Supporting production strategies as refinements of the production process

Oscar Díaz, Salvador Trujillo, Felipe I. Anfurrutia

ONEKIN Group. University of the Basque Country
PO Box: 649
20009 San Sebastián (Spain)
Phone: + 34 943 018 064
{oscar.diaz, struji, felipe.anfurrutia}@ehu.es

Abstract. The promotion of a clear separation between artifact construction and artifact assembling is one of the hallmarks of software product lines. This work rests on the assumption that the mechanisms for producing products considerably quicker, cheaper or at a higher quality, rest not only on the artifacts but on the assembling process itself. This leads to promoting production processes as "first-class artifacts", and as such, liable to vary to accommodate distinct features. Production process variability and its role to support either production features or production strategies are analyzed. As prove of concept, the *AHEAD Tool Suite* is used to support a sample application where features require variations on the production process.

1 Introduction

1.1 Problem statement

Software Product Lines (SPLs) are defined as "*a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [7]. In this paper, we focus on "the prescribed manner" in which products are manufactured: **the production plan.**

A production plan is "*a description of how core assets are to be used to develop a product in a product line*" [6]. Among the distinct concerns involved in a production plan, this paper focuses on the **production process** which specifies how to use the production plan to build the end product [6]. As stated in [14] "*product production has not received the attention that software architecture or programming languages have*". It is often so tightly coupled to the techniques used to create the product pieces that both are indistinguishable. For example, integrated development environments (e.g. *JDeveloper*) make it seamless by automatically creating a build script for the project or system under development so that the programmer can be unaware of the process that leads to the end product.

Indeed, production plans have been traditionally considered as mere scripts, and left to the programmers that built the other artifacts. In a traditional setting, build scripts are often kludged together for that, built by people who would rather be writing source code than developing a process. Such scripts are notorious for their poor or misleading documentation [9], which was thought to be consumed by other core-asset developers.

SPLs change this situation by explicitly distinguishing between core-asset developers and product developers where the latter are involved in intertwining the core assets to obtain the end product. This distinction not only reinforces a separation of concerns between programming and assembling, but explains the preponderant and strategic role that production plans have in SPLs. That is, there is a growing evidence that the mechanisms for producing products considerably quicker, cheaper or at a higher quality, rest not only on the components but on the assembling process itself. Despite this observation, most approaches just support a textual description of the production plan [5], where variability or requirements specific to the production plan are almost overlooked.

1.2 Our contribution

Based on these observations, we strive to turn production processes into "*first-class artifacts*". Specifically, the main contribution of this paper rests on observing how the explicit and separate specification of the product process permits to account for variations at both the product and process level. To this end, the paper distinguishes between "product features" and "build-process features". By "product features" we meant those that characterize the product as such, whereas "build-process features" refer to variations on the associated process. Hence, two end products can share the same "product features" but being produced along distinct process standards.

We attempt to show some evidence of how this process variability impacts both the modifiability (i.e. variability along time) and the configurability (i.e. variability in the product space) of SPLs. To this end, these ideas are supported for *AHEAD* [3], a methodology for SPLs based on step-wise refinement. So far, the companion tool suite*, AHEAD TS,* (1) hides the production process into the integrated development environment, and (2) excludes build scripts from refinement. Hence, the upgrades include, (1) an explicit representation of the production process that AHEAD implicitly conducts, and (2) a refinement operator for production processes. Production processes are specified using *Ant* [12], a popular script language in the Java world.

The rest of the paper is structured as follows. Section 2 outlines the AHEAD methodology. Section 3 introduces the running example. Process refinement at work is the subject of section 4 and 5. Finally, conclusions are given.

2 A brief on the AHEAD methodology

Step-wise refinement (SWR) is a paradigm for developing a complex program from a simple program by incrementally adding details [11]. *GenVoca* is a design methodology for creating application families and architecturally extensible software, i.e., software that is customizable via module additions [2]. It follows traditional SWR with one major difference: instead of composing thousands of microscopic program refinements, *GenVoca* scales refinements so that each adds a whole **feature** to a program, being a feature a "*product characteristic that is used in distinguishing programs within a fam-*

ily of related programs" [3][1]. Hence, a final program (i.e. a product) is characterized as a sequence of refinements (i.e. features) applied to the core artifacts. This permits the authors to conceptualize the production process as a mathematical equation where core assets are mapped into constants, and refinements are functions that add features to the artifacts.

This approach is supported by the ***AHEAD Tool Suite** (AHEAD TS)* [3] where refinements to realize a feature are packaged into a **layer**. Broadly speaking, the base layer comprises the core artifacts, where lower layers provides the refinements that permit enhancing the core artifacts with a specific feature. The base layer is then, the root of the refinement hierarchy[2].

Layer composition implies the composition of the namesake artifacts found in each layer. Implementation wise, a layer is a directory. Hence, feature composition is directory composition. Artifact composition depends on the nature of the artifact. Hence, the composition operator is polymorphic. *Java* files, *HTML* files, *Ant* files will each have their own unique implementation of the composition operator.

For the perspective of the production process, it is most important to distinguish between:

– the intra-layer production process, which specifies the production process for the set of artifacts included within a layer or upper layers (from which artifacts are "inherited"). This is specified as *Ant* files in *AHEAD TS*. This would correspond to the "*product-build process*" in Chastek's terminology [6].
– the inter-layer production process, which specifies how layers are intertwined to obtain the end product. This is hard-coded in *AHEAD TS*. This is referred to as "*product-specific plan"* in Chastek's parlance [6].

Unfortunately, *AHEAD TS* does not consider yet *XML* artifacts. Since the production process is an *XML* document [3], production processes are not refined as such. Lower layers always override the *build.xml* file of upper layers so that the *build.xml* of the leaf layer is the only one that remains.

This implies that leaf layers should be aware of how to assemble the whole set of artifacts down in the refinement hierarchy. This could be a main stumbling block to achieve loose coupling among layers, and leads to increasing complex *build.xml* files as you go down in the layer hierarchy.

Turning production processes into first-class artifacts makes production processes liable to be refined as any other artifact. This permits to account for both "product features" and "process features". By "product features" we meant those that characterize the product as such, whereas "process features" refer to variations on the associated process.

[1] Other definitions of features include "*a logical unit of behavior that is specified by a set of functional and quality requirements*" [4] or "*a recognizable characteristic of a system relevant to any stakeholder*" [10].

[2] Design rules checking is also introduced to specify feature dependencies (e.g. selection of feature F1 disables feature F2) [1].

[3] *AHEAD TS* names it *ModelExplorer.xml*, but it plays the same role than *build.xml* in traditional Java projects.

It is worth noting that "product features" commonly impact the intra-layer production process (i.e. the process adds a new artifact to build the end product). By contrast, "process features" influence the inter-layer production process (i.e. the process that indicates how layers are intertwined). Again, this distinction reinforces the separation of concerns between asset developers and product developers.

Different upgrades were conducted into *AHEAD TS* to accommodate variability into the production process, namely

– intra-layer production processes are currently specified as *ANT* files [4]. Since *AHEAD TS* does not consider yet *XML* artifacts, and *ANT* files are *XML* document, production processes are not refined as such. Lower layers always override the *build.xml* file of upper layers so that the *build.xml* of the leaf layer is the only one that remains. This implies that leaf layers should be aware of how to assemble the whole set of artifacts down into the refinement hierarchy. This could be a main stumbling block to achieve loose coupling among layers, and leads to increasing complex *build.xml* files as you go down in the layer hierarchy. To overcome this situation, the *refinement* operator has been extended to handle *ANT* files.
– the inter-layer production process is hard-coded into the *AHEAD TS*. This production process is made explicit, and hence, subject to refinement.

Next sections illustrate the advantage of bringing refinement to the process realm through a running example.

3 The sample problem: *WebCalculator*

Batory et al. uses a Java-based calculator to illustrate how *AHEAD* can nicely accommodate the refinement process whereby features are gradually added to the core assets till the end product is obtained. In their example, refinements affect artifacts other than the product process [3].

We have used a similar domain but in a Web setting, and where variations mainly affect the production process. *WebCalculator* is a *J2EE* Web applications [16] which has been developed using Apache Struts[5]. A Web application refers to an aggregate of artifacts, namely

– *Java* class files (action classes), needed libraries, and resource files,
– *JSP* pages and their helper Java classes,
– Static documents: images, *HTML* pages, and so on,
– Web deployment descriptors, configuration files, and tag libraries.

Web applications (also known as *Web Modules*) are packaged into a *Web ARchive* (WAR) which follows a directory structure defined in *Java Servlet Specification* [8]. Mostly, this structure corresponds with *public_html* content which is shown on the left of figure 1.

[4] *AHEAD TS* names it *ModelExplorer.xml*, but it plays the same role than *build.xml* in traditional Java projects.
[5] http://struts.apache.org/

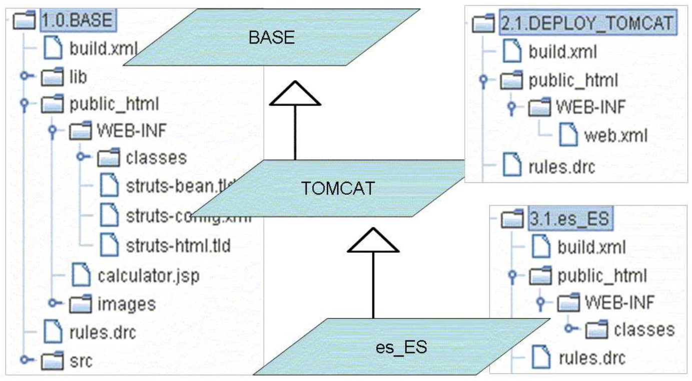


Fig. 1. Refinement layers: each layer accounts for a "product feature".

Broadly speaking, a layer comprises the set of artifacts that realize a given feature. This might include a process. Being in a Java setting, *Ant* is used to specify this process; the so-called, *build.xml* file. [15].

Ant is a Java-based tool for scripting build processes. Scripts are specified using XML syntax: <*project*> is the root element whose main child is <*target*>. A target describes a unit of enactment in the production process. This unit can be an aggregate of atomic tasks such as *compile, copy, mkdir* and the like.

The process itself (i.e. the control flow between targets) is described through a target's attribute: *"depends"*. A target is enacted as long as the target it depends to, has already been enacted. This provides a backward-style description of the process flow. Data sharing between targets is achieved through the external file directory.

Figure 2[6] shows a snippet of the specification of the production process for the *base* layer. According with this figure, the production plan includes the following steps:

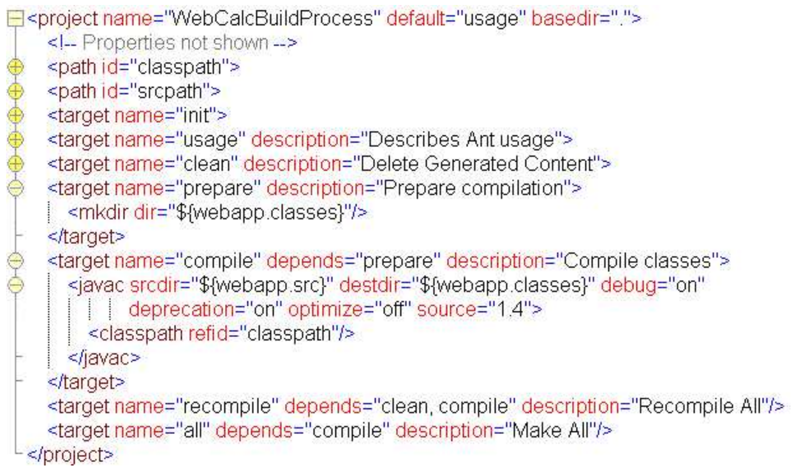
1. compile *Java* classes, and get the byte code,
2. package artifacts (classes, libraries, pages, resources, etcetera) into a *WAR* file,
3. deploy web application into a container.

The use of *Ant* for specifying product processes is not new. After all, *Java* programmers have been using *Ant* as a scripting language for years. However, instead of burying it into the integrated development environment, we make it explicit as any other artifact. This allows for refinements.

4 Intra-layer production process refinement

Previous section describes the *base* layer of *WebCalculator*. This *base* layer might then be refined to account for distinct "product-features". The example introduces two features which imply a refinement in the production process, namely

[6] Space limitations prevent us for given the full *build.xml* files. Some targets are collapsed and variables are defined in properties files.



```
<project name="WebCalcBuildProcess" default="usage" basedir=".">
    <!-- Properties not shown -->
    <path id="classpath">
    <path id="srcpath">
    <target name="init">
    <target name="usage" description="Describes Ant usage">
    <target name="clean" description="Delete Generated Content">
    <target name="prepare" description="Prepare compilation">
      <mkdir dir="${webapp.classes}"/>
    </target>
    <target name="compile" depends="prepare" description="Compile classes">
      <javac srcdir="${webapp.src}" destdir="${webapp.classes}" debug="on"
             deprecation="on" optimize="off" source="1.4">
        <classpath refid="classpath"/>
      </javac>
    </target>
    <target name="recompile" depends="clean, compile" description="Recompile All"/>
    <target name="all" depends="compile" description="Make All"/>
</project>
```

Fig. 2. Product process: base artifact

– the container feature. The variants include *Tomcat* [7] and *JBoss*[8]. By default, there is no *base* web container [9],
– the locales feature. The alternatives are *EN* (i.e. English), *es_ES* (Spanish at Spain), and *eu_ES* (Basque at Spain). The base locale is *EN*.

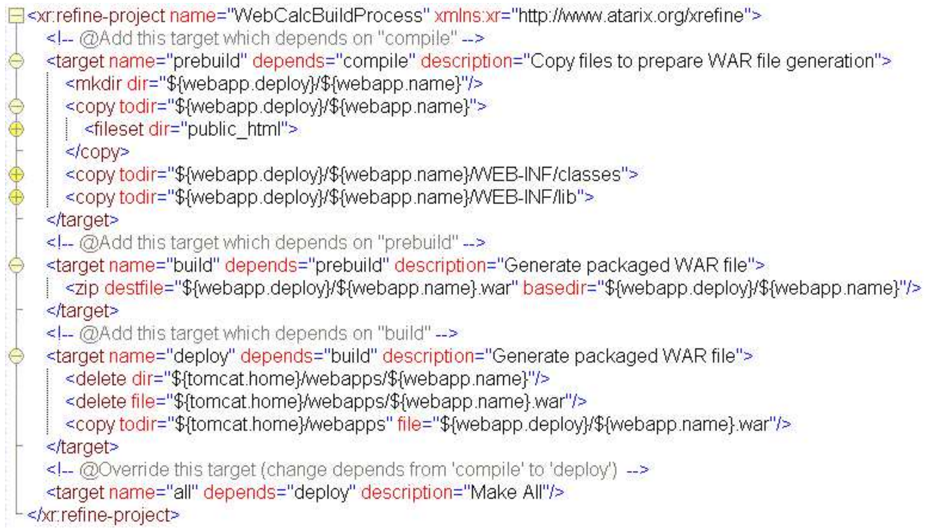
Figure 1 shows one possible layer composition, which equation is *"es_ES(Tomcat(base))"*. The *base* layer contains the base artifacts, whereas the other layers contain either refinements on existing artifacts or new artifacts. The important point to notice is that both *Tomcat* and *es_ES* features imply the refinement of the product process. That is, deploying *WebCalculator* in *Tomcat* requires to refine the *build.xml* accordingly.

AHEAD does not provide a way to refine *XML* artifacts. However, Batory et al. state the Principle of Uniformity whereby *"when introducing a new artifact* (type), *the tasks to be done are (1) to support inheritance relationships between instances and (2) to implement a refinement operation that realizes mixin inheritance*"[3].

This principle is realized for *build.xml* artifacts as follows. Inheritance is supported by building on the uniqueness of the <*target*> name within a given <*project*>. Basically, the project maps to the notion of class, and the target corresponds to a method. This permits to re-interpret inheritance for *Ant* artifacts by introducing the following tags:

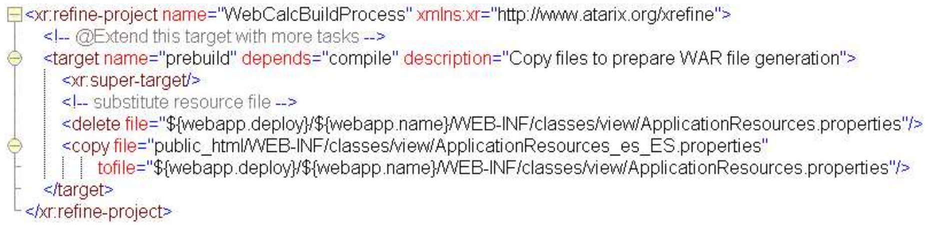
1. <*xr:refine-project*> which denotes a project refinement (a kind of "is_a"),
2. <*xr:super-target/*> which is the counterpart of the *"super"* constructor found in object-oriented programming languages

[7] http://jakarta.apache.org/tomcat/
[8] http://www.jboss.org/
[9] A design rule can be used here to ensure that the final product will have a container.



```
<xr:refine-project name="WebCalcBuildProcess" xmlns:xr="http://www.atarix.org/xrefine">
  <!-- @Add this target which depends on "compile" -->
  <target name="prebuild" depends="compile" description="Copy files to prepare WAR file generation">
    <mkdir dir="${webapp.deploy}/${webapp.name}"/>
    <copy todir="${webapp.deploy}/${webapp.name}">
      <fileset dir="public_html">
    </copy>
    <copy todir="${webapp.deploy}/${webapp.name}/WEB-INF/classes">
    <copy todir="${webapp.deploy}/${webapp.name}/WEB-INF/lib">
  </target>
  <!-- @Add this target which depends on "prebuild" -->
  <target name="build" depends="prebuild" description="Generate packaged WAR file">
    <zip destfile="${webapp.deploy}/${webapp.name}.war" basedir="${webapp.deploy}/${webapp.name}"/>
  </target>
  <!-- @Add this target which depends on "build" -->
  <target name="deploy" depends="build" description="Generate packaged WAR file">
    <delete dir="${tomcat.home}/webapps/${webapp.name}"/>
    <delete file="${tomcat.home}/webapps/${webapp.name}.war"/>
    <copy todir="${tomcat.home}/webapps" file="${webapp.deploy}/${webapp.name}.war"/>
  </target>
  <!-- @Override this target (change depends from 'compile' to 'deploy') -->
  <target name="all" depends="deploy" description="Make All"/>
</xr:refine-project>
```

Fig. 3. Product process: refinement for feature *Tomcat*.



```
<xr:refine-project name="WebCalcBuildProcess" xmlns:xr="http://www.atarix.org/xrefine">
  <!-- @Extend this target with more tasks -->
  <target name="prebuild" depends="compile" description="Copy files to prepare WAR file generation">
    <xr:super-target/>
    <!-- substitute resource file -->
    <delete file="${webapp.deploy}/${webapp.name}/WEB-INF/classes/view/ApplicationResources.properties"/>
    <copy file="public_html/WEB-INF/classes/view/ApplicationResources_es_ES.properties"
        tofile="${webapp.deploy}/${webapp.name}/WEB-INF/classes/view/ApplicationResources.properties"/>
  </target>
</xr:refine-project>
```

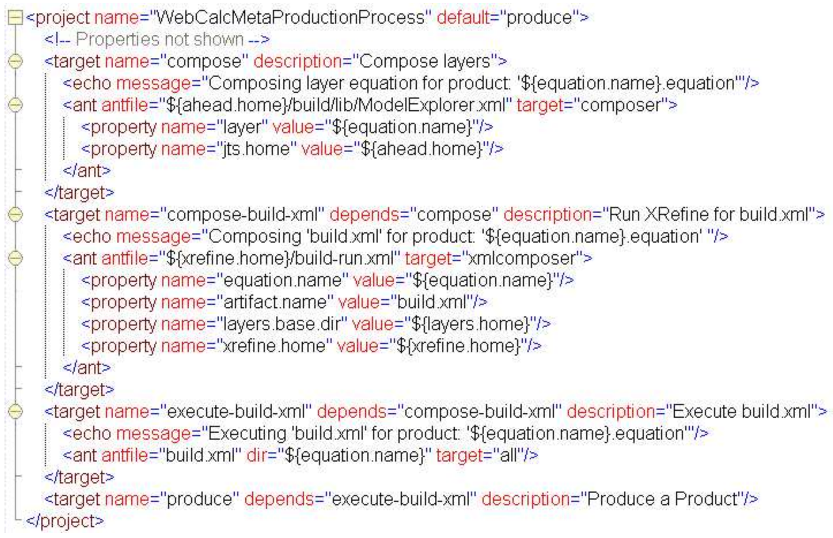
Fig. 4. Product process: refinement for feature *es_ES*.

Hence a <*refine-project*> can refine a <*project*> by introducing a new <*target*>, extending a previously existing <*target*> (calling <*super-target*>) or overriding a <*target*> (by introducing this target with new content).

An example is given for the *Tomcat* feature (see figure 3). Feature *Tomcat* permits to deploy *WebCalculator* in the namesake container. This requires the refinement of the *build.xml* artifact found in the base layer, as follows:

– a new <*target*> is added to prepare *WAR* building (prebuild),
– a new <*target*> is added to build *WAR* (build) specific for *Tomcat*,
– a new <*target*> is added to deploy it into the *Tomcat* container,
– target <*target name= "all"*> is overridden.

Likewise, feature *es_ES* overrides the English locale of the *base* layer to the Spanish locale. The counterpart refinement is shown in figure 4. It includes extending <*target*



```
<project name="WebCalcMetaProductionProcess" default="produce">
    <!-- Properties not shown -->
    <target name="compose" description="Compose layers">
        <echo message="Composing layer equation for product: '${equation.name}.equation'"/>
        <ant antfile="${ahead.home}/build/lib/ModelExplorer.xml" target="composer">
            <property name="layer" value="${equation.name}"/>
            <property name="jts.home" value="${ahead.home}"/>
        </ant>
    </target>
    <target name="compose-build-xml" depends="compose" description="Run XRefine for build.xml">
        <echo message="Composing 'build.xml' for product: '${equation.name}.equation' "/>
        <ant antfile="${xrefine.home}/build-run.xml" target="xmlcomposer">
            <property name="equation.name" value="${equation.name}"/>
            <property name="artifact.name" value="build.xml"/>
            <property name="layers.base.dir" value="${layers.home}"/>
            <property name="xrefine.home" value="${xrefine.home}"/>
        </ant>
    </target>
    <target name="execute-build-xml" depends="compose-build-xml" description="Execute build.xml">
        <echo message="Executing 'build.xml' for product: '${equation.name}.equation'"/>
        <ant antfile="build.xml" dir="${equation.name}" target="all"/>
    </target>
    <target name="produce" depends="execute-build-xml" description="Produce a Product"/>
</project>
```

Fig. 5. Layer-composition product Process: *base* artifact.

name= "prebuild"> to copy appropriate resource files. Here the *<xr:super-target/>* constructor is used[10].

Both examples illustrate how refinements have been realized for *Ant* artifacts. Implementation wise, the composition operator for *Ant* is implemented using *XSLT* and *XUpdate* [13]. This operator can be integrated within *AHEAD TS* so that when *build.xml* artifacts are found, the composition process is governed by the *Ant* plug-in.

5 Inter-layer production process refinement

Previous section focuses on *Ant* artifacts found within a layer. These artifacts describe the "*product-build process*" within a layer. By contrast, this section focuses on layer-composition processes that state how layers themselves should be composed. This comprises the steps of the methodology being used. For AHEAD, these steps include:

1. feature selection. Output: a feature equation (e.g. *"es_ES(Tomcat(base))")*.
2. feature composition (i.e. layer composition in Batory's parlance). Output: collective of artifacts that support an end product.
3. enactment of the *build.xml* associated with the end product. Output: end product ready to be used.

Figure 5 illustrates the targets that realize previous steps *(*the *equation.name* property holds the feature equation):

[10] It is worth noticing that the *es_ES* refinement requires the container been already selected. This implies a design rule to regulate how layers are intertwined.

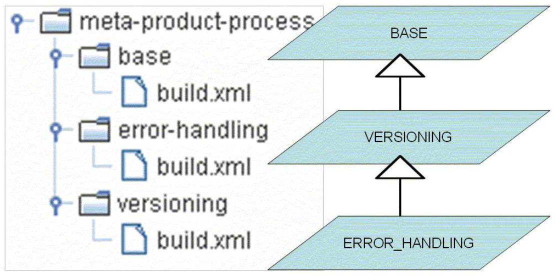


Fig. 6. Refinement layers: each layer accounts for a "process feature".

- *compose*, which calls the *AHEAD TS* composer,
- *compose-build-xml*, which supports the composition operator for *build.xml* artifact that *AHEAD TS* lacks,
- *execute-build-xml,* which runs the *Ant* script supporting the production process of the end product,
- *produce*, which performs the whole production.

The enactment of this layer-composition script leads to an end product that exhibits the features of the input equation. *AHEAD TS* hard-codes this script.

However, this work rests on the assumption that the mechanisms for producing products considerably quicker, cheaper or at a higher quality, rest not only on the artifacts but on the assembling process itself. From this viewpoint, the inter-layer production process can accommodate important production strategies that affect the process rather than the characteristics of the final product. These strategies can affect the product costs, increase product quality, or improve the production process.

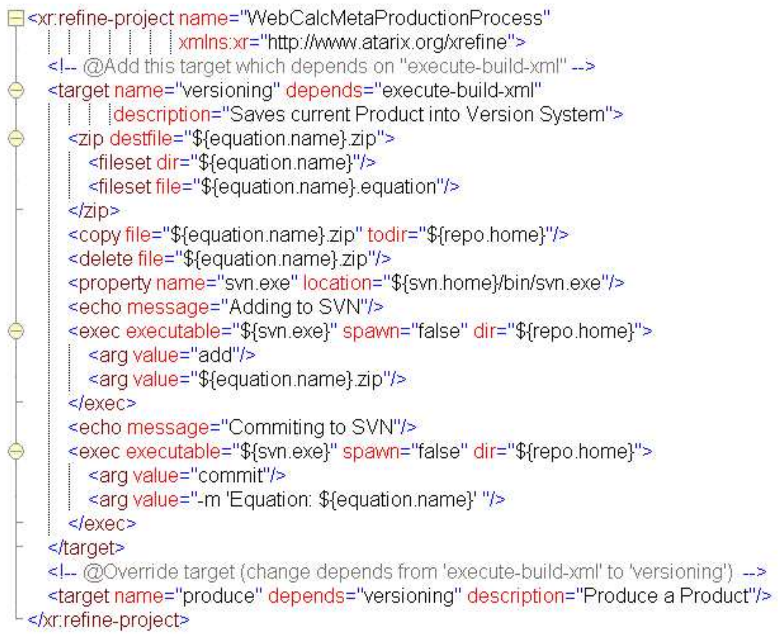
Based on this observation, the previous *base* layer might be refined to account for distinct "process-features". The example introduces two features which imply a refinement in this layer-composition process, namely

- the *version* feature. Consider that security reasons recommend to version each new delivery of an end product. This implies that artifacts that conform the end product, should have appropriate backups.
- the *errorHandling* feature. Errors can rise during the production process. How these errors are handled is not a characteristic of the product but depends on managerial strategies. Hence, the base process can be customized to support distinct strategies depending on the availability of resources or the quality requirements of the customer.

Figure 7 shows how the *base* process can now be refined to account for the *version* feature, namely:

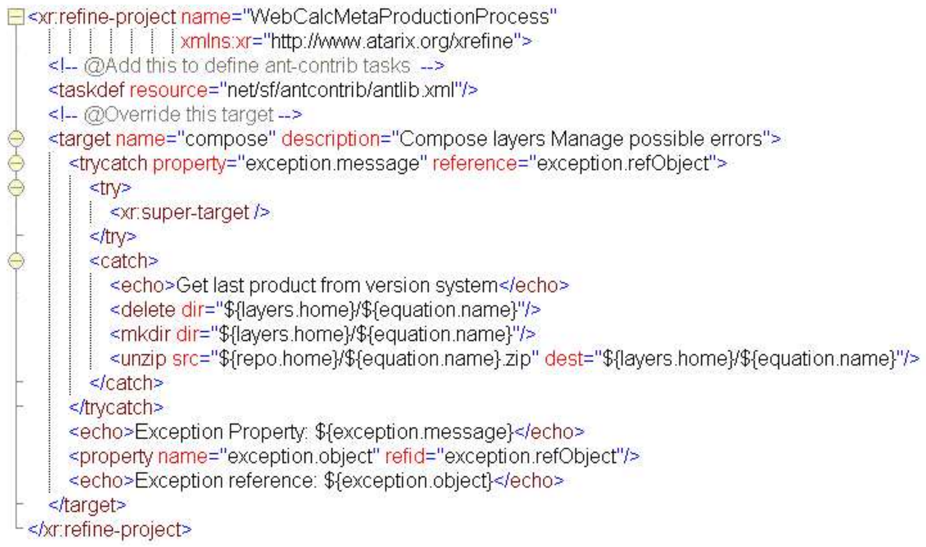
- a new <*target*> is added to back up artifacts into the versioning system. For this purpose, *Subversion* is used[11],

[11] http://subversion.tigris.org/



```xml
<xr:refine-project name="WebCalcMetaProductionProcess"
                   xmlns:xr="http://www.atarix.org/xrefine">
  <!-- @Add this target which depends on "execute-build-xml" -->
  <target name="versioning" depends="execute-build-xml"
          description="Saves current Product into Version System">
    <zip destfile="${equation.name}.zip">
      <fileset dir="${equation.name}"/>
      <fileset file="${equation.name}.equation"/>
    </zip>
    <copy file="${equation.name}.zip" todir="${repo.home}"/>
    <delete file="${equation.name}.zip"/>
    <property name="svn.exe" location="${svn.home}/bin/svn.exe"/>
    <echo message="Adding to SVN"/>
    <exec executable="${svn.exe}" spawn="false" dir="${repo.home}">
      <arg value="add"/>
      <arg value="${equation.name}.zip"/>
    </exec>
    <echo message="Commiting to SVN"/>
    <exec executable="${svn.exe}" spawn="false" dir="${repo.home}">
      <arg value="commit"/>
      <arg value="-m 'Equation: ${equation.name}' "/>
    </exec>
  </target>
  <!-- @Override target (change depends from 'execute-build-xml' to 'versioning') -->
  <target name="produce" depends="versioning" description="Produce a Product"/>
</xr:refine-project>
```

Fig. 7. Layer-composition process: refinement for feature *versioning.*



```xml
<xr:refine-project name="WebCalcMetaProductionProcess"
                   xmlns:xr="http://www.atarix.org/xrefine">
  <!-- @Add this to define ant-contrib tasks -->
  <taskdef resource="net/sf/antcontrib/antlib.xml"/>
  <!-- @Override this target -->
  <target name="compose" description="Compose layers Manage possible errors">
    <trycatch property="exception.message" reference="exception.refObject">
      <try>
        <xr:super-target />
      </try>
      <catch>
        <echo>Get last product from version system</echo>
        <delete dir="${layers.home}/${equation.name}"/>
        <mkdir dir="${layers.home}/${equation.name}"/>
        <unzip src="${repo.home}/${equation.name}.zip" dest="${layers.home}/${equation.name}"/>
      </catch>
    </trycatch>
    <echo>Exception Property: ${exception.message}</echo>
    <property name="exception.object" refid="exception.refObject"/>
    <echo>Exception reference: ${exception.object}</echo>
  </target>
</xr:refine-project>
```

Fig. 8. Layer-composition process: refinement for feature *errorHandling.*

– target <*target name*= *"produce"*> is overridden.

The equation *versioning(base)* leads to a "*product-specific plan"* that supports the naive security policy of the organization. As further experience is gained, and stringent demands are placed, more sophisticated plans can be defined.

Likewise, figure 5 shows the *"substitution_eh"* policy for error handling:

– a new <*taskdef/*> is added in order to extend *Ant* targets with try&catch routines[12].
– target <*target name*= *"compose"*> is overridden in order to handle possible errors. The base task *"compose"* is monitored so that when an error occurs, the last error-free version of the artifacts outputted by *"compose"* are taken. This policy might be applicable under stringent time demands or if debugging programmers are on shortage.

The "process feature" equation (*"substitution_eh(version(base)))"*) then strives to reflect the managerial and strategic decision that govern the production plan. Making theses strategies explicit facilitates knowledge sharing among the organization, facilitates customization, eases evolution, and permits to manage resources for product production in the same way as the product itself.

The latter is shown for the *version* and *errorHandling* features: a design rule is needed to state that the *substituion_eh* policy requires the *version* feature to be in place. The *version* feature in turn requires a new artifact, namely, *subversion*. It is a well-known fact among programmers of complex systems, that setting the appropriate environment is a key factor for efficient and effective throughput. SPLs are complex systems, and SPL techniques should be used not only to manage the artifacts of the product itself, but also those artifacts that comprise the environment/framework where these products are built. These include a large number of artifacts such a compilers, debugger, monitors or backup systems. Making explicit the layer-composition process facilitates this endeavour.

6 Conclusions

The clear separation between artifact construction and artifact assembling is one of the hallmarks of software product lines. However, little attention has been devoted to the assembling process itself, and how this process might realize important process strategies.

This work strives to illustrate the benefits of handling production processes as "*first-class artifacts*", namely:

– it permits to focus on how the product is produced rather than at what the product does. Programmers and assemblers can wide their minds to ascertain how features might affect the process itself so that scripting is no longer seen as a byproduct of source code writing,
– it extends variability to the production process itself.

[12] This is achieved using Ant-Contrib at http://ant-contrib.sourceforge.net/

Using *Ant* for process specification, and *AHEAD* as the SPL methodology, this work illustrates this approach for a sample application. Our next steps include to increase the evidence of the benefit of the approach by addressing more complex problems, and to investigate on the impact that distinct SPL quality measures have into the production process.

Acknowledgments

This work was partially supported by the Spanish Science and Education Ministry (MEC) under contract TIC2002-01442. Salvador Trujillo enjoys a doctoral grant for the MEC.

References

1. D. Batory and B.J. Geraci. Composition Validation and Subjectivity in Genvoca Generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, February 1997.
2. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.
3. D. Batory, J.Neal Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.
4. J. Bosch. *Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.
5. G. Chastek, P. Donohoe, and J.D. McGregor. Product Line Production Planning for the Home Integration System Example. Technical report, CMU/SEI, September 2002. CMU/SEI-2002-TN-029.
6. G. Chastek and J.D. McGregor. Guidelines for Developing a Product Line Production Plan. Technical report, CMU/SEI, June 2002. CMU/SEI-2002-TR-06.
7. P. Clements and L.M. Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, 2001.
8. D. Coward and Y. Yoshida. JSR 154, Java Servlet 2.4 Specification, 2003. http://www.jcp.org/en/jsr/detail?id=154.
9. J. Creasman. Enhance Ant with XSL Transformations, 2003. http://www-128.ibm.com/developerworks/xml/library/x-antxsl/.
10. K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.
11. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
12. Apache Software Foundation. Apache Ant. http://www.ant.apache.org/.
13. A. Laux and L. Martin. XUpdate - XML Update Language. http://xmldb-org.sourceforge.net/xupdate.
14. J.D. McGregor. Product Production. *Journal Object Technology*, 3(10):89–98, November/December 2004.
15. N. Serrano and I. Ciordia. Ant: Automating the Process of Building Applications. *IEEE Software*, 21(6):89–91, November/December 2004.
16. I. Singh, B. Stearns, and M. Johnson. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley, 2002.