Transforming User Experience Model To Presentation Layer Implementations

Wojtek Kozaczynski (wojtek@rational.com)
Jim Thario (jim.thario@rational.com)
Rational Software Corporation, USA

Abstract

The paper presents a simple modeling language for describing technical aspects of interactions between users and the enterprise, and in particular Web, applications. Such applications use screen-based user interfaces and commonly utilize a framework, like the Jakarta Struts or Microsoft ASP.NET, to implement those interfaces.

The presented language is defined by a UML profile and can be easily implemented by a UML modeling tool. More interesting however, models described in the language can be automatically, or semi-automatically, transformed into an implementation of the interface on top of a selected framework. We describe both in the paper.

Introduction

Our visual language is referred to as the User-Experience (UX) Modeling Language. It is defined by a UML profile that captures concepts used for representing the engineering aspects of screen-based user interfaces concerned with:

- The screens the user sees
- The actions the user can perform on the screens, like clicking a button, that the system must react to (handle)
- The dynamic data the system must display (or put) on the screens
- The data the user puts on the screen that the system must gain access to
- The decomposition of screens into smaller areas that should be managed separately from other areas, and
- Screen transitions caused by user's actions and application's reactions.

The engineering aspects of user interfaces are not concerned with the creative aspects such as use of colors, fonts, layout, statically supplied graphics, etc.

The paper has the following structure. First we discuss user experience modeling and describe the UML profile of the language. Then we give a short overview of the user experience modeling guidelines and show fragments of UX models. We conclude with a discussion of the transformations of UX models into code-level models for the Jakarta Struts framework.

We also demonstrate the language and model transformations are an instance of Model-Driven Development and as such, an example of an automated development of enterprise applications.

User-Experience (UX) Modeling

The Rational Unified Process defines UX Models as follows: "The UX Model describes the screens of the system, the dynamic content that appears on the screens, and how the user navigates through the screens to execute the system functionality (i.e., the use cases)." [1]

The following participants of the software development effort use UX models most:

- User-experience designers, who prototype the user interface in order to validate the user interactions with the system
- Designers of the presentation logic system elements, who want to understand how screens participate in use cases, so they can design and implement the presentation logic elements (e.g., web pages, applets, etc.) that will implement the screens
- Designers of the business logic system elements, who must understand what business content is required by the screens and must be provided by the business logic elements
- Testers, to test the systems' use cases [2]

A UX model is a visual representation of the screens, screen transitions, dynamic screen data and user-initiated input events that need to be handled by the system. A UX model is a diagram composed of the elements defined in the UX UML profile. A UML profile is a collection of metamodel elements, which have been customized for a specific domain. UML metamodel elements can have their semantics customized using stereotypes, tagged value definitions and constraints.

Below we describe the primary modeling elements introduced in the UX UML profile.

Stereotypes[1]:

Name	Extends	Description
screen	Class	An abstraction of a rendered web page in a client browser. Represents the final visualized user interface presented to the user of the system.
compartment	Class	Represents a well-defined region of a screen that is reused by multiple screens. It is an abstraction of a part of a rendered web page in a client browser.
input form	Class	Represents a group of data fields that can be displayed to and manipulated by the user
form	Attribute	Indicates that an attribute (field) on a screen is associated with an input form (it is an input form field). See tag values of those attributes.

[1] "A new type of modeling element that extends the semantics of the meta-model. Stereotypes must be based on certain existing types or classes in the meta-model. Stereotypes may extend the semantics, but not the structure of pre-existing types and classes. Certain stereotypes are predefined in the UML, others may be user defined." [3]

storyboard	Package	Designates a package containing a description of a use-case storyboard.

Tagged Value Sets[2]:

Attached to Metamodel Element: Transition			
Tagged Value Name	**Type**	**Default Value**	**Description**
System Action	String		One or more names of the "business domain" components of the system that will participate in handing the transition. This is both a bleed-through from the design and "horizontal" link to the business-logic modeling domain.

Attached to Metamodel Element: «form» Attribute			
Tagged Value Set Name: FormAttribute			
Tagged Value Name	**Type**	**Default Value**	**Description**
Editable	Boolean	True	Associated with «form» attributes/fields on a screen Indicates if the field can be edited by the user (True) on not (False).
Source Form	String		Associated with «form» attributes/fields on a screen. Contains the name of the form that the field comes from.
Field Visibility	String: 1 – Visible 2 – Hidden	Visible	Associated with «form» attributes/fields on a screen. Indicates if the attribute is displayed on the form or hidden.

Constraints[3]:

Constrains are checked before transforming a UX model to an Implementation model to assure its well-formedness. The constraints check for the model properties described in the UX modeling guidelines discussed later.

Below we give examples of three constraints defined in the UX UML profile[4]:

Constraint:

context «Storyboard»Package **inv** Storyboard Package Content:

‘ if the context package is stereotyped «storyboard», it must contain a class diagram called "Participants" and a state machine with the same name as the

[2] "A name-value pair. In a tagged value, the name is referred to as the tag. Certain tags are predefined in the UML; others may be user defined." [3]

[3] "A semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined." [3]

[4] Here the constraints are described in English. There is an OCL version of the constraints, but they would be hard to explain without additional meta-model discussion.

name of the package. The class diagram shows the UX elements participating in the use-case storyboard and the use-case screen transitions. The state machine shows the specific distinct states of the use case and refines (disambiguates) the flows of the use case. The name of the main state machine diagram must be equal to the name of the storyboard package.'

Constraint:

context Association **inv** Screen Transition End Names:

' if the context directed association represents a transition from screen (or compartment) A to screen (or compartment) B then (a) the name of provider-end (the A screen in our case) of the association should be equal to a name of one of the operations on screen A AND (b) the association must have a name. The consumer end name must be empty.'

Constraint:

context Transition **inv** Screen Transition Source and Target:

'the source of the context transition can be a screen, a compartment or an input-form. The target of a transition can be only a screen or a compartment'

User-Experience Modeling Guidelines

Profiles describe modeling elements that should be used while describing a specific aspect of an application and the syntactic rules of well-formed models. They don't tell the developer, however, how to construct a meaningful and an easy-to-read model. This is done in the modeling guidelines. Below are a few excerpts from the UX modeling guidelines:

The primary UX modeling elements are:

- Screens (represented by classes stereotyped as «screen»)
- Screen compartments (represented by classes stereotyped as «compartment»)
- Input forms and (represented by classes stereotyped as «input form»), and
- Data structures (represented by classes with no stereotypes)

An attribute on a screen or a screen compartment can be stereotyped as «form» as shown in Figure 1. The «form» stereotype indicates that the screen field represented by the attribute comes from (is a part of) an input form. Each such field has two tagged-vale attributes that contextualize the use of the field on the form:

1. *Editable*, which is set to "True" if the field can be edited by the user and to "False" if the field cannot be edited
2. *Source Form*, which contains the name of the form the field belongs to (this is in case there are multiple fields with the same name that come from different forms).

An attribute without a stereotype represents a screen field, which output "directly" on a screen.

Operations can be placed on screens, screen compartments and forms. An operation represents an action that the user can initiate and the system has to handle.

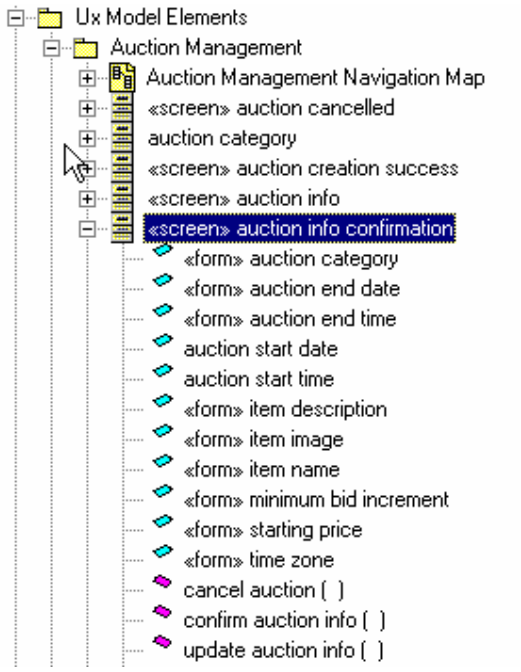

Figure 1. Explorer view of a Screen description

Screen compartments can be used everywhere screens can be used. A screen compartment is always a "part" of a screen, which is represented as a composition as shown in Figure 2.

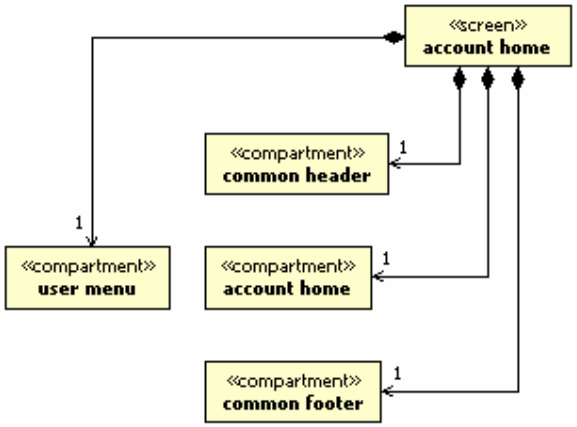

Figure 2. Example of a Screen containing Screen Compartments

A storyboard is a description of the screen flow of a use case. A storyboard is represented by (described in) a package stereotyped as «storyboard» as shown in Figure 3.

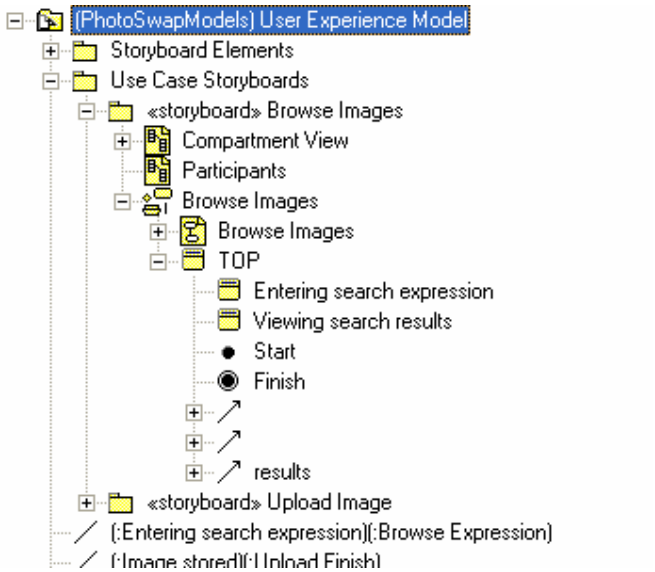

Figure 3. Example of a storyboard package

Placing the storyboard information in a package is a modeling requirement, as the convention is used during model transformations. A storyboard package must contain two elements:

1. A class diagram called Participants showing screen-flow dependencies between the UX model elements participating in the storyboard, and
2. A state machine named exactly like the storyboard package. The state machine refines the flow of the storyboard and introduces distinct, named states of that flow.

A participant diagram is shown in Figure 4. A directed association between two screens or an input form and a screen indicates that there is a flow from the "from" screen/form to the "to" screen initiated by an action preformed by the user on the "from" screen/form. There cannot be flows to forms or compartments, but only to screens.

In an association the name of the "from" end must be the same as the name of one of the screen operations. Each flow is also uniquely named. Since it is not common to model both an association name and a role name, it is worth describing the need for both. The role name matches user actions, which makes it easy to see the outgoing transitions resulting from a particular user action. The same user action can cause multiple transitions (outgoing associations) with the same role name (see the "submit" action of the "auction info form" in Figure 4). The different/unique association names allow us to distinguish these transitions and are also used on the state diagrams described in the next section.

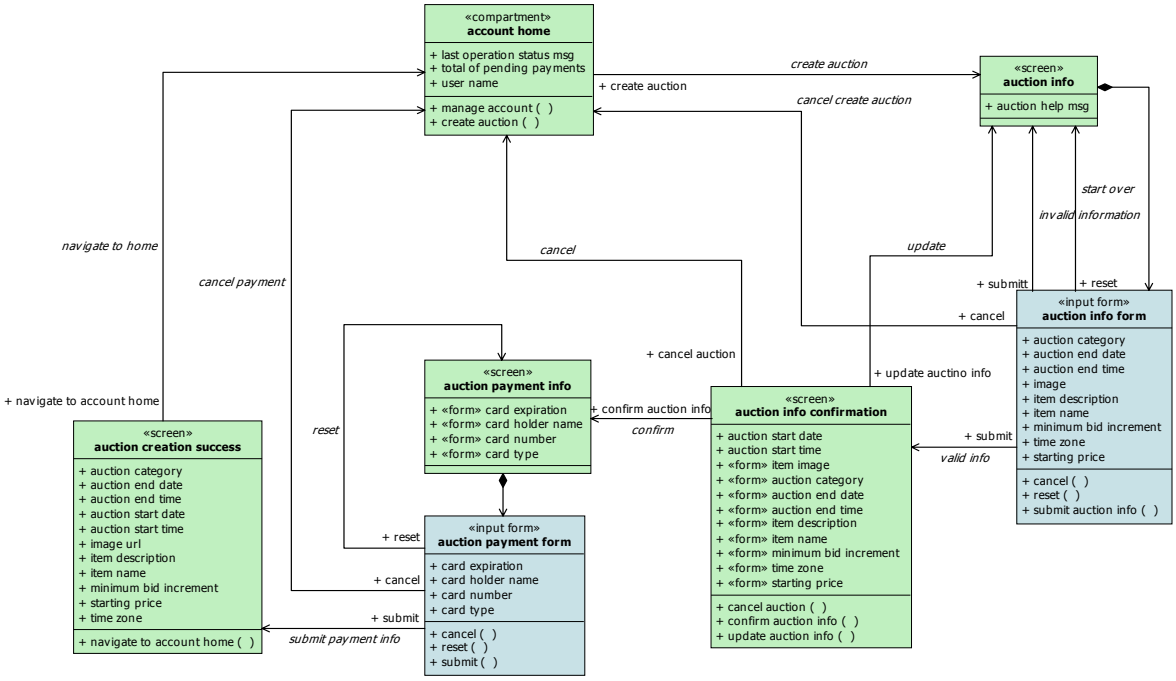

Figure 4. Example of a use-case storyboard

A state machine associated with a storyboard complements and refines the description of the flow of a use case. The state machine associated with the storyboard shown in Figure 4 is shown in Figure 5. Below are the highlights of the state machine representation:

- Each state is associated with at least one screen and represents the state of the flow at the time the user initiates an action that resulted in a transition
- Each state transition, except a transition to a choice point (see more about choice points below), has the same name as the corresponding flow association in the participants diagram
- The standard UML tagged-value *Action* on a transition gives a name to the system action that supports (or handles) the transition
- The UX *Change Event* associated with a transition gives the name of the screen operation that represents the user action that initiated the transition
- A transition from a state to a choice point has no name. However, the transitions from a choice point have names of the corresponding screen-flow associations in the participant diagram of the storyboard. A choice point disambiguates alternative flows initiated by the same user action.

Transformations of UX Models to Implementation Models

UX models can be directly transformed into implementation models that in turn can be used to generate code. The UX-to-implementation model transformations illustrate the conceptual framework of Model-Driven Development (MDD). The framework

provides a set of concepts that can be used to think about the artifacts and activities of complex software systems development. The framework is based on three principles:

- The principle of Separation of Concerns
- The principle of Abstraction and Refinement, and
- The principle of Reuse.

Although these principles a strongly interrelated, we will attempt to describe them one at a time.

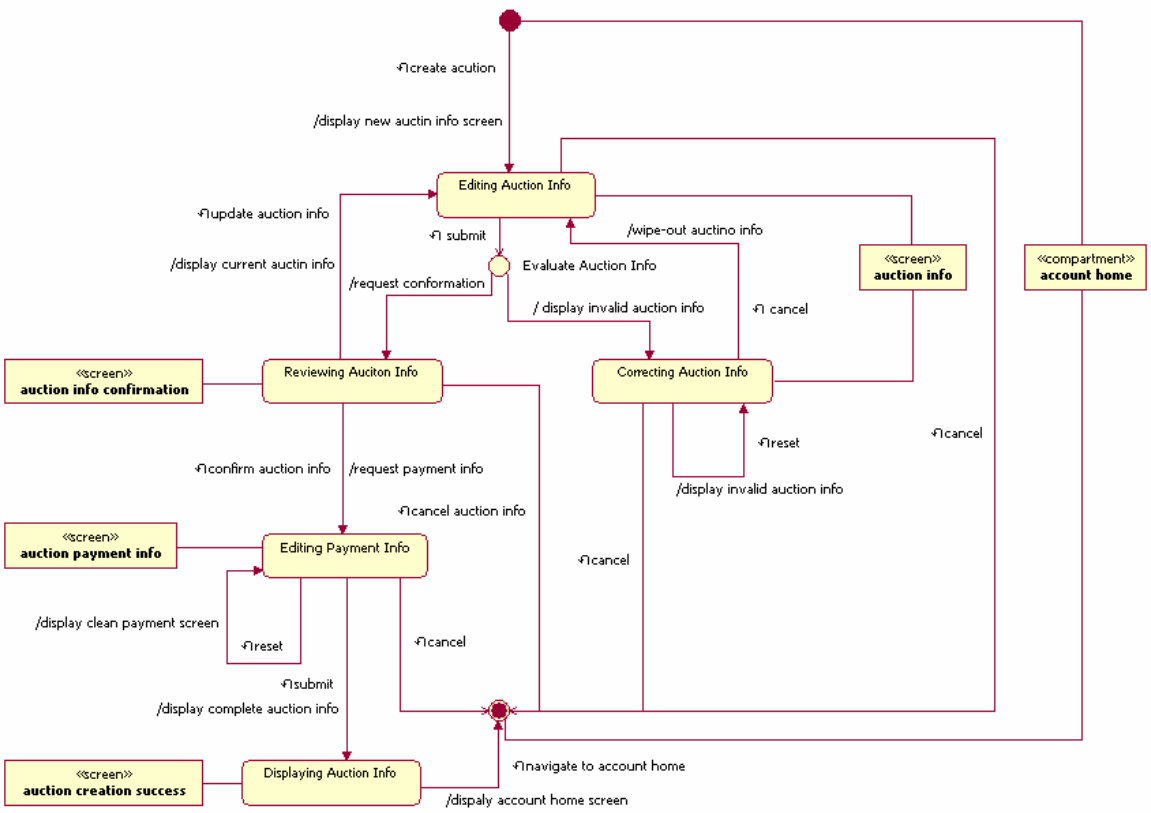

Figure 5. Example of a state machine associated with a use-case storyboard

Separation of Concerns

A complex software system has a number of "aspects" that have to be considered and designed for. These aspects are not equally important to all stakeholders. Stakeholders look at a system from different "viewpoints" and may be interested in seeing only those aspects that concern them the most.

UX models are an example of a system view that addresses the engineering aspects of the interactions of the user with the system.

Abstraction and Refinement

The process of system design and development is a process of transforming a set of domain (or business) requirements into working code (the system) that meets those requirements. Practice shows that it is very hard, if not impossible, to map domain

requirements directly into working code. Instead, we progressively map the requirements into sets of more and more specific concepts starting from concepts that bridge the requirements and design spaces and ending with concepts representing the programming model and the target platform. The progressive mapping of more abstract to more concrete concepts is usually referred to as refinements.

Transformations of use-case descriptions and user scenarios into UX models and then transformation of those models to implementation models are examples of refinement transformations.

Reuse

It is recognized, that best system designs are based on proven solution patterns. The MDD framework assumes that separation of concerns as well as the abstractions and refinements should naturally lend themselves to capturing and reuse of best design experiences.

A specific set of transformations that take a UX model and generate a Java model for the Jakarta Struts framework is an example of encoding a design style (a collection of design principles) in a set of design patterns.

Transformations between models capture particular design styles. A design style is a prescribed way of transforming structures of a higher-level model into the structures of the lower level model(s) while taking into account the system requirements[5].

Our set of UX-model-to-Java-model transformations captures a particular style of designing and implementing the web presentation layer using the Jakarta Struts 1.0.2 framework. The rules of the style are as follows:

- Transform each use-case storyboard into a subclass of the Struts Action class. Give the Action class the same name as the use case. Correct the name so that it is a Java compatible class name.
- Transform each UX screen into a Java Server Page. Give the JSP the same name as the UX screen. Correct the name so that it is compatible with the file system naming requirements.
- Transform the set of attributes on each UX screen into a value object with get/set methods for each attribute of the screen.
- Transform each operation on the UX screen into an HTML anchor on the Java Server Page with the target of the anchor being the UX screen on the opposite end of the directed association.
- Transform each UX compartment into a Java Server Page. Give the JSP the same name as the UX compartment. Correct the name so that it is compatible with the file system naming requirements.

---

[5] Our definition of design styles is similar to the definition of architecture styles: "An architectural style defines a family of software systems in terms of their structural organization. An architectural style expresses components and the relationships between them, with the constraints of their application, and the associated composition and design rules for their construction." [4].

- Transform the set of attributes on each UX compartment into a value object with get/set methods for each attribute of the compartment.
- Transform each UX input form into a Java Server Page. Give the JSP the same name as the UX compartment. Correct the name so that it is compatible with the file system naming requirements.
- Create an HTML form in the JSP with an input field and label for each attribute on the UX input form.
- Transform the set of attributes of each UX input form into a subclass of the Struts ActionForm class.
- Transform each state in the storyboard state machine into a handler method on the storyboard Action class. Implement the Action perform() method so that it routes control to the correct handler method based on the known state of the storyboard.

Automating Transformations with Modeling Tools

The transformations described above can be either performed by hand in an IDE or code editor, or they can be automated with a modeling tool. Below we briefly discuss how we use Rational XDE (a new generation of Rational modeling and code generation tool) to automate the transformations.

The XDE has a facility to define and apply model templates called the Pattern Engine. The developer applies a transformation pattern by providing arguments for each template parameter and a target expansion location. In our case, each time the developer applies a UX transformation, he/she provides elements from the UX model as arguments and chooses a location in an implementation model for the expansion. As a result, information from a UX model is used to generate fragments of implementation model(s).

Even though this approach provides large gains in productivity, it can become tedious when transforming large UX models or even transforming the same model multiple times after modifications. To help alleviate this issue, the authors wrote an XDE add-in that applies the transformations automatically. The add-in analyzes a UX model, selects appropriate transformations and controls their application. In other words, the add-in uses the individual patterns as atomic transformation units and executes a design strategy we discussed earlier.

To use the add-in the user first creates a UX model following the guidelines discussed earlier. Then she/he selects the root of that model and the root of each implementation model. In common cases there will be a model for Java classes and a model for web elements such as Java Server Pages. From here the add-in is able to navigate the model, find the appropriate UX model elements and perform a complete (hands-free) creation of the implementation models with classes and web elements.

Author's Experience with UX Transformations

Early users of UX models found them bit overcomplicated. In particular, the state diagrams provide information that can be inferred, with some additional information,

from the participants diagrams. Hence, the next version of the UX profile will most likely extended the participant diagram semantics, but will have no state diagrams.

One of the requested domain extensions is access control. UX models can be enhanced to describe aspects of application security by tagging screens with access control information and then this information can be passed to the code via templates.

Although the described transformations can save quite a bit of work, the developers still need to hand-code the connection from the generated code to data and business services of the enterprise. Generation of complete, executable code brings us back into the MDD discussion. In order to do that, the enterprise business services would have to be described by a related UML profile and a third profile may have to be created to "connect" the two specifications.

After writing the described transformations, we wrote the transformation driver (or engine) using the "brute force" of coding it in a programming language. This makes changes to the driver difficult. Describing the transformations steps using "transformation" UML profile would be a useful leap.

Finally, as soon as we showed the forward application of UX-to-Struts transformations, the dreaded question about reverse engineering came up.

Conclusion

In the paper we have described a simple visual modeling language based on UML. The language has no name, but the domain it is used for has. It is called User-Experience (UX) Modeling and is used to describe the technology aspects of the user's interaction with a Web-based system.

Both the language and the transformations of the UX models into code-level models for the Jakarta Struts framework are an example of what is commonly referred to as Model-Driven Development (MDD). In MDD an application is described in a number of problem (or domain) oriented "small" languages capturing specific aspects of the application. The UX modeling language is one of them. Other common specification languages include data modeling language, workflow modeling language or business components modeling language.

References

[1] Rational Software Corporation, *Rational Unified Process®,* http://www.rational.com/products/rup, 2002
[2] Rational Software Corporation, *RUP® Configuration for Java™ Developers*, Rational XDE Professional v2002 Release 2 — Java Platform Edition Kit, http://www.rational.net, 2002
[3] Object Management Group, *Unified Modeling Language Specification*, Version 1.4, http://www.omg.org/uml, 2001

[4] Buschmann, Meunier, Rohnert, Sommerlad, Stal, *A System of Patterns* (England, West Sussex: John Wiley & Sons, Ltd., 1996)