# Towards Composing Software Components in Both Design and Deployment Phases

Kung-Kiu Lau, Ling Ling, and Perla Velasco Elizondo

School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
{kung-kiu,lling,pvelasco}@cs.man.ac.uk

**Abstract.** In component-based software development, the design of components should be carried out separately from the deployment of components, in order to enable composition by independent third-parties. However, current component models are biased towards either the design phase or the deployment phase. In this paper, we argue that ideally component models should include both design and deployment phases, and it should be possible to compose components in both phases. We also demonstrate a preliminary implementation of composition in both phases in a component model we have defined.

## 1 Introduction

In component-based software development (CBD), components should be produced and used by independent parties. That is, component developers need not be the same people as component customers such as system developers. This implies that the *design* of components is carried out separately from the *deployment* of components.

In current component models [6,10], components are either objects or architecture units. These models tend to be heavily biased towards either the design phase or the deployment phase. In architecture-based models[6,10] like ADLs and UML2.0, components are design entities by definition, with or without corresponding binary components in the deployment phase. On the other hand, in object-based models[6,10] like COM, .NET, CCM and Fractal, components are objects that are executable binaries, and are therefore more deployment phase entities than design phase entities.

In this paper, we argue that ideally component models should include both design and deployment phases, in order that CBD can meet its objective of building systems from pre-existing components with maximum reuse and minimum time-to-market. In particular, it should be possible to compose components in both design and deployment phases, in an idealised life cycle for components.

We motivate and define the idealised life cycle, based on commonly accepted desiderata for CBD. We discuss composition in each phase, and demonstrate a preliminary implementation of composition in both phases in a component model we have defined.

## 2 An Idealised Component Life Cycle

The life cycle of components [4] consists of three stages: (i) the *design phase*, when components are designed, defined and constructed in source code, and possibly compiled into binaries; (ii) the *deployment phase*, when binaries of components are deployed into the execution environment; and (iii) the *run-time* phase, when component binaries are instantiated and executed in the running system. Ideally, composition should be possible in both the design and the deployment phase while the system is being constructed. Composition means component reuse, and therefore composition in both phases will maximise it. It also means design flexibility in the sense that the deployed components, in particular composite components, can be designed, by composition in either phase.

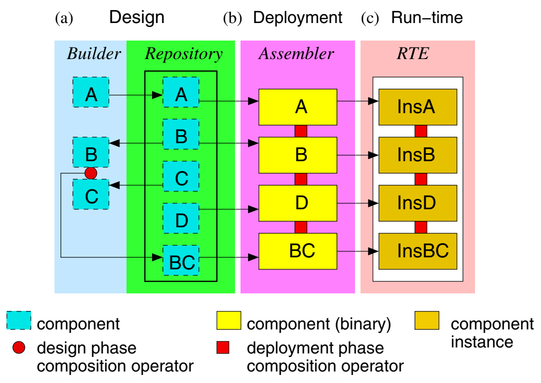

**Fig. 1.** An idealised component life cycle

Accordingly, we have defined an idealised component life cycle [11,10], and the kind of composition meaningful in its phases (Fig. 1). The idealised life is based on the following commonly accepted desiderata of CBD [2,5,14,12].Firstly, components should be pre-existing reusable software units, which developers can reuse for different applications. This necessitates the use of a repository in the design phase. Secondly, components should be produced and used by independent parties, i.e. component developers and system developers. This is important for ensuring that components are truly reusable by third parties and requires the use of proper tools that can interact with a repository, in the design and deployment phases. Thirdly, it should be possible to copy and instantiate components, so that their reuse can be maximised, both in terms of code reuse and in terms of components' scope of deployment. Thus, components should be distinguished from their instances, and therefore differentiate the design and deployment phases from the run-time phase. Fourthly, components should be composable into composite components which in turn can be composed with (composite) components into larger composites (or subsystems), and so on. This requires that composites can be deposited in and retrieved from a repository.

**Design Phase.** In the design phase, components have to be constructed, catalogued and stored in a *repository* in such a way that they can be retrieved later, as and when needed. Components in the repository are in *source code*, or they may have been compiled into *binary*.

Components here should be composed into well-defined *composites* using suitable *composition operators*, ideally supported by a composition theory. It should be also possible to store composites in, and retrieve them from the repository, and use them for further composition, like any components.

A *builder* tool can be used to (i) construct new components, and then deposit them in the repository, e.g. A in Fig. 1 (a); (ii) retrieve components from the repository, compose them and deposit them back in the repository, e.g. in Fig. 1 (a), B and C are composed into a composite BC that is deposited in the repository.

To promote its reuse, components in design phase should be *templates* that provide services. They should be normally identified and designed by *domain experts* as basic building blocks for the domain in question. They should be generic, rather than system-specific so that they should be (re)used to build many different applications. Similarly, composition operators in design phase should be generic *composition schemes* to coordinate components which can be customised for many different systems.

To support its reuse, a composite should expose a proper *interface*. This interface should be generated during the composition process and its content should be determined according to the semantics of the composition operator involved.

Components in design phase should also include information of the *environmental dependencies* or *resources* needed for its deployment. Composition in design phase should generate such information for composites. For instance, deployment contracts [8] could be used to specify this kind of information.

**Deployment Phase.** Ideally, composition in deployment phase should follow on from, and thus exploit composition in design phase. That is, as far as possible, the composites here should be built directly from the (composite) components created in design phase.

In the deployment phase, components have to be retrieved from the repository, and if necessary compiled to *binary* code and then composed. The result of deployment phase composition is a whole *system* in binary code, and so this is the end result of system design and implementation. The completed system should be then *ready for execution*.

As in design phase, composition should be carried out via *composition operators*. However, here they should be able to specify detailed coordination between components as required by the specific application.

An *assembler* tool can be used to retrieve components from a repository, compile them into binary code, and then assemble them into a system. For example, in Fig. 1 (b), binaries of A, B, D and BC are retrieved and composed into a system.

Composite components in the deployment phase should have *interfaces* that allow them to be instantiated and executed at run-time. These interfaces should be generated during the composition process.

Composition in deployment phase should be supported by suitable *deployment tools*, for example, for checking component compatibility with one another and with the execution environment, a tool for checking deployment contracts would be useful. Also with such tools, it should be possible to deploy a composite in many different systems, possibly with different execution environments.

**Run-time Phase.** In the run-time phase the constructed system is *instantiated* and *executed* in the run-time environment, e.g. A, B, D and BC in Fig. 1 (c). Although there is no further composition in this phase, it may be desirable to adapt component instances or composition operators so as to dynamically re-configure the executable system. We do not discuss this here, since our focus is on composition.

## 3 Towards Composition in Both Design and Deployment Phases

We have done some preliminary work to realise composition in both design and deployment phases. Our work is based on a component model we have defined [7].

In our component model, there are two basic entities: (i) *computation units* and (ii) *connectors*.[1] A computation unit performs only computation (by providing a set of methods) and does not invoke any computation outside itself. There are two kinds of connectors: (i) *invocation* connector, which is used to invoke a computation unit; and (ii) *composition* connector, which composes components.

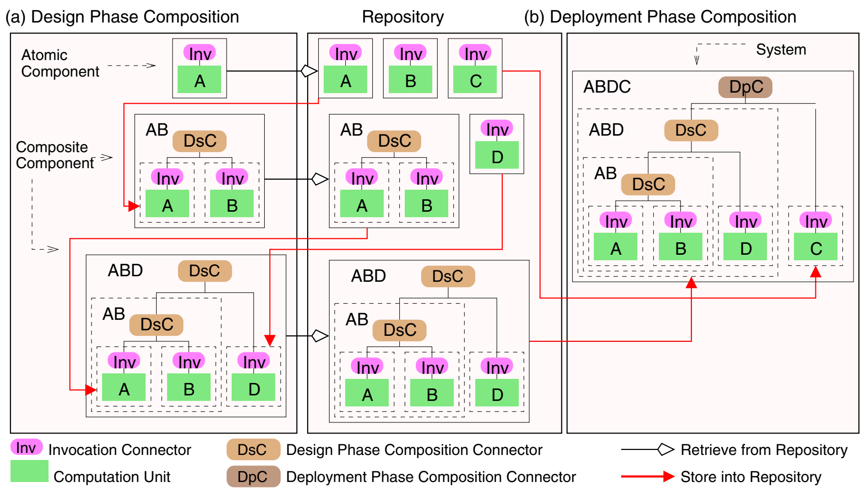

**Fig. 2.** (a) Design phase and (b) deployment phase composition in our component model

Components are defined in terms of computation units and connectors. There are two kinds of components: (i) an *atomic* component, which consists of a computation unit with an *invocation* connector (e.g. A in Fig. 2 (a)); and (ii) a *composite* component, which consists of a set of components (atomic or composite) composed by a composition connector (e.g. AB and ABD in Fig. 2 (a)).

In [9], we have introduced a basic set of composition connectors which encapsulate the three standard control structures: *sequencing*, *branching* and *looping*.

Composition connectors are defined by a *type hierarchy*, so that they allow hierarchical component composition. Every (composite) component has one top-level connector, which is either an *invocation* connector (for an atomic component) or a composition

[1] They are exogenous connectors [9].

connector (for a composite component). This connector represents the only access point to the component, and also its *interface* for further composition.

The semantics of components and composition operators in our component model is such that composition can take place in both the design and deployment phase. Fig. 2 illustrates this, in a direct comparison to the idealised life cycle.

In the design phase (Fig. 2 (a)), the composite AB is built from atomics A and B by the *design phase composition connector*, and in turn it can be further composed with atomic D by having its top-level connector connected by another composition connector, to build up the composite ABD which is deposited back into the repository.

In the deployment phase (Fig. 2 (b)), the composite ABD is retrieved form the repository and composed via a *deployment phase composition connector* with component D to yield system ABDC. If required, further composition can be done. At the end of the composition process, the final system should be ready to execute in the target execution environment.

## 3.1 Preliminary Implementation

In our preliminary work, we have implemented composition connectors in both design and deployment phases, but not full-blown tools for the builder, repository or assembler. Neither have we incorporated deployment contracts in the design phase, or implemented deployment tools for deployment phase. Our implementation is in Java, and we have also assumed a simple execution environment throughout, namely JVM.

**Design Phase.** A software component is implemented in source code by a set of classes (Fig. 3) in design phase. We define a type Component in a Java interface. For each component, there is a class that implements the Component interface, and it keeps a reference to a Connector type as the top connector. The super class Connector is extended by The $Invocation$ connector class and composition connector classes such as $Pipe$, $Sequencer$ which are used to construct atomic or composite components respectively.

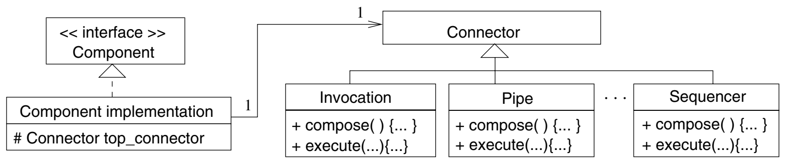

**Fig. 3.** Overall structure of a set of classes for constructing components

A builder tool in design phase is used to construct components. For an atomic component, the computation unit is a Java class that implements the services and does not call services outside itself. The builder tool specifies the computation unit name in the source code of $Invocation$ connector (`compose` method), and generates an atomic component class which refers to the $Invocation$. For a composite component, it is constructed by builder tool by specifying the top level connectors of the constituent components in the source code of the composition connector. Because according to the hierarchical composition, the connection point for the sub component is always its top level connector. The generated composite component class file refers to the top composition connector, which again serves as the connection point when this composite component is connected by a higher level connector, so as to create a bigger composite component.

Component interface specifies all the services provided by the component and desired data for instantiation. An atomic component interface is given by the component developer and presented in an XML format. The interface of a composite is generated by the composition connector automatically in terms of the interfaces of the constituents and the composition scheme.

The way to invoke a component is calling the top level connector (`execute` method) with the method name and parameters. Internally it calls the lower level connector recursively until it reaches a computation unit. One point worth noting is the components in this phase are templates, therefore their behaviour is not fixed with specific set of calling methods at this stage.

Currently, the component repository is a java file directory. In the next step, repository needs to be fully interacted with the builder tool and both of them need to be enhanced to support (atomic or composite) component deposit after construction automatically and multiple copies of component retrieval. Besides, deployment contracts specification needs to be integrated when the builder tool automates the construction of components.

**Deployment phase.** For deployment composition we have a set of classes which integrate our *composition framework* (Fig.4 (a)). Deployment phase connectors are implemented as a set of classes with the superclass DPConnector. Each subclass defines a *constructor* to instantiate it, and overrides the `execute` method to implement its corresponding logic. A class System defines a valid composition in this phase. According to our model, the class System holds a reference to deployment phase connector, which represents the interface and the only access point to it.

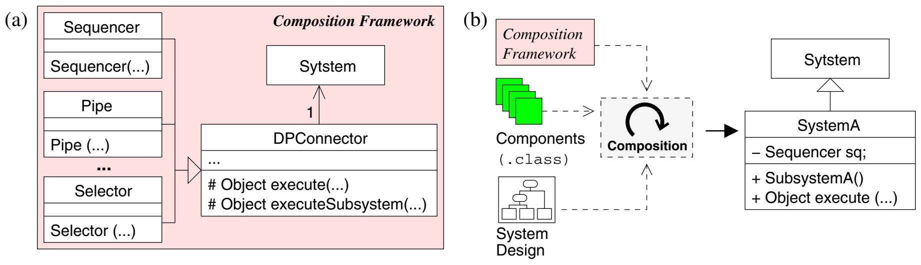

**Fig. 4.** (a) Our deployment phase composition framework and (b) its use to compose a System

To build a system, during the composition process we (re)use the binaries of the components generated in the design phase, our composition framework, as well as the design of the composition for the desired system (Fig.4 (b)). The result of the composition process is new class that extends System and declares a constructor containing the code for setting up the composition, and a `execute` for calling its top-level connector's to allow the system's execution, e.g. `sq.execute(...)` when a *Sequencer* connector `sq` is the top level connector of the SystemA shown in Fig.4 (b).

Making a system a new class allows to generate a binary that can be packaged as a named, versioned, shippable and deployable unit. The final system is meant to be deployed within a execution environment and eventually be executed on it.

In the current implementation, systems' interfaces are generated as a XML file containing very basic information such as the mechanisms to instantiate and execute it, but it can be extended to include more detailed information for its proper deployment.

A system instantiated via its constructor and executed by calling its `execute` method. The `execute` method contains a call to `executeSubsystem` method defined in the DP-Connector superclass for each one of the components and/or subsystems it connects. In the `executeSubsystem`, the hierarchical execution of each connected element is carried out until reach the $Invocation$ connectors of atomic components, where reflection techniques are used to dynamically execute the required operation in their computation units.

**An Example.** Consider a Drink Vending Machine System which serves different kinds of drinks, i.e. coffee, juice and tea. Besides the traditional paying mechanism, it accepts coinless dispensing of drinks to holders of drink cards. The architecture for this system is shown in Fig.5 and it includes the atomic components: ProductManager and Receipe-Manager –which deal with the drinks' prices and recipes; CoinBox, CardReader –which deal with paying for the drinks; and a set of Dispensers –which deal with the pouring of ingredients during the drink making.[2]

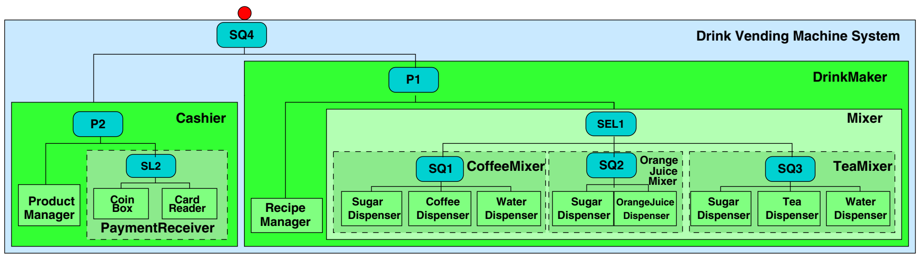

**Fig. 5.** Drink Vending Machine architecture

In the design phase, we build the composite Mixer by hierarchically composing the dispenser components as depicted in Fig.5. The composite Mixer can deal with the making of different drinks according to the top-level *Selector* condition's provided value, e.g. *product name* = "*coffee*". Due to the Mixer encapsulates functionality suitable for similar applications in the same domain, constructing it in the design phase and putting it in the repository facilitates its further reuse.

[2] Each one of these components is created at design phase by connecting a *Invocation* connector to the Java class representing the corresponding computation unit.

The Mixer can be compiled into a `.class` file, and reused at deployment-phase to create the final system by firstly composing it with the RecipeManager, and then with the Cashier subsystem –which has been composed from the CoinBox, CardReader and ProductManager atomic components.

The interface of the final system exposes the way it can be instantiated and executed. For buying a drink the *product name*, *type of payment*, *amount* or *card number* are required.

## 4 Discussion and Conclusion

The advantages of composition in both phases composition are not present in current component models. In most of these models, composition is carried out in the design phase only (e.g. architecture description languages (ADLs), UML 2.0, PECOS, Pin, Fractal, EJB, COM, CCM, Koala, SOFA and KobrA), leaving the deployment phase with the only task of implementing what is defined in design phase [10]. In JavaBeans and POJO [13], composition is carried out in the deployment phase only.

Our approach also allows design flexibility. Developers can choose either to build up composite components in design phase for reuse purpose, or assemble components with application specific configuration in the deployment phase. However, there is a balance between design and deployment phase composition. The former is carried out by the component builder guided by domain knowledge for constructing reusable building blocks; the latter is carried out by the system developer targeting particular applications with environmental settings.

As future work, we intend to implement builder, repository and assembler tools to automate the composition in both phases, as well as deployment tools. In addition, we will investigate reference semantics where a constituent component is used by different composite components, and the issue of further composition and deployment of the component in such a scenario. We will also consider adaptation and re-configuration at run-time, as in approaches such as Hadas [1] and Gravity [3].

## References

1. Ben-Shaul, I., Holder, O., Lavva, B.: Dynamic adaptation and deployment of distributed components in hadas. IEEE Trans. Softw. Eng. 27(9), 769–787 (2001)
2. Broy, M., Deimel, A., Henn, J., Koskimies, K., Plasil, F., Pomberger, G., Pree, W., Stal, M., Szyperski, C.: What characterizes a (software) component? Software - Concepts and Tools 19(1), 49–56 (1998)
3. Cervantes, H., Hall, R.S.: Autonomous adaptation to dynamic availability using a service-oriented component model. In: Proc. ICSE04, pp. 614–623. IEEE Computer Society Press, Los Alamitos (2004)
4. Christiansson, B., Jakobsson, L., Crnkovic, I.: CBD process. In: Crnkovic, I., Larsson, M. (eds.) Building Reliable Component-Based Software Systems, pp. 89–113. Artech House (2002)
5. Heineman, G.T., Councill, W.T.: Component-based software engineering: putting the pieces together. Addison-Wesley, Reading (2001)

6. Lau, K.-K.: Software component models. In: Proc. ICSE '06, pp. 1081–1082. ACM Press, New York (2006)
7. Lau, K.-K., Ornaghi, M., Wang, Z.: A software component model and its preliminary formalisation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 1–21. Springer, Heidelberg (2006)
8. Lau, K.-K., Ukis, V.: Defining and checking deployment contracts for software components. In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 1–16. Springer, Heidelberg (2006)
9. Lau, K.-K., Velasco Elizondo, P., Wang, Z.: Exogenous connectors for software components. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 90–106. Springer, Heidelberg (2005)
10. Lau, K.-K., Wang, Z.: A survey of software component models. 2nd edn., Pre-print CSPP-38, School of Computer Science, The University of Manchester (May 2006) `http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp38.pdf`
11. Lau, K.-K., Wang, Z.: A taxonomy of software component models. In: Crnkovic, I., Larsson, M. (eds.) Proc. of 31st Euromicro Conference, pp. 88–95. IEEE Computer Society Press, Los Alamitos (2005)
12. Meyer, B.: The grand challenge of trusted components. In: Proc. ICSE03, pp. 660–667. IEEE Computer Society Press, Los Alamitos (2003)
13. Richardson, C.: POJOs in Action: Developing Enterprise Applications with Lightweight Frameworks. Manning Publications Co., Greenwich, CT (2006)
14. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Reading (2002)