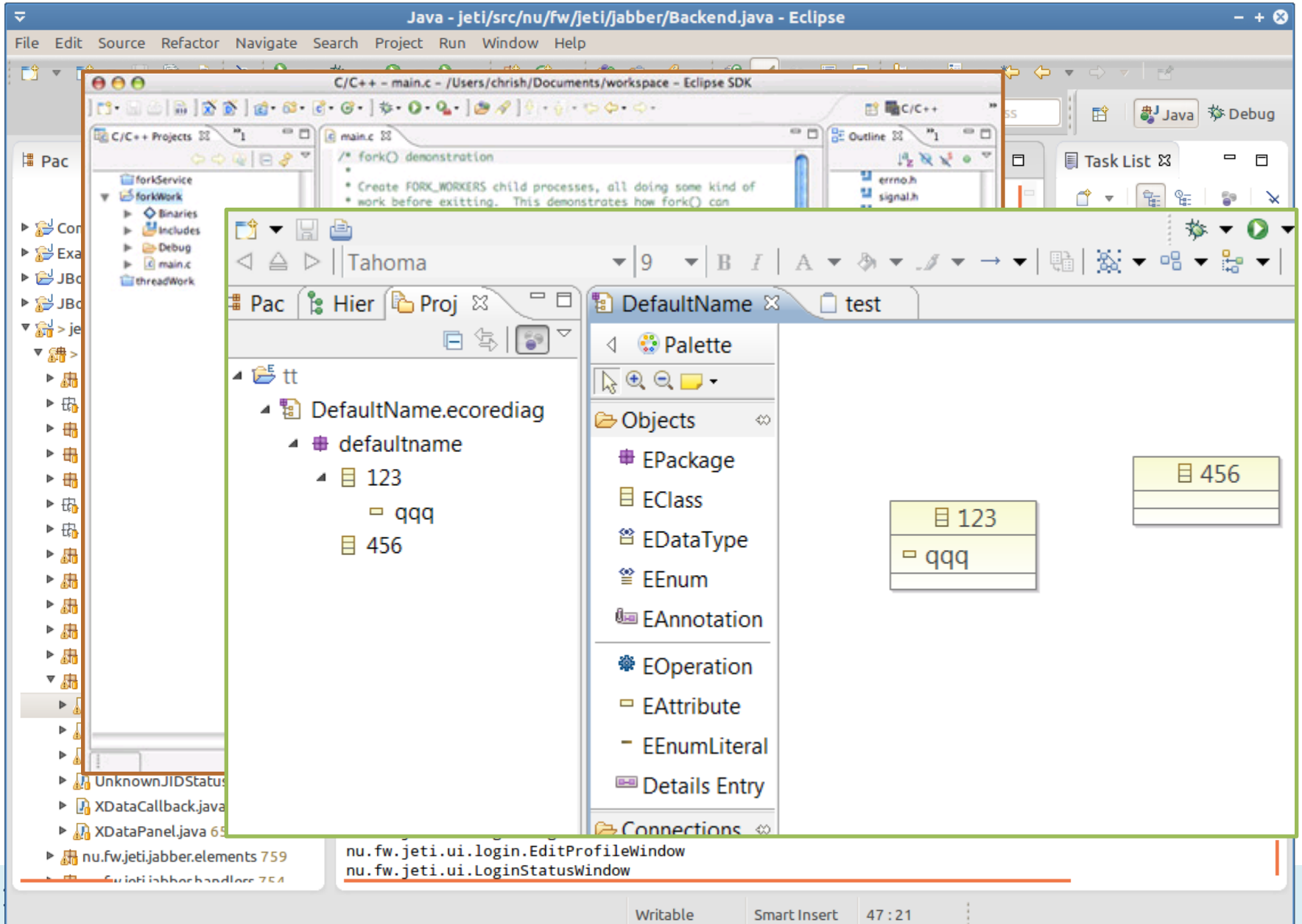
- “If you get into the habit of using interfaces as types, your program will be much more flexible.”

Earlier in this course: Class-level reuse

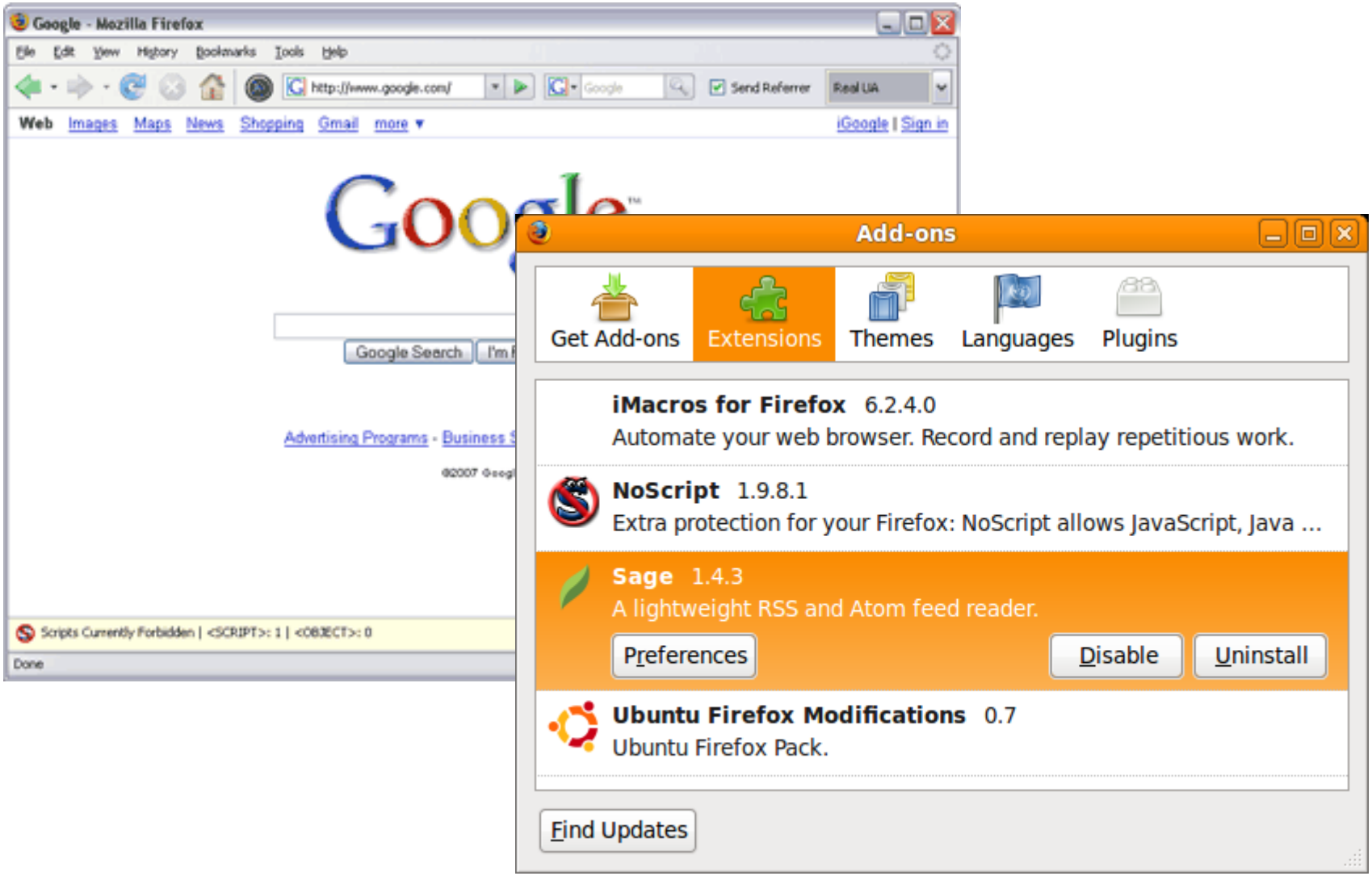
- Language mechanisms supporting reuse
  - Inheritance
  - Subtype polymorphism (dynamic dispatch)
  - Parametric polymorphism (generics)*
- Design principles supporting reuse
  - Small interfaces
  - Information hiding
  - Low coupling
  - High cohesion
- Design patterns supporting reuse
  - Template method, decorator, strategy, composite, adapter, …

* Effective Java items 26, 29, 30, and 31

Reuse and variation: Family of development tools

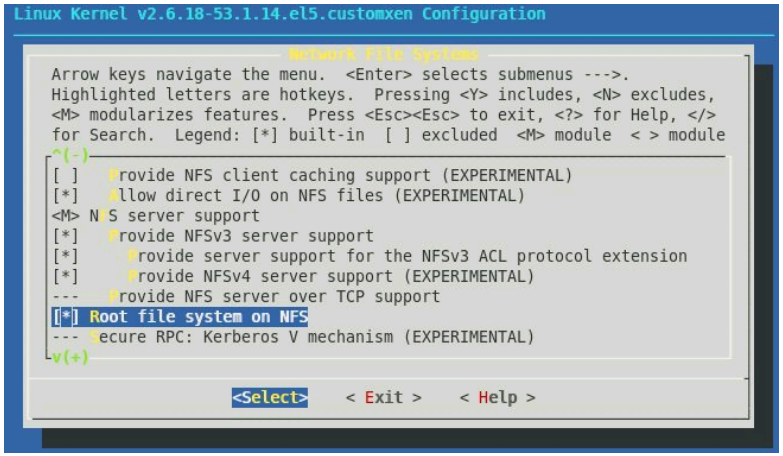


Reuse and variation: Web browser extensions



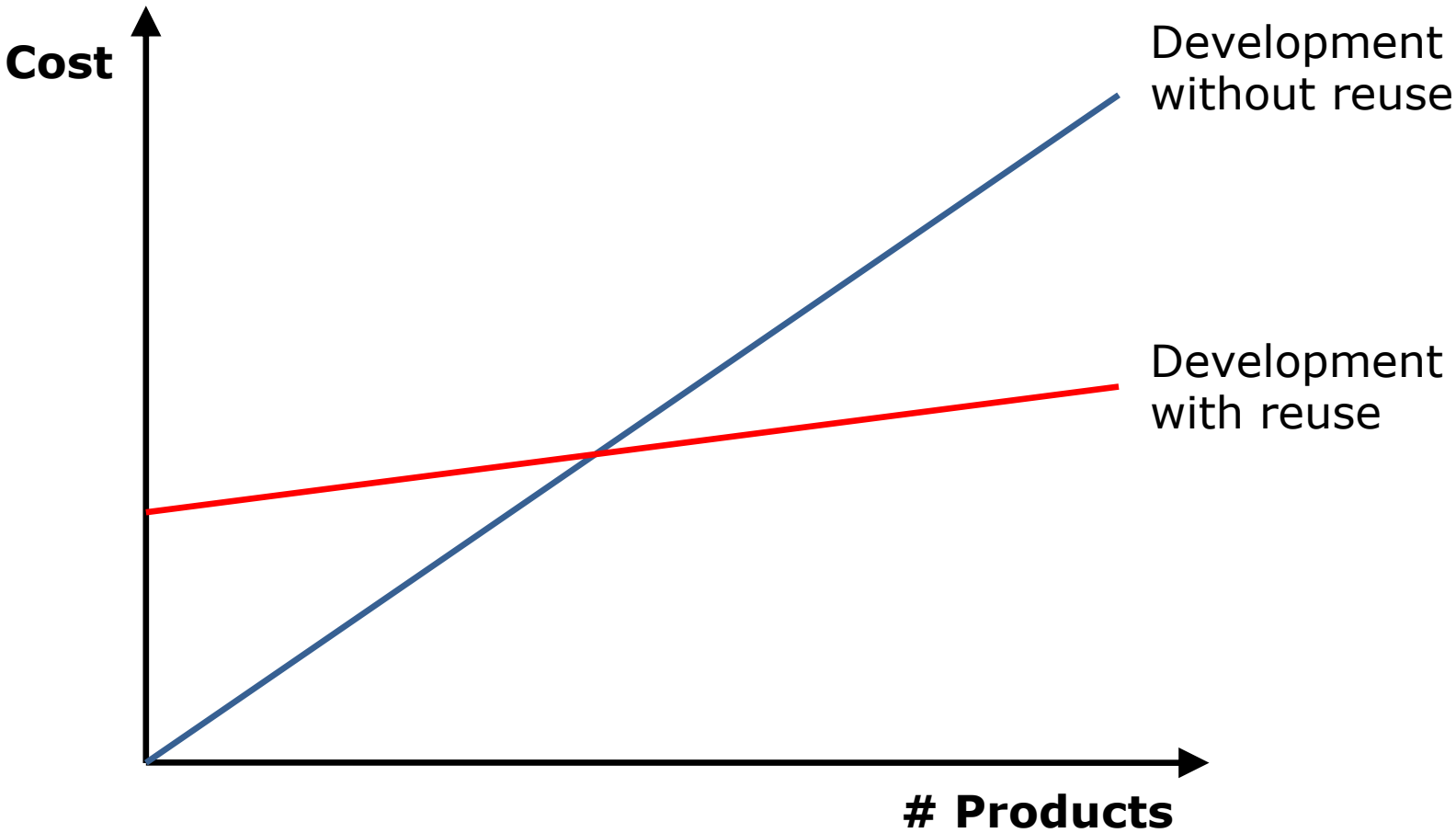
institute for SOFTWARE RESEARCH

Reuse and variation: Flavors of Linux



Reuse and variation: Product lines

The promise


Cost
Development without reuse
Development with reuse
Products

Today: Libraries and frameworks for reuse

- Terminology and examples
- Whitebox and blackbox frameworks
- Designing a framework
- Implementation details

Today: Libraries and frameworks for reuse

- **Terminology and examples**
- Whitebox and blackbox frameworks
- Designing a framework
- Implementation details

Terminology: Library

- *Library*: A set of classes and methods that provide reusable functionality
- Client calls library; library executes and returns data
- Client controls
  - Program structure
  - Control flow

```
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

- E.g.: Math, Collections, Graphs, I/O, Swing

Terminology: Frameworks

- *Framework*: Reusable skeleton code that can be customized into an application
- Framework calls back into client code
  - The Hollywood principle: “Don’t call us. We’ll call you.”
- Framework controls
  - Program structure
  - Control flow

```
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

Framework

- E.g.: Eclipse, Firefox, Spring, Swing

Framework or library?

- Eclipse
- Java Collections
- The Java Logging Framework
- Java Encryption Services
- Wordpress
- Django

Today: Libraries and frameworks for reuse

- Terminology and examples
- **Whitebox and blackbox frameworks**
- Designing a framework
- Implementation details

Whitebox vs. blackbox framework

Whitebox frameworks	Blackbox frameworks
Subclassing and overriding methods	Composition; implementing a plugin interface
Subclass has `main()` but gives control to framework	Plugin-loading mechanism loads plugins and gives control to framework

Today: Libraries and frameworks for reuse

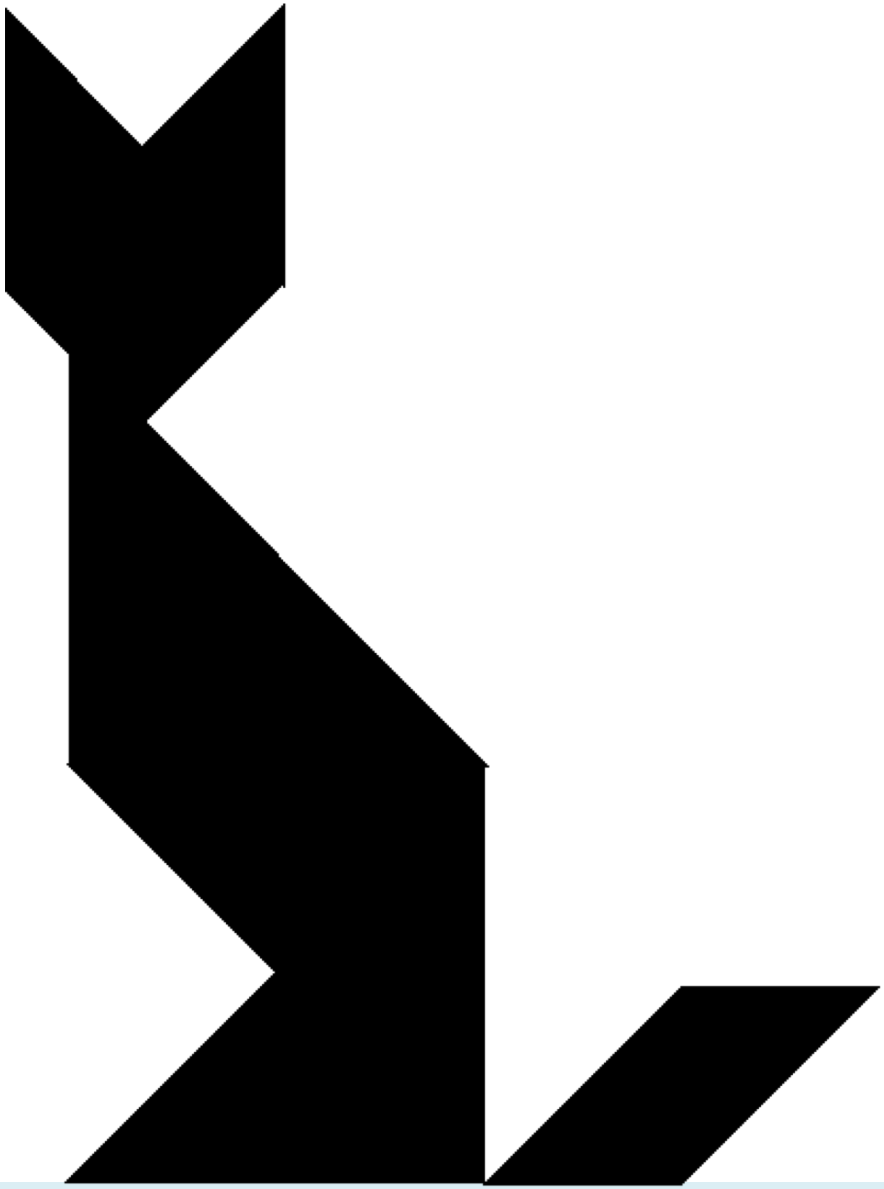
- Terminology and examples
- Whitebox and blackbox frameworks
- **Designing a framework**
- Implementation details

Framework design considerations

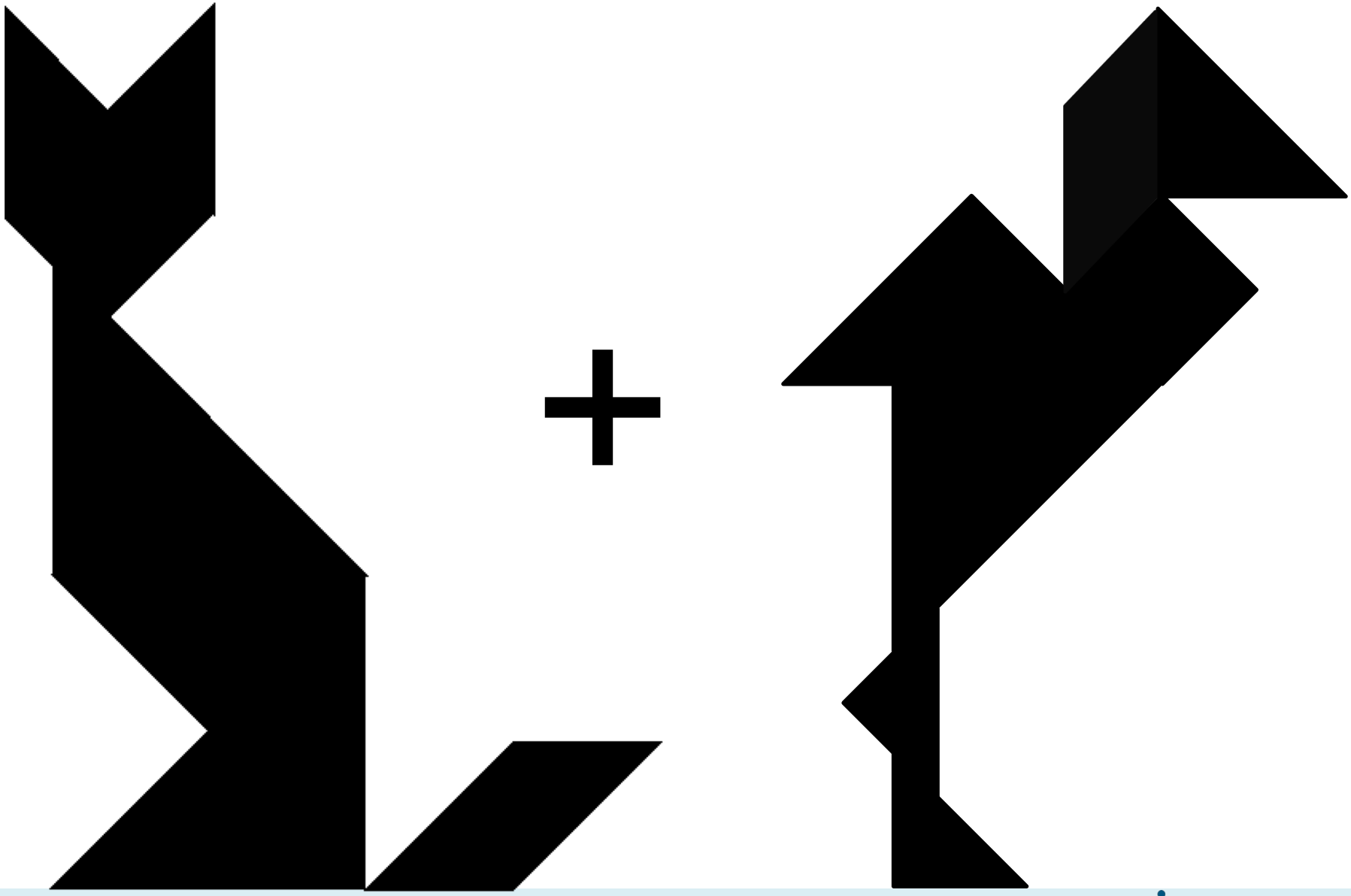
- Once designed there is little opportunity for change
- Writing a plugin or extension should NOT require modifying the framework source code
- Key decision: Separating common parts from variable parts
- Possible problems:
  - Too few extension points: Limited to a narrow class of users
  - Too many extension points: Hard to learn, slow
  - Too generic: Little reuse value

The use vs. reuse dilemma

The use vs. reuse dilemma

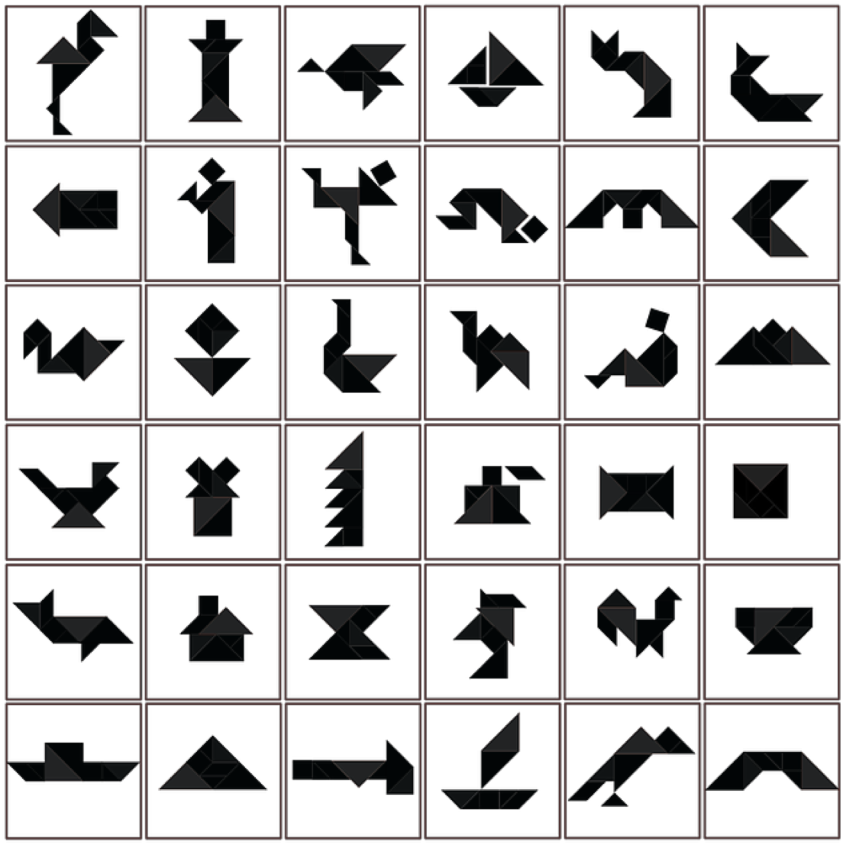


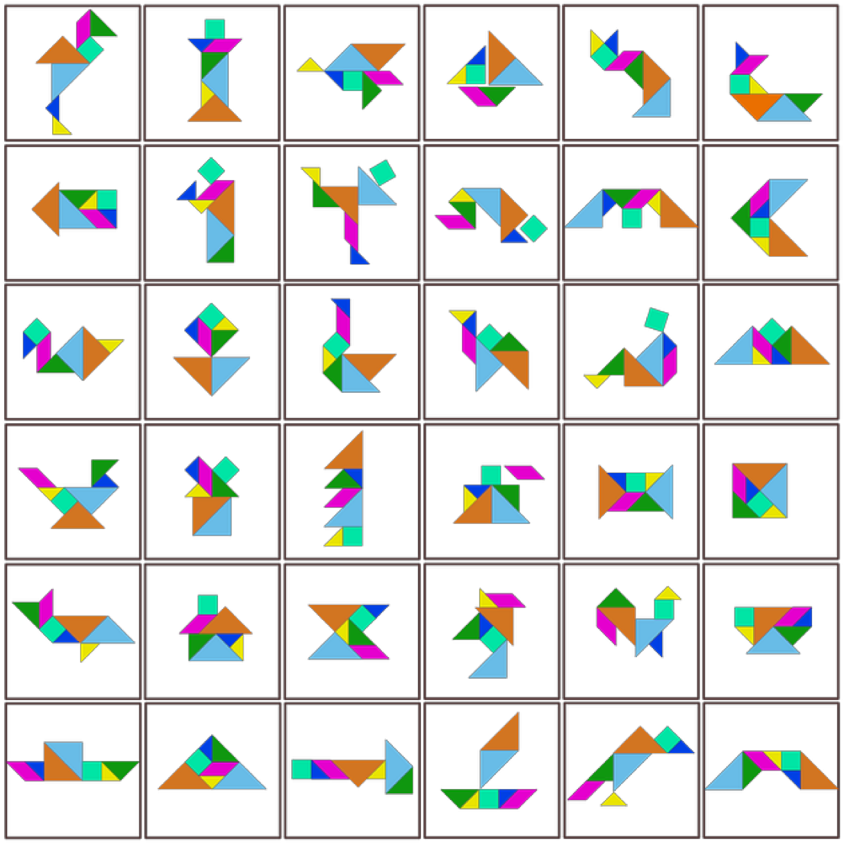
The use vs. reuse dilemma



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

Today: Libraries and frameworks for reuse

- Terminology and examples
- Whitebox and blackbox frameworks
- **Designing a framework**
- Implementation details

Today: Libraries and frameworks for reuse

- Terminology and examples
- Whitebox and blackbox frameworks
- Designing a framework
- **Implementation details (more in recitation)**

Running a framework

- Some frameworks are runnable by themselves
  - E.g., Eclipse
- Other frameworks must be extended to be run
  - E.g.: Swing, JUnit, MapReduce, Servlets

Methods to load plugins

1. Client writes `main()`, creates a plugin, and passes it to framework
2. Framework writes `main()`, client passes name of plugin as a command line argument or environment variable
   - E.g., use reflection* to dynamically load plugins:

```
Plugin p = (Plugin) Class.forName(args[1]).newInstance();
```

3. Framework looks in a magic location
   - Config files or .jar files are automatically loaded and processed
   - E.g., use `java.util.ServiceLoader` to load classes from a standard configuration (`META-INF/services/…`)
4. GUI for plugin management
   - E.g., web browser extensions

* Effective Java item 65

Summary: Libraries and frameworks for reuse

- Terminology and examples
  - Library, framework, callback, lifecycle method, …
- Whitebox frameworks vs. blackbox frameworks
- Designing a framework
  - Domain engineering
  - Key decision: Separating common parts from variable parts
  - Writing a plugin should NOT require modifying the framework
  - Use vs. reuse
- Implementation details (more in recitation)

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

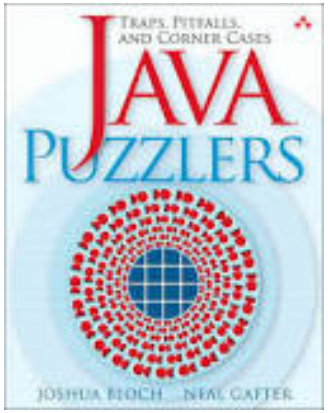
- An API design process
- The key design principle: information hiding
- Concrete advice for user-centered design

Based heavily on "How to Design a Good API and Why it Matters" by Josh Bloch.

1. “Time for a Change” (2002)

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

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



What does it print?

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

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

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

```
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")));
    }
}
```

Prints 0.90

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

Prints 90

The moral

- Avoid `float` and `double` where exact answers are required
  - For example, when dealing with money
- Use `BigDecimal`, `int`, 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?

```
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:

`0.89999999999999991118215802998747676610946655273437 5`

We used the wrong `BigDecimal` constructor

Another look

The spec says:

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

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

```
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?

```
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));
    }
}
```

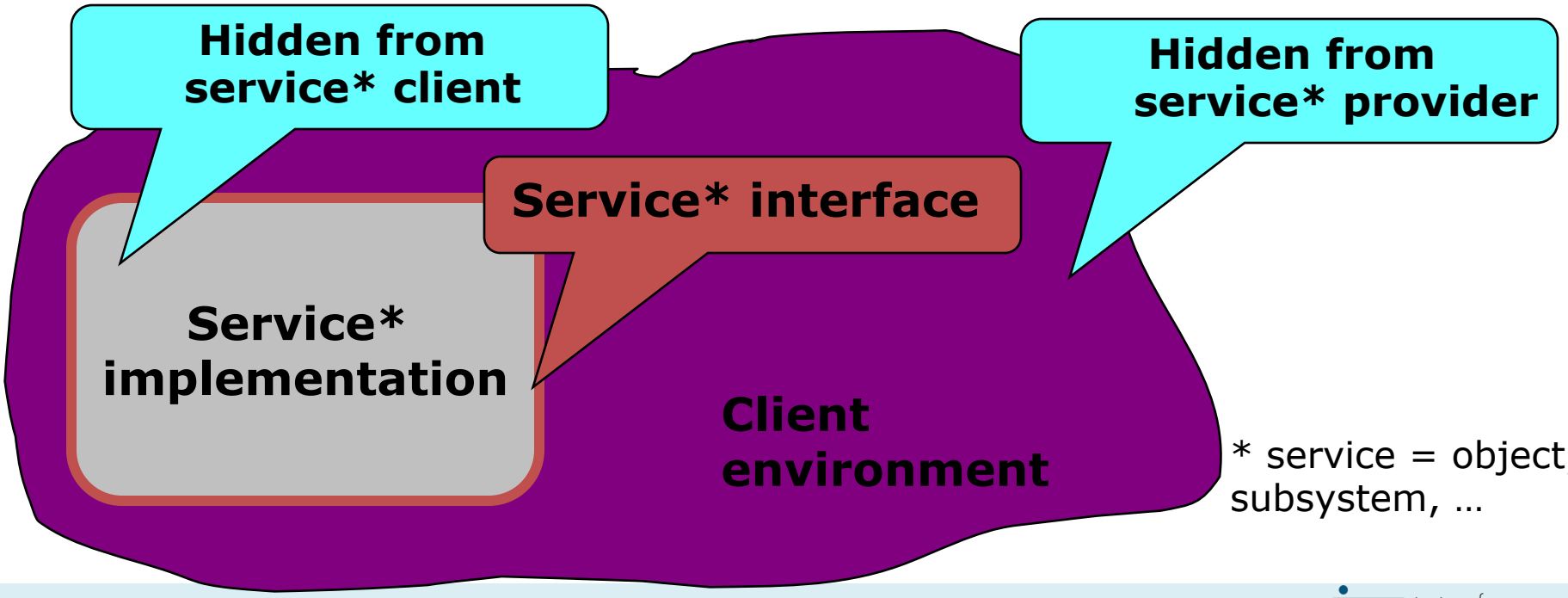
Prints `0.90`

The moral

- Use `new BigDecimal(String)`, not `new BigDecimal(double)`
- `BigDecimal.valueOf(double)` is better, but not perfect
  - Use it for non-constant values.
- 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 little implementation detail as necessary
- Allows to change hidden details later



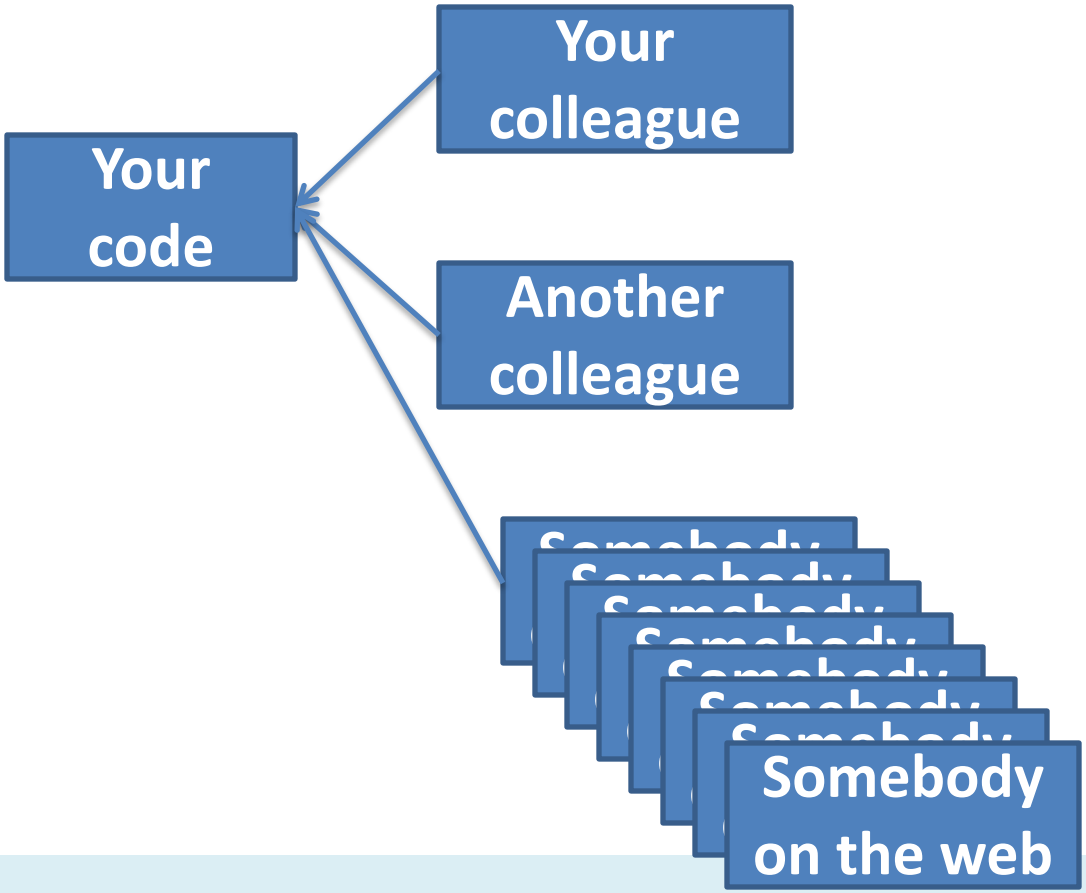
* service = object, subsystem, …

Motivation to create a public API

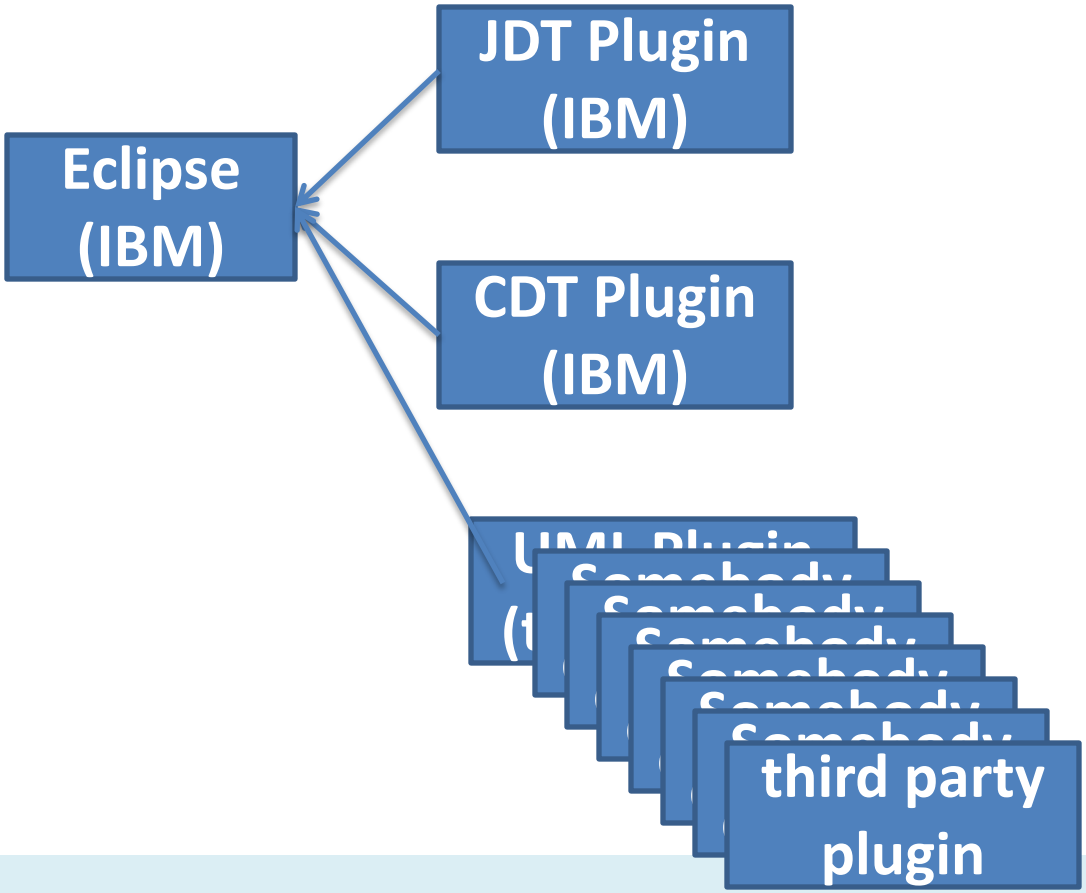
- Good APIs are a great asset
  - Distributed development among many teams
    - Incremental, non-linear software development
    - Facilitates communication
  - Long-term buy-in from clients & customers
    - Users invest heavily: acquiring, writing, learning
    - Cost to **stop** using an API can be prohibitive
    - Successful public APIs capture users

- Poor APIs are a great liability
  - Lost productivity from your software developers
  - Wasted customer support resources
  - Lack of buy-in from clients & customers

Public APIs are forever

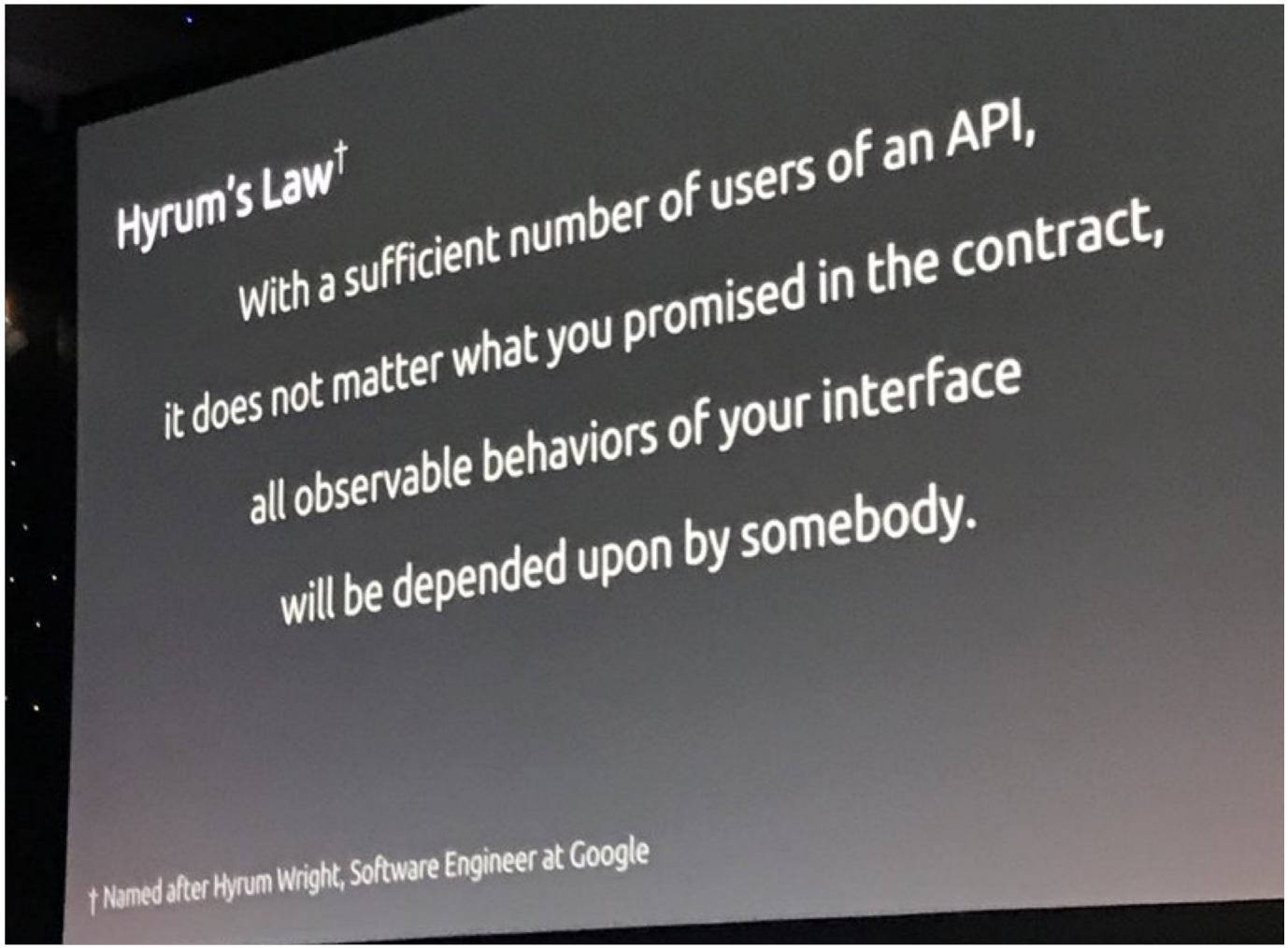


Public APIs are forever



Hyrum's Law

Hyrum Wright
@hyrumwright
Infrastructure software engineer. Googler. Father. Occasional professor.



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

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

’10s – Machine learning, IOT, 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