The software engineering discipline faces many challenges; one of the important challenges is managing the complexity of software [5]. We are building increasingly large and complex software systems, enabled by steady improvements in software engineering. Such systems encompass a substantial amount of inherent complexity; partially inherent in the problem domain, and partially in the solution domain. But some of the complexity can also be avoided. A key argument, as coined e.g. by Brooks [2] and adopted here, is that the limited ability of realization models to accurately represent the concepts and their interdependencies in a conceptual solution is the main cause of accidental complexity. As a result when we implement the conceptual solution in a programming language supporting specific techniques, the complexity of our realizations is typically substantially larger than the complexity of the conceptual solution.

Using a different language that supports more suitable implementation techniques leads to reduced complexity. Here, abstraction is considered as the key implementation technique [3]; this means decomposing software into smaller units to manage complexity and delivering working systems by expressing proper compositions of these units. The figure below illustrates composition by means of abstraction: The boxes visualize modules; the dots correspond to the usage of abstractions; and the arrows represent compositions of modules and abstractions. Different composition mechanisms, illustrated by different line styles (dotted, dashed or solid), provide different means to resolve the target of a composition at runtime, and possibly how to transfer control from one module to another. Thus, composition mechanisms define the semantics of the arrows.

The relevance of abstraction and advanced composition mechanisms can be seen in the continuous progress in the history of programming language research [6]. The fact that new such mechanisms are developed frequently shows the need for them, but it also shows that a fixed set of composition mechanisms will always be limiting. Our message is that there is substantial benefit in offering languages where abstractions and composition techniques are not hard-wired; rather they should be easy to introduce on demand by software engineers as new types of compositions are identified.

We have developed a prototype language, Co-op [4], that allows normal application objects to act as composition operators by means of message rewriting, to show that realizing these ideas is feasible. Since message passing between objects is able to fully cover the objects’ behavior in object-oriented languages, message rewriting is a powerful mechanism to arbitrarily control the composition of behavior. But also other implementations are conceivable. As part of future work, we will investigate more explicit interfaces (e.g. [7]), and type checking (e.g. [1]).

1. REFERENCES