A Short Catalogue of Abstraction Patterns for Model-Based Software Engineering

Bran V. Selic
(Malina Software Corp., Nepean, Ontario, Canada)

Abstract Abstraction is a fundamental component of software design. In particular, it is crucial in the conception of large and complex software systems, whose implementations are so intricate that they tax our cognitive capacities. The need to design, comprehend, and manage the evolution of such systems has given rise to the notion of software architecture—an abstract representation of a software system which focuses solely on its essential elements and relationships. Unfortunately, the process of abstraction by which such models are derived is typically complex and highly subjective, making it difficult to validate and document. This leads to a number of practical problems including, notably, the potential for difficult-to-detect discrepancies between design intent and implementation. To mitigate this, we identify and describe some of the most common abstraction patterns used by software architects to render a complex system into a more comprehensible form. For greater clarity, each pattern is described in detail using a provisional graph-based formalism. The intent is to not only provide a precise description of the abstraction process for each pattern, but also to open the possibility of providing automation support for documenting and validating the steps involved in transitioning between system representations at different levels of abstraction.

Key words: model-based software engineering; software patterns; software architecture


1 Acknowledgement and Dedication

It is perhaps unusual to commence a technical paper with a personal acknowledgement. However, considering the nature of the special issue in which it appears as well as the inspiration that drove the work reported, this seems quite appropriate.

Namely, this work was directly inspired by a comment by Prof. Manfred Broy, which he made in the course of one of our technical discussions. We had been collaborating on a joint research project seeking to define a formal semantics for the latest version of the Unified Modeling Language (UML), and I had just remarked that we needed a proper theory of modelling language design, similar to the kind of sound theoretical underpinnings we have established for traditional computer programming languages. Prof. Broy agreed, but suggested that the problem was a greater one and that what interested him was a theory of software modelling in general. This casual remark has haunted me ever since, opening a much broader perspective on the
problems that I had been studying up to that point and acting as a catalyst in my subsequent work.

Consequently, it is with great gratitude and humility that I dedicate this work to my colleague and friend, Prof. Manfred Broy, whose insight inspired it. However, it must be noted clearly that any deficiencies and defects in it are purely my own.

2 Introduction

Since the process of abstraction is at the very heart of all modelling, it seems an obvious topic in developing a comprehensive theory of modelling. However, while the process of abstraction and its inverse, refinement, have been explored and theorized about for thousands of years in various disciplines (e.g., Aristotle’s writings on the subject are still as current today as they were in his time[13]), including in computer science (Refs. [3, 4, 6]), with the exception of a few specialized cases (e.g., Ref. [7]), to the best of the author’s knowledge the specific forms of abstraction commonly used by software architects have not been systematically collected, catalogued, and specified. This is somewhat surprising given the abundance of work on software patterns since the publication of the now renowned “Gang of Four” book[8], which initiated an important trend to capture commonly used general solutions encountered in software practice. Consequently, the primary purpose of this article is to rectify this apparent shortcoming by providing a starter set of such patterns.

Like any technical reference, a catalogue of abstraction patterns is useful because it provides a clear and comprehensive set of potential choices to designers faced with a design problem. Moreover, if the process involved in applying a pattern is defined precisely enough, it may be possible to automate some of the more mechanistic aspects of that operation. To be clear, the selection and specification of the appropriate pattern in a given situation is still the prerogative of the human designer, since abstraction (which is, after all, a means to facilitate understanding) requires deeper insight into human needs and perspectives and is not readily automatable. But, once a particular pattern has been chosen and specified, much of the mechanics involved in its application and subsequent exploitation (e.g., searching and tracing) can be delegated to a computer. This not only cuts down on menial work and saves time but also reduces the likelihood of errors due to carelessness or oversight. In particular, such computer supported abstraction transformations enable the creation and maintenance of formal computer-tractable bidirectional links between two or more models at different levels of abstraction or based on different viewpoints. And, even though it does not eliminate the possibility of inappropriate abstractions—since those are ultimately selected and defined by humans—a such facility can be extremely useful in tracking down and identifying inappropriate use of or corruption of an abstraction. Thus, abstractions need not remain hidden in the minds of designers or buried inside informal and untrustworthy documents.

In the next section, we examine where a catalogue of abstraction patterns might fit in the relationship between abstraction and models. Abstraction has different forms and, in Section 4, we identify the particular form that is used throughout this work and introduce featured graphs, a provisional formalism chosen for its suitability for describing certain major categories of abstraction patterns. The actual catalogue of patterns starts in Section 5, with a specification of common patterns for abstracting
structural aspects of a software system. Behavioural patterns are described in Section 6. Finally, a basic set of common temporal patterns, which abstract time in various ways, completes this initial catalogue. Temporal patterns differ qualitatively from the previous two categories and are not described using featured graphs. They are included primarily to provide a glimpse of the broader perspective on abstraction, of which the work reported here is merely the beginning. The concluding section summarizes the work and identifies possible future directions of research.

3 On Software Models and Abstraction

Model-Based methods are increasingly being recognized as an effective approach to the design and development of software that can mitigate some of the pressing problems plaguing more traditional code-centric approaches. A primary distinction of these methods is the use of higher-level domain-specific concepts to express both problems and solutions, in place of the more traditional implementation-oriented concepts, which often reflect far too much of the confusing idiosyncrasies of the underlying computing technology. Clearly, by abstracting away technological detail and using concepts and terminology that are closer to the application domain, it is easier to focus on the problem at hand and to describe proposed solutions in a concise and effective manner.

However, there is inherent danger lurking in any kind of modelling – the problem of accuracy. That is: how can we ascertain that a model is an accurate rendering of the reality that it represents? As the old proverb states and engineering experience amply demonstrates, the devil often hides in the details. In progressing from the concrete to the abstract, there is always the very real possibility that something crucial has been abstracted away due to an inadequate understanding of its relevance. This problem is particularly pernicious in software, because traditional programming languages are notoriously sensitive to even the smallest flaws in logic. Misaligned pointers, uninitialized variables, or even misplaced punctuation symbols have been known to cause dramatic and sometimes catastrophic failures. Some of the more advanced model-based technologies can eliminate whole categories of these problems by automatically generating programming language implementations from models, often without any direct human intervention in the generation process. Unfortunately, experience has shown that, despite their use of higher-level concepts, such comprehensive implementation-ready models are generally far too rich in detail to support focused human reasoning about the system as a whole. In other words, even with the wonder of fully automated model to code translation, there is still a need for more abstract high-level models, that is, models of models.

This then gets us back to the original problem: how can we ensure that these higher-level models are accurate renderings of the software? The approach proposed here does not solve the issue, but provides a facility that can mitigate it. It does so not only by providing a convenient list of useful forms of abstraction but also by defining them in a way that allows us to exploit the power of computers to assist us in applying them, as illustrated in Fig. 1.

Needless to say, these patterns can also be applied in the opposite direction, going from the abstract model to the refined one. Therefore, they can also be called refinement patterns. In this article, however, we shall describe all the patterns as
mappings from a lower to a higher abstraction level, with the understanding that the mappings may be bidirectional.

Figure 1. The process of model abstraction using the catalogue of abstraction patterns

In describing these patterns, we shall not frame them in terms of any specific modelling language, but will use the terminology and many of the concepts found in UML 2, which is relatively well known and intended as a general-purpose language that serves many different domains. This is simply a choice based on convenience, the patterns can be used equally well with most other modeling languages.

4 A Utilitarian Model of Abstraction

As noted above, abstraction is a subtle concept much studied in both philosophy and science, serving different purposes and having multiple different manifestations\(^1\). For our needs, we adopt a rather simplified model in which abstraction is defined as the process of selective reduction of information from some source set of entities—based on a chosen criterion or viewpoint\(^2\)—resulting in a new entity which represents the original\(^3\). We shall refer to the entity derived by this process as the abstraction of the original more concrete source. The original source, or, if you prefer, the outcome of the inverse process, is called a refinement. (Note that this view of abstraction is referred to as the reduced model, which differs from the relaxed model that defines abstraction as the process of weakening some of the constraints applying to refinement\(^12\)).

Although abstraction involves a reduction of information, it is often much more than just simplistic removal of detail from a specification or a model. Two models at different levels of abstraction may involve different conceptual spaces, possibly involving different concerns. For instance, in the Reference Model for Open Distributed Processing (RM-ODP), a standardized conceptualization of computing systems, a given system is described from five different viewpoints, representing escalating levels

---

\(^1\) For example, Wikipedia lists seven different interpretations of the term abstraction (http://en.wikipedia.org/wiki/Abstraction).\(^2\) The IEEE 1471 standard, defines viewpoint as “a pattern or a template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis”\(^10\). The question of how viewpoints are defined and selected for a given abstract model is outside the scope of this article.\(^3\) To be complete, we should also include the notion that the abstraction must preserve the relevant properties of the original based on the chosen viewpoint. However, the definition of relevance and ways of ensuring it are outside the scope of this article. Readers interested in a more comprehensive view of various theories of abstractions are referred to the report by Guinchiglia et al.\(^9\). More information on the differences between reduced and relaxed models of abstraction can be found in the work by Knoblock\(^12\).
of abstraction, using a different modeling language at each level\cite{17}. Thus, elements found at one level of abstraction may take on completely different forms in some other level.

Nevertheless, abstraction can be modeled as a transformation, and, given that models are typically specified with graph-based modelling languages, it is convenient to specify it as a graph transformation. One common and proven approach for representing graph transformations is via \textit{triple-graph grammars}, or \textit{TGGs} for short\cite{18}. The core idea here is that a given graph can be transformed into a different graph by applying a set of mappings, which is itself defined as a graph (referred to as the \textit{correspondence graph}). In general, each graph in such a triple can have its own schema to which it must conform. Consequently, TGGs are often used to translate models from one graph-based language to another.

The choice of a suitable formalism for describing abstraction patterns—whether TGGs or some other graph-based grammar (e.g., Ref. \cite{19})—is an issue that requires further study. Since this article is intended primarily as an introduction to abstraction patterns, we use here a rather ad hoc graph-based formalism called \textit{featured graphs}, chosen for its convenient proximity to the widely used UML notation. Moreover, to further simplify the descriptions of our abstraction patterns, we decree that both the abstraction and the refinement graphs conform to the same syntactical schema (or \textit{metamodel} in modelling language terminology).

It is important to note here that the critical issue of \textit{semantic conformance} between a refinement and its abstraction is not discussed in this work. Semantic conformance is the property that an abstraction is phenomenologically consistent with its corresponding refinement in the sense that the relevant structural and behavioural properties of the refinement are retained in the abstraction. Although this is a critical topic when studying abstraction in software, it is a distinct and separable concern (see, for example\cite{4,6,11}). Instead, we focus primarily on the \textit{syntactic} aspects of abstraction transformations, assuming that the responsibility for ensuring semantic conformance rests with the designer who selects and applies a given pattern.

4.1 \textit{Featured graphs}

The patterns we are dealing with here can be grouped into three separate categories, depending on the domain that they are modelling: \textit{structure}, \textit{behaviour}, and \textit{time}. Despite the obvious differences between these domains, for visually-oriented modelling languages such as UML, the first two categories can be represented by abstract graphs that conform to a common schema. We refer to these types of graphs as \textit{featured graphs} (Fig. 2)—so called because both their nodes and edges may have associated features (or attributes).

The nodes in a featured graph (e.g., $n_1$, $n_2$, and $n_3$ in Fig. 2) can represent either structural or behavioural concepts. For example, a node may represent a role or part in a composite structure or an action within a UML activity. Each node and edge can own a set of \textit{features} (e.g., $f_1^1$, $f_2^2$, etc.) that characterize it in some way. A distinctive kind of node feature is called a \textit{pin} (e.g., $p_1^1$, $p_2^2$, etc.). Pins serve as anchor points for the edges in a graph. In general, edges can join more than just two nodes (e.g., edge $e_{12}$) and have a direction. The latter is defined by specifying the set of \textit{source} pins and the set of \textit{destination} pins for each edge.
This model is quite general. The semantic interpretation of the entities in a graph depends on what they represent. Thus, in the case of UML, a node may represent an instance of a structured class, a port, a collaboration role, or a component. On the other hand, if the graph represents a behavioural model, a node may represent a state, an action, or an activity. Similarly, an edge may represent a structural connector joining two or more ports, a data or control flow connecting pins in an activity, or a transition in a state machine.

In general, a featured graph is defined as a tuple

\[ \text{FG} = \langle \text{Nodes}, \text{Edges} \rangle \]

where \text{Nodes} denotes the set of nodes and \text{Edges} the set of edges. Each node \( n \) in the set \text{Nodes} is defined by a triple:

\[ n = \langle \text{id}, \text{Ftrs}(n), \text{Pins}(n) \rangle \]

where \text{id} is a unique node identifier and \text{Ftrs}(n) is the set of (non-pin) features of \( n \) and \text{Pins}(n) denotes the set of its pins. We shall use the notation form \( n::f \) and \( n::p \) to designate individual features and pins respectively of node \( n \).

Each edge \( e \) of the set \text{Edges} is defined by a triple of the form:

\[ e = \langle \text{id}, \text{Ftrs}(e), \text{Srcs}(e), \text{Dests}(e) \rangle \]

where \text{id} is a unique edge identifier, \text{Ftrs}(e) is the set of features of \( e \), and \text{Srcs}(e), \text{Dests}(e) are pins that identify its sources and destinations, respectively. Note that the pins belong to nodes and are not considered features of an edge. The notational form \( e::n::p \) is used to refer to a particular end of edge \( e \) that is attached to pin \( p \) on node \( n \).

4.2 Describing Abstraction patterns with featured graphs

Featured graphs help us specify two of the three categories of abstraction patterns in a more precise way. This is done by defining them as a mapping between two such graphs, one representing the abstraction and the other the refinement; that is:

\[ \text{Pattern} = \langle \text{RefGraph}, \text{Mappings}, \text{AbsGraph} \rangle \]

where \text{AbsGraph} is a graph representing the abstraction obtained when the set of mappings defined by \text{Mappings} is applied to the original refinement graph \text{RefGraph}. Each mapping in \text{Mappings} is a pair that identifies the correspondence
between an element in $\text{AbsGraph}$ and exactly one element in the $\text{RefGraph}$. Of course, more than one $\text{RefGraph}$ element can map onto the same $\text{AbsGraph}$ element since abstraction usually involves reduction. Note that the mappings involve not just the node and edge elements, but also their pins and features. In this article we do not specify any feature mappings, since these are not easily generalized and vary to a large extent on the types of features involved. However, in practical situations, feature mappings would be part of the abstraction specifications. An example of such a mapping might be the translation of the individual bandwidth attributes for a collection of communications channels in a refinement model into a corresponding bandwidth for the single abstract channel that represents that collection within the abstract model (as described in the Cable pattern below).

Requiring that each refinement element must be mapped to a corresponding abstraction element ensures that all elements of source are accounted for explicitly\(^4\) and, more importantly, that the mappings are fully reversible. This in turn enables automated traversal of the models hierarchy in either direction. Furthermore, it allows us to automatically derive an abstract representation of a system from a refined one by simply re-applying the chosen abstraction transformations. Automation can also help us in the opposite direction, allowing the possibility of semi-automated generation of a refined model from an abstract one. As noted before, this is not intended to replace human decision making but to complement it. Modellers still need to select the refinements that they want to apply, specifying the necessary details of each mapping.

It is important to note that, in general, a given refinement element can participate in more than one abstraction mapping. For instance, a set of abstract connectors in the abstraction graph may all be realized by a single physical wire in the refinement. In such cases, when the abstraction mappings are reversed (i.e., when an abstraction is refined), the results may overlap and care must be taken to ensure that there are no semantic conflicts or order dependencies between the mappings (this is generally not a problem for nodes and edges, but may be an issue for some types of features).

Formally, a mapping $\text{map} \in \text{Mappings}$ is defined as a relationship between an element of the refinement graph and a corresponding element in the abstraction graph:

$$\text{map} = <\text{absElem}, \text{refElem}>$$

where $\text{absElem}$ is an element in the set of elements of the abstract graph (defined by the union of all edges, nodes, node features, edge features, and ports), and $\text{refElem}$ is the corresponding element in the refinement graph.

As noted in Section 3, although syntactically simple, a mapping may involve a paradigm shift, due to potentially different semantics between the two levels of abstraction. For instance, an active component in UML may be mapped to an operating system process or task, and, even though there is a one-to-one relationship between the two matching concepts, they exist in very different worlds. The concrete operating system domain is replete with technology-specific concepts such as stacks, heaps, priorities, schedulers, and the like, all of which are abstracted away in the UML domain.

\(^4\) Many errors in abstraction occur due to simple oversight whereby an important feature is accidentally left out of consideration.
Note that, in general, despite the fact that both the abstraction and the refinement are expressed using the same graph schema there are no type restrictions on the mappings. For example, a structural component may be mapped to a port, or a connector may be mapped to a network of interconnected components, and so on.

5 Structural Patterns

Patterns in this group apply to structure. By “structure” we refer to elements that, ultimately, take up space at run time (e.g., memory) and which, as a result, may have a state and, in some cases, an identity. Although we distinguish structural patterns from behavioural patterns, many of them in one domain have an almost perfect analogue in the other. This is because the individual patterns are simply domain-specific manifestations of the common general principles of abstraction.

5.1 The Black Box pattern

This is perhaps the most commonly used structural pattern, in which a set of interconnected structural components that cohere to each other in some sense (e.g., they collaboratively realize some higher-level function) are replaced by a monolithic component such that the refined components and their interconnections are no longer visible.

Figure 3 illustrates how the Black Box pattern works. In this example, all the components inside the dashed box labelled “glass box”\(^5\) are to be abstracted, along with their features and connectors (represented by edges) into a single monolithic node called a'\(^6\). However, special treatment is required for connectors e\(_1\) and e\(_2\) which are only partly within the glass box since they are joined to elements outside it. These are mapped one-to-one to the distinct connectors at the higher abstraction level (e\(_1'\) and e\(_2'\) respectively). Connectors that cross the glass box boundary are called cross-over connectors (or, more generally, cross-over edges).

Note that, in situations such as the one in Fig. 3, where the abstraction level supports the concept of communication ports, it is also necessary to map the ports that are connected to the cross-over connectors (c\(_1::p_1\) and c\(_3::p_2\) in this example) to corresponding higher-level ports on the abstract black box component (a'::p\(_1'\) and a'::p\(_2'\) respectively).

Using the terminology of featured graphs, we can specify this pattern more precisely in the following manner:

**RefGraph:**

- The set of nodes (Nodes) of this graph consists of the set of components in the refined model to be abstracted (i.e., the components encompassed by the glass box). If those components have ports, then these represent the set of pins of the **RefGraph**. Finally, the set of edges (Edges) of **RefGraph** consists of the connectors of the refined model, including all cross-over connectors.

\(^5\) We use the term “glass box" to refer to the specific subset of elements out of the total set of elements in a refinement model that are directly involved in the pattern.

\(^6\) Note that components cx\(_1\) and cx\(_2\) and their matching correspondents cx\(_1'\) and cx\(_2'\), are outside the glass box and are, strictly speaking, not a part of this pattern. Instead, they are presumed to be part of a different abstraction. In this and other diagrams, such contextual elements are identified by being drawn either with an alternating dot-dash line, or filled in gray.
Mappings:
For the subset of all elements of \textbf{RefGraph} that excludes (1) the cross-over connectors and (2) any ports that are connected to the cross-over connectors, there exists a mapping to a single abstract component \(a'\) in \textbf{AbsGraph}.

For each cross-over connector \(c\) in \textbf{RefGraph}, that is connected to a port \textbf{Cmp}::\textbf{p} of a component in \textbf{RefGraph}, there exists a mapping to a corresponding abstract port \(a'::p'\) on the abstract component \(a'\).

For each cross-over connector \(c\) in \textbf{RefGraph}, there exists a mapping to a corresponding abstract connector \(c'\) in \textbf{AbsGraph}. If ports are supported in the abstraction level, then one end is connected to the matching abstract port \(a'::p'\), otherwise it is connected directly to the abstract component \(a'\).

\textbf{AbsGraph}:
The set of nodes of this graph comprises a single component \(a'\). If the model supports the concept of ports and there are cross-over connectors in \textbf{RefGraph}, then there is a port \(a'::p'\) on this component for each port in \textbf{RefGraph} that is the source of a cross-over connector. The set of edges consists of a connector for each cross-over connector in \textbf{RefGraph}. This connector is anchored at one end to either component \(a'\) itself or, if ports are supported, to a port \(a'::p'\) on this component.

A common and somewhat simpler variant of the Black Box pattern occurs when the glass box includes a container component, which encapsulates the internal network of collaborating components (Fig. 4). Because of the encapsulation, there are no cross-over connectors to deal with. However, the encapsulating component may have ports on its outer shell. This simplifies the mappings, since the entire glass box—excepting its own ports—with all its contained components (including their ports) and connectors are mapped to the single abstract component. The ports of the containing component are mapped to corresponding ports on the abstract component. (For brevity, we omit the detailed specification of this variant of the pattern.)

Note that any one of the contained components in the refinement graph may encapsulate other finer-grained components. As might be expected, the mapping of a contained components implicitly signifies the mapping of any its contained components and connectors to the same high-level abstraction. Thus, there is no need to provide explicit additional mappings for deeply nested elements.
5.2 The Black Line pattern

This is, in essence, a variant of the Black Box pattern, but applied to the very special case of components that serve as communication channels. In many modelling languages, communication channels are depicted simply as lines connecting two or more collaborating components (or their ports). This is usually a significant simplification (abstraction) of their true nature, since most communication channels are not simple wires but aggregations of finer-grained components and communication channels, as illustrated by the example in Fig. 5. Thus, when we use a simple line segment to represent the connection between a Web-based client communicating with a remote server over the Internet, that line subsumes the full complex functionality of the Internet. However, the simple line representation is useful because it helps us focus on the component couplings that these channels realize rather than the details of their implementation.

Note that in this pattern, the cross-over connectors of the refinement map to the connector end points of the abstraction. These may be attached either to ports or directly to components, as the case may be. The remaining connectors and components (with or without ports) within the glass box all map to the black line connector.

RefGraph:
The “glass box” in this case encompasses all those components and connectors that participate in the communication function that allows elements attached to the cross-over connectors to communicate with each other. The nodes of this graph are the components of the glass box and the edges are the connectors within the box.

Mappings:
For each cross-over connector in the RefGraph there exists a mapping to a connector end point in the abstraction graph AbsGraph.

For all components and connectors, except the cross-over connectors, there exists a mapping to the single connector \( e' \) in the AbsGraph.

AbsGraph:
The set of nodes of this graph is empty and the set of edges consists of just the single connector edge \( e' \).

5.3 The Cable pattern

This pattern is often used in conjunction with the Black Box pattern and has some similarity to the Black Line pattern. It applies in the case where there are multiple connectors between two components or the ports of those components as shown in the example in Fig. 6. This collection of connectors is abstracted into a single connector at the higher abstraction level called a cable.

The description of this pattern is straightforward:

RefGraph:
The set of nodes of this graph is empty. The set of edges consists of the collection of connectors to be abstracted all of which are connected to the same pair of refined components (or, if ports are used, to the ports of those components).

Mappings:
For each connector \( c_i \) in the refinement graph there exists a mapping to the single connector \( e' \) in the AbsGraph.

AbsGraph:
As with the refinement graph, the set of nodes is empty. There is a single edge \( e' \) connecting the two components that are the abstractions corresponding to the refined components (or, if ports are used, to the corresponding group ports as described in the Port Group pattern below).

As noted earlier, there is some similarity between the Cable pattern and the Black Line pattern. However, they are different because in the Cable pattern the
various connector end points of the refinement are collapsed into a common end point in the abstraction, whereas they are kept separate in the Black Line pattern.

5.4 The Port Group pattern

As indicated above, this pattern complements the Cable pattern. It is used when a collection of ports, possibly of different types, needs to be abstracted into a single abstract port called a port group. An example is depicted in Fig. 7.

This pattern is described simply by:

RefGraph:
In this case, the nodes of the graph are the ports \( p_i \) all of which are ports of the same refined component (NB: the component itself is not part of the graph).

Mappings:
For each node \( p_i \) in RefGraph there exists a mapping to a single port \( p' \) in the abstraction graph AbsGraph.

AbsGraph:
There is a single node in the graph consisting of the group port \( p' \). The set of edges is empty.

5.5 The Platform Layering pattern

This is one of the most commonly used patterns in architectural specifications. The objective behind layering is to separate elements of a system that are application specific from elements that provide generic support services to the applications, thereby allowing each group to be considered separately. The result is usually depicted as a vertical two-layer structure suggesting that the lower layer acts as a platform for the upper (Fig. 8). The support services are encapsulated within the lower layer while the applications are contained in the upper. The platform provides functionality that is often shared by multiple co-resident applications and, although required by the applications, it is itself application agnostic. The applications, on the other hand, fully depend on the platform for their implementation, but are freed from concerns dealing with how the platform services are realized. A prototypical example of this pattern can be seen in the relationship between software applications and operating
systems. The operating system platform offers a set of general capabilities used by applications, such as controlling access to the CPU, memory resources, multi-tasking support, inter-process communications services, file system access, etc.

![Diagram](image)  
**Figure 8. Common representation of Platform Layering**

If the platform layer is realized by software, then it too will rely on some lower-level platform, and so on, resulting in a vertical stack of platform layers that bottoms out at the hardware layer.

It is important to differentiate this form of layering from the syntactically similar but semantically distinct notion of abstraction layering\(^7\). In the latter case, a given system is represented by succession of descriptions (layers), each one describing the complete system, but distinguished by the amount and type of detail that they expose. Each of the abstraction patterns described in this article is an example of abstraction layering, with the abstract model and the corresponding refined model representing different levels of abstraction of some system. In contrast, each individual layer in platform layering represents a different set of entities.

An example of platform layering is shown in Fig. 9. The refined model shows a flat (non-hierarchical) configuration of components and connectors. However, we would prefer to render this in layered fashion such that the general-purpose service components (in this case components \(c_1\) and \(c_2\) and connector \(c_3\)) are subsumed into the platform layer \(p'\), while the remaining application-specific elements are represented by the application layer \(a'\).

![Diagram](image)  
**Figure 9. An example of Platform Layering**

It can be seen from the diagram that this pattern is actually a composite of two other abstraction patterns. Two separate applications of the Black Box pattern are used to derive the abstraction components \(a'\) and \(p'\) from the contents of two respective glass boxes, while the Cable pattern is used to obtain the connector \(e'\) pointing from \(a'\) to \(p'\). (This connector may be omitted in the concrete representation of the abstraction graph, being implied by the proximity of the two components.)

\(^7\) For a more complete discussion on the subtle differences between various forms of hierarchy, see the paper by Parnas\(^{14}\). Atkinson and Kuehne also discuss the distinction between these often confused concepts\(^{2}\). They also deal with the issue of abstraction (refinement) mappings, but do not provide a description of individual patterns.
6 Behavioural Patterns

These are dynamic patterns that abstract behaviour. As a practical measure, the following patterns are based on the different ways that behaviour is represented in the UML language. These are quite representative of a number of different modelling languages. However, a closer examination quickly reveals that the abstraction patterns involved are simply specialized variants of the same underlying approach used in some of the structural patterns. As noted earlier, the pattern specifications catalogued here deal primarily with syntactical aspects and not with matters of actual behavioural similarity, or semantic conformance. General treatments of semantic conformance between abstractions and refinements can be found in Refs. [3, 4, 6], as well as for the special case of state machine-based formalisms (such as UML statecharts) in Refs. [11, 16].

6.1 The Summary Message pattern

A common form of behavioural abstraction involves the decomposition of a monolithic high-level event into a dynamic sequence of lower-level events. For example, at a high level of abstraction, the initiation of a telephone call can be viewed as a simple request from a caller to the phone system, whereas a more refined view reveals a more complex sequence involving a non-trivial protocol between the two parties, as illustrated in Fig. 10. The high-level message is called the summary message, which is also the name of the pattern.

![Figure 10. A simple Summary Message example](image)

There is noticeable similarity between this pattern and the Cable structural pattern, except that in this case the edges (messages) are directed and are ordered. In
fact, the ordering is actually specified by the relative order of message departures and arrivals (“message occurrences” in UML terminology) at the individual lifelines. We will henceforth assume that this information is either captured directly within the message occurrences or through some associated model elements\(^8\) and not consider it further in this discussion. The mappings of this pattern are based on mapping the event occurrences (message arrival and departure events), which represent the nodes of the refinement graph. The pattern is, therefore, specified as follows:

**RefGraph:**

The nodes of this graph consist of the (ordered) set of message event occurrences. The nodes are characterized by whether they belong to the left or right lifeline. The edges are represented by the collection of messages selected to be abstracted.

**Mappings:**

For each of the nodes of the **RefGraph** associated with the left lifeline, there exists a mapping to the \(l\) node of the abstraction graph **AbsGraph**.

For each of the nodes of the **RefGraph** associated with the right lifeline, there exists a mapping to the \(r\) node of the abstraction graph **AbsGraph**.

For each edge (message) in the **RefGraph** there is a mapping to the single edge \(m_s\) of the abstraction graph **AbsGraph**.

**AbsGraph:**

There are two nodes in this graph: the \(l\) node is an event occurrence that occurs on the left\(^9\) lifeline in the abstract interaction and \(r\) is the node for the event occurrence on the right lifeline. The set of edges has one member corresponding to the abstract summary message \(m'\). The direction of this edge is selected by the modeller.

In the above version of the Summary Message pattern, a direct one-to-one mapping exists between the structural entities involved in the communications between the abstraction and the refinement. However, a more complex form of this pattern also involves structural abstraction as illustrated by the example in Fig. 11. In this case, a high-level (summary) message \(m'\) is sent from abstract component \(a'\) to abstract component \(b'\) over connector \(e'\) (not shown in the diagram here, but present in the model). A more detailed representation of the situation is depicted in the bottom part of the diagram, where it can be observed that the summary message resolves into a more complicated interaction involving a number of finer-grained messages passed between the refinements of abstract components \(a'\) and \(b'\).

---

\(^8\) In UML, for example, this information is captured by an ordered collection maintained by the lifeline object that owns the event occurrences.

\(^9\) The convenience terms left and right are introduced here merely to differentiate the two parties involved. They do not imply any specific orientation in the structure or its concrete representation.
There are clearly multiple abstraction patterns in play here. First, there are the structural mapping from the refined components corresponding to abstract components $a'$ and $b'$ (indicated by the respective glass boxes in the diagram), which are realized by two Black Box abstractions, and a Black Line abstraction pattern for the $e'$ connector between them. The abstraction is completed by a more sophisticated version of the Summary Message pattern as follows:

**RefGraph:**
In contrast to the earlier version of this pattern, the set of nodes in this case includes the lifelines rather than just their pins (ordered message events).
The edges comprise the selected set of messages that interconnect the pins of the lifelines.

**Mappings:**
For each of the nodes of the RefGraph associated with the left lifeline $l$, there exists a mapping to the lifeline $l'$ node of the abstraction graph $\text{AbsGraph}$. For each of the nodes of the RefGraph associated with the right lifeline $r$, there exists a mapping to the lifeline $r'$ node of the abstraction graph $\text{AbsGraph}$. For each edge (message) in the RefGraph there is a mapping to the single edge $m'$ of the abstraction graph $\text{AbsGraph}$.

**AbsGraph:**
There are two nodes in this graph: $l'$ for the left lifeline and $r'$ for the right one. Each has exactly one pin for the sending and arriving events respectively. The edge set has just a single member, the summary message $m'$. The direction of this edge is selected by the modeller.

### 6.2 The Summary State pattern

In this pattern, a certain detailed level behaviour, described by a state machine, is abstracted into a state at the higher level. The similarity between this pattern and the Black Box pattern is striking and is almost an exact analogue. Similar to the structural pattern, we introduce the concept of a “glass state” to represent the set of states and transitions that are being abstracted as shown in the example in Fig. 12. And, we distinguish “cross-over” transitions, which cross in or out of the glass state and which have corresponding representatives in the abstract graph. All elements within the glass state except for the cross-over transitions and their triggers are mapped to a single abstract state $\text{AbsS}$.

The primary difference between this and its structural analogue is that, in con-
Contrast to connectors, transitions have a distinct trigger feature, which represents the event and, where applicable, guard condition that determine when a transition will be fired. In moving from the refinement to the abstraction, the triggers for cross-over transitions may also be raised to a higher level of abstraction, representing events and guards expressed in terms of the higher-level domain concepts (e.g., trigger $t_1$ in the refinement is represented by a corresponding high-level event $t_1'$ in the abstraction). As was the case with structural features, the nature of trigger abstraction is application specific and cannot be generalized.

The Summary State pattern is specified as follows:

**RefGraph:**
The nodes of this graph are the states enclosed within the glass state, that is, the set of states that are to be abstracted. The edges are the transitions, including the cross-over transitions that have at least one end outside the glass state boundary. Each transition has a trigger feature.

**Mappings:**
For each state of the refinement graph there exists a mapping to the single state $\text{AbsS}$ of the abstract graph.
For each transition within the glass box of the refinement graph, there is a mapping to the single state $\text{AbsS}$ of the abstract graph.
For each cross-over transition in the refinement graph, there exists a mapping to a corresponding unique transition with the same (incoming or outgoing) orienta-
tion in the abstraction graph. This mapping includes the appropriate transformation of the trigger feature of the cross-over transition.

**AbsGraph:**
There is a single node in this graph: the abstract state \( \text{AbsS} \). The edges consist of the collection of transitions corresponding one-to-one to the cross-over transitions, each with a trigger condition.

For state machine formalisms such as UML, which support hierarchical states (corresponding to structural containers), and state entry and exit points (corresponding to ports), there is also a direct analogue matching the type of Black Box abstraction mapping shown in Fig. 4.

### 6.3 The Group Transition pattern

This pattern is somewhat similar to the Cable structural pattern. In this case, a set of transitions emanating from different states but all having equal triggers and terminating on the same target state are abstracted into a single transition emanating from a single summary state that abstracts the original source states (Fig. 13). This pattern assumes application of the Summary State pattern for abstracting out the different source states in the refined view.

The pattern is specified as follows:

**RefGraph:**
The set of nodes of this graph is empty. The set of edges consists of the transitions selected to be abstracted. All of them must have the same value, \( t \), for their trigger features and all must terminate on the same target state.

**Mappings:**
For each transition in the refinement graph, there exists a mapping to the single transition \( e' \) in the abstraction graph. The mapping includes a mapping of the trigger feature to the corresponding high-level trigger.

**AbsGraph:**
The set of nodes of this graph is empty. The set of edges consist of a single transition, $e'$, which has a trigger, $t'$, that is an abstraction of the refined trigger $t$.

### 6.4 The Summary Activity pattern

This is yet another behavioural pattern that is reminiscent of the Black Box structural pattern. In this case, we are abstracting activities or actions as well as other types of nodes (e.g., decision nodes, fork nodes, join points) along with all the control and data flows that interconnect them into a single high-level summary activity with appropriately defined high-level pins. (Fig. 14).

![Diagram of Summary Activity pattern](image)

Figure 14. An example of the Summary Activity pattern

This pattern is defined by the triple:

**RefGraph:**
The set of nodes of this graph consists of the collection of all activities, actions, and other types of nodes enclosed by the glass activity (i.e., the set of nodes to be abstracted). The set of edges is defined by the set of flows that have at least one end within the glass activity.

**Mappings:**
For each node within the refinement graph, there exists a mapping to the single node in the abstraction graph, $AbsA$.
For each flow in the refinement graph that is not a cross-over flow, there exists a mapping to the single node in the abstraction graph, $AbsA$.
For each cross-over flow in the refinement graph, there exists a mapping to a pin on the abstraction graph. The direction of this pin, input or output, is based on whether the cross-over flow is incoming or outgoing respectively.
For each cross-over flow in the refinement graph, there exists a mapping to a corresponding flow in the abstraction graph, which is anchored to a corresponding pin on the abstract activity, $AbsA$. The orientation of this flow matches the orientation of its corresponding cross-over flow.

**AbsGraph:**
This graph consists of a single node, $AbsA$, with a corresponding pin (control or data, input or output) for each cross-over flow in the reference graph. Its

---

10) In UML these two concepts, although quite similar, are syntactically distinct; actions being the semantic equivalent of a leaf (i.e., atomic) activity.
11) We do not distinguish between data pins and control pins in this discussion, nor between control and data flows, and, furthermore, the term pin here subsumes the UML activity parameter concept.
edges comprise the set of flows corresponding one-to-one to the set of cross-over flows in the reference graph and with the appropriate orientation (incoming or outgoing) and connected to the appropriate pins on \textit{AbsA}.

7 Temporal Patterns

As some wit noted once, time is merely a device created to prevent everything from happening at once. In software as in many other engineering disciplines, the relentless nature of time can make things difficult for designers. Consequently, several different abstract representations of time and time-dependent behaviours have been devised to circumvent or reduce some of that complexity. An excellent discussion on different ways of representing time and the relationships between them can be found in Prof. Broy's paper on time refinement[5].

This group of abstraction patterns deals with a qualitatively different set of concepts that are not well suited to be represented as graphs. Consequently, they are not described using the graph mapping approach that was applied to the structural and behavioural patterns.

7.1 Time compression

This is a case of a quantitative rather than qualitative abstraction, whereby an interval of time at a lower level of abstraction is compressed into a single instant with no discernible duration (the so-called “zero-time assumption”). Any behaviour occurring within this interval is similarly compressed, although any causally-related events that occur in that interval are retained.

This pattern can simplify many formal analyses, but it should be used judiciously since it is only practical in situations where the duration of the compressed interval is insignificant relative to the duration of other behaviours occurring in the system. For example, the time taken to execute a fragment of computer code and generate an output in response to a trigger can be deemed instantaneous when compared against the rate of change of many physical phenomena.

This pattern should not be confused with the Logical Time pattern described below.

7.2 Logical time

In contrast to the Time Compression pattern, which is concerned with relative durations of time intervals, in the Logical Time pattern the notion of duration is abstracted away completely. This is applicable in cases where the only dynamic relationships of interest are cause-effect orderings. This pattern is used in many temporal logics[15].

7.3 Time discretization

Instead of viewing time as a continuous flow in which events can occur at any instant, in this pattern, time is represented as a series of discrete intervals of equal length. All events occurring within the same discrete interval are deemed to be simultaneous, which implies that there cannot be any causal relationships between them. Instead, the effects caused by those events are only perceived in subsequent intervals. Clearly, this pattern is only applicable if the granularity of the time discretization
is such that the inaccuracies caused by discretization are small and do not increase over time. At the very least, the interval should be short enough to ensure that the observable consequences of an event in a particular interval are insignificant within that interval.

7.4 Behaviour interleaving

While this pattern actually abstracts behaviour rather than time, since time discretization and duration play a prominent part in it, it is grouped here with other time-related patterns. As with Time Compression, it is primarily a consequence of the exceptional speed of electronic computation relative to other physical phenomena. Behaviour Interleaving is used to model physically concurrent behaviour on a sequential computer, representing the progress of concurrently executing behaviours as a series of alternations of discretized sequential behaviours. It has similarities with the Time Discretization pattern in that behaviours are sliced up into discrete intervals, although the intervals do not necessarily have to be of equal duration. Still, the discretization must be such that any potential causal couplings between the concurrent behaviours are respected and that the observed serialized behaviour is equivalent to an actual concurrent execution. Equivalence in this case implies not only equality in terms of observable events produced by the concurrent behaviours but, at least in some cases, in terms of durations as well.

Behaviour Interleaving is used by modern operating systems to create the illusion of concurrency primarily for reasons of efficiency. However, in model-based engineering its primary purpose is to support formal analysis and simulation. Specifically, it is used to create a more abstract model of a physically concurrent system.

8 Summary

The catalogue of patterns described in this article is certainly not complete, nor was that the intent. The primary objective of this article is to introduce the topic and to propose an approach to how such patterns can be described more precisely. Three basic categories of patterns were introduced, based on the nature of the modeled phenomena: structural patterns, behavioural patterns, and temporal patterns. As with design patterns in general, one obvious benefit of specifying abstraction in pattern format is an explicit recognition and clear description of common and proven design practices. But, in addition to identifying and describing the patterns to serve as a reference in design, it is suggested here that a precise definition of the patterns, utilizing graph transformation or similar formalisms which can then be realized on a computer, can provide some significant advantages. These benefits include formal traceability between models at different levels of abstraction or the ability to automatically regenerate an abstract model by extracting it from the concrete one.

Future work includes a more precise, that is, a formal, specification of these patterns as well as extending the catalogue with further patterns and even additional categories.

References


