Principles of Software Construction: Objects, Design, and Concurrency

Part 5: Large-scale Reuse

Principles of API Design

Jonathan Aldrich    Charlie Garrod
Administrivia

- Homework 4c due tonight
- Homework 5 coming soon
  - Team sign-up deadline Monday night
- Midterm exam next Thursday
  - Exam review session Tuesday, 7:30 p.m. in DH 1112
Key concepts from Tuesday
Today: API design

- Introduction to APIs: Application Programming Interfaces
- An API design process
- Key design principle: Information hiding
- Concrete advice for user-centered design
Today: API design

- Introduction to APIs: Application Programming Interfaces
- An API design process
- 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"
  - If you have "Java" in your resume you should own Effective Java, our optional course textbook.
Learning goals for today

• Understand and be able to discuss the similarities and differences between API design and regular software design
  – Relationship between libraries, frameworks and API design
  – Information hiding as a key design principle

• Acknowledge, and plan for failures as a fundamental limitation on a design process

• Given a problem domain with use cases, be able to plan a coherent design process for an API for those use cases
  – "Rule of Threes"
API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system
API: Application Programming Interface

• An API defines the boundary between components/modules in a programmatic system
API: Application Programming Interface

• An API defines the boundary between components/modules in a programmatic system
API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system
Libraries and frameworks both define APIs

```java
public MyWidget extends JComponent {
    public MyWidget(int param) { // setup internals, without rendering

        // render component on first view and resizing
        protected void paintComponent(Graphics g) {
            g.setColor(Color.red);
            g.drawRect(0, 0, d.getWidth(), d.getHeight());
        }
    }
}
```

Your code
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

• Poor APIs are a great liability
  – Lost productivity from your software developers
  – Wasted customer support resources
  – Lack of buy-in from clients & customers
Evolutionary problems: Public APIs are forever

- "One chance to get it right"
- You can add features, but never remove or change the behavioral contract for an existing feature

Motivation to create an 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

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

- Define the scope of the API
  - Collect use-case stories, define requirements
  - Be skeptical
    - Distinguish true requirements from so-called solutions
    - "When in doubt, leave it out."
- Draft a specification, gather feedback, revise, and repeat
  - Keep it simple, short
- Code early, code often
  - Write client code before you implement the API
Respect the rule of three

• Via Will Tracz (via Josh Bloch), *Confessions of a Used Program Salesman*:
  – "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 work fine."
Documenting an API

• APIs should be self-documenting
  – Good names drive good design

• Document religiously anyway
  – All public classes
  – All public methods
  – All public fields
  – All method parameters
  – Explicitly write behavioral specifications

• Documentation is integral to the design and development process
Key design principle: Information hiding

• "When in doubt, leave it out."
Documenting an API

• APIs should be self-documenting
  – Good names drive good design

• Document religiously anyway
  – All public classes
  – All public methods
  – All public fields
  – All method parameters
  – Explicitly write behavioral specifications

• Documentation is integral to the design and development process

• Do not document implementation details
Key design principle: Information hiding (2)

- Minimize the accessibility of classes, fields, and methods
  - "You can add features, but never remove or change the behavioral contract for an existing feature"
Key design principle: Information hiding (3)

- Use accessor methods, not public fields
  - Consider:
    ```java
    public class Point {
        public double x;
        public double y;
    }
    ```
  vs.
    ```java
    public class Point {
        private double x;
        private double y;
        public double getX() { /* ... */ }
        public double getY() { /* ... */ }
    }
    ```
Key design principle: Information hiding (4)

- Prefer interfaces over abstract classes
  - Interfaces provide greater flexibility, avoid needless implementation details
  - Consider our earlier example:
    ```java
    public interface Point {
        public double get();
        public double getY();
    }
    
    public class PolarPoint() implements Point {
        private double r; // Distance from origin.
        private double theta; // Angle.
        public double getX() { return r*Math.cos(theta); }
        public double getY() { return r*Math.sin(theta); }
    }
    ```
Key design principle: Information hiding (5)

• Consider implementing a factory method instead of a constructor
  – Factory methods provide additional flexibility
    • Can be overridden
    • Can return instance of any subtype
      – Hides dynamic type of object
    • Can have a descriptive method name
Key design principle: Information hiding (6)

• Prevent subtle leaks of implementation details
  – Documentation
  – Lack of documentation
  – Implementation-specific return types
  – Implementation-specific exceptions
  – Output formats
  – implements Serializable
Minimize conceptual weight

- Conceptual weight: How many concepts must a programmer learn to use your API?
  - APIs should have a "high power-to-weight ratio"
- See java.util.*, java.util.Collections

```java
static <T> Collection<T> synchronizedCollection(Collection<T> c)>Returns a synchronized (thread-safe) collection backed by the specified collection.
static <T> List<T> synchronizedList(List<T> list)Returns a synchronized (thread-safe) list backed by the specified list.
static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)Returns a synchronized (thread-safe) map backed by the specified map.
static <T> Set<T> synchronizedSet(Set<T> s)Returns a synchronized (thread-safe) set backed by the specified set.
static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.
static <T> Collection<T> unmodifiableCollection(Collection<T> c)Returns an unmodifiable view of the specified collection.
static <T> List<T> unmodifiableList(List<? extends T> list)Returns an unmodifiable view of the specified list.
static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)Returns an unmodifiable view of the specified map.
static <T> Set<T> unmodifiableSet(Set<? extends T> s)Returns an unmodifiable view of the specified set.
static <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,? extends V> m)Returns an unmodifiable view of the specified sorted map.
static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)Returns an unmodifiable view of the specified sorted set.
```
Apply principles of user-centered design

- Other programmers are your users
- e.g., "Principles of Universal Design"
  - Equitable use
  - Flexibility in use
  - Simple and intuitive use
  - Perceptible information
  - Tolerance for error
  - Low physical effort
  - Size and space for approach and use
Good names drive good design

• Do what you say you do:
  – "Don't violate the Principle of Least Astonishment"
    public class Thread implements Runnable {
      // Tests whether current thread has been interrupted.
      // Clears the interrupted status of current thread.
      public static boolean interrupted();
    }

Good names drive good design (2)

• Follow language- and platform-dependent conventions
  – Typographical:
    • get_x() vs. getX()
    • timer vs. Timer, HTTPServlet vs HttpServlet
    • edu.cmu.cs.cs214
  – Grammatical:
    • Nouns for classes
    • Nouns or adjectives for interfaces
Good names drive good design (3)

- Use clear, specific naming conventions
  - `getX()` and `setX()` for simple accessors and mutators
  - `isX()` for simple boolean accessors
  - `computeX()` for methods that perform computation
  - `createX()` or `newInstance()` for factory methods
  - `toX()` for methods that convert the type of an object
  - `asX()` for wrapper of the underlying object
Good names drive good design (4)

• Be consistent
  – computeX() vs. generateX()?
  – deleteX() vs. removeX()?
Do not violate Liskov's behavioral subtyping rules

- Use inheritance only for true subtypes
- Favor composition over inheritance

// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);  
    ...
}

public class Properties {
    private final HashTable data = new HashTable();
    public String put(String key, String value) {
        data.put(key, value);
    }
    ...
}
Minimize mutability

- Immutable objects are:
  - Inherently thread-safe
  - Freely shared without concern for side effects
  - Convenient building blocks for other objects
  - Can share internal implementation among instances
  - See `java.lang.String`
Minimize mutability

- Immutable objects are:
  - Inherently thread-safe
  - Freely shared without concern for side effects
  - Convenient building blocks for other objects
  - Can share internal implementation among instances
    - See java.lang.String
- Mutable objects require careful management of visibility and side effects
  - e.g. Component.getSize() returns a mutable Dimension
- Document mutability
  - Carefully describe state space
Overload method names judiciously

- Avoid ambiguous overloads for subtypes
  - Recall the subtleties of method dispatch:
    ```java
    public class Point() {
        private int x;
        private int y;
        public boolean equals(Point p) {
            return this.x == p.x && this.y == p.y;
        }
    }
    ```
- If you must be ambiguous, implement consistent behavior
  ```java
  public class TreeSet implements SortedSet {
      public TreeSet(Collection c);  // Ignores order.
      public TreeSet(SortedSet s);   // Respects order.
  }
  ```
Use consistent parameter ordering

- An egregious example from C:
  ```c
  char* strncpy(char* dest, char* src, size_t n);
  void   bcopy(void* src, void* dest, size_t n);
  ```
Avoid long lists of parameters

• Especially avoid parameter lists with repeated parameters of the same type
  
  HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName,
                   DWORD dwStyle, int x, int y, int nWidth, int nHeight,
                   HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
                   LPVOID lpParam);

• Break up the method or use a helper class to hold parameters instead
Fail fast

- Report errors as soon as they are detectable
  - Check preconditions at the beginning of each method
  - Avoid dynamic type casts, run-time type-checking

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

  // Throws ClassCastException if this instance
  // contains any keys or values that are not Strings
  public void save(OutputStream out, String comments);
}
```
Avoid behavior that demands special processing

- Do not return null to indicate an empty value
  - e.g., Use an empty Collection or array instead
- Do not return null to indicate an error
  - Use an exception instead
- Do not return a String if a better type exists
- Do not use exceptions for normal behavior
- Avoid checked exceptions if possible
  
  ```java
  try {
    Foo f = (Foo) g.clone();
  } catch (CloneNotSupportedException e) {
    // Do nothing. This exception can't happen.
  }
  ```
Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```java
public class Throwable {
    public void printStackTrace(PrintStream s);
}
```
Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

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

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

• Accept the fact that you, and others, will make mistakes
  – Use your API as you design it
  – Get feedback from others
  – Hide information to give yourself maximum flexibility later
  – Design for inattentive, hurried users
  – Document religiously