Problems, Subproblems and Concerns

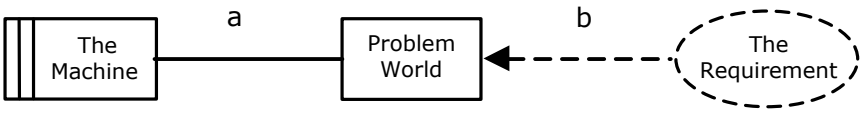
Position Paper submitted to
Early Aspects 2004

Michael Jackson

Abstract. Inevitably, aspect-oriented programming has focused on solutions; 'early aspects' aims to focus on problems. This position paper sketches how problems may be understood from a perspective based on problem frames. Problem analysis from this perspective reveals structural issues in a clearer light. It leads to a need for composition, both in the problem world and in the solution world. The goals of aspect technology would be clarified by such analysis, and the aspect technology may in turn offer some power in understanding and implementing the compositions.

1. The Machine and the Problem World

The goal of a software development is to produce a system to satisfy some r*equirement*. The system may be regarded as having two fundamentally distinct parts. The *machine* consists of the software to be built, executed by one or more general-purpose computers. The *problem world* is where the problem is located: it is here that the machine's behaviour will take effect and the success of the system must be evaluated. Essentially, the problem world is given[1]; the machine is what we must construct. This view [Jac00] can be represented in a simple diagram:



The interface *a* consists of a set of shared phenomena (for example, events and states) by which the machine can detect the behaviour and state of the problem world and can affect the problem world through its control of some of those shared phenomena. The requirement is a condition on the behaviour and states of the problem world whose satisfaction the machine must ensure. The requirement is expressed in terms of a set of phenomena *b*. In general these phenomena *b* are distinct from the phenomena of the interface *a*: the requirement, therefore, can not be expressed solely in terms of the behaviour and properties of the machine. It follows that satisfaction of the requirement depends not only on the specification of the machine's behaviour at *a*, but also on the causal properties of the problem world by which the machine's behaviour at *a* can be guaranteed to produce the required effects in terms of the problem world phenomena *b*.

The early development stages[2] must therefore include explicit articulation of the requirement, detailed specification of the appropriate machine behaviour at interface *a*, and identification and description of those properties of the problem world on which the specified machine will rely to ensure satisfaction of the requirement.

[1] That is, the causal and structural properties of the problem world are given, and some of its behavioural properties. It is usually the purpose of the machine to alter the problem world's behaviour within the given constraints.

[2] That is, the *conceptually* early stages. No position is being taken here on the *temporal* ordering of development activities.

2. Problem Decomposition

Since any realistic system will be too large and complex to handle as a whole, developers need ways of structuring that will allow them to master its size and complexity. Three interrelated structures are immediately identifiable:

- The requirement may be decomposed into *subrequirements*. For example, the requirement "provide lift service" may be initially decomposed into "service user requests" and "maintain safe operation".
- The problem world may be decomposed into *domains*. For example, the problem world of the lift control system may be decomposed into "users", "buttons", "lift mechanism" and "doors".
- The machine may be structured into *submachines*. For example, there may be one submachine devoted to scheduling lift service, another devoted to detecting malfunctions of the lift winding gear, and a third devoted to emergency action.

These decompositions may be understood as facets of the decomposition of the whole problem into *subproblems*. A subproblem is itself a problem, in the sense that it has a machine, a problem world and a requirement. The problem world for a subproblem is some projection of the problem world of the whole problem. For example, the problem world for the "maintain safe operation" suproblem may have only the "lift mechanism" and "doors" domains, and only certain phenomena and properties of those domains: the "users" and "buttons" may be irrelevant.

3. Subproblem Concerns

A central advantage of this kind of decomposition is that subproblems can be restricted to known problem classes, each characterised by a *problem frame*. Object patterns [Gam94] capture classes of design solutions; problem frames may be thought of as patterns in the problem space. Because each subproblem is of a familiar and documented class, the *concerns* it raises can become, over time, fully documented and well known to all competent developers. These concerns may be associated with particular types of problem domain, of requirement, or of relationship between the machine and the problem domains.

When the concerns associated with a problem are well known and documented, development can become more reliable. Many of the failures described in the literature on risks in computer-based systems—for example, in the work of Neumann[Neu95]—stem from neglect of concerns that should have been immediately recognised. Two examples of such concerns are:

- Initialisation. When software execution begins it is—typically—necessary to ensure that the machine and the problem world are in corresponding initial states. A failure of this kind led to friendly fire deaths in the Afghanistan war [Was02][3].
- Identities. When a problem domain contains multiple instances of an entity type, care is needed to ensure that the machine interacts on each occasion with the appropriate instance. A failure of this kind either led to, or contributed to, at least one air crash in which many people died [Neu95 pp44-45]. Failure to address an identities concern is so common in certain engineering fields that its symptom—'cross-wiring'—has an established name.

By structuring in terms of subproblems of known types, with known concerns, the developer can more readily apply the corpus of existing knowledge to their analysis and solution.

[3] I owe this illustration to Steve Ferg.

4. Subproblem Composition

Decomposition alone is never enough: it is always necessary to recombine the decomposed parts. At first sight it may seem that subproblem requirements can be combined by logical conjunction, and subproblem machines by concurrent execution: subproblem domain projections need no composition because they are projections of an already composed physical reality, and software implementation demands no more than a mechanism for appropriate distribution of shared events.

However, this optimistic view is far too simple. Wherever two subproblems have problem domain phenomena in common there is a potential interaction that must be appropriately handled in the composition. Three examples of these *composition concerns* are:

- Interference. This is a well-known concern. Wherever there are two machines, one controlling phenomena in a common domain and another monitoring the domain state—that is, a *writer* and a *reader*—some form of mutual exclusion may be necessary.
- Interleaving. Consider a one-way traffic light system in which the schedule of light phases can be edited at a console by a privileged operator: the two subproblems, editing and using the schedule, must be interleaved. In addition to avoiding interference it is also necessary to ensure that the changeover between two valid schedules does not give rise to an invalid sequence—for example, one in which a green light for northbound traffic is followed immediately by a green light for southbound traffic.
- Conflict. Two subproblems may require contradictory effects in the domain. For example, the requirements "service user requests" and "maintain safe operation" may be in conflict when an equipment fault is detected. The requirement to service user requests demands that the motor be turned on in order to move the lift car in response to a request; but the requirement to maintain safe operation may demand that the motor be switched off and the emergency brake applied to prevent further movement of the lift car.

The composition task, then, may demand introduction of an appropriate communication mechanism, or the choice and enforcement of requirement precedence to resolve conflict, or even the recognition and analysis of the composition itself as an additional subproblem.

5. Why Defer Composition?

At first sight, the composition task may seem to be a self-inflicted wound. Why adopt an analysis and decomposition technique that gives rise to an apparently gratuitous additional difficulty? The answer, of course, is that the difficulty of composition is neither gratuitous nor additional. The difficulty is always there: the issue is simply whether the composition concerns should be dealt with earlier or later—before or after the subproblems have been identified and analysed. Here we advocate that composition should be deferred until the subproblems to be composed have been thoroughly understood.

Premature composition is the source of much unnecessary complexity in software development, because it forces the treatment of each subproblem—whether separately recognised or not—to address both the subproblem's own concerns and the composition concerns in which it participates. The developer's view of the subproblem is polluted and confused by the composition concerns, and in this way important specific subproblem concerns can more easily be left unrecognised and neglected.

Deferring composition, by contrast, reveals the subproblems in their pure forms, in which they correspond more exactly to the patterns defined in their problem frames. Documented relevant knowledge of the concerns associated with each class of subproblem can therefore be more reliably applied. Deferring composition also allows the composition

concerns themselves to be addressed in a context in which the composition task itself is well defined because the components to be composed—the subproblem requirements, machines, and problem domains—have been adequately captured and analysed.

The argument underlying this approach has a clear relevance to the consideration of aspect orientation in general and of early aspects in particular. It is no distortion to say that a fundamental motivation of aspect orientation at any stage of development is to separate consideration of the parts from consideration of how they should be, and can be, composed. This motivation is given particularly clear and salient expression by the use of problem frames for requirements and problem analysis.

References

[Gam94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides; *Design Patterns: Elements of Object-Oriented Software*; Addison-Wesley, 1994.

[Jac00] Michael Jackson; *Problem Frames: Analysing and Structuring Software Development Problems;* Addison-Wesley, 2000.

[Neu95] Peter G Neumann; *Computer-Related Risks*; Addison-Wesley 1995.

[Was02] Staff writer Vernon Loeb; 'Friendly Fire' Deaths Traced to Dead Battery: Taliban Targeted, but U.S. Forces Killed; Washington Post, Page A21, March 24, 2002.