# Continuous Design

Rob Nagler
bivio Software, Inc.
nagler@bivio.biz

# Continuous Design in the Physical World

# After Enhancement

# Change Happens

- "The Only Constancy is Change Itself ... The first step is to accept the fact of change as a way of life, rather than an untoward and annoying exception."

  -- Fred Brooks, Mythical Man Month, 1975
- Software changes, and you control how
- Continuous design is engineered change

# Overview

- Test-Driven Design (prerequisite):
  - Write test, run it, write code, run test
  - Exponential moving average
  - Self-evident test data
  - Deviance testing and fail fast
- Continuous Design (eng'd change):
  - Refactoring (improving existing code)
  - Enhancement: Simple moving average
  - Defect repair

# Test-Driven Design

- Tests provide structure for change
- New tests specify intention to modify behavior (features and fixes)
- Existing tests validate that all other behavior stays constant
- Test-driven design is an engineering discipline

# Test-First Programming

- Test clarifies how API will work
- Establish usability before implementation
- Well-designed API speeds implementation
- Reproducible feedback simplifies debugging
- Executable documentation

# Write The Test

- Create structure for simplest test:

```
import junit.framework.TestCase;
public class TestEMA extends TestCase
{
  public TestEMA (String name) {
    super(name);
  }
  public void testNothing() {
    EMA ema;
  }
}
```

# Run The Test

- Run test without implementation:

```
% javac TestEMA.java
TestEMA.java:12: cannot resolve symbol
symbol  : class EMA
location: class TestEMA
EMA ema;
          ^
1 error
```

# Satisfy The Test

- Write enough code to satisfy test:

```
public class EMA {
}
```

- Just like fixing bugs: test tells you what's “wrong” so you know what to “fix”

# Run The Test

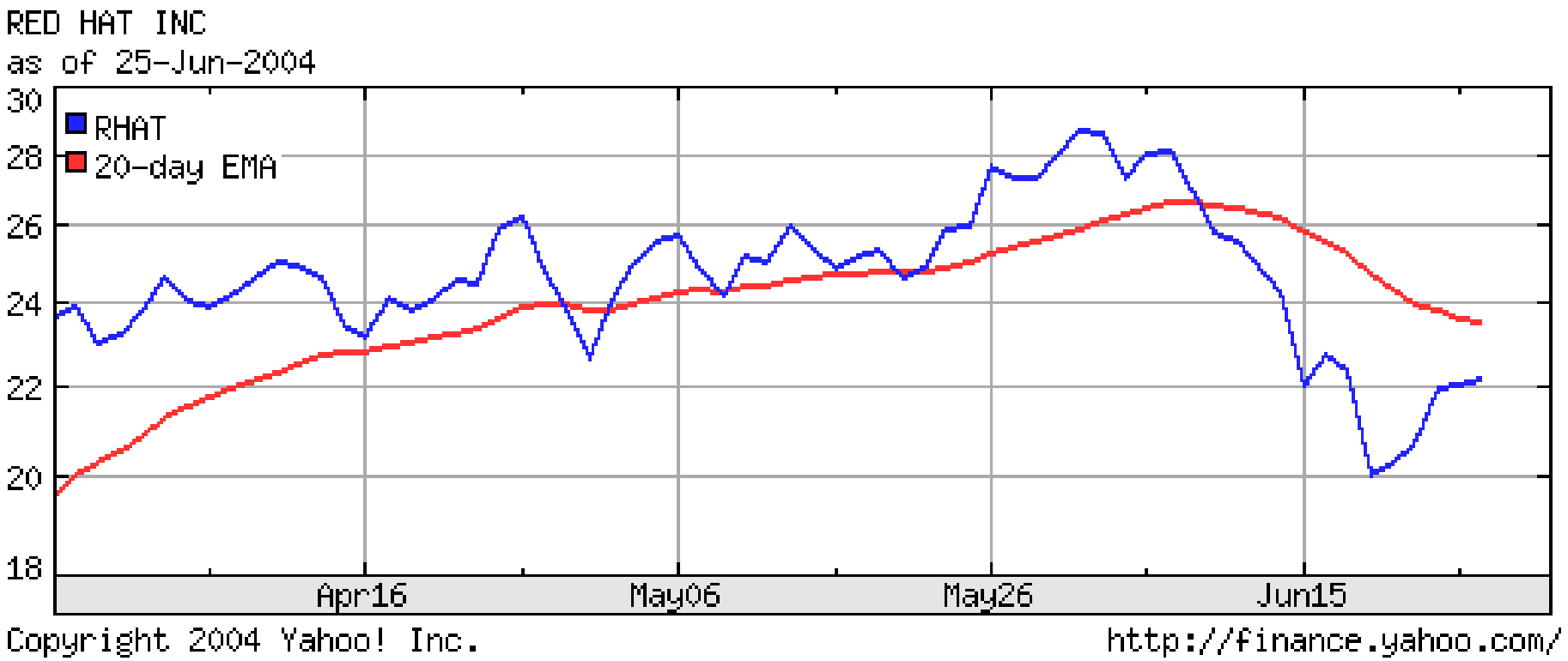
- Run test again to validate code:

```
% javac TestEMA.java
% java junit.textui.TestRunner TestEMA
.
Time: 0.059

OK (1 tests)
```

- Deliberate baby-steps are useful at first
- Get rhythm: test-run-code-run

# Exponential Moving Average



# EMA Algorithm

- Hypothetical customer wants running average of stock prices on website
- Exponential Moving Average (EMA) commonly used smoothing function
- Weighted average with exp. decay: price x weight + average x (1 – weight)
- Weight (alpha):
  2 / (number of days + 1)

# Test What Might Break

- Test that constructor takes an integer:
```
public void testConstructor() {
  new EMA(3);
}
```
- Implement empty constructor:
```
public EMA(int length) {
}
```
- Keep test as simple as possible, but no simpler

# Test Base Cases First

- Validate basic assumptions first:

```
public void testCompute() {
  EMA ema = new EMA(3);
  assertTrue(ema.compute(1) == 1);
  assertTrue(ema.compute(1) == 1);
}
```

- Test progressively, ease reader into subject matter

# Don't Trick Implementation

- Don't do this:

```
public double compute(double v) {
  return 1;
}
```

- Some people recommend it; to me, better to get on with it

# Implement EMA

```
public class EMA {
  private int alpha;
  private boolean isFirst = true;
  private double avg;
  public EMA(int length) {
    alpha = 2 / (length + 1);
  }
  public double compute(double value) {
    if (isFirst) {
      avg = value;
      isFirst = false;
    } else {
      avg = value * alpha
        + avg * (1 – alpha);
    }
    return avg;
  }
}
```

# Choose Self-Evident Data

- Can't run algorithm to get test data
- Test-first requires you to calculate data
- Don't use calculator or other program, rather choose readable data
- Test is ideal documentation if it reveals the algorithm

# Use The Algorithm, Luke!

- Work backwards through algorithm:
  avg = price x alpha + avg x (1 – alpha)
  alpha = 2 / (length + 1)
- In this example, integer length, prices, and averages are most readable
- Alpha is most constrained value

# Solving For Alpha

- Starting at length 1, alpha is: 1, 2/3, 1/2, 2/5, 1/3, 2/7, ...
- 1 is uninteresting case
- 2/3 won't yield integers
- 1/2 yields symmetric equation: price x 0.5 + avg x 0.5
- 2/5 is asymmetric and allows integers: price x 0.4 + avg x 0.6

# Conformance Testing

- Test of “normal” behavior:

```
public void testCompute() {
  EMA ema = new EMA(4);
  assertTrue(ema.compute(5) == 5);
  assertTrue(ema.compute(5) == 5);
  assertTrue(ema.compute(10) == 7);
}
```

- Ignores deviant behavior

# Deviance Testing

- Fail fast when assumptions invalid
- Test of “abnormal” behavior:

```
try {
  new EMA(-2);
  fail("expected exception not thrown");
} catch (IllegalArgumentException e) {}
try {
  new EMA(0);
  fail("expected exception not thrown");
} catch (IllegalArgumentException e) {}
// Test boundary condition
new EMA(1);
```

# Implement Assertions

- Convert assumptions into assertions:

```
public EMA(int length) {
  if (length <= 0) {
    throw new IllegalArgumentException(
      "length must be positive");
  }
  alpha = 2d / (length + 1);
}
```

- Separate concerns: Only test and assert what's unique to the API (e.g. don't test NaN)

# Solid Foundation

- Unit tests form the basis for continuous design
- House held up by continuously validating assumptions

<<< kurze Pause >>>

# Continuous Design

- All systems change even after years of planning
- Change happens in three ways:
  - Enhancements
  - Bug fixes
  - Code reorganization (aka Refactoring)

# Refactoring

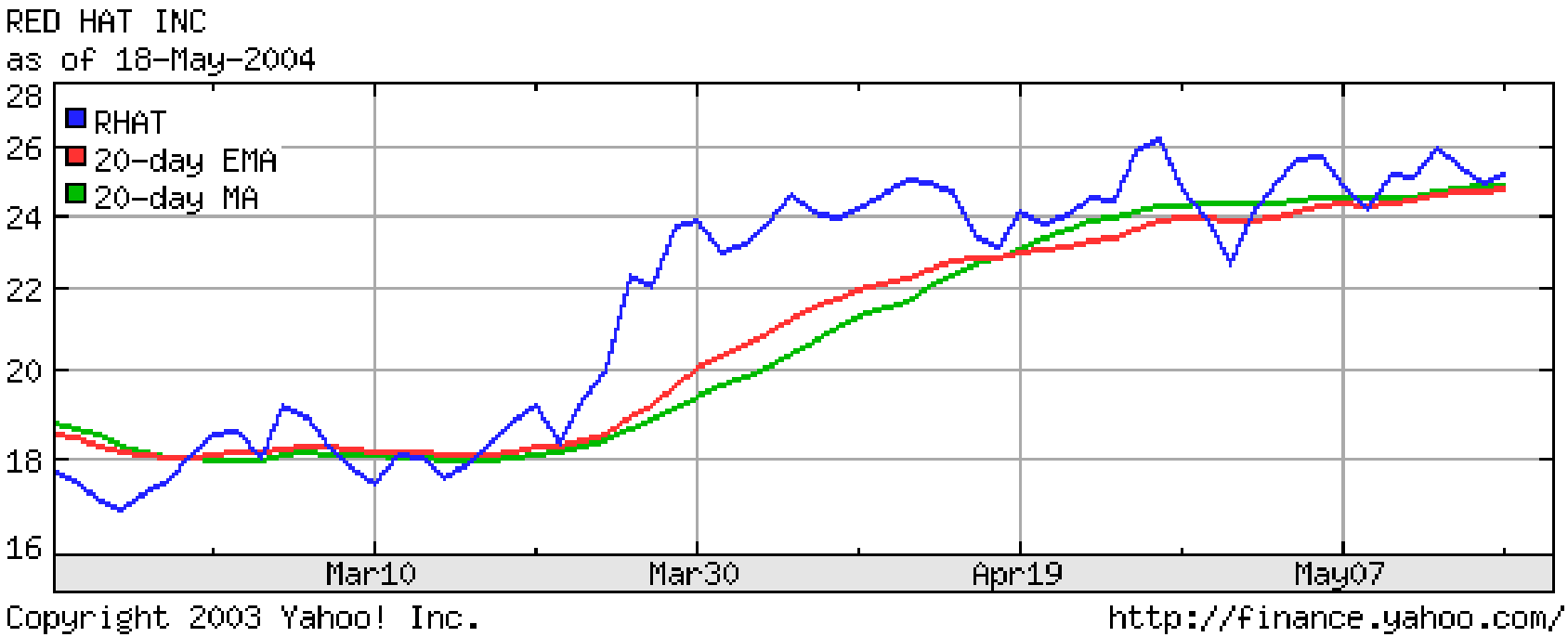
- Refactoring as defined by Fowler:
  - (noun): a change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior.
  - (verb): to restructure software by applying a series of refactorings without changing its observable behavior.

# Simple Refactoring

- Simplify this equation:
```
avg = value * alpha
  + avg * (1 - alpha);
```
- Regroup:
```
avg = (value – avg) * alpha + avg;
```
- Use Java to its fullest:
```
avg += alpha * (value – avg);
```
- Rerun tests after each step

# Simple Moving Average (SMA)

- Customer wants to be like Yahoo!:



# SMA Unit Test

```
import junit.framework.TestCase;
public class TestSMA extends TestCase {
  public TestSMA(String name) {
    super(name);
  }
  public void testConstructor() {
    try {
      new SMA(-2);
      fail("expected exception not thrown");
    } catch (IllegalArgumentException e) {}
    try {
      new SMA(0);
      fail("expected exception was thrown");
    } catch (IllegalArgumentException e) {}
    new SMA(1);
  }
```

**continued on next slide...**

# SMA Unit Test

**...continued from previous slide**

```
  public void testCompute() {
    SMA sma = new SMA(4);
    assertTrue(sma.compute(5) == 5);
    assertTrue(sma.compute(5) == 5);
    assertTrue(sma.compute(11) == 7);
    assertTrue(sma.compute(11) == 8);
    assertTrue(sma.compute(13) == 10);
  }
}
```

# DTSTTCPW: Copy and Paste

- Copy and paste EMA to SMA, and replace all “EMA” with “SMA”
- Run test, which fails:

```
% java junit.textui.TestRunner TestSMA
..F
Time: 0.01
There was 1 failure:
1) testCompute(TestSMA)
junit.framework.AssertionFailedError
at TestSMA.testCompute(TestSMA.java:22)
...snip...
FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

# SMA Implementation

```
import java.util.LinkedList;
import java.util.Iterator;
public class SMA {
  private LinkedList values
    = new LinkedList;
  private int length;
  private double sum = 0;
  public SMA(int length) {
    if (length <= 0) {
      throw new IllegalArgumentException(
        "length must be greater than zero");
    }
    this.length = length;
  }
```

**continued on next slide...**

# SMA Implementation

**...continued from previous slide**

```
  public double compute(double value) {
    if (values.size() == length) {
      sum -= ((Double)values.getFirst())
        .doubleValue();
      values.removeFirst();
    }
    sum += value;
    values.addLast(new Double(value));
    return sum / values.size();
  }
}
```

# Refactoring: Move up to Super Class

- Common refactoring:

```
public class MABase {
  int length;
  public MABase(int length) {
    if (length <= 0) {
      throw new IllegalArgumentException(
        "length must be positive");
    }
    this.length = length;
  }
}
```

- Run tests “as is”

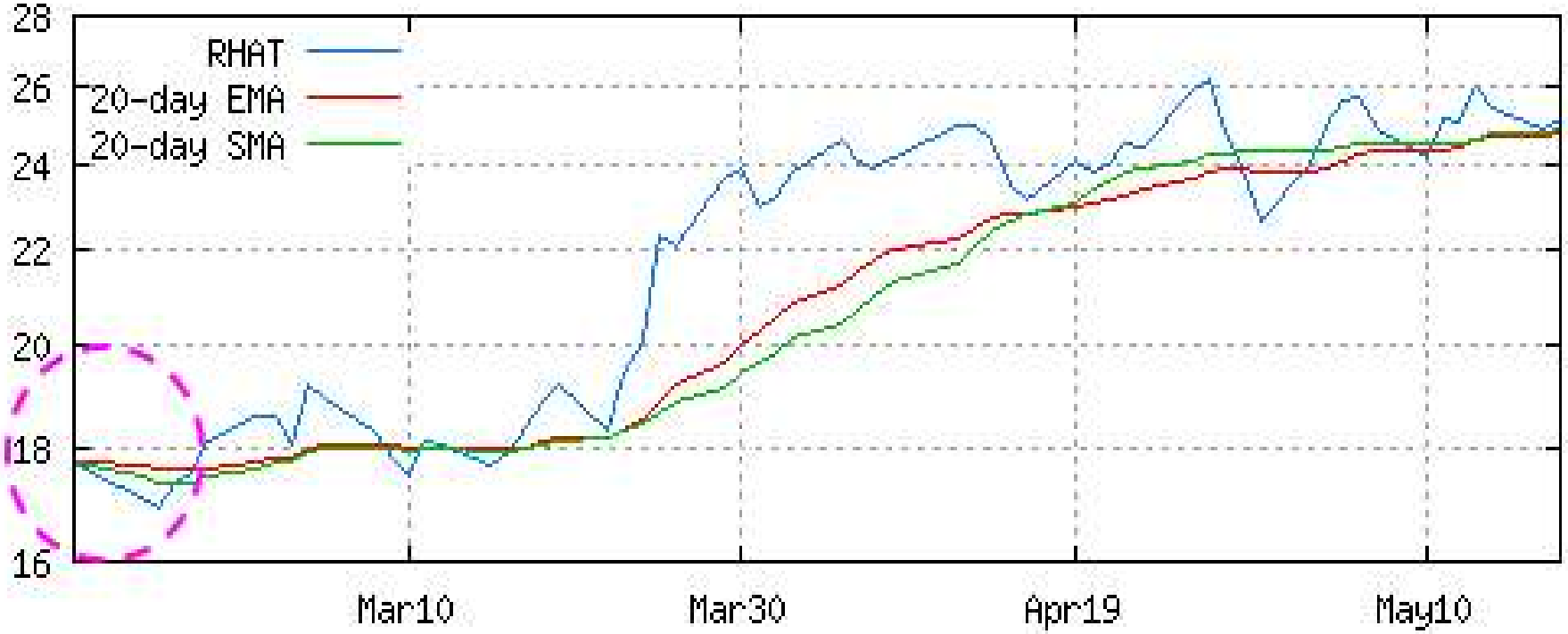
# Refactor Unit Tests

- Just as important to refactor tests:

```
import junit.framework.TestCase;
public class TestMABase extends TestCase {
  public TestMABase(String name) {
    super(name);  }
  public void testConstructor() {
    try {
      new new MABase(-2);
      fail("expected exception");
    } catch (IllegalArgumentException e) {}
    try {
      new new MABase(0);
      fail("expected exception");
    } catch (IllegalArgumentException e) {}
    new MABase(1);
  }}
```

# Woops!

- Customer discovers a defect by comparing with Yahoo! (slide 25):



# Moving Average Build Up

- Moving average requires a build-up period to be correct (stable)
- SMA is straightforward, average stable when all buckets full
- Values influence EMA longer, build-up period is 2x (1%)
- Add getBuildUpLength() so client knows when MA is valid

# Adding buildUpLength

- Add test case to TestSMA:
```
SMA sma = new SMA(4);
assertTrue(sma.getBuildUpLength() == 3);
```
- Add test case to TestEMA:
```
assertTrue(ema.getBuildUpLength() == 8);
```
- Reminder: Run tests before fixing code

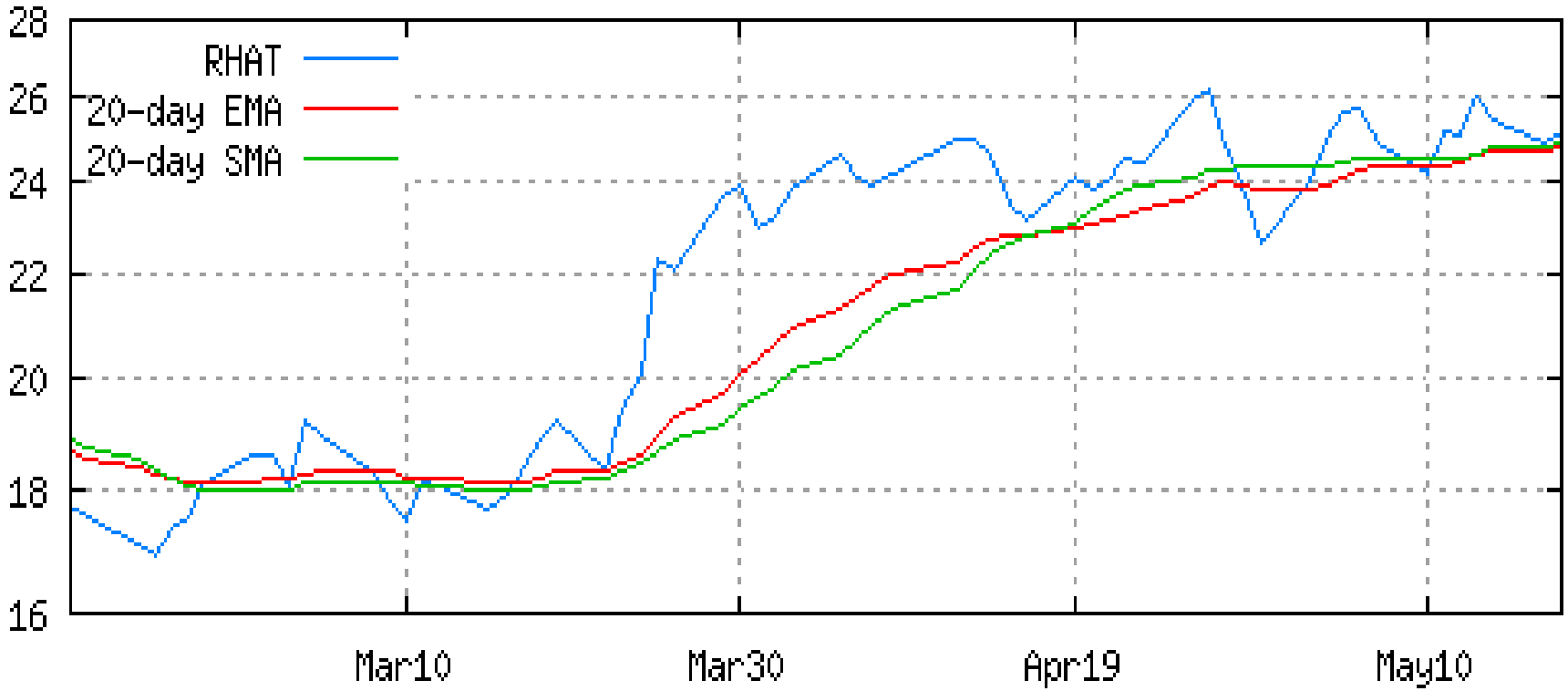
# Adding getBuildUpLength()

- Add buildUpLength to constructor:

```
int buildUpLength;
public MABase(
    int length,
    int buildUpLength
  ) {
  if (length <= 0) {
    throw new IllegalArgumentException(
      "length must be positive");
  }
  this.length = length;
  this.buildUpLength = buildUpLength;
}
```

# Defect Repaired

- Customer happy now:



# Explicit Coupling

- getBuildUpLength is an explicit coupling
- length is used implicitly by clients (e.g. to label graph)
- Explicit couplings easier to change than implicit, add getLength to MABase
- Global Refactoring: API and its clients change simultaneously
- Tests make global refactorings easy

# Add getLength() Tests

- ```
  public void testConstructor() {
    EMA ema = new EMA(4);
    assertEquals(ema.getLength(), 4);
    assertEquals(ema.getBuildUpLength(),
      ema.getLength() * 2);
  }
  ```
- Refactored getBuildUpLength case for better documentation
- Also refactored cases into separate method (formerly in testCompute)
- Take steps at your comfort level
  ```

# Global Refactoring: Get Attribute Value Explicitly

- Change clients first, run tests, expect errors (else harder to find all uses)
- Add to MABase:

```
public int getLength() {
  return length;
}
```

- Run all tests again

# Summary

- Test-driven programming enables continuous engineered design
- Write test, run it, satisfy test, run it
- Tests are spec, first use, and doc
- Implement simply, refactor afterwards
- Refactoring is rigorous discipline validated by continuous testing

# Conclusion

- Change Happens, Be Prepared
- Thank you!
- Questions?