ECaesarJ Programming Language: Motivation and Concepts

Motivation
The structure of variation in a product line is an important criterion for decomposing its implementation into modules. Such a modularisation facilitates independent evolution of the products of a software product line (SPL), as well as the introduction of new variations required by new products, because implementations of the specific features of such products are separated. Ideally, a change in the common code base of a product line would affect only the products that really use the changed functionality. This does not mean, however, that there must be a one-to-one correspondence between the features as they appear in the feature model describing variations in the product line and the modules of its implementation. With respect to the implementation level, it is more useful to talk about implementation features as cohesive units of functionality, the cohesion being determined by their correspondence to the user features, describing functionality and variations of the product line from the perspective of its users.

Structural Decomposition
Since features are units of functionality they are often not aligned to the structure of software objects and operations. On the one hand, the state of an object may include information necessary to different features; an operation can include behaviour that contributes to different features. On the other hand, multiple objects and operations can be involved in the implementation of a feature.

Thus, if we want to modularise features, we should be able to split definitions of objects and operations and distribute them into multiple modules. The modules in conventional programming languages are not sufficiently extensible: they do not provide mechanisms to extend definitions of existing classes and functions. An individual class can be extended by means of inheritance, but this leads to creating a new class rather than to refining the existing class. Indeed, the subclass does not automatically replace its superclass class in its relationships with other classes.

Implicit Invocation
Modularisation of features also requires modularisation of the dynamic structure of the program, i.e. the structure of its control flow. For modularisation reasons, a feature may need to be called within the control flow of another feature, but the latter logically independent of the former. A complete independence can be achieved by using some form of implicit invocation, i.e. source components can declare events and destination components can manifest their interest in them.

An object-oriented solution for implicit invocation is provided by the Observer design pattern. There are several problems with the Observer design pattern, though. It requires a significant amount of glue code for registration and notification of observers. Moreover, the need for observation of particular events must be preplanned, or invasive changes are required afterwards in order to support this observation. Since the observer infrastructure is a part of the design of the observed objects, client-specific observation protocols are not directly supported and thus either complicate implementation of observers or cause additional complexity of the notification code.

State-Dependent Behaviour
State machines are widely used for software design, but are not appropriately supported by mainstream object-oriented languages. This leads to various problems with variation of state-dependent behaviour.
With conventional inheritance the only extensible abstractions are methods, and the behaviour of an object can only be extended by overriding methods. Since object states, state transitions, and dependencies of behaviour on states are not represented by explicit abstractions, it is difficult to extend software with new states and their corresponding state-dependent behaviour, and to ensure consistency and stability of such extensions.

**ECaesarJ Concepts**

ECaesarJ is an aspect-oriented programming language which integrates the features of its predecessor languages CaesarJ and EAOP and further improves them for a better support of modularisation of variation in SPL. The following text gives an overview of the major language features of ECaesarJ and explains how they address the stated problems.

**Virtual Classes**

ECaesarJ improves modularisation of SPL features by providing large-scale extension and composition mechanisms. These mechanisms are based on the idea of defining type-safe extensions of groups of classes, declared as so-called *virtual classes*. Virtual classes are inner classes that can be refined in the subclasses of the enclosing class, also known as a *family class*. In their refinements we can add new methods, fields and inheritance relationships as well as override the inherited methods. In each family class all references to a virtual class are always bound to its most specific refinement. As a result, virtual classes provide a large-scale extension mechanism, which supports a consistent extension of a group of classes.

The possibility of consistent extension of groups of classes can be exploited for defining features as extensions of other features. This can be achieved by modelling a feature as a family class and domain objects as its virtual classes. In such a design, since virtual classes can be refined in subclasses, a feature can extend the functionality of another feature by inheriting from it and selectively refining its classes.

For composition of features, ECaesarJ supports *mixin composition*, which is a form of multiple inheritance, based on linearization of the inheritance graph. ECaesarJ implements a *propagating* form of mixin composition, which means that the composition of family classes propagates into virtual classes: all inherited declarations of virtual classes with the same name are composed by mixin composition.

*Abstract family classes* can be used for description of interfaces between features. In order to distinguish between the implementation of an interface and its use, the interfaces used by a class are listed in a special inheritance clause, starting with the keyword *requires*, and are considered as required interfaces of that class.

**Events**

ECaesarJ supports implicit invocation by introducing *events* as explicit behavioural abstractions in the programming language. In a general sense an event denotes something of interest happening in time. An event has a source (or a subject) and destinations (or observers) interested in the event. From the point of view of the source, events can be *explicit* or *implicit*. Explicit events are explicitly triggered by the source and serve as notifications to the destinations that need to react to a change in the source. Besides explicitly triggered events, we can consider all identifiable changes of program state as implicit events. Examples of such events are method calls or value changes of object attributes.

ECaesarJ supports explicit and implicit events. Abstract events can be used in interfaces between features. An event can be defined by triggering or declaratively as an expression over other events, which can involve
various forms of quantification. Event definitions themselves can be distributed into multiple features and composed using mixin-composition semantics.

**State Machines**

ECaesarJ extends objects with a new event handling mechanism: a state machine, which makes it possible to organize event handling by the logical states of the object. By making state machines explicit in a programming language we not only enable expressing such interactive behaviour in a more natural and concise way, but also give an explicit structure to it, which, like in case of events, improves flexibility and stability of feature-based decomposition of the behaviour.

In ECaesarJ machines can be extended using inheritance. Subclasses can introduce new events and states, as well as refine existing states by adding new transitions or by overriding the existing ones. A subclass implicitly inherits all events and states, as well as the initial state declaration. A state can be refined by redeclaring it in the subclass. The refined state implicitly inherits all event handlers of its predecessors, except the ones that are redefined. This section shows extension of state machines.