Management of Composites in Software Engineering Environments

Jacky Estublier, German Vega, Thomas Leveque
Laboratoire Informatique de Grenoble
F-38041, Grenoble cedex 9, France
{Jacky.Estublier, German.Vega, Thomas.Leveque}@imag.fr

Abstract

Design and development scalability, in any engineering, requires information hiding and a specific composition mechanism in which composite items are made-up of other items. This paper shows that scalability is currently ill supported and that software engineering composites are rather special with respect to other engineering disciplines. Indeed, a software engineering composite is simultaneously a model element, an engineering artifacts and a real object, which is unprecedented in the history of engineering. This paper analyzes the requirements that composites must satisfy in order to support scalability in software engineering.

We have developed CADSE (Computer Aided Domain Specific Environment in which composites are first class elements from which workspaces, concurrent engineering and view point support are provided. The paper discusses our experience with using the proposed system over the last years.

1. Introduction

In all disciplines, design and development scalability relies on the concept of abstraction, because by definition, abstraction consists of ignoring “details” and therefore reducing the amount of information to manage at a given level. Models and composites are the usual ways to build abstractions. In this context, a composite [1] represents an entity as a whole made up of other entities that are considered its parts. As such, a composite reduces complexity by treating multiple entities (the parts) as a single entity (the composite).

In order to support scalability, a composite must be seen, from an abstraction level, as an atomic component. For example, for a certain purpose, it suffices to say that a car contains an engine, even though an engine is actually a complex artifact containing multiple elements. The larger the subject entity is, the more critical the use of composites as abstraction mechanisms becomes. For example, it would be rather difficult to manage a car if seen as an assembly of thousands of independent atomic pieces.

Of course, the same occurs in software engineering; abstraction and composites have been intensively used, including high-level languages for reducing the amount of lines of code, classes as composites for hiding internal code, or more recently models as general abstractions for hiding underlying details. However, software engineering faces problems not addressed in other engineering disciplines. At least two factors explain the differences.

The first factor is that most of the other engineering disciplines are designing and producing real objects, such as a car, or a bridge. Real objects have the natural part/whole property, which is a special case of a composite. The physical structure of real objects imposes a “natural” hierarchical decomposition into parts and therefore a “natural” composite concept. Conversely, software is an abstraction that does not feature any “natural” decomposition. Instead, software parts, their nature and their structure can be rather arbitrary. Hierarchical software decomposition is not natural and is often inconvenient since it is common for different composites to share the same sub-components (e.g. libraries). Therefore, in software engineering the very concept of composite may take a large range of definitions [2], [3]. Software composites are not necessarily hierarchical, nor complete nor homogeneous (i.e. the parts are not of the same nature as the whole). In software, relationships among elements, such as dependencies, are rather pervasive, meaning that they can cross the composite boundaries.

The second factor lies in the fact that, in other engineering disciplines, there is a clear distinction between the model and the real artifact. An engine can be an atomic element in a given model but whatever the model’s abstraction level, the model of an engine is not an engine. For example, an engine’s weight can be computed as the sum of the corresponding weights of its parts, but still the engine model does not weigh anything.

In software engineering, the source code can be seen as a model of the application to build, and the executable code is the application. Since the executable can be derived almost “for free” from the source, the model and the actual application are frequently confused and the model is considered as the real object. In contrast with other engineering disciplines, the Software Engineer considers some models to be the target object, and therefore they ask some composites to really produce their intended behavior. For example, one expects a software configuration not only to be a list of components (i.e. a composite), but also to be executable, i.e. to be the target object. This is unprecedented in engineering history and has far reaching
consequences, especially on the concepts of model, composite and abstraction.

This paper describes CADSE (Computer Aided Domain Specific Environment) which provides an advanced system model including composites that support the software engineering properties discussed above.

The rest of the paper is organized as follows. Section 2 presents the general issue of supporting models in software engineering. Section 3 discusses the properties required for a composite and the semantics of the composition relationship. Section 4 presents the different roles that composite can play in Software Engineering. Section 5 shows how composites can be built. Section 6 presents our experience and evaluation. Finally, sections 7 and 8 discuss the related work and conclusions, respectively.

2. Domain-specific system models

A system model is an abstract, coarse grain view of the system's main parts (e.g., product, sub product, configuration, component) and interrelations. A system model aims at abstracting away from the underlying data (e.g. often files and directory structure), in order to describe and manage the logical organization of the software product [17], [4], [5]. A system model can be described using a generic metamodel like eCore which describes the structure but lacks any kind of semantics. To get more semantics, a solution is to use metamodels specific to an application domain (called simply a domain later on) defining its main concepts, architecture and constraints. In a similar way, the Microsoft Software Factory initiative [6] and the Product line technologies [7], [8] focus on modeling the targeted application domains. Most of the semantics attached to a domain is found in the mapping from the model items toward the development platform...

![Diagram of a simple Component model](image)

**Figure 1. A simple Component model.**

Figure 1 provides an example of a domain intended to support the development of applications in a simple component model. This domain defines the composites CComp, DeploymentUnit and TestSuite based on the concept of Component. Although the model only captures the structure of artifacts in the domain, the characteristics associated with the relationships specify some of the intended semantics of these associations.

2.1 Relationship characteristics

We have slightly extended the eCore relationship characteristics in order to indicate some predefined relationship semantics and constraints.

Aggregation (empty diamond). The only constraint we added is that aggregate links are not cyclic. This property is used to structure items. Aggregate links can be used to define any set of objects, not necessarily composites. This is indicated by the <<A>> stereotype in Figure 1.

Part (black diamond). Part links define a containment relationship in which there is a life cycle dependency between the containing element and the contained element [9]. The constraints we enforce are the following:

- An item can be destination of at most one part link,
- Part is a non-cyclic relationship,
- Deleting a container deletes all the contained items,
- Deleting a part link deletes the destination item.

Part links define a strict life cycle dependency between a container and its contained items. Part imposes a hierarchically structured decomposition and does not allow for component sharing. This is indicated by the black diamond and <<P>> stereotype in Figure 1.

Require. The require relationship means that the successful functioning of the relationship’s origin item depends on the successful functioning of corresponding destination item like a compilation or an execution dependency. This is indicated by the <<R>> stereotype.

The defined relationship characteristics are independent from one another and are composable. Consequently, the system recognizes eight different relationship semantics.

It is important to note that CADSE is driven by a number of models, only one of which is the domain model presented above. CADSE is most importantly driven by models describing the “ideal” software engineering environment to be used in the targeted application domain. Indeed, most CADSE models are related to the production environment and tools, including composite and build capabilities (subsection 0), versioning and evolution, concurrent engineering, view points and IDE mappings. A CADSE is a Domain-Specific Application Environment (DSAE) as advocated in [12].

3. The composition relationship

The system presented above proposes eight possible combinations of relationship characteristics (based on aggregate, part and require), but no combination satisfies the fundamental requirements expressed in the introduction:

- A composite must be able to hide its content (we will say the composite is closed).
• A composite must exhibit the behavior of the composition of its components, even when closed.

Therefore, we have defined a fourth link characteristic called \textit{composition}, which is dedicated to the support of software engineering composites (indicated by the \texttt{<<C>>} stereotype in Figure 1). An item that is the origin of at least one composition link is called a composite.

A composite is an item composed of all the items that constitute the destination of a composition link starting from that item.

The composition link is such that composites can be nested, they are not necessarily homogeneous, and components can be shared by multiple composites.

The life cycle of a component is not related to the life cycle of the composite(s) it pertains to. Therefore, \textit{composition} and \textit{part} are two unrelated link characteristics; a link can be both \textit{composite} and \textit{part} making 16 different predefined link semantics possible.

A composite can be used in two ways: 1) as a black box, in which case it is an abstraction mechanism and the items it contains are not visible (\textit{closed composite}); or 2) as a white box, in which case it is a structuring mechanism and the items it contains are visible (\textit{open composite}).

3.1 Closed composites and links

\textit{Closing} a composite, means to safely remove from the model the items pertaining to the composite. This approach maintains the consistency and behavior of the model unchanged (i.e. the consistency constraints are still satisfied, and the questions asked to the model provide the same answers) whether the composite is open or closed.

A closed composite hides the items it contains, but the contained items may have links towards items not contained in the composite. Therefore, a composite has indirect relationships towards items that its components reference. For that reason, closing a composite creates “derived links” towards these items.

For illustration, consider the simple scenario depicted in Figure 2 in which a Composite Component CC contains components C1 and C2, and C2 references a library L1. This model is an instance of the component metamodel presented in Figure 1. Part (a) of Figure 2 depicts a situation in which the CC composite is open and part (b) presents the same situation after CC has been closed: components C1 and C2 are removed from the model.

Because of the \texttt{Lib} link between C2 and L1, closing CC creates the \textit{derived link} “\texttt{#Lib}” between CC and L1. Since CC, when closed, is supposed to provide the behaviors of its components, links leading to CC’s components are replaced by derived links toward CC. For example, because of the \texttt{Uses} link between Ap and C1, closing CC creates the derived link \texttt{@Uses Between Ap and CC} In this case, the \texttt{Uses} link is not deleted, but leads to a \textit{missing item}. The \texttt{Uses} link is said to be \textit{unresolved}.

Derived links inherit the characteristics of their original link. For example, if C2 \texttt{requires} L1, then CC, when closed, requires L1 too. Similarly, if Ap \texttt{requires} C1, then it will require CC (as a C1 substitute) since C1 is no longer available.

3.2 Closed composite and real behavior

A composite must be such that when closed, it can be safely used in place of the set of items it contains. To that end, we require from each composite type to define a \texttt{build} operation, which is automatically called when the composite is closed. The build function associated with a composite type is in charge of computing (or building) out of its components a single value that will supersede those of its components. For example, the \texttt{weight} attribute of the engine can be computed as the sum of the \texttt{weight} attributes of each of its components.

A composite can be safely used in place of its components only if the behavior of the system is the same whether using the composite-built value or the individual values of its components. In other words, when closed, the components of a composite are not allowed to change in such ways that opening the composite again and rebuilding it would lead to a different build result. For example, one should rely on the engine weight when working on the car suspension (the engine is closed). The weight of a component must only be changed when the engine is open (i.e. working on the engine) or when it leads to another engine version.

Closing a composite implies freezing its built value and freezing the corresponding value of its components. Therefore, when a composite is closed, its components are removed from the model. This behavior enforces the abstraction and scalability property of a composite (lower level entities can be ignored) and it preserves the consistency property of the composite.
Conversely, since the build function requires the availability of all the composite’s components, all the components of an open composite must be present. As a consequence, if a component is shared by different open composites, closing one of them will not remove that component, as long as the component is part of an open composite; but that component is set to a ReadOnly mode.

4. Composites in Software Engineering

CADSE link the system model items with the corresponding engineering artifacts as described in [23]; therefore, composites are not only a structuring concept but also a real engineering object. This has deep implications: (1) an item is not merely a collection of attributes and links, but also a set of electronic entities that may be very “heavy” (mega bytes or gigabytes); (2) working on an item may (and does) take time and (3) as an item represents the real object, many engineering operations may be associated with the item, such as building, executing, testing, measuring, or packaging the object.

As any other item, a composite is defined for a given purpose, which include developing (like CComp), testing (like TestSuite), debugging, deploying and so on. This has consequences on the build function. If a composite’s purpose is testing, then the build function can collect the unit tests of each component and compute the integration tests; if the composite’s purpose is deployment, then the build function can compute a JAR file. Therefore, a composite is the result of a computation (i.e. the build function) that supersedes its components for the composite purpose. If one is only interested in these attributes values, the composite can be seen as a substitute for all its components. This has a number of consequences and the composite’s definition is extended as follows:

A composite must be such that, when closed, it can be safely used, for a given purpose, in place of the set of items it contains.

Having multiple operational views is especially important in software engineering because there is no “natural” decomposition of the model and therefore many arbitrary views can be defined. As a composite is related to a single purpose, with respect to the sum of its components, a composite is lighter weight and cheaper to build. For example, the deployable composite only contains the packaged executable.

5. Composite Building

From an engineering perspective, composites can easily be related to activities such as design, development, debug, packaging, deployment, or documentation. It is therefore “natural” to relate the concept of composite to such activities. Since software engineering activities are performed in workspaces, it follows that the concepts of
composite, workspace and concurrent engineering are closely related, and should as such work in symbiosis.

5.1 Workspaces, activities and composites

One can populate a workspace by importing a composite from a repository; opening the composite brings into the workspace all the information (model and artifacts) required for the activity for which the composite has been designed. Having different workspaces containing composites of different types over the same shared components is the basic mean for concurrently working on the same entities from different perspectives.

If a model is an abstraction of a system for a certain purpose [14], [15], then a workspace can be seen as containing a system model and its associated engineering artifacts; which provides a point of view and a given abstraction level over the system.

The most significant difference with respect to most modeling approaches is that the proposed workspace does not only contain a view, but also the artifacts required for the task at hand (and only those artifacts). It means that activities such as editing, executing, testing, or building the documentation, can really be performed whatever the level of abstraction. The workspace is both an abstract model and an operational point of view.

5.2 Composite building

Building a composite consists of computing the value of the composite from a sub-set of the value of its components. For example, the DeployableUnit composite is only interested in the binary of its components. The composite build function is structured into two operations: exporters and composers. An exporter is a filter that provides access to only a subset of an item’s content. A composer is an action that is executed over the information provided by exporters, with the goal of computing (building) the composite’s content.

An item type can be associated with several exporters and a composite type is associated with a builder. A builder is defined as an ordered list of triples (composer, relation, exporter), which allows the components to be handled differently based on their actual composition relationship, irrespective of their actual internal structure. In contrast, usual build systems only manage one type of dependency and need to know the implementation details.

To illustrate, take our running example. The CADSE designer could map a Component item type to a Java project, CComp to an AspectJ Eclipse project, and DeploymentUnit to a Jar file. Then, a java_binary exporter can be associated with Component. The CComp builder can be defined as \{copy, contains, java_binary\} and the DeploymentUnit builder can be defined as (jarMaker, unitOf, java_binary), (weave, null, null). The jarMaker composer produces a jar file from the classes provided by the exporter. The weaver composer weaves the aspects only if its composite item is not also a component of a higher level composite. Composers are ordered so as to be sure that aspects are weaved after copying compiled Java classes.

The predefined Build command associated with each composite is a generic builder that analyzes the current state of the workspace (composition relationships) and subsequently invokes the right exporter for each component, gathers the result and invokes the composers in the right order. In order to keep composites up to date, the building process includes a notification mechanism that triggers the rebuilding of the affected composites as soon as a component’s attribute changes. The set of composites to rebuild is computed based on the composition links leading to the modified item, by following them in the reverse direction.

Since the mapping between item types and IDE artifacts is very flexible, in CADSE the build granularity can be anything, from a large number of projects down to a few lines in a file. In contrast, most build systems are building only complete projects, with predefined knowledge of their internal structure. Our system currently provides default exporters and composers for a number of common software engineering entities and languages. Like for mappings, the system is extensible and allows defining new kinds of composers or exporters.

If compared with make, Maven or Ant, the actions are described in the composer and the data is defined by the exporters. The process and dependencies are defined in the workspace model. Exporters and composers are defined at the type level, the model is available for free (and is up to date) and the building process does not require writing any makefiles or build.xml files.

6. Experience and Evaluation

The system described here is daily used since early 2006 to develop itself and a dozen applications spanning a large range of domains: SOA Engineering environments [22], Service developments (in OSGi and iPOJO), operating system, orchestration and process support, document management systems and so on.

Composite management design and implementation has been a major challenge, because a composite is simultaneously an abstraction, a point of view, an operational object and a model in an OO context.

Abstraction. Closing a composite proved to be a powerful abstraction mechanisms. On average, workspaces are one or two orders of magnitude smaller than working with Eclipse alone (because of the closed composites). A typical large workspace (docManager), with its composites open, contains 210 items in the model, 243 Eclipse projects and 2967 files. Working on one of its composite (Apel) and closing the other composites reduces the same workspace
down to 35 projects and 408 files. This aspect of the system has been greatly appreciated by our customers.

The clear separation between the system model and its mapping on Eclipse artifacts is one of CADSE’s most distinctive features. The automatic synchronization between Eclipse and CADSE proved to be greatly appreciated like, for example, automatic classpath computation based on the transitive closure of the require links.

**Point of view.** We soon realized that a composite is the fundamental mechanism from which complete viewpoints-based environment can be designed and built. The current state described here provides a large fraction of the expected viewpoint features, but it is a rather pragmatic approach. There is no formal definition of what is a viewpoint, we ignore the semantics of items and relationships, and we ignore the purpose of composites and the task to be performed in workspaces.

For example, our ambition with derived links was to allow transparent navigation through closed composites as if their contained components were present, depending on the workspace’s purpose. The design of such a system has been realized, but the complexity of the associated mechanism and the additional work for CADSE developers determined us to temporarily put this ambition on hold.

**Operational object.** Operational composites and builds proved to be a very challenging issue. It involves copying inside the composite the required information, such as copying all the java .class files and building a Jar file out of them. Since Eclipse does not support nested and operational composites, Eclipse build processes do not make any copies. For that reason, composite builds need either to write an Eclipse builder, or to use our exporter/composer mechanism. For a 3 levels nested composite with 16 Java classes and 8 directories at each level, the specific Eclipse builder takes 14.9 seconds, while our build only takes 4.3 seconds (the same as an ANT script executed outside Eclipse). The reason is that Eclipse tries to rebuild everything at each level, while our composers know if a local rebuild is required or not and rebuilds only when needed. Our builds are highly optimized, while still leaving Eclipse consistent.

Integrating different kinds of build mechanism like Eclipse, Make, Ant, or Maven is a heavy task. Our generic build mechanism transparently synchronizes heterogeneous build systems (including the Eclipse one). This facility proved to be highly appreciated, since we simultaneously use Eclipse, Ant and Maven.

To simplify composer and exporter development, we added the possibility to write Ant or Maven code to describe their behavior. Nevertheless, in its actual form, defining new builders remains complex. Future work will extend our generic build manager, adding priority, defining build lifecycle and a better listener management.

Being primarily focused on large software applications, with continuous integration strategy, it was only when our build became efficient that our customers started to appreciate the system (original versions were much slower than an external Ant). Eclipse slowdown when working is not visible at all but the Eclipse footprint is increased in average by a 1.3 factor.

**Model and semantics.** The system model has very little semantics, as it “only” describes the structural constraints imposed to each component application. Most of the semantic and domain-specific knowledge is contained in the other models: mapping model (specifying how each concept must be represented in the actual IDE), building model (specifying how to compute a composite value), evolution model (how versioning must be handled) and a configuration model (how to select the components of a composite).

In short, the system model describes the business domain concepts, while the other models describe the computing environment in which applications in this domain will be designed and built.

**Composite management.** The system presented above results from years of work and experience and solves many conceptual and practical issues. CADSE implementation proved to be of high usability and efficiency. This was made possible because our composite definition is closely tailored to Software Engineering practices. This definition states that, when closed, a composite can be used in place of its components. However, even in Software Engineering, many composites do not satisfy this definition, as for example in dynamic service_based applications. Future work includes an attempt to generalize the approach for different kinds of composite, with different properties.

7. Related work

CADSE is at the intersection of three domains: Software Configuration Management systems (SCM), design environment and Integrated Development Environments (IDE).

CADSE environments share with SCM systems the fact that they both manage a model and the associated software artifacts. Nonetheless, SCM system models are rather low level, describing the organization of files and directories in a development machine [4], [17]. In SCM, the system model is manually defined and maintained, and therefore expensive, unreliable and easily outdated [5]. For that reason, commercial SCM systems do not use system models, there is no general composite concept and configurations are ad hoc entities with ad hoc building and management. Both SCM and CADSE are addressing workspace, concurrent engineering and versioning functionalities.

In contrast, the CADSE (system) model is automatically synchronized with reality and therefore always up to date while incurring zero maintenance costs. Composites are general model entities subsuming the usual software configuration concept. CADSE closely supports (with the
help of the IDE) the activity in the workspace and proposes the viewpoint concept. In CADSE, artifact versioning relies on usual versioning systems, similarly to SCM systems. Nonetheless, model versioning required some additional specific work, as described in [13].

With respect to design environments, CADSE shares the general goal of providing an abstract view of the system to build. However, design environments are primarily focused on defining the logical structure and behavior of the target software system. Recent environments propose generating source code but only those targeting a limited domain (e.g., Real time for Rose RT (Rational) and Rhapsody (IBM); embedded system for BridgePoint (MentorGraphics); system engineering for Tau (Telelogic)) can generate a significant amount of code; the others often simply transform a modeled concept into a class skeleton in a programming language.

CADSE system model is only structural; the semantics of the model is not the target application behavior but the software environment semantics, expressed in associated models (mapping, build, evolution, composite and so on). The usual modeling formalisms often propose a concept of composite (black diamond in UML), but with different goals. For example, UML2 extensions are targeting the description of physical entities, like in PDM. This can be a consequence of the fact that O.O. emphasizes the inheritance structure (ISA relationship), and provides little support for other structuring concept (e.g. part of, or composite). In all cases, no engineering operation (like execute or build) can be performed on the model. No system we know of currently addresses the issues raised here: the modeling and management of operational software composites.

Most meta-level environments (XMF[19], GME, GMF[11], Kermeta[21]), and transformation languages (QVT) provide formalism for the synchronization between two models. The Triple Graph Grammar approach follows a similar goal [18]. Automatically updating the model when the source code is changed is only provided by a few tools, provided strict conventions are being followed. This round-trip engineering issue is well acknowledged in the MDE community [12].

The reverse engineering community has developed concepts and techniques for “understanding” programs and therefore “finding” a model corresponding to a set of sources. However, support for automatic synchronization was not provided [20].

With respect to IDEs, CADSE share the concern of managing the activities inside a workspace, for integrating third-party tools and for providing as much help as possible to the software engineer in his/her daily tasks. Unfortunately, the IDEs’ high generality comes at the cost of not knowing anything on the nature and specificities of the software application to build.

In contrast, CADSE uses the system model and the other available models as knowledge about the application to build, about its structure and the operations that can be performed, or about how to perform build operations. This knowledge is used to make IDEs more “intelligent” and to extend the Eclipse capabilities, automate a number of activities such as complex build operations, or automatic class path maintenance. The concept of composite extends the Eclipse concept of workspace with composite projects, viewpoints and dynamic abstraction levels. Conversely, CADSE relies on the IDE capability for fine granularity actions such as editing, or compiling. CADSE and IDEs are in a synergistic relationship.

8. Conclusion

A major contribution of our work is showing that in software engineering a model (a structural system model) and a real object (e.g. a software application) can be so closely synchronized that the model can be seen as an operational high-level viewpoint of the software. It follows that an engineering environment can take advantage of this property, relying on the model in order to assist the software engineer during the different phases of an application’s life cycle. This work shows that a model can be fully operational. The operations defined on the model elements represent the interface governing the engineering environment behavior, and the work performed in workspaces is carried out on model elements.

In this paper we show that the concept of composite is very special because their nature and structure is not constrained by physical reality. This implies that many different intertwined composites can be defined over the same elements. For composites to play an abstraction and scalability role, advanced mechanisms are required (see the semantics of the composition relationship and the build, open and close operations).

Composites are simultaneously a structuring mechanism (relating a number of items as being part of something), an abstraction mechanism (hiding its content), a scalability mechanism (being potentially nested), an operational mechanisms (being “executable”), a point of view mechanism (containing only the information related to a purpose), and a concurrent engineering mechanism (associated to the workspace concept).

CADSE introduces a high-level operational model into software engineering practice in which the concept of operational composite provides the properties needed for scalability, as well as an essential mechanism for the support of high-level engineering activities. We believe that our system reconciles traditional SCM systems, IDEs, design environments, and software engineering tools, methods and environments. This result was possible to achieve thanks to a highly specialized composite concept, closely tailored for Software Engineering needs.
References


