Extended Abstract of Work in Progress

Dynamic Components as Semantic Entities: Concept and Static Support

Arnd Poetzsch-Heffter

FernUniversität Hagen, D-58084 Hagen, Germany
poetzsch@fernuni-hagen.de

1 Introduction

The notion of software components is widely used. Usually, a component is considered as a static program entity. Here are two examples to illustrate this position: Booch in [Boo87]: "A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction". Szyperski in [Szy97]: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". Essentially, a component is part of the code constituting a software system that has certain compositionality properties.

In this extended abstract, we look at components from a dynamic point of view. The idea is to conceptually structure the storage of a running program into a hierarchically organized set of encapsulated components with well-defined interfaces. The extended abstract reports on work in progress. It is organized as follows. In section 2, we informally define dynamic conmponents and illustrate the definition by an example. Then, we introduce the invariant expressing the component hierarchy. We discuss the benefits of a component structure for program specification, verification, and reuse. Section 3 summarizes related work.

2 The Concept of Dynamic Components

The concept of dynamic components can be introduced independent of the underlying programming paradigm. However, we will assume a simple object-oriented setting here to keep things focussed. In this section, we first describe the used setting and define dynamic components. Then, we look at the motivations and goals for investigating such structuring mechanisms and the relation to programming language constructs.

2.1 What is a Dynamic Component?

An *object* is an instance of a class that has an identity, a type, a set of instance variables representing the local state of the object, and a set of methods. The methods of an object X can change the state of X, can create new objects, and can invoke methods on other objects. Methods are only allowed to modify the instance variables of the self-object. Like in many current object-oriented languages (in particular Java), objects *cannot* be composed of other objects. Objects are created at runtime. At any execution point, the set of the created objects and their linkage is captured by the so-called *object store*.

Like objects, dynamic components are runtime entities. A dynamic component D partitions the objects of a given object store into three sets: the internal objects $intn(D)$, the boundary objects $bndr(D)$, and the external objects $extn(D)$. We say that D consists of the objects $objs(D) =_{def} intn(D) \cup bndr(D)$. $bndr(D)$ always contains at least one object the so-called *representative* of D ($rep(D)$). As example for a dynamic component, we consider the objects implementing a doubly linked list given in figure 1. The grey oval indicates the boundary of the list component. The two objects on the boundary are of type `LinkedList` and `ListIterator`. The `LinkedList`-object is the representative of the component (indicated by the filling). The internal objects of type `Entry` realize the list structure. The list elements of type `Object` are external to the component.

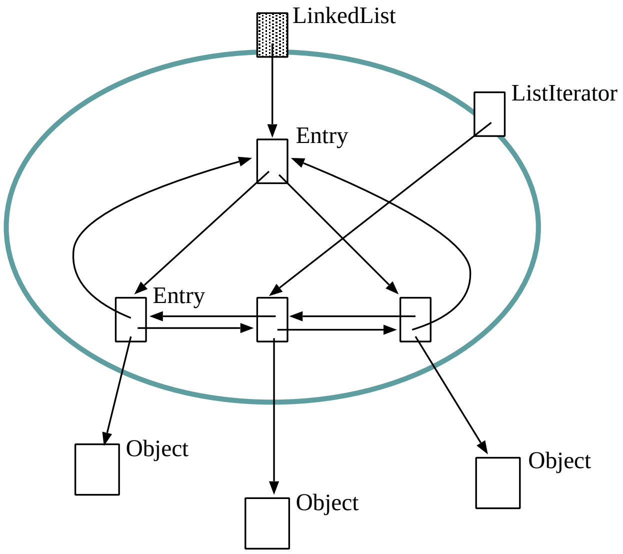

Fig. 1. Linkage structure of a doubly linked list component.

A dynamic component D has an identity given by its representative. It has a local state corresponding to the local state of its objects. We say a dynamic component D_1 is part of another dynamic component D_2 if $objs(D_1) \subset intn(D_2)$. In a given object store, a dynamic component D will be denoted as a triple $(rep(D), bndr(D), intn(D))$. An object store is called *structured* (into dynamic components) if there is an injective mapping δ from the set of objects to dynamic components such that:

1. Objects are mapped to the dynamic components of which they are representatives: For all objects X there are object sets BD and IT such that $\delta(X) = (X, BD, IT)$.
2. The dynamic components form a hierarchy: Let X_1 and X_2 be different objects and $D_i =_{def} \delta(X_i)$; then either $D_1 \cap D_2 \neq \emptyset$ or D_1 is part of D_2 or D_2 is part of D_1.
3. Internal objects are encapsulated and only accessable through the objects on the boundary: If D is a dynamic component, $XI \in intn(D)$, and $XE \in extn(D)$, then every reference path from XE to XI passes through an object in $bndr(D)$.

These properties are called the *structure invariant.* Notice that every object store can be trivially structured by mapping each object X to $\delta(X) = (X, \{ X \}, \emptyset)$.

2.2 Why Should We Investigate Dynamic Components?

Essentially, there are two reasons why we are interested in dynamic components:

- A hierarchically structured object store simplifies program understanding, testing, specification, and verification: What has been done to control structures in the sixtees, namely replacing gotos by structured programming, has to be done to object structures in the next years. Completely general link structures are rarely needed. Thus, OO-programs should maintain helpful invariants on their object stores.
- A better understanding of the dynamic component structure of programs can help to clarify the notion of static program components.

In addition to these general reasons, there are more technical advantages of a programming model based on dynamic components: Dynamic components enable to support specialization for collection of classes not only for one class. They provide units for synchronization in a parallel processing setting. They can be used as entities for distribution.

When introducing a new concept, it is interesting to relate it to other language concepts. In several aspects, dynamic components are similar to objects: They have an identity; they are runtime entities with local state; they reference each other (via the references of their objects). The difference is that they can be built from other dynamic components and the interface/boundary of a dynamic component may consist of more than one object. It should be stressed here that objects cannot be built from other objects in the sense of dynamic components. Even languages supporting part objects (like e.g. Beta) do not enable a variable number of part objects and do not maintain the structure invariant described above.

Dynamic components can be realized by current modularization and encapsulation constructs. However, these constructs do not enforce the structure invariant. Thus, the situation is similar to writing typed programs in an untyped programming language. Consider the following simple class `MyString` written in Java that stores the characters of the string in an array object:

```
public class  MyString {
  private size;
  private char[] content;

  public MyString( char[] ar ){
    content = ar;
  }
  ... // methods of class MyString
}
```

The intention of this implementation might have been that class `MyString` encapsulates the char-array so that no one from the outside can modify it. In other words, the intention might have been that `MyString` realizes dynamic components of the following form:

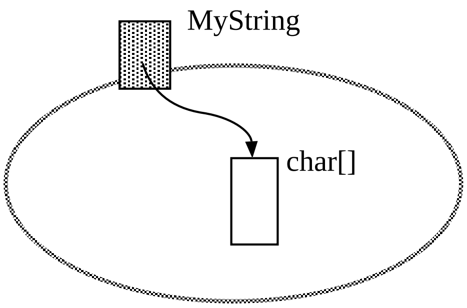

Fig. 2. MyString as dynamic component.

However, the intention is not *declared* in the class definition above. Thus, no tool can detect that the given constructor breaks the encapsulation: It “imports” an character array to the dynamic component. The caller of the constructor may keep the reference to the array, and modify the array without using the methods of class `MyString`. This is not prevented by declaring the array to be private to the `MyString`-object. Thus, language constructs for the declaration of dynamic components can make structuring information explicit that is very helpful for static program checking.

3 Related Work and Conclusions

In this section we sketch related work and provide some concluding remarks.

Related Work Many approaches to control the linkage structure of the object store are based on typing mechanisms. E.g. balloon types enforce a strictly hierarchical structuring similar to dynamic components where the so-called balloons correspond to dynamic components ([Alm97]). Balloons always have exactly one boundary object and do not support references leaving balloons (cf. fig. 1 for outgoing references). Data flow techniques are used for checking the constraints imposed by the balloon types. The type system in [CPN98] supports a structure invariant that allows for outgoing references. A parameterization technique keeps the approach sufficiently flexible to maintain reusability of classes. On the other hand, the type system does not support inheritance and is too restrictive to handle several boundary objects that are e.g. needed to implement list iterators (cf. fig. 1).

The universe type system presented in [MPH00b] does without parameterization. It gains its flexibility by supporting read-only references going *into* dynamic components. This way, several boundary objects can be simulated, and programming patterns can be realized that refer to more than one dynamic component. The disadvantage of this type system is that dynamic type casts become necessary in certain situations. The use of the universe type system for modular specification and verification of programs is described in [MPH00a].

A different goal for the structuring of object stores is followed in [GTZ98]. In this paper, additional type information is used to improve the storage locality of class-based programs. One reason to do so is the optimization of garbage collection. In contrast to the approaches described above, the additional information is not provided by the programmer, but tried to be extracted from the program by data flow techniques. This makes the approach applicable without programming support, but restricts the patterns in which helpful structure information can be deduced.

Conclusions and Future Work We defined the concept of dynamic components in a way that is independent of techniques to declare dynamic components in programs. Dynamic components enforce the structuring of the runtime store. We sketched some motivations for investigating this concept and summarized related work. Just as normal type information, programs can be enhanced by declaration information expressing the boundaries of dynamic components. As in the cited papers, this information can be incorporated into extended type systems. We are currently further inverstigating and refining the type system presented in [MPH00b]. As a different line of research, we try to develop declaration constructs for dynamic components that are implementation independent. Such declarations can be added as annotations to a program without changing its types. A checker can proof that the program satisfies these annotations.

References

[Alm97] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In M. Akşit and S. Matsuoka, editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, 1997.

[Boo87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin-Cummings, 1987.

[CPN98] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.

[GTZ98] Daniela Genius, Martin Trapp, and Wolf Zimmermann. An approach to improve locality using sandwich types. In *Proceedings of the 2nd Types in Compilation Workshop*, Lecture Notes in Computer Science 1473, pages 194–214. Springer-Verlag, 1998.

[MPH00a] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[MPH00b] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programmiersprachen und Grundlagen der Programmierung, Kolloquiumsband '99*, Informatik Berichte 263–1. Fernuniversität Hagen, 2000.

[Szy97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.