Writing Maintainable Code

Make your life and other's easier

Andreas - June 2015

What this talk is not about

- How to write scalable code
- How to write high performance code
- How to design re-usable, extensible, long-lived APIs
- How to use Java efficiently
- What Scala teaches us about writing better Java code
- The Paxos Algorithm

There are plenty of books about this.

What this talk *is* about

- How to write maintainable code
- How to write debuggable code
- How to write understandable code
- How to use Javadoc efficiently

Most books don’t talk about this

Why bother?

Have you ever heard this:

- *Good code needs no documentation*
- *API is self-explaining*
- *Good code has no errors*
- *Javadocs are a waste of time because they get out of sync with the code*
- *Error handling makes the code ugly*

Why bother?

Have you ever heard this:

- *Good code needs no documentation*
- *API is self-explaining*
- *Good code has no errors*
- *Javadocs are a waste of time because they get out of sync with the code*
- *Error handling makes the code ugly*
- *…*
- *A good driver does not need a seat belt or air bag*

Why bother

- You work in a large team
  - Cask has two dozen developers
  - One day it may have 2 thousand
  - Cask's code is Open Source
- Your code may live a long time
  - used by developers that you never met
  - used by customers that you never dreamed of
- You don't want to
  - explain your code to others for the next 10
  - wake up to a pager when your code failed
  - debug other people's code who did not bother

Why bother

- Problems will happen
  - In environments that we cannot access
  - In situations that require immediate resolution
  - In ways that you never expected
- When that happens, we have
  - Only the logs
  - Perhaps some metrics
  - With a lot of luck, a heap dump
- Who will deal with it? People who
  - Who don’t write code
  - Don’t know how to read code written by others

Why bother

- you got it?

Goals

- Other Cask developers can understand my APIs
- Customer's developers can understand my APIs
- Cask customer support can debug problems in my code
- Customers can debug problems in my code
- Customers can fix problems not caused by my code
  - Every support ticket costs money
  - A Google search costs (us) nothing
- Customers understand what is going wrong
  - Perception of complexity
  - Perception of not having control
- Empower the person who observes a problem to fix it!

Topics

- Naming
- Javadocs
- Annotations
- Comments
- Logs
- Error Handling
- Testing
- Code analysis

Naming

The name is the first and strongest documentation

- Use descriptive variable/parameter names

```
void setTimeout(long i, long ts)
void setTimeout(long id, long timeout)
void setTimeout(long id, long timeoutSeconds)
```

- Use informative method names

```
void setTTL(long ttl)
void setTimeToLive(long secondsToLive)
```

- Longer names are worth the real estate on your screen

Naming

- Don’t use meaningless Names

```
abstract class BaseWhatever { ...
class SimpleWhatever extends BaseWhatever { ...
class ActualWhatever extends SimpleWhatever { ...
```

- Better

```
abstract class AbstractWhatever { ...
class PersistingWhatever extends AbstractWhatever
class HBaseWhatever extends PersistingWhatever
```

Javadocs

Purpose: Help understand the API

- describe what a method does
- describe what a parameter means
- describe what the return value means
- describe what happens in case of error

For whom?

- Cask developers
- Open source developers
- Customer’s developers

Javadocs

Principles

- Do not document the obvious

```
ConsumerID getConsumerID();
```

- What's obvious to you is not obvious to others

```
public Map<String, String> getProperties() {
  ...
}
public Map<String, String> getResolvedProperties() {
  ...
}
```

(This is actual CDAP code...)

Javadocs

- What's obvious to you is not obvious to others

```
long getTTL()
/**
 * @return the time to live in seconds
 */
long getTimeToLive()
```

- Do document border conditions

```
* @return instance of {@link Row}; never {@code null};
      returns an empty Row if nothing read
```

Javadocs

What deserves Javadocs?

- Public methods and constants of public classes
  - Unless they are self-explaining
  - Getters, setters, default constructor
- Protected methods
  - Important because public to subclasses
- Abstract methods
  - Subclasses must override these methods
- Public classes
- All interfaces and their methods
  - Caller cannot know implementation

Annotations

- Annotations are a short form of commenting:

```
@Nullable
byte[] toBytes(@Nullable String input)
```

- Better than Javadocs?
  - Often makes Javadoc unnecessary

- Do not use `@NotNull` or `@Nonnull`
  - We assume that non-null is expected
  - We document if that is not the case (`@Nullable`)

Annotations

- can also be used to suppress warnings

```
class NoOpPersistence implements Persistence {
    @SuppressWarnings("unused")
    void persist(String key, String value) {
        // do nothing
    }
```

Comments

- Comments help others understand your code
- Focus on the why, not the what

```
switch (programType) {
case ...: { ...
...
}
case WORKFLOW: {
  // can never happen
}
```

- Code (and invariants) change over time

Comments

- Better to say why:

```
switch (programType) {
case ...: { ...
...
}
case WORKFLOW: {
  // can never happen because this method should only
  // be called for real-time pograms
}
```

Comments

- Better to deal with it:

```
switch (programType) {
case ...: { ...
...
}
case WORKFLOW: {
  // this method should only be called for real-time pograms
  throw new IllegalArgumentException(
      "This method should only be called for real-time "
        + " programs, but programType is " + programType);
}
```

- Now the comment is actually unnecessary

Comments

- Don’t comment the obvious

```
// get the consumer id
Id consumerId = getConsumerId(context);
```

- Comment on
  - What the code does
  - How the code does it, and why
  - How it can be improved

```
// TODO: incredibly non-efficient:
//       it is performed for each metrics data point
return value.getTags().hashCode();
```

  - Even better: Include a Jira for the To-Do

```
// TODO: [CDAP-2281] test schedules in a better way
```

Logging

- Some logs are too chatty
- Some logs are too shy

What to log at what level?

- Error: Unexpected Failures that may indicate system failure or serious problems
  - Out of disk space
  - Cannot connect to Database
  - Socket already in use
  - ...
  - A user/client error is *expected* and is at most Info

Logging

What to log at what level?

- Warning: Unexpected Events that are not fatal
  - Zookeeper connection lost
  - Use of a deprecated configuration property
- Info: Expected Events that are useful to the Admin
  - Flow started at time T
  - Reconfiguration request received
  - Version of Hive detected as 0.14

Logging

What to log at what level?

- Debug: Information that helps pinpoint a problem
  - Flowlet uses these three datasets
  - Transaction failed due to conflict. Retrying
  - Invalid client request received
- Trace: Anything that would only distract in normal cases
  - Replaces a debugger
  - Actual parameter values
  - Sequence of execution within a method
  - ...

Logging

Who should log an exception?

- The code that throws it? No!
  - It has the most context
  - Actual values of variables that led to the error
  - But it does not know whether the error is expected
  - Include all meaningful information in the exception
- The code that catches it? Yes!
  - Can decide whether this error is serious
  - Can decide to log the error and continue
  - Can decide to react to the error without logging
  - Can decide to throw a different exception

Messages

- Include all *meaningful* information
- What is meaningful? For example:
  - The context in which an error happens
  - The reason for an error
  - The steps that can be taken to fix it

```
LOG.error("Flow failed to start because of a dataset problem")
```

```
LOG.error(String.format("Flow %s failed to start because of a
                         problem loading dataset %s: ",
                        flowId, datasetName),  exn);
```

Error Handling

- Never ignore an exception
  - Empty catch blocks are evil
  - If you really know that the exception can be ignored, add a comment

```
} catch (AlreadyExistsException e) {
  // another thread has already created it
}
```

  - If you know that this exception can never be thrown, rethrow it using an IllegalStateException

```
} catch (UnsupportedTypeException e) {
  // this should never happen
  throw new IllegalStateException("...", e);
}
```

Exceptions

Java has checked and unchecked exceptions

- Checked Exceptions
  - declared by method using `throws` clause
  - callers of the method must handle them
  - represent *expected* errors
- Unchecked Exceptions
  - need not be declared
  - need not be handled
  - represent *unexpected* or *unrecoverable* errors

Exceptions

Guidelines

- Throw checked exceptions when possible
- Add javadocs for your checked exceptions
  - To help others understand what gets thrown when
- Use unchecked exceptions for errors that are unexpected and cannot be handled
- Do not convert checked exceptions in unchecked ones
  - Avoid `Throwables.propagate()`

```
} catch (IOException e) {
    // This is fatal, since jar cannot be expanded.
    throw Throwables.propagate(e);
}
```

Preconditions

- Purpose
  - Validate the inputs of a method
  - Throw unchecked exceptions
  - This means “internal error”
- Do not use Preconditions on the results of a computation or a method call

```
Preconditions.checkState(job.isSuccessful(),
    "MapReduce execution failure: %s", job.getStatus());
```

- Do not use Preconditions to validate arguments from an external client or a user
  - These are *expected* and should throw meaningful exceptions

Testing

- Always test for border conditions
- Always test negative case
  - Assert that the correct exception is thrown
- If a test case fails, do not `@Ignore` it, but fix it.
- Document your tests
  - Others will maintain and extend your tests
  - Others need to understand how the test works
- Write unit tests where possible
  - Integration tests run much longer
  - Unit tests can provide much better coverage
- Every time you fix a bug: Add a test case

Compiler / IDE Warnings

- These warnings are meaningful
- All code should compile without warnings
  - and no warnings from IntelliJ
- Use `@SuppressWarnings` if you can’t avoid the code that produces the warning
  - add a comment why you think the warning can be suppressed (unless it is obvious)

Conclusion

- We can all write better code
- A minute of work can save hours of support
- These are guidelines and not dogmas
- Everybody should apply common sense
- Every code review should pay attention to this