

Entity, Boundary, Control as Modularity Force Multiplier

George Heineman
WPI
heineman@cs.wpi.edu

Jeremy Denham
WPI
jdenham@cs.wpi.edu

Abstract

The Entity/Boundary/Control (EBC) paradigm provides a fundamental approach to modularizing software systems. While it is most commonly associated with GUI-based Object-Oriented (OO) applications, its principles are more widely applicable. If one follows an OO design methodology using EBC, the final system implementation often includes “artificial” code that aims to maintain loose coupling among the EBC constituents while supporting the flow of data as needed. Rather quickly, such code becomes brittle and weakens the overall coding effort. In this position paper we suggest how AOP and FOP can eliminate (in different ways) much of the artificial code and enable the implementation to more closely follow the designers’ intent.

Categories and Subject Descriptors D [2]: 10 Design

General Terms Modularity

Keywords Design, Features, Aspects

1. Introduction

The essential strategy for designing complex software systems has been to decompose the larger system into smaller subsystems which are then composed together. While there are numerous approaches one could choose for the initial decomposition, there invariably are concerns which cannot so cleanly be encapsulated by the *dominant decomposition* selected by the designers [1]. Numerous researchers, most particular those investigating Aspect Oriented Programming (AOP), have described examples that show the benefits of using a different modularization technique. What often remains lacking is a discussion (that must somehow take place) to focus on identifying when multiple modularity mechanisms can enhance each other’s effectiveness.

In this position paper, we aim to show how one specific object-oriented design methodology – Entity, Boundary and

Control (EBC) – should enhance the effectiveness of two specific modularity mechanisms: AOP and Feature-Oriented Programming (FOP). Similarly, using these advanced modularization in combination with EBC enables EBC to maintain a clean separation in its constituent parts. There are still “gaps” in the technology that enable designers to take full advantage of the compatible modularity mechanisms, and reducing this gap should be on the research agenda for our community.

In the following sections, we briefly review the EBC model and describe some of its core variations. We then explore the challenge facing EBC coders in maintaining a clean separation and low coupling between the designed classes. We present two discussions on the use of AOP and FOP to eliminate these challenges. We conclude with some ideas for improving the coexistence of multiple modularity mechanisms.

2. Entity, Boundary, Control

Object-oriented Design methodologies have promoted the use of *Entity*, *Boundary*, and *Control* objects for well over a decade [2, 4]. Also known as the Model, View, Control (MVC) paradigm, EBC is a pervasive technique that separates responsibilities in software to avoid overly restrictive coupling that otherwise might occur [4]. While EBC has most commonly been associated with GUI programming, it can also be applied to separately manage the input, processing, and output of software [6].

Both Jacobson [2] and Cockburn [3] describe an approach to identify objects from Use Cases by defining three overall divisions: Entity, Boundary (or Interface to Jacobson), and Control. Entity objects represent the persistent data used by an application. Boundary objects provide the functionality to interact with the environment and receive requests from system actors. Control objects contain functionality “not contained in any other object” and encapsulate business logic; in other words, the Control objects are responsible for “realizing” the use cases. These divisions are reflective of the EBC division that appeared quite early in the evolution of object-orientation, starting with SmallTalk.

Domain experts have considerable expertise in using inheritance to capture a *model*, the rich information to be stored in Entity objects. HCI experts show how to assem-

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACoM '09 October 26, Orlando, Florida.
Copyright © 2009 ACM [to be supplied]...\$10.00

ble user interfaces from GUI frameworks containing a rich set of classes and behaviors that decouple the model from the view presented to the users. But the complex logic found in Control objects can quickly be unmanageable because of the inherent limitations of the basic extension constructs in OO programming languages. Since business logic is encapsulated within control objects, EBC may actually be an impediment to the proper reuse or extension of business logic. Rather quickly one sees the limitations of using inheritance (a typing mechanism) as a means of capturing the way that one (complex) behavior is related to, or extends, another; this is especially true when one requires multiple sets of simultaneous extensions.

There are two primary communication patterns in EBC. In the *Control-centric* version shown in Figure 1, the boundary objects detects a user interaction and pushes an event to a Control object. The Control object then accesses (and possibly updates) Entity objects. When its logic is complete, the Control object updates and refreshes the Boundary object to reflect the change. In the *Entity-centric* version shown in Figure 2, the Control object receives an event as with Control-centric, but the Entity objects themselves are responsible for publishing changes. Boundary objects register themselves as *listeners* to the Entity objects, and the designer must choose whether to publish each individual change to an Entity object or aggregate a set of changes to be published as a unit.

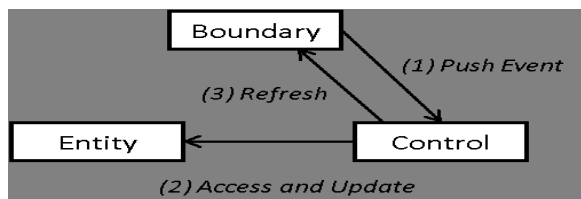


Figure 1. Control-centric EBC behavior

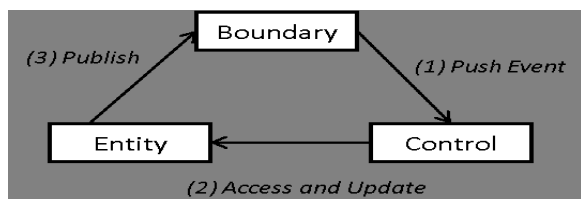


Figure 2. Entity-centric EBC behavior

While it seems we have begun to discuss implementation concerns, we believe that these issues are both architecturally significant and hard to alter once the design phase is underway. Additionally, within the same code base, the designer might choose different communication patterns based upon the needs of the application domain. There is one final concern. While the Entity and Boundary objects are often designed within rich class hierarchies, most of the Control objects seem unable to be designed to take advantage of subclasses. When two Control objects perform quite similar

```
private JMenuItem getExitMenuItem() {
    if (exit == null) {
        exit = new JMenuItem();
        exit.setText("Exit");
        exit.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_Q,
                Event.CTRL_MASK, true));

        exit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // TODO Auto-generated method stub
            }
        });
    }
}

return exit;
}
```

Figure 3. Sample generated code for Menu Item in Eclipse GUI builder plugin

tasks, they often are coded using copy/paste as stand-alone classes. The lack of class relationships between controllers often leads to brittle controllers that must each be modified when changes occur.

EBC is a successful strategy for separating the core responsibilities found in an object-oriented system. However, experience shows that much of the coded implementation governing the communication between the constituent parts is brittle. In this paper we consider how AOP and FOP modularization mechanisms can be used to improve the situation. To overcome the limitations imposed by thinking solely in terms of communicating objects, we must address the following practical concerns.

2.1 Use design tools without embedding business logic

There are numerous development environments that allow programmers to construct Boundary classes graphically. For example, the NetBeans development environment from Sun Microsystems allows users to drag-and-drop GUI widgets to construct GUI classes. The Eclipse environment has similar plugins. What these environments have in common is the ability for the user to create event handlers within the GUI design tool that respond to events generated by the GUI, such as selecting menu items or clicking on buttons. In Figure 3, note how the Eclipse GUI design tool creates an anonymous class to be the event handler called in response to the selection of the “Exit” menu item. Other GUI building tools provide similar functionality. However, developers must be careful not to include business logic arbitrarily within the `actionPerformed` method, otherwise the clean separation between Boundary and Control objects is lost. How do we take advantage of the power of GUI design tools without inadvertently inserting business logic which otherwise should be placed only in Control objects?

2.2 Assemble Control Logic

Control objects realize the business logic as specified by system use cases. As mentioned earlier, often the coded Control objects are unable to share implementation fragments lead-

```

if (d.isModified()) {
    int x = JOptionPane.showConfirmDialog(null,
        "Do you really want to close this window?",
        "Unsaved Changes", JOptionPane.YES_NO_OPTION);
    if (x == JOptionPane.NO_OPTION ||
        x == JOptionPane.CANCEL_OPTION) {
        return;
    }
}

```

Figure 4. Code fragment to force user to confirm action

ing to dispersed code fragments. For example, an application could have two Use Cases *CloseDocumentWindow* and *ExitApplication*. In both Use Cases there is going to be a check to see if the current document has been modified; if so, the user must verify that the document can be closed without saving changes. Both controllers will be coded to include a code fragment such as found in Figure 4. However, the Control object for the *CloseDocumentWindow* Use Case will only check to see if the current document *d* is modified while the Control object for *ExitApplication* will cycle through all open documents to execute this logic. Note that this trivial example only has a few lines of code, but more substantial logical sub-steps can be found easily in real applications.

Even though these controllers are quite similar, no attempt is made to relate the two Control objects using inheritance. Indeed, any attempt to do so would lead to *implementation inheritance*, the undesirable situation where a subclass relationship is defined solely for the purpose of reusing a specific method implementation.

There is an opportunity to identify shared sub-steps within two or more use cases; this situation occurs quite frequently in real applications. The trouble is that the implemented code fragments for these sub-steps cannot readily be shared among multiple Control objects. Given two Use Case behaviors which share a specific action a_{shared} , it is possible to describe the linear logic within the corresponding Control objects as $B_1 = a_1; a_2; \dots; a_{shared} \dots; a_n$ and $B_2 = b_1; b_2; \dots; a_{shared} \dots; b_m$. How do we take advantage of alternate modularization mechanisms to assemble these behaviors from encapsulated actions?

3. Proposal

In this section we briefly sketch two solutions that address the concerns raised in Section 2. In constructing the paper in this way, we hope to demonstrate two points:

- *AOP becomes more effective when the underlying code base conforms to a well-defined structure.* – While it is true that AOP can be applied to any code base, one must be careful to write aspects so they do not weave unexpectedly into areas of the code base where they shouldn't belong. This danger is a common problem with aspects because of the way one can use regular expressions to identify the join points into which advice is to be woven. The separation embodied within EBC makes it easier to

write aspects that are highly targeted and maximize utility while minimizing the chance that unexpected weavings occur.

Researchers have begun studying how to properly structure the way aspects are written to ensure comprehensibility as well as ease of change (such as XPI for AspectJ [7]). We believe that one can also improve the effectiveness of aspects by ensuring a suitable structure for the underlying code base.

- *FOP provides a useful algebra for constructing Control objects* – When software developers follow an object-oriented design methodology, it is possible to strongly correlate Use Cases with Control objects. Can one take the next logical step and propose developing a “Control design tool” (analogous to the GUI design tools already in use) to manage the construction of all Control classes? Having such a tool would improve programmer productivity and increase reuse.

3.1 Use AOP to govern EBC communication

One of the most frustrating aspects of working with EBC code is the number of places where one must hard-code a specific communication pattern in the actual classes. We have identified two such locations:

- Mapping Boundary events to Control object invocation.
- Implementing an Observer design pattern [4] that captures the way Boundary objects update themselves in response to changes in Entity objects.

Because the GUI design tool generates stylized code, it is possible to write targeted aspects whose sole purpose is to weave code into the (previously) stubbed methods as shown in Figure 3. In particular, the `actionPerformed` method should simply invoke an `ExitController` class. The desired code result might be:

```

public void actionPerformed(ActionEvent e) {
    new ExitController().attemptExit();
}

```

which can readily be enacted using an aspect. Moving such logic into an aspect offers immediate flexibility in determining how Boundary objects are to be controlled. As an additional benefit, there is no need to make any changes directly to the code generated by the GUI design tool (although most modern design tools now allow for “round-trip” engineering of source code to model, so this issue is not as problematic as it used to be). We expect that using aspects will make it easier to update communication channels rather than making modifications by hand. For example, if it became evident that the `ExitController` required information that would be passed as an argument to the `attemptExit` method, the aspect could be easily rewritten to pass the desired argument along. Without having a structured code base, we expect it would be quite challenging to envision writing an aspect to

accomplish this task because of the many ways by which one could write GUI Java code.

Our second application of aspects is to implement the Observer design pattern. One can readily find examples showing how to apply this design pattern using aspects [10]. The reason why it is useful to even consider performing this task using aspects is that it allows the essential EBC classes to be designed and implemented separate from the (rather arbitrary) way in which the Observer pattern is implemented within the Entity and Boundary objects. Using aspects will allow for greater experimentation when designing how the Boundary objects are to properly reflect changes in the underlying Entity objects. By separating the concerns related to the Observer pattern, one can make the underlying Entity classes more comprehensible.

3.2 Use FOP to assemble Control objects

A Control design tool must be able to identify sub-steps within a Use Case and produce an encapsulated fragment that represents that sub-step. Then it must be able to assemble these fragments in arbitrary, though ordered, ways to produce a final Control object. We envision using a Feature-Oriented approach, such as enabled by Batory’s Algebraic Hierarchical Equations for Application Design (AHEAD) tool suite [5]. Whereas FOP considers assembling features, we believe the approach can also work with Use Case sub-steps. In this brief paper we can only include a few details about AHEAD. A layer $l = (a_1, a_2, \dots, a_m)$ contains a set of m Jak artifacts that are composed together to produce a set of Java classes. Each artifact a_i is either a refinement of an existing class or a newly defined class. A linear equation $h \cdot j$ can be defined to represent the composition of two layers. Thus, for example, the equation $[h(a_1, a_3) \cdot j(a_2, a_3)]$ results in three artifacts and the order of the composition shows that design artifact a_3 in h refines the existing design artifact a_3 in j . Another way to remember this ordering is to view equation $h \cdot j$ as being $h(j)$ where h refines j .

In our vision, a Control object is composed of an ordered set of n sub-steps as defined by a Use Case, where each sub-step is modeled by an AHEAD layer. Thus the Control design tool would be made possible by invoking AHEAD with carefully designed layers. While we have experimented in using AHEAD to construct sample Control objects, there are some challenges to overcome. First, jak2java (the AHEAD composition tool) assumes that layers are composed together “in place” which makes it hard to reuse layers in more than one Control object. Second, the Jak artifacts within a layer must uniquely identify the Jak or Java artifact being refined. What is needed is a type parameterization mechanism to allow encapsulated logic to be applied to multiple Control objects. Third, the fragment code one might associated with a Use Case sub-step may require input that otherwise would be stored in local variables. Current Jak layers can communicate only through parameters or shared class variables, which makes it challenging to make this happen.

Pre-aspect Model	Post-aspect Model
+ final String untitled ~ String name ~ transient boolean modified ~ transient ArrayList<IModelUpdate> listeners - transient IModelUpdate[] template	+ final String untitled ~ String name
# void resetTransient() + void setName(String) + String getName() + boolean isModified() + boolean setModified(boolean) + void addListener(IModelUpdate) + void removeListener(IModelUpdate) # void publish() + abstract Model blank() + abstract void revert(Model) + void close() + final boolean equals (Object)	+ void setName(String) + String getName() + abstract Model blank() + final boolean equals (Object)

Figure 5. Reduced Model after using aspects

Earlier we discussed how we envision using AOP, but we believe it to be impractical to use aspects to weave together Control objects. The primary difficulty is how to ensure the proper ordering of the sub-steps which make up a Use Case. While AOP does provide a means to determine the ordering by which a number of aspects are woven together, experience shows that enforcing such ordering constraints invariably weakens the declarative power of aspects. The FOP style, in contrast, is ideal for assembling sub-steps in an ordered fashion.

3.3 Case Study

We are in the midst of completing a case study where we are putting these ideas into practice. As part of a course on Software Design, the first author of this paper developed an Application Framework (called *MetaFraming*) that was used to develop four separate applications within a graduate course on Software Design. This code base is fully implemented. As part of his Master’s Thesis, the second author of this paper is refactoring the code to integrate aspects (using AspectJ) as well as features (using Batory’s AHEAD tool suite). Focusing on a single case study across numerous techniques makes it possible to accurately compare the different modular techniques with each other.

3.4 AOP work to date

Recall that we envisioned using aspects to (a) map Boundary events to Control objects; and (b) apply the Observer design pattern to ensure synchrony between the Entity objects and their corresponding Boundary objects. We now provide preliminary evidence to support our approach for using aspects.

Over time in the *MetaFraming* framework, a Model class became a focal point for storing information and managing the distribution of that information to interested parties. In Figure 5 we show the structure of the Model class before and after using aspects to weave in the requisite “publish/subscribe” behavior demanded by the application. Note that 75% of the methods in Model were related to Observer pattern issues, and moving this logic into an aspect simplified the existing artifacts. The Observer pattern is still

```

public class MetaApplication {
    ...

    /** new attribute for menu item. */
    private JMenuItem MetaApplication.printPreviewMenuItem = null;

    ...

    /** This method initializes printMenuItem. */
    private JMenuItem getPrintPreviewMenuItem() {
        if (printPreviewMenuItem == null) {
            printPreviewMenuItem = new JMenuItem();
            printPreviewMenuItem.setActionCommand(Intl.printPreviewCommand);
            printPreviewMenuItem.setText(Intl.printPreviewCommand);
        }

        return printPreviewMenuItem;
    }

    /** This method initializes the File menu. */
    private JMenu getFileMenu() {
        ...
        // insert the Print Preview as a menu option
        fileMenu.add(getPrintMenuItem());
        fileMenu.add(getPrintPreviewMenuItem());
        fileMenu.add(getPageSetupMenuItem());
        ...
        // Have menu item event be sent to the File Menu Controller
        getPrintPreviewMenuItem().addActionListener(fmc);
        ...
    }
}

public class FileMenuController {
    ...

    /** controller method of FileMenuController */
    public void actionPerformed(ActionEvent e) {
        String action = e.getActionCommand();
        if (action.equals (Intl.newCommand)) {
            ...
        } else if (action.equals (Intl.printPreviewCommand)) {
            // new logic for PrintPreview command
        }
    }
}

```

Figure 6. Original code

present, but instead of being embedded within the core artifacts, it is cleanly encapsulated in some aspects. For lack of space we omit the four aspects we implemented, which involve two pointcuts and six advice blocks.

When developing the *MetaFraming* framework, we decided to add a feature to enable a “Print Preview” option common to most GUI-based applications. The original Java code which includes this feature is shown in Figure 6. This code was generated using a GUI-builder tool (Visual Editor for Java) and you can observe its highly stylized nature. Every change to the visible GUI required making these changes “in place” to the code base, leading to an incrementally complex code base.

The alternative is shown in Figure 7. By taking advantage of the EBC structure and the stylized code generated by the GUI design tool, the aspect has a fairly simple set of pointcuts and advice. Adding the PrintPreview command is a more complex example of writing aspects whose purpose is to “tie” a particular feature into a GUI.

```

public privileged aspect PrintPreviewFeature {

    /** Constant for referring to Print Preview command. */
    public static final String Intl.printPreviewCommand = "Print Preview";

    /** Fields accessed within print Preview. */
    public static final String Intl.twoPage = "Two Page";
    public static final String Intl.onePage = "One Page";
    public static final String Intl.prevPage = "Prev Page";
    public static final String Intl.nextPage = "Next Page";

    /** Advise FileMenuController to check for PrintPreview. */
    after (FileMenuController fmc, ActionEvent e) returning:
        this(fmc) && args(e) &&
        execution(void FileMenuController.actionPerformed(ActionEvent)) {
        String action = e.getActionCommand();

        if (action.equals (Intl.printPreviewCommand)) {
            // new logic for PrintPreview command
        }
    }

    /** Add new field and access methods to MetaApplication. */
    private JMenuItem MetaApplication.printPreviewMenuItem = null;

    private JMenuItem MetaApplication.getPrintPreviewMenuItem() {
        if (printPreviewMenuItem == null) {
            printPreviewMenuItem = new JMenuItem();
            printPreviewMenuItem.setActionCommand(Intl.printPreviewCommand);
            printPreviewMenuItem.setText(Intl.printPreviewCommand);
        }
        return printPreviewMenuItem;
    }

    /** Insert within GUI builder access. */
    pointcut fileMenuCreation(MetaApplication appl): this(appl) &&
        execution (JMenu MetaApplication.getFileMenu());

    before (MetaApplication appl): cflow (fileMenuCreation(appl)) &&
        call (JMenuItem MetaApplication.getExitMenuItem()) {

        appl.fileMenu.add(appl.getPrintPreviewMenuItem());
    }

    /** Add controller as listener for PrintPreview Menu item */
    after (MetaApplication appl): cflow (fileMenuCreation(appl)) &&
        call (JMenuItem MetaApplication.getExitMenuItem()) {
        appl.getPrintPreviewMenuItem().addActionListener(appl.fmc);
    }
}

```

Figure 7. Using Aspects to integrate Print Preview feature

3.5 FOP work to date

In our earlier discussion on Control objects we presented a code fragment in Figure 4 to confirm the user’s action whenever a document was modified. Figure 8 contains a hand-assembled Jak layers [5] that we envision could be generated by a Control design tool. The stylized Jak code assembles the `internalFrameClosing` event handler for the `InternalFrameController` class by assembling fragments (appropriately bracketed as user-specific blocks within the boiler-plate template). The object reference `d` refers to the document being closed; this can be referenced because the original Control class (which is being refined here) exposes that object as a protected class attribute. The equation which constructs the working Control class is `confirm • close • base`. Lack of space prevents us from showing the other Jak layers.

```

refines class InternalFrameController {
    // all extensions via fop must call this
    protected void init_internalFrameClosing (InternalFrameEvent ife) {
        if (!isInitialized()) {
            Super(InternalFrameEvent).init_internalFrameClosing(ife);
        }
    }

    public void internalFrameClosing(InternalFrameEvent ife) {
        init_internalFrameClosing(ife);

        // begin user-specific block
        if (d.isModified()) {
            int x = JOptionPane.showConfirmDialog(ife.getInternalFrame(),
                "Do you really want to close this window?",
                "Unsaved Changes", JOptionPane.YES_NO_OPTION);
            if (x == JOptionPane.NO_OPTION ||
                x == JOptionPane.CANCEL_OPTION) {
                return;
            }
        }
        // end user-specific block

        // Now confirmed, pass along.
        Super(InternalFrameEvent).internalFrameClosing(ife);
    }
}

```

Figure 8. confirm Jak layer to represent confirm action

4. Related Work

The most closely related concept to the envisioned assembly of Control objects work is the Presentation, Abstraction, Controller (PAC) design pattern [8] that forms a hierarchy of agents, each of whom is responsible for a particular aspect of system functionality. The primary limitation of using PAC is its complexity. First, one must select the appropriate level of granularity for each PAC; second, the control components increasingly become mediators between the Abstraction/Presentation, as well as with other PAC agents. Third, PAC agents are distinct objects and do not share the ability of AOP or FOP to compose together and share state. Finally, while PAC is extensible, allowing one to readily insert new PAC agents into an existing hierarchy, the lifecycle management of the agents quickly becomes a major concern. What we propose is to compose together encapsulated logic fragments so there is no need to maintain or instantiate objects for each fragment, as one would need to do for each PAC object.

One common shortcoming with using AOP to manage the composed Control object composition is that it does not scale when several aspects are to be woven together over the same artifact. The problem may be the lack of fine-grained control over the ordering of the weaving. AOP simply fails to lay the foundation for designed variability because of its focus on the implementation artifacts. Some methodologies, realizing this limitation, have sought to model the generic creation and customization of modules. OPM is a rich modeling methodology [9], whose weakness appears to be a steep learning curve and lack of visibility in the greater community. Inheritance and delegation both offer mechanisms to extend existing behavior by “bracketing” a method invocation; a delegate can intercept a method re-

quest and perform additional work before and after. With inheritance, a subclass can override a method `C.m()` with `{preWork();C.m();postWork();}`. While these techniques work well for “localized” behavioral modifications, they simply do not scale when unanticipated (seemingly arbitrary) behaviors need to be composed together.

5. Conclusion

The research described in this paper is most definitely a work in progress. We were originally motivated to pursue this research agenda by our past experience with complex Object-Oriented systems where much of the complexity was “incidentally” present only because of the limitations of using native OO language constructs to capture the desired communication between Entity, Boundary and Control objects. At the very least, we hope that our results will show circumstances where different modularity mechanisms perform their best, and where multiple co-existing modularity mechanisms can enhance each other’s capabilities.

References

- [1] H. Ossher, P. Tarr, “Hyper/J(tm): Multi-Dimensional Separation of Concerns for Java(tm)”, *23rd International Conference on Software Engineering (ICSE)*, 2001, pp. 08–21.
- [2] I. Jacobson, G. Booch, and J. Rumbaugh *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.
- [3] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2000.
- [4] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*, Addison Wesley, 1995.
- [5] D. Batory, J. Sarvela, and A. Rauschmayer, “Scaling Stepwise Refinement”, *International Conference on Software Engineering*, Portland, Oregon, May, 2003.
- [6] B. Kotec, *MVC design pattern brings about better organization and code reuse*, Builder.com: beyond the code, October 2002, <http://builder.com.com/5100-6386-1049862.html>
- [7] K. Sullivan, W. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle and N. Tewari, “Modular Aspect-Oriented Design with XPIs”, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009.
- [8] G. Calvary, J. Coutaz, L. Nigay, “From Single-User Architectural Design to PAC*: a Generic Software Architecture Model for CSCW”. *Proceedings of the ACM CHI 97 Human Factors in Computing Systems Conference*, 1997, pp. 242–249.
- [9] D. Dori, *Object Process Methodology*, Springer-Verlag, August 2002.
- [10] R. Miles, *AspectJ Cookbook*, O’Reilly Media, Inc., 2004.