From patterns to components

Author(s):
Arnout, Karine Marguerite Alice

Publication Date:
2004

Permanent Link:

https://doi.org/10.3929/ethz-a-004715194 →

Doctoral Thesis ETH No. 15500

From Patterns to Components

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Sciences

presented by
Karine Marguerite Alice ARNOUT
Diplôme d'ingénieur en télécommunications, ENST Bretagne
born 18.05.1978
French citizen

accepted on the recommendation of
Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Peter Müller, co-examiner
Prof. Dr. Emil Sekerinski, co-examiner

2004

Abstract

If design patterns are reusable design solutions to recurring design problems, they are not reusable in terms of code. Programmers need to implement them again and again when developing new applications. The challenge of this thesis was to bring design patterns to a higher degree of reusability: transform design patterns into reusable components that programmers could use and reuse without recoding the same functionalities anew for each new development.

The contributions of this thesis do not only target program implementers. They should also be useful to program designers, library developers, and programming language designers. Indeed, the transformation of patterns into components, which I call "componentization", revealed that the traditional architecture of some design patterns was not optimal. Rethinking the design yielded solutions that are easier to use, easier to extend, and covering a wider range of application problems. Besides, considering programming language extensions permitted to find better solutions in some cases.

The first mention of the word "componentization" was in [Arnout 2003b].

This thesis reviews all patterns described in *Design Patterns* by level of componentizability (possibility to transform a design pattern into a reusable component) and describes the corresponding software component whenever applicable. It uses Meyer's definition of component: a reusable software element, typically some library classes, usually in source form (not binary form), which differs from Szyperski's view of components. The reusable components (the Pattern Library) are written in Eiffel because the language offers several object-oriented mechanisms that were useful for the pattern componentization: genericity (constrained and unconstrained), multiple inheritance, agents. The support for Design by Contract™ was also a key to the success of this work. However, the approach is not bound to Eiffel. It would be easy to develop the Pattern Library in another programming language on condition that this language provides the object-oriented mechanisms needed for the componentization process. Chapter 22 gives a few examples going in that direction, using Java and C# as examples.

[Gamma 1995].

"componentization" is defined on page 26.

[Meyer 1997].

[Szyperski 1998].

See [Dubois 1999] *and chapter 25 of* [Meyer 200?b] *about agents.*

Around 65% of the patterns described in *Design Patterns* could be turned into reusable components. For example, the componentization of the *Observer* pattern resulted in the Event Library, which covers both the *Observer* pattern and the general idea of publish-subscribe and event-driven development. The *Visitor* pattern resulted in a Visitor Library, which simplifies the implementation of the double-dispatch mechanism by using the Eiffel agent mechanism. (It could also be achieved through reflection in other languages although it would not be type-safe anymore.)

See chapter 6 for a complete description of the patterns' componentizability and the corresponding pattern componentizability classification.

Among the remaining 35% of patterns that are not componentizable, less than 10% could not been improved at all because they rely on context-dependent information. It is the case of the *Facade* and *Interpreter* design patterns.

For the other 25% that are not componentizable, it is possible to write skeleton classes, and sometimes even provide a method to fill in these classes. One of the concrete outcomes of this thesis is a tool called Pattern Wizard, which generates these skeleton classes automatically. Chapter 21 presents the design and implementation of the wizard, and explains when and how to use it.

See "Conventions", page 14 for a definition of "componentizable patterns" and "non-componentizable patterns".

Résumé

Si les patrons de conception sont des solutions réutilisables — au niveau design — à des problèmes de conception récurrents, ils ne sont pas réutilisables au point de vue code. Les programmeurs doivent les réimplanter à chaque nouveau développement. Le défi de cette thèse était d'apporter un nouveau degré de réutilisabilité: transformer les patrons de conception en composants réutilisables que les programmeurs peuvent utiliser et réutiliser sans avoir à réécrire les mêmes fonctionnalités à chaque nouveau développement.

Les contributions de cette thèse ne sont pas simplement destinées aux programmeurs. Elles devraient également être utiles aux concepteurs d'applications, aux développeurs de bibliothèques logicielles et aux concepteurs de langages de programmation. En effet, la transformation de patrons en composants a montré que l'architecture traditionnelle de certains patrons de conception n'était pas optimale. Repenser la conception a permis d'obtenir des solutions plus faciles a utiliser, plus faciles à étendre et couvrant un plus grand nombre de problèmes. Par ailleurs, le fait de considérer des extensions du langage de programmation a permis de trouver de meilleures solutions dans certains cas.

Cette thèse examine les patrons de conception décrits dans le livre *Design Patterns* en suivant leur niveau de "componentizabilité" (possibilité de transformer un patron de conception en composant réutilisable), et décrit le composant logiciel correspondant chaque fois que cela est possible. La définition de composant utilisée est celle de Bertrand Meyer : un composant est un élément logiciel réutilisable, typiquement un ensemble de classes de bibliothèque, habituellement sous forme de code souce (non sous forme binaire), ce qui diffère de l'idée de composant selon Szyperski. Les composants réutilisables sont écrits en Eiffel parce que le langage offre plusieurs mécanismes à objets qui se sont avérés utiles pour la transformation de patrons de conception en composants : généricité (contrainte ou non contrainte), héritage multiple, agents. Le support pour la conception par contrats (Design by Contract™) contribua aussi largement au succès de ce travail. Toutefois, l'approche ne se limite pas à Eiffel. Il serait facile de développer une "bibliothèque de patrons de conception" dans un autre langage de programmation à condition que ce langage fournisse les mécanismes à objets nécessaires à la transformation de patrons en composants. Le chapitre 22 donne quelques exemples allant dans cette direction, utilisant Java et C# en exemples.

[Gamma 1995].

"componentization" est défini page 26.

[Meyer 1997].

[Szyperski 1998].

Voir [Dubois 1999] *et le chapitre 25 de* [Meyer 200?b] *à propos des agents.*

Environ 65% des patrons de conception décrits dans le livre *Design Patterns* ont pu être transformés en composants réutilisables. Par exemple, la transformation du patron de conception *Observer* a abouti à une bibliothèque nommée *Event Library* couvrant non seulement le patron *Observer* mais aussi l'idée générale de développement géré par événements. Le patron de conception *Visitor* a abouti à une bibliothèque (*Visitor Library*) simplifiant l'implantation du mécanisme de "*double-dispatch*" en utilisant le mécanisme Eiffel des agents. (Ce comportement pourrait s'obtenir par la réflexion dans d'autres langages de programmation bien que la sécurité des types ne serait plus garantie.)

Voir chapitre 6 pour une description complète de la componentisabilité des patrons de conception et la classification de componentisabilité des patrons de conception correspondante.

Parmi les 35% de patrons de conception restants, moins de 10% n'ont pu être améliorés du tout car ils reposent sur des informations dépendant du contexte. C'est le cas des patrons *Facade* et *Interpreter*.

Pour les autres 25% qui ne peuvent être transformés en composants réutilisables, il est possible d'écrire des classes "squelettes" et parfois même de fournir une méthode pour compléter ces classes. L'un des résultats concrets de cette thèse est un outil nommé *Pattern Wizard* générant ces squelettes de classes automatiquement. Le chapitre 21 présente la conception et l'implantation de l'outil, et explique quand et comment l'utiliser.