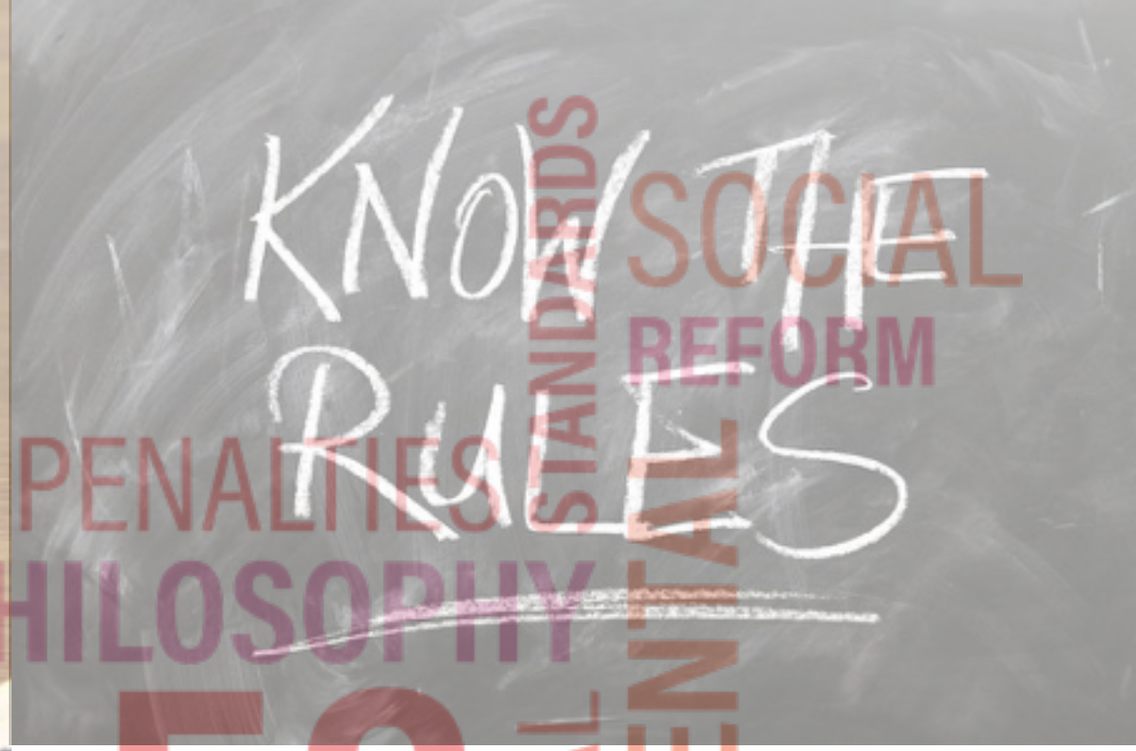
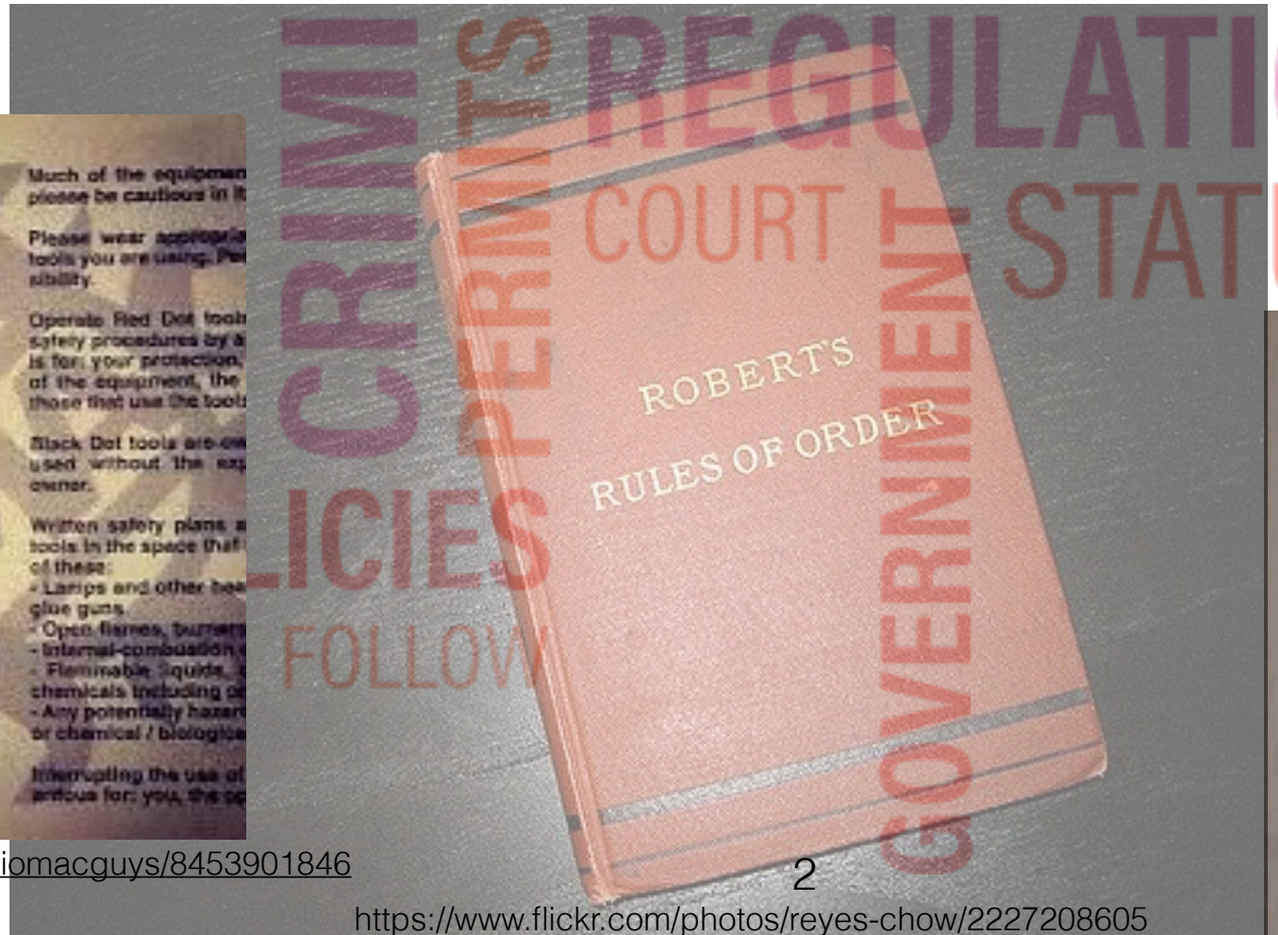
Rules as an Architectural Pattern For Development

Steve Swing
@sswing

http://www.thebluediamondgallery.com/scrabble/r/rules.html

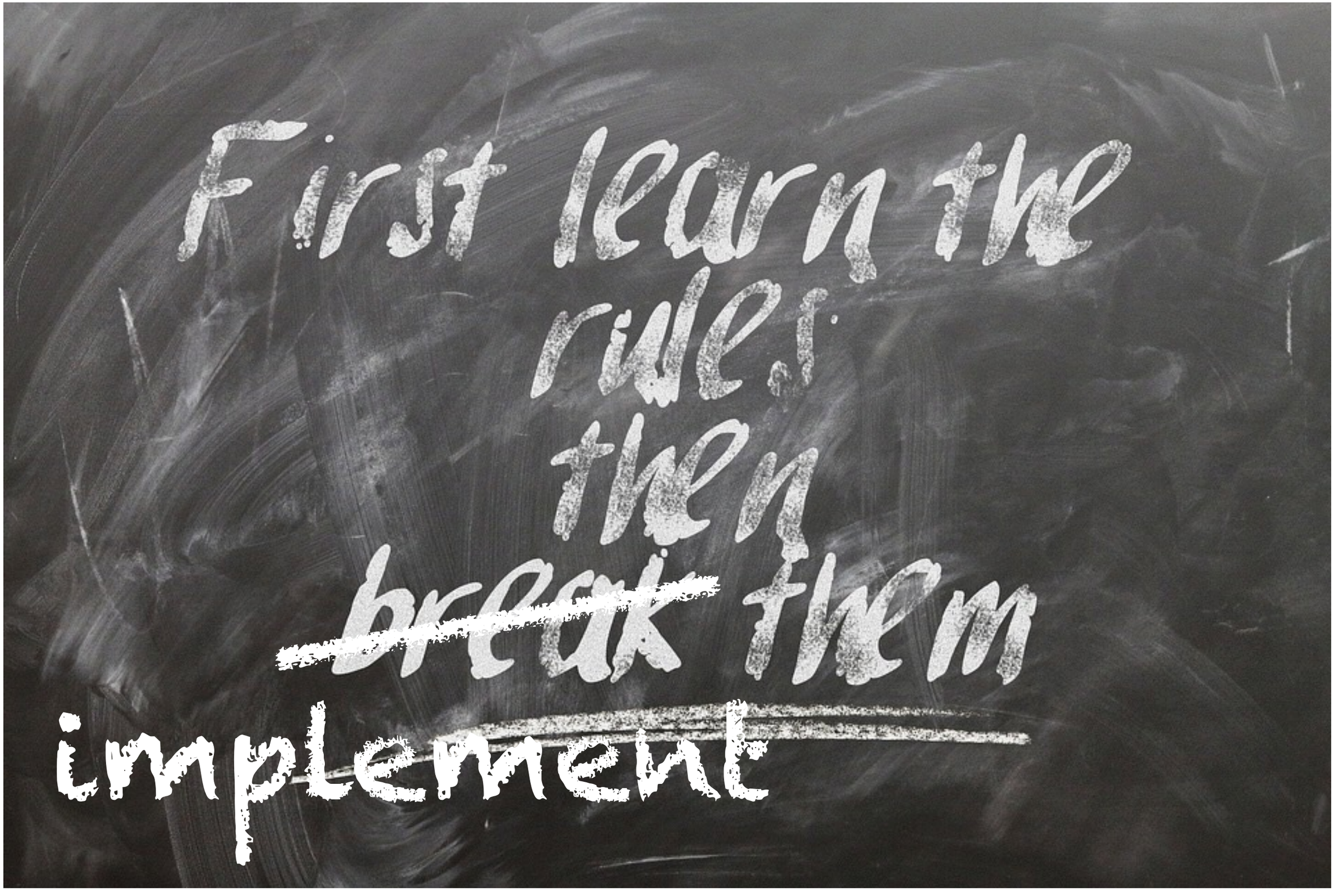


https://www.flickr.com/photos/radiomacguys/8453901846



https://www.flickr.com/photos/reyes-chow/2227208605

http://i.imgur.com/qu68T.jpg


First learn the rules
then
break them
implement

Disclaimer

- Already using a rule engine like Drools? Carry on.
- Don't agree? Learn Drools or other rule engine you like
- However…
  - Use rule design patterns
  - Training
  - Architectural approval
  - External dependencies
  - Control freak

“If you can't be a good example, then you'll just have to be a horrible warning.”

– Catherine Aird

Simple Implementation

- Use Predicates to replace complex if then else logic

```
final CustomerType customerType = customer.getType();
final int totalSales = customer.getTotalSales();
if (CustomerType.SMALL.equals(customerType)) {
    if (totalSales > 1000) {
        // SMALL VIP Customer
    }
} else if (CustomerType.MEDIUM.equals(customerType)) {
    if (totalSales > 5000) {
        // MEDIUM VIP Customer
    }
} else if (CustomerType.LARGE.equals(customerType)) {
    if (totalSales > 10000) {
        // LARGE VIP Customer
    }
}
```

Predicates

```java
/**...*/
@FunctionalInterface
public interface Predicate<T> {

    /**...*/
    boolean test(T t);

    /**...*/
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    /**...*/
    default Predicate<T> negate() { return (t) -> !test(t); }

    /**...*/
    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }

    /**...*/
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
                ? Objects::isNull
                : object -> targetRef.equals(object);
    }
}
```

Predicates

```
public class Predicates {
    public Predicate<Customer> small = c -> CustomerType.SMALL.equals(c.getType());

    public Predicate<Customer> medium = c -> CustomerType.MEDIUM.equals(c.getType());

    public Predicate<Customer> large = c -> CustomerType.LARGE.equals(c.getType());
```

Predicates

```
public class Predicates {
    public Predicate<Customer> small = c -> c.getType() == CustomerType.SMALL;
    public Predicate<Customer> medium = c -> c.getType() == CustomerType.MEDIUM;
    public Predicate<Customer> large = c -> c.getType() == CustomerType.LARGE;
    public Predicate<Customer> smallOrLarge = small.or(large);

    public Predicate<Customer> notMedium = medium.negate();
```

Predicates

```
public class Predicates {
    private static final Map<CustomerType, Integer> salesThresholds = initializeSalesThresholds();
    private static final Instant sixMonthsAgo = offset(systemDefaultZone(), of(-6L, MONTHS)).instant();
    public Predicate<Customer> small = c -> CustomerType.SMALL.equals(c.getType());
    public Predicate<Customer> medium = c -> CustomerType.MEDIUM.equals(c.getType());
    public Predicate<Customer> large = c -> CustomerType.LARGE.equals(c.getType());
    public Predicate<Customer> smallOrLarge = small.or(large);
    public Predicate<Customer> notMedium = medium.negate();
    public Predicate<Customer> vip = c -> c.getTotalSales() > salesThresholds.getOrDefault(c.getType(), MAX_VALUE);
    public Predicate<Customer> mediumVip = medium.and(vip);
    public Predicate<Customer> recentSales = c -> sixMonthsAgo.isBefore(asInstant(c.getRecentSalesDate()));
    public Predicate<Customer> currentMedVip = mediumVip.and(recentSales);

    private static Map<CustomerType, Integer> initializeSalesThresholds() {
        final Map<CustomerType, Integer> result = new TreeMap<>();
        result.put(CustomerType.SMALL, 1000);
        result.put(CustomerType.MEDIUM, 5000);
        result.put(CustomerType.LARGE, 10000);
        return result;
    }

    private Instant asInstant(final LocalDateTime d) { return d.toInstant(of(systemDefault().getId())); }
}
```

What is a rule?

- "When... then..."
- A rule is a list of conditions and a list of actions
- If all conditions evaluate to true the actions are performed
- Pattern match (simple to extremely complex)

What is a rule?

```java
@FunctionalInterface
public interface Rule<T> extends Serializable {
    boolean fire(T t);
}
```

What is a rule?

```java
@FunctionalInterface
public interface Rule<T> extends Serializable {
    default RuleType getRuleType() {
        return RuleType.STANDARD;
    }

    boolean fire(T t);
}
```

What is a rule (impl)?

```java
public class RuleImpl<T> implements Rule<T> {
    private Collection<Condition<T>> conditions = new ArrayList<>();
    private Collection<Action<T>> actions = new ArrayList<>();

    public Collection<Condition<T>> getConditions() {
        return Collections.unmodifiableCollection(conditions);
    }

    public Collection<Action<T>> getActions() {
        return Collections.unmodifiableCollection(actions);
    }

    @Override
    public boolean fire(final T t) {
        if (!conditions.stream().allMatch(c -> c.test(t))) {
            return false;
        }

        actions.forEach(a -> a.perform(t));

        return true;
    }
```

What is a rule (impl)?

```java
public class RuleImpl<T> implements Rule<T> {
    public static final long serialVersionUID = 1L;
    private String id;
    private Collection<Condition<T>> conditions = new ArrayList<>();
    private Collection<Action<T>> actions = new ArrayList<>();
    private RuleType ruleType = RuleType.STANDARD;

    public Collection<Condition<T>> getConditions() { return Collections.

    public Collection<Action<T>> getActions() { return Collections.unmodi

    @Override
    public RuleType getRuleType() { return ruleType; }

    @Override
    public boolean fire(final T t) {
        if (conditions.stream().allMatch(c -> c.test(t))) {
            actions.forEach(a -> a.perform(t));
            return true;
        }
        return false;
    }
```

Conditions & Predicates

```java
@FunctionalInterface
public interface Condition<T> {
    boolean test(T t);
}
```

Conditions & Predicates

```java
/**...*/
@FunctionalInterface
public interface Predicate<T> {

    /**...*/
    boolean test(T t);

    /**...*/
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    /**...*/
    default Predicate<T> negate() { return (t) -> !test(t); }

    /**...*/
    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }

    /**...*/
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
                ? Objects::isNull
                : object -> targetRef.equals(object);
    }
}
```

BaseCondition

```java
public class BaseCondition<T> implements Condition<T> {
    public static final long serialVersionUID = 1L;
    protected boolean affirmative;

    public BaseCondition() { this( affirmative: true); }

    public BaseCondition(final boolean affirmative) { this.affirmative = affirmative; }

    @Override
    public boolean test(final T t) { return !affirmative; }

    @Override
    public boolean equals(final Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        final BaseCondition<?> that = (BaseCondition<?>)o;
        return affirmative == that.affirmative;
    }

    @Override
    public int hashCode() { return Objects.hash(affirmative); }
}
```

Actions & Functions

```java
@FunctionalInterface
public interface Action<T> extends Serializable {
    void perform(T t);
}
```

Actions & Functions

```java
@FunctionalInterface
public interface Consumer<T> {

    /**...*/
    void accept(T t);

    /**...*/
    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

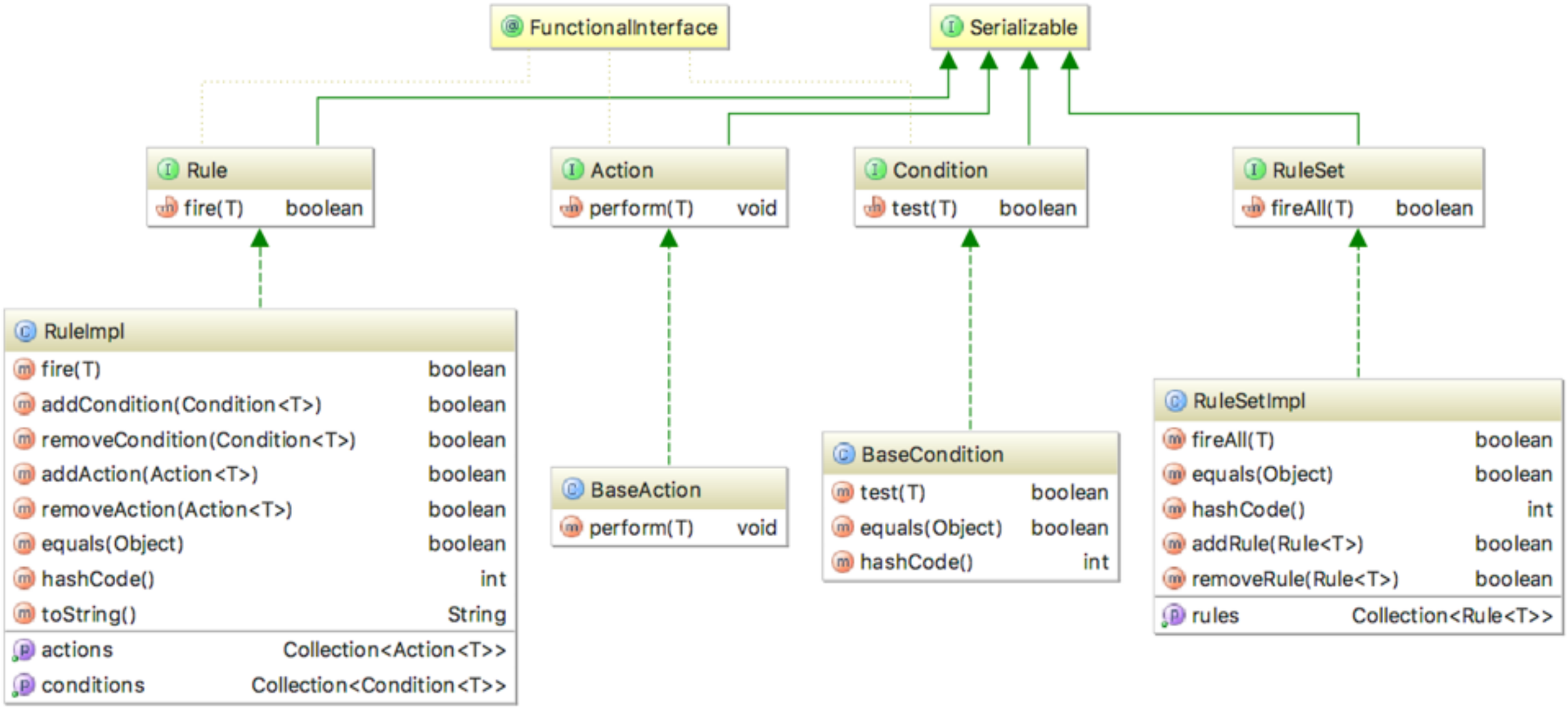
Actions & Functions

```java
/**...*/
@FunctionalInterface
public interface Function<T, R> {

    /**...*/
    R apply(T t);

    /**...*/
    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

    /**...*/
    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    /**...*/
    static <T> Function<T, T> identity() { return t -> t; }
}
```

Actions & Functions

```
public Function<Customer, Customer> applyDiscount = c -> c;
public Function<Customer, Customer> freeShipping = c -> c;

private Map<Predicate<Customer>, Function<Customer, Customer>> initializeActions() {
    final Map<Predicate<Customer>, Function<Customer, Customer>> result = new TreeMap<>();
    result.put(currentMedVip, applyDiscount);
    result.put(recentSales, freeShipping);
    return result;
}
private void applyActions() {
    final Map<Predicate<Customer>, Function<Customer, Customer>> actions = initializeActions();
    final List<Customer> customers = getCustomers();
    for (final Customer customer : customers) {
        actions.entrySet().stream().filter(e -> e.getKey().test(customer)).forEach(e -> e.getValue().apply(customer));
    }
}
```

Moderate Complexity



Moderate Complexity

- Small number of rules
- Number of objects in memory is large
- “Facts” known up front

- Custom Embedded Rule Engine
- Brute Force
- Not Rete (non-inference)
- In memory

Extreme Complexity

- Large number of rules and small number of objects in memory
- Use a inference engine with Rete Algorithm:
    - Open Source
        - Drools
        - Jess (JSR 94: Java Rule Engine API)
    - Commercial ($$$)
        - ILOG JRules
        - Blaze

Rules of Rules

Guiding Principles

- Immutable rules, conditions, actions
- First and final
- Avoid chaining
- Seek rule & rule set independence
- Truth tables
- Short-circuit conditions

Features

- Testing conditions
- Testing Actions
- Tracer capabilities
    - Right answer for the wrong reasons

Testing Conditions

```
public boolean test(final Object o) {
    final LocalDateTime now = LocalDateTime.now();
    return null == effectiveDate && null == expirationDate
            ? affirmative : (!effectiveDate.isAfter(now) && !expirationDate.isBefore(now)) == affirmative;
}

public String toString() {
    String result = "now must ";
    if (effectiveDate != null && expirationDate != null) {
        if (effectiveDate.equals(expirationDate)) {
            return format("%sbe %s%s", result, affirmative ? "exactly " : "any date except ", effectiveDate);
        } else {
            return format("%s%sbe between %s and %s (inclusive)", result,
                    affirmative ? "" : "not ", effectiveDate, expirationDate);
        }
    } else {
        if (effectiveDate == null && expirationDate == null) {
            return "current date may be any value.";
        } else {
            return format("%s%sbe on or %s%s", result, affirmative ? "" : "not ",
                    effectiveDate == null ? "before " : "after ", effectiveDate);
        }
    }
}
```

Testing Conditions

```
public class IsEffectiveTest {
    public Condition<?> _default;
    public Condition<?> affirmative;
    public Condition<?> negative;
    private LocalDateTime effective;
    private LocalDateTime expires;

    @Before
    public void setUp() throws Exception {
        _default = new IsEffective();
        effective = LocalDateTime.now().minusDays(1L);
        expires = LocalDateTime.now().plusDays(1L);
        affirmative = new IsEffective(effective, expires, affirmative: true);
        negative = new IsEffective(effective, expires, affirmative: false);
    }

    @Test
    public void testNull() throws Exception {
        assertTrue( message: "expected true", _default.test( t: null));
        assertTrue( message: "expected true", affirmative.test( t: null));
        assertFalse( message: "expected false", negative.test( t: null));
    }

    @Test
    public void testToString() throws Exception {
        final String expectedDefault = "current date may be any value.";
        assertEquals( message: "expected match", expectedDefault, _default.toString());
        final String expectedAffirmative = format("now must be between %s and %s (inclusive)", effective, expires);
        assertEquals( message: "expected match", expectedAffirmative, affirmative.toString());
        final String expectedNegative = format("now must not be between %s and %s (inclusive)", effective, expires);
        assertEquals( message: "expected match", expectedNegative, negative.toString());
    }
}
```

Serialized Rules

- XStream
  - JSON
  - XML
  - Java Serialization
  - Other

RuntimeRuleLoader

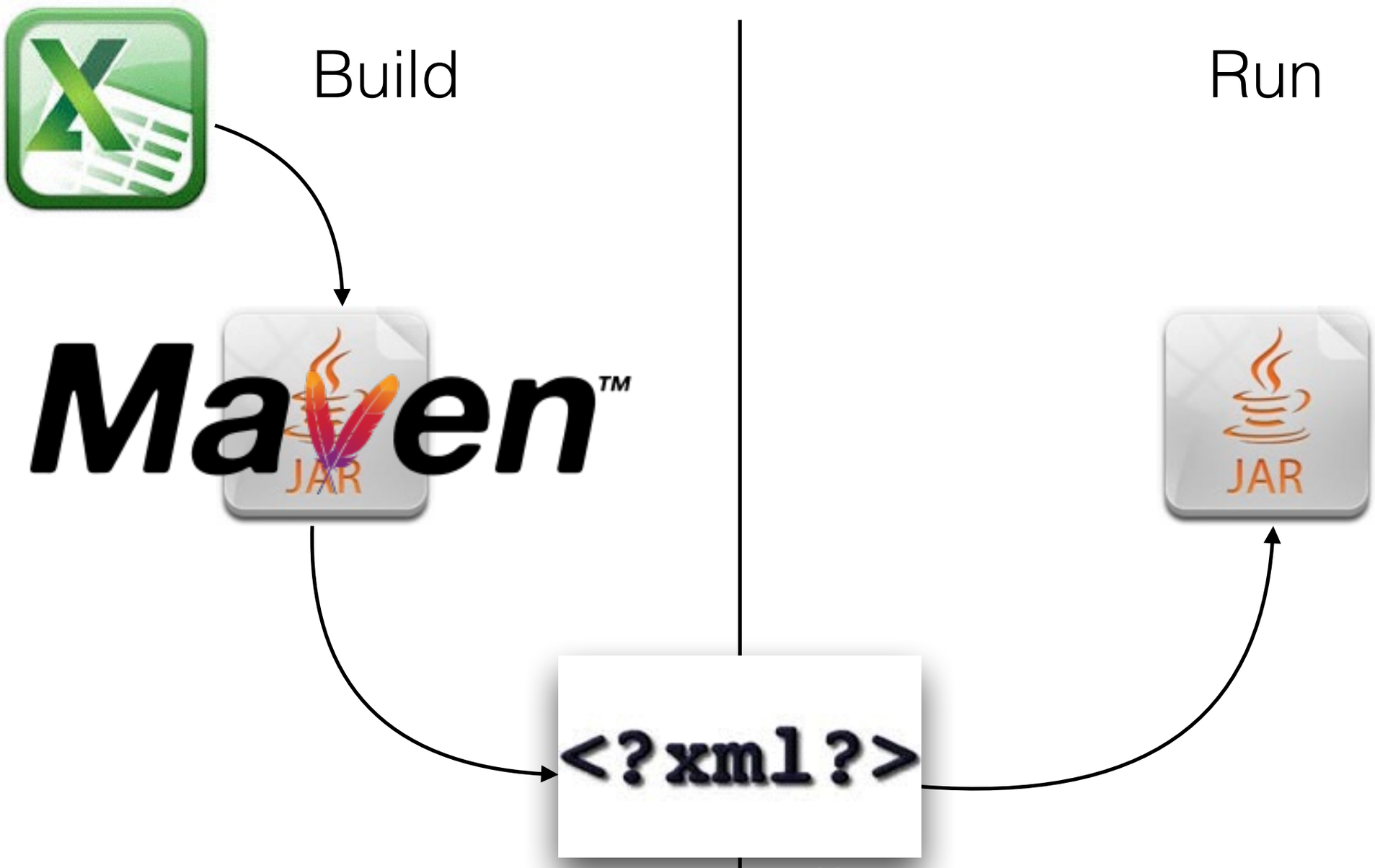
```java
public class RuntimeRuleLoader {
    private RuleEngine engine;

    public RuntimeRuleLoader(final RuleEngine engine) { this.engine = engine; }

    public void initialize() {
        XStream xStream = new XStream(new PureJavaReflectionProvider());
        final Collection<RuleType> ruleTypes = (Collection<RuleType>)xStream
                .fromXML(getClass().getResourceAsStream( name: "/RuleType.xml"));
        if (null != ruleTypes) {
            ruleTypes.stream().flatMap(new Function<RuleType, Stream<Rule>>() {
                @Override
                public Stream<Rule> apply(final RuleType rt) {
                    return ((Collection<Rule>)xStream
                            .fromXML(getClass().getResourceAsStream( name: "/" + rt.name() + ".xml")))
                            .stream();
                }
            }).forEach(r -> engine.add(r.getRuleType(), r));
        }
    }
}
```

Parse Excel

- Don't recommend reading directly from Excel at runtime in production.
- Parse Excel, transform into conditions, actions, rules, rule sets in memory.
- Serialize object graph to a suitable runtime storage format.
- Re-read serialized object graph for functional tests.
- Version controlled serialized object graph.

Maven Plugin

- Integrates as part of build step



Performance Considerations

- Faster to be brute force and repeat the same conditions.

- Focus on fast-fail short-circuiting.

- Build rule object graph so more expensive Predicates are tested later.

- Organize rules into rule-sets so first winner skips remaining rules in the set.

- Avoid multiple potential winners if possible.

Performance Cont.

- Parallelize Predicate test()
  - Defeats many short-circuit optimizations.
- Parallelize firing rules simultaneously on model objects.
- Parallelize firing rules of different rule sets on the same object simultaneously.
  - Defeats rule set precedences
- Possible to build an acyclic directed graph of rules connected by common Predicates.
  - …but now you're building an inference engine… don't do this!

References

http://www.martinfowler.com/bliki/RulesEngine.html

https://www.drools.org/

http://www.jessrules.com/

https://jcp.org/en/jsr/detail?id=94

http://www.sparklinglogic.com/category/business-rules/

https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html

https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

- https://github.com/steveswing