Verifying Configuration Files Alongside Component Code

Ciera Jaspan

Doctoral Candidate in Software Engineering

Carnegie Mellon

isr institute for SOFTWARE RESEARCH

Configuration files

- Used in many software systems
  - Allows modification at runtime
  - Eases extension of frameworks
- Common format is XML
- No verification specific to the system
  - Can verify schema, but not semantics
- FUSION verifies systems that use XML-based config files
  - Considers how the config file works with the component code
  - Basic principles can be used for other config files as well

Creating a Spring web application

- Uses an MVC architecture
  - Create a model class

```
public class ProductCommand implements Serializable {
  public String getName() {...}
  public void setName(String name) {...}
  public String getUPC() {...}
  public void setUPC(String upc) {...}
}
```

Creating a Spring web application

- Uses an MVC architecture
  - Create a model class
  - Extend from a controller class…

```
public class EditProductController extends SimpleFormController {
  private ProductManager pManager;
  protected ModelAndView onSubmit(Object command) throws Exception {
    ProductCommand product = (ProductCommand)command;
    pManager.setProduct(product);
    return new ModelAndView(new RedirectView(getSuccessView()));
  }
  protected Object formBackingObject(HttpServletRequest request) {
    int id = ServletRequestUtils.getIntParameter(request, "id", -1);
    return (id != -1) ? pManager.getProduct(id) : new ProductCommand();
  }
}
```

Creating a Spring web application

- Uses an MVC architecture
  - Create a model class
  - Extend from a controller class and helper classes

```
public class ProductValidator implements Validator {
  public boolean supports(Class clazz) {
    return clazz.equals(ProductCommand.class);
  }
  public void validate(Object command, Errors errors) {
    ProductCommand question = (ProductCommand)command;
    ...
  }
}
```

Creating a Spring web application

- Uses an MVC architecture
  - Create a model class
  - Extend from a controller class and helper classes
  - Provide a JSP for the view

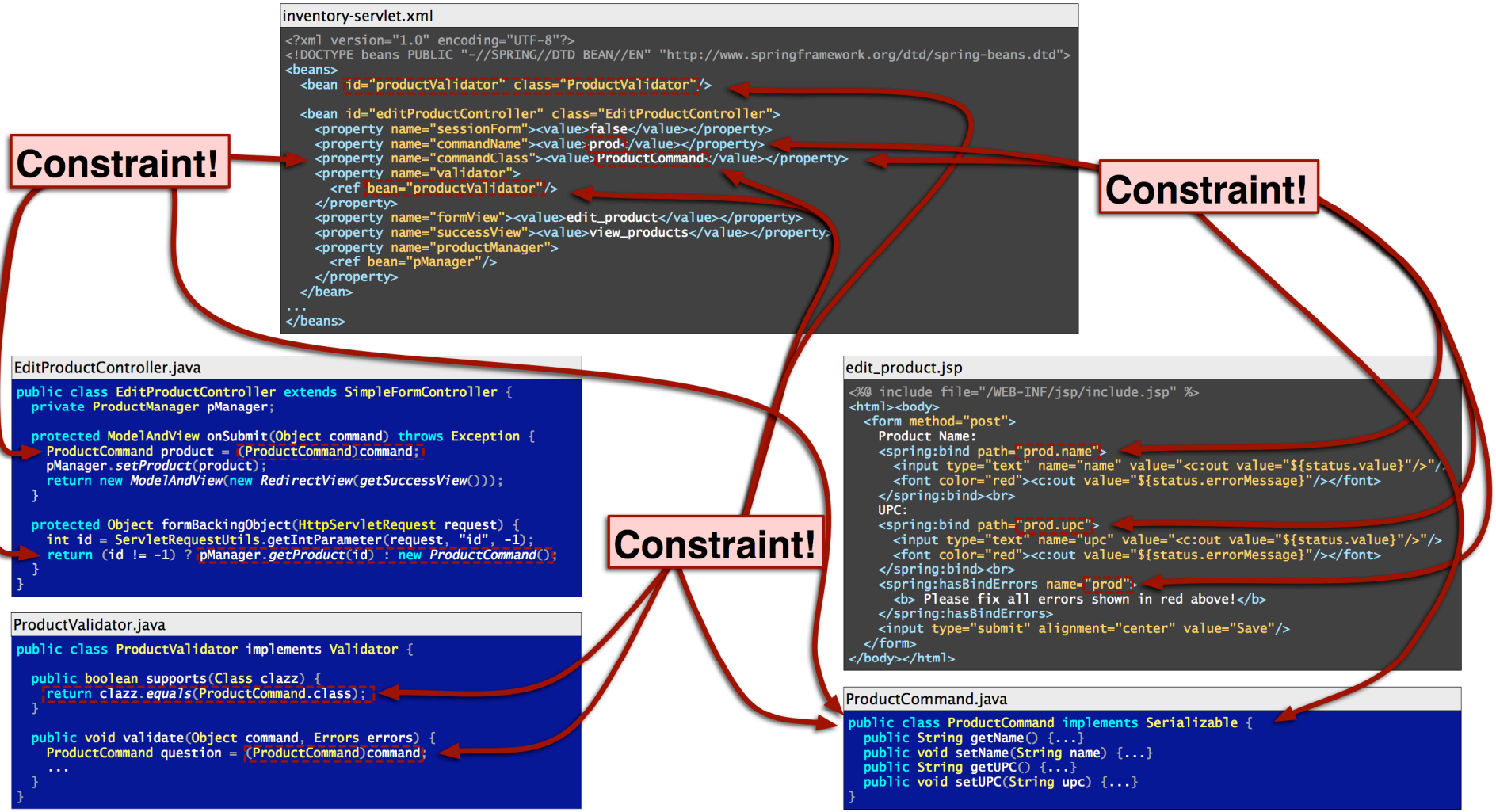
```
<%@ include file="/WEB-INF/jsp/include.jsp" %>
<html><body><form method="post">
  Product Name:
  <spring:bind path="prod.name">
    <input type="text" name="name" value="<c:out value="${status.value}"/>"/>
    <font color="red"><c:out value="${status.errorMessage}"/></font>
  </spring:bind><br>
 UPC:
 <spring:bind path="prod.upc">
   <input type="text" name="upc" value="<c:out value="${status.value}"/>"/>
   <font color="red"><c:out value="${status.errorMessage}"/></font>
 </spring:bind><br>
 <spring:hasBindErrors name="prod">
   <b> Please fix all errors shown in red above!</b>
 </spring:hasBindErrors>
 <input type="submit" alignment="center" value="Save"/>
 </form></body></html>
```

Creating a Spring web application

- Uses an MVC architecture
  - Create a model class
  - Extend from a controller class and helper classes
  - Provide a JSP for the view
- And then hook them all up with XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="productValidator" class="ProductValidator"/>
  <bean id="editProductController" class="EditProductController">
    <property name="sessionForm"><value>false</value></property>
    <property name="commandName"><value>prod</value></property>
    <property name="commandClass"><value>ProductCommand</value></property>
    <property name="validator"><ref bean="productValidator"/></property>
    <property name="formView"><value>edit_product</value></property>
    <property name="successView"><value>view_products</value></property>
    <property name="productManager"><ref bean="pManager"/></property>
  </bean>
</beans>
```

Classes don't exist by themselves



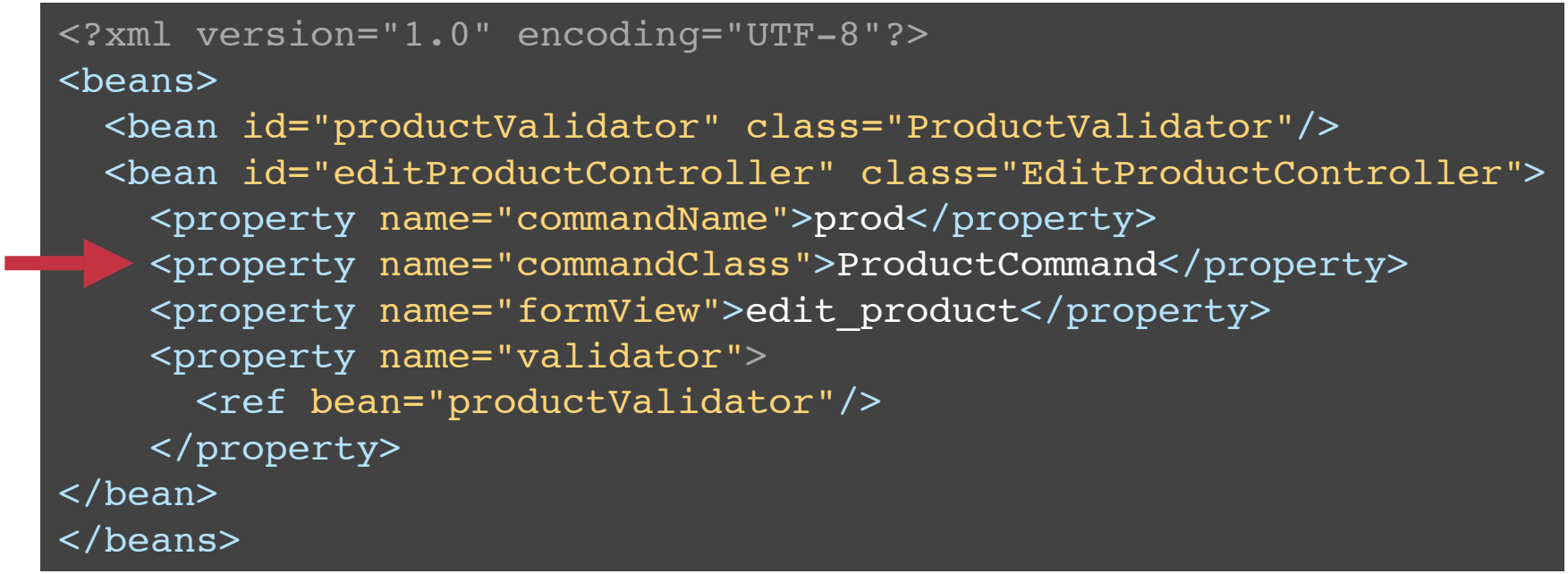
In the XML config file…

- XML declares a controller

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="productValidator" class="ProductValidator"/>
  <bean id="editProductController" class="EditProductController">
    <property name="commandName">prod</property>
    <property name="commandClass">ProductCommand</property>
    <property name="formView">edit_product</property>
    <property name="validator">
      <ref bean="productValidator"/>
    </property>
</bean>
</beans>
```

In the XML config file…

- XML declares a controller
- The controller has a command type



```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="productValidator" class="ProductValidator"/>
  <bean id="editProductController" class="EditProductController">
    <property name="commandName">prod</property>
    <property name="commandClass">ProductCommand</property>
    <property name="formView">edit_product</property>
    <property name="validator">
      <ref bean="productValidator"/>
    </property>
</bean>
</beans>
```

In the XML config file…

- XML declares a controller
- The controller has a command type
- The controller also has a validator

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="productValidator" class="ProductValidator"/>
  <bean id="editProductController" class="EditProductController">
    <property name="commandName">prod</property>
    <property name="commandClass">ProductCommand</property>
    <property name="formView">edit_product</property>
    <property name="validator">
      <ref bean="productValidator"/>
    </property>
</bean>
</beans>
```

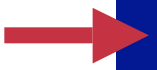
In the XML config file…

- XML declares a controller
- The controller has a command type
- The controller also has a validator
- All three classes must exist

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="productValidator" class="ProductValidator"/>
  <bean id="editProductController" class="EditProductController">
    <property name="commandName">prod</property>
    <property name="commandClass">ProductCommand</property>
    <property name="formView">edit_product</property>
    <property name="validator">
      <ref bean="productValidator"/>
    </property>
</bean>
</beans>
```

In the validator…

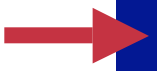
- The validator must return true when asked if it supports the command type



```
public class ProductValidator implements Validator {
  public boolean supports(Class clazz) {
    return clazz.equals(ProductCommand.class);
  }
  public void validate(Object command, Errors errors) {
    ProductCommand question = (ProductCommand)command;
    ...
  }
}
```

In the validator…

- The validator must return true when asked if it supports the command type
- The validator should cast the command parameter to the appropriate type



```
public class ProductValidator implements Validator {
  public boolean supports(Class clazz) {
    return clazz.equals(ProductCommand.class);
  }
  public void validate(Object command, Errors errors) {
    ProductCommand question = (ProductCommand)command;
    ...
  }
}
```

Collaboration constraints

- State-based restriction on how multiple objects may interact
- Cross boundary between configuration and component code
- FUSION
  - A specification language to describe collaboration constraints across language boundaries
  - A static analysis to check plugin code against the specs

Associating several objects

- Need a way to associate validator, controller, and command type
- Use relationships to do this abstract association

```
FormValidator(Controller, Validator, Class)
```

- Can describe constraints as predicate logic over relationships

Constraints as annotations

```
@Constraint(
  op = "EOM: Validator.supports(Class class)",


)
```

Operation

- The operation we are constraining

Constraints as annotations

```
@Constraint(
  op = "EOM: Validator.supports(Class class)",
  trg = "FormValidator(controller, target, class)",

)
```

Operation

- The operation we are constraining

Trigger predicate

- Predicate logic over relationships
- When to constrain the operation

Constraints as annotations

```
@Constraint(
  op = "EOM: Validator.supports(Class class)",
  trg = "FormValidator(controller, target, class)",
  req = "result",
)
```

Operation

- The operation we are constraining

Trigger predicate

- Predicate logic over relationships
- When to constrain the operation

Requires predicate

- Pre-condition
- Must be true if the constraint is triggered

Constraints as annotations

```
@Constraint(
  op = "EOM: Validator.supports(Class class)",
  trg = "FormValidator(controller, target, class)",
  req = "result",
  eff = {})
```

Operation

- The operation we are constraining

Requires predicate

- Pre-condition
- Must be true if the constraint is triggered

Trigger predicate

- Predicate logic over relationships
- When to constrain the operation

Effect list

- Post-condition
- Will be applied if the constraint is triggered

Constraints as annotations

```
@Constraint(
  op = "EOM: Validator.supports(Class class)",
  trg = "FormValidator(controller, target, class)",
  req = "result",
  eff = {})
```

```
public boolean supports(Class clazz) {
  return clazz.equals(Foo.class);
}
```

Fails!
trg *matches,*
but ***req*** *is false!*

```
public boolean supports(Class clazz) {
  return clazz.equals(ProductCommand.class);
}
```

Passes! ***trg*** *and*
req *is true!*

Config file provides the relationships

- Relationships come from the XML config file
- Extract FormValidator relationship

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="prodVal" class="ProductValidator"/>
  <bean id="editProdCtrl" class="EditProductController">
    <property name="commandName">prod</property>
    <property name="commandClass">ProductCommand</property>
    <property name="formView">edit_product</property>
    <property name="validator">
      <ref bean="productValidator"/>
    </property>
</bean>
</beans>
```

FormValidator(prodVal, editProdCtrl, ProductCommand.class)

Extracted using XQuery

- Extracts the objects and the relationships
- Used as the starting lattice for the static analysis

```
for $bean in /beans/bean
for $val in /beans/bean
where isSubtype($bean/@class,
           "org.springframework.web.servlet.mvc.SimpleFormController")
  and isSubtype($val/@class,
           "org.springframework.validation.Validator")
  and $bean/property[@name='validator']/ref/@bean = $val/@name
return
  <Relationship name="FormValidator">
    <Object name="{$bean/@name}" type="{$bean/@class}"/>
    <Object name="{$val/@name}" type="{$val/@name}"/>
    <Object name="{$bean/property[@name='commandClass']/@value}"
            type="java.lang.Class"/>
  </Relationship>
```

Related Work

- Relationships
  - Associations between types [UML]
  - As a programming language construct [Bierman and Wren 2004; Balzer, Gross, Eugster 2007]
- Verifying config files
  - Framework-specific checkers [SpringIDE, EclipsePDE]
  - XML Types [Hosoya, Vouillon, Pierce 2005]
  - Static checking for Ruby on Rails [An, Chaudhuri, Foster 2009]
  - Checking dynamic web applications [Sunshine and Aldrich 2010]
- Core semantics in [Jaspan and Aldrich 2009]

Verifying config files with component code

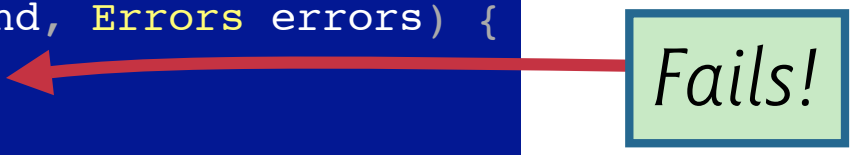
- Relationships to verify consistency of config files with components
- Config file used as a repository of relationships
- Extracted relationships used in FUSION specifications and analysis

Questions?

Constraints as annotations

```
@Constraint(
  op = "BOM: Validator.validate(Object cmd, Errors errs)",
  trg = "FormValidator(controller, target, class)",
  req = "TRUE",
  eff = {"cmd instanceof class"})
@Constraint(
  op = "(class) cmd",
  trg = "TRUE",
  req = "cmd instanceof class",
  eff = {})
```

```
public void validate(Object command, Errors errors) {
  Foo question = (Foo)command;
  ...
}
```




```
public void validate(Object command, Errors errors) {
  ProductCommand question = (ProductCommand)command;
  ...
}
```