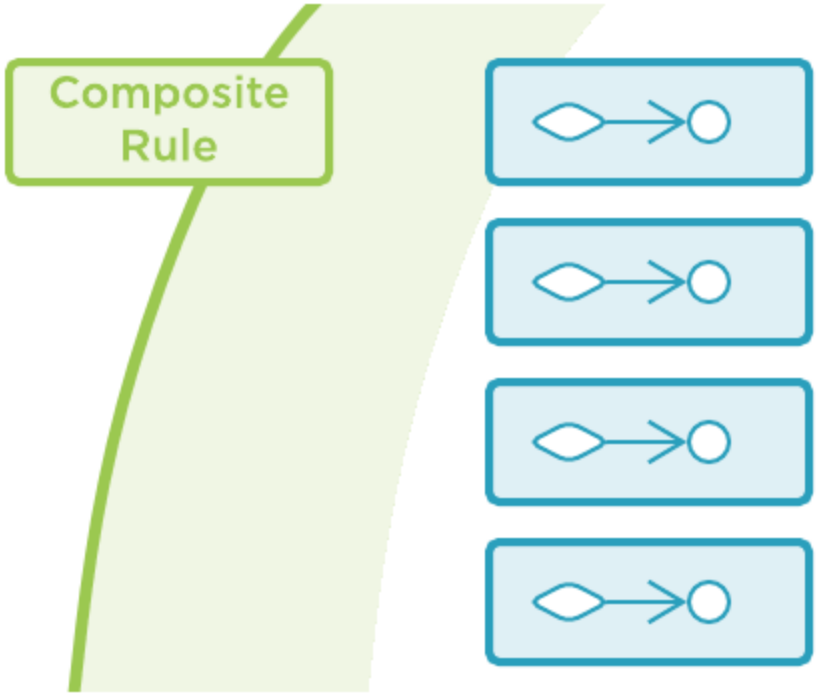
Turning Chained Branching into the Chain of Rule Objects

Zoran Horvat

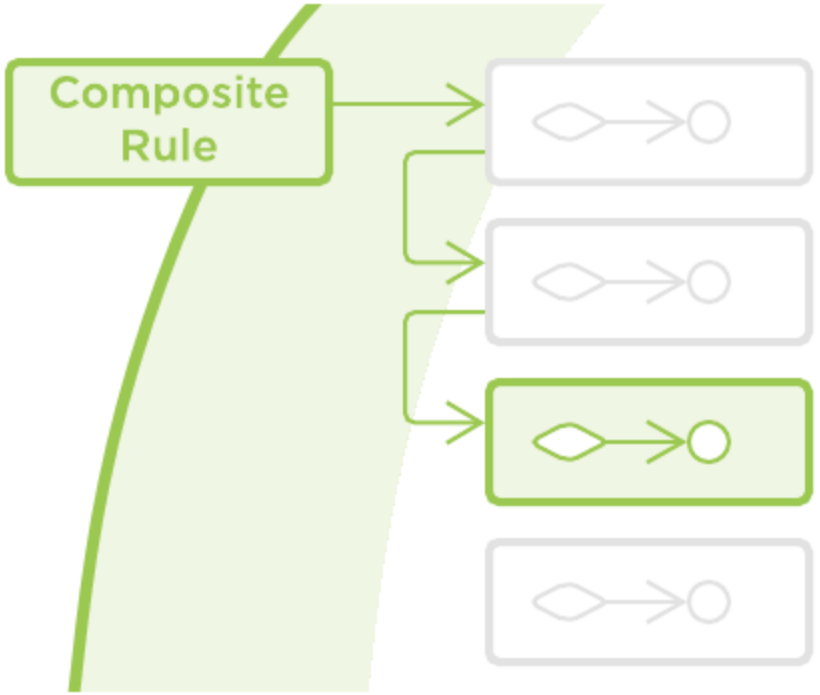
CEO AT CODING HELMET

@zoranh75 http://codinghelmet.com

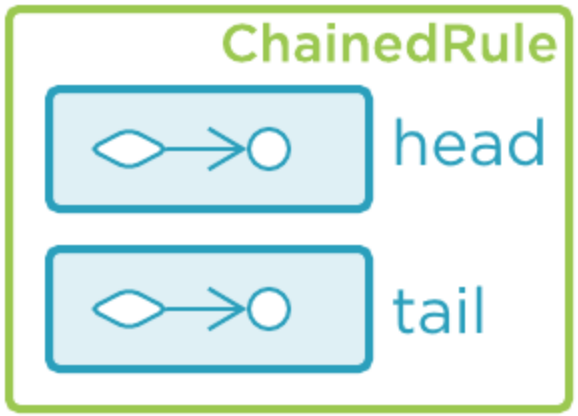
```
14
15      private void claimWarranty(Article article, DeviceStatus status, Optional<LocalDate> sensorFailureDate) {
16          LocalDate today = LocalDate.now();
17
18          StatusEqualityRule.match(
19                  DeviceStatus.allFine(),
20                  () -> this.claimMoneyBack(article, today))
21              .orElse(StatusEqualityRule.match(
22                  DeviceStatus.notOperational(),
23                  () -> {
24                      this.claimMoneyBack(article, today);
25                      this.claimExpress(article, today);
26                  }))
27              .applicableTo(status)
28              .ifPresent(action -> action.apply());
29
30          Runnable allFineAction = () ->
31              this.claimMoneyBack(article, today);
32
33          Runnable notOperationalAction = () -> {
34              this.claimMoneyBack(article, today);
35              this.claimExpress(article, today);
36          };
37
38          if (status.equals(DeviceStatus.allFine())) {
39              allFineAction.run();
40          } else if (status.equals(DeviceStatus.notOperational())) {
41              notOperationalAction.run();
```



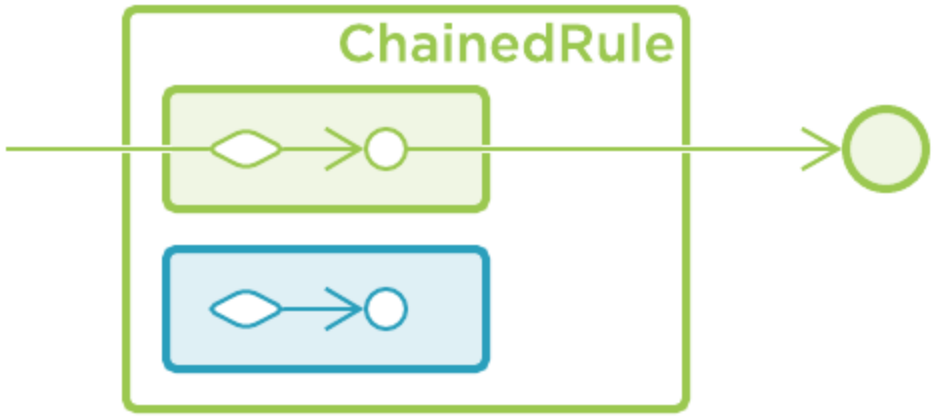
```
14
15      private void claimWarranty(Article article, DeviceStatus status, Optional<LocalDate> sensorFailureDate) {
16          LocalDate today = LocalDate.now();
17
18          StatusEqualityRule.match(
19                  DeviceStatus.allFine(),
20                  () -> this.claimMoneyBack(article, today))
21              .orElse(StatusEqualityRule.match(
22                  DeviceStatus.notOperational(),
23                  () -> {
24                      this.claimMoneyBack(article, today);
25                      this.claimExpress(article, today);
26                  }))
27              .applicableTo(status)
28              .ifPresent(action -> action.apply());
29
30          Runnable allFineAction = () ->
31              this.claimMoneyBack(article, today);
32
33          Runnable notOperationalAction = () -> {
34              this.claimMoneyBack(article, today);
35              this.claimExpress(article, today);
36          };
37
38          if (status.equals(DeviceStatus.allFine())) {
39              allFineAction.run();
40          } else if (status.equals(DeviceStatus.notOperational())) {
41              notOperationalAction.run();
```



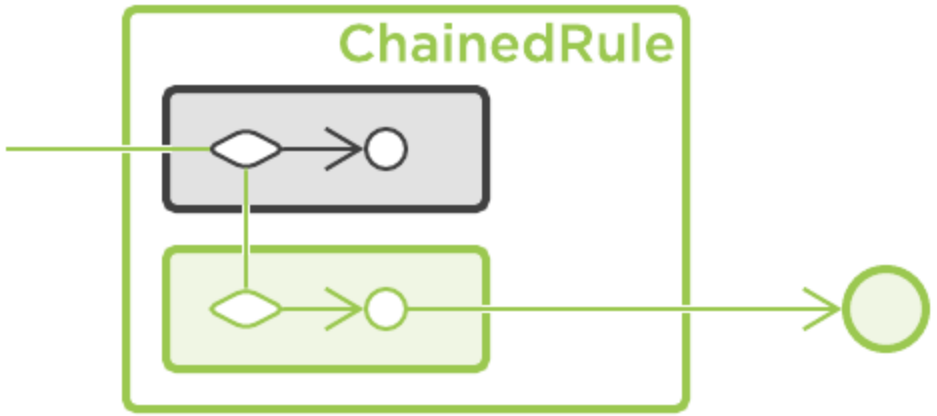
```java

public class ChainedRule implements ClaimingRule {
    private ClaimingRule head;
    private ClaimingRule tail;

    public ChainedRule(ClaimingRule head, ClaimingRule tail) {
        this.head = head;
        this.tail = tail;
    }

    @Override
    public Optional<Action> applicableTo(DeviceStatus status) {
        return this.head
            .applicableTo(status)
            .map(Optional::of)
            .orElse(tail.applicableTo(status));
    }
}
```



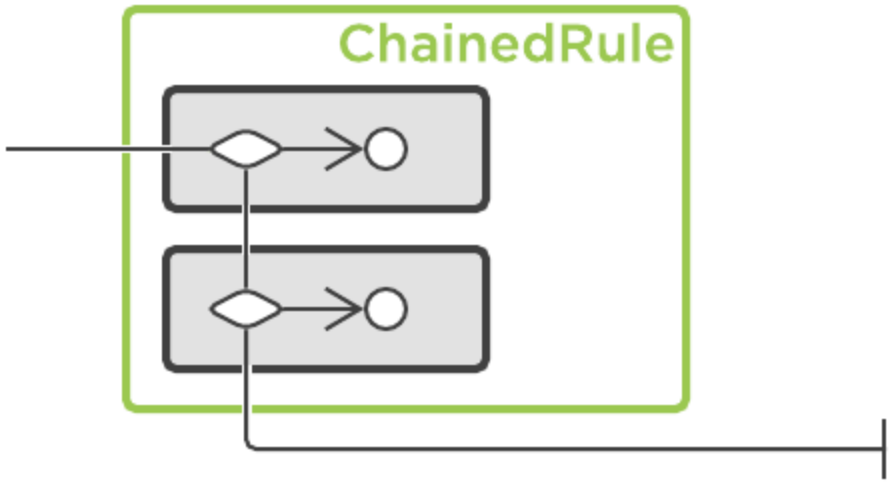
```java
public class ChainedRule implements ClaimingRule {
    private ClaimingRule head;
    private ClaimingRule tail;

    public ChainedRule(ClaimingRule head, ClaimingRule tail) {
        this.head = head;
        this.tail = tail;
    }

    @Override
    public Optional<Action> applicableTo(DeviceStatus status) {
        return this.head
            .applicableTo(status)
            .map(Optional::of)
            .orElse(tail.applicableTo(status));
    }
}
```



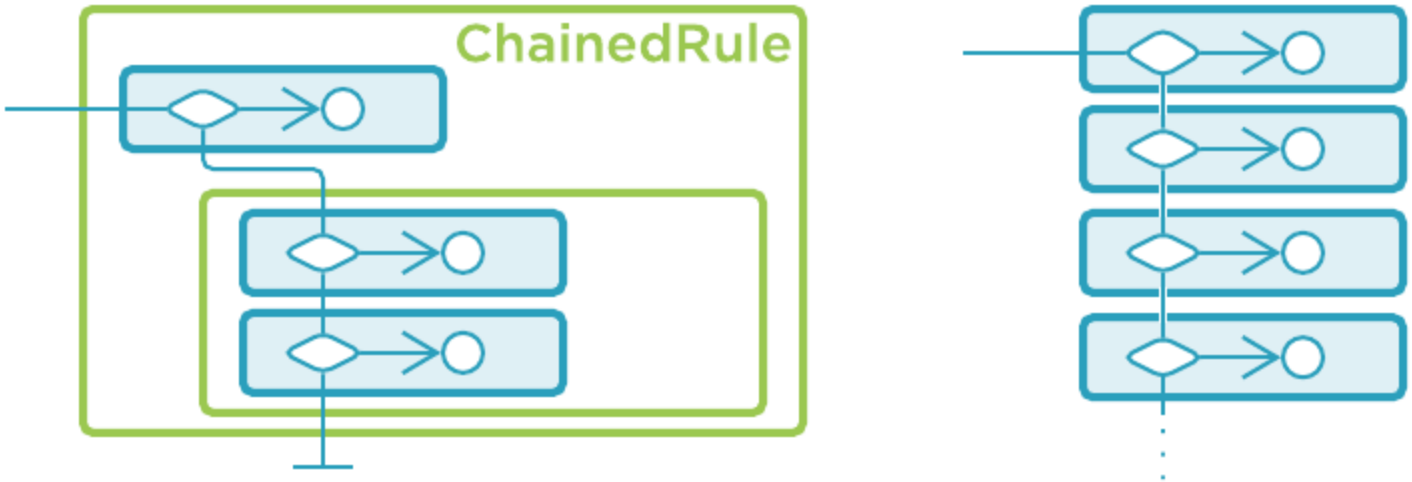
```java
public class ChainedRule implements ClaimingRule {
    private ClaimingRule head;
    private ClaimingRule tail;

    public ChainedRule(ClaimingRule head, ClaimingRule tail) {
        this.head = head;
        this.tail = tail;
    }

    @Override
    public Optional<Action> applicableTo(DeviceStatus status) {
        return this.head
            .applicableTo(status)
            .map(Optional::of)
            .orElse(tail.applicableTo(status));
    }
}
```



```java

public class ChainedRule implements ClaimingRule {
    private ClaimingRule head;
    private ClaimingRule tail;

    public ChainedRule(ClaimingRule head, ClaimingRule tail) {
        this.head = head;
        this.tail = tail;
    }

    @Override
    public Optional<Action> applicableTo(DeviceStatus status) {
        return this.head
            .applicableTo(status)
            .map(Optional::of)
            .orElse(tail.applicableTo(status));
    }
}
```

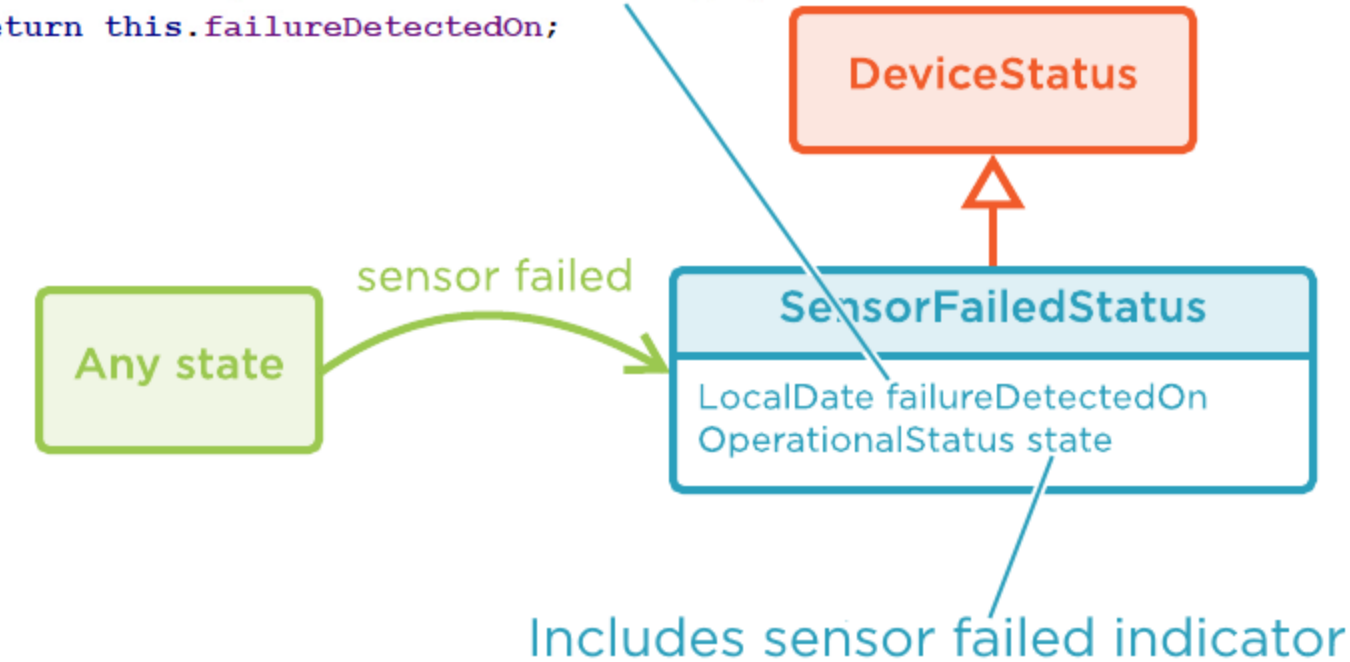


```
4
5   public class ChainedRule implements ClaimingRule {
6       private ClaimingRule head;
7       private ClaimingRule tail;
8
9       public ChainedRule(ClaimingRule head, ClaimingRule tail) {
10          this.head = head;
11          this.tail = tail;
12      }
13
14      @Override
15      public Optional<Action> applicableTo(DeviceStatus status) {
16          return this.head
17              .applicableTo(status)
18              .map(Optional::of)
19              .orElse(tail.applicableTo(status));
20      }
21  }
22
23
24
25
26
27
28
29
30
31
```



```
4
5   public class SensorFailedStatus extends DeviceStatus {
6       private LocalDate failureDetectedOn;
7
8   @   public SensorFailedStatus(LocalDate detectedOn, OperationalStatus inState) {
9           super(inState.andSensorFailed());
10          this.failureDetectedOn = detectedOn;
11      }
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
```

```
4
5   public class SensorFailedStatus extends DeviceStatus {
6       private LocalDate failureDetectedOn;
7
8   @   public SensorFailedStatus(LocalDate detectedOn, OperationalStatus inState) {
9           super(inState.andSensorFailed());
10          this.failureDetectedOn = detectedOn;
11      }
12
13      public LocalDate getFailureDetectionDate() {
14          return this.failureDetectedOn;
15      }
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
```



Pluralsight › src › com › codinghelmet › moreoojava › rules

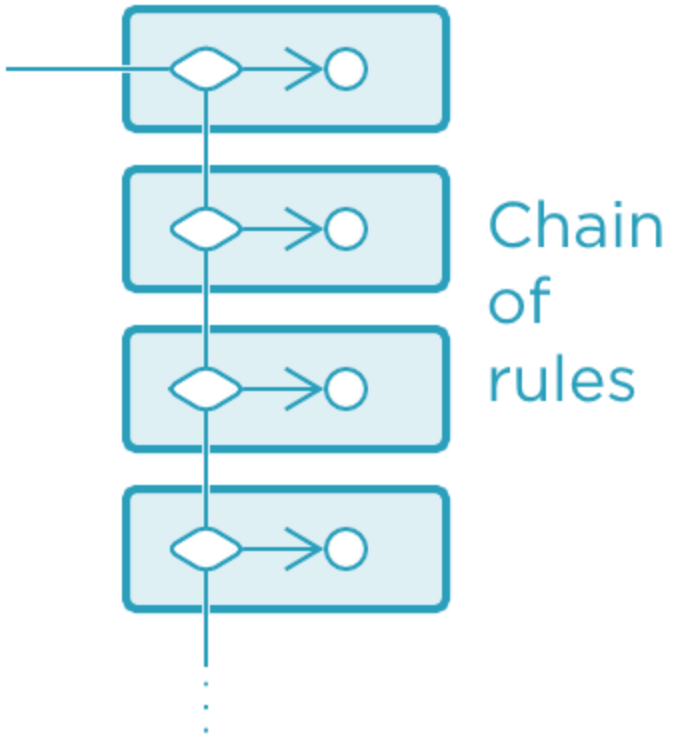
Main

Project

Pluralsight [Code] C:\Pluralsight
- .idea
- out
- src
  - com.codinghelmet.moreoojava
    - rules
      - AppendingCondition
      - ChainedRule
      - ExtendingCondition
      - OperationalCondition
      - ReducedRule
      - RootCondition
      - RuleFixture
      - State
      - StatusTypeCondition
    - states
    - warranties
    - Action
    - Article
    - ClaimingRule
    - Demo
    - Main
    - Part
    - Warranty
  - Code.iml
- External Libraries
- Scratches and Consoles

Demo.java | DeviceStatus.java | SensorFailedStatus.java | Article.java | ClaimingRule.java | Action.java

```
7
8   public interface ClaimingRule {
9       Optional<Action> applicableTo(DeviceStatus status);
10
11      default ClaimingRule orElse(ClaimingRule next) {
12          return new ChainedRule( head: this, next);
13      }
14  }
```

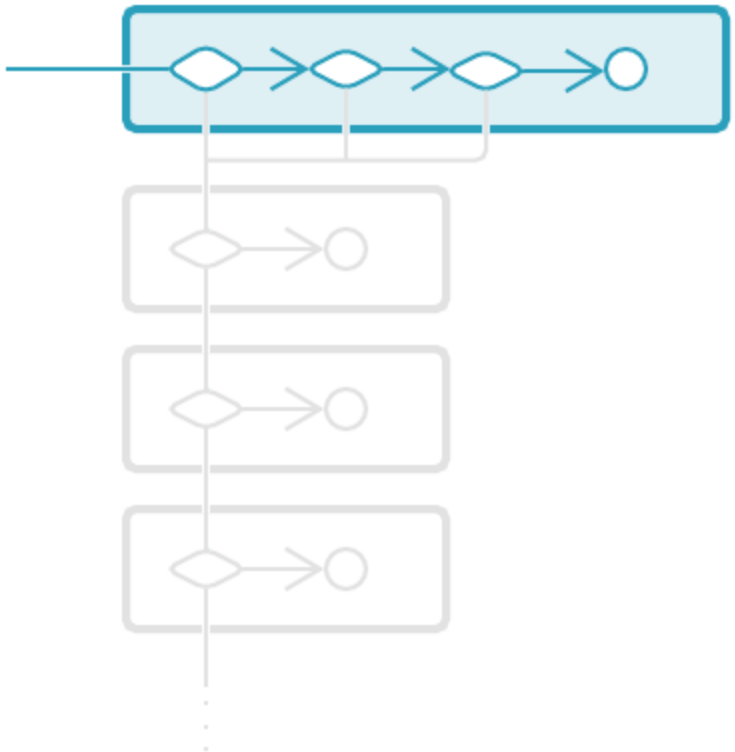


ClaimingRule

3: Find | 5: Debug | 6: TODO | Terminal | 0: Messages | 4: Run | Event Log

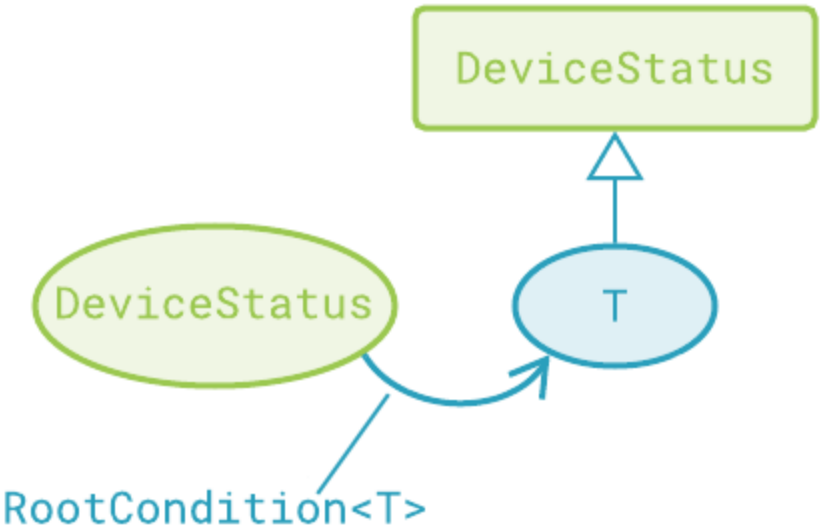
1: Project | 2: Favorites | 7: Structure | Ant Build | Maven

```
7
8   public interface ClaimingRule {
9       Optional<Action> applicableTo(DeviceStatus status);
10
11      default ClaimingRule orElse(ClaimingRule next) {
12          return new ChainedRule( head: this, next);
13      }
14  }
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
```



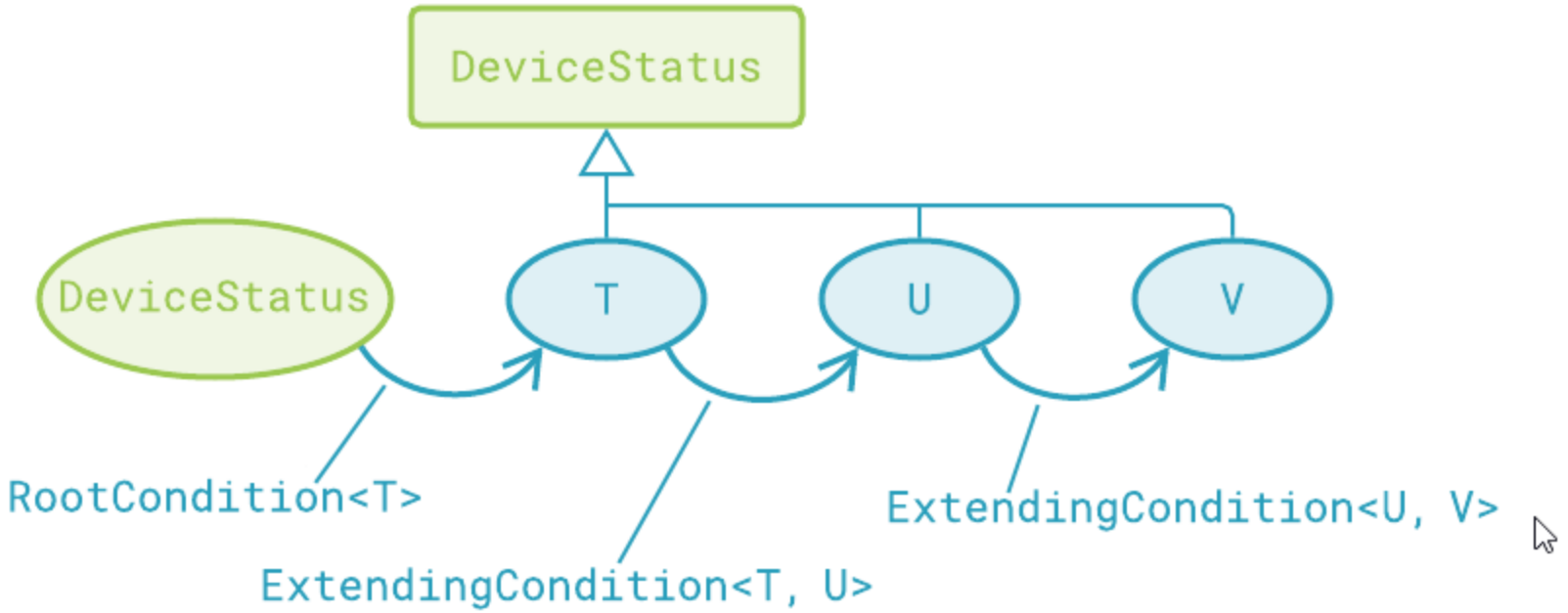
Chain of
conditions

```
8
9   public interface RootCondition<T extends DeviceStatus> {
10      Optional<T> applicableTo(DeviceStatus status);
11
12      default ClaimingRule applies(Consumer<T> action) {
13          return new RuleFixture<>( condition: this, action);
14      }
15  }
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

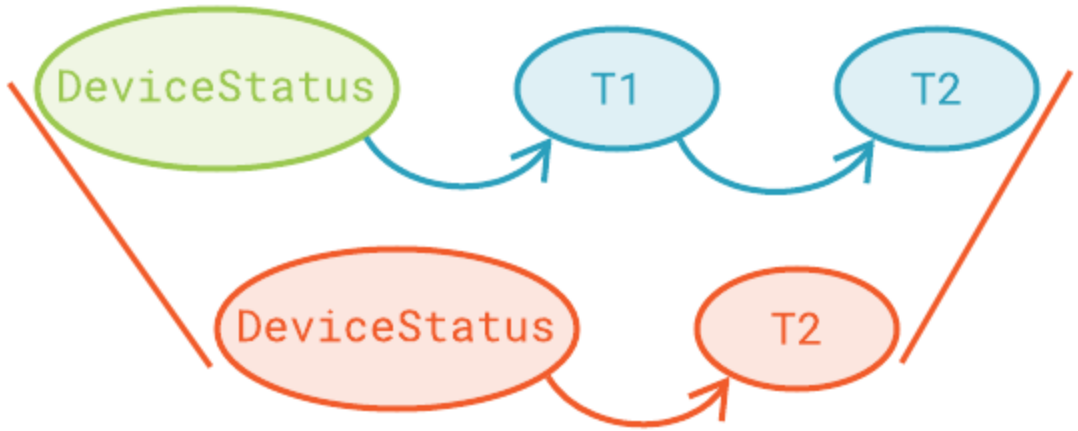


RootCondition

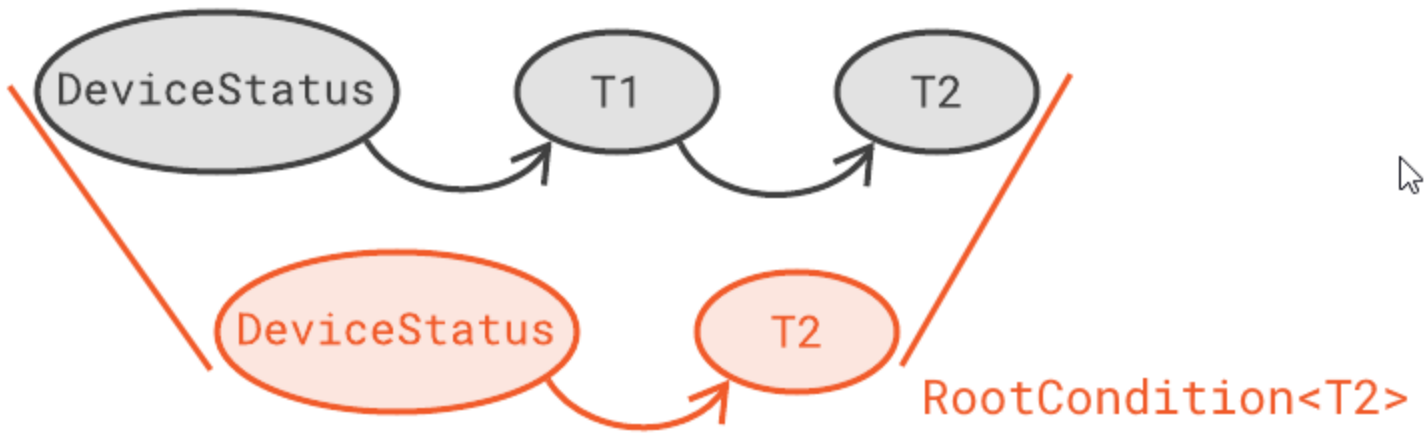
```
6
7   public interface ExtendingCondition<T1 extends DeviceStatus, T2 extends DeviceStatus> {
8       Optional<T2> applicableTo(T1 status);
9   }
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
```



```
6
7   public class AppendingCondition<T extends DeviceStatus, T1 extends DeviceStatus>
8           implements RootCondition<T1> {
9       private RootCondition<T> first;
10      private ExtendingCondition<T, T1> second;
11
12      public AppendingCondition(RootCondition<T> first,
13                                ExtendingCondition<T, T1> second) {
14          this.first = first;
15          this.second = second;
16      }
17
18      @Override
19      public Optional<T1> applicableTo(DeviceStatus status) {
20          return this.first.applicableTo(status)
21              .map(this.second::applicableTo)
22              .orElse(Optional.empty());
23      }
24  }
25
26
27
28
29
30
31
32
33
```



```
6
7   public class AppendingCondition<T extends DeviceStatus, T1 extends DeviceStatus>
8           implements RootCondition<T1> {
9       private RootCondition<T> first;
10      private ExtendingCondition<T, T1> second;
11
12      public AppendingCondition(RootCondition<T> first,
13                                ExtendingCondition<T, T1> second) {
14          this.first = first;
15          this.second = second;
16      }
17
18      @Override
19      public Optional<T1> applicableTo(DeviceStatus status) {
20          return this.first.applicableTo(status)
21              .map(this.second::applicableTo)
22              .orElse(Optional.empty());
23      }
24  }
25
26
27
28
29
30
31
32
33
```



- Pluralsight [Code] C:\Pluralsight
  - .idea
  - out
  - src
    - com.codinghelmet.moreoojava
      - rules
        - AppendingCondition
        - ChainedRule
        - ExtendingCondition
        - OperationalCondition
        - ReducedRule
        - RootCondition
        - RuleFixture
        - State
        - StatusTypeCondition
      - states
      - warranties
      - Action
      - Article
      - ClaimingRule
      - Demo
      - Main
      - Part
      - Warranty
    - Code.iml
  - External Libraries
  - Scratches and Consoles

```
8
9   public interface RootCondition<T extends DeviceStatus> {
10      Optional<T> applicableTo(DeviceStatus status);
11
12      default ClaimingRule applies(Consumer<T> action) {
13          return new RuleFixture<>( condition: this, action);
14      }
15  }
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

RootCondition

Pluralsight › src › com › codinghelmet › moreoojava › rules › RuleFixture

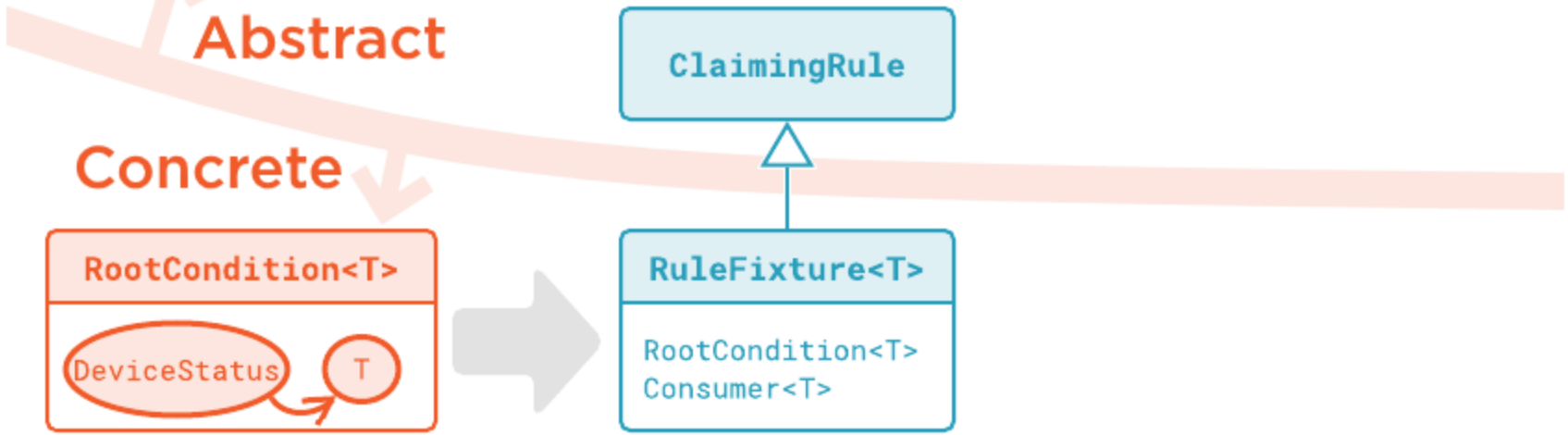
Main

Project

Pluralsight [Code] C:\Pluralsight
- .idea
- out
- src
  - com.codinghelmet.moreoojava
    - rules
      - AppendingCondition
      - ChainedRule
      - ExtendingCondition
      - OperationalCondition
      - ReducedRule
      - RootCondition
      - RuleFixture
      - State
      - StatusTypeCondition
    - states
    - warranties
    - Action
    - Article
    - ClaimingRule
    - Demo
    - Main
    - Part
    - Warranty
  - Code.iml
- External Libraries
- Scratches and Consoles

RootCondition.java | RuleFixture.java | ExtendingCondition.java | OperationalCondition.java | StatusTypeCondition.java

```
9
10  public class RuleFixture<T extends DeviceStatus> implements ClaimingRule {
11      private RootCondition<T> condition;
12      private Consumer<T> action;
13
14      public RuleFixture(RootCondition<T> condition, Consumer<T> action) {
15          this.condition = condition;
16          this.action = action;
17      }
18
19      @Override
20      public Optional<Action> applicableTo(DeviceStatus status) {
21          return this.condition
22              .applicableTo(status)
23              .map(s -> new ReducedRule<>(s, this.action));
24      }
25  }
26
27
28
29
30
31
32
33
34
35
36
```



RuleFixture

3: Find | 5: Debug | 6: TODO | Terminal | 0: Messages | 4: Run | Event Log

1: Project | 7: Structure | 2: Favorites | Ant Build | Maven

Pluralsight › src › com › codinghelmet › moreoojava › rules › ReducedRule

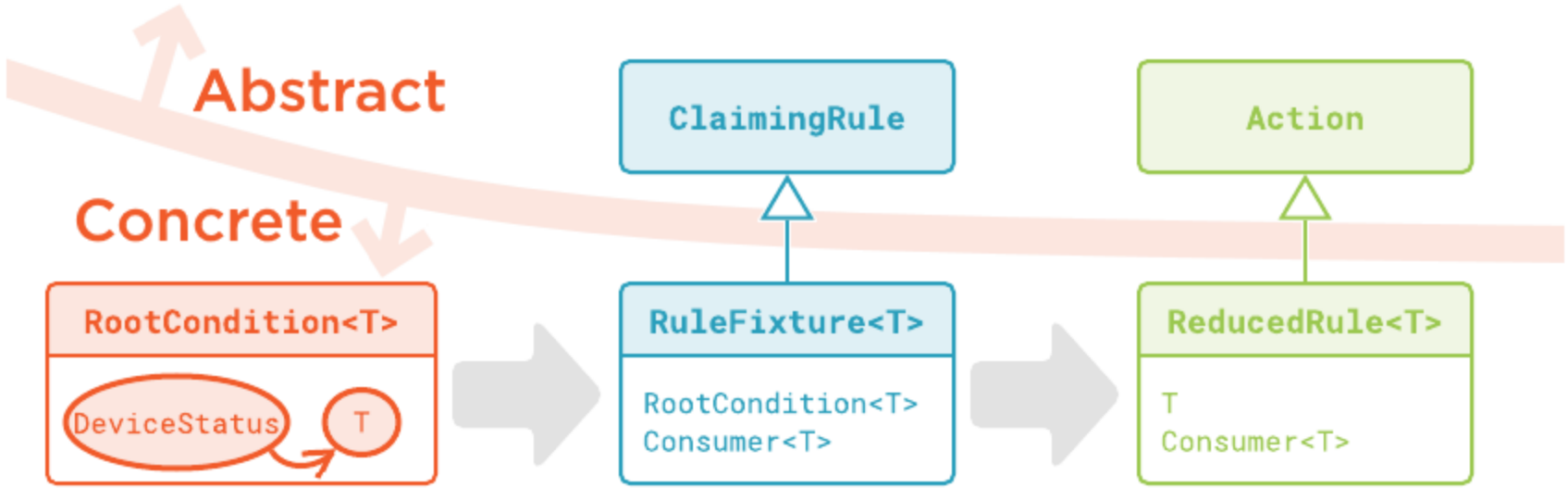
Main

Project

Pluralsight [Code] C:\Pluralsight
- .idea
- out
- src
  - com.codinghelmet.moreoojava
    - rules
      - AppendingCondition
      - ChainedRule
      - ExtendingCondition
      - OperationalCondition
      - ReducedRule
      - RootCondition
      - RuleFixture
      - State
      - StatusTypeCondition
    - states
    - warranties
    - Action
    - Article
    - ClaimingRule
    - Demo
    - Main
    - Part
    - Warranty
  - Code.iml
- External Libraries
- Scratches and Consoles

Condition.java | RuleFixture.java | ReducedRule.java | ExtendingCondition.java | OperationalCondition.java | StatusTypeCondition.java

```
7
8   public class ReducedRule<T extends DeviceStatus> implements Action {
9       private T status;
10      private Consumer<T> action;
11
12      public ReducedRule(T status, Consumer<T> action) {
13          this.status = status;
14          this.action = action;
15      }
16
17      @Override
18      public void apply() {
19          this.action.accept(this.status);
20      }
21  }
22
23
24
25
26
27
28
29
30
31
32
33
34
```



ReducedRule

1: Project | 7: Structure | 2: Favorites | Ant Build | Maven

3: Find | 5: Debug | 6: TODO | Terminal | 0: Messages | 4: Run | Event Log

```
45
46      private ClaimingRule allFineRule() {
47          return State.matching(
48              OperationalStatus.allFine()).applies(
49              this::moneyBack);
50      }
51
52      private ClaimingRule notOperationalRule() {
53          return State.matching(
54              OperationalStatus.notOperational()).applies(
55              s -> {
56                  this.moneyBack(s);
57                  this.express(s);
58              });
59      }
60
61      private ClaimingRule visiblyDamagedRule() {
62          return State.matching(
63              OperationalStatus.visiblyDamaged()).applies(
64              this::doNothing);
65      }
66
67      private ClaimingRule sensorFailedRule() {
68          return State.matching(
69              OperationalStatus.sensorFailed(),
70              SensorFailedStatus.class).applies(
71              s -> {
72                  this.moneyBack(s);
```

When state

```
45
46      private ClaimingRule allFineRule() {
47          return State.matching(
48              OperationalStatus.allFine()).applies(
49              this::moneyBack);
50      }
51
52      private ClaimingRule notOperationalRule() {
53          return State.matching(
54              OperationalStatus.notOperational()).applies(
55              s -> {
56                  this.moneyBack(s);
57                  this.express(s);
58              });
59      }
60
61      private ClaimingRule visiblyDamagedRule() {
62          return State.matching(
63              OperationalStatus.visiblyDamaged()).applies(
64              this::doNothing);
65      }
66
67      private ClaimingRule sensorFailedRule() {
68          return State.matching(
69              OperationalStatus.sensorFailed(),
70              SensorFailedStatus.class).applies(
71              s -> {
72                  this.moneyBack(s);
```

When state is matching

```
45
46      private ClaimingRule allFineRule() {
47          return State.matching(
48              OperationalStatus.allFine()).applies(
49              this::moneyBack);
50      }
51
52      private ClaimingRule notOperationalRule() {
53          return State.matching(
54              OperationalStatus.notOperational()).applies(
55              s -> {
56                  this.moneyBack(s);
57                  this.express(s);
58              });
59      }
60
61      private ClaimingRule visiblyDamagedRule() {
62          return State.matching(
63              OperationalStatus.visiblyDamaged()).applies(
64              this::doNothing);
65      }
66
67      private ClaimingRule sensorFailedRule() {
68          return State.matching(
69              OperationalStatus.sensorFailed(),
70              SensorFailedStatus.class).applies(
71              s -> {
72                  this.moneyBack(s);
```

When state is matching operational status

```
45
46      private ClaimingRule allFineRule() {
47          return State.matching(
48              OperationalStatus.allFine()).applies(
49              this::moneyBack);
50      }
51
52      private ClaimingRule notOperationalRule() {
53          return State.matching(
54              OperationalStatus.notOperational()).applies(
55              s -> {
56                  this.moneyBack(s);
57                  this.express(s);
58              });
59      }
60
61      private ClaimingRule visiblyDamagedRule() {
62          return State.matching(
63              OperationalStatus.visiblyDamaged()).applies(
64              this::doNothing);
65      }
66
67      private ClaimingRule sensorFailedRule() {
68          return State.matching(
69              OperationalStatus.sensorFailed(),
70              SensorFailedStatus.class).applies(
71              s -> {
72                  this.moneyBack(s);
```

When state is matching operational status "not operational"

```
45
46      private ClaimingRule allFineRule() {
47          return State.matching(
48              OperationalStatus.allFine()).applies(
49              this::moneyBack);
50      }
51
52      private ClaimingRule notOperationalRule() {
53          return State.matching(
54              OperationalStatus.notOperational()).applies(
55              s -> {
56                  this.moneyBack(s);
57                  this.express(s);
58              });
59      }
60
61      private ClaimingRule visiblyDamagedRule() {
62          return State.matching(
63              OperationalStatus.visiblyDamaged()).applies(
64              this::doNothing);
65      }
66
67      private ClaimingRule sensorFailedRule() {
68          return State.matching(
69              OperationalStatus.sensorFailed(),
70              SensorFailedStatus.class).applies(
71              s -> {
72                  this.moneyBack(s);
```

When state is matching operational status "not operational" it applies

```
45
46      private ClaimingRule allFineRule() {
47          return State.matching(
48              OperationalStatus.allFine()).applies(
49              this::moneyBack);
50      }
51
52      private ClaimingRule notOperationalRule() {
53          return State.matching(
54              OperationalStatus.notOperational()).applies(
55              s -> {
56                  this.moneyBack(s);
57                  this.express(s);
58              });
59      }
60
61      private ClaimingRule visiblyDamagedRule() {
62          return State.matching(
63              OperationalStatus.visiblyDamaged()).applies(
64              this::doNothing);
65      }
66
67      private ClaimingRule sensorFailedRule() {
68          return State.matching(
69              OperationalStatus.sensorFailed(),
70              SensorFailedStatus.class).applies(
71              s -> {
72                  this.moneyBack(s);
```

When state is matching operational status "not operational" it applies money back

```
45
46      private ClaimingRule allFineRule() {
47          return State.matching(
48              OperationalStatus.allFine()).applies(
49              this::moneyBack);
50      }
51
52      private ClaimingRule notOperationalRule() {
53          return State.matching(
54              OperationalStatus.notOperational()).applies(
55              s -> {
56                  this.moneyBack(s);
57                  this.express(s);
58              });
59      }
60
61      private ClaimingRule visiblyDamagedRule() {
62          return State.matching(
63              OperationalStatus.visiblyDamaged()).applies(
64              this::doNothing);
65      }
66
67      private ClaimingRule sensorFailedRule() {
68          return State.matching(
69              OperationalStatus.sensorFailed(),
70              SensorFailedStatus.class).applies(
71              s -> {
72                  this.moneyBack(s);
```

When state is matching operational status "not operational" it applies money back and express claim

```
45
46      private ClaimingRule allFineRule() {
47          return State.matching(
48              OperationalStatus.allFine()).applies(
49              this::moneyBack);
50      }
51
52      private ClaimingRule notOperationalRule() {
53          return State.matching(
54              OperationalStatus.notOperational()).applies(
55              s -> {
56                  this.moneyBack(s);
57                  this.express(s);
58              });
59      }
60
61      private ClaimingRule visiblyDamagedRule() {
62          return State.matching(
63              OperationalStatus.visiblyDamaged()).applies(
64              this::doNothing);
65      }
66
67      private ClaimingRule sensorFailedRule() {
68          return State.matching(
69              OperationalStatus.sensorFailed(),
70              SensorFailedStatus.class).applies(
71              s -> {
72                  this.moneyBack(s);
```

When state is matching operational status "not operational" it applies money back and express claim

Summary

Why decomposing a chain of `if-else`s?

- Control flow is rigid
- Doesn’t support changing requests
- Implemented at the consuming end

Summary

Introducing rules instead of `if-elses`

- Boolean condition + action are a rule
- A rule is either applicable or not
- Multiple rules form a chain
- Control is passed down the chain
- Rules can also be independent
- Rule can test multiple conditions
- Complex rules can be constructed

Summary

Surviving the complexity of rules

- Use rule builders to contain complexity
- Concrete builders implement concrete views on business
- Rule builders can be substituted

Summary

Managing a complex business domain

- State pattern to model discrete state
- Polymorphic calls to substitute Boolean tests
- Filtering interface and optional objects to model object existence
- Composite pattern to manage multitudes of objects
- Strategy pattern to vary behavior
- Rules pattern to model business rules
- Domain-specific Language to make the code composable

Summary

Next module:
Refactoring to Bind it All Together