

Instant Multi-Tier Web Applications without Tears

Gautam Shroff

Tata Consultancy Services R&D
249 D&E Udyog Vihar, Gurgaon,
Haryana 122016 India
+91 124 4165188
gautam.shroff@tcs.com

Puneet Agarwal

Tata Consultancy Services R&D
154B, Block A, Sector 63,
Noida, Uttar Pradesh 201301 India
+91 120 6657047
puneet.a@tcs.com

Premkumar Devanbu

Dept. of Computer Science
UC Davis,
Davis, CA 95616, USA
+1 530 752 7004
devanbu@cs.ucdavis.edu

ABSTRACT

We describe how development productivity for multi-tier web-based database ‘forms’ oriented applications can be significantly improved using ‘InstantApps’, an interpretive framework that uses efficient runtime model interpretation and features an integrated ‘wysiwig’ ‘point-and click’ design editor for developing forms, database schema, control flow, and functional logic. As compared to related academic as well as commercial work, our approach has the distinct advantage of retaining an industry standard architecture that yields high performance and enables model driven functionality to be augmented with hand-written extensions using a well known architectural style and leveraging standard skill sets. In particular, the interface’s ‘look and feel’ can be completely custom built even as the application functionality is developed using the instant ‘WYSIWYG’ editor. Efficient implementation of interpretation and reflection ensures that performance does not suffer, and performance benchmarks support this. Significant productivity benefits are demonstrated with case-studies of real-life applications developed and deployed on this platform, including actual costs vs. estimates using industry-standard function-point based measures. Finally, we describe some interesting features of the platform, including multi-tenancy and weak meta-circularity, and how these are being exploited successfully in practice.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques]: Computer-aided software engineering

General Terms

Design, Languages

Keywords

Model Driven Architecture, Multi-Tier Development, Interpreters

1. INTRODUCTION

Popular, web server platforms such as Struts, J2EE and .NET are

* Prem Devanbu acknowledges support from the National Science Foundation. We would also like to acknowledge Philip Wadler for inspiring the title of this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISEC’09, February 23–26, 2009, Pune, India.

Copyright 2009 ACM 978-1-60558-426-3/09/02...\$5.00.

based on a multi-tier architecture. This architecture provides performance advantages, allows developers to gainfully specialize on specific tiers, and simplifies system operations. However, in practice, we have found that this multi-tiered design also scatters and tangles concerns such as data management, security and workflow across multiple layers. This scattering and tangling complicates development, deployment, integration, and testing. As a result providers of turn-key web applications struggle to evolve functional features rapidly in response to market conditions and customer demands.

Our goal was to retain the advantages of the industry-standard multi-tier architecture, while mitigating these attendant difficulties. Beginning with a standard J2EE multi-tiered platform, we built a model-driven, interpreted, end-user (visually) programmable platform for web based forms-oriented database applications, called InstantApps, around it. This platform embodies a novel *multi-tiered interpreter*, which is visually programmable from a web-browser. The interpreter uses a domain-specific meta-model which makes typical, common web forms and page flows simple to implement, and enables extensions when needed. The interpreter’s run-time is carefully engineered to reduce interpretation overheads. InstantApps has been used to deliver commercial, high-quality applications at significantly reduced costs.

The main contributions of this work are:

1. A multi-tier, model-driven, interpretive, efficient runtime for web based forms applications.
2. A WYSIWYG ‘point-and-click’ designer integrated with the runtime application platform, enabling instant changes to application functionality.
3. Extensibility using standard, widely-known APIs, for server and client-side logic, including complete customizability of the ‘look-and-feel’ and behavior of the user interface, while retaining the benefits of model interpretation.
4. Quantitative evidence of how the interpretive architecture improves development productivity, along with productivity figures from real-life industrial applications
5. Multi-tenancy and weak meta-circularity¹ of the deployment architecture and exploiting these features in practice.

To set our work in the proper research context, in the next section we present related work comparing our approach and experience with prior work that has similar motivations. In Section 3, we

¹ By meta-circularity we mean that our system (like Lisp) is modeled using the same data structure that it manipulates: this allow the system to be developed/extended through the same interface via which it is used

describe the software engineering problems that arise in multi-tier web-based applications. In section 4 and 5, we describe how we address these problems, through a user-programmable, visual, model-based approach and the interpretive architecture. Meta-circularity and multi-tenancy features are covered in Section 6. In section 7, we present our experience using InstantApps in industry projects, and performance benchmarks in Section 8. We conclude with some lessons that can be learnt from ours as well as similar approaches so that the gains we have seen from the interpretive approach may be applied widely.

2. Related Work

Forms development platforms have a long history, and some of these also use an interpretive approach: The FADS system [18] represents early work in this area, which is actually an interpretive system similar in many ways to ours, but on a VAX VMS platform! A similar approach was later re-used in the academic PICASSO framework [19] and in the ABF (application by forms) tool of the INGRES platform [20]. We will call these ‘first generation’ forms interpreters. (Various form ‘painters’ have also been included in 4GL languages, such as [25], but these are different in that they either generate code or executables rather than interpret specifications.)

In recent years some web-hosted applications have been providing ‘create your own application’ services. Examples of these include Salesforce.com’s Apex platform [15], Coghead, described in [16], as well as others such as Bungee Labs, Jotspot and Nsite. These web based services that allow users to create their own forms based applications over a browser interface, and these applications are immediately available for use. Each of these services are interpretive and, unlike earlier work above, multi-tenant. Some [15] also allow users to add hand-written functionality in their own proprietary scripting languages. We will refer to these as ‘web-based forms interpreters’.

Other visual editors and interpretive runtimes for interfaces have also been proposed in literature and commercially: WebRB [2] is an XML language for describing web applications that manipulate relational databases, which lends itself to visual representation and editing using a graphical editor. The WebRB editor directly manipulates the descriptions of behavior stored in XML which is interpreted at runtime by the WebRB engine which is a PHP program. Similar to WebRB, Statesoft [10], is a commercial product that uses state-charts to modeling user interface interactions. These we refer to as ‘graphical editor-interpreters’.

While InstantApps shares some properties with each one of these related interpretive approaches, there are significant differences with each as well. First generation forms interpreters were not based on a distributed multi-tier architecture (there was no web then). These, as well as more recent, web-based forms interpreters do not offer a ‘wsywig’ ‘point-and-click design’ interface as does InstantApps. Further, all three, including the graphical editor-interpreters, require one to use a ‘development’ interface distinct from the application ‘play’ interface; however, in InstantApps changes are made at the ‘point-of-use’ and instantly enabled.

A very important feature of InstantApps is the ability to embed custom code using standard programming paradigms (JavaScript and Java). Because of this feature, InstantApps allows *completely customizable* user interface behavior, as well as integration of business logic and data manipulation using an industry standard architecture, which is essential for industry acceptance and is not present in any of the earlier approaches. Meta-circularity and

multi-tenancy are also important features of InstantApps and have been exploited in practice.

Recent web-based interpreters are the closest in philosophy to InstantApps, in particular they are also multi-tenant (though we have not seen meta-circularity in these so far). Additionally, while this is not a technical differentiator, we note in passing that while the web-based interpreters are all public, web-based application development services, we are using InstantApps to create industry grade applications that are deployed behind the firewall so that using proprietary databases and integration with legacy/ERP systems is easy, because of its open standard architectures, which the web-based interpreters do not offer.

Traditional model driven development using code generation is also closely related to our interpretive approach, as exemplified by [8, 9]. Also are related generative approaches based on dynamic architectural styles and architecture description languages [3, 4]. In Section 4 we compare the generative route to an interpretive approach in more detail.

Several systems [23, 13] describe a programming model which combines the client and server programming components into one unit, providing session-scoped variables. Finally, the advantages of a visual programming approach in general as advocated in [5], and the idea intentional programming [7] are also expositions that share the underlying philosophy of InstantApps, even though neither of them address the domain of web based multi-tier applications.

Finally, we note that in this paper we report productivity benefits in real-life industry projects using the interpretive approach, which has not been seen before.

3. Multi-Tier Web Applications

Web-based IT applications have had enormous impact, but are difficult to engineer. Drawing on our experiences (from building dozens of turnkey web applications), we summarize the main challenges of building and evolving web-based multi-tier applications.

The current best-practice design for industrial-scale web applications is a multi-tiered style which provides many advantages, including scalability, high-performance, and felicitous division of labour during development. However, it leads to scattering and tangling concerns across layers as we discuss in section 3.2. This leads to 2 main problems:

1. Adding new functional features is difficult, error-prone, and time-consuming, requiring a co-ordination among different teams. This is discussed in section 3.4.
2. It is cumbersome to compile, configure, integrate and deploy new versions. This slows development and reduces agility, which is crucial in business-critical applications. We describe this issue in Section 4.3

We illustrate these challenges (and our solutions later in Section 4 and 4) using a running example, which we now present.

3.1 Motivating Example.

InstantApps was used to automate a web-based real-estate broker consortium. The application involved fairly complicated use cases, with complex, multi-page forms and intricate navigation. The functionality needed to be easily re-configurable after release of the application, as the business model evolves. While the size of the application in function points [11] was modest by industry standards, *i.e.* in the hundreds, it needed to support potentially hundreds of concurrent, distributed users.

In this context, we introduce a simple example feature, which will be used to illustrate our approach through the remainder of the paper. Consider first a form for capturing the sale of a property, including fields for the details of property itself, seller, buyer and the price. If this were all that was needed, there need be only one *Property* table tracking the property sales, created through this form. Now suppose we want to add a cumulative billing functionality to the application, so that buyers and sellers who participate in several transactions would receive consolidated invoices. It is then better to maintain separate tables for buyers, sellers, and property sales, with foreign keys (e.g. *BuyerID*, *SellerId*, *PropertyID*) linking the tables. The same property sale form then needs to trigger access to many tables. In the next section we first describe how this simple functionality is normally created in multi-tier web architectures.

3.2 Layered Multi-Tier Web Architecture

The classic, original (mid-late 90's) servlet architecture conflated the HTML display (view), business logic (controller) and JDBC data access (model) all into one layer, resulting in a complex, unappetizing mush of Java, SQL, and HTML. This style also made it difficult to delegate engineering work based on skills, e.g., designating an HCI person to handle the web page look & feel. The layered MVC pattern² separates user interaction (view), business logic (controller) and data access (model) each into its own layer, leading to a much cleaner, modular, separately evolvable code base, as well as enabling skill-based delegation of work. It also supports horizontal scaling, to accommodate large and evolving feature sets. Each layer will, in practice, include multiple components, reflecting the functional features in the application. In Figure 1, we show multiple views, controllers and models, reflecting different features, with a central 'front' controller handling workflow, dispatching etc. Each layer is implemented using different techniques within the same overall architectural platform; e.g. in J2EE, views are JSPs and JavaScript, controllers involve a controlling servlet forwarding 'action' methods in a number of Java classes, and the model is often a number of Enterprise Java Beans (EJBs). Information is passed between layers using instances of 'form-bean' and 'data-transfer-object' (DTO) classes. A similar set of layers are present in other architecture platforms, such as .NET.

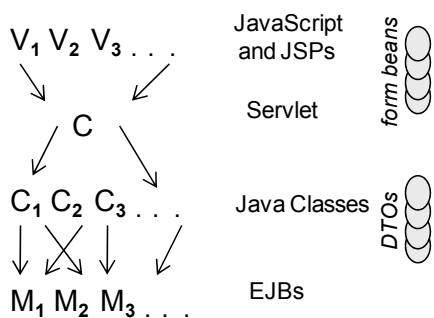


Figure 1: Layered MVC Multi Tier Architecture

The realization of the layered MVC style in current web platforms (illustrated in Figure 1) is quite refined, and complex, with roots in JSPs, Servlets, Java Struts, J2EE, .NET, and the GoF patterns [23]. The full history and rationale of this realization of the MVC layered style is fascinating and instructive, but beyond the scope of this paper, and can be found elsewhere² [14]. What is

² E.g., <http://java.sun.com/blueprints/corej2eepatterns/index.html>

important for our purposes is that implementing a single user feature has implications for all the different layers. This layering, while complicated, actually provides many advantages. It scales very well (can handle a thousands of users and hundreds of requests per second). Layers can be on different machines; processing can also be horizontally distributed across available computing resources within each layer (e.g. multiple cores, CPUs or blades) depending on the transaction profile and load. It also provides design advantages: it allows appropriate implementation approaches for each area and delegation of work based on skills. Layers are substitutable, depending on context, for example, applications can be easily adapted for desktop browsers and mobile clients. As a result, this style is widely adopted, and boasts extremely refined implementations, and is now well-known to literally tens of thousands of developers. Nevertheless, it is a mixed blessing: while it provides some design & operational benefits---it also makes evolution difficult.

3.3 Design: Scattering of Concerns

The MVC architecture described above achieves separation of *presentation*, *control flow* and *information model* concerns from an architectural perspective, enabling any of these to be replaced by alternative technologies or deployed on separate platforms; thus also enabling distributed computing and scalability. However, this also results in functional concerns being scattered across tiers, as illustrated below using our example scenario:

Consider what happens to our example form, when we decide to store the buyer and seller details in separate tables and retain only the *BuyerId* and *SellerId* columns included in the *Property* table. This requires new/changed components: 4 JSPs (for search, result, add and edit of property sales), and several new or changed JavaScript functions, action classes, form beans, data transfer objects, EJBs, and other classes for database accesses well as of course changes to database schemas.

Changing the handling of a single attribute, stored in a column in the database and relating to fields in a few forms (e.g. for entry, edit, search and view of data values of the attribute, possibly in many different contexts), impacts all layers in the architecture.

3.4 Process Complications

The multi-tier style influences the software process, especially downstream (coding and test). This style enables skill specialization, with UI experts building the view layer, and data access experts building the model layer. Consequently it is the norm (rather than exception) to have multiple developers involved in the development of each functional feature. This necessitates a technical integration phase, whereby functional elements are tested end-to-end across all tiers. Multi-tier layering also results in a relatively large set of configurable items per functional feature (the JSPs, Java Classes, EJBs etc. as described earlier) which complicates assembling, compiling and building a running system as a precursor to each iteration of functional system testing. As an example, in the property sale example discussed above, adding cumulative billing (per buyer or seller) requires Javascript/JSP changes for forms in the view layer, by UI experts, data access layer changes, by experts familiar with physical and logical data design, and new controller layer code. These separate changes must be later integrated and system tested.

Thus, the multi-tier style, while enabling specialization and division of labour, also complicates co-ordination, integration, configuration and testing. As a result, cost and cycle times are increased.

Automation can help with the issues described above, provided one can control development and change at the level of functional features. Code generation from domain specific languages or models can be used for greater control. In the following sections we introduce an alternative (and also complementary) interpretive approach; and also compare it with the generative approach.

4. Model Interpretation Approach

4.1 InstantApps WYSIWYG Interpreter

As we have seen, the implementation of functional features are scattered across tiers in a layered multi-tier architecture, and this results in prolonged cycle times for even simple changes. Our overall approach is depicted in Figure 2. Users manipulate data in a database through an application which is rendered through *interpretation* of meta-data by the run-time interpreter, with the meta-data is itself stored in a database. Further, certain users, called designers (i.e. users with ‘design’ privileges) are also allowed to manipulate the meta-data, through the same user interface via a wysiwig approach, and thereby change *and immediately test* application functionality.

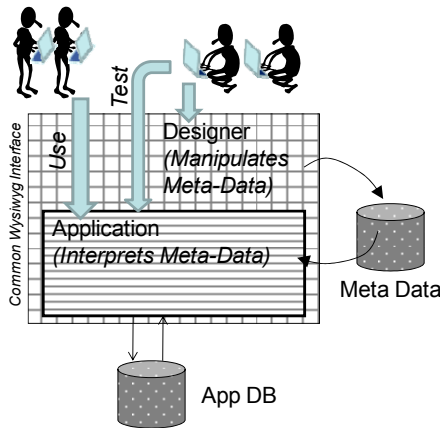


Figure 2: WYSIWYG Model Interpreter

We use scenario from the real estate application. Key features of the platform are underlined: Consider a *Property* form to add, search, edit and delete properties. Starting with a ‘blank’ application, we select the option to create functional menu via ‘*New Tab*’, followed by a “*New Form*” options in a ‘right-click’ menu (highlight no. 1 in Figure 3). We enter a form name, and an empty form is instantly created, with a blank field which we can customize with a fieldname, and type; a fully functional new field immediately appears on the screen and is as well, i.e. is included in the database and in all layers in between (highlight no. 2 in Figure 3). These steps are repeated to add several fields, e.g., address, area, offering price, etc. Having created the *Property* form, similar forms can be created by simply copying and pasting this form onto a newly added blank form before adding fields, where additional fields can be added and unwanted ones deleted. Note that during the process, a new table for this form has been automatically created in the database; additional forms created using copy-paste as above share the same table. The relationship between forms and database tables is discussed later in Section 5.

Many tasks in forms based systems are quite common, and use standardized template, for routine operations such as create, search, result, edit and delete functionality. Pages for these basic templates are available as soon as a form is created, and can be linked to menu items as desired. Each type of form behavior is immediately available when the form is created, i.e. these forms

are all “live” and can be instantly tested and refined if required, and then retested. Changes made to forms are instantly reflected in the application model, and reflected in modifications not just to the GUI, but also to database schema, and all layers in-between as well. The technical details of this core feature are presented in section 5.

For example, in the above scenario, the user can create properties using the ‘add’ page of the *Property* form, then search for them and view the results. If additional fields or field placement modification is required in any of these pages, these can be made directly on the respective working page and tested instantly.

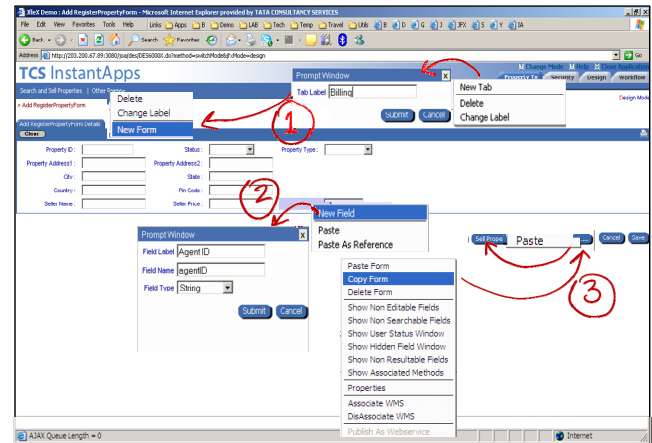


Figure 3: WYSIWYG Form Design in the Browser

Further, as changes made to forms are maintained in a database, they are also immediately visible to all developers working on the project, thus allowing fine grained concurrent development.

Fields can also be copied and pasted between forms, If a field *f1* from one form *F1* (associated with database table *T1*) is copied into form *F2* (associated with table *T2*) by default, this field *f1* is added as a foreign key to *T2*.

Most web applications have multiple forms, linked in different ways for navigation purposes. We include an intuitive way to link forms via copy-paste. For example, one can link a *SellProperty* form by copying this form and pasting it ‘as button’ on the edit page of the *Property* form, thus providing a link that offers a property for sale (highlight no. 3 in Figure 3). Further, if the *PropertyID* field was copied from one of these forms to the other during design, at run-time the value of this field in the called form will be automatically filled with value from the calling form.

Changes made to forms can include many other aspects of form behavior; such as associating pre-defined search criteria with fields on a search form, for example to create a form for displaying only ‘available’ properties.

Apart from forms, applications also include program logic, either for validations on screen or for functional processing en-route from the screen to/from the database. The InstantApps platform allows for form-specific custom logic to be uploaded and attached to various events during form processing. This can include screen validation functions in JavaScript as well as server-side logical processing in Java.

Finally, the look-and-feel of the user interface can be customized by replacing the styles and widgets used by custom ones, a process of creating a new ‘skin’ for the application (Section 5.3).

In the next section we continue this example and illustrate more complex functionality including how custom functional logic is

attached to forms, altering the look-and-feel of the user interface via ‘skins’, as well as creating forms on multiple tables (again using the intuitive copy-paste ‘as reference’ feature).

Summarizing the key features of InstantApps (some illustrated above and some which will be covered in later sections along with their implementation and usage):

Forms & Menus:

- (a) Wysiwig creation of multi-level functional menus, forms and fields, automatically creating tables and columns appropriately, including forms on *multiple* tables (Section 5.2).
- (b) Pages providing Create, Search, Result, Edit and Delete templates instantly functioning for each form created.
- (c) Copy-paste of forms and fields, thereby ensuring consistency and intuitively creating foreign key relationships between tables. Navigational links between forms are also created likewise; when navigating to a form, values are automatically copied as indicated by the foreign-key relationships.

User Interface & Logic

- (d) Custom client-side validation and server-side processing logic can be attached to controls on forms using standard tools, i.e. JavaScript and Java (Section 5.2).
- (e) The look-and-feel can be customized with custom styles and widgets, by creating application specific ‘skins’ (Section 5.3).

Concurrency, Multi-Tenancy, and Meta-Circularity

- (f) Design changes are instantly reflected and can be immediately tested. Changes are also immediately visible to all other developers, enabling fine grained concurrent development.
- (g) Multi-tenancy and weak meta-circularity (Section 6).

4.2 Implications for the Development Process:

The availability of wysiwig application development comes with all the traditional advantages of rapid prototyping platforms, including *reduction in cycle times* and *reduction in defects* due to *better requirements elicitation*, because of immediate feedback on changes and addition. (We note here that because of these features, InstantApps is particularly suited for *agile* development methodologies [21, 22], such as extreme programming and pair programming; however we do not cover that aspect in this paper.)

In addition, some special process complications due to multi-tier architectures, as described in Section 3.4, are ameliorated, especially those relating to deployment and testing:

Changes made to forms, or flows between forms, are stored in a model which is interpreted by the run time. Changes made to the model result in changes in different layers of the system. For example, a change made to add a field as a form, changes not only the visual appearance of the form (view layer), but also the database schema (model layer). Likewise, changes to navigation between forms result in changes in the form (view layer) and the sequencing logic (controller layer).

In current multi-layered J2EE or .NET implementations, such changes require multiple steps: 1) coordinated changes in multiple layers, followed by 2) the unit-testing of each layer, followed by 3) coordinated deployment of new code at each layer, 4) the integration testing of each the newly integrated configuration,

followed by 5) a system test. In the InstantApps interpretive approach, the integrated behavior of all layers is enabled instantly with each change; so we directly go to step 5, system test, right after the changes are made, which results in significant productivity gains, as documented in Section 8. Further, iterative requirements prototyping and development become much easier. The compressed development process is illustrated in Figure 4.

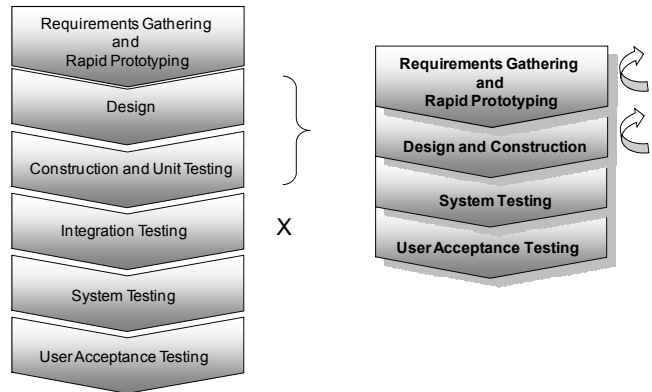


Figure 4: Compressed development cycle

Other process benefits arise from multi-tenancy, which enables configuration management as well as re-use, and weak meta-circularity, which allows platform extensions; these features and their use is described in Section 6.

4.3 Comparison with Code-generation

The model-driven approach has been used earlier with code generation. Instead of generating code, we have taken an interpretive approach. There are similarities but also significant differences that come about because of this:

In the multi-tier context, one must write code generators to generate code for each layer, from a common model; after generation, and compilation, this code would then need to be redeployed, integrated, and the system would have to be reinitialized and restarted. If the code generators in each layer are known to be a) correct and b) produce code that correctly inter-operates with code generated in other layers, the unit and integration testing phases can be eliminated; however, the deployment, reinitializing, and restarting steps cannot be eliminated. Thus generation does shorten the cycle, but we can do better.

In contrast, our interpreted “design and immediately play” approach of allows functional changes to be made from within a running application, so they can be tested just as soon as the change is made. Each developer can immediately see the changes made by other developers. Thus, we have continuous integration with a single, unified application model being interpreted by all the layers in the run-time. Changes are implemented without changing running code, avoiding the build and deploy process so that system testing can be done at any level of granularity. Thus in the multi-tier web-based setting, the interpretive approach offers significantly more benefits in practice, provided it can be implemented efficiently, as is described below.

5. Implementation of InstantApps

The challenge here is to realize an interpretive approach within a multi-tier layered architecture, while not losing the inherent advantages of the standard multi-tier design.

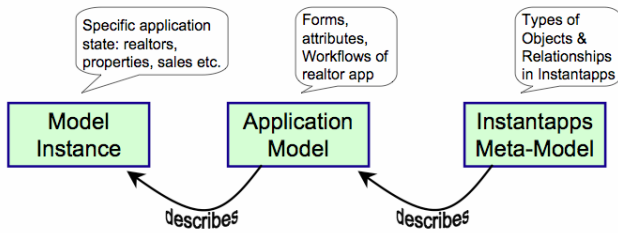


Figure 5: Models in InstantApps

To explain the concepts involved our model driven interpreter and to avoid confusing the “model” in MVC with the “model” in model-driven architectures (MDA), we clarify with figure 5. The *Model Instance* on the left indicates the specific state of a particular instance of the application, e.g., referring to the state of the real-estate market in a particular area at a particular time. This is the “model” in MVC. The *Application Model* is the description, in this case of the forms, fields, and attributes, data and flows of the real-estate application used in our discussion. Finally, the *Meta-model* describes the types of entities (forms, fields, attributes, objects, etc) that exist in an InstantApps-based application, their properties, and relationships between these. Incidentally, a relational database stores both the model instance as well as the application model; this supports a type of meta-circularity which we discuss further in Section 6.

5.1 Model Interpretation across Tiers

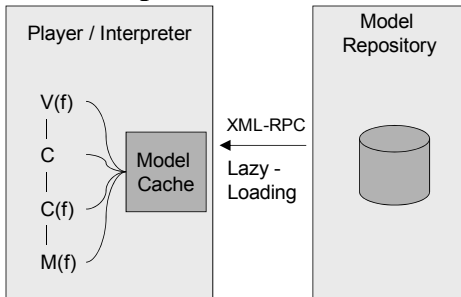


Figure 6: Model interpretation across tiers

The standard multi-tier style scatters the implementation of a single conceptual form across multiple source code components, hard-coded into JavaScript, JSPs, forwarding ‘action’ classes and EJBs, along with ‘form-beans’ and DTOs for data transfer between layers, as we outlined in Section 3.2. InstantApps seeks to preserve this multi-tier architectural style in an interpreted runtime for all the reasons mentioned above. Every tier also exists in the interpreted InstantApps run-time. However these tiers interpret the data stored in the application model that describe the forms. From these, the classes in all the tiers must be emulated by the run-time interpreter. We show this schematically in figure 6 as the MVC layers parameterized by the description of form *f* read in from the application model repository.

In fact, InstantApps has only a small number of parameterized components (for add, edit, search, result pages): *three* action

classes, *four* EJBs, and *four* JSP. Further, there is only *one* parameterized DTO class, and just a few parameterized form bean classes (again for add, edit, search, and result pages), with the DTOs and form beans created via reflection after interpreting the application model. Such interpretation could be normally expected to suffer from heavy overheads of reflection, interpretation, and model traversal. In a later section we describe why we believe these overheads are minimized, along with performance figures.

5.2 Meta-Model for the Domain

We model the functional behavior of a web-forms processing application via a meta-model, as shown in Figure 7.

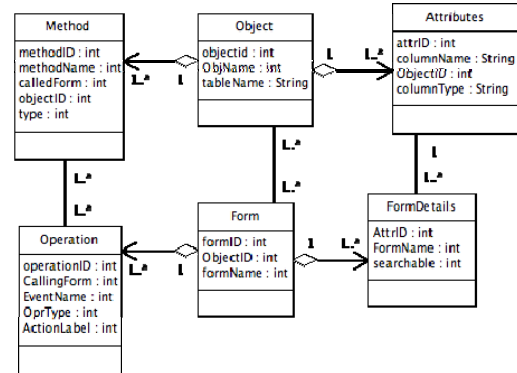


Figure 7: Elements of an InstantApps meta-model

In this meta-model, *Form* is the central meta-object. A form can be associated with one or more real-world *objects*. Thus a form can be associated with a property object, and used for registration, update, search etc. *Objects* have *attributes*, thus properties can have a seller, a price, an address etc. *Objects* are stored in a specific table in the database, specified by the *tableName* meta-property. Each *Form* can have several *FormDetails*, roughly corresponding to fields in the form; each of these *FormDetails* correspond to *attributes*. An *attribute* may occur as a field in several different *forms*. One can also have fields representing *attributes* from different *objects* in the same *form*. Some fields may be searchable, whereby the user can fill in these fields with values and ask for *objects* whose *attribute* values match the entered values. *Operations* are associated with one or more *methods*, which can represent server-side functional logic coded as Java methods associated with objects: for example, a method might change the selling price of a property. Methods can also represent page-flows, i.e. navigable links between pages.

As described in the previous section, when a user creates forms and fields from a wysiwig designer running inside the web-browser; in the background this also makes *asynchronous* AJAX [23] calls to the server to effect changes in the application model.

We now describe how this meta-model is instantiated into a specific application model, and how changing the application model (in MDA fashion) changes its behavior. Figure 8 depicts modeling of an application (i.e. an Application model) using several relational tables.

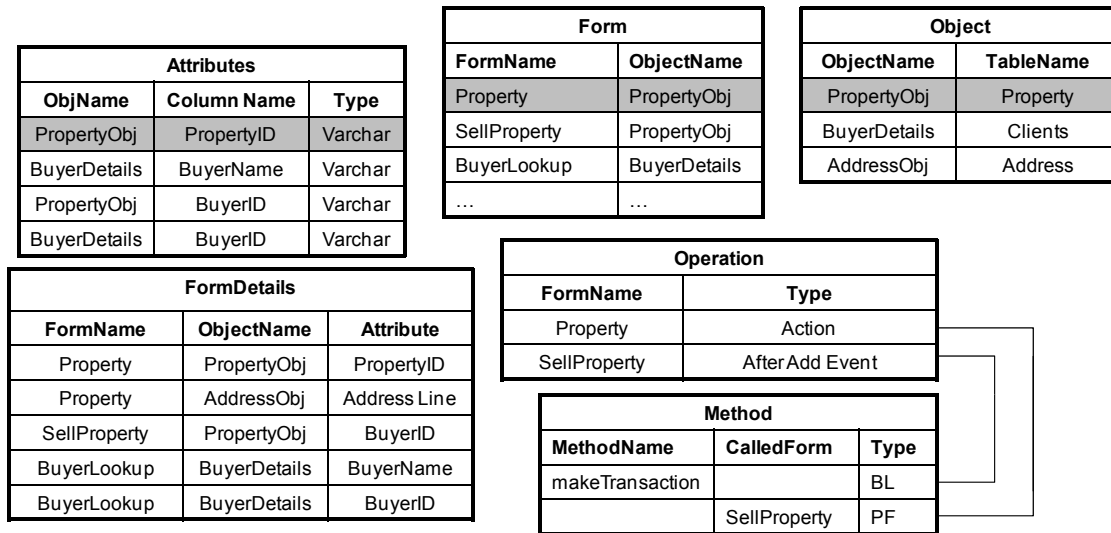


Figure 8 : Application Model as a set of Relational Tables

Basic form model

The grey rows describe a *Property* form associated with a *PropertyObj* object, which in turn links to a *Property* table.

Forms on multiple tables

A form can be associated with more than one object: As indicated in the *FormDetails* model-table, the *AddressLine* field in the *Property* form comes from the *AddressObj* object while others are from *PropertyObj*. InstantApps checks that the underlying tables have at least one common column (by name, e.g. *PropertyID*). This is interpreted as an *n* to *1* foreign key relationship between the referencing and referenced tables, so that search, edit, add, delete functionality on the form is unambiguously defined.

Navigation between forms

Consider extending the above form to add the functionality to sell & bill for the sale: In this case, we want to add a “sell property” form where a buyer for a property can be assigned. This new form *SellProperty* is based on the same object as before, i.e. *PropertyObj*, and includes a field *BuyerID* (tuple 3 in the *FormDetails* model-table). Note: this would have been created by copying and pasting the *BuyerID* field from the *BuyerLookup* form (tuple 5), thereby creating a column in the *Property* table, as also a pick-list on this form from which a buyer can be selected by name.

Now, this new *SellProperty* form needs to be linked to the page where the property is registered. This link is represented by an entry in the operations meta-table, and the corresponding tuple in the *Method* model-table, of type “PF” (page flow), indicating navigation from the *Property* form to the *SellProperty* form. Note that this is achieved by a copy-paste of the form as a whole, as described earlier in Section 4, highlight no. 3 in Fig. 3.

Integrating custom functional logic (code)

Fig 8 also shows how the application model captures hand-written “background” business logic (BL) operations. In this case, when the property is sold, billing records need to get generated. This is done by the *makeTransaction* server-side method, as indicated by tuple 1 in the *Method* model-table. This operation is triggered “after add” (tuple 2 in the *Operation* meta-table) of the *SellProperty* form, i.e. this handwritten code is called when the

‘Add’ button is pressed on the *SellProperty* form, after the actual sale record has been created.

5.3 Customizing the User Interface

It may appear from the examples given above that our template based approach also restricts the user interface ‘look-and-feel’ so that all applications look pretty much like Figure 3. This is *NOT* the case, and a *key* feature of InstantApps is the ability for applications to have custom user interfaces, by allowing complete freedom to developers to override the default UI behaviour and create new ‘skins’ at an application level, as well as form specific UI behaviour. At the same time all the benefits of an interpretive approach remain.

Figure 9 shows an application created in InstantApps using a new ‘skin’. Note the major differences between this UI ‘skin’ and that of Figure 3 (which we call the “classic” skin): Apart from fonts and colors, there are multiple forms on the same page, some decorative wall paper on the left, the menu has been repositioned, and there is also a Google Maps ‘mash-up’ integrated on the page. This presents some challenges: how do we retain the model-driven convenience, while retaining the ability to create such rich interfaces

We now describe how this is achieved. When we create forms using the application designer, in addition to these being available immediately for use, these forms are also available as a JavaScript ‘AJAX’ [23] API that connects to the InstantApps server, in much the same way a Google Map is made available for use within a web page. The API also allows *events* to be published by one form and subscribed to and captured by others (e.g. the ‘find’ event on a *search* form can be captured by the *result* form so that the search parameters are passed to it; similarly an ‘edit’ event on the result form can be captured by an edit form etc.).

Recall from the previous section that InstantApps allows custom code to be uploaded into the platform; in addition to server-side Java code as described there, this also includes JavaScript code that can be uploaded and attached at an application level or specific form level (i.e. this code is included for all forms in the application or for a specified form.) UI customizability relies and builds on this capability, i.e. to render multiple forms on the same page, form specific JavaScript code is uploaded which overrides the default behavior of the ‘classic’ skin and instead renders

multiple forms as in Figure 9, positioning them as desired and linking them via events. It is important to note that the forms rendered by API calls are fully model driven, i.e. if additional fields are added using the application designer, these also show up in these forms; similarly if navigational links between forms are set up, the same flows are enabled. Further, such flows are aware of whether the target form has already been rendered on the same page or not, so that the behaviour is intuitive.

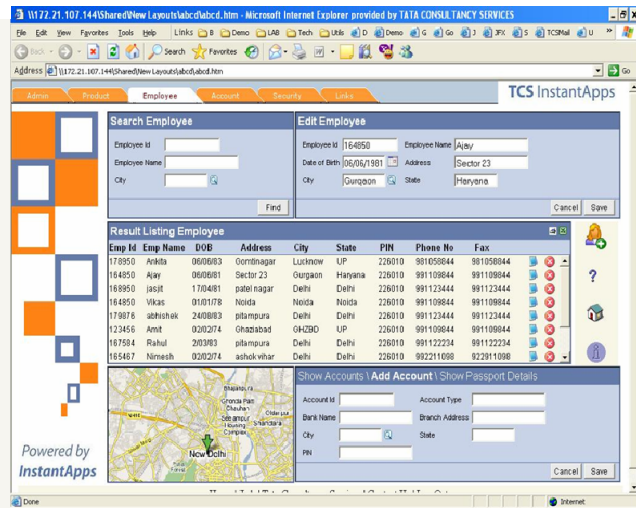


Figure 9: A custom user interface ‘skin’

Additional features such as the wallpaper on the left and custom icons on the right can be added as well using JavaScript code, or calling the Google Map and linking it to form data, which is largely independent of InstantApps. The API also publishes methods for retrieving the menu structure, from which a custom menu can be written in JavaScript. Further, some of these functions can be added in a separate JavaScript file attached to the *application* rather than a specific form, resulting in a new ‘skin’.

Finally, the atomic behaviour of fields, labels, buttons, menu items etc can be modified by overriding JavaScript functions for each of these widgets. Function overriding is achieved using JavaScript introspection to parse function names against a naming convention, e.g. the default atomic function to create a button, say `CreateButton()` is overridden by including a new function called `CreateButton_<AppName>()`. The default function `CreateButton()` checks whether such a function exists before doing what it normally does, and if so overrides its normal behaviour by calling the latter function instead. Similar overrides for a specific form are effected by writing a function called `CreateButton_<AppName>_<FormName>()`.

Using the above features, virtually any user interface can be created while still using model driven forms that are instantly configurable (e.g. new fields can be added instantly, etc.). In fact, the JavaScript API is callable from *any* web page, similar to the Google Map API, so InstantApps forms can actually be rendered *outside* of the instant apps platform, as ‘mash-ups’.

6. Meta-circularity and Multi-tenancy

What is meta-circularity? LISP, for example, is meta-circular: the data structure used to represent LISP programs, and the data structure manipulated by LISP programs are the same: lists. This allows LISP programs to be generated and executed in the same runtime, thus enabling e.g., powerful macros. InstantApps is

meta-circular in a weaker (but still important) sense, as in fact, InstantApps itself is bootstrapped on InstantApps:

Figure 10 shows the deployment configuration of InstantApps. The runtime manipulates the application data. This runtime (IR) interprets the application model which is also stored in a relational database, which is managed by *another copy* of the InstantApps runtime. In other words, *the semantics of an InstantApps-based application are controlled by the same type of data that it manipulate, and manipulated by a copy of itself.*

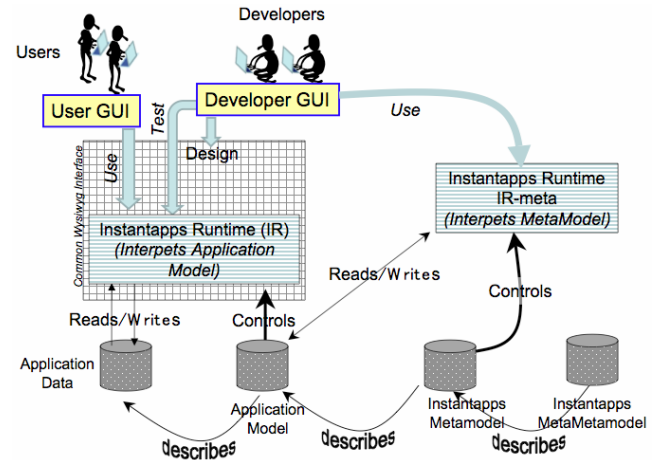


Figure 10: Deployment architecture

Exploiting meta-circularity

We describe an example of this from a real case: The need was to develop an application that manages questionnaires. The questions and their allowable answers would be set up in a set of tables via forms. Each such questionnaire would need a form, whose details are not known when the application is developed, so we had to dynamically assemble, forms at runtime rather than at design time. We exploited meta-circularity here: InstantApps APIs are called within the application logic to create models of the questionnaire forms, i.e. the application updates its own application model.

Multi-tenancy: As all application functionality is defined by meta-data, several InstantApps applications can coexist within a single deployment of the platform: the runtime maintains a parameter ‘AppName’ in the meta-model that partitions the meta-data into virtual private data sets, each representing a separate application. Additionally, each application is associated with a different application database *schema* (this mapping also being stored in the model), and uses the appropriate database connection. When a user logs in, the application name needs to be specified along with the user name. In this manner, InstantApps can be used as a multi-tenant platform for hosting many applications, with many attendant advantages.

Exploiting multi-tenancy

Multi-tenancy in InstantApps has been exploited in practice in two ways: (a) to enable configuration and release management and (b) to support application level re-use. In practice users often maintain two (or more) InstantApps applications for the same functional need, i.e. the production instance (where design functionality is restricted or turned off) and a development instance. Thus the development version is always ahead of the one in production. When changes need to be incorporated in production, *logs* of changes made to the application model are applied to the production environment along with updating any application specific custom

logic (and this is provided as a semi-automated facility) Further, before significant changes are to be made, the entire application version is cloned as a copy which can be reverted back to if needed during the development process. The same features of application import and cloning have been used for re-use at the application level, i.e. when a different set of users would like to create a copy of an existing application and modify it for their own particular usage patterns on their own data.

7. Productivity Case Studies

InstantApps has been used commercially to build apps in a number of domains. These applications are moderate in size, as measured in function points [11] or number of tables and screens, and range in the 100s of function points and fewer than 50 tables and screens. (Very large application suites can have thousands of tables and screens and over 50,000 function points). We describe some of the applications briefly:

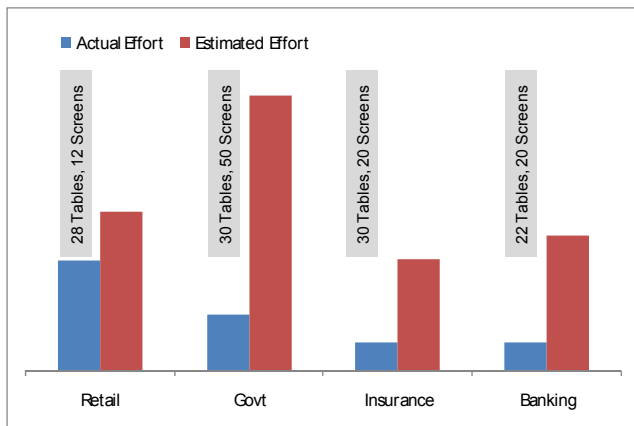


Figure 11: Productivity Improvements

The first case is a support application used to manage the roll-out of new software across thousands of branches of a retail chain. The second is also a program management tool helping different teams collaborate with respect to change requests and enhancement requests in running a large e-government data centre. Next is a training module developed to manage the training of sales agents of an insurance company. The fourth was a re-engineered collaboration application in a large bank. The last case is the real estate case from which the example used throughout the paper was derived.

The productivity gains in these projects from using InstantApps are evident from the figures in Figure 11. In all cases, the actual effort was much less than the estimate. Here the *estimated effort* figures were obtained by taking the *lowest* of different estimates available, including an *a priori* one produced during requirements analysis, and one based on a *posteriori* calculation of the function point size [11]. We assume an industry standard figure of 12 hours per function point. The *actual effort* is the amount of effort spent developing the application using InstantApps, i.e. the number of people deployed on the project times the *end-to-end* elapsed time of the project, including all phases. In certain cases, where the specifications from an earlier version of the application were available at the start of the project, or when the work actually was to re-engineer an existing suite (e.g., the insurance and banking cases), the InstantApps project did not have to do requirements analysis. To be fair in such cases, the actual effort was *increased* (i.e. penalized) by 40%.

It is clear that in spite of these conservative measures, using InstantApps provides *significant* benefits in productivity. Finally, while these are all small applications by industry standards, this is actually encouraging since it opens up the possibility of breaking large projects in to smaller pieces that can be done significantly faster; with the expectation that even in the event of major re-work resulting from issues arising while integrating the pieces, we would still gain – however, this has yet to be tried in practice.

8. Performance Benchmarks

We evaluated the performance of a simple application developed in InstantApps. The application consists of a single table ‘Customer’, with close to 4 million records (3870913 to be precise) generated randomly. Search-result, edit, and insert functions were performed on two machines (each having Intel Xeon 2CPU, 3.2 Ghz, 2 GB RAM), one used as a database server and the other used as an application server. The tests were ran successfully upto 150 concurrent users with very few requests failing to complete. For 200 or 250 concurrent users, the HTTP connection would time out most of the time, leading to the conclusion that the system performance limit had been achieved.

Throughput obtained ranged from 50 to 120 transactions per second, for different types of transactions, with response times of 1-2 seconds for inserts to 7-8 seconds for wildcard search.

We also compared the above with tests on a similar test application hand-coded using the standard multi-tier J2EE architecture and performance results were comparable; these results are also similar to those reported in [1], where performance of alternative implementations for multi-tier J2EE applications is analyzed.

These preliminary results suggest that InstantApps does not entail significant performance overhead (despite the use of interpretation) as compared to the hand-written code. We believe this is because of several optimizations we have made in the implementation, as outlined below:

Optimizations for performance

Fixed number of classes re-used across forms: In a standard J2EE approach, multiple JSPs, controller classes, form-beans, and data transfer objects would be used for different forms, along with some reflection to make the code more general and maintainable. In our interpretive approach, however, we have a small fixed number of JSPs, form-beans, DTOs etc., thus actually *reducing* the number of reflection calls for object allocation. Also, as object creation in Java is expensive many JVM garbage collectors retain class instances as “skeletons” when they are out of scope; which are re-animated when a constructor is called.) In interpretation we regularly reuse the same Java class for different forms, which benefits from such re-use at the JVM layer even more than usual.

Efficient access to meta-data: The meta-data is stored in relational representation in the model repository: To avoid repeated and expensive retrieval and navigation of this data, it is read in and cached in an in-memory object-based structure. This provides fast and efficient navigation of the meta-data for the interpreter.

System Footprint: In the interpretive approach, the memory footprint of the program code is smaller because of the significantly lower number of classes. Classes are allocated in permanent memory that is never garbage collected. This increases the amount of heap memory available, and this increases JVM performance.

Finally, the biggest overhead remains database access at the model instance layer, for which our implementation is identical to the standard approach, i.e. dynamic SQLs in JDBC, so we do not add any overhead here.

9. Conclusions & Future Work

We have described InstantApps, an interpretive architecture for multi-tier applications that uses efficient runtime model interpretation and features an integrated ‘wysiwig ‘point-and click’ design editor. A key feature as compared to earlier as well as recent work on interpretive forms development environments is the ability to integrate custom methods, especially to support completely custom user interface ‘look-and-feel’. Efficient implementation of interpretation and reflection ensures that performance does not suffer, and performance benchmarks have been provided that support this.

Our approach has weak meta-circularity features, that allow full integration of user and design view and enables fine grained concurrent application development in a multi-developer setting.

Further, we have described how the InstantApps architecture improves development productivity for multi-tier applications, and substantiated this claim with productivity figures from real-life projects demonstrating *significant* improvements.

While we have used InstantApps, we expect similar benefits in some measure to accrue using other interpretive approaches also, as referenced in Section 2; and in conclusion we submit that such approaches are ripe for wider adoption and further research.

Future work remaining in InstantApps, some of which is ongoing, includes (i) support for finer grained configuration management and re-use (instead of entire applications) (ii) tracking requirements to functional features and automating their release into the production environment (such functionality has been developed using InstantApps in cases, but it is not built in), (iii) visual modeling of workflow (this is already done but omitted from the current paper as it has also been done in other tools), (iv) visual modeling of server-side functional logic (this work is ongoing), (v) semantically describing and importing functional models from one interpretive environment to another (e.g. from InstantApps to Salesforce.com) is also an area for deeper research.

10. REFERENCES

- [1] Cecchet, E., Marguerite, J., and Zwaenepoel. W., Performance and Scalability of EJB Applications. *Proceedings of OOPSLA 2002*.
- [2] Leff A., and Rayfield, J.,. WebRB: Evaluating a Visual Domain-Specific Language for Building Relational Web-Applications. *Proceedings, OOPSLA 2007*.
- [3] Oreizy, P., et al. An Architecture-based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*. 14(3), p.54-62, May-June, 1999.
- [4] Medvidovic, N. ADLs and Dynamic Architecture Changes. *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*. p.24-27, San Francisco, CA, October 14-15, 1996.
- [5] Harel, D., "On Visual Formalisms", *Comm. Assoc. Comput. Mach.* 31:5 (1988), 514-530.
- [6] Executable UML, 6, S.J., Balcer M.J., *Addison-Wesley* New York, 2002.
- [7] Simonyi, C., Christerson, M., Clifford S., Intentional Programming, *Proceedings OOPSLA 2006*
- [8] Frankel, F., Model Driven Architecture: Applying MDA to Enterprise Computing, *John Wiley & Sons, Inc.*, 2002
- [9] Kulkarni K., and Reddy S., - A model-driven approach for developing business applications: experience, lessons learnt and a way forward, *1st India Software Engineering Conference ISEC 2008*
- [10] Dobrowolski J., and Kolodziej J., A Method of Building Executable Platform-Independent Application Models: *OMG's MDA Implementers' Workshop*, 2004 .
- [11] Function point counting practices manual – International Function Point Users Group (<http://www.12.org>)
- [12] Web Programming Without Tiers. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. To appear in the post-proceedings of *FMCO '06, Springer-Verlag Lecture notes in Computer Science*, LNCS 4709.
- [13] Atkins, DL and Ball, T. and Bruns, G. and Cox, K., 15: a domain-specific language for form-based services, *IEEE Transactions on Software Engineering*, 25(3), 1999.
- [14] Alur , D., Malks , D., Crupi, J., Core J2EE Patterns: Best Practices and Design Strategies, *Prentice Hall*, 2001
- [15] Fisher, S. 2007. The Architecture of the Apex Platform, salesforce.com's Platform for Building On-Demand Applications. In *Companion to the Proceedings of ICSE 2007*.
- [16] Olsen, G. 2006. From COM to Common. *ACM Queue* 4, 5 (Jun. 2006), 20-26.
- [17] Rowe, L.A. and K.A. Shoens. A Form Application Development System. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*. 1982.
- [18] Lawrence A. Rowe, A retrospective on database application development frameworks, *ACM SIGMOD Record*, v.21 n.1, p.5-10, March 1992
- [19] L.Rowe, et.al. "The PICASSO Application Framework," *Proceedings 1991 ACM Symposium on User Interface Software and Technology*, Hilton Head, SC, November, 1991.
- [20] INGRES ABF (Application By Forms) User's Guide, Ingres Corporation, Alameda, CA, June 1990.
- [21] Pair Programming vs. Side-by-Side Programming J Nawrocki, M Jasinski, L Olek, B Lange - *Proceedings of EuroSPI*, Budapest, November, 2005
- [22] Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional (1999).
- [23] Linda Dailey Paulson, Building Rich Web Applications with Ajax, *IEEE Computer*, v.38 n.10, p.14-17, October 2005
- [24] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.
- [25] Albert Lulushi, *Oracle Forms Developer's Handbook*, Prentice Hall PTR, Upper Saddle River, NJ, 2000