Java OOP Principles

CS356 Object-Oriented Design and Programming
http://cs356.yusun.io
October 3, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

Announcements

- Quiz 1 on Monday (10/6)
- GitHub Username!

Recommended Reading

- Part of the content in this lecture comes from:
- Joshua Bloch, *Effective Java*, Addison Wesley, 2008

1. Accessibility of Java Classes and Members

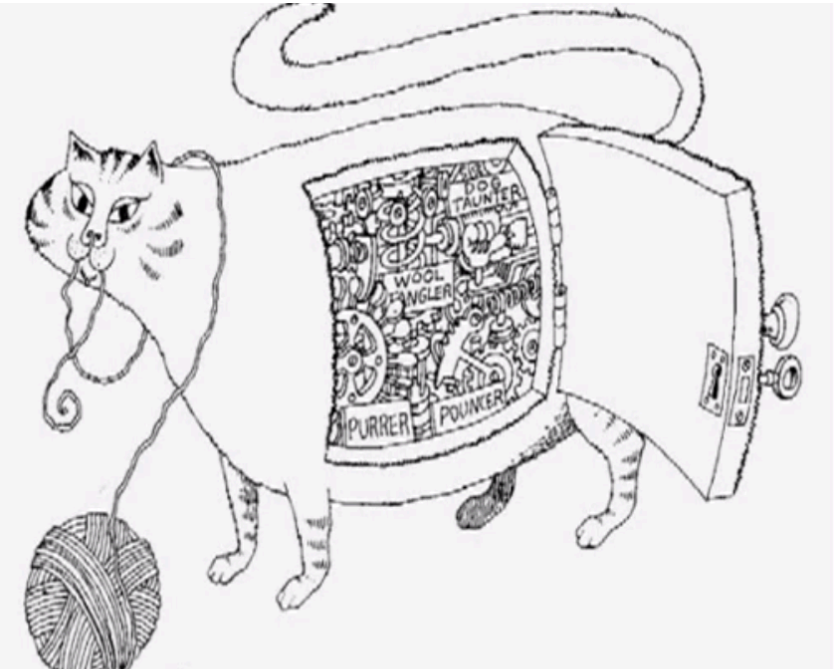
Access Control in Java

private	The member is accessible only from the top-level class where it is declared.
default	The member is accessible from any class in the package where it is declared. Technically known as default access, this is the access level you get if no access modifier is specified.
protected	The member is accessible from subclasses of the class where it is declared and from any class in the package where it is declared.
public	The member is accessible from anywhere.

- How should we decide which accessibility to use?

What is called well-designed software?

- The most important factor to distinguish a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details from other modules.



Encapsulation hides the details of the implementation of an object.

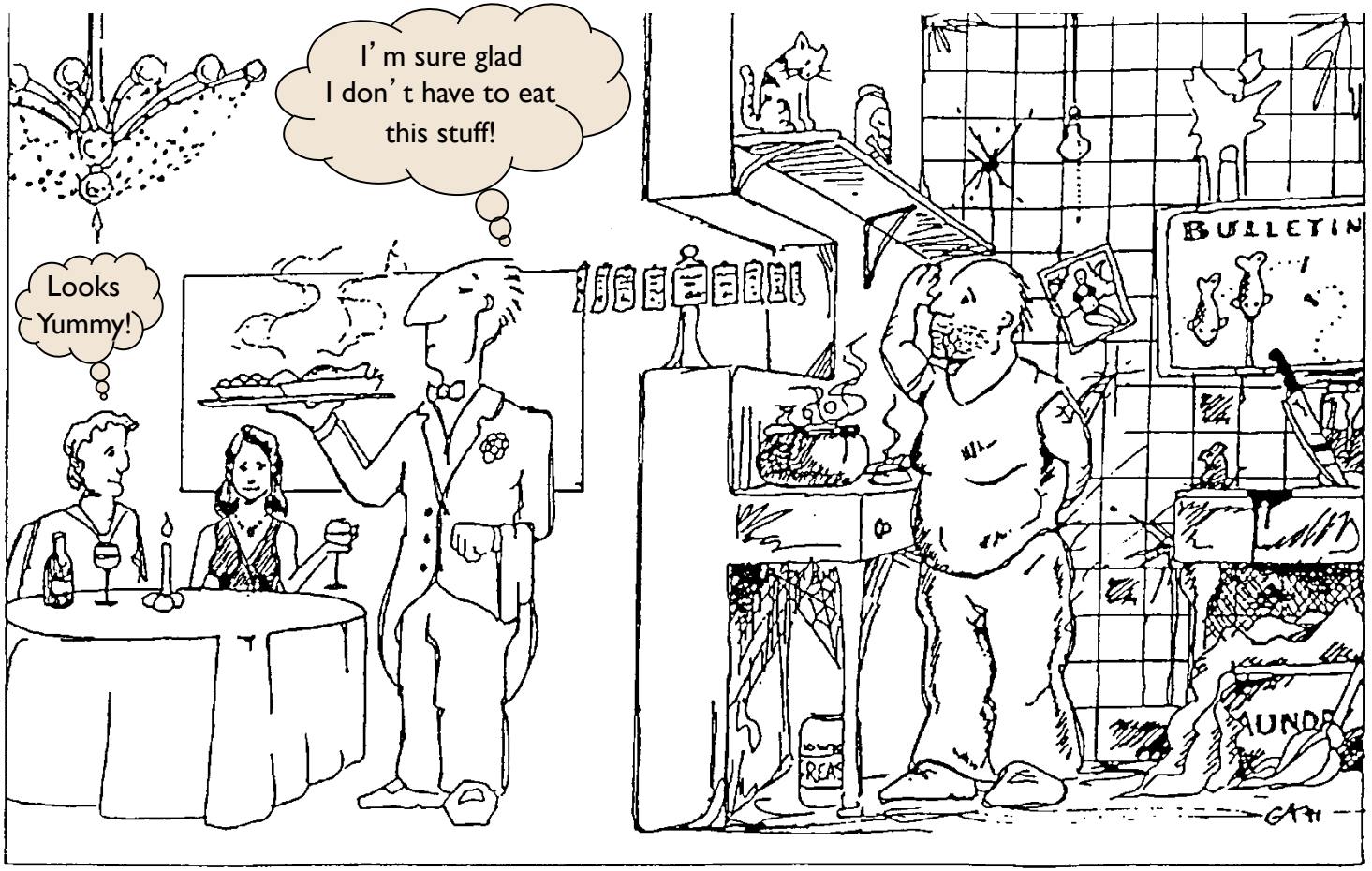
Information Hiding

- Hides the design decisions and implementation details from other modules

- "The second decomposition was made using 'information hiding' ... as a criterion. The modules no longer correspond to steps in the processing. ... Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings." – [Parnas, 1972b]

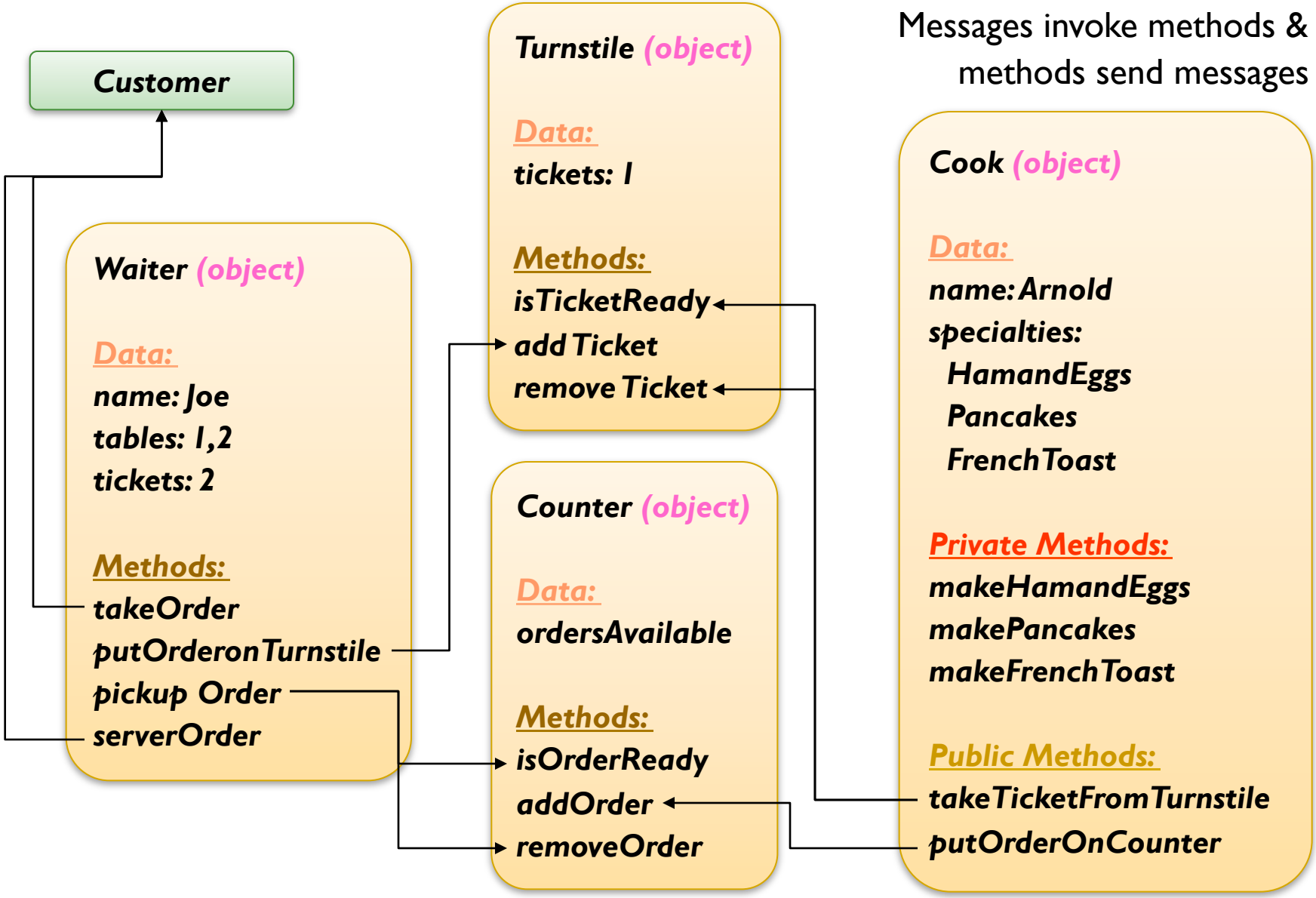
- "... the purpose of hiding is to make inaccessible certain details that should not affect other parts of a system." – [Ross et al., 1975]

The Restaurant

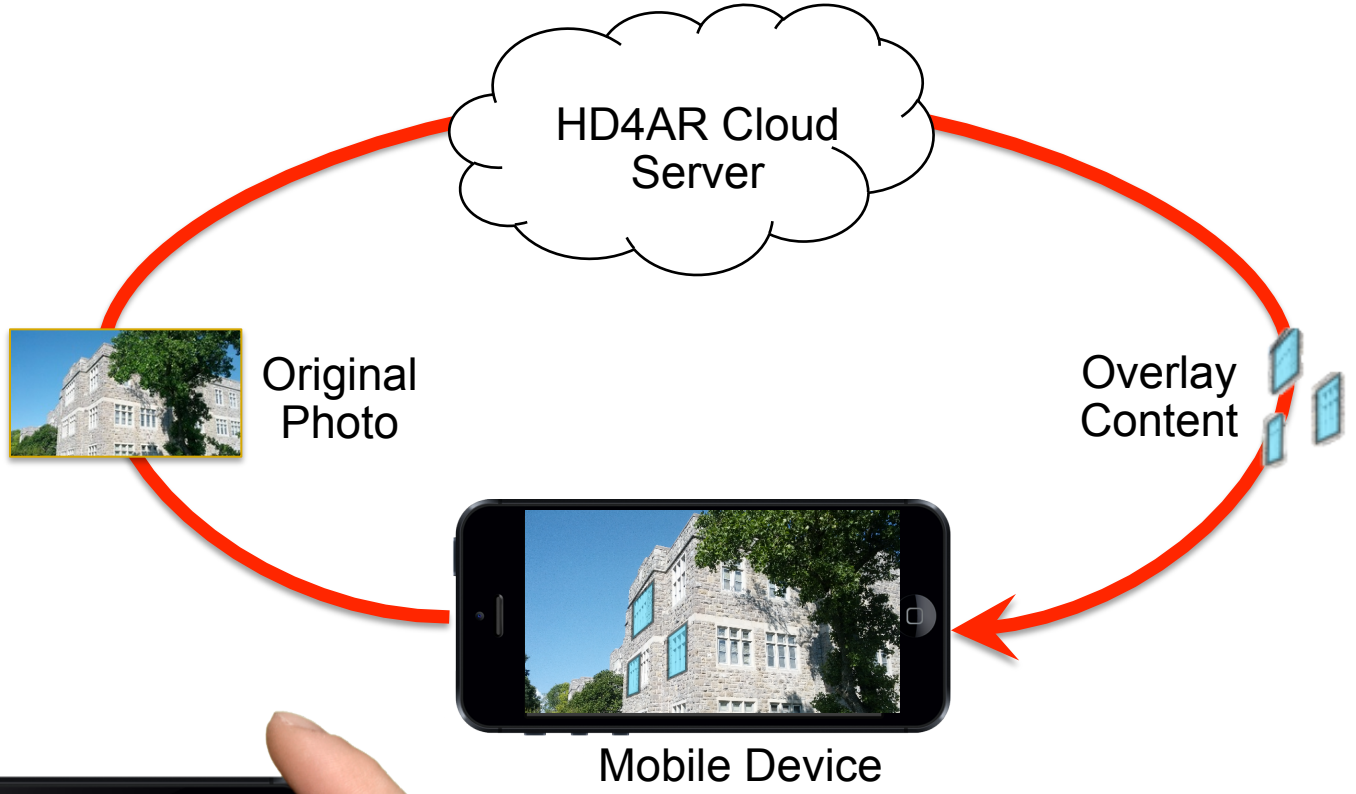


Once a message has been passed to an object, objects outside can't know and don't care how processing takes place.

The Restaurant



Cloud-based Computer Vision Architecture



Overview of the Algorithms

Photos Captured → Feature Extraction → Feature Matching → Track Creation → Structure from Motion

~15+ photos are taken of a physical object, such as a building, engine, etc.

Overview of the Algorithms

Photos Captured → **Feature Extraction** → Feature Matching → Track Creation → Structure from Motion

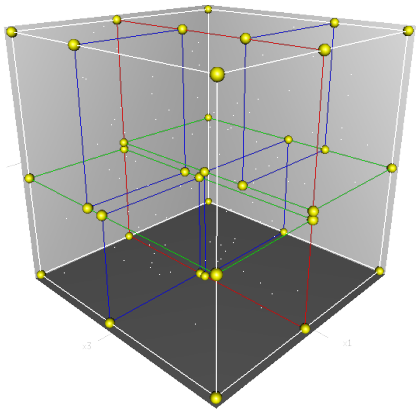
Image features (e.g. prominent points in the image) are extracted and represented using descriptors

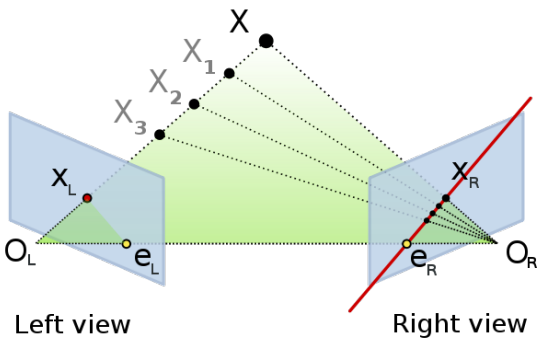
- Example Feature Detectors/Descriptors
  - SIFT
  - SURF
  - FREAK

Overview of the Algorithms

Photos Captured → Feature Extraction → **Feature Matching** → Track Creation → Structure from Motion

K-D Tree and Fast Approximate Nearest Neighbors (FANN) used to find initial feature correspondences

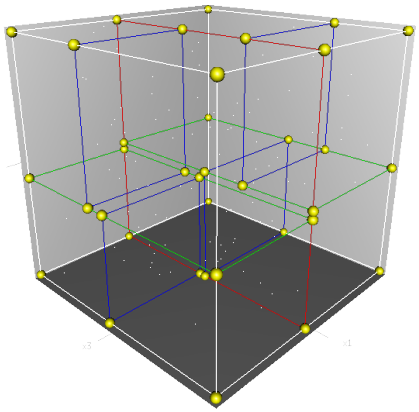


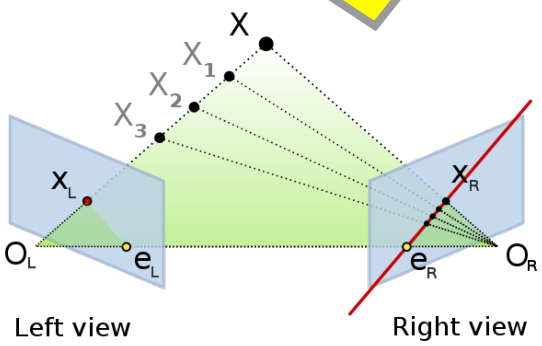


Overview of the Algorithms

Photos Captured → Feature Extraction → **Feature Matching** → Track Creation → Structure from Motion

RANSAC algorithm and 8-point method is used to estimate a fundamental matrix between image pairs and points more than σ pixels from an epipolar line are eliminated



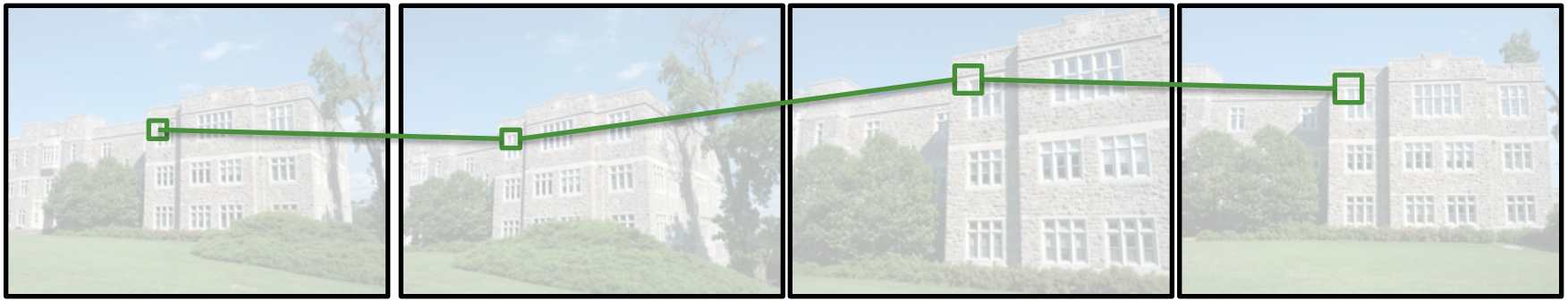


Overview of the Algorithms

Photos Captured → Feature Extraction → Feature Matching → **Track Creation** → Structure from Motion

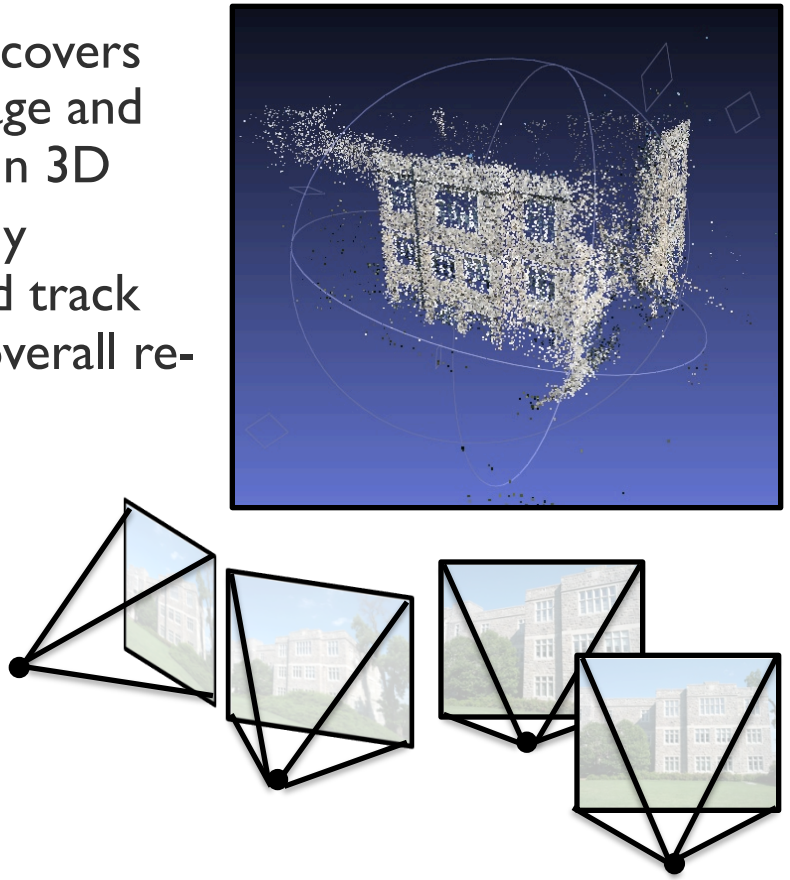
Feature tracks are created that track features across multiple images and a track cleaning operation is performed

- Track Cleaning: the Euclidean distance between matching points is compared to the minimum Euclidean distance between any pair in the track and outliers are removed

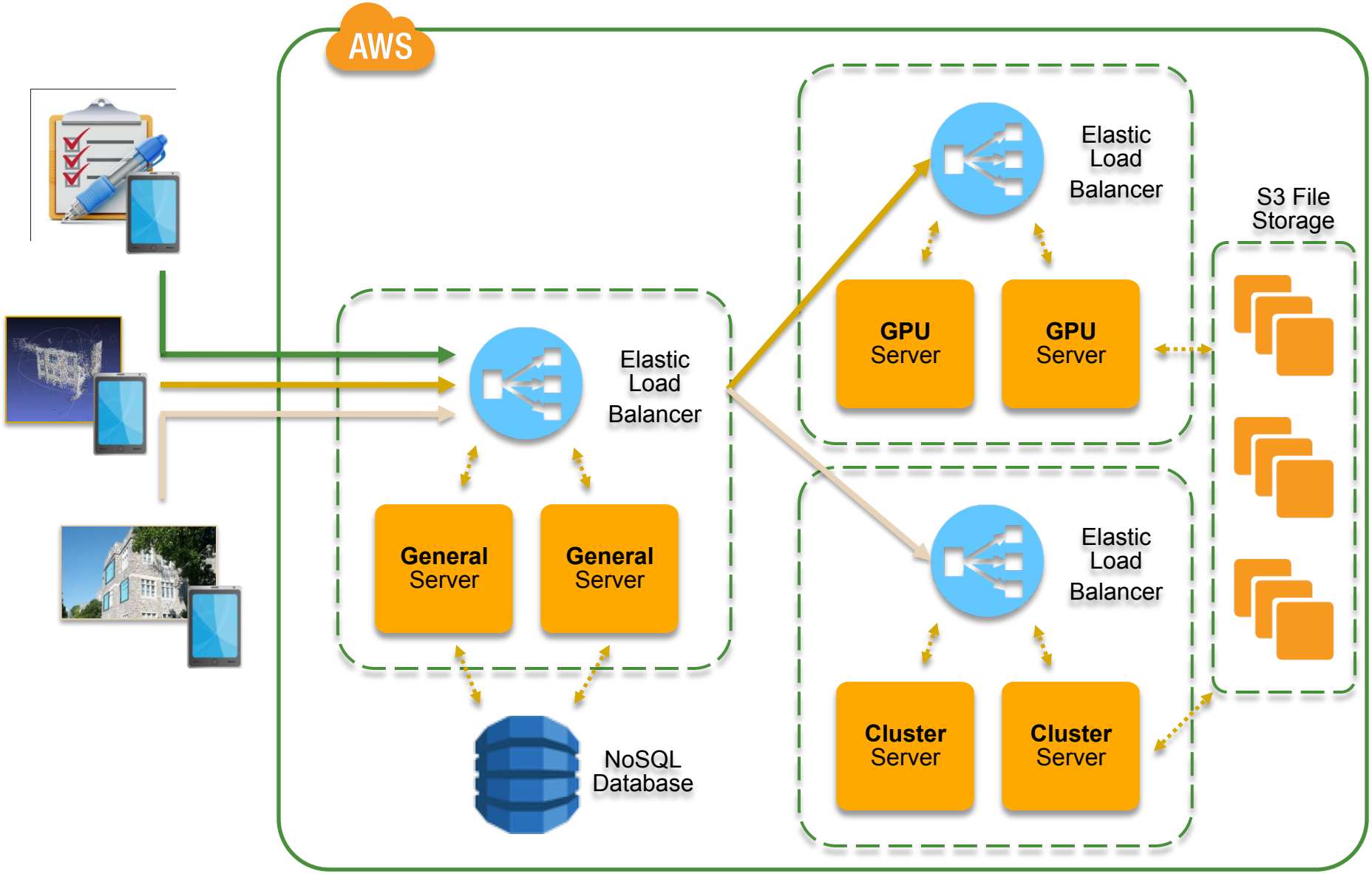


Overview of the Algorithms

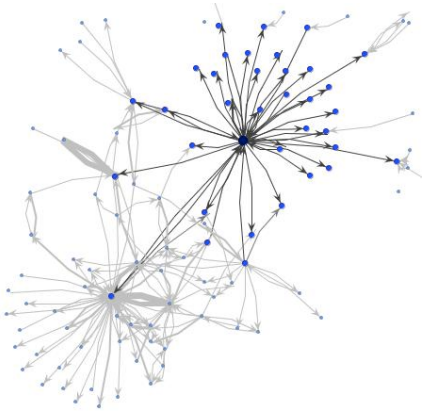
- Structure from Motion iteratively recovers the camera parameters for each image and then triangulates the feature tracks in 3D
- Bundle Adjustment is used to globally optimize the camera parameters and track locations in order to minimize the overall re-projection error

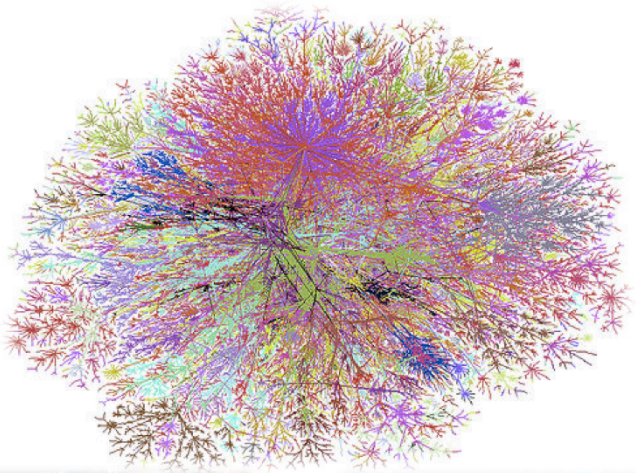


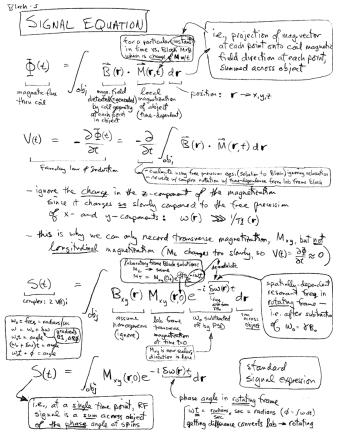
Cloud-based Computer Vision Architecture



What's behind Google Search?







Benefits of Information Hiding

- Allows modules to be developed, tested, optimized, used, understood and modified in isolation
  - Development in parallel
  - Ease maintenance burden
  - Effective performance tuning and debugging
  - Increases software reuse
  - Reduce system development risk

Rule of Thumb

Make each class or member as inaccessible as possible.

In other words, use the lowest possible access level consistent with the proper functioning of the software that you are writing.

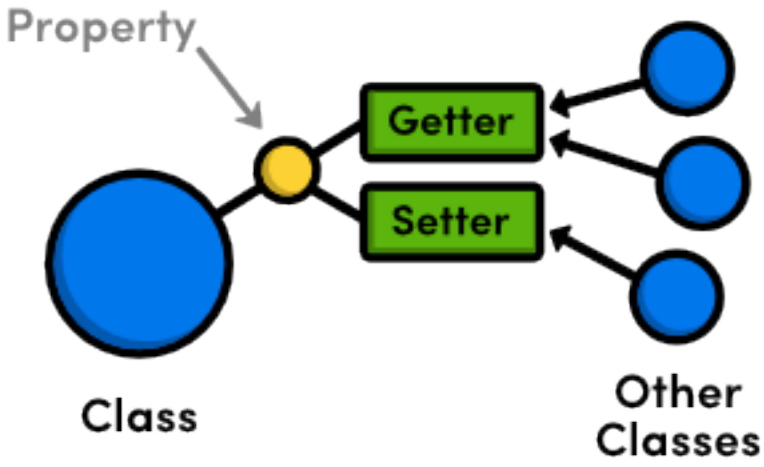
- Exposing too many public members violates the principle of information hiding
  - Exposing members requires others to understand them
- Exposed members are contracts
  - The methods may be used by other modules, and you have to support and maintain the compatibility forever
  - Private members are safe and flexible to change

Accessibility Suggestions

- Carefully decide the public APIs (methods) – make them as few as possible
- Make the rest of methods private
- When other classes in the same package really need certain access, make it as package-private (default)
  - If you see doing this too often, there might be a chance for a decomposition of the class
- A huge increase in accessibility occurs when the access level from package-private to protected
  - A protected method is part of the exposed API and must be supported forever

1. Minimize the Accessibility of Classes and Members!

2. Use Accessor Methods, or Public Fields ?



Instance Variables Should Never be Public

- By making the instance variable public:
  - You give up the ability to limit and control the values
  - You give up the ability to enforce the invariants involving the field
  - You give up the ability to take any action when the field is modified

```
class Point {
   public double x;
   public double y;
}
```

Rule of Thumb

If a class is accessible outside its package, provide access methods, to preserve the flexibility to change the class's internal representation.

```
class Point {
        private double x;
        private double y;

        public Point(double x, double y) {
                this.x = x;
                this.y = y;
        }

        public double getX() { return x; }
        public double getY() { return y; }
        public void setX(double x) { this.x = x; }
        public void setY(double y) { this.y = y; }
}
```

Using the Accessor Methods

- You can limit and control the values

```
public void setX(double x) {
    if (x >= 0) {
        this.x = x;
    } else {
        thrown new RuntimeException("Invalid Value");
    }
}
```

- You can enforce the invariants involving the field

```
public double getX() {
    if (x > 100 && y > 100) {
        return x % 100;
    }
}
```

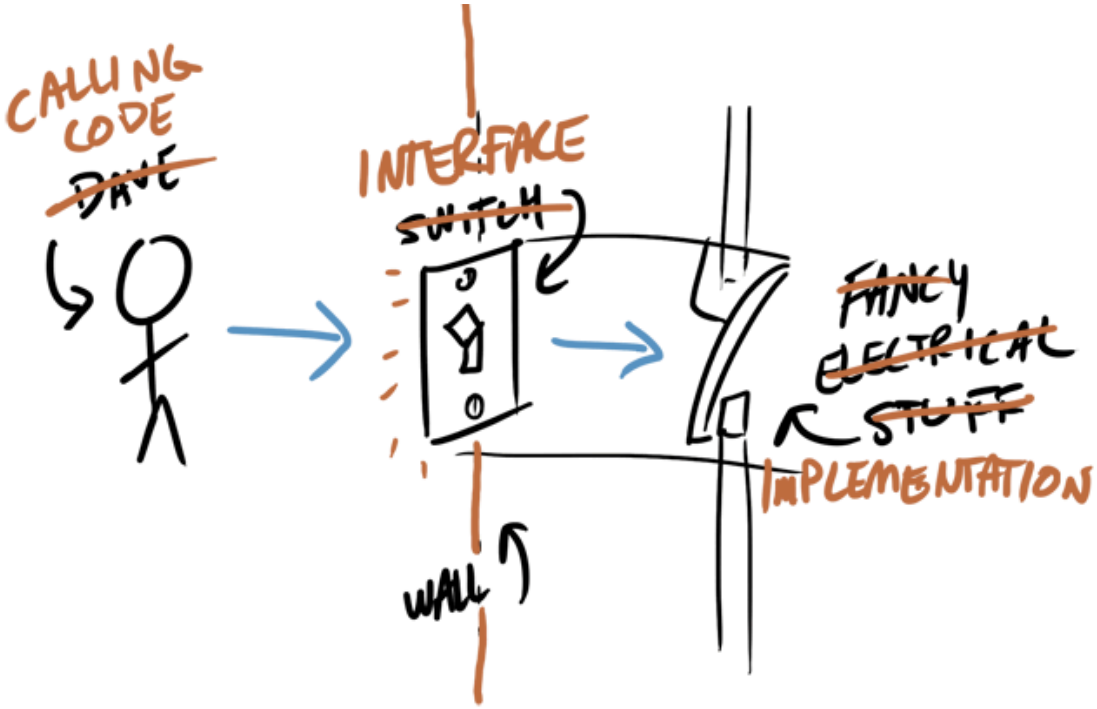
Using the Accessor Methods

- You can limit and control the values
- You can enforce the invariants involving the field
- You can take any action when the field is modified

```
public void setX(double x) {
    this.x = x;
    log.info("User: " + UserManager.getCurrentUser()
        + "modified the field.");
}
```

2. Use Accessor Methods, NOT Public Fields!

3. Program to Interfaces, or Implementation ?



Program to Interface

```
interface IPizza {
 public int getCalories();
}

class PepperoniPizza implements Ipizza {
 public int getCalories() {
   return 500;
 }
}

class CheesePizza implements Ipizza {
 public int getCalories() {
   return 300;
 }
}

class PizzaEater {
 public int getPizzaCalories(IPizza pizza) {
   return pizza.getCalories();
 }
}

PizzaEater eater = new PizzaEater();
eater.getPizzaCalories(new PepperoniPizza());
eater.getPizzaCalories(new CheesePizza());
```

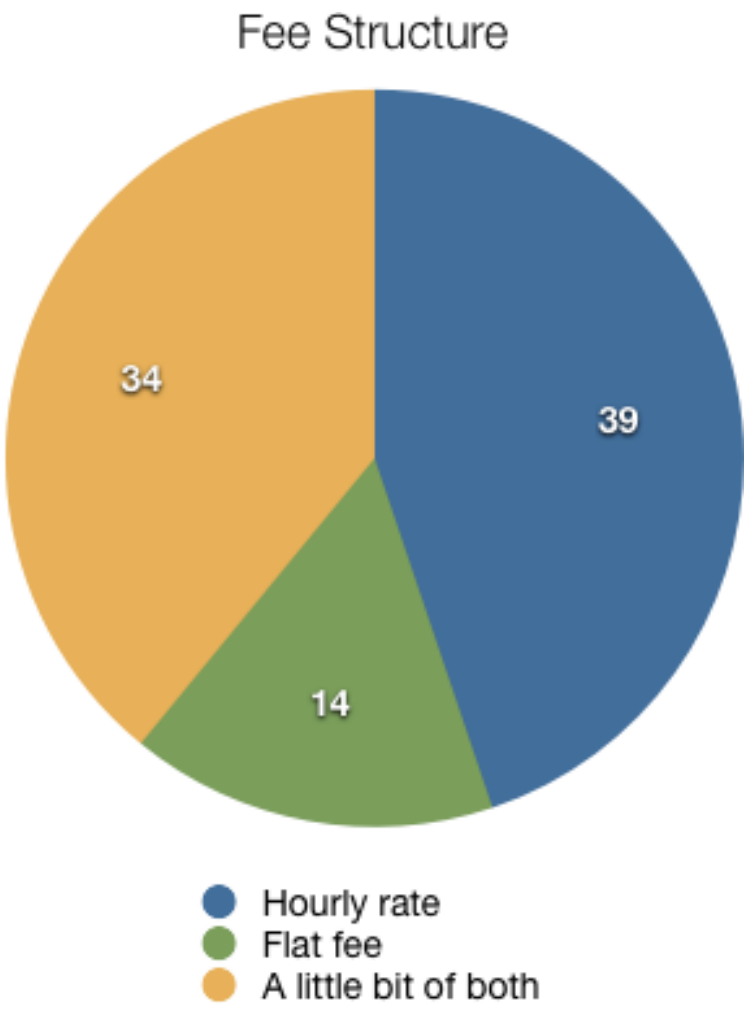
Program to Implementation

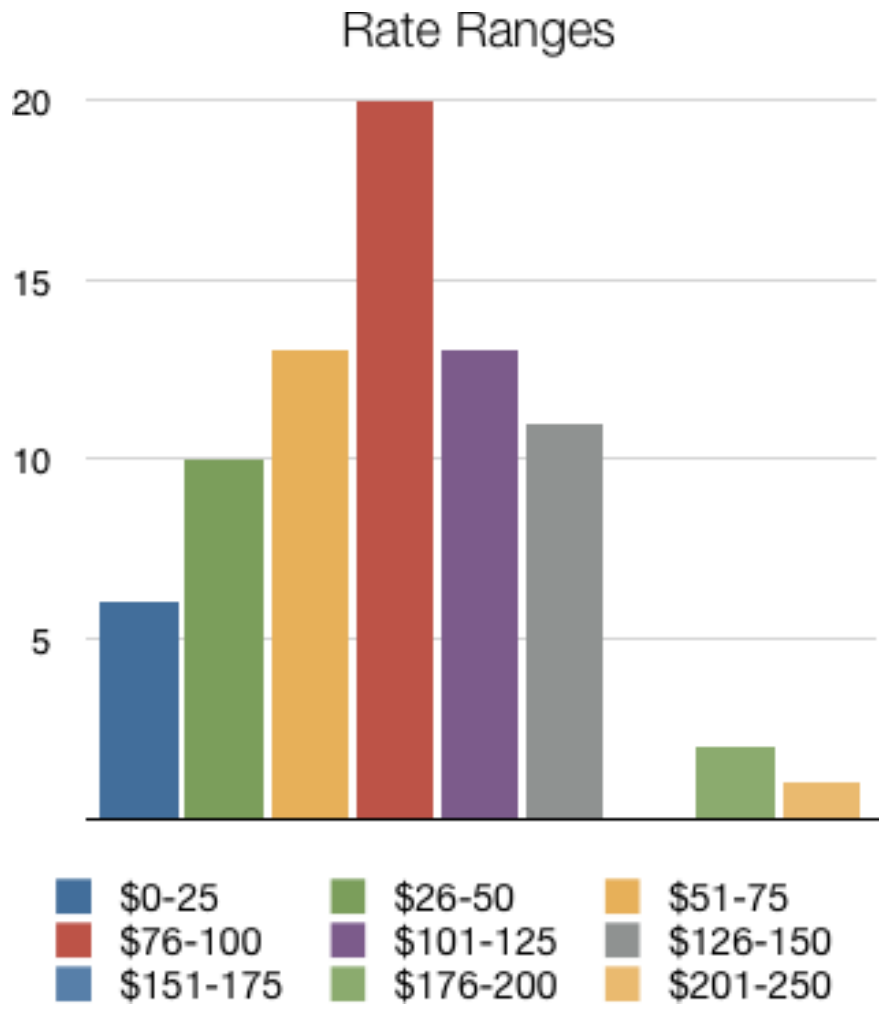
```
class PepperoniPizza {
 public int getCalories() {
   return 500;
 }
}

class CheesePizza {
 public int getCalories() {
   return 300;
 }
}

class PizzaEater {
 public int getPizzaCalories(PepperoniPizza pepperoniPizza) {
   return pepperoniPizza.getCalories();
 }
 public int getPizzaCalories(CheesePizza cheesePizza) {
   return cheesePizza.getCalories();
 }
}

PizzaEater eater = new PizzaEater();
eater.getPizzaCalories(new PepperoniPizza());
eater.getPizzaCalories(new CheesePizza());
```

Benefits of Programming to Interface

- Increased code reuse
- Improved flexibility to extend
- Since a lot of programmers are paid by the hour, the more time we spend writing reusable code and the less time we spend maintaining old code, the better

Developers Hourly Rate

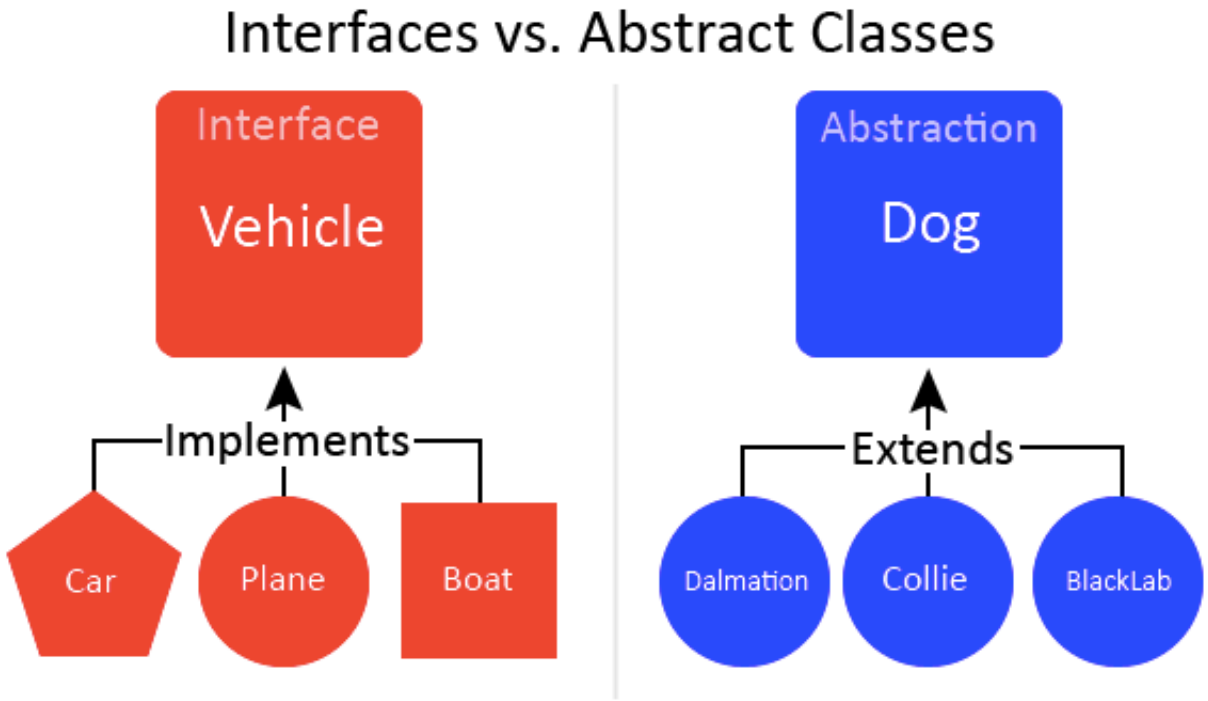




- iOS Developers Rate Survey
- http://mobileorchard.com/ios-rate-survey-results/

3. Program to Interfaces, NOT Implementation!

4. Prefer Interfaces, or Abstract Class ?



Ease Retrofitting Existing Classes

- Interfaces allow any class to be retrofitted
  - *Comparable* interface
- Extending an abstract class may cause conflicts with existing methods
- If we want 2 classes to extend the same abstract class, we have to put the abstract class high up in the ancestor of both classes.

Ideal for Defining Mixins

- A *mixin* is a type that a class can implement in addition to its "primary type" to declare that it provides some optional behavior
  - *Comparable, Cloneable, Serializable*
- A class cannot have more than one parent class.
- There is no reasonable place in the class hierarchy to insert a mixin.

Allow Nonhierarchical Type Frameworks

- Organize some things, but other things don't fall neatly into a rigid hierarchy

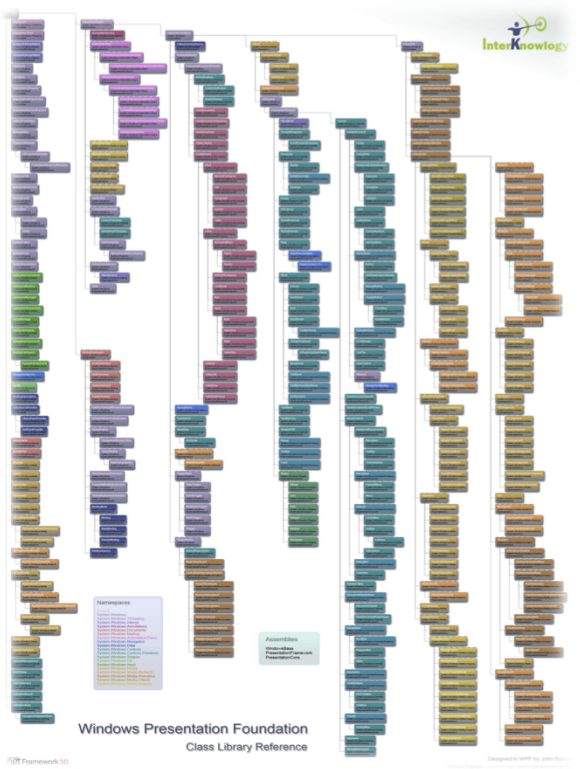
```
public interface Singer {
    AudioClip sing(Song s);
}

public interface Songwriter {
    Song compose(boolean hit);
}

public interface SingerSongwriter
       extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}
```

Things to be Cautions about Interfaces

- It is far easier to evolve an abstract class than an interface
  - Adding a concrete method containing a reasonable default implementation in an abstract class will not affect existing classes.
  - Adding a new method to a public interface will break all the existing classes that implement that interface.
- Once an interface is released and widely implemented, it is almost impossible to change
  - If an interface is severely deficient, it can doom an API.

4. Prefer Interfaces to Abstract Class!

5. Prefer Class Hierarchies, or Tagged Classes?



Class Hierarchy

```
abstract class Figure {
    abstract double area();
}

class Circle extends Figure {

    final double radius;

    Circle(double radius) { this.radius = radius; }
    double area() { return Math.PI * (radius * radius); }
}

class Rectangle extends Figure {

    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    double area() { return length * width; }
}
```

Tagged Class Example

```
class Figure {
    enum Shape { RECTANGLE, CIRCLE };

    // Tag field - the shape of this figure
    final Shape shape;

    // These fields are used only if shape is RECTANGLE
    double length;
    double width;

    // This field is used only if shape is CIRCLE
    double radius;

    // Constructor for circle
    Figure(double radius) {
        shape = Shape.CIRCLE;
        this.radius = radius;
    }

    // Constructor for rectangle
    Figure(double length, double width) {
        shape = Shape.RECTANGLE;
        this.length = length;
        this.width = width;
    }
```

Tagged Class Example

```
    ……

    double area() {
        switch(shape) {
            case RECTANGLE:
                return length * width;
            case CIRCLE:
                return Math.PI * (radius * radius);
            default:
                throw new AssertionError();
        }
    }

}
```

- Tagged classes are verbose, error-prone, and inefficient
  - Poor readability
  - Constructor sets the right data and type
  - Keep unused data in memory

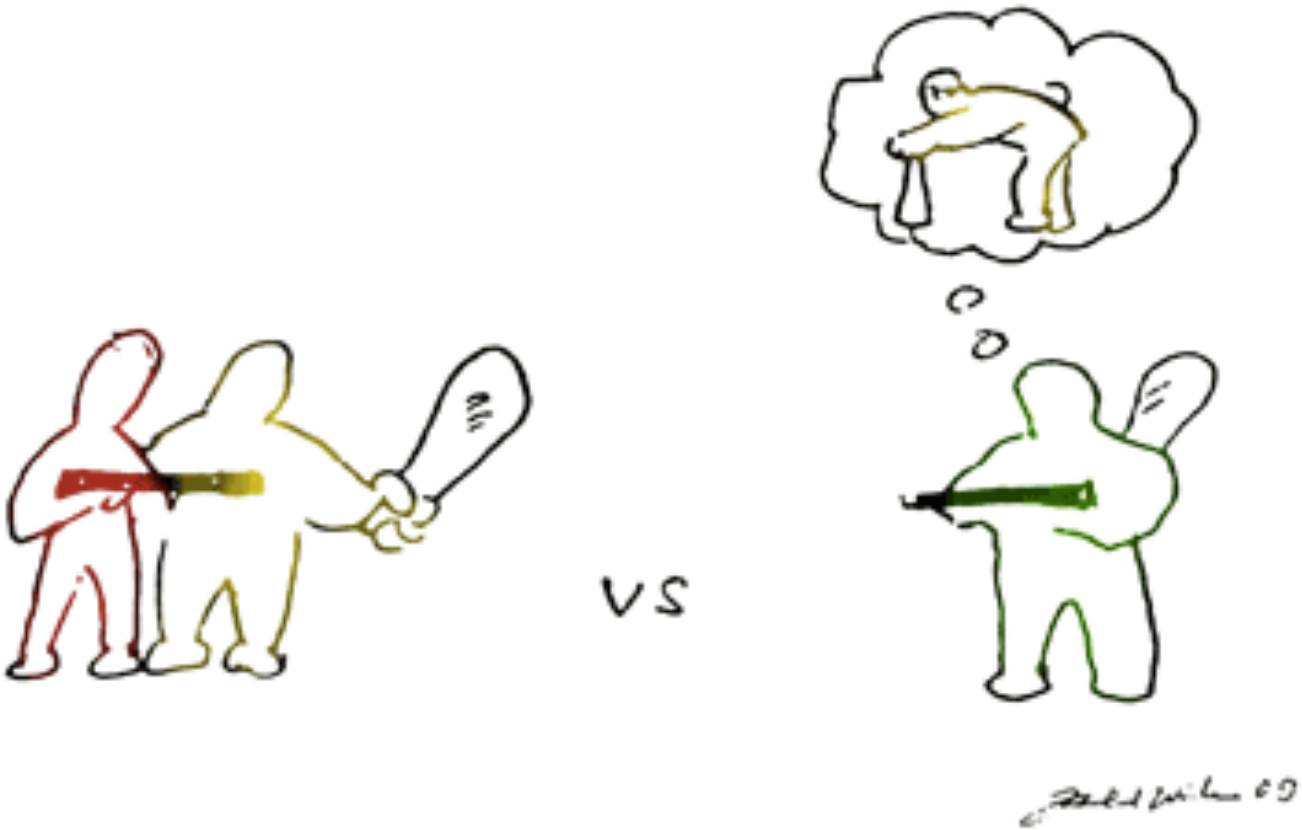
Class Hierarchy

- Simple and clear code
- No irrelevant data
- Final fields
- No runtime error due to missed type
- Separated type and data fields
- Increased flexibility and compile-time type checking

```
class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }
}
```

5. Prefer Class Hierarchies to Tagged Classes!

6. Favor Composition, or Inheritance ?



Inheritance is Powerful, but …

- Good reuse but has problems

Inheritance Violates Encapsulation

- A subclass depends on the implementation details of its superclass for its proper function
- The superclass's implementation may change from release to release, and if it does, the subclass may break, even though its code has not been touched
- A subclass must evolve in tandem with its superclass

Example – Count Added Elements

```
public class InstrumentedHashSet<E> extends HashSet<E> {

    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }

    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

Example – Count Added Elements

```
InstrumentedHashSet<String> s =
        new InstrumentedHashSet<String>();

s.addAll(Arrays.asList("Snap", "Crackle", "Pop"));
```

- What's the output?

- Internally, *addAll* method is implemented on top of its *add* method, although it is not documented in detail

Potential Problems of Inheritance

- Without knowing the actual implementation details, inheritance could cause unexpected errors
- Newly added methods in superclass could change the semantics of the superclass, but the subclasses never know
- A new method in the superclass has a duplicated signature but different return type

Use Composition

- Instead of extending an existing class, give your new class a private field that references an instance of the existing class
- Each instance method in the new class invokes the corresponding method on the contained instance of the existing class and returns the results. This is known as *forwarding*

Example – Use Composition

```
public class ForwardingSet<E> implements Set<E> {

 private final Set<E> s;

 public ForwardingSet(Set<E> s) { this.s = s; }

 public void clear() { s.clear(); }
 public boolean contains(Object o) { return s.contains(o); }
 public boolean isEmpty() { return s.isEmpty(); }
 public int size() { return s.size(); }
 public Iterator<E> iterator() { return s.iterator(); }
 public boolean add(E e) { return s.add(e); }
 public boolean remove(Object o) { return s.remove(o); }
 public boolean addAll(Collection<? extends E> c)
  { return s.addAll(c); }
 public boolean removeAll(Collection<?> c)
  { return s.removeAll(c); }
 public boolean retainAll(Collection<?> c)
  { return s.retainAll(c); }
 public Object[] toArray() { return s.toArray(); }
 public <T> T[] toArray(T[] a) { return s.toArray(a); }
 @Override public boolean equals(Object o)
  { return s.equals(o); }
}
```

Example – Use Composition

```
public class InstrumentedSet<E> extends ForwardingSet<E> {

 private int addCount = 0;

 public InstrumentedSet(Set<E> s) {
  super(s);
 }

 @Override public boolean add(E e) {
   addCount++;
   return super.add(e);
 }

 @Override public boolean addAll(Collection<? extends E> c) {
   addCount += c.size();
   return super.addAll(c);
 }

 public int getAddCount() {
   return addCount;
 }
}
```

6. Favor Composition Over Inheritance!