SD 2007 BEST PRACTICES CONFERENCE & EXPO

September 18-21, 2007
Hynes Convention Center
Boston, MA

User Interface Principles in API Design

Elliotte Rusty Harold
elharo@metalab.unc.edu
http://www.cafeaulait.org/

the future of software development

“API usability is the intersection of user-centered design and excellent coding practices”

- **--David Koelle & Geertjan Wielenga**

Programmers Are People Too

- Eat Like Humans
- Sleep Like Humans
- *Think* Like Humans

User Interface Design is a Science

- Based on hypothesis, observation and experiment
- Well-proven, well-tested theories

Fundamental Principles

- Consistency is next to godliness
- Simpler is better
- Visible complexity is bad
- Smaller equals easier to use

Libraries vs. Applications

- Applications are monolithic
- Only other programmers on the same team use an application’s API
- Libraries can make very limited assumptions about how, when, where, and why API will be invoked
- Boundary is fuzzy

Remember the *People*

- Why you need an API
- Who uses the API
- Who designs the API

Focus on the *User*

- Ask what the user wants to do with your API
- Do not ask what the internal data structures and algorithms look like
- High level API is better than lower level--Reduce the number of method calls needed to accomplish the task
- Design from the outside in
- Start with the end in mind

What to put in an API

- Write sample programs first; Sample-first programming
- 80/20 rule
- Maximal vs. Minimal APIs
- YAGNI
- When in doubt, leave it out!
- Why I’m a conservative

Dependencies

- Platform version
- Library dependencies
- Built-in vs. 3rd party libraries

Data Encapsulation

- Public vs. Published
- Fields are private
- Methods are mostly private
- Methods are atomic
- Constructors and destructors
- Communicating with the user

Constraints

- APIs must enforce domain validity
- Preconditions
- Postconditions
- Class invariants
- System invariants
- Construct complete objects only (Builder pattern)

Error Handling

- Specify what happens on bad input as well as good
- Important for security
- No undefined behavior
- Don’t silently swallow exceptions
- Error messages should be verbose but clear
- Don’t warn the user

Naming Conventions

- Review naming conventions
- Use standard terminology
- Do not abbreviate
- Use domain specific vocabulary
- Consistent terminology: always use the same word for the same idea
  - e.g. add vs. append
- Do not use two words for one idea

Avoid Complexity

- Prefer classes to interfaces
- Prefer constructors to factory methods
- Avoid excessive abstraction
- You usually don’t need multiple implementations
- Refactor to patterns; don’t start with them. Avoid pattern overload!

Inheritance

- Prefer finality
  - – (at least on methods)
- Factories and interfaces
- The proper use of protected

Plays well with others (Java):

- Serializable
- Cloneable(*)
- Comparable
- equals()
- hashCode()
- toString()
- Exception handling
- Thread safety

Plays well with others (.NET):

- Equals() / GetHashCode()
- ToString()
- IEquatable<T> / IComparable<T>
- “Collection” suffix for IEnumerable classes
- Icloneable*
- Override ==, etc. for value types (only)
- No pointer arguments to public methods
- Don’t throw exceptions from overloaded operators and implicit casts

Testability

- The API itself
- Client code that uses the API
- This is a *secondary* concern

Documentation

- Specification
- Quick Start
- Tutorials
- Example code
- API Documentation
- Per method checklist

Conformance Testing

- Specifications
- Test Suites
- Implementation dependent behavior
- Implementation dependent extensions

Maintenance

- Planning for the future
- Forwards compatibility
- Backwards compatibility
- Unexpected limits
- Deprecation
- Breaking compatibility
- Interfaces vs. classes

The Last Concern (Performance)

- Speed
- Size
- Energy

Case Study: JMidi vs. JFugue

JMidi: Play Middle-C

```
Sequencer sequencer = MidiSystem.getSequencer();
Sequence sequence = sequencer.getSequence();
Track track = sequence.createTrack();
ShortMessage onMessage = new ShortMessage();
onMessage.setMessage(ShortMessage.NOTE_ON, 0, 60, 128);
MidiEvent noteOnEvent = new MidiEvent(onMessage, 0);
track.add(noteOnEvent);
ShortMessage offMessage = new ShortMessage();
offMessage.setMessage(ShortMessage.NOTE_OFF, 0, 60, 128);
MidiEvent noteOffEvent = new MidiEvent(offMessage, 200);
track.add(noteOffEvent);
sequencer.start();
try {
  Thread.sleep(track.ticks());
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
} // courtesy of David Koelle
```

JFugue: Play Middle C

```
Player player = new Player();
player.play("C");

// Play first 2 measures (and a bit) of “Für Elise”
player.play("E6s D#6s | E6s D#6s E6s B5s D6s C6s | A5i.");
// courtesy David Koelle
```

Lessons Learned

- Domain Specific Language
  - Takes advantage of domain specific knowledge
  - Easier to write; easier to read
  - Java is not the right notation for all use cases (nor is XML, nor Ruby, nor JSON, nor SQL, nor…)
- Focus on what the client wants to do; not how the software does it
- Avoid Abstract Factory; don't catch "patternitis"

Case Study: Java Message Service

- To put a message on queue:

```
String queueName = null;
Context jndiContext = null;
QueueConnectionFactory  queueConnectionFactory = null;
QueueConnection         queueConnection = null;
QueueSession            queueSession = null;
Queue                   queue = null;
QueueSender             queueSender = null;
TextMessage             message = null;
final int              NUM_MSGS;
queueName = new String(args[0]);
try {
     jndiContext = new InitialContext();
} catch (NamingException e) {
  System.exit(1);
}
```

Continued

```
try {
   queueConnectionFactory = (QueueConnectionFactory)
   jndiContext.lookup("QueueConnectionFactory");
   queue = (Queue) jndiContext.lookup(queueName);
} catch (NamingException e) {
   System.exit(1);
}
try {
   queueConnection =
    queueConnectionFactory.createQueueConnection();
   queueSession =
    queueConnection.createQueueSession(false,
                                Session.AUTO_ACKNOWLEDGE);
   queueSender = queueSession.createSender(queue);
   message = queueSession.createTextMessage();
   message.setText("This is message 1");
   queueSender.send(message);
 }
```

Finally

```
finally {
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSException e) {}
    }
}
```

What should this look like?

```
try {
  Queue q = new Queue("jms://example.com/message");
  q.send("First message");
  q.send("Second message");
} catch (JMSException ex) {
  System.err.println(ex);
}
```

Case Study: BoxLayout vs. GridBagLayout

Gridbag Calculator

BoxLayout Calculator

Lessons Learned

- Follow naming conventions
- Focus on what the user wants to do; not the internal data model and algorithms

Further Reading

- *Effective Java*: Joshua Bloch
- *Effective C#*: Bill Wagner
- *Framework Design Guidelines*: Krzysztof Cwalina, Brad Abrams
- *Tog on Interface*: Bruce Tognazzini
- *GUI Bloopers*: Jeff Johnson