A Semantic Approach
To User Interface Design

Mordechai Porat
A Semantic Approach
To User Interface Design

Project Thesis

In Partial Fulfillment of The Requirements for the Degree of
Master of Science in Computer Science

Mordechai Porat

Submitted to the Senate of
The Technion - Israel Institute of Technology

Kislev 5773 Haifa December 2012
The Project Thesis Was Done Under The Supervision of Prof. Eliezer Kantorowitz in the Faculty of Computer Science.

Dedicated to Prof. Eliezer Kantorowitz, with great appreciation for his endless encouragement, caring, and all our fascinating talks.

# Table of Contents

Abstract ................................................. 1

1. Introduction .......................................... 2
    1.1 Interaction Design Patterns .................. 2
    1.2 State of the Art GUI Development .......... 3
    1.3 Semantic GUI Programming ................... 6

2. The WebSIX Framework ................................. 8
    2.1 General Structure and Terms ................. 8
    2.2 AL ............................................. 9
    2.3 The Semantic Tree and Semels ............... 12
    2.4 The Three Stages of Transformation ....... 12

3. Implementation ....................................... 14
    3.1 General Flow .................................. 14
        3.1.1 Environment Classes ................. 14
        3.1.2 The Activation Class ................. 16
        3.1.3 Object States and Lifetime .......... 16
        3.1.4 Notes about Overhead ............... 17
    3.2 AL Classes .................................... 18
        3.2.1 Application and UseCaseBasedApplication . 18
        3.2.2 Semantic Control Classes ............. 18
    3.3 IS Classes - Transformers ................. 19
        3.3.1 Loading Transformers ................. 20
        3.3.2 Composite Transformers ............... 21
        3.3.3 Basic Transformers ................... 22
        3.3.4 HtmlMaker .................................. 23
Table of Contents (continued)

4. Examples of Interaction Design Patterns . . . 24
   4.1  Global Structure . . . . . 24
   4.2  Global Navigation . . . . . 26
   4.3  Tree Layout . . . . . . . 27
   4.4  Preview Pane . . . . . . . 28
   4.5  Row Striping . . . . . . . 30
   4.6  Form Alignment . . . . . . . 30
   4.7  Titled Sections . . . . . . . 32
   4.8  Error Message Near Input Field . . . . 32
   4.9  Example Applications . . . . . . . 34

5. Evaluation . . . . . . . . . 37
   5.1  IDP's and Their Requirements from the AL . . . 37
   5.2  Transformer dependencies . . . . 38
   5.3  Difficulties Using WebSIX . . . . 39
   5.4  Considerations in Designing WebSIX . . . . 40
   5.5  Suggested Future Work . . . . . . . 41

6. Related Works . . . . . . . . . 42
   6.1  Pattern Collections and Languages . . . . 42
   6.2  GUI Generation and the Multiplatform Problem . 43
   6.3  State-of-the-art GUI Development Techniques . . 45

7. Conclusions . . . . . . . . . . . 47
List of Figures

Fig. 1: Hand-written GUI's in Swing . . . . 3
Fig. 2: The general flow of WebSIX . . . . 9
Fig. 3: A possible resulting GUI for the code in Listing 2 . 10
Fig. 4: The data flow during one action . . . . 15
Fig. 5: Two ways to implement the main transformer of the IS . 22
Fig. 6: Two possible layouts for a two-level navigation tree . 27
Fig. 7. Two styles of Form Alignment . . . . 31
Fig. 8: The Chess application, saving a riddle . . . . 33
Fig. 9: The Riders application . . . . . 35
Fig. 10: The Chess application, playing page . . . . 36
Fig. 11: The Chess application, loading a riddle . . . . 36

Code Listings

Listing 1: Java code for the Swing input panel . . . . 4
Listing 2: The WebSIX AL code for the Celsius/Fahrenheit program 10
Listing 3: The semantic tree resulting from the code in Listing 2. 11
Listing 4: Outline of a typical IS definition . . . . 20
Listing 5: The elements added by GlobalStructure . . 25
Listing 6: Extracting the preview into a preview pane . . 29
Listing 7: The transformation performed by FormAlignment . 31
Listing 8: The AL code of Riders's global structure. . . . 34

Tables

Table 1: Object States and Lifetime . . . . . 17
Table 2: IDP's and their requirements from the AL . . . 37
Abstract

It is sometimes required to run the same application program on different platforms, from smartphones to PC's. The graphical user interface (GUI) designed for use of the application on a large PC screen may need a redesign in order to fit to a small smartphone screen. In addition, different applications running on the same platform may employ different GUI's to meet different requirements of the different applications and of different user populations.

Industrial experience and behavioral research have resulted in a number of published recommendations and conventions for designing GUI of different applications on different platforms. The purpose of these recommendations is to promote the design of GUI's providing a satisfactory user experience (UX; this includes usability and aesthetics). These recommendations and conventions are specified by design rules and Interaction Design Patterns (IDP's). This project has resulted in a novel method for generating GUI's that adhere to specified IDP's. The input to the method is a semantically specified GUI of the kind introduced by Kantorowitz and Lyakas. In their method, the code of the application, called Application Logic code (AL code), specifies its user interaction semantically, i.e. by what is to be done and not by the employed GUI components. For example, the semantic control "select one of $n$ choices" may on one platform correspond to a menu and on another platform to a group of $n$ radio buttons. Another component, called Interaction Style code (IS code), converts the semantically specified user interface to a real-life ("concrete") GUI.

The method introduced in this work transforms the semantic GUI into a concrete GUI by employing a number of customizable transformation rules. The customization enables adaptation to a specific platform and to specified IDP's. Experiments with the method suggest its applicability. The produced GUI's require, however, some polishing, for which further research is required.

List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Application Logic</td>
</tr>
<tr>
<td>BL</td>
<td>Business Logic</td>
</tr>
<tr>
<td>BSC</td>
<td>Basic Semantic Control</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphic User Interface</td>
</tr>
<tr>
<td>IDP</td>
<td>Interaction Design Pattern</td>
</tr>
<tr>
<td>IS</td>
<td>Interaction Style</td>
</tr>
<tr>
<td>Semel</td>
<td>Semantic Element</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface (when not regarding the graphic aspect)</td>
</tr>
<tr>
<td>UX</td>
<td>User Experience</td>
</tr>
</tbody>
</table>
1. Introduction

This study regards interactive application programs, i.e. applications controlled by human end users. Interactive applications are important as they enable solving difficult problems by combining human intuition with the power of computers. Furthermore, interactive applications with friendly user interfaces enable computer utilization by non-programmers. The design and implementation of the user interfaces of interactive applications may, however, be both difficult and labor intensive. A considerable effort has therefore been invested in developing tools and methods for alleviating the design difficulties and development costs.

Design of Graphical user Interfaces (GUI's) is difficult because many different, sometimes conflicting, factors must be met. For example, the business need to show more information or suggested actions, versus the users' need for simplicity. The designer should take into account the experience, skills, and expectations of the target users, as well as natural aspects of users' behavior: in which order our eyes scan the page, how the user explores the application or searches for ways to perform tasks, and much more [6, 13]. Constructing a GUI with satisfactory User Experience (UX) is somewhat an art, but a good approach would be to use common and proven patterns.

1.1 Interaction Design Patterns

An Interaction Design Pattern (IDP) is a recommendation of a piece of GUI designed for a particular kind of user interaction, e.g. entering complex input, initiating an action, or seeing the position of the current page within the entire application. IDP's appear in all scales, from aesthetics advices, to choices of specific GUI elements (controls), up to IDP's that govern the high-level structure of the GUI and the flow of operations. Broad literature and collections of IDP's, e.g. [13, 14, 18], were gathered over the last 15 years, based on analysis of real-life applications and websites. ([7] estimates that about 750 IDP's are available today).

Each IDP specifies the context and reasons in which it applies, and the solution – the piece of concrete GUI to be employed, its layout and behavior. This specification is written in a natural language and in a high-enough level so that it is not bound to any specific GUI technology. Therefore, the IDP collections are meant for programmers and designers to learn and implement manually.

The main purpose of this project is to develop a method for automatically producing GUI's that adheres to a specified set of IDP's.

In order to facilitate this, we have set two goals:

1. Experience with the implementation of IDP's as reusable code. In order to be reusable, they must be independent of the application that uses these IDP's. We choose to apply them over a semantic UI model (explained below).

2. Make it as easy as possible to combine a set of IDP implementations in order to devise the entire GUI of an application.

These attempts should yield better understanding of what properties should be in a semantic UI in order to apply some IDP's on it. More importantly, the result shall be a library of IDP's, where for each one the "how to implement" section will be represented by its actual implementation instead of textual prose.
1.2 State of the Art GUI Development

GUI development costs may be reduced by employing libraries of reusable software component for construction of GUI's. Such a library typically includes components for construction of controls (widgets), e.g. buttons and menus. Further tools are provided for managing the layout of the controls on the computer screen. In addition, "listeners" are provided for detecting user events such as mouse clicks and keyboard hits, and initiating the corresponding actions. The construction of a GUI using such a library can, however, still be quite labor intensive, as there are many design details that have to fit to each other.

The advents of new different kinds of platforms, such as smartphones and tablets, enhanced the need of methods for porting applications from one platform to another, i.e. reusing the same application on different platforms. Such porting is facilitated by "write once run everywhere" languages such as Java. The use of such languages did however not solve all application porting problems. Consider, for example, an application written for a PC, where the piece of GUI enabling the end user to select one of \( n \) given choices is implemented by \( n \) large legible radio buttons (Fig. 1 (a)). If this program is ported to a smartphone, the \( n \) large radio buttons may require more space than is available on the small smartphone screen. One possible solution is to implement the same activity by a space-saving dropdown menu. There will thereafter be two different versions of the application – a PC version and a smartphone version. The code employed for the select operation in the PC versions, that produces large radio buttons, is shown in Listing 1, and the places that have to change in order to use the smaller dropdown menu, are marked. The code for handling the layout is omitted.

It is observed that the code of the two solutions is quite different. Notice in addition that the application code and the GUI are interwoven in each other. Exchanging the radio button code of Fig. 1 (a) with the dropdown menu code of Fig. 1 (b), is therefore quite intricate and requires the skills of an able programmer.

![Fig. 1: Hand-written GUI's for a Swing demo application for converting between Celsius & Fahrenheit degrees. The user chooses the kind of conversion by radio buttons in (a), and by a dropdown menu in (b).](image-url)
Listing 1: Java code for the input panel that is shown in Fig. 1 (a)

Green – Application Logic code (prompt, IO, calculations, and flow)
The rest – GUI construction and manipulation code.
Arrows – Places that need to change for the small GUI (dropdown menu).

private void createAskingPanel() {
    askingPanel = new JPanel();
    // CONTROLS
    JLabel lblDegrees = new JLabel("Degrees: ");
    inputDegrees = new JTextField();
    optionCtoF = new JRadioButton("Celsius to Fahrenheit");
    optionFtoC = new JRadioButton("Fahrenheit to Celsius");
    optionCtoF.setSelected(true);
    JButton btnConvert = new JButton("Convert");
    btnConvert.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            doConvert();
        }
    });
    // LAYOUT
    ...
}

private void doConvert() {
    String degreeStr = inputDegrees.getText();
    boolean cToF = optionCtoF.isSelected();
    double d;
    try {
        d = Double.parseDouble(degreeStr);
    } catch (Exception ex) {
        //Stay on the same page and move the cursor to the input field.
        inputDegrees.selectAll();
        inputDegrees.grabFocus();
        return;
    }
    String answer = cToF ?
            (d + "° C = " + calculateFahrenheit(d) + "° F") :
            (d + "° F = " + calculateCelsius(d) + "° C");

    //Prepare and show the answer page
    answerLabel.setText(answer);
    mainWindow.setContentPane(answerPanel);
    mainWindow.validate();
}

private double calculateFahrenheit(double celsius) {
    return celsius*1.8 + 32;
}

private double calculateCelsius(double fahr) {
    return (faht - 32) / 1.8;
}
The introduction of the Internet enabled a new kind of interactive applications, where a single application server interacts with numerous different clients residing on different platforms. For example, a bank may use such a system to enable its customers to manipulate their bank accounts from their home computers. In this technology the GUI is specified by the standardized Hypertext Markup Language (HTML). The client side is implemented by a Web browser which produces the GUI from its HTML specification and manages the client side of the communication with the application running on the server. Web browser clients are found on PC's, tablets, and smartphones. The GUI employed on a PC must sometimes be modified to fit into a small smartphone screen, from the same reasons that were mentioned above. In order to implement such two different GUI's, two separate HTML's must be employed for the PC and the smartphone. The application code that interacts with the two different platforms must also be adapted to generate and correspond with the two different HTML's. The result in this case is having both a PC version and a smartphone version of the HTML and of the application code.

1.2.1 Business Logic vs. Application Logic

The last claim, that the application code must adapt to a specific GUI, might seem odd. After all, it is an elementary practice to separate the system's pure logic layer from the GUI implementation. The core functionality, such as data structures, database access, and calculations, does not depend on the GUI. These tasks comprise the system's Business Logic (BL).

However, the GUI implementation also entails a kind of logic of its own, which we call the Application Logic (AL). The AL "sews" the BL to the GUI. For example, in the program in Listing 1, the BL is composed of the methods calculateCelsius and calculateFahrenheit. Their signature reflects their meaning: each one of them takes a number and returns a number. The AL, on the other hand, features the method doConvert, which has an empty signature. Instead of arguments, it uses the controls on the GUI for reading the input and showing the output, and it should also take care of possible input errors.

Some decisions that the AL layer should make are:

- Phrasing of text displayed to the user.
- Which BL actions to offer to the user at which stage.
- What to do after a successful action? and after an error? e.g. return to the main page of the application, or show a special page with the result/message.

Since the AL layer connects the BL and the GUI, it depends on both of them: on the BL, of course, because it must provide the means to invoke the BL actions. And, as demonstrated above, the AL code depends on the types of controls employed by the GUI.

This work takes a semantic approach, meaning that the AL interacts with a semantic UI model that is independent of any concrete GUI. Therefore we must generate the final concrete GUI automatically from the AL, and this is done with the help of IDP's.
1.3 Semantic GUI Programming

Having a number of different versions of an application in order to convey the same logic to a number of different platforms, is not an ideal solution. Each bug and each application extension must be done in each one of the versions. In order to avoid the need of having a separate version of an application for each platform, Kantorowitz and Lyakas [4] suggested a framework, called WebSI, where the same single application code runs unchanged on all platforms for which WebSI was designed. Each application has thus only one version of Application Logic (AL) code, and all bug corrections and application modifications are done in this single piece of code. The AL code has no specification of the user interface details at all. All interactions with the end user are specified by calling appropriate methods such as $\text{Select}.\text{one}$ for selecting one of $n$ choices and $\text{Output}.\text{scalar}$ for presenting a text to the user. These methods do not specify anything about the GUI controls employed (color, size, fonts, and even the type of controls). In other words, the interactions of the AL code with the end user are specified semantically – by what to do, but not how to do it. A separate code called Interaction Style (IS) handles the creation of the appropriate real-life controls. The IS code is designed for a specific platform, e.g. a PC with a standard Web browser. One or more IS codes may be developed for the same platform for implementing different user-computer interaction styles.

An application in the WebSI architecture will thus be composed of an AL code component and an IS code component. If the same AL code is to be employed in $n$ different platforms, all the IS codes of these platforms should provide the same IS interface, such that the AL code may readily interface to any of the IS codes.

The approach demonstrated by the WebSI system has a number of advantages over state of the art systems:

- The developer of the AL code need not know anything about the platforms where the application will run and the kind of GUI employed in each one of them.
- The AL code is easier to write than state of the art application code where all the details of the GUI are specified, i.e. the employed GUI components, their layout on the screen, their sizes, colors, and so on.
- The developer of the IS code is a platform specialist; he need not know the applications that will run on top of it, but only provide the expected IS interface.
- The same single AL code is deployed in all platforms. The same IS code serves all applications.
- All bug corrections and application modifications are done in the single AL code.
- An interactive application is produced by combining an AL code together with an appropriate IS code.

Weakness of the approach:

- The semantical invocations of the IS code through the IS interface, is an indirect way of addressing the real life user interface. This contrasts with state of the art methods, where an easier-to-manipulate real-life GUI is employed.
In the next chapter—"The WebSIX Framework"—we discuss the method developed in this project for construction of GUI's that adhere to a specified set of IDP's. We have decided that the starting point (the input) for such a method is a semantically specified user interface and not a state of the art user interface, such as an HTML. (that is, we do not convert, for example, a given PC GUI into a smartphone GUI). We felt that transforming an abstract UI into a concrete GUI is more interesting, and probably easier, than transforming one concrete GUI to another, because the semantic AL code specifies the UI in a simpler way than that employed in state of the art GUI code.
2. The WebSIX Framework

The WebSIX framework was developed in this project. WebSIX stands for Web Simple Interfacing by XML manipulations.

WebSIX employs the general structure of WebSI [4] and the two main terms that it had defined, namely Application Logic (AL) and Interaction Style (IS), which are explained below. WebSIX focuses on methods for developing the IS layer; it employs a novel method to facilitate the transformation of the semantic UI specified by the AL code. In this method the IS code applies a number of rules for transforming the semantic user interface into a concrete GUI.

The goals of WebSIX are:
- To build the IS as a composition of simple rules, which are easy to program and customize.
- To experience with the development of IDP's as general, reusable IS-rules.

2.1 General Structure and Terms

The WebSIX framework is designed as a client-server system; the server is a Web server and the client is a Web browser. We refer to the server by the more general term Environment\(^1\). It binds the AL and IS layers together. The environment is provided by the framework, whereas the other two parts—the AL and IS codes—are meant to be plugged-in by their respective developers\(^2\).

The environment repeatedly handles users' requests. A single invocation of WebSIX by a user is called an activation, and the piece of AL code that it invokes is called an action. (Simply speaking, an action is like a function and activation is like a function-call).

Requests from the user come in the form of an action name and a set of named parameters (key-value pairs). The environment orders the AL to invoke the required action, resulting in a semantic tree. This is an in-memory XML document, whose nodes represent semantic controls.

Next, the environment passes the semantic tree to the IS code, which applies a series of transformations (or "rules") that gradually change it to become a concrete GUI. (Since the tree always remains an XML document, its final content is in XHTML format). The transformations are implemented in Java, allowing them to have multiple steps and make decisions with Turing-complete computational power.

The resulting GUI is then sent back to the client. The GUI contains the concrete incarnations of the controls that the AL has created during the current activation. Some of these controls are input fields, buttons, and links, that let the user invoke further actions. (The initial action was invoked when the user first browsed to the application server).

---

1 Two other experimental environments were tried - one that generates GUI's in Swing and one command-line utility that reads and writes text files, useful for testing.
2 Note of terminology: It should be clear from the context whether the terms AL and IS refer to a specific application and a specific sequence of transformations, or to the roles of these layers in general.
To summarize, the WebSIX framework employs two modular components: the AL, which is written by application developers, and the IS, which is platform- (and not application) dependent and is written by GUI specialists. Users' requests are handled by the AL, yielding a semantic tree, which is then transformed to a concrete GUI by the IS.

Fig. 2 depicts the general flow. The following sections describe the stages in more detail.

2.2 AL

The Application Logic code (AL) is the code of the computations done in the application. It invokes the core functionality, e.g. data-related actions, and also specifies the interaction with the end user, by using semantic controls (see below).

The AL code is usually organized in classes that represent use-cases; a use-case [3] is a series of user interactions with the application in order to perform a well-defined task. Their code is normal Java code; it can do general purpose tasks, like calculations, validity checks, loops and branches, calls to external components such as databases, and so on. Regarding the user interaction, the code can use input values from previous actions, show output, ask for more input, and let the user choose the next action.

The code invokes these interactions with the user by using methods that are provided by WebSIX, such as Input.string, Output.string, Input.selectOne, and Input.action. These method calls generate instances of semantic controls. That is, each such method call creates an XML element that wraps its details, and appends it to the semantic tree that is being built. There are also methods like Section.start and Section.end that allow for grouping of related semantic controls in the tree.
Listing 2: The WebSIX AL code for the Celsius/Fahrenheit program.

Highlighted – method calls that generate semantic controls.
Notice also the Fetch.* calls, for using input from previous action.

```java
package demo;
...
public class FtoC extends UseCase {

    public void start() {
        Input.number("P_DEGREE", "Degrees");
        Input.selectOneByKey("P_DIRECTION", //parameter name
            "C2F", //default value
            "C2F", "Celsius to Fahrenheit", //options
            "F2C", "Fahrenheit to Celsius");
        Input.action(this.getClass(), "doConvert", "Convert");
    }

    public void doConvert() {
        double d;
        try {
            d = Fetch.number("P_DEGREE");
        }
        catch (Exception ex) {
            start();  //Ask again for input.
            return;
        }

        boolean cToF = Fetch.string("P_DIRECTION").equals("C2F");
        String answer = cToF ?
                    (d + " C = " + (d*1.8 + 32) + " F") :
                    (d + " F = " + ((d-32)/1.8) + " C");

        Output.string(answer);
        Input.action(this.getClass(), "start", "Restart");
    }
}
```

Fig. 3: A possible resulting GUI for the code in Listing 2.
Notice that the AL code calls these methods exactly where it needs them, similarly to printf and scanf in the C language. There is one limitation, however, that is inherent to all interactive GUI systems: The system initiates all the controls, including those for entering input, before the user actually sees the page and enters his input. In other words, the call to Input.number is non-blocking—it doesn't really wait for the user's immediate input like scanf in C does. From this reason, only the code of the next action will be able to use the entered inputs (by calling methods of the Fetch class).

Consider, for example, Listing 2 which repeats the Celsius/Fahrenheit example. The method start() is invoked when the user browses to http://the.server/websix/demo/demo.FtoC.start; it generates the semantic controls that a typical IS may turn to the GUI in Fig. 3 (a). When the user, in turn, shall click the "Convert" link, it will browse to http://the.server/websix/demo/demo.FtoC.doConvert and send the parameters P_DEGREE and P_DIRECTION to the server. This new activation of the server will calculate the result, whose GUI is shown in Fig. 3 (b).

Listing 3: The semantic tree resulting from the code in Listing 2.
<m:root ...>
  <m:params>
    <f:usecase>demo.FtoC.start</f:usecase>
    <f:user>anonymous</f:user>
    <f:title>demo.FtoC.start</f:title>
  </m:params>

  <m:WorkspaceContent>
    <m:InputText>
      <f:paramName>P_DEGREE</f:paramName>
      <f:type>text</f:type>
      <f:label>Degrees</f:label>
    </m:InputText>

    <m:Selection>
      <f:paramName>P_DIRECTION</f:paramName>
      <f:idField>key</f:idField>
      <m:String>Celsius to Fahrenheit <f:key>C2F</f:key></m:String>
      <m:String>Fahrenheit to Celsius <f:key>F2C</f:key></m:String>
    </m:Selection>

    <m:Action>
      <f:action>demo.FtoC.doConvert</f:action>
      <f:label>Convert</f:label>
    </m:Action>
  </m:WorkspaceContent>
</m:root>
2.3 The Semantic Tree and Semels

The semantic tree is composed of semels – Semantic Elements. A semel is an XML element identified by the namespace "http://local-host/websix/semantic", which we denote by 'm:'.

A semel is an equivalent representation of an object (which is, basically, a group of fields with values). The data inside a semel comes in two forms:

- **Fields** – have names and values. Their names are unique within the semel. Clearly, they are equivalent to fields of an object. Technically, they are encoded as elements of the form `<f:name>value</f:name>`.

- **Sons** – values without a name; they can be accessed by a numeric index.

Each value of a field or a son can be text, a plain (non-semel) XML element, or another semel. Therefore a semel can naturally comprise a tree.

Being an XML element, a semel also has a name, which we sometimes call "type", even though in all other regards the semel is weakly-typed – new fields can be added to it on the fly. (This is the reason that we use XML and not Java objects).

The meaning of a semel is one of the following:

- Application-domain objects, e.g. a person, product, etc.
- Basic Semantic Controls (BSC’s), like InputText and Action.
- Composite Semantic controls, like Section and Selection.

As a simple example, this semel represents a semantic control named "Action":

```xml
<m:Action>
  <f:label>Press me</f:label>
  <f:action>MyApp.MainUsecase.showWelcomeMsg</f:action>
</m:Action>
```

Listing 3 shows the entire semantic tree produced by the start() method of the Celsius/Fahrenheit example.

The distinction between fields and sons is not merely syntactic. Fields are used for holding values that have a significant meaning to the object containing them, like the two fields of m:Action shown above. Sons, on the other hand, are often items that "just sit there", e.g. the various controls in the page's header, or sub-menus of a menu.

2.4 The Three Stages of Transformation

The job of the transformations is to turn the semantic tree—a semel with many child semels and fields—into an XML document without semels and fields, but with information about style, layout, and other features needed for the target platform.

The transformations are composed of three major stages. These stages can be thought of by what they do to the semantic tree: Stage 1 is the **metadata** stage – it adds information about application-domain semels. Stage 2 is the **enriching** stage – it modifies the semantic tree in ways that usually add information to it; and Stage 3 is the **trimming** stage (or "picking the fruit") – it removes all the semantic information from the tree and leaves the concrete results.

---

3 The full namespace of the fields is "http://localhost/websix/semantic_fields".
More accurately, the first stage is provided by the application code, whereas the other two comprise the IS.

1. The **application-specific** stage. It adds metadata to application-specific semels. It usually designates the fields that should be visible to the user, either by setting the `display` attribute on them, or by adding a field named `f:display` with a piece of semantic UI to be displayed.

   For example, on a Person semel it may either change existing fields such as: 
   ```xml
   <f:firstName display="true">John</f:firstName>
   ```
   or add a new field:
   ```xml
   <f:display><m:String>Mr. John A. Smith</m:string></f:display>
   ```

   This stage may also add, for `m:Action` controls, a field that says which one is the default action of the page, and whether an action has irreversible effects (typically the IS will show it as a button rather than a link).

   It may even be used to find specific strings that the AL stored in the semantic tree, and replace them with translations in other languages or with shorter strings.

   This stage is expected to be quite simple. We don't consider it as AL code but as the first stage of transformations, from two reasons: (a) The AL writer may prefer to code these details in one place, since they depend on general properties like semel types, and not on the specific action that the AL has just performed. (b) Whether we want to apply them, e.g. to shorten some strings, might depend on the target platform.

2. The **generic** stage. This stage belongs to the IS. It enriches the semantic tree with display hints and new elements. On one hand, these new details start to take on a concrete meaning; the most notable example is controlling the GUI's general structure by adding new groups of controls with the `f:layout` field. On the other hand, these details are still specified in a platform-independent way, e.g. setting `f:layout="row"` to a group of semels does not specify whether to implement the layout by HTML tables or `div`s, or whether the platform is HTML at all.

   The generic stage is expected to become the most developed one in terms of its abilities and expressiveness. It can be composed of many transformations that represent rules that are common to many applications – namely, the IDP's that the IS should apply.

3. The **platform-specific** stage of the IS converts the basic semantic controls (BSC's) into controls of the target platform, and groups of controls into concrete containers with the wanted layout (e.g. `span`, `div`s, and tables in HTML). It also sets the graphical properties of all these elements, based on their original semantic data (name and the hints that were added). In HTML this is done by adding CSS styles.

   This stage should be able to handle all the BSC's that the AL generated, as well as all the hints that the generic stage has added (`f:layout`, other fields, and special-purpose semels). Notice that the set of standard BSC types is pretty much constant (see section 3.2.2). The set of display hints, on the other hand, depends on which IDP's are included in the IS (see chapter 4).
3. Implementation

As said above, WebSIX is composed of a thin Environment layer, which invokes the two main layers, namely AL and IS. The classes that comprise the AL and IS are plugged into the system by its developers/designers/administrator. The framework defines their interfaces and provides a library of classes that facilitate their development.

Section 3.1 describes how everything connects. The following sections describe the classes that serve the AL and IS layers. Programmers of WebSIX-based applications, or IS implementations, may refer to these sections.

3.1 General Flow

3.1.1 Environment Classes

This section describes only the common case, where WebSIX is deployed as a Web server. In this case, the class `websix.environ.ServletHandler` implements the environment. It runs as a Java Servlet [17] and it is able to serve multiple WebSIX applications to multiple users.

When a user browses to an address of the form `http://the.server/websix/application.name/action.name`, the servlet dispatches the request into the required application. For this purpose, the servlet extracts the following details:

- The application's main class (a class that extends `websix.app.Application` – see below). The full name of that class is taken from the file `web.xml` (a standard definition file of Servlets), using the `/application.name/` part of the URL.
- The action name – it is taken directly from the URL. It is the application's responsibility to verify its validity. The action name may be empty, in which case the application should perform its default action (normally, showing the homepage/main menu).
- The parameters of the current action – they are provided by the user in one of the standard ways as defined by the HTTP protocol: GET or POST⁴.

An important service that the environment provides is the `session`. It is a standard mechanism of the Java Servlet system, and a separate session is stored for each user that connects to the server. The session is a key-value collection that allows the AL code to maintain a `state` across multiple actions of the same user. That is, the actions of the AL can store any key-value pair in the session, for later use. Note: issues like session persistence and memory usage are out of the scope of this work.

⁴ Although the GET and POST methods have different semantics (POST is used for actions with side-effects), when the user invokes an action the servlet does not care whether GET or POST is used. In order to use the appropriate method, the IS should have created the preceding page (in which the user clicked on a link/button to invoke the current action) with the appropriate setting for each link/button.
The next steps of the Environment are (see also Fig. 4):
1. Wrap all the above details in an Activation object.
2. Pass it to the application object's run() method. This completes the AL part of the action.
3. Obtain the main transformers for the given application and target platform. The transformer that performs the application-specific stage is obtained from the method Application.getAppTransformer(), and the main transformer of the pure IS layer is Environment.getInteractionStyle().
4. Apply the main transformer on the root of semantic tree and return the resulting concrete XHTML page to the user.

Fig. 4: The data flow during one action.
3.1.2 The Activation Class

The class Activation comprises the context in which a single action runs. On each action, the environment creates an Activation object and passes it through the AL and IS.

The main data members of class Activation are:

- Environment environment;
- Application application;
- String actionName;
- Map<String, String> actionParams;
- Map<String, Object> sessionParams;
- Semel responseRoot;

It should be clear how these data members map to the terms defined in the previous sections. Just to rehearse: 'application' and 'actionName' identify the action to be invoked. 'actionParams' are the parameters given to the current action by the user, and 'sessionParams' contains the values stored by the AL code in previous actions; i.e. it maintains a long-living state. Note that sessionParams is actually maintained by the Environment; the Activation object has a reference to it only for convenience.

'responseRoot' is the root of the semantic tree, which the AL fills with semantic controls and later the IS gradually modifies it into a concrete GUI.

The decision that the entire Activation object would pass through the AL and IS, was made for the sake of simplicity. However, in order to keep the AL and IS independent, the actions in the AL must not depend on the environment type and properties, and the transformers in the IS must not depend on the current application and action.

Another feature worth mentioning, is the static method Activation.getCurrent(), that returns the Activation object that is being used in the current thread. Each action is performed on a single thread, therefore it is safe to call Activation.getCurrent() from within the action and the various utility methods that it calls. Indeed, this behavior seems like programming with global variables, but it spares the need to pass many parameters to frequently used functions (especially those that create semantic controls), which improves the code readability.

3.1.3 Object States and Lifetime

Now that we have seen the main objects, it should be noted that the execution model of WebSIX is very simple: very few objects have a state that can change during the various AL actions and IS transformations. These objects are the current Activation, which contains the input and output (i.e. semantic tree) of a single action, and the sessionParams, which maintains the application's long-term, per-user state.

In contrary, the actions don't even change the objects that contain their own code, namely the instances of the application class and the use-cases. These classes are stateless – have no data at all.

In the IS, similarly, the transformer objects do not change their own state – they are immutable. (Technical note: we say "immutable" and not "stateless" because transformers usually hold some data that describes their exact operation, but this data is only set when the transformer object is being constructed).
Table 1: Object States and Lifetime

<table>
<thead>
<tr>
<th>Type of Object/Entity</th>
<th>Lifetime</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation</td>
<td>Single action</td>
<td>Modifying</td>
</tr>
<tr>
<td>Session</td>
<td>Entire server runtime</td>
<td>Modifying</td>
</tr>
<tr>
<td>Environment</td>
<td>Entire server runtime</td>
<td>Holds the session</td>
</tr>
<tr>
<td>Application/Usecase</td>
<td>Doesn't matter</td>
<td>None</td>
</tr>
<tr>
<td>Transformer Objects</td>
<td>Doesn't matter</td>
<td>Immutable</td>
</tr>
<tr>
<td>Transformer Data</td>
<td>Forever (files)</td>
<td>Immutable</td>
</tr>
</tbody>
</table>

Table 1 summarizes the kind of state and the lifetime of these objects. Technical note: for optimization, the objects whose lifetime doesn't matter are created on the first demand and stay cached.

3.1.4 Notes about Overhead

GUI Creation Time:
There are three factors that might hinder the runtime of each action:
- The AL rebuilds the entire semantic tree on every action.
- Each transformer in the IS usually performs its own traversal on the semantic tree.
- The client's browser has to parse a new HTML page after every action.

These operations, however, have a negligible performance hit, given that, roughly, the number of controls (semantic or concrete) in a typical page is \( n = 10..100 \); an IS is composed of about 10 IDP's; each one of them performs an \( O(n) \)-time XPath query in order to find the relevant nodes, and a much simpler manipulation on each node.

Server Memory Usage:
The way of work with the semantic tree does not pose a memory burden on the server; after the tree that has been built during one operation is sent to the client, the server needs not remember it. The only problem regarding the memory usage is the fact that the AL code can save objects in the session of each user. The session is not garbage-collected (because the objects stored in it are, technically, always accessible), and we make no attempt to move idle user sessions to the hard drive. These issues are simply not relevant to this project's goals; of course, in real-life there must be a solution to the session management, but that is not different from any other Web server.
3.2 AL Classes

3.2.1 Application and UseCaseBasedApplication

Every application should have a main class that derives from websix.app.Application and implements the method run(Activation act). This method performs a single action and fills the field act.responseRoot with the resulting semantic tree.

The application developer is free to derive directly from class Application and implement the run() method however he wants. That is, he is free to decide which values of the parameter actionName are valid and how to perform each action.

However, it is common that each action is implemented by its own method, which resides in a class of related actions – namely, a use-case class – and therefore the action name is the qualified method name ("package.class.method"). The class UseCaseBasedApplication does exactly that – its run() method parses the action name and runs the respective method.

UseCaseBasedApplication.run() also calls two methods before and after running the required action, namely runBeforeAction() and runAfterAction(). The application may override runBeforeAction() to perform general checks (it may return false, to terminate the action), but more importantly, it can override runAfterAction() in order to add parts of the GUI that are common to all the use-cases, e.g. a header and a navigation bar. The reason to add these parts after the action was performed, is that sometimes their content depends on the result of the action (examples, 1. If the page header shows the name of the logged user, and the current action is the log-in action itself; 2. Some actions would like to start a form state [6, p.63], i.e. show a page without the navigation bar, in order to make the user concentrate on the current task). Anyway, adding these global parts to the GUI after performing the action does not affect their order on the screen, since the IS lays out these elements in its own way.

3.2.2 Semantic Control Classes

Following the WebSI approach ([4]; though giving up its unique naming convention), the AL code calls methods provided by the framework, that add semantic controls into the semantic tree that is being generated. For example,

\[
\text{Input.text("P\_NAME", "Your name");}
\]

adds the following semel to the semantic tree:

\[
\begin{align*}
&\text{\textless m:InputText} \\
&\quad \text{\textless f:paramName>P\_NAME</f:paramName>} \\
&\quad \text{\textless f:label>Your name</f:label>} \\
&\text{\textless /m:InputText>}
\end{align*}
\]

There is a global field named Activation.getCurrent().insertionNode, under which the new node is added. The programmer can ignore this fact and thus let all the semantic controls be added one after the other. Conversely, he may change the insertion node in order to group together some GUI parts. For example, if the AL code calls:

1  Section.start("PersonalInfo");
2  Input.text("P\_NAME", "Your name");
3  Input.number("P\_AGE", "Your age");
4  Section.end();
then a semel named m:PersonalInfo will be added at the default position, and then this semel itself will become the insertion node. Lines 2-3 will create semantic controls under that semel. Line 4 restores the insertion node to the value it had before line 1.

Some of the methods that generate semantic controls in WebSIX are:

- **In class Input:**
  - `action(Class<?> inClass, String methodName, String label)`
    - Will turn to a link or button that lets the user invoke an action.
  - `text/password/number/bool/dateTime(String paramName, String prompt, [String defaultValue])`
    - Create an input control, including the prompt (label) that appears on/in front of it. When the user invokes the next action, a parameter with the name given in paramName will be sent to the server.
  - `selectOne(String paramName, String... options), selectOne(String paramName, List<Semel> options, String keyFieldName)`
    - Create a Selection semel with the given list of options content. The options may be more entire semels, as seen in the second version.

- **In class Output:**
  - `string(String value)`
    - shows plain text.
  - `status(String intent, String message), status(String intent, Semel control)`
    - show text, or a whole semel, with intent. An intent is one of a few predefined strings, like "error", "success", "tip". When a control has the f:intent field, the platform-specific stage can show it with appropriate style.

- **Section.start/end** – a composite control.
- **Semantic.addSemel(Semel s)** – a utility method for adding any semel to the semantic tree. The AL can use it to add application-specific semels.

### 3.3 IS Classes – Transformers

A transformer is an object whose type derives from websix.trans.Transformer. It should implements the abstract method

```java
public abstract DidWhat handleElement(Activation act, Element node).
```

The returned value, of type `enum DidWhat { Nothing, Node, Subtree }`, tells whether the transformer has changed the given node, its entire subtree, or nothing.

A transformer can be composed of other transformers in various ways, as described in the following sections. In particular, the entire IS is a composite transformer, such that a single call to its handleElement method on the root of the semantic tree will indirectly convert the entire tree into a concrete GUI.
3.3.1 Loading Transformers

A transformer object usually has some content, as defined in the following sections. One can either construct the object in Java, or encode the transformer's content in an XML file and load it by using `websix.trans.TransformerFactory`.

Listing 4 (a) shows the two methods combined: a `ListDoAll` that some of its inner transformers (and itself) are constructed purely in Java, and some others are loaded by the factory. Listing 4 (b) shows how the same transformer is encoded in XML.

Listing 4: Outline of a typical IS definition
Shows the same IS definition in Java and in an XML file.
The XML definition can refer to Java classes by the `class` attribute, and to other XML-encoded transformers by the `ref` attribute.

(a)

```
Transformer IS = new ListDoAll(
    //Generic rules
    TransformerFactory.load(
        "classpath:///websix/trans/generic/GlobalStructure.xml"),
    new GlobalNavigation(),
    new PreviewPaneCreator(),
    new ErrorMessageNearInputField(),
    ....,
    //Platform
    TransformerFactory.load(
        "classpath:///websix/trans/html/DefaultHtmlMaker.xml"
    );
```

Is equivalent to:

(b)

```
<transformer class='websix.trans.ListDoAll'>
  <!-- Generic rules -->
  <transformer
    ref='classpath:///websix/trans/generic/GlobalStructure.xml'/>
  <transformer class='websix.trans.generic.GlobalNavigation' />
  <transformer class='websix.trans.generic.PreviewPaneCreator' />
  <transformer
    class='websix.trans.generic.ErrorMessageNearInputField' />
  ...
  </transformer>

  <!-- Platform-specific -->
  <transformer
    ref='classpath:///websix/trans/html/DefaultHtmlMaker.xml'/>
</transformer>
```
3.3.2 Composite Transformers

In the following transformers, the `handleElement(act, node)` method calls `handleElement()` of some other transformers on the same node:

- `websix.trans.ListDoAll` is composed of a list of other transformers, and to handle a node it invokes all of them on the given node (it can stop, in case that one of the transformers in the list deletes the node).
- `websix.trans.ListFirstMatch` is similar to `ListDoAll`, but it invokes its components only up to the first one that reports that it has handled the node (i.e. returns a code other than `DidWhat.Nothing`). This is roughly the equivalent of an `if..else` chain.
- `websix.trans.MapByType` contains a mapping between a semel type (e.g. "Action") and the transformer that should handle it. When called on a semel for which it has no mapping, or on an element that is not a semel, it returns `DidWhat.Nothing`.

There must also be transformers whose `handleElement()` method propagates to other nodes; these are:

- `websix.trans.Recursive`, whose `handleElement(act, X)` method traverses the entire subtree of X and applies an inner transformer on (possibly) every node Y in the subtree. Note that if `inner.handleElement(Y)` returns `DidWhat.Subtree`, the recursion will not proceed under Y. In addition, if the handling of Y replaces it with a new node or subtree, it will be handled as well.
- `websix.trans.Selective`, is the base of a powerful family of composite transformers. It contains an inner transformer and an abstract method `getElementsToHandle(act, node)` whose meaning is probably self-evident. Thus, `Selective.handleElement(act, X)` method invokes `getElementsToHandle(act, X)` on the given node X and then applies the inner transformer on each one of the returned nodes (we expect it to be used only to get nodes in the subtree of X, but do not enforce that).
- `websix.trans.XPathSelector`, as the name suggests, uses an XPath query to select the on which elements to apply the inner transformer. It is useful for defining a rule that applies to semels based on their name, value of a certain field, properties of children, etc.

Examples: To create an IS that, when invoked on the root of the semantic tree, goes down to all the leaves (in order to converts the basic semantic controls to concrete ones), there are two ways:

1. Use a `Recursive` that traverses the tree and runs the same operation on each element. This operation is a `MapByType`, that decides which specific transformer to apply on the element in question.
2. Use a `ListDoAll` that invokes a series of other transformers, still on the root of the tree. Each one of them is a `Selective`, that searches the entire tree for semels of a specific type and invokes a specific transformer on them.

Fig. 5 depicts the two ways. Clearly, way 1 is more efficient, as it converts the entire semantic tree in a single pass. On the other hand, way 2 is oriented to the approach of composing an IS as a set of independent rules.
Assume also that we want to convert every m:Action semel that has the flag f:irreversible=true to a concrete button, and all the other m:Actions to links. Then if we use way 1 for the entire IS, we can map the type "Action" to a ListFirstMatch that holds two transformers: the first one checks if the action has f:irreversible=true and handles it if so. The second one handles the remaining cases.

If we use way 2 then, naturally, we will add a rule that looks for all actions with f:irreversible=true and handles them.

### 3.3.3 Basic Transformers

In order to transform BSC's (basic semantic controls, i.e. leaves of the semantic tree) or containers (inner nodes in the tree), the IS designer may write his own classes in Java, deriving from websix.trans.Transformer. But if the transformation is mostly about rearranging XML tags, one can use the declarative approach of XSLT [19].

websix.trans.XsltTransformer contains an XSLT transformation which the handleElement(act, X) method applies on X and then replaces X with the result.

For example, the following XML fragment encodes an XsltTransformer that converts the m:String semel to the HTML element <span> with the string's value inside it.

```xml
<transformer class='websix.trans.XsltTransformer'>
  <xsl:stylesheet version='1.0' ...>
    <xsl:template match='*'>
      <span>
        <xsl:value-of select='./text()'/>
      </span>
    </xsl:template>
  </xsl:stylesheet>
</transformer>
```

The inner XSLT transformation can take parameters; our XsltTransformer passes it the required parameters. Their values are taken from fields with the same name in a specific node in the semantic tree.

Fig. 5: Two ways to implement the main transformer of the IS.
Another basic transformer is `websix.trans.AddUnique`. It contains an XML element, and its operation is to add that element to the root of the semantic tree, but only once. This is useful in case that the conversion of some semantic control requires that a certain CSS or JavaScript content will be added to the HTML. In this case, the handling of that control will include an `AddUnique` that adds that CSS or JavaScript.

### 3.3.4 HtmlMaker

`websix.trans.html.HtmlMaker` is the transformer that encapsulates the entire platform-stage (stage 3) in our implemented IS'es. This composite transformer converts the semantic tree to HTML by these steps:

1. In each semel, turn the fields that should be visible (that is, `f:display` and all the fields that have the attribute `display='true'`) to regular sons.
2. Convert all BSC's to HTML.
3. Convert all remaining semels to HTML containers (span, div, table) with the required layout (according to their `f:layout` field).
4. Copy the values of some fields from each semel to the class attribute of the HTML elements that the semel has turned into.
5. Final tasks: delete all the remaining fields, add HTML headers and CSS styles.

Although these structural changes generally drop the semantic data, some of it does survive in the form of HTML's class attributes, for example, the semantic subtree

```xml
<m:Section>
  <f:title display="true">Help</f:title>
  <f:intent>helpbox</f:intent>
  ...content...
</m:Section>
```

can turn to:

```html
<div class="m_Section  v_intent_helpbox">
  <span class="f_title">Help</span>
  ...content...
</div>
```

The class `m_Section` denotes that the div was an `m:Section`;
The class `v_intent_helpbox` means that the div had the field `f:intent` with the value "helpbox".
The class `f_title` of the inner span means that it was the field `f:title`.

The class attribute is a very convenient "hook" for applying CSS styles [16] to these elements.

Note that the details of steps 2-5, namely how to transform the BSC's, how to transform containers with certain layouts, which field names or values to retain in the HTML class attributes, and the CSS styles to be added, are in fact parameters; different instances of `HtmlMaker` can be created with different configurations of all these details. For example, `DefaultHtmlMaker.xml` that is used in Listing 4 is one such customized instance.
4. Examples of Interaction Design Patterns

This chapter describes a few IDP's and their implementation in WebSIX. The general suggestions ("What" and "Why (for the user)"") can be found in the wide literature about IDP's. In addition, we explain why the usage of each IDP in WebSIX contributes to the IS in its task of converting a concise semantic UI into a GUI with recommended styles and layouts, and even how the IDP's can induce a better planning of the AL code.

4.1 Global Structure

**What**: Krug [6, p.62] lists the common parts that usually appear on web pages, and their layout. For each website, the exact composition of elements that appear in it is *global* within that site; that is, the same elements appear almost on every page of the site. (He also notes on which pages they don't appear and why).

These elements are divided into a header, a global navigation section, and sometimes a footer. The header, of course at the top of the page, contains a logo on the left (in left-to-right languages), which often serves as a link to the home page. Next to it is an indication of the logged-in user, or a suggestion to log in. Following is some sort of "you are here" sign – an indication of the position of the current page within the site structure; A set of **utilities** such as links to help, changing language, and a search box, often aligned to the right end.

The global navigation area is explained in the next section. The footer repeats some of the links and utilities from the header and navigation, and often contains links to legal notes, contact info/feedback, etc.

**Why (for the user)**: The recurrence of the same structure and style in all of the site's pages (Visual Framework, in the words of Tidwell [13]), helps the user notice whether his browsing operations keep him on the same site and in which section he is. The links and utilities are obviously helpful. An alternative to navigating by links is to use the search box (preferred by many "search-oriented users", says Krug).

**Why (for WebSIX)**: Due to the semantic nature of the AL, information on how to lay out the GUI must be added by the IS. The IDP's, which represent expert recommendations, are thus an invaluable source of layout guidelines (what is a row and what is a column, alignment), and possibly also exact information (size in pixels of each global part, graphic properties of the elements inside the various global parts – fonts, colors, logo image, background images).

For the AL, having to add some constant content to the semantic tree, is a very easy task. (The code for this goes in UseCaseBasedApplication.runAfterAction()). The AL should give predefined names to the semels of the header, navigation, footer, and workspace (the central area that remains between all the global areas) in the semantic tree.

---

Krug calls all the global elements together "Global Navigation". We, like Tidwell, use this term only for the area of the links – see next pattern.
Above all stands the question of legitimacy – is this requirement "semantic enough"? In the case of the global structure IDP, the answer is clearly yes: The list of main sections and the list of utilities of the application are semantic features.

**How:** Listing 8 shows an example of the AL code that generates its relevant global elements.

The transformer `websix.trans.generic.GlobalStructure` sets up the header and global navigation areas. It does so by adding to the semantic tree an `<m:UI_root>` element with predefined sections. Listing 5 shows the semantic tree with the semels that the AL is expected to create and the subtree that the transformer adds.

It then looks for sons of the original root with the names `<m:Header>`, `<m:Footer>`, and `<m:WorkspaceContent>`, and moves their content to the new sections respectively. The new sections contain some layout information, but not everything; details about the appearance of the element inside these areas (e.g. telling the search box in the header to stick to the right side) are yet to be specified by other transformers in the IS.

Notice that this was a generic rule (belongs to stage 2 from the three stages of transformation). Later, in the platform-specific stage, the transformer `HtmlMaker` will add a CSS stylesheet to the tree, giving the desired style to the controls inside the various global sections.

Listing 5: The elements added by GlobalStructure

```xml
<m:root>
  <!-- The original content, created by the AL -->
  <m:WorkspaceContent> ... </m:WorkspaceContent>
  <m:Header> ... </m:Header>
  <m:Navigation> ... </m:Navigation>
  <m:Footer> ... </m:Footer>

  <!-- Added by the GlobalStructure transformer -->
  <f:display>
    <m:UI_root>
      <f:layout>column</f:layout>
      <m:Header>
        <f:layout>row</f:layout>
        </m:Header>
      <m:Body>
        <f:layout>row</f:layout>
        <m:Navigation>
          <f:layout>column</f:layout>
          </m:Navigation>
        <m:Workspace>
          <f:layout>column</f:layout>
          </m:Workspace>
        </m:Body>
      <m:Footer>
        <f:layout>row</f:layout>
        </m:Footer>
      </m:UI_root>
    </f:display>
  </m:root>
```
4.2 Global Navigation

**What:** "Using a small section of every page, show a consistent set of links or buttons to key sections of the site or application" ([13, p.66]). The links are often arranged in a two-level hierarchy, sometimes even three. Thus we call them "navigation tree". Many different styles exist for displaying the secondary links in relation to their parents; see examples in Fig. 6 and in [14] (under "Main Navigation").

The *inner* nodes of the navigation tree may or may not be themselves links (For example, if "News" is the parent of "Local" and "International", it's up to the application logic to decide whether "News" itself is a link to a page with mixed news topics, to a fancy introductory page, or perhaps the word "News" is just a passive title in the navigation area).

Often, the link to the current page is highlighted in the navigation area.

Note that there are other means of navigation, that are not global, such as links to the previous, next, parent, and top sections.

**Why (for the user):** The global navigation conveys the high-level structure of the application/site, and helps to quickly find the wanted information/operation. This pattern is very common and the user most probably expects to see it.

**Why (for WebSIX):** As with the global structure, this IDP and the various ways to carry it out, are a rich source of layout and graphical styles that the IS can apply. And on the AL side, expecting the application to specify the list or tree of its sections (at least the main ones) is justified and semantic, and quite easy to implement. It may even help the developers if they have to plan and maintain a navigation tree. The tree can serve IDP's of other (not global) navigation styles as well.

**How:** The AL provides the navigation tree, which is a subtree named `<m:Navigation>` (under the root of the entire semantic tree). Inside it are semels named "NavigationNode"; each node contains the field `f:label` – human-readable text of the node; The field `f:action` – the name of the action that this node would invoke, or omitted if that node represents a passive title; and possibly other NavigationNodes as sons.

One of the navigation nodes should also be marked with `f:isCurrent=true`, both in order to highlight it, and to support some navigation styles, like in Fig. 6 (b), that only show the subtree that contains the current action. But why does the AL have to put this flag? After all, the name of the current action can be found by other means (it is held in the Activation object). However, the *real* current action—the action whose result we are seeing now—may be different from the name of the highlighted navigation node, because if a use-case is a sequence of actions, the navigation node refers to the initial action of the use-case whereas the real current action may be any of them.

So far with AL requirements. As for the IS, different IS'es can contain different transformers to implement this IDP in different styles. For example, the transformer `websix.trans.generic.GlobalNavigation` is used to display the entire navigation tree in one continuous area (which the GlobalStructure had designated for navigation)⁶, like in Fig. 6 (a). It converts the tree to a simple list of `m:Actions` and

---

⁶ In XPath terms: the original navigation tree is at `/m:root/m:Navigation`, and the transformer writes its content, in a different format, into `/m:root/f:display/m:UI_root/m:Navigation`. 

26
m: Strings, with a field f: depth that denotes each node's original depth. This format makes it convenient for the platform-stage to attach different styles to items in the primary and secondary (and other) levels.

The transformer websix.trans.generic.GlobalNavigationSplit, on the other hand, displays the global navigation in the form of Fig. 6 (b). It creates a row under the m: Header semel, to which it copies the main level links. Then it copies just the secondary level menu that contains the current action to the left panel.

4.3 Tree Layout

What: When displaying a set of data items that have hierarchical relations (e.g. files and directories, messages and responses), lay them out as a column (top-down) but also indent each item by a distance that is proportional to its depth in the hierarchy.

Sometimes the tree structure is portrayed by thin lines, and near each item is a small button that lets the user show or hide its children.

Why (for the user): Whether the user expects to see a tree layout, depends on his experience and the kind of items in question. When it is used, this layout is quite compact and immediately conveys the structure of the item set (compare this, for example, to showing only the children of one item at a time).

Why (for WebSIX): This is another case where the IDP adds layout hints of a special kind. The requirements from the AL (see below) are reasonable and semantical.

Fig. 6: Two possible layouts for a two-level navigation tree. (Inspired by [14], implemented in WebSIX)
How: First, the application should designate the items that have hierarchical relationships. Y is treated as a logical son of X if: (a) the semel of Y is a son of the semel of X; (b) X has a field f:logicalSonTypes with a list of the names of semels that are logical sons of X. This information is needed because not every semel that is a son of X represents a logical son; some of them are simply the content of X, e.g. its description text.

It is most suitable for an application-specific (stage 1)-rule to add that field to the appropriate semels. E.g. in a file manager application, for each m:Directory semel add the field f:logicalSonTypes="File Directory" (the names are space-separated, i.e. the logical sons of a directory are both directories and files).

Next, the generic (stage 2) transformer websix.trans.generic.TreeLayoutMarker finds all the semels that have the f:logicalSonTypes field, and sets their own f:layout field to "treeNode".

Finally, the platform (stage 3)-transformer HtmlMaker converts the elements with this layout type to HTML elements like <div class="layout_treeNode">. These div's are nested, preserving the original tree structure. This enables us to show the indentation by a simple CSS trick:

```
.layout_treeNode { padding-left: 30px; }
```

(that is, each node is indented by 30 pixels more than its parent – The padding in HTML is relative to the parent container).

### 4.4 Preview Pane

**What:** When displaying a list of items and each item has a relatively large content, e.g. email messages, use two adjacent panels: the first one (top or left one) shows the list of items in a short format and allows the user to select any item. The other panel shows the full content of the selected item. ("Two-Panel Selector" in [13, p.31])

**Why (for the user):** Allows the user to handle any particular item while keeping track of the large scale – the item's position in the entire list, how many unread messages remain, etc.

**Why (for WebSIX):** For the IS, as said before, an IDP that provides layout hints is always helpful.

The AL also benefits from using this IDP in the way proposed here, instead of implementing it manually. By "manually" we mean that the AL could have created a section with a list of semels with short content, and another section with the current item's whole content. But the better way is to create one list of semels, set the fields f:isCurrent=true and f:preview=(long content) of the selected semel, and let the IS move things around. This way the AL code is simpler than in the manual way, it retains the relation between the item's description and content, and the data can be used by other IDP's too.

**How:** The generic transformer websix.trans.generic.PreviewPaneCreator finds the semels that contain such lists; that is, if a semel X has a son Y that has the f:preview field, then X is the list and Y is the item. The transformer then adds the preview area as a new sibling of the list, as seen in see Listing 6.
Listing 6: Extracting the preview into a preview pane
(some details were omitted).

```xml
<m:MessageList>
  <m:Message>
    <f:title>Meeting</f:title>
    <f:preview>(... long content ...)</f:preview>
    <f:isCurrent>true</f:isCurrent>
  </m:Message>
  ...
</m:MessageList>

----- Is transformed to:

```xml
<m:TwoPanelSelector>
  <f:layout>row</f:layout>
  <m:MessageList>
    <m:Message>
      <f:title>Meeting</f:title>
      <f:isCurrent>true</f:isCurrent>
    </m:Message>
    ...
  </m:MessageList>
  <m:PreviewPane>
    <m:String>(... long content ...)</m:String>
  </m:PreviewPane>
</m:TwoPanelSelector>
```

Notes:

1. Also items that are not marked as f:isCurrent may have an f:preview. The transformer PreviewPaneCreator makes these previews invisible.
2. If we want the preview pane to be created near W which is some ancestor of Y (and not near X which is the direct parent of Y), we do it by setting the field f:previewParent=true on W.
   Possible reasons: (a) The data items compose a tree, so that X is just another item whereas W is real container of that tree.
   (b) for layout purposes; of course, this case is harder to justify semantically.
3. The application should relate to each item an action that selects it, i.e. that calculates its preview and sets its f:isCurrent field. Whether this task should be automated by the IS depends on the circumstances. That is, if the AL stores in the semantic tree the previews of all the items, the IS may create a GUI with them all in invisible state, and use JavaScript to select which one to show. On the other hand, if it is expensive to calculate the preview of each item then it is quite reasonable that the action of selecting an item will be in the AL.
4.5 Row Striping

**What:** "Use two similar shades to alternatively color the background of the table rows" ([13, p.187]). The colors are usually white and bright gray. This pattern is also suitable for lists that are not tables.

**Why (for the user):** The alternating coloring helps the user track the rows of the table, which is important if it is wide. It also clarifies the table's structure when there are alignment problems (caused by too long or too short content of a cell). It is an aesthetic alternative to drawing table borders as lines.

**Why (for WebSIX):** Actually this IDP has little contribution to the way the IS works. It does not add layout hints (although other styling hints, like background colors, are welcome too). On the other hand, it gives more professional looks with almost no effort.

**How:** The generic transformer `websix.trans.generic.StripedRowsMarker` is applied on the semels that contain the wanted rows. Unfortunately, we have no general criteria to determine which sets of items are good candidates for applying this style. Thus, the application (AL or a stage-1 rule) should mark the parents of such row-sets with the field `f:stripedRows=true`.

Once the relevant parent semels are chosen, the rest is very simple: `StripedRowsMarker` sets the `f:visualHint` field of each son to "lighter" or "darker". Later, the platform-specific stage will convert these sons to HTML elements with, e.g., `class="v_visualHint_lighter"`, and add the appropriate CSS styles.

4.6 Form Alignment

**What:** In a page that contains several input controls and each one is preceded by a label, arrange them in a two-column structure. One possibility is that all the labels will be aligned to the right edge of the left column, and all the input controls will be aligned to the left edge of the right column, so that they all meet at an imaginary vertical line across the page. ([13, p.116] suggests this way, thus she calls this IDP "Right/Left Alignment"). Alternatively, the labels can be aligned to the left, resulting in a more traditional form layout, as seen in Fig. 7.

**Why (for the user):** This is quite more professional-looking than the default layout (that puts each input control tightly after its label), and it facilitates fast reading of all the questions (labels) or all the answers, e.g. when the user is re-checking the values that he entered.

**Why (for WebSIX):** As with Row Striping, this IDP improves the aesthetics with no cost at all, in this case (no hints are required from the AL). Notice that it is easy to know which label belongs to which input control, because each method that creates a semantic control puts its label in its `f:label` field (instead of putting a separate string control before the input control), as this is clearly a semantic relation.
**How:** The generic transformer `websix.trans.generic.FormAlignment` finds "forms", which are semels that contain more than one input control. This transformer knows which types of input controls it should handle (see section 5.2). It then extends each input control as shown in Listing 7.

Notice the "correlatedRow" layout of each label-input pair (`FormQuestion`). In the platform stage, when `HtmlMaker` converts the layouts to HTML elements, it will turn the entire form into a table, each semel with `f:layout="correlatedRow"` to a row, and each one of its sons to a cell in the row. Therefore the \(n^{th}\) column of the table will contain the \(n^{th}\) son of every `FormQuestion`. This seems obvious, but remember that if a semel has the standard "row" layout, then its content will be arranged as a tight row, independent of other rows.

Listing 7: The transformation performed by `FormAlignment`.

```xml
<m:MyForm>
  ...
  <m:InputNumber>
    <f:label>Your age</f:label>
    <f:paramName>P_AGE</f:paramName>
  </m:InputNumber>
  ...
</m:MyForm>

---- is transformed to:

```xml
<m:MyForm>
  <f:layout>column</f:layout>
  ...
  <m:FormQuestion>
    <f:layout>correlatedRow</f:layout>
    <m:String>Your age:</m:String>
    <m:InputNumber>
      <f:label display="false">Your age</f:label>
      <f:paramName>P_AGE</f:paramName>
    </m:InputNumber>
  </m:FormQuestion>
  ...
</m:MyForm>
```

Fig. 7: Two styles of Form Alignment: left-left and right-left.
4.7 Titled Sections

**What:** Divide the content of the page to sections with prominent titles. For example, input controls that are related to the same logical purpose can be grouped together in one section. [13, p.107]

**Why (for the user):** It helps the user quickly access the control or the piece of information that he needs – both on the first time that he sees the page (he can skim over the sections' titles) and in the following times (users tend to have spatial memory).

**Why (for WebSIX):** Organizing the semantic content in sections is, of course, much more preferred than using a plain sequence of semantic controls with some `m:Strings` for the titles between them. The AL code is more readable this way, with the calls to `Section.begin()` being easy to track.

For the IS, this IDP provides style and layout hints. The layout information is still incomplete – this IDP does not state whether sibling sections should be put horizontally or vertically – but the fact that some semel is a section does enable the IS to assign useful properties to it, like spacing and border style.

The division to sections can also serve other IDP's, like "Closable Panels" [13, p.111].

**How:** Simply by CSS rules, relying on the way that `HtmlMaker` transforms semels to HTML elements that retain some of the semantic properties. Recall that if the semantic tree contains

```xml
<m:Section>
  <f:title>Personal Info</f:title>
  ...
</m:Section>
```

then `HtmlMaker` will convert it to something like:

```html
<div class="m_Section">
  <span class="f_title">Personal Info</span>
  ...
</div>
```

And CSS rules can be easily related to these classes.

4.8 Error Message Near Input Field

**What:** When the user submits a form with errors, show again the filled form and show the error messages near the erroneous fields. (similar to "Input Feedback" in [18]).

**Why (for the user):** Quickly shows the place of the error, thus spares some frustration during the relatively stressing step of form submission (compare this to showing an error message at the top or bottom of the entire form). It also gains the user's trust by the fact that his other entered values are kept intact along the process.

**Why (for WebSIX):** Allows the AL code to have separate methods – one that creates the form and another one that prints the error messages (i.e. the AL is written as if it were going to show the error messages at the end of the form), and then the IS will
move them to the right place. The AL is required to relate the error message to the name of the erroneous parameter (see below); this is easy to do and meaningful.

**How:** The AL should implement the logic that recreates the form with all the previously entered values. (This task is probably suitable to be automated in the future).

The AL should also set the field `f:errorFor="P1"` on the `semel` that contains the error message, where P1 is the erroneous parameter (remember that each input control has a corresponding parameter name). It is also recommended that AL sets the flag `f:intent="error"`; a CSS rule will then give a style to the error message.

The rest is simple: the IS sees the `f:errorFor` field and moves the `semel` that has it (i.e. the error message) to be after the input control.

---

**Fig. 8:** The Chess application, saving a riddle.
4.9 Example Applications

The following applications demonstrate some of the IDP's that were discussed.

Riders

Riders is a simple application for planning carpools. It was written mainly in order to demonstrate a proper structure of the AL code for information systems. But for our discussion about IDP's, the interesting part is how the AL code creates the semantic controls that are used by the Global Structure and Global Navigation IDP's. Listing 8 shows that code, and its result is in Fig. 9. As can be seen, it is perfectly normal to call the methods that create semantic controls from within code with conditions.

Listing 8: The AL code of Riders's global structure.

```java
protected void runAfterAction() {
    //Show Header
    Section.start("Header");
    Output.string("Welcome to Riders!");
    if (User.getCurrent()!=null) {
        Input.searchBox("P_SEARCH_TEXT","Search a ride for tomorrow", "riders.SearchRide.start");
    }
    Section.end();

    //Show Navigation
    if (User.getCurrent()!=null) {
        Semel navroot = Navigation.createRoot();
        Navigation.createNode(navroot, "riders.SearchRide.start", "Search a ride");
        if (User.getCurrent().hasACar()) {
            Navigation.createNode(navroot, "riders.OfferRide.start", "Offer a ride");
        }
        Navigation.createNode(navroot, "riders.SeeMyPlans.start", "See my schedule");
        Navigation.createNode(navroot, "riders.LogOut.start", "Log out");
    }
}
```
**Chess**

The Chess application is used as a game for two players, or for a player interested in composing and solving Chess riddles (e.g. How the white wins from a given board state). The page that is shown when playing or solving a riddle, contains three parts: a header, the game board, and the history of moves that were made so far. The history is, in fact, a tree of game states, because the player would like to try several ways to proceed from each state until he solves the riddle.

**Application logic:** The entire state of the application (current board, whose turn it is, etc.) is kept in Java objects that are held in the *session* in the server's memory. Each action of the user calls a function of the AL code, which updates the game's state and recreates the semantic GUI that conveys the new state to the user.

**Semantic controls:** The game board is a semel, which is in fact a basic semantic control. Indeed, it is not very common that an application defines basic semantic controls of its own, but it is expected to happen for other complex data displays as well. Clearly, a game board is entitled to be a basic semantic control, because its layout and its state as a whole, are meaningful; the board is not just a set of 64 cells that happen to be displayed in an 8*8 layout. We therefore wrote a platform-specific transformer that converts the board to HTML.

**Employed Patterns:** In Fig. 10, obviously, the tree of game states employs the Tree Layout IDP. An application-specific rule finds all the *m:GameState* semels and adds to them the field *f:logicalSonTypes="GameState"*.

The title "Game History" is prominent because it, together with the tree, belong to a Titled Section.

Fig. 11 shows the Striped Rows pattern on the list of riddles, and the Preview Pane in which the board of the selected riddle is presented. Notice that there is a conflict between two CSS rules here: the row of the selected riddle might get its background color either from the CSS rule that paints the gray & white striped rows, or from the rule that highlights the selected item (part of the Preview Pane IDP). Therefore even the CSS rules should be either added in the correct order, or use some other way to resolve their precedence ([16]; we used the '!important' directive in this case).

Fig. 8 shows the IDP of Error Message Near Input Field.

---

**Fig. 9:** The Riders application
Fig. 10: The Chess application, playing page.

Fig. 11: The Chess application, riddle loading page.
5. Evaluation

5.1 IDP's and Their Requirements from the AL

Table 2 summarizes the pieces of semantic data that our experimental IDP's require the AL to provide. Note that some IDP's can be implemented by several transformers that provide different styles. Yet, usually the requirements of each IDP from the AL depend only on the nature of that IDP and not on its particular implementation; this is because the information in the AL describe the meaning of the semantic controls (or application-domain objects) and the relations between them.

Table 2: IDP's and their requirements from the AL

<table>
<thead>
<tr>
<th>IDP</th>
<th>Requirements</th>
<th>Semantic?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Structure</td>
<td>&lt;m:Header&gt;, &lt;m:Footer&gt;, and &lt;m:WorkspaceContent&gt; (all optional)</td>
<td>yes</td>
</tr>
<tr>
<td>Global Navigation</td>
<td>&lt;m:Navigation&gt;, f:isCurrent</td>
<td>yes</td>
</tr>
<tr>
<td>Tree Layout</td>
<td>f:logicalSonTypes, arranging the semels as a tree</td>
<td>yes</td>
</tr>
<tr>
<td>Preview Pane</td>
<td>f:preview, f:isCurrent, selection logic, f:previewParent (optional)</td>
<td>depends, no</td>
</tr>
<tr>
<td>Row Striping</td>
<td>f:stripedRows</td>
<td>no</td>
</tr>
<tr>
<td>Form Alignment</td>
<td>none</td>
<td>no</td>
</tr>
<tr>
<td>Titled Sections</td>
<td>&lt;m:Section&gt;, &lt;m:Subsection&gt;, f:title</td>
<td>yes</td>
</tr>
<tr>
<td>Error Message Near Input Field</td>
<td>f:errorFor</td>
<td>yes</td>
</tr>
</tbody>
</table>

As said before, the field f:stripedRows is not semantic; it only means "We want to apply this IDP here", and the Striped Rows IDP is merely about style. One may also doubt whether the semels &lt;m:Header&gt; and &lt;m:Footer&gt; are semantic, because their names imply their location and layout. We believe that it is meaningful to define them in the AL, since they represent special groups of controls, and the Header can be used by other IDP's too.
5.2 Transformer dependencies

Order

Some IDP's have to be applied in a certain order. For example, Global Navigation uses the areas that Global Structure created before it. Also rules that apply to the same type of basic controls, e.g. converting some m:Actions to buttons and all the rest to links, have to be applied in the correct order. Even in the concrete level there might be conflicts, such as the one with CSS rules (see the Chess example in Fig. 11).

The order in which HtmlMaker works – handling all the basic controls before all the layouts – is currently not very significant, because the IDP's that we used to affect the global layout are not related to those that choose the concrete basic controls. But this might change as well, for example if concrete controls should be chosen according to the available space, which depends on the layout.

Two Stages for each IDP

The most common kind of dependency, as can be seen in most of our IDP's, is that HtmlMaker should handle the elements that the generic transformers have created; that is, provide CSS styles to them and know how to handle the special layouts assign to some of them (e.g. f:layout="correlatedRow").

A typical IS should therefore look like the one in Listing 4, having a series of generic transformers followed by one instance of HtmlMaker; the content of the HtmlMaker (not shown in Listing 4) would be properly customized to the IDP's that are included in the IS – it should contains all the required CSS styles, and a set of transformers for handling all the required layout types.

Notice that we do not see any problem with this kind of dependency; We expect that if a library of IDP implementations is to be created, each IDP will contain one generic transformer (perhaps two), followed by several examples of concrete styles from which the IS designer can select one.

Cross-Transformer Knowledge

Another interesting kind of dependency is found in the Form Alignment IDP – it should know which semels represent input controls whose label is drawn on their left side. This might depend on the choice of concrete controls (e.g. a radio-button shows the label on its right side). In other words, the generic (stage 2) transformer FormAlignment should know what the platform-specific (stage 3) transformers that create the concrete controls will do. Our solution is that the instance of FormAlignment which belongs to a specific IS will be provided with the names of the semantic input controls that it should treat. We prefer this solution, which only deals with adding knowledge to some transformes (similarly to the previous section), over some other solution that we considered, which involved actual manipulations on the semantic tree.

Conclusion

We are separating the "logic" of the IDP's from their style, in a way somewhat similar to the separation between AL and IS. This is an advantage, as it allows the IS designer to see and edit all the visual styles in one part of the IS code.
In order to create a whole IS from the set of wanted IDP's, for each IDP the designer has to add the appropriate transformer into the series of generic transformers in the IS, and add the concrete style information into the target-specific part. In a few cases there are constraints on the order, or other required pieces of information. If they are to be documented or formalized in some way, it should be relatively easy to construct a correct IS.

5.3 Difficulties in Using WebSIX

**Fine-Tuning**

What we expect to be the major shortcoming of WebSIX in regard to developing large applications, is the fine-tuning problem. The very nature of WebSIX is to provide a global, uniform solution to all the controls of the same type. One can easily determine whether all selection controls will be incarnated as combo-boxes or radio buttons. In order to affect only a subset of the controls of some type, there are three ways that are "semantic enough":

- Using a semantically-meaningful field (and expecting the AL to provide its value), e.g. `f:intent='error'` to paint error messages in red.
- Testing some other property that is general enough to be considered an IDP. e.g. set the alignment of adjacent controls of the same type.
- Treating differently controls that are under a specific, meaningful container, e.g. the links that are under `m:Navigation`.

In some cases, however, the designer would like to give a different style to a specific control (instance), either in order to convey some meaning or just to fix layout problems. For example, in Fig. 8 it is needed to increase the spacing between the color selection and the actions below it. In Fig. 11 we would like to set a minimum size to the riddle list so that even when no preview is displayed (i.e. no riddle is selected), the list will be at least as high as it is when a board is displayed.

The lack of fine-tuning will probably deter some developers from using WebSIX. Notice that, technically, fine-tuning is not impossible; as a last resort, one can write an IS-rule that handles a specific control in a specific application. Or even write a rule like "if currentUsecase==xyz then emit a hardwired HTML page and stop". But this is obviously not the intended way of work.

**End-To-End Argument**

The End-To-End Argument [11] implies that no matter what services are provided by the intermediate layers of a system, there will remain some issues that require coordination of the two ends of the system. Its manifestation in WebSIX applications is in cases that the AL writer has to do things that depend on the target platform and employed IDP's. For example, even simple choices such as phrasing of text labels may vary according to several factors: The AL may want to show short or long text depending on the platform target and layout; Labels such as "see below" also depend on the layout; Error messages about user's input may or may not specify which field they relate to, based on whether they appear near that field.
While the argument implies that WebSIX cannot eliminate all of these dependencies, we would like to facilitate their handling. This is the reason for the application-specific (stage-1) transformations. What this stage does, depends both on the application and, possibly, on the target platform and the IS. It can, for example, look for certain \texttt{m:String} controls that the AL has created, and replace them with a shorter version. Thus, the code of the application-specific transformer helps the application writer concentrate all these issues in one manageable place.

More complex end-to-end cases are those that affect the design of whole actions in the AL. For example, in the Chess game, if the platform supports a dragging gesture then the AL should have an action that moves a piece from square A to B. But if the platform only supports single clicks, then the AL should implement two actions – one for the first click and one for the second. As for these cases that require new AL action, we believe that they will be quite rare, and that, anyway, the new AL actions will be meaningful and not look like an ad-hoc solution a certain platform.

### 5.4 Considerations in Designing WebSIX

**Where to Apply IDP’s**

Each IDP in the literature contains an advise about "when to use"; the equivalent in WebSIX is the need of the IDP implementation to find which nodes it has to work on in the semantic tree. In the beginning of this work we anticipated that many transformers would be able to detect these nodes by testing the structure of the semantic tree. For example, when converting the \texttt{m:Selection} semantic control to concrete controls, one can decide that if the selection has at most three options it will turn to a set of radio buttons, and otherwise to a dropdown menu.

Eventually, that was not the case. We observed that the conventions about which concrete controls to use (and which IDP’s to apply), are based on something in the meaning of the controls. For example, a hyperlink for a viewing-only action and a button for an action with side effects. Even the concrete control for selection are sometimes chosen by the meaning: radio-buttons for a choice between essentially different options (like yes/no) and a dropdown menu for a choice between similar items (like cities).

Therefore we chose the approach that the AL would add semantic "hints" to the relevant semantic controls. Although this might slightly hinder the AL code, it yields a more satisfactory concrete GUI.

**Limited Control over the Layout**

The two building blocks that we provide for layout are "row" and "column", namely, to divide a container horizontally or vertically. We chose this scheme because (a) it is simple to implement; (b) it is a very common design, where the entire page is divided in one dimensions, then each part is divided independently, and so on.\(^7\)

\(^7\) This structure is called "kd-tree". See for example the Eclipse IDE.
We considered writing an algorithm that would recursively assign "row" and "column" layouts to various parts of the UI. Eventually we gave up from the same reason of the previous section: it turns out that the layout is usually based on conventions.

Clearly, adding the f:layout hint is appropriate in the generic stage of the transformation: On one hand it is not a semantic hint, so it must not be provided by the AL, and on the other hand, it is independent of any concrete GUI library.

Yet, the effect of the f:layout hint on the final layout is very much limited. f:layout="column", for example, does not specify whether the column should have exactly the width of its content, or try to grow further. There is also no specification for the alignment of items inside the containers (e.g. the search box which sticks to the right). One option was to invent a "mini-language" of secondary layout hints, e.g. f:layout="column;50%". But this would require us to implement quite elaborated translations of these hints to concrete HTML. So instead we chose to leverage the power of CSS: Since the elements that get a special style are usually those that have a special role, like header, navigation, search box, etc., it is legitimate that the IS would contain special CSS rules for these elements. These rules are application-independent. Of course, with CSS we can set many more style properties than just the layout.

5.5 Suggested Future Work

Enhanced traceability: The fact that we create a new GUI on every action is a drawback of WebSIX. In contrast, with concrete GUI libraries like Swing, their very nature is that each action manipulates only the relevant controls. This trend also became prevalent in Web applications thanks to AJAX technology. This is known to improve the user experience. Our work does not address this ability, since it focuses on methods to create a single GUI. Maintaining relations between the semantic tree nodes and the concrete GUI elements in order for actions to change only the relevant GUI parts, would be desired.

Fine-tuning: Due to the problem of fine-tuning in WebSIX, a semi-automatic approach for GUI design should be considered. For example, by using WebSIX as a plug-in inside a development environment – to create an initial concrete GUI that can then be edited.

IS Builder: The conclusions mentioned above about the construction of an IS from a set of required IDP's, imply that this process may be done automatically. We would like to create a tool that takes as input a set of IDP's, and generates the IS that applies them all. For this purpose it is required to create a collection of transformers that implement various IDP's and to encode their dependencies in a formal way (similar to Pattern Languages – see below).

That IS builder may also serve as an optimizer; as we said about composite transformers (section 3.3.2), a list of rules that are applied separately, may be equivalent to a Recursive transformer that handles the semantic tree in a single traversal. The latter is clearly more efficient. Methods for testing a whole set of rules at once should be investigated.
6. Related Works

Interaction design recommendations, which are based on behavioral experiments and industrial experience, are expressed by GUI design rules, Interaction Design Patterns (IDP's), and pattern languages (see below).

In the state of the art, these recommendations are employed by a human designer for designing of GUI's that are expected to have satisfactory UX properties. In a later, second phase, the designed GUI is implemented by a human developer in some technology, for example by employing a GUI component library. This contrasts with WebSIX, whose software constructs GUI's that readily adhere to specified IDP's. We did not find earlier works for GUI construction, whose software exploits design rules, IDP's, or pattern languages for the same purpose.

There are works that employ software engineering design patterns for automatic software generation. These works do, however, not address the problems of GUI design patterns. A pioneering work in this area is the method of [2], which employs a number of transformations on a given object code, resulting in an abstract representation of this code. To this abstract representation, the elements needed by the employed software design patterns are added. They assume that the employed design pattern is applicable to the processed code. The case of IDP's is more difficult than that of software design patterns, because they address both code and a two dimensional graphical layout of GUI controls. Moreover, when using a "write once run everywhere" language, such as Java, software design pattern are platform independent. This contrasts with IDP's where specific platform capabilities matter. The method of this work is similar to [2] in employing transformations for producing an abstract representation. However in WebSIX, it is a representation of a semantic GUI, not of code. The following step of adding elements needed for implementing the IDP, is also analogous to [2]. Similarly, the conclusion is that we can define IDP's by showing the transformers that apply them on a semantic UI.

The next sections review some works about the specification of IDP's (but without automatically applying them), and works about automatic generation of concrete GUI's from their abstract specification (but without explicitly choosing IDP's. It can be said that those methods use IDP's implicitly, by definition, in that they provide common solutions to common problems).

Last, we show some concepts in state-of-the-art GUI development techniques which facilitate the logic/GUI separation.

6.1 Pattern Collections and Languages

As Interaction Design Patterns have become a prominent subject in GUI design, many works in the recent years strive to formalize Pattern Languages, as opposed to ordinary IDP collections. The problem is that in IDP collections, like those of Tidwell [13] and van-Welie [14] (which are highly regarded and cited), each design pattern is defined by a small number of sections, with natural language "prose" in them. For example, each one of Tidwell's patterns is simply described by five sections: What (a brief description), Use When (the context of the application in which the design problem appears), Why (the user's expectation and needs), How, and Example. Related
patterns are mentioned fluently in the text and not in a field of their own.

Thus each pattern is easy to read, but choosing which patterns to implement is a matter of personal taste and knowledge. It is hard to compare and search patterns by precise details of their context, because the When and Why sections are quite general. [10]

Therefore the Pattern Languages enhance pattern information in several ways:

- More fields and more structured fields: As there are more fields, the content of each one is shorter, and thus, in some of the fields, it can be taken from a predefined set of values. The structure and some of the values can be formalized by XML DTD [19] or ontologies [8]. See also the program in [19], which allows to browse and search for patterns, and displays them in a highly structured way.
- Formal links between patterns: [7] uses UML notation to denote related, contained, and mutually-exclusive patterns. [10] uses links between patterns, which the reader can follow and decide between (like walking through a flowchart) in order to get groups of patterns that are used together, and also see community-given "rating" of each pattern and warnings against anti-patterns.
- Formal content of patterns: [7] defines the set of controls and properties of each pattern, but in a very general level (e.g. a Search Box contains a label, input field, and a button). The ontologies of [8] contain entities for some UI elements, also in a very abstract level (e.g. links, entry point).

Despite of the structure and formalization, the patterns' content is still meant for humans to learn and manually implement.

Since WebSIX does not specify how a human designer should choose which IDP's to employ, pattern collections and tools for browsing them may serve as a complementary aid for IS designers. Moreover, it is desired to add to IDP languages some information about which IDP is available in WebSIX, which transformers implement it, and their requirements from the AL.

6.2 GUI Generation and the Multiplatform Problem

The effort to adjust data from a single source (application or Web page) onto multiple different device types, is the essence of Content Adaptation, and is highly motivated by the field of Ubiquitous Computing.

Content Adaptation [15] is the generation of data or modification of existing Web pages, to match the user's needs. Some of the tasks in this field are purely logical, e.g. creating a short news feed by filtering items from news sites. But our interest is on the physical level, namely Adaptive Content Presentation. It tries to fit Web pages that are usually designed for the PC's large screen (and bandwidth) into the limitations of handheld devices. It uses technical means such as decreasing the font size and graphics, and collapsing parts of the page (hiding them and leaving just a title that the user can click; facilitated by the hierarchical structure of HTML files). In WebSIX content adaptation is obtained, for example by employing controls that fit the platform and application. In addition, the IS ability to perform calculations allows it to deliver text and images in different sizes according to the target platform.

43
**Ubiquitous Computing** (Ubicomp; or Pervasive Computing) deals with the design of a remote control device implemented on a smartphone. This smartphone has software for controlling nearby appliances, e.g. air conditioning controllers, video players, and printers. These appliances send on a wireless network an XML file that specifies how to control them. This specification is written in a User Interface Description Language (UIDL), which the remote control software is able to parse and adapt its GUI accordingly. In addition, the end user may customize the GUI. (In WebSIX it is the job of the IS designer to customize the set of IDP's that are most suitable for displaying his application).

An interesting work about UIDL specifications for home appliances is PUC – Personal Universal Controller [9]. Their UIDL is a tree whose leaves represent the commands, parameters, and current state of the appliance, and the inner nodes are used for grouping together leaves and other groups, so that related controls will appear adjacently.

The UIDL has the following features:

- Label dictionaries: Each state and command can have several labels, from which the concrete GUI generator uses the one with the most suitable length for the client size. E.g. "Vol." and "Volume", for a stereo player.

- Yes for grouping, No for layout hints: The UIDL authors are encouraged to divide the UIDL tree to deeply nested groups. This way, a user device with a small screen will be able to show one small group of controls at a time, while a device with a larger screen will show a subtree of several groups at once. However, no layout hints are allowed, and of course, no other concrete platform details.

- Smart Templates: this is a powerful feature for assigning meaning to UIDL elements and generating customized pieces of GUI. Each element can be marked with the attribute `is-a="templateName"`. For example, if the group containing the commands "Play", "Pause", and "Stop" is marked as `is-a="media-controls"`, then instead of just showing buttons with these words, the GUI generator may render this group in the traditional way (buttons with the icons ▶ ▼ ■ and the Play button larger than the others).

These specialized displaying rules are hand-coded pieces of GUI in the concrete GUI generator (the user's hand-held device).

In analogy to WebSIX, the UIDL is the semantic tree. It includes grouping (which is semantic) and no concrete hints, and it marks some elements so that the "IS", namely the GUI generator, will convert them in a special way. These conversions are like IDP's to some limited extent, as they only regard the choice of concrete controls, and are triggered by simple flags in the UIDL.
6.3 State-of-the-art GUI Development Techniques

As explained in the introduction, reducing the dependencies between different layers of an application is a key to a better development process. It allows for each part to be developed separately (possibly by different programmers/designers), and to be modified more easily. Listed below are some common GUI technologies that separate the logic and display layers in various ways.

**Separating the Client and Server**

This separation means to actually run the two parts of the system on two different computers. The main goal is, of course, to make the software accessible to many end users. There are two modes of the client-server design:

- **Thin Client**, e.g. a Web browser. This mode does not separate the GUI generation code from the server; the GUI is created on the server and only displayed on the client. Therefore it does not guarantee that the GUI and the logic be independent.

- **Rich Client**, that is, a program that runs the specific application's GUI on the end user's computer. This demonstrates the difference between AL and BL: the client program is the AL, since it decides how to show everything. It invokes the BL by communicating with the server. Therefore, as said in the introduction, the client code depends both on the server that it interacts with and on the concrete GUI that it generates.

WebSIX makes use of a thin client. This mode was chosen merely for its simplicity (and it is the natural choice, as we use HTML). But, as shown herein, the way of client-server separation is immaterial to the subject of AL-GUI dependencies.

**Separating GUI Creation and AL Code: MVC, MVVM**

With the Model-View-Controller design pattern (MVC; [5]), the business logic interacts with a data structure called a Model. The GUI (View) is connected to the Model in two ways: in order to display data the view reads it directly from the model; and in order to invoke command, the view activates the Controller, which in turn manipulates the model. In our terms, these two kinds of relations between the view and the model are equivalent to the AL layer. This code has to be programmed manually, and of course, it depends on the controls employed by the GUI.

The Model-View-ViewModel design pattern (MVVM [12], introduced by Microsoft) automates the access to the GUI controls: instead of writing a controller that interacts with the GUI, the application developer creates a class called the ViewModel, with simple data members (such as strings), and an automatic mechanism called Binding keeps the values of these members in sync with the GUI. The GUI's developer is responsible for declaring which GUI control is bound to which member. Therefore the GUI can undergo significant changes without affecting the AL (which is the ViewModel).

We can conclude that MVC dictates how to implement the AL in relation to the GUI, whereas MVVM makes the AL code independent of the GUI. So clearly, MVVM is an advancement toward the semantic approach. However, while it simplifies the AL, designing the GUI requires the same amount of work. For each combination of a page
in the application and a target platform, the appropriate GUI should be constructed manually.

**Separating GUI Creation and GUI Style: CSS**

The CSS technology (Cascading Style Sheets, [16]) deals with the presentation of HTML pages and XML data. From the CSS point of view, the logic is the content of the HTML or XML file, and CSS applies visual style to the elements of this content. A CSS stylesheet is composed of *rules*, which are able to "selects" elements according to their names, attribute values and/or hierarchy within other elements.

This approach is indeed similar to the IS in WebSIX, but to a more limited extent. CSS cannot replace the types of controls (e.g. from radio buttons to a dropdown menu) and cannot manipulate the structure of the content in the ways that we use in the IDP's.
7. Conclusions

The experiments with the developed WebSIX framework have produced the different GUIs depicted in Figs 8–11. They stem from both a small information system (car pool application) and a chess game application. These results suggest the ability of the developed method in producing GUI's that adheres to the employed IDP's within different application domains. The user experience (UX) quality of the produced GUI's was not evaluated by behavioral experiments. Such experiments would involve a considerable effort beyond the effort invested in developing the WebSIX framework. It is, however, noted that it is reasonably easy to modify global properties of the GUI produced by WebSIX, e.g. spacing and color of all controls of a certain type. It is therefore reasonably easy to make changes motivated by findings of behavioral UX evaluations. It is known in the art that it is difficult to design a GUI with satisfactory UX. There is therefore a general recommendation that GUI's should in any case be extensively tested behaviorally and modified accordingly [6]. The method employed in WebSIX is thus comparable to other known GUI design methods with respect to the need for polishing through behavioral testing.

The XML tree that specifies the semantic UI is produced automatically by the AL code. Using semantic controls within the AL code is simple and straightforward. Adding, in the AL code, the pieces of information that are needed for applying some of the IDP's, is somewhat intricate, because the required information depends on the IDP's in question. On the other hand, the majority of that information is semantically meaningful and is thus a legitimate part of the AL layer.

Constructing an IS as a set of given transformations is relatively easy, but requires attention due to dependencies between them. Improved techniques for specifying the requirements of the different transformations and IDP's may also be the subject of future research.

The XML manipulation that each transformer does can usually be written concisely. Therefore most transformers are quite easy to understand, once they are developed correctly. Yet, they must be the result of careful planning.

The transformation of the semantic UI to the real-life concrete GUI is done at runtime at negligible computing costs.

The major shortcoming of WebSIX is lack of ability to fine-tune specific elements in the resulting GUI. The WebSIX way of indirectly affecting the GUI might make it harder to use than state of the art GUI libraries in some cases. The use of WebSIX may pay off, however, as the number of applications (or pages in one application) that require a uniform look and behavior increases.

In conclusion, this work presents a novel approach for producing GUI's that adhere to specified IDP's, from a platform independent semantic specification. The experimental work indicates the feasibility of the approach. Further research is needed to understand its industrial applicability.
Bibliography


    http://www.welie.com/patterns


[16] "Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification",
    http://www.w3.org/TR/CSS2/

[17] Java Servlet Technology Overview,

[18] The "UI-Patterns" webpage,
    http://ui-patterns.com/

[19] "XSL Transformations (XSLT)", Version 1.0, specification,
    http://www.w3.org/TR/xslt
We choose to represent the interaction style as a menu (controls) using the WebSI framework. This allows us to embed various types of interaction, from simple HTML elements to more complex interactions, such as dropdown menus or list boxes.

In WebSI, the interaction style is defined in a separate file, usually named `interaction.js`. This file contains the logic for handling user interactions, such as clicks or selections.

The WebSI framework provides a set of predefined interaction elements, which can be used directly in the HTML file. These elements are fully customizable, allowing for a high level of flexibility in designing the user interface.

For example, a simple dropdown menu might be defined as follows in the `interaction.js` file:

```javascript
function handleDropdownClick(event) {
  // Handle the click event
}
```

This function can then be called from the HTML file, where the dropdown menu is defined:

```html
<select id="dropdown">
  <option value="option1">Option 1</option>
  <option value="option2">Option 2</option>
</select>
```

By defining the interaction style in a separate file, we can easily change the behavior of the interface without modifying the HTML code. This not only improves the maintainability of the code but also enhances the modularity of the application.

In conclusion, using WebSI for interaction design allows for a more flexible and maintainable approach to user interface development. It provides a powerful tool for creating dynamic and responsive interfaces that can adapt to different user needs and preferences.
Interaction Design Patterns

(Nondle Bohr Master's Thesis, Computer Science Department, Technion)

Using familiar interfaces in GUI (Graphical User Interface) applications enables us to create familiar and intuitive interfaces. This thesis explores the use of familiar patterns in GUI applications.

The thesis is divided into several sections:

1. Introduction
2. Familiarity
3. Interaction Design Patterns
4. Conclusion

The thesis concludes with a list of references and an appendix containing additional materials and examples.
הפרויקט נועד בהנהלת פרופ’ אליעזר קונרוביצ’ בפקולטה למדעי המחשב.

המרתקות השיחות وكل החום האינסופי, ה으면ורINESS המופתונות, והسوف היחס
בהערכה המקודש לאלייעזר רבה

מקודש ברוחכה רבה לאלייעזר על העידוד וה الغرف, היחס וכסף והישרונות המרותכות.

אני מודה לৎנינו על התמיכה והכפים וה다가 והבישולים.
研究成果

בתוכי

לשם מחקר חלקי של הדירישות לكيفת הרהרה
ממסטר למדעי במדעי המחשב

פרדכי פרת

הון לסנט הטכני - מון טכני לישאר

כסף תשיא"ג

דצמבר 2012
גיישה סמנטית לעיצוב מעויקים משותפים

מרדכי פורת