Software Evolution based on Formalized Abstraction Hierarchy

Timo Aaltonen
Tampere University of Technology
P.O.Box 553
FIN-33101 Tampere, Finland
Timo.Aaltonen@tut.fi

Tommi Mikkonen
Nokia Mobile Phones
P.O.Box 68
FIN-33721 Tampere, Finland
Tommi.Mikkonen@nokia.com

Abstract

Software evolution is about visions and abstractions. The success in finding the right visions, i.e., directions of future evolution, and abstractions, i.e., concepts by which the system is understood, provide a good starting point for the evolution of a software system. In contrast, a failure makes the system practically unevolvable. Unfortunately, there is no universally accepted set of visions or abstractions that could be applied in all systems. Instead, it is up to the developers to find and document them in particular domains. Then, criteria are needed for determining the quality of interconnected abstractions and visions. This can be achieved by modeling the abstractions incorporated in the system as a hierarchy, where abstraction levels exceeding that of implementation facilities are used. The hierarchy can then be used for examining new visions and requirements that emerge over time as well as for supporting associated modifications. This paper introduces an approach where formalism is used for deriving the hierarchy, and provides an example on the evolution of abstractions.

1. Introduction

While software evolution can be considered as a force of nature, its sophisticated management is a necessity for the maintenance of complex systems. In engineering, this leads to a situation where discussions can be raised, to what extent systems built in an evolutionary fashion are different from systems developed from scratch. The answer to this question, however, lies outside the scope of actual systems themselves. Instead, we need to look at models of the systems, the abstractions needed for comprehending and developing them, and visions and expectations that we have on their future.

Based on artifacts without direct physical qualities, software as such is immensely flexible by its nature. Therefore, it is possible to define any system using any kind of partitioning into modules[1]. However, as pointed out in a classical paper by Parnas[16], some modules are more favorable than others. The design of modules, their enforcement in implementations, and the underlying rationale for selecting the modules provide a basis for arguing about qualities of different implementations. This is emphasized in practices like software architecture reviews [3], where the views of different stakeholders form the basis of evaluation. In creating the views, it is possible to use different models to highlight the important aspects, like modifiability of some parts of the system or implementability of some future visions.

In the technical sense, there is practically only one way to emphasize anything in software: To design a module dedicated to that particular issue. This, however, is not always an optimal solution, as components are also subjected to other concerns. They should also be units of compilation, reflect available effective implementation technologies, and, due to the introduction of recent practices, like design patterns [6], reuse acknowledged good design decisions. In addition to the above technical hinges, human capabilities also play a big role in decomposition. People need to be assigned the responsibility of the development and maintenance of components. Tackling all the above issues simultaneously with one architecture requires delicate architecting and excellent engineering at best, and is impossible at worst.

The biggest enemy of software evolution is increasing complexity. In coping with complexity, the most effective weapon is abstraction. In an ideal world, abstractions would always be incorporated in individual modules, and, furthermore, obey available implementation interfaces. In reality, however, abstractions related to conceptual properties often extend from one module to another. This is evident in design patterns [6], and in the use of centralized state of a distributed system [2], for instance.

The clarity of concepts in an implementation architecture also enables the determination of whether a modification af-

[1]For the purposes of this paper, we will treat packages, components, processes, etc. as modules without any exact definition.

fects the system in a fundamental fashion, or is a minor update that leads to minimal redesign. Therefore, mastering and maintaining the abstractions and their relations to the modules of the system is a key issue for preserving the clarity of concepts. This often means denial of the temptation to extend abstractions or related software modules with some application-specific additions. The temptation is increased by the fact that the actual part where the change belongs may not be clear in a legacy system. Then, it is easiest to implement the new function in the scope of familiar code instead of studying all possible alternatives. Furthermore, in the short run, a straightforward implementation can be much more effective than a laborious process of identifying all the alternatives and selecting the most justified one. However, giving in to the temptation leads to difficulties in the long run.

Conventional software engineering approaches provide little support for determining whether a change is a minor one, or such that major reengineering of key abstractions is required. Based on the above discussion, such suppport is a necessity for introducing a robust framework for software evolution. The rest of this paper addresses this issue as follows. Section 2 discusses the notion of an abstraction hierarchy, which aims at relating the different abstractions needed for comprehending the system. The section also introduces the notion of an abstraction hierarchy that can be used for evaluating different design decision. Section 3 discusses maintenance based on abstraction hierarchy. Section 4 provides a discussion on abstractions incorporated in a mobile switch and their relation to actual implementation. Finally, Section 5 concludes the paper.

2. Towards an Abstraction Hierarchy

Software engineering abstractions are two-fold. Some of the abstractions are such that they directly reflect available implementation facilities, whereas some others exceed limitations of direct implementation concerns. We will refer to these categories as *primitive* and *non-primitive* abstractions, respectively.

Primitive abstractions are straightforward to describe. They are what we think about when considering software. They represent conventional components or software modules that can be compiled into executables with available tools, or run with interpreters or virtual machines. However, straightforward use of primitive abstractions has been found to harden rather than simplify rigorous reasoning [13]. Therefore, while needed for effective implementations, the role of primitive abstractions is not to ease reasoning about the system as a whole.

Non-primitive abstractions, in contrast, are difficult to describe in terms of conventionally used software artifacts. They represent cross-cutting concerns that cannot be located in one module [9], provide a design step that has been acknowledged as universally favorable [6], or model collective state distributed in multiple implementation components [8], for instance. The special role of such abstractions has also been pointed out in [4], where patterns are advocated as something that extend over objects and tie them together. As these examples make obvious, there are several levels of non-primitive abstractions already in the approaches that are already available. For instance, aspect-oriented programming relies on implementation level sequences of program code, whereas design patterns are intended to be used as design guidelines.

Based on the above discussion, completed systems potentially include several levels of non-primitive abstractions. Therefore, formalizations of such systems require semantically sound and practically manageable representation of collaborative properties [11]. The DisCo method [7, 5] enables addressing of such abstractions without being tied to individual implementation techniques. DisCo is a formal method, whose semantics are in the temporal logic of actions [12], a state-based formalism. In addition to well-defined semantics, the DisCo method introduces stepwise specification capabilities as a methodological guideline. Each step forms a *layer* in the complete system, where state variables as well as actions modifying the values of the layer's variables are given. For the purposes of this paper, a simple layer can be given, for instance, as follows:

```
layer L = {
  class C = {b : boolean};
  action A(c1, c2: C): c1.b ≠ c2.b →
                       c1.b′ = c2.b ∧
                       c2.b′ = c1.b;
} -- layer L
```

Layer `L` introduces class `C`, which has one attribute `b` of type `boolean`. Moreover, the layer has one action: `A`, in which two objects of the class `C` can participate. The action can be executed for such objects, which have different values in their attributes `b`. In the body of the action the participating objects swap their values of attribute `b`.

Layers can also refer to contents of other layers by importing them. The following example depicts this:

```
layer LL = {
  import L;
  class C = L.C + {i : integer};
  invariant I = ∀ c: C :: ∃ i: integer :: i < c.i;
  action A(c1,c2:C) refines L.A(c1, c2) →
                            c1.i′ = c2.i ∧
                            c2.i′ = c1.i;
} -- layer LL
```

The capabilities of the DisCo method can be used in a fashion where abstractions are mapped to their implementations with invariants that uniquely determine the values of more abstract variables. The scheme can then be used so that abstract versions of specifications refer to abstract concepts. Then, these concepts can be refined towards an implementation by introducing lower-level abstractions, and by proving the associated invariant. For more details regarding the refinement scheme incorporated in the formalism, the user is referred to [10].

When the above procedure is applied in a recursive fashion, a hierarchy of abstractions is obtained [14]. Each level of the hierarchy describes the system with its own concepts. These concepts can be mapped to more concrete ones when advancing towards implementation, or traced back to higher-level concepts where more abstract descriptions of the system can be found. The top level of the hierarchy is the most abstract description of the system where everything is possible. In this paper, we will refer to this specification as ϵ. The lowest level refers to actual code modules.

By establishing an abstraction hierarchy, it becomes possible to measure the relative complexity of the implementation with respect to its abstract specification. For the purpose of software evolution, this is a key concept to manage the direction the implementation is heading. The divergence of actual code modules from the abstractions included in the hierarchy provides evidence on potential future problems for future evolution.

A primitive abstraction hierarchy where all abstractions follow intermodule interfaces is a layered architecture. For instance, a file is an abstract concept that often has a layered implementation. We, however, allow abstractions as an auxiliary concept that can be used to support software evolution and the creation of related visions.

3. Maintenance based on Abstraction Hierarchy

When an abstraction hierarchy has been established, it provides a reference for any new features of the system. When a new requirement emerges, it can be related with the abstractions already incorporated in the system in terms of the hierarchy. Further, based on the level of abstraction in the hierarchy, the relative cost for implementing the new requirement can be justified due to the following. When a change is required at a very abstract level, it is likely that many implementation modules require changes, because the cross-cutting of the abstraction is large. On the other hand, if a change is related to a low-level abstraction only, it is likely that required modifications can be handled locally within the scope of that particular abstraction. In fact, at the level of primitive abstractions, interfaces can remain unchanged provided that the design of the abstraction has been appropriate. Obviously, based on the information obtained from the hierarchy, the designer can analyse different implementation alternatives, and their related effect in coding, testing, and integration.

For more details on the management of evolution, consider the following. Whenever a new requirement is identified, it is associated with a certain abstraction in the hierarchy by analysing the effects of the change. The lowest-level abstraction that will remain unchanged will be referred to *stable root*. This abstraction, all the abstractions above this level, and abstractions that are independent of stable root remain unchanged. In contrast, abstractions that are needed for deriving stable root into more concrete form potentially need to be reengineered. In order to identify the needed changes, the layers below stable root specification need to be analysed with respect to the new requirements. Then, the lower-level abstractions are modified to support the higher-level upgrades recursively. In reality, new layers are often required, or at least provide a justifiable way to specify the newly emerged properties.

In addition to the use of stable root as an indicator for changes in the specification level, verification and validation effort can be focused. As we know that only abstractions below stable root are modified, it is enough to re-test abstractions below the root. In reality, however, it is often desirable to run e.g. old test cases as a regression test to validate the preservation of unchanged abstractions. Still, even this case is made easier because we know that the test outcomes should remain unchanged, resulting in straightforward automatic analysis of test results.

Ideally (and also usually in practice) the top levels of the abstraction hierarchy experience little or no evolution. In contrast, towards the lower levels of abstraction, more and more changes occur. This reflects the intuitive assumption that maintenance is not risking the fundamental concepts of the system, but extends implementation with new details thus enhancing the system.

Based on the above discussion, the abstraction hierarchy supports separation of implementation details and high-level abstractions reflecting fundamental concepts. This is crucial for software evolution. Without such separation, it is difficult to justify the decisions taken to manage evolution except as a reflection of resulting implementation architecture. Then, evolution is effectively code manipulation with little possibilities for fact-based management of main concepts.

4. Example: Abstractions in a Mobile Switch

As an example we give an abstraction hierarchy for a mobile switch, and show how new properties could be attached to the specification. The switch routes calls[2] from callers to callees. In some cases a call is first routed to one subscriber and then forwarded to some other. The example is a simplified version of more comprehensive work carried out in DisCo project, where selected parts of an existing mobile switch were modeled.

[2]We do not give exact meaning to the notion *call*, which is perhaps the most intuitive starting point in modeling a switches. However, starting with call leads to difficulties, as pointed out by Zave in [17].

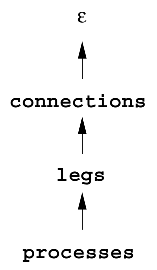

Figure 1. Abstraction hierarchy.

4.1. Deriving an Abstraction Hierarchy

The abstraction hierarchy derived in this subsection are *connection*, *leg* and *process*. The hierarchy is depicted in Figure 1. Abstractions are discussed in detail in the following.

The highest abstraction in the hierarchy is *connection*. Informally a caller has connection to the subscriber (callee) to whom a call has been routed. For example, if subscriber `A` calls B then after successful routing `AB`-connection is created. If the call is then forwarded to some other subscriber (`C`), `AB`-connection is replaced by `AC`-connection. And if it is again routed to `D`, `AC`-connection is replaced with `AD`-connection.

Formally connections are DisCo objects (introduced in layer `connections`) which have state machines with three states: `unborn`, `active` and `terminated`. Two variables (`from` and `to`) referring to subscribers are embedded in state `active`:

```
class Connection = {
  state = (unborn,
           active(from, to : reference Subscriber),
           terminated)}
```

In this layer three actions are introduced for changing the states of connections: `connect`, `redirect` and `disconnect`. Only action `redirect` is given as an example:

```
action redirect(to: reference Subscriber;
                c: Connection):
  c.state = active →
  c.active.to' = to;
```

Connections are implemented with *leg*s, which form chains from subscriber to subscriber. In our earlier example where `A`'s call was first routed to `B` and then to `C` and finally to `D` there are three legs: `AB`- `BC`- and `CD`-leg, which all together implement an `AD`-connection (see Figure 2).

Formally legs are DisCo objects given in layer `legs`, which imports the layer `connections`.

```
class Leg = {
  state = (unborn,
           active(a, b : reference Subscriber;
                  next, prev: reference Leg)
           terminated)}
```

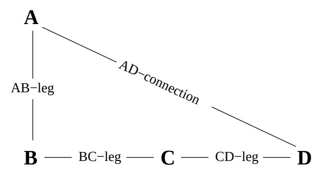

Figure 2. Abstractions connection and leg.

Variables `a` and `b` are references to the subscribers related by the `leg`; `next` and `prev` are used to form linked lists of legs.

In addition to plain `Legs`, the layer introduces relation `isPartOf` between legs and connections, which states that there is an arbitrary number of legs for each connection, and either no or one connection for each leg.

```
relation isPartOf(Leg, Connection) is
  0..*:0..1;
```

Invariant `legChainImplConnection`[3] relates connections and legs. Intuitively it states that if there is an active connection between two subscribers, then a chain of active legs (implementing the connection) exists between the two subscribers (as is the case in figure 2).

```
invariant legChainImplConnection =
  ∀ c: Connection | c.state.active ::
    ∃ first, last: Leg |
      isFirstLegInChain(first) ∧
      isLastLegInChain(last) ∧
      areMembersOfTheSameLeg(first, last) ::
        first.state.active.a = c.state.active.from ∧
        (first, c) ∈ isPartOf ∧
        last.state.active.b = c.state.active.to ∧
        (last, c) ∈ isPartOf;
```

The layer has five actions: `startLeg`, `addLeg`, `startTearingDown` and two actions for tearing down a chain of legs. Action `addLeg` is given below as an example. The action is a refinement of the action `redirect` in the layer `connections`. It states that if there is a chain of legs ending in subscriber `sa`, then a new leg can be set from `sa` to any subscriber `sb` (and `connection c`, which is partly implemented by the legs `lPrev` and `lNext`, is atomically redirected to `sb`).

```
action addLeg(sa, sb: Subscriber;
              c: Connection;
              lPrev, lNext: Leg)
refines connections.redirect(sb, c) ∧
  lPrev.state.active.b = sa ∧
  isLastLegInChain(lPrev) ∧
  (lPrev, c) ∈ isPartOf ∧
  lNext.state.unborn →
  lNext.state' = active(a'=sa, b'=sb, next'=null) ||
  lPrev.state.active.next' = lNext ||
  isPartOf' = isPartOf + {(l, c)};
end;
```

[3]Moreover, the layer has three more invariants stating that there is one `Connection` for each `active Leg`, and there exists at least one `Leg` for each `active Connection`, and that consecutive `Legs` are implementing the same `Connection`. These are omitted here for brevity.

The next step in the abstraction hierarchy is this layer `processes`, where legs are implemented with processes. The layer is omitted here.

4.2. Evolution

Having the three-level abstraction hierarchy described above enables us to measure how big is the cross-cutting of our visions of changes to the system. If, for example, the change is such that a connection is the stable root, we can conclude that the change is relatively large (or our original understanding of the system was poor). On the other hand, if the change affects only the process level (connection and leg remain unchanged) then it is minor upgrade. In the following, we give some examples on how to manage software evolution with the abstraction hierarchy established above.

Difficult Modification: Eavesdropping

An example of a difficult evolutionary step is adding eavesdropping to the mobile switch. In some countries the government requires that there must be a possibility for legal authorities to listen calls of suspicious customers. In our abstraction hierarchy the stable root is an empty specification ϵ above `connection` abstraction. In other words, all layers require modification.

The modifications are as follows. In layer `connections`, we must add reference to possible eavesdropper to the state `active`:

```
class Connection = {
  state = (unborn,
           active(from, to: reference Subscriber;
                  eavesdropper: reference Subscriber),
           terminated)}
```

After this we must investigate and possibly reengineer the actions handling connections. For example, in action `redirect` we must take care that attribute `eavesdropper` is updated properly[4]. If the callee to whom the call is rerouted is suspicious then the eavesdropper starts to listen the call, else the eavesdropping status remains as it was before the action:

```
action redirect(to: reference Subscriber;
                c: Connection):
  c.state = active →
  c.active.to′ = to ∧
  c.active.eavesdropper′ = if isSuspicious(to) then
                             theEavesdropper
                           else
                             c.active.eavesdropper;
```

Changes to layer `legs` are similar. We must add new attribute (`eavesdropper`) to class `Leg` and reengineer actions referring to leg objects. Moreover, invariant `legChainImplConnection` must be revisited. Changes to layer `Processes` are omitted here for brevity.

[4]For simplicity we have only one legal authority carrying out eavesdropping.

$$\begin{tikzcd}
\varepsilon \\
\texttt{conference calls} \arrow[u] \\
\texttt{connections} \arrow[u] \\
\texttt{legs} \arrow[u] \\
\texttt{processes} \arrow[u]
\end{tikzcd}$$

Figure 3. Upgraded abstraction hierarchy.

Related verification and validation activities also require major attention. In fact, all test cases should be rerun as such (regression tests) and in a fashion where eavesdropping is active. In reality, this degree of testing for eavesdropping only is of course unrealistic.

In order to add eavesdropping to the specification we made changes to existing abstractions. This is not the case always, it is also possible to add totally new abstractions for the system. For instance, conference call would be a totally new abstraction for our example. A normal call could then be derived from conference calls by limiting the number of participants to two. The upgraded hierarchy is illustrated in Figure 3. Obviously, the changes related to this upgrade as well as related validation and verification effort can be estimated to be considerable.

Simple Modification: Knocking

Example of a minor cross-cutting is *knocking*. If subscriber `a` is speaking on a phone with `b` and she is called by a third subscriber `c` then `a` hears a voice of knocking in her phone and can answer that call. In this case the stable root is the layer `Legs` because only the layer `Processes` is changed.

Obviously, verification and validation effort implied by this modification is also minimal.

5. Conclusions

We have presented an approach to handling a hierarchy of non-primitive abstractions to ease software evolution. The main contribution of the paper is in showing that such hierarchies can be rigorous. Moreover, we have outlined an example of using abstraction hierarchies in a mobile switch, and showed how this makes software evolution more manageable. The example was a simplified version of a more comprehensive case study carried out during DisCo project.

A similar approach has already been introduced in [15], although in an informal setting. In that context, the relation between higher-level abstractions and their implementations is handled with links of a browser tool and the underlying data base. This practical example also supports our claim that lower levels of abstraction evolve more than higher abstractions. While the demonstration in that context provides justification on industrial applicability of this approach. The introduction of the related formalism in this paper is an obvious improvement in the theoretical sense. In practice, this also results in the option to use the tools associated with the formalism [1].

In real life software engineering, the approach requires more work in short turn. We must investigate the effect of evolution to the specification, and reflect the changes to implementation level via the abstraction hierarchy. However, more comprehensive understanding of changes, and related documentation in the specification, compensates this in the long run.

Acknowledgements

This work is partly funded by the Academy of Finland (No. 100005).

References

[1] T. Aaltonen, M. Katara, and R. Pitkänen. DisCo Toolset – The New Generation. *Journal of Universal Computer Science*, Vol 7:1, 3–18, Springer-Verlag, 2001.

[2] M. Awad, and J. Ziegler. A practical approach to object-oriented state modeling. *Software - Practice and Experience*, 27(3):311–328, March 1997.

[3] G. Abowd, L. Bass, P. Clements, R.Kazman, L. Northrop, and A. Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. Software Engineering Institute, Technical Report CMU/SEI-96-TR-025, January 1997.

[4] J. O. Coplien. Idioms and patterns as architectural literature. *IEEE Software,* 14(1):36-42, January 1997.

[5] DisCo homepage. `http://disco.cs.tut.fi/`.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns*. Addison Wesley, Reading, MA, 1995.

[7] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering,* pages 63–71, 1990.

[8] P. Kellomäki and T. Mikkonen. Modeling distributed state as an abstract object. In *Distributed and Parallel Embedded Systems, proceedings of the IFIP WG 10.3 / WG 10.5 International Workshop on Distributed and Parallel Embedded Systems (DIPES'98),* pages 223–230. Kluwer Academic Publishers, 1999.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (Eds. M. Aksit and S. Matsuoka),* pages 220–242, Springer-Verlag LNCS 1241, 1997.

[10] R. Kurki-Suonio and T. Mikkonen. Abstractions of distributed cooperation, their refinement and implementation. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 94–102. IEEE Computer Society, 1998.

[11] R. Kurki-Suonio and T. Mikkonen. Harnessing the power of interaction. In H. Jaakkola, H. Kangassalo, and E. Kawaguchi, editors, *Information Modeling and Knowledge Bases X,* pages 1–11, IOS Press, 1999.

[12] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[13] L. Lamport. Composition: A way to make proofs harder. Digital Systems Research Center, Technical Note 1997-030a, December 1997.

[14] T. Mikkonen and H.-M. Järvinen. Specifying for releases. *International Workshop on Principles of Software Evolution*, pages 118–122. April 20–21, Kyoto, Japan, 1998.

[15] T. Mikkonen, E. Lähde, M. Siiskonen, and J. Niemi. Managing software evolution with the service concept. *Proceedings of the International Symposium on Principles of Software Evolution* (Eds. Takyo Katayama, Tetsuo Tamai, and Naoki Yonezaki), pages 43–47, Kanazawa, Japan, November 1–2, 2000.

[16] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15:2,* December 1972.

[17] P. Zave. 'Calls considered harmful' and other observations: A tutorial on telephony. *Services and Visualization: Towards User-Friendly Design* (Eds. T. Margaria, B. Steffen, R. Rückert and J. Posegga), pages 8–27, Springer-Verlag LNCS 1385, 1998.