

Towards Goal-Driven Design

Vaidas Gasiunas
Software Technology Group
Darmstadt University of Technology, Germany
gasiunas@informatik.tu-darmstadt.de

Thiago Tonelli Bartolomei
Dep. Computer Science and Electrical Eng.
Kiel University of Applied Sciences, Germany
thiago.bartolomei@gmail.com

ABSTRACT

Aspect-oriented languages provide a lot of flexibility for software modularization. On one hand it is an advantage, because when choosing from more alternatives we can find designs that better meet our needs. On another hand there is also a greater risk to misuse this flexibility and to produce too complicated designs. In this paper we show that there can be different solutions to the same problem depending on the assumed design goals. Thus, we shouldn't speak about the best design, but rather about the best design with respect to the stated design goals; and the success of design largely depends on proper selection of design goals. We believe that the design process could be improved by making goal analysis more explicit. Non-functional requirements should be specialized up to specific design goals that constrain implementation decisions. These goals should be documented and used for understanding and evaluating designs. Furthermore, design patterns should assist developers in the goal-driven design process by providing a catalogue of design goals that should be considered in the context of the design pattern and a guide on how these design goals could be translated into implementation decisions.

1. INTRODUCTION

Finding the best software design for a given problem often seems to be an unsolvable problem, because there are a lot of conflicting criteria to be considered. If we try to maximize reuse, extensibility and stability of our designs, we produce more complex and less efficient designs. Only when these criteria are translated to specific design goals, they can be properly evaluated and prioritized. What developers are usually doing, consciously or not, are identifying such design goals and looking for one or another implementation strategy that fulfills them. We believe that such goal-driven design process should be made more explicit. Non-functional requirements should be specialized up to specific design goals that constrain implementation decisions. These goals should be documented so that they can be used to understand and evaluate designs, as well as to detect when

refactorings are necessary.

Aspect oriented languages make the process of software design even more complicated, because each new language feature gives an additional freedom in design what inevitably increases the set of alternative designs to be considered. Thus, design patterns, as solutions in specific contexts, are even more important for aspect-oriented languages than for the object-oriented ones. Since aspect-oriented languages provide a lot of design flexibility, it is difficult to find the best solution for a large set of problems. Therefore, we believe that design patterns for aspect-oriented languages should be rather guides that help to evaluate the criteria that are relevant to the problem. Further, they should present the design alternatives in form of independent design variation points and relate the decisions for each variation point with the results of evaluation.

In the following section we present a case study of the goal-driven design, where we show three different solutions to the same problem that, however, differ by the stated design goals. In Sec. 3 we explain why none of these solutions can be considered as the best and the implications of this observation to evaluation of designs and development practices. In Sec. 4 we explain how design patterns could support the goal-driven design process. Finally, we give a short overview of related work and a summary.

2. CASE STUDY

In order to exemplify the goal-driven design process we take a case of observing using aspects. Consider an application to visualize and edit graphs. The application uses a simple graph data model, consisting of nodes and edges (List. 1). Various editing actions are encapsulated in separate classes (List. 2). The application allows to open and edit multiple graphs at the same time, but we assume that the editing operations are not concurrent.

Among other functionality, the editor provides the possibility to create bookmarks to nodes and to display the bookmarked nodes in a sorted list. The bookmarks must be updated after various changes in the graph. First, if a bookmarked element is deleted from the model it must also be removed from the list. Second, if elements from the list change their state, the list of bookmarks must be reordered. So we have a typical situation of Observer design pattern, where the bookmarks component plays the role of observer and the graph model is the subject.

```

1 class Graph { ...
2   List getNodes() { ... }
3   List getEdges() { ... }
4   void addNode(Node node) { ... }
5   void removeNode(Node node) { ... }
6 }
7 class Node { ...
8   String getName() { ... }
9   void setName(String name) { ... }
10 }
11 class Edge { ...
12   Node from, to;
13 }

```

Listing 1: Graph model

```

1 class Editor { ...
2   Graph getGraph() { ... }
3   Bookmark getBookmarks() { ... }
4   Iterator getSelNodes() { ... }
5   String getInputString() { ... }
6 }
7 class RenameSelectionAction extends EditorAction { ...
8   void execute(ActionEvent e) {
9     String newName = getEditor().getInputString();
10    for (Iterator it = getEditor().getSelNodes();
11         it.hasNext();) {
12      ((Node)it.next()).setName(newName);
13    }
14  }
15 }
16 class DeleteSelectionAction extends EditorAction { ...
17   void execute() { ... }
18 }
19 ...

```

Listing 2: Editor actions

2.1 First Design

After having stated all functional requirements we can begin with design. We start by analyzing the context of the problem and formulating our design goals. Since bookmarks are a quite specific feature of the graph editor, we surely don't want that such core features as graph data model would depend on it. Furthermore, we want to keep the complexity minimal and do not want to pollute graph data model code with complicated observer management and event notification, neither do we want that the bookmarks functionality depends on the events that it does not need. We consider that sorting bookmark elements is an expensive operation, therefore, we should avoid redundant updates that require reordering. The goal-driven design requires that we translate our considerations into explicitly stated design goals:

Goal 1. *The graph model should be independent of its concrete observers.*

Goal 2. *The graph model should be independent from the functionality to detect changes in it.*

Goal 3. *Bookmarks should not depend on any observation functionality that it does not need.*

Goal 4. *Redundant expensive updates of bookmarks should be avoided.*

The design goals should be stated in the most constructive

```

1 cclass Bookmark implements PerThisDeployable {
2   List bookmarks = new ArrayList();
3   Graph graph;
4   boolean reorder = false;
5   public Bookmark(Graph g) {
6     this.graph = g;
7     DeploySupport.deployOnObject(this, g);
8     deploy new TransactionObserver();
9   }
10  public void addBookmark(Node node) {
11    bookmarks.add(node);
12    DeploySupport.deployOnObject(this, node);
13  }
14  ...
15  /* Remove corresponding bookmark after a node is removed */
16  after (Node node) :
17    execution(void Graph.removeNode(*) && args(node) {
18      removeBookmark(node);
19    }
20  /* Reordering is needed when node name is changed */
21  after (Node node) :
22    execution(* setName(..) && this(node) {
23      reorder = true;
24    }
25  public cclass TransactionObserver {
26    /* Reorder after a logical transaction */
27    after() : execution(* EditorAction+.executes(..) {
28      if (reorder) { sort(); }
29    }
30  }
31 }

```

Listing 3: The bookmark component

form, so that they clearly constrain design decisions. The design goals that are implied by other design goals should not be stated separately. The implied goals can be documented, but they are not directly used for selection and evaluation of the implementation decisions.

Listing 3 shows an implementation of the bookmarks component in the CaesarJ programming language[1, 11] that fulfills the stated goals and is relatively simple and efficient. **Bookmark** is implemented as an aspect that observes the changes in the graph model by intercepting the corresponding joinpoints. CaesarJ allows to create instance level aspects, which can be dynamically deployed on various scopes. A **Bookmark** object is meant to observe only one instance of **Graph** in the system. We use instance level aspect deployment and deploy **Bookmark** only on one graph object (line 7) and on bookmarked nodes (line 12).

The piece of advice at line 17 detects when a node is removed and removes the bookmark from that node. The state changes of nodes are detected by the pointcut at line 22, but, in order to fulfill our efficiency goal, the list of bookmarks is reordered only after the end of a logical model change transaction. We assume that the editor actions correspond to the logical transactions on the data model. Thus, the **TransactionObserver** (line 25) reorders bookmarks if needed at the end of editor actions.¹

2.2 Second Design

The presented design satisfies the stated goals, but we could have stated also other design goals. We may assume that

¹We design **TransactionObserver** as a separate class, because differently from **Bookmark** it must be deployed on the global scope (line 8)

```

1 cclass GraphXPI {
2   pointcut nodeRemoved(Node node) :
3     execution(void Graph.removeNode(*) && args(node);
4   pointcut nodeChanged(Node node) :
5     execution(* setName(..) && this(node);
6 }

```

Listing 4: XPI interfaces

```

1 cclass Bookmark implements PerThisDeployable {
2   ...
3   /* Remove corresponding bookmark after a node is removed */
4   after (Node node) : GraphXPI.nodeRemoved(node) {
5     removeBookmark(node);
6   }
7   ...
8 }

```

Listing 5: Bookmark implementation using XPI interfaces

the application will be extended with new observers that are also interested in the same events. Analogously, we can consider that specifying the boundaries of logical change transactions is not a concern of the bookmark functionality and this information may be useful to other components of the application. We also assume that the structure of the graph model and the editor will change in the future: their methods can be renamed, new methods can be added. We don't want that these changes would destabilize the implementation of bookmarks. The previously stated design goals do not address these problems, therefore, we decide to extend our set of goals:

Goal 5. *The change detection in graph model should be made reusable.*

Goal 6. *The detection of change transaction boundaries should be made reusable.*

Goal 7. *Bookmarks should be stable relative to structural changes in the graph model.*

Goal 8. *Bookmarks should be stable relative to structural changes in the graph editor.*

The design to accommodate the new goals employs crosscutting programming interfaces (XPI)[7]. Such interfaces are used to define stable pointcuts, decoupling aspects from the internal details of the code they advise. Listing 4 displays the XPI created for the graph model (the editor has a similar XPI). `GraphXPI` defines pointcuts for logical events in the graph model. Clients depend only on the pointcut's name and parameters, while it is a responsibility of the model to define how this logical event is translated to a pointcut on the internal structure. Listing 5 shows the `Bookmark` code using XPI instead of directly referencing to the model's joinpoints. Analogously, we define another XPI to specify the logical transaction boundaries in the editor.

The design using XPIs achieves our stated goals, because the pointcuts detecting events are defined outside `Bookmark` aspect and, thus, can be reused in other aspects. The XPIs

```

1 abstract class BookmarkCI {
2   /* expected facet */
3   abstract class BookmarkItem implements Comparable {
4     abstract String getName();
5     abstract int compareTo(Object o);
6   }
7   /* provided facet */
8   abstract void addBookmark(BookmarkItem node);
9   abstract void removeBookmark(BookmarkItem node);
10  abstract void onItemChanged(BookmarkItem node);
11  abstract void onItemRemoved(BookmarkItem node);
12  abstract void onTransactionEnded();
13 }

```

Listing 6: Bookmark Collaboration Interface

also isolate `Bookmark` from the structural changes in the data model and the user interface.

2.3 Third Design

The newly stated design goals forced us to reduce coupling in our design and to increase reusability. Nevertheless, we may think that we didn't maximize the potential reusability in our solution. For example, we could define bookmarks not only to graph nodes, but also to some other data structures. Thus, we are not satisfied with our current design and state one more design goal:

Goal 9. *It should be possible to reuse bookmarks with various data models.*

In order to decouple the reusable bookmark functionality we employ the technique of separation of reusable aspect functionality from its binding to a concrete context[1, 11]. We define the collaboration interface `BookmarkCI` (List. 6) that includes both expected and provided facets of the generalized bookmark component. The collaboration interface introduces `BookmarkItem`, an abstraction intended to be linked to specific objects of an application. The provided facet contains methods to manage bookmarks, e.g. `addBookmark`, as well as methods to notify the component about the changes of interest, e.g. `onItemChanged` or `onTransactionEnded`. `Bookmark`, which now extends `BookmarkCI` and implements only the provided facet of the interface, constitutes the reusable part of the bookmarks functionality.²

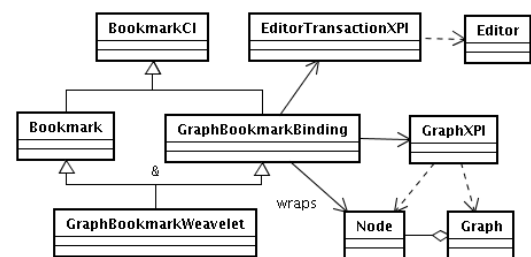


Figure 1: Design with reusable bookmarks

The binding `GraphBookmarkBinding` (List. 7) integrates the bookmark component with the graph editor by implementing the expected facet of `BookmarkCI`. The binding defines

²The implementation of `Bookmark` is omitted for brevity

```

1 abstract cclass GraphBookmarkBinding
2   extends BookmarkCI implements PerThisDeployable { ...
3   GraphBookmarkBinding(Graph g) {
4     this.graph = g;
5     DeploySupport.deployOnObject(this, g);
6     deploy new TransactionObserver();
7   }
8   cclass NodeItem extends BookmarkItem wraps Node {
9     String getName() { return wrappee.getName(); }
10    int compareTo(Object o) { ... }
11  }
12  void addNodeBookmark(Node node) {
13    addBookmark(NodeItem(node));
14    DeploySupport.deployOnObject(this, node);
15  }
16  /* Remove corresponding bookmark after a node is removed */
17  after (Node node) : GraphXPI.nodeChanged(node) {
18    onItemChanged(NodeItem(node));
19  }
20  ...
21 }
22 cclass GraphBookmarkWeavelet
23   extends GraphBookmarkBinding & Bookmark { }

```

Listing 7: Binding Bookmark to Graph

```

1 class Editor {
2   Editor(Graph graph) {
3     this.graph = graph;
4     this.bookmarks =
5       new GraphBookmarkWeavelet(this.graph);
6     ...
7   }
8   ...
9 }

```

Listing 8: Using bookmarks functionality in the editor

a concrete subtype of the `BookmarkItem` that *wraps* `Node` objects (line 8) and adapts them to the methods of `BookmarkItem`. The binding implements advice that use the graph model XPI to intercept the events of interest and to call the corresponding provided methods.

The reusable `Bookmark` aspect is then combined with the binding `GraphBookmarkBinding` as shown at line 23 of List. 7. The resulting weavelet can be instantiated and used by the graph editor as shown in List. 8. Fig. 1 gives an overview of the final design.

3. DISCUSSION

We have presented three alternative designs for the same functional requirements. It would be difficult to say which of these designs is the best. We could think that the design that achieves the most of design goals (the third one) should be considered the best. However, if we take a critical look at it, we will notice that the `Bookmark` module of the first design has been replaced by six modules in the third one (Fig. 1), so the latter is much larger and more complicated.

The new modules have been added in order to increase stability and reusability. We have added the `GraphXPI` and `EditorXPI` modules in order to decouple the aspect from the structure of the advice code and to make the pointcuts reusable. We separated bookmarks functionality into reusable module `Bookmark` and its binding `GraphBookmark-`

`Binding`, and defined `BookmarkCI` as a layer of abstraction between them. Of course, it is still not the ultimate list of all possible design goals for these functional requirements. We could state further design goals, e.g. that the event "a node of a graph have been changed in the transaction" should be reusable or that the aspect scoping on graph and its dependent objects should be reusable.

We can see that the more design goals we state, the more complex is the design that supports all of them. Hence, if we want to evaluate designs, we should at first evaluate how important are the design goals on which they are based and only then examine if they are the best relative to those goals. This implies that we cannot evaluate designs just by analyzing the source code, but we should also consider the context of the problem. For example, if we want to decide how important it is to make an XPI layer between bookmarks functionality and the rest of the editor, we should know if they are developed by different people and how stable the transaction boundaries are going to be. If we want to know how important is the extraction of reusable functionality of bookmarks, we should consider similar products that are developed by the company and also see how stable are the functional requirements for the bookmarks.

Before trying to prioritize various software properties, such as reusability, stability, simplicity and efficiency, we should at first decompose them into specific design goals, because the goals supporting the same property may be of different importance. The more specifically we state a design goal, the more clearly we see what are the assumptions behind it, and, thus, the more precisely we can evaluate its importance. A common practice is to evaluate software properties after the implementation has been made, as if we at first implement our requirements and then want to see if the design of our implementation is good enough. A better approach is to determine the important properties from the very beginning, express them in form of constructive design goals and then translate those goals into appropriate solutions.

In our case study we showed that design process basically consists of two phases: 1) identifying and analyzing design goals, and 2) looking for implementation strategies that fulfill those goals. We have also shown that the results of the first phase have decisive impact on the software design. Unfortunately, the goal analysis phase is often done unconsciously and its results are not documented. A typical documentation of software design tells what the major architectural parts are and how they are related, what patterns are applied, what are the responsibilities of one or another software entity. It is documented what the design of software is, but not why this design was chosen and on what assumptions it is based.

The information about design goals is important not only for the design's evaluation, but also for its understanding and further development. When software is extended with new functionality the existing design goals should be preserved. On the other hand a change in business or technical context may break existing design assumptions and, thus, require reformulation of design goals and appropriate refactorings. If the design goals are not documented they must be implicitly reverse-engineered every time when the soft-

ware needs to be changed. Such reverse-engineering process is error-prone, because, as we already explained, the source code alone is not sufficient to determine the reasons for one or another design decision.

The ability to identify correct design goals and relate them with concrete design decisions is an important part of design experience. The fact that this process is not done explicitly and its results are not documented is a major problem for experience transfer. This explains why pair programming[2] and interactive code reviews are so efficient. The information that is exchanged during the discussion about design contains something more than it is usually documented in source code comments and design descriptions. This "something more" is the information about the design goals, the reasons behind them and their relationship to the design decisions.

4. THE ROLE OF DESIGN PATTERNS

Design patterns[6] aim to encode valuable design experience by providing generalized descriptions of solutions to reoccurring problems. The example presented in the previous chapter corresponds to the problem statement of the Observer design pattern. As we can see, aspect-oriented languages provide a lot of flexibility to deal with this situation. There are a lot of possible design variations that support different design goals. Thus, we think that aspect-oriented languages need more flexible pattern descriptions with more focus on design variations.

In the light of the observations that we made in the previous chapters, we can see that design experience consists basically of two parts: the ability to formulate appropriate design goals and the ability to find solutions that match those goals. Thus, design patterns should also encode the experience of identifying and analyzing the goals. For example, the design goals that we have stated for our example can be made reusable by reformulating them in a generalized way. We can generalize the goals by replacing application specific names with the names of the abstract participants of the design pattern (in our case subject and observer):

Goal 1. *The subject should be independent of its concrete observers.*

Goal 2. *The subject should be independent from the functionality to detect changes in it.*

Goal 3. *Observer should not depend on any observation functionality that it does not need.*

Goal 4. *Redundant expensive updates of observers should be avoided.*

Goal 5. *The change detection in subject should be made reusable.*

Goal 6. *The detection of change transaction boundaries should be made reusable.*

Goal 7. *Observer should be stable relative to structural changes in the subject.*

Goal 8. *Observer should be stable relative to changes of the transaction boundaries.*

Goal 9. *It should be possible to reuse observer with various subjects.*

Our specific implementation decisions can also be expressed in a more general way, e.g.:

1. Use pointcuts and advice to intercept events on the subject.
2. Use pointcuts and advice to determine change transaction boundaries.
3. Use instance level deployment so that pointcuts match only one instance.
4. Create an abstraction layer between observers and subjects in form of XPIs.
5. Abstract observer from subject by using a collaboration interface and listing the expected notifications as methods in the interface.

Subsequently, we should define the relationship between the implementation decisions and the design goals that motivate them. So, in order to support the goal-driven design process design patterns should include the following items in their descriptions:

Design Goals Provides a list of important design goals that should be considered in the context of the design pattern. Each design goal is accompanied by a list of assumptions on which the goal is based and the software qualities supported by the goal.

Solution Describes the set of available implementation decisions, their structure, examples and relationships between them.

Design Guide Describes how the design goals determine the implementation decisions.

It is important to notice that the available implementation decisions depend on the selected programming language. Although most languages support general features for a given paradigm, some specific decisions may not be generalized to all languages. For example, in our list of implementation decisions we propose the use of collaboration interfaces, what may not be possible if the implementation language does not provide support for it. In general we may state that the more flexible a language is, the more implementation decisions and consequently design goals can be fulfilled using this language.

5. RELATED WORK

[3] reviews the state of the art of aspect-oriented design methods. It is important to note that we consider design from somewhat other perspective. We consider design process as the process of making decisions about the implementation

structure. We seek to identify and document design intentions in the implementation rather than to raise the level of abstraction by using language-independent design models.

There have been attempts to provide traceability of non-functional requirements. Gross and Yu [8] propose the use of design patterns to trace non-functional requirements to a goal tree. Cleland-Huang [4] proposes goal centric traceability in order to trace non-functional requirements to architectural assessment models. Both works focus on a top-down design approach and do not discuss the lower level implementation issues.

Extreme programming[2, 10] could be improved with elements of goal-driven design. Software design in extreme programming is driven by the rules to keep design as simple as possible and to refactor it when "code smells" are detected [5, 12]. The process of looking for design problems just by analyzing the source code is not efficient, because the source code does not encode the design intentions that could be motivated by external factors. Thus a better approach would be to document the design goals of each new piece of developed code and refactor it only when the previous design assumptions are not valid anymore or new design goals are stated. We think that goal-driven design would be a lightweight approach to make extreme programming more systematic and won't compromise its agility.

The example of our paper is based on aspect-oriented implementations of Observer design pattern that integrate various design elements from other papers. Hannemann and Kiczales [9] show how Observer design pattern can be replaced with joinpoint interception. The reusable Observer pattern implementation with CaesarJ is demonstrated in [11]. The advantages of crosscutting programming interfaces (XPI) are explained in [7]. In our designs we additionally employ dynamic aspect instantiation and object local deployment and consider the issue of transaction boundaries. None of these papers demonstrate that there can be multiple alternative designs of Observer depending on the stated design goals.

6. CONCLUSIONS AND FUTURE WORK

We have shown that there may be multiple different designs for the same functional requirements and that we cannot judge which is the best one without considering the design goals behind them. Thus, the quality of design largely depends on how good design goals are identified. Following these observations, we argue that the design process should follow an explicit goal-driven approach. By following this process the developer consciously analyzes the context of the problem, identifies the necessary software properties and translates them to specific design goals that constrain design decisions. The results of this process must be explicitly documented, because the design goals are an important source of information for design evaluation, further development and refactorings. Furthermore, design patterns could be improved by extending them with a catalogue of abstracted design goals and relating these with the design options of the pattern. This is especially important for aspect-oriented languages, because they enable a lot of design variations.

We intend to elaborate the notion of design goals, to refine

the process of describing design patterns further and to provide a set of design pattern descriptions in the new form. In addition we are going to try out the goal-driven development process, supported by design patterns, in case studies and evaluate its advantages and problems.

7. REFERENCES

- [1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. *Transactions on AOSD I, LNCS*, Vol 3880:135–173, 2006.
- [2] K. Beck and D. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999.
- [3] R. Chitchyan, A. Rashid, P. Sawyer, J. Bakker, M. P. Alarcon, A. Garcia, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of aspect-oriented analysis and design. Technical report, AOSD-Europe Project Deliverable No: AOSD-Europe-ULANC-9, 2005.
- [4] J. Cleland-Huang. Toward improved traceability of non-functional requirements. In *Proceedings TEFSE'05*, pages 14–19, New York, NY, USA, 2005. ACM Press.
- [5] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [7] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60, 2006.
- [8] D. Gross and E. Yu. From non-functional requirements to design through patterns. *Requirements Engineering Journal*, 2001.
- [9] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings OOPSLA'02*, pages 161–173, New York, NY, USA, 2002. ACM Press.
- [10] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
- [11] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings AOSD'03*, pages 90–99. ACM Press, 2003.
- [12] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings AOSD'05*, pages 111–122, New York, NY, USA, 2005. ACM Press.