How to reduce costs of business logic maintenance

Tomas Cerny
Department of Computer Science and Engineering
Czech Technical University,
Charles Square 13, Prague, Czech Republic
Email: tomas.cerny@fel.cvut.cz

Michael J. Donahoo
Department of Computer Science,
Baylor University,
Waco, TX, USA
Email: jeff_donahoo@baylor.edu

***Abstract*—Three tier enterprise applications introduce multiple challenges for software engineers. Although we can divide the application into three tiers, we still need to design properly each tier internally to achieve multiple design qualities. The middle business tier captures logic in which we associate objects, validate business rules, etc. Often multiple cross-cutting concerns are mixed in the services which results in bloated, highly coupled design with very low cohesion. In this paper we present a case study that we develop based on our four year experience with enterprise application that struggled from multiple weak design decisions. We emphasize multiple aspects that should be decoupled from the rest of the services which increase service cohesion and results in better readability, maintenance, testability, reuse and error-avoidance. Our "best practices" suggestions for business tier are generally applicable and allow the designer to separate service concerns into multiple units allowing to achieve the mentioned quality attributes.**

***Keywords*—Business layer, best practices, refactoring**

I. Introduction

Application design and development trends settled for the last decades on the use of *object oriented analysis* (OOA) and *design* (OOD) [1] [2] [3]. OOD is a special type of *call-and-return* architectural styles [4] [5] and extends approaches such as *main and sub-routine call* and *abstract data type*. The most basic element of OOD is an *object* that encapsulates data and provides operations on the top of them via methods. Multiple objects interact via method calls, which implies that objects must know about each other to collaborate. The aim of object use is to provide information hiding and quality attributes [6] [7] such as modifiability and reuse. Object can in addition inherit from another one in order to extend and specialize parent's capabilities. Objects provide many possibilities for our design, but there is no prevention of making poor design. Few software quality metrics exist to provide feedback on the design maturity and to reduce maintenance and modification costs [8]. *Low coupling*, a measure of how strongly is one object connected to another one and *high cohesion* that says, how strongly related are the responsibilities of an object [1] are good examples of such metrics. To have a good design that contains readable and reusable components it should provide both low coupling and high cohesion. Unfortunately these metrics does give us a solution to a problem that we may encounter during our design. In order to use OOD effectively we often apply design patterns [2] [3], that represent the best available solution also referred as "best practices". Design patterns has grown to multiple categories and spread from analyzes over design to coding.

Besides multiple patterns simplifying designers decisions another innovative contribution and complement to the OOP appeared known *aspect oriented programming* (AOP) [9]. AOP that introduces the idea of cross-cutting concerns in objects. These concerns then spread through multiple architectural layers and cannot be isolated with the sole use of OOP. AOP defines the location where multiple concerns cross as a *pointcut*. This pointcut as reached by the program can use an additional code (separately defined) known as an *advice* and together form an *aspect*. This way our original code does not experience messed fragments with multiple concerns, but rather focus on the main concern and the extension ones are hooked to it. Every extension to the main concern is defined and maintained separately which supports modifiability, reuse, maintenance, low coupling and high cohesion.

Every larger application of these days employs design patterns, these may provide the architectural structure or just a simple creational factory [2]. Large enterprise applications often employ *3-tier* architecture in which multiple design patterns are applied [3]. Such 3-tier consists of tiers for data persistence, handling business logic and presentation.

Each tier may consist of specific design patterns and some concerns exist that may cross all the tiers, such as logging or security. We often deal with object-relational mapping in the *persistence tier* and provide access to objects. Higher *business tier* then uses the access interfaces and applies business logic with added constraints placed on object collaboration. The *presentation tier* deals with providing the view on the data and allows clients to interact with what they see. The presentation and persistence tiers are pretty straight forward. On the other hand the business tier it the most variable and often also the most complicated one [3].

Business tier contains business logic, captures how business objects interact with one another and makes logical decisions based on business rules. It deals with functional validation and exception handling, it may contain management of transactions and sessions, checks user access rights and makes integration of services transparent to the presentation tier above. To deal with these, multiple application development frameworks, such as [10] [11], suggest to logically separate further to *controllers* [1] (also called handlers, actions, managers, etc.) that take care of transaction and session boundaries, exceptions handling and

978-1-4244-8728-8/11/$26.00 ©2011 IEEE

to *services* that implement business rules, business validation and call persistence. In addition, controllers need to know the appropriate services to use. To support low coupling we centralize this wiring by the use *Dependency injection* pattern (DI) [12] or *Service locator* [13] . User rights are unfortunately crossed by both controllers and services, but also by persistence so presentation tiers and we must apply AOP. To simplify the design of stateless controllers and make transaction management transparent to the developer we may apply *Open session in view* pattern [14]. On the other hand in stateful cases we may let the application framework to deal with both session and transaction by annotating the $start$ and end method in between which the state and transaction is kept active [10].

The difficulty comes in when we start to deal with exception handling. We will most likely define our own local exceptions such as *BusinessRuleException*, *ServiceException*, *EntityExistException*, etc., but we also need to count on low-level exceptions that come from the libraries we use. Unfortunately, if we want to provide user with the most accurate exception information we must provide complicated exception handling for each controller method. Such design suffers from complex maintenance as one change must be reflected on multiple places. We may note the existence of coupling between an exception and the controller method as the method is sensitive towards changes in an exception signature.

Another issue might come in for business rules. It is nice to have business rules distributed in services, but what if at one point we want to see all rules that apply for one domain object, this way we must search all occurrences of the domain object and check the business rule settings, again the maintenance of this is difficult. From the point of single business rule there exists coupling with services and also low cohesion as the rule is distributed on multiple places.

Further more, what if a new business rule come to apply for domain objects which denotes that all objects older than one year should be read-only, so no changes are allowed. This business rule will be again distributed among multiple services, which harden the maintenance.

In this paper we argue that multiple applications that follow framework suggestions suffer from a design that distributes business rules in the service which in result increases the costs connected with application maintenance. We suggest a solution that provides low coupling, high cohesion, concern centralization and support maintenance with minimal impact on performance.

This paper is organized as follows in Section II we discuss a case study which elaborates and apply our suggestions. In Section III we discuss related work. The Section IV concludes.

II. Case Study

To present our suggestion we provide a case study where we deal with competitions which may take place in multiple locations. Every team has multiple person members and a coach, the team can attend the competition that allows to build a hierarchy and to promote successful teams to super contest. Our domain model is shown on Fig. 1.

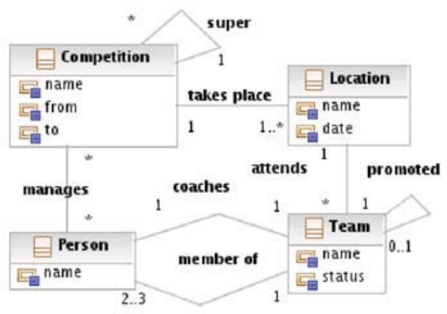

Fig. 1. Study domain model

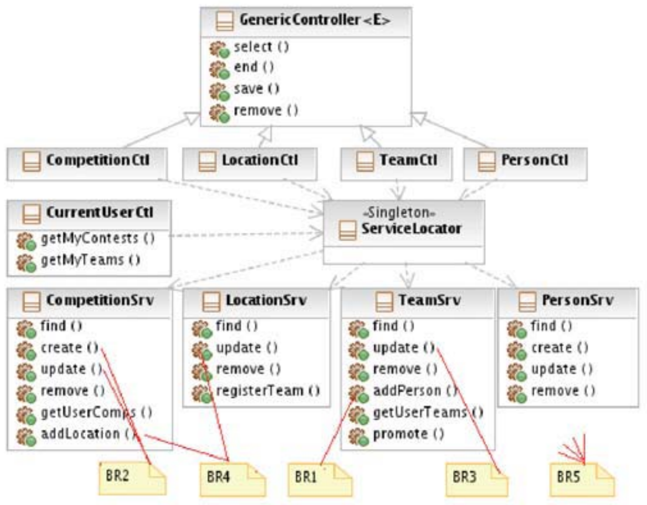

Fig. 2. Study business tier

The business tier contains a controller for each domain object as well as every domain object has its services. Our business rules (BR) might be that:

(1) Team can attend only with 2 or 3 team members
(2) Competition dates must have proper precedense
(3) Team cannot be modified once it is accepted
(4) Location date must be within the competition dates
(5) No changes are allowed 7 days after the competition ends (only promotion).

The business tier as shown on Fig. 2 consists of controllers and services and use *service Locator* pattern for wiring. In every service we provide selection, modification and remove method, plus ones related to context. Services then provide lower-level methods that are connected to the persistence tier. The business rules are attached as highlighted on Fig 2.

A. Exception handling

Every service method can throw an exception that might be low-level or local. In order to provide simple exception handling maintenance we would like to centralize it. We keep in mind that every exception is different and for example when $DuplicateEntityExceptions$ is caught we also want to show the duplicate object. All this can be achieved by one additional method in $GenericController$.

```
// PersonController
public class PersonController extends GenericController<Person> {
..  try {  ..
  } catch (Exception e) {
    handle(e)
  } ..
// GenericController
public class GenericController<E> { ..
  protected void handle(Exception e) {
    if(e instanceof BusinessRuleException) { ..
    } else if (e instanceof NullPointerException) {  ..
    } else if ..
```

Listing 1. Exception handling first thoughts

Well, but this way we need to identify the exception and in order to do this we need to dynamically check the type of the class in run-time. Then we have a large $if - else$ block to handle the actual exceptions. This way we provided centralization, but this might not be as effective and the maintenance is still discutable as large $if - else$ block is generally not the best practise.

Instead we could try to use *visitor* pattern [2]. We can apply the trick where a virtual method enters the handler code with its actual exception type. We can apply this for each local exception which removes the need to retype in run-time. Unfortunately we cannot apply this for low-level exceptions. If our language also provides reflective API [15] we apply dynamic exception retyping and then via the reflection method call the appropriate handler.

```
// LocalException
public class LocalException { ..
  void visit(ExceptionHandler handler) {
     handler.handle(this);
  } ..
// PersonController
public class PersonController extends GenericController<Person> {
..  try { ..
    } catch (LocalException e) {
      super.dispatch(e)
    } catch (Exception e) {
      super.dispatch(e)
    }
// GenericController
public class GenericController<E> {
..  private ExceptionHandler exceptionHandler;
..  protected void dispatch(LocalException e) {
      e.visit(exHandler);
    }
    protected void dispatch(Exception e) {
      try {
        Method m = exceptionHandler.getClass()
                    .getMethod("handle", e.getClass());
        msg = (String) m.invoke(exceptionHandler, e);
      } catch {
      // no handler found
        exceptionHandler.handle(Exception e)
     }
    }
// ExceptionHandler
public class ExceptionHandler {
..  public void handle(BusinessRuleViolation e) { .. }
    public void handle(LocalException e) { .. // default local }
    public void handle(NullPointerException e) { .. }
    public void handle(Exception e) { .. // default low-level }
```

Listing 2. Exception handling second thoughts

This way we centralized all the exception handling to one class $ExceptionHandler$, if we add a new exception we overload the $handle(..)$ method with its argument type and specify how to deal with the exception.

B. Business rules first thoughts

Business rules can spread all over the services as we see on Fig. 2. The design we target will allow to centralize the business rules and will allow us to quickly see all the setting which we can modify. To deal with this we could use previous approach, but instead a single dispatch with method overload is all we need. We introduce new class $BusinesRuleValidator$ and this class has an overloaded method $validate(Entity\ e)$.

```
// BusinesRuleValidator
void validate(Team t) {
 if (t.getPersonSet().size() > 3 or t.getPersonSet().size() < 2){
    throw new BusinessRuleException(..); // (1)
 }
 if (Status.ACCEPTED.equals(t.getStatus())) {
    throw new BusinessRuleException(..); // (3)
 }
 if (is7DaysAfter(t.getLocation().getCompetition())) {
    throw new BusinessRuleException(..); // (5)
 }
}
void validate(Competition c) {
 if (c.getStart().after(c.getEnd())) {
    throw new BusinessRuleException(..); // (2)
 }
 if (is7DaysAfter(c)) {
    throw new BusinessRuleException(..); // (5)
 }
}
void validate(Location l) {
 if (c.getStart().after(l.getCompetition().getDate())
       || c.getEnd().before(l.getCompetition().getDate())) {
    throw new BusinessRuleException(..); // (4)
 }
 if (is7DaysAfter(l.getCompetition())) {
    throw new BusinessRuleException(..); // (5)
 }
}
..
// TeamService - the service can call the validation.
void addPerson(Team t, Person p) {
  businesRuleValidator.validate(t);
  t.getPersonSet().add(p);
  persist(t);
}
```

Listing 3. Business rule decoupling

This way we have all the business rules defined on one place, which provides very high cohesion and allows us to modify the rules. We could also consider to have a distinct validators for create, update or delete. Perhaps one thing little messing the code above is the business rule (5). The thing is that it crosses all the concerns and business validations.

C. Business rules second thoughts

It might be reasonable to look at what we could do when we deal with crossed concerns. We already mentioned that security must deal with these. Multiple frameworks [16] [17] allow us to separate the definition of access rights in standalone configuration. This configuration for example defines an access rule allowing to add person to a team. Such rule is evaluated in run-time based on user's context. We could define access rule that looks like this:

```
<securityRule controller="teamController"
              method="addPerson" context="team">
  <role>ADMIN</role>
  <role>OWNER</role>
</securityRule>
```

Listing 4. Security rules

It is parsed by security framework that also intercepts all the presentation, business and persistence tier method calls and if a class $TeamController$ method $addPerson()$ is called then we evaluate on the active context $team$ whether the current user has a role *ADMIN* or *OWNER*, if not then we throw an $AuthorizationException$ and redirect user away.

Well perhaps we could deal in similar manner with the business rule (5). We said that once the competition ended and it is 7 days after the end we should restrict users to read-only access to all related domain objects.

Well our security allows the owner and admin to access all the methods in the controllers, but we want them only to be able to read and promote. Fortunately we can extend the security framework with *veto rules*. These vetoes can add addition constraints that are evaluated before the access decision is made. We could add here rules that veto access to all non-$select$ and non-$promote$ methods when the active competition ended before 7 days.

```
//Veto Rules - Rule {controller, method regex, context, eval}
Set<VetoRule> initRules() {
 Rule readOnlyPerson = new Rule(
  "personController","(?!select).*","person",
  "#{fc:is7DaysAfter(person.team.location.competition)}"));
 Rule readOnlyTeam = new Rule(
  "teamController","[(?!promote)|(?!select)].*","team",
  "#{fc:is7DaysAfter(team.location.competition)}"));
  ..
 vetoRules.add(readOnlyPerson); vetoRules.add(readOnlyTeam);
}
```

Listing 5. Veto rules

To transform the business rule (5) into a veto rule will cost us less maintenance. This way we do not need to apply the rule (5) in the $BusinesRuleValidator$ as it will make our design more clear and more easy to read and maintain. Since we already use the security framework the veto rule costs the same as other security rule.

D. Summary

The approach for exception handling allows to centralize all exception translation to one place. We could also propagate the low level exceptions all the way up, which would cause page redirect to a default page. This would be good from the perspective of maintenance, but we would not be able to influence much what to do next and that way it might be less user friendly. The business rules are centralized with respect to business logic and the domain object relation to a particular rule. It, although a beneficiary solution, also degrades when one business rule is spread on multiple places. For such a cross-business rules we may find a better solution that involves AOP and perhaps also the security which we most likely have in the project. We may extend the security framework with vetoes that apply in places where the business rule is violated. This allows us to remove the cross-business rule from all places and define it just once.

III. Related Work

There can be found a significant amount of work on software maintenance [2] [3] [6] [8] [9] [13] [18] [19] [20] [21] [22]. In addition the maintenance is tightly connected to the reusability of elements, readability, portability, extendability and modifiability [4] [6] [7]. The known practices of last decades useful for OOD are known as design patterns [2] [3] [13]. Design patterns are the best practices for problems they relate to. Multiple industrial experts have shown case studies with their use [18].

In [19] and [20] authors argue that 65-75% of the total life-cycle time of SW development is consumed by maintainability. The [23] then says that "Research indicates that some 40-60% of the maintenance effort is devoted to understanding the software to be modified". For long time the measures for maintainability, coupling and cohesion are used. Stevens et. al. [8] statement "A structure is stable if cohesion is strong and coupling is low" provides a solid ground for most of literature commonly used these days [1] [2] [3]. In consequence we see that when the software design has such qualities it simplify the maintenance and furthermore decreases costs related with the development.

Significant improvement to maintenance in the design is brought by the Hollywood principle and Dependency Injection pattern [19]. This patterns allows to decouple the knowledge of interconnection of components and passes this knowledge on external assembler (container). This way components are dependant only based on their interfaces and the replacement is very simple. On the other hand in [19] authors apply multiple metrics to see the impact of DI on existing projects. The result is that they cannot confirm that DI reduces dependencies and make software more maintainable. In our paper we expect that developer uses DI as it is a common practice for enterprise application development.

Another direction we mentioned is aspect oriented programming (AOP) [9]. There is an application of AOP in static way or in run-time. With the first type, a duplication which is the main contributor to poor maintenance can be seen very different now. The duplication that is auto-generated from cross-cutting *pointcut* with addition of an *advice* is not an issue [21] [22] and can provide benefits. Generation may result in replication of significant amount of code and not cause any negative effect as the definition for the replicated code is being centralized in the *pointcut* and *advice*. This will support maintainability and will not affect performance. On the other hand a run-time AOP will provide dynamic evaluation with the same keened effect on centralization, but this may slightly influence the performance. We show an example where AOP is used for security in run-time. We extend it for vetoes which can reduce the business rules crossing multiple elements.

An interesting way to deal with business rules brings a business logic integration platform Drools [16]. It consists of modules for rules, workflow and event processing. The Drools

Expert responsible for rules allows developer to set business rules in a domain specific language and to isolate the business logic from the software applications. In addition the logic can be modeled using activity diagram and debbuged. The business rules design then looks as follows.

```
rule "Is valid competition date"
when
   $c : Competition ($s: start!= null, $e: end!=null)
   eval($s.after($e))
then
   $c.addError("start","Start date cannot be after end")
   $c.addError("start","End date cannot be after start")
end

rule "Can modify competition"
when
   $c : Competition()
   eval(is7DaysAfter($c))
then
   $c.addError("Cannot be modified")
end
```

Listing 6. Drools

Drools are compliant with Java Rule Engine API that defines a Java run-time API for rule engines. Such a rule engine may be viewed as a sophisticated if/then statement interpreter [24]. The bad side with experience with Drools is that it brings significant overhead. It may also need to load all domain objects before the rule processing, although explicit improvements to this can be applied. Our approach uses the underlying language and centralizes the rules regards the domain object, the exception translation is decoupled to another module what makes it easy to replace or modify.

IV. Conclusion

Software maintenance is the major factor in software development cycle. Designers attention should point on how to simplify it. To design readable and reusable software components can be one approach to deal with this. Enterprise applications should consist of multiple such components, but unfortunately there often exist a third dimensions than makes the components coupled as some constraints that cross-cut them. This can be solved by applying AOP which decouples cross-cutting constraints and involved components.

In this paper we focused on improvement of business layer. This layer is the least straight forward in 3-tier architecture. With compatibility towards existing frameworks we suggest to delegate and centralize business rules in a separated component that accumulates business rules connected with a domain objects. This in addition to low coupling provides high cohesion which is a measure of components with high maintenance potential. Further more, we decoupled and centralized exception handling to a component that provides exception translation to user readable error messages or perhaps internationalization. In the exception handling we employ visitor pattern with double dispatch for our local exceptions and single dispatch with dynamic re-cast for low level exceptions, we argue that such a solution will not affect performance as low level exceptions for frameworks such as Seam [10] or Spring [11] means a fatal error, so an exception like that should happen only exceptionally. In addition to both suggestions above we show how to deal with business rules that apply on multiple places. A simple extension mechanism to AOP security is provided where granted access may be vetoed by additional business rule check. The disadvantage of such design might be separation of business rules to two places, but on the other hand logically these are separable as the first group cross-cut multiple places and the others relate to a domain object.

The suggestion we present come from our four year experience we have with large enterprise application, where business rules make significant part of the application and they change with the application evolution. We designed multiple approaches in the past where some meant to cross-cut multiple places, couple the code and make it hard to read and maintain, such design led to ineffective costs related with application evolution and changes. Most complications come with searching all rules related to a domain object. Our design presented in this paper reflects needs described above, which results with reduced costs connected with maintenance.

As future work we could provide a study with application of metric measurements as they did in [19]. A tool for Chidamber and Kemerer Java Metrics [25] can provide measurement of six metrics, such as weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, response for a class, lack of cohesion in methods and also afferent couplings and number of public methods.

References

[1] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[4] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[5] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[6] L. J. Bass, M. Klein, and F. Bachmann, “Quality attribute design primitives and the attribute driven design method,” in *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. London, UK: Springer-Verlag, 2002, pp. 169–186. [Online]. Available: http://portal.acm.org/citation.cfm?id=648114.748917

[7] P. Praus, S. Jaromerska, and T. Cerny, “Sscac: Towards a framework for small-scale software architectures comparison,” in *SOFSEM 2011: Theory and Practice of Computer Science*. London, UK: Springer-Verlag, 2011.

[8] W. P. Stevens, G. J. Myers, and L. L. Constantine, “Structured design,” *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[9] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar, “Aspect-oriented programming,” in *In ECOOP’97-Object-Oriented Programming, 11th European Conference*, vol. 1241. Springer, June 1997, pp. 220–242.

[10] “Seam enterprise java development framework,” 2010. [Online]. Available: http://seamframework.org

[11] “Spring framework,” 2010. [Online]. Available: http://www.springsource.org/

[12] M. Fowler, “Inversion of control containers and the dependency injection pattern,” 1 2004. [Online]. Available: http://martinfowler.com/articles/injection.html

[13] D. Alur, D. Malks, J. Crupi, G. Booch, and M. Fowler, *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*, 2nd ed. Mountain View, CA, USA: Sun Microsystems, Inc., 2003.

[14] "Open session in view pattern," 2009. [Online]. Available: http://community.jboss.org/wiki/OpenSessioninView

[15] J. Sobel and D. P. Friedman, "An introduction to reflection-oriented programming," 1996.

[16] "Drools: Business logic integration platform," 2010. [Online]. Available: http://jboss.org/drools

[17] "Spring security: Access-control framework," 2010. [Online]. Available: http://static.springsource.org/spring-security

[18] K. Beck, R. Crocker, G. Meszaros, J. Coplien, L. Dominick, F. Paulisch, and J. Vlissides, "Industrial experience with design patterns," in *Software Engineering, 1996., Proceedings of the 18th International Conference on*, Mar. 1996, pp. 103 –114.

[19] E. Razina and D. Janzen, "Effects of dependency injection on maintainability," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*. Anaheim, CA, USA: ACTA Press, 2007, pp. 7–12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1647636.1647639

[20] S. Muthanna, K. Ponnambalam, K. Kontogiannis, and B. Stacey, "A maintainability model for industrial software systems using design level metrics," in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, ser. WCRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 248–. [Online]. Available: http://portal.acm.org/citation.cfm?id=832307.837117

[21] T. Cerny and E. Song, "A profile approach to using uml models for rich form generation," in *Information Science and Applications (ICISA), 2010 International Conference on*, 2010, pp. 1 –8.

[22] T. Cerny and M. J. Donahoo., "Formbuilder: A novel approach to deal with view development and maintenance," in *In SofSem 2011 Proceedings of Student Research Forum*. OKAT, January 2011, pp. 16–34.

[23] P. Bourque and R. Dupuis, "Guide to the software engineering body of knowledge 2004 version," *Guide to the Software Engineering Body of Knowledge, 2004. SWEBOK*, 2004.

[24] "Jsr 94: Javatm rule engine api," 2010. [Online]. Available: http://www.jcp.org/en/jsr/detail?id=94

[25] "Chidamber and kemerer java metrics," 2010. [Online]. Available: http://www.spinellis.gr/sw/ckjm/