On the evolution complexity of design patterns

Tom Mens [1]

Software Engineering Lab, Université de Mons-Hainaut
B-7000 Mons, Belgium

Amnon H. Eden [2]

Department of Computer Science, University of Essex
and Center For Inquiry International

Abstract

Software co-evolution can be characterised as a way to "adjust" any given software implementation to a change ("shift") in the software requirements. In this paper, we propose a formal definition of evolution complexity to precisely quantify the cost of adjusting a particular implementation to a change ("shift") in the requirements. As a validation, we show that this definition formalises intuition about the evolvability of design patterns.

Key words: software evolution, design pattern, complexity

1 Introduction

According to Lehman's first law of software evolution, "*An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.*" [10]. Despite growing awareness of this law, evolution of industrial quality software systems is notoriously expensive. It is therefore paramount to investigate the flexibility or evolvability of software and to find ways to quantify it.

Claims about the evolvability of design patterns, architectural styles and object-oriented programs have appeared in numerous publications. Most authors, however, stop short from quantifying the benefits gained by using a particular implementation policy or qualifying the claim by the class of "shifts" (i.e., changes in the software requirements) it best allows.

Experience shows that flexibility is relative to the change. Every manufactured product is designed to accommodate to a specific class of changes,

[1] Email:tom.mens@umh.ac.be
[2] Email:eden@essex.ac.uk

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

which makes it flexible towards these changes but inflexible towards others. Locomotives, for example, are very flexible with relation to the type of cars they can pull but they can hardly be adapted to tracks of a different size.

The same applies to software: Every implementation policy (e.g., architectural style [14], design pattern [5]) is flexible towards the class of changes it was designed for. For example, a program designed using Layered Architecture style, such as the Unix operating system, adapts easily to changes in the uppermost layer (the *application layer*) but changes to the lowermost layer (the *kernel*) are much more difficult to implement. As another example, the Visitor design pattern "*makes adding new operations easy*" while "*adding new concrete element classes is hard*" [5] (pp. 335–336).

There is a common misconception in the software engineering community, however, that flexibility and evolvability are absolute qualities. Consider for example the following standardised definitions:

Flexibility The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [6]

Extendability The ease with which a system or component can be modified to increase its storage or functional capacity [6]

Changeability The capability of the software product to enable a specified modification to be implemented [7]

These definitions fail to observe that flexibility of a program is relative to a particular class of changes. As a notable exception, Gamma *et al.* were more precise in characterising the quality of flexibility [5] : "*Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a* ***particular kind of change***."

In this paper we examine the relationship between *shifts*, i.e., changes in the requirements, and *adjustments*, i.e., the respective changes in the implementation. We offer a precise definition of *evolution complexity* and propose various metrics to approximate its cost. To illustrate its usefulness, we compute the evolution complexity of adjusting a selected number of design patterns to specific shifts and prove informal statements about how difficult is each. For example, we prove that the "*Visitor pattern makes adding new operations easy*" by showing that the evolution complexity of adding an operation is constant, and that "*Adding new concrete element classes is hard*" by showing that the evolution complexity of adding a new concrete element is linear in the number of visitors.

Furthermore, we use evolution complexity to show that whether one implementation policy is more evolvable than another depends on the class of changes in question. For example, given the requirements for representing a deterministic finite-state automaton, we demonstrate that a procedural implementation is more evolvable towards shifts in the alphabet, while an object-oriented implementation (using the State pattern) is more evolvable towards

shifts in the set of states.

To summarise, the intended contributions of this paper are: (a) To formalise and prove the intuition behind flexibility and evolvability of specific implementation policies, in particular of design patterns; (b) To provide means for choosing a particular implementation policy; (c) To provide means for quantifying the cost of implementing specific changes;

2 Setting the scene

In this section, we clarify the terminology that will be used in the remainder of the paper. These definitions are summarized in Table 1 and illustrated in Figure 1.

Table 1
Key to notation

requirements	$r_{dfsa}, r_{gui}, ...$
implementations	$i_{before}, i_{after}, ...$
problem domain	$\mathcal{D} = \{D_1, \ldots, D_n\}$
shift	$\sigma = \sigma_\delta(r, U, u)$
adjustment	$\alpha = (i, i')$
co-evolution step	$\epsilon = (\sigma, \alpha)$

Requirements. A well-defined specification of the program's expected behaviour, expressed in terms of the problem domain $\mathcal{D}$. For example:

r_{dfsa} := *Implement a deterministic finite state automaton with states S and alphabet L.*

In this case, the problem domain $\mathcal{D}_{dfsa} = \{S, L\}$.

r_{gui} := *Implement a GUI to represent and instantiate a family of widgets (e.g., button, window and menu) in a specified set of operating systems (e.g., Windows and MacOS).*

In this case, the problem domain $\mathcal{D}_{gui} = \{W, O\}$, where W = { button, window, menu } and O = { Windows, MacOS }.

Implementation. The actual program that is the subject of the evolution effort. Each implementation is designed to satisfy a specific set of requirements and must be adjustable to changes therein. In this paper, we will use the term *implementation policies* to reflect that fact that we abstract away from language-specific details.

Shift. A specific change to a given set of requirements. More specifically, a shift may add entities to, or remove entities from, the sets contained in the problem domain $\mathcal{D}$. A shift is represented as a function $\sigma_\delta(r, D, d)$ where

$D \in \mathcal{D}$, and d is added to D if $\delta =$"+", while d is removed from D if $\delta =$"−".

For example, the shift $\sigma_+(r_{dfsa}, L, l)$ adds a letter l to the alphabet L in r_{dfsa}.

Adjustment. A specific change to a given implementation, triggered by a shift in the requirements. As shown in Figure 1, each implementation needs to be "adjusted" in order to satisfy the changed requirements. An adjustment is represented as a pair $\alpha = (i, i')$ where i is the old implementation and i' is the new implementation.

Co-evolution step. A pair $\epsilon = (\sigma, \alpha)$ consisting of a shift σ transforming r into r' and an adjustment $\alpha = (i, i')$, such that implementation i satisfies requirements r and implementation i' satisfies requirements r'.

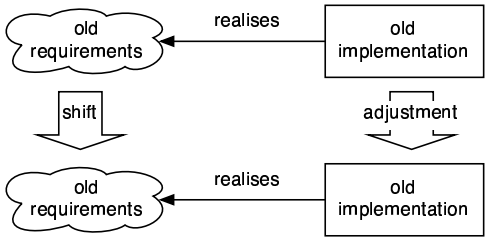

Fig. 1. A co-evolution step.

3 Evolution complexity

In this section, we use the previously introduced vocabulary to define a notion of *evolution complexity*. With this definition we attempt to quantify the cost of a co-evolution step, i.e., the effort that is required to *adjust* a specific implementation to a particular *shift*.

In order to define evolution complexity, we draw an analogy with the notion of *computational complexity*. "The theory of computational complexity is concerned with estimating the resources a computer needs to solve a problem." [16] In analogy, evolution complexity is concerned with estimating the resources required to "evolve" an implementation. Using metaprogramming as a central metaphor, we suggest that evolution complexity, i.e., the complexity of the process of adjusting the implementation to changed requirements, can be approximated by calculating the computational complexity of a metaprogramming algorithm that actually makes these adjustments to the software implementation. This leads us to the most general formulation of evolution complexity:

Definition 3.1 The complexity $\mathcal{C}(\epsilon)$ of a co-evolution step ϵ is the complexity of the meta-programming algorithm that implements it.

Note that there are as many meta-programming algorithms (for imple-

menting the adjustment) as there are manual ways for implementing them. We seek to approximate the manual (adjustment) process by measuring the respective (hypothetical) metaprogramming process.

Since $\epsilon = (\sigma, \alpha)$ is a pair, this definition implies that the actual effort required to "evolve" a program correlates primarily with two factors:

(i) The **shift** σ, i.e., the distance between the old and the new requirements;

(ii) The **adjustment** α, i.e., the distance between the implementation before and after the change.

The obvious question is: How can we measure these distances? In the following section, we illustrate some of the ways to approximate evolution complexity.

4 Case studies

In this section, we use evolution complexity to quantify the evolvability of different implementation policies. We do this in two different ways. To compare the evolvability of different implementations (e.g., procedural versus object-oriented implementation), we fix the shift and calculate the evolution complexity for each implementation. To compare the difficulty of implementing different shifts, we fix the implementation and calculate the evolution complexity for each shift.

The presentation of each case study will be structured in the same way: *Example*; *Requirements*; *Shifts*; *Implementation*; *Metric*; *Analysis*; *Discussion.*

4.1 Case study 1: Visitor

The Visitor design pattern [5] can be used to "*Represent an operation to be performed on the elements of an object structure.*" The class of shifts that the Visitor supports is declared from the outset: "*Visitor lets you define a new operation without changing the classes of the elements on which it operates.*" Further on in the chapter, the authors are more explicit: "*Visitor makes adding new operations easy*" and "*Adding new concrete element classes is hard*". In this case study, we prove these statements and quantify how easy or hard is each one of these co-evolution steps.

Example

Gamma *et. al* describe the representation of abstract syntax trees as object structure and the collection of operations that a compiler performs on each element in the tree. Figure 2 illustrates this example.

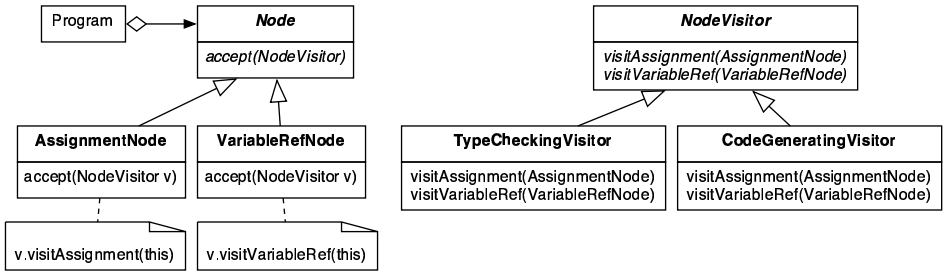

Fig. 2. Example for the Visitor pattern.

Requirements
r_{visit} := *Represent a set of operations* O *that need to be performed on a family of elements* E.
The problem domain is $\mathcal{D}_{visit} = \{O, E\}$

Shifts
In this case study we will consider the following two shifts to r_{visit}:
$\sigma_+(r_{visit}, O, op)$ = *add operation op to* O
$\sigma_+(r_{visit}, E, e)$ = *add element e to* E

Implementation
i_{visit} := The implementation policy described by the Visitor pattern as illustrated in Figure 2.

By this policy, two class hierarchies, `NodeVisitor` and `Node` are used to represent the set of operations O and the family of elements E, respectively. Each operation $op \in O$ is represented as a class in the `NodeVisitor` hierarchy, and each element $e \in E$ is represented as a class in the `Node` hierarchy.

Metric
Below we definite a simple metric that calculates the complexity of a co-evolution step by counting the number of classes affected by it.

Definition 4.1 Let (σ, α) be a co-evolution step such that $\alpha = (i, i')$. Let Δ be the symmetric set difference, i.e., $A \Delta B = (A \backslash B) \cup (B \backslash A)$. Let $Classes(i)$ = the set of classes in i .
$\mathcal{C}^1_{Classes}(\sigma, \alpha) := |Classes(i) \Delta Classes(i')|$

In class-based languages such as C++, Smalltalk and Java, $Classes(i)$ yields the set of classes defined in the program. For example,
$Classes(i_{visit})$ = {`Node`, `AssignmentNode`, `VariableRefNode`, `NodeVisitor`, `TypeCheckingVisitor`, `CodeGeneratingVisitor`}

Analysis
Gamma *et. al* recognized the class of shifts that the visitor accommodates to. We will use the above metric to prove some of their claims about the flexibility

of the Visitor pattern:

- "*Adding new concrete element classes is hard.*" ([5], p.335) To prove this statement, consider the shift $\sigma_+(r_{visit}, E, e)$. In order to adjust our implementation i_{visit} to this shift, we need to add a method to every class in the visitors hierarchy O. Thus, the number of implementation entities affected by this shift equals the number of operations:

$$\mathcal{C}^1_{Classes}(\sigma_+(r_{visit}, E, e), i_{visit}) = |O|$$

  Read: *The class-level evolution complexity of adding an element to the Visitor pattern is linear in the number of operations.*

- "*Visitor makes adding new operations easy.*" ([5], p.336) To prove this statement, consider the shift $\sigma_+(r_{visit}, O, op)$. In order to adjust our implementation i_{visit} to this shift, we need to add a new class to the visitors hierarchy. Thus, the number of implementation entities affected by this shift is 1:

$$\mathcal{C}^1_{Classes}(\sigma_+(r_{visit}, O, op), i_{visit}) = 1$$

  Read: *The class-level evolution complexity of adding an operation to the Visitor pattern is constant.*

The results of this case study are summarized in Table 2.

$\mathcal{C}^1_{Classes}$	$\sigma_+(r_{visit}, E, e)$	$\sigma_+(r_{visit}, O, op)$
i_{visit}	$\|O\|$	1

Table 2
Class-level evolution complexity for shifts of r_{visit}

Discussion
Our analysis proves the intuitions of Gamma *et al.* about the Visitor pattern, but it also goes beyond that: The evolution complexity metric quantifies the effort required to add a new element to the elements hierarchy. Specifically, it indicates that effort required to add a new operation is proportional to the number of operations.

4.2 Case study 2: Abstract Factory

The Abstract Factory pattern can be used to "*Provide an interface for creating families of related or dependent objects without specifying their concrete classes.*" [5] Below, we show that Abstract Factory is only flexible towards specific shifts, and that the adjustments necessary can also be very expensive.

Example
A typical example is an object-oriented graphical user interface (GUI) for creating graphical "widgets", such as windows, buttons and menus. Each win-

dowing environment offers a variation on each one of these widgets. Figure 3 illustrates the "families" of widgets for three windowing environments.

A client in cross-platform implementation that needs a new button, for example, must decide which variation of button is appropriate. The obvious way to do this would be to use complex conditional statements that determine the appropriate variation. The alternative, offered by the Abstract Factory design pattern, is to offer each such client a uniform interface for generating each widget, and to delegate the decision which version is appropriate to a "concrete factory" object. The dashed lines in Figure 3 illustrate the "create" relation between factory method, concrete factories and the products hierarchy.

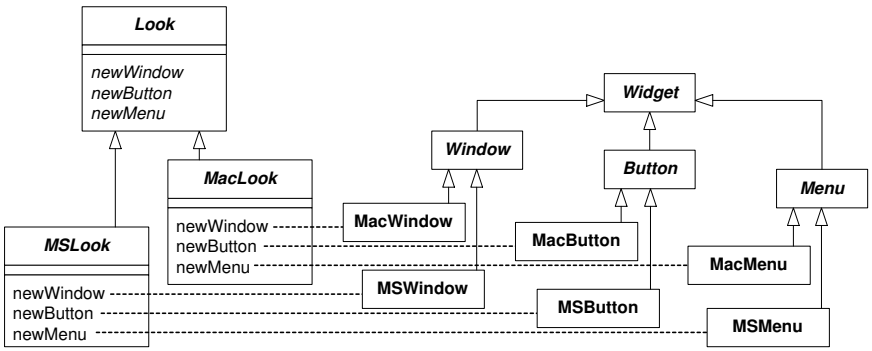

Fig. 3. Example for *Abstract Factory*.

Requirements

$r_{conf/prod}$:= *Given a set of clients* K*, a family of configurations* $C = \{c_1, ...c_n\}$*, and a family of products* $P = \{p_1, ...p_m\}$*. Every client* k *in* K *needs a new instance* p *in* P *depending on the current configuration* c *in* C*.*
The problem domain $\mathcal{D}_{conf/prod} = \{K, C, P\}$.

Shifts

In this case study we will consider the following two shifts to $r_{conf/prod}$:

$\sigma_+(r_{conf/prod}, C, c)$:= *add configuration* c *to* C
$\sigma_-(r_{conf/prod}, P, p)$:= *remove product* p *from* P

Implementation policies

We will compare two implementation policies to $r_{conf/prod}$:

$i_{abs-factory}$:= use the Abstract Factory design pattern ([5], page 87). The solution this pattern dictates consists of two class hierarchies, *factories* and *products*. Each class in the *factories* hierarchy ("concrete factory") is responsible for creating products of a specific configuration $c \in C$. For this purpose, each concrete factory defines a method ("factory method") for each product $p \in P$. Overall, there are $|C|$ concrete products with $|P|$ methods each in this implementation.

i_{cond} := use conditional style. Each client that needs a specific product implements a switch statement that has a conditional branch for each configuration $c \in C$.

In both implementations, we assume that there are $|K|$ separate clients (i.e., $|K|$ classes in a class-based language) such that each client needs a new instance of one or more products.

Analysis

An example of shift $\sigma_-(r_{conf/prod}, P, p)$ would be to remove the product `Menu` in Figure 3. This requires us to remove all concrete subclasses of `Menu`, as well as the methods `newMenu` that are implemented in each concrete factory.

Table 3
Class-level evolution complexity for shifts of $r_{conf/prod}$

$\mathcal{C}^1_{Classes}$	$\sigma_+(r_{conf/prod}, C, c)$	$\sigma_-(r_{conf/prod}, P, p)$
$i_{abs-factory}$	$\|P\|$	$\|C\| + \|P\|$
i_{cond}	$\|K\|$	$\|K\|$

The effect of implementation policy $i_{abs-factory}$ on the evolution complexity is described informally in [5], pages 89 and 90: "*It makes exchanging product families easy*"; "*supporting new kinds of products is difficult*". Again, we will quantify exactly how easy or difficult it is.

As shown in Table 3, the evolution complexity at class level of $i_{abs-factory}$ is linear for both shifts. For shift $\sigma_+(r_{conf/prod}, C, c)$, it is *linear in the number of products*, since we need to add a new concrete product class for each possible product, to specify how each product needs to be addressed by the new configuration. For shift $\sigma_-(r_{conf/prod}, P, p)$, it is *linear in the number of configurations* ***and*** *products*, since we need to remove all concrete product classes for the considered product, and we need to remove a corresponding method in each of the configuration classes.

For implementation i_{cond}, the evolution complexity is *linear in the number of clients* for both shifts.

Discussion

The results presented in Table 3 demonstrate that the decision to use the Abstract Factory design pattern is not straightforward. If the number of clients is very small then the overhead in using the pattern is not justified. Similarly, it shows that removing a product can be a labour-intensive task even if the pattern is used.

Obviously, the evolution complexity is not the only criterion that should be used for preferring a particular implementation policy over another. For example, the *Abstract Factory* has a number of other important advantages that are

not measured by our evolution complexity measure: "*it isolates clients from implementation classes*"; "*it promotes consistency among products*". Thus, evolution complexity only captures one of many concerns that guide designers in the choice of a particular implementation policy.

4.3 Case study 3: Procedural vs. object-oriented implementation

Object-oriented programming is hailed for promoting flexibility. Experienced programmers, however, observe that flexibility is relative to a specific class of changes that our implementation must specifically be designed to accommodate. As our analysis of the Visitor pattern demonstrated, changes to the interface of the base class in a large inheritance class hierarchy can be very expensive to implement, but adding a new "leaf" class is usually very easy.

In this case study, we use evolution complexity to compare the flexibility of a procedural and an object-oriented implementation policy in the context of a specific problem: the representation of a finite-state automaton. Our analysis will demonstrate that an object-oriented implementation is more flexible towards changes in the set of states, while the procedural implementation is more flexible towards changes in the alphabet.

Example
Consider a digital clock with three display states *Display Hour*, *Display Seconds* and *Display Date* and two setting states *Set Hour* and *Set Date*. The clock accepts input from two buttons b_1 and b_2, which are used to change between states or to perform a specific action depending on the current state. The clock behaviour is modelled in Figure 4 as a deterministic finite state automaton with set of states S={*Display Hour*, *Display Seconds*, *Display Date*, *Set Hour*, *Set Date*} and an alphabet L={b_1,b_2}.

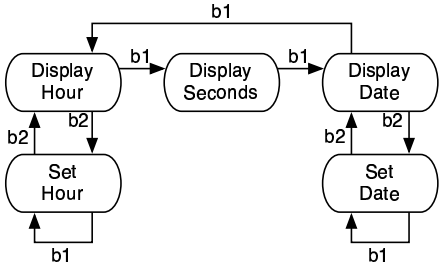

Fig. 4. State machine for a digital clock.

Requirements
r_{dfsa} := *Implement a deterministic finite state automaton (DFSA) with a set of states S and a set of letters L (i.e., an alphabet).*
The problem domain $\mathcal{D}_{dfsa} = \{S, L\}$.

Shifts

In this case study we will consider the following two shifts to r_{dfsa}:

$\sigma_+(r_{dfsa}, L, l) :=$ *add letter* l *to the alphabet* L

$\sigma_+(r_{dfsa}, S, s) :=$ *add state* s *to the set of states* S

Implementations

We will compare two implementation policies for the digital clock:

i_{state} = use the *State* design pattern ([5], p.305). This pattern dictates that each state $s \in S$ is represented by a separate class such that every state class defines a method for each $l \in L$. In all, there are $|S|$ classes with $|L|$ methods per class in this implementation. This implementation is illustrated below:

```
interface ClockState {
  void b1(); \\ button 1 pressed
  void b2(); \\ button 2 pressed
}
class DisplayHour implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
class DisplaySecond implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
class DisplayDate implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
class SetHour implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
class SetDate implements ClockState {
  public void b1() {\* b1 pressed *\}
  public void b2() {\* b2 pressed *\}
}
```

i_{cond} = use *conditional style*, ([5], p.307): "*An alternative is to use data values to define internal states and have context operations check the data explicitly. But then we'd have look-alike conditional or case statement scattered throughout the context's implementation.*" In this style of implementation, we define one class that contains a method for each letter $l \in L$ such that the body of each method consists of a switch statement that contains a conditional branch for each state $s \in S$. This implementation is given below:

```
enum states = {
  DisplayHour, DisplaySecond, DisplayDate,
```

```
  SetHour, SetDate} state;  // Current state

void b1() { \\ button 1 pressed
  switch (state) {
    case DisplayHour: \*...*\;
    case DisplaySecond: \*...*\;
    case DisplayDate: \*...*\;
    case SetHour: \*...*\;
    case SetDate: \*...*\; }
}
void b2() { \\ button 2 pressed
  switch (state) {
    case DisplayHour: \*...*\;
    case DisplaySecond: \*...*\;
    case DisplayDate: \*...*\;
    case SetHour: \*...*\;
    case SetDate: \*...*\; }
}
```

Metric

The class-level metric of Definition 4.1 is too coarse since it assumes that all software entities require approximately the same effort to be adjusted. Such an approximation is useful when the number of software entities is large or when there is insufficient detailed information about individual entities. For example, at the design level, before the implementation is complete, we may only have information about the classes and their variables, but not necessarily about their methods. When the implementation is complete, however, and when it is evident that some software entities are more difficult to adjust than others, we may replace the class-level metric by a generalized metric that measures the (software) evolution complexity of individual modules:

Definition 4.2 Let (σ, α) be a co-evolution step such that $\alpha = (i, i')$. Let Δ be the symmetric set difference. Let $Modules(i)$ = the set of modules in i. Let μ be a software complexity metric that is defined for all $x \in Modules(i) \cup Modules(i')$,

$$\mathcal{C}^{\mu}_{Modules}(\sigma, \alpha) := \sum_{x \in Modules(i) \Delta Modules(i')} \mu(x)$$

Let us show how the generalized metric can be used in specific cases:

$\mathcal{C}^{\mathbf{1}}_{\mathbf{Classes}}$ When $Modules = Classes$ and the software complexity $\mu(c) = 1$ for all $c \in Classes(i)$, the generalized metric is reduced to the class-level evolution complexity metric of Definition 4.1.

$\mathcal{C}^{\mathbf{1}}_{\mathbf{Methods}}$ When $Modules = Methods$ (the set of all methods in i) and the software complexity $\mu(m) = 1$ for all $m \in Methods(i)$, we have a method-level evolution complexity metric. This metric measures the complexity of

a co-evolution step by counting the number of methods affected by it. It treats all methods as equal with respect to their change effort.

$\mathcal{C}^{\mathbf{CC}}_{\mathbf{Methods}}$ When $Modules = Methods$ and $\mu = CC$ (*cyclomatic complexity*, [12]), the metric takes into account the cyclomatic complexity of each method such that adjustments applied to a "complicated" method (e.g., a method with many conditional statements) are more expensive than adjustments applied to a "simple" method.

Analysis

The effect of the two implementation policies i_{state} and i_{cond} on the evolution complexity is described informally in [5]. For example, pages 307 and 308 mention for i_{state}: "new states ... can be added easily"; "decentralizing the transition logic in this way makes it easy to modify or extend the logic". For i_{cond}, page 307 mentions: "Adding a new state ... complicates maintenance." Our formal framework lets us precisely quantify this intuition in terms of the entities of the problem domain $\mathcal{D}_{dfsa} = \{S, L\}$ that are affected by a particular shift, assuming a particular implementation policy.

$\mathcal{C}^{1}_{Modules}$	$\sigma_+(r_{dfsa}, L, l)$	$\sigma_+(r_{dfsa}, S, s)$
i_{state}	$\lvert S\rvert$	1
i_{cond}	1	$\lvert L\rvert$

Table 4
Class-level evolution complexity for shifts of r_{dfsa}

Analysing the evolution complexity at the class level (Table 4), we observe for shift $\sigma_+(r_{dfsa}, L, l)$ that implementation policy i_{state} is more difficult to evolve than implementation policy i_{cond}. Indeed, the *State* design pattern requires us to add a new method (corresponding to the new letter to be added) to each of the $|S|$ state classes. Hence, the evolution complexity for i_{state} is *linear in the number of states*, whereas it is constant for i_{cond}.

$\mathcal{C}^{CC}_{Methods}$	$\sigma_+(r_{dfsa}, S, s)$
i_{state}	$\lvert L\rvert$
i_{cond}	$\lvert S\rvert \times \lvert L\rvert$

Table 5
Method-level evolution complexity for shifts of r_{dfsa}

Focussing on the method level (Table 5) shows us two other important results. At this level of abstraction, the evolution complexity for shift $\sigma_+(r_{dfsa}, L, l)$ is the same for both implementation policies (linear in the number of states). Shift $\sigma_+(r_{dfsa}, S, s)$, on the other hand, is more difficult to

evolve for implementation policy i_{cond} than for i_{state} (complexity $|S \times L|$ and $|L|$, respectively). These results were obtained by using the cyclomatic complexity metric CC for each affected method. It reflects the intuition that an implementation policy with lots of conditional statements is more difficult to evolve than one with few conditionals.

Discussion
While Table 4 shows that *adding a state* is constant in the number of states, *removing a state* with implementation policy i_{state} is potentially linear in the number of states, since we need to modify the state transition function, which is embedded in, and dispersed over, all state classes. As an alternative implementation policy, we could opt for a variant of the *State* design pattern, where state transitions are implemented in the context. Needless to say, this implementation policy will yield different results for the evolution complexity when compared to i_{state} and i_{cond}.

5 Related work

Despite its importance, cost estimation in the context of software maintenance and software evolution remains relatively unexplored. Jørgensen [8] used several models to predict the effort of randomly selected software maintenance tasks. The size of individual maintenance tasks was measured in LOC. Sneed [15] proposed a number of ways to extend existing cost estimation methods to the estimation of maintenance costs. Ramil *et al.* [13] provided and validated six different models that predict software evolution effort as a function of software evolution metrics.

All of the above approaches rely make use of software metrics, and can rely on experimentally validated results of the software metrics community [3,4,12,17]. While our definition can also incorporate existing software complexity metrics (e.g., cyclomatic complexity) easily, an important distinction is that our approach goes beyond existing attempts to measure the "evolvability" of implementations, since we have shown that it is not possible to give an absolute measure of the evolvability of a particular program. Instead, the evolvability depends on the chosen implementation policy and on the changes in the requirements that are likely to be performed.

Perhaps a better (i.e., less absolute) way to measure the changeability of a software system relies on algorithms or measures that compute the impact of changes [11]. For example, Chaumun *et al.* [2] report on experimental results with a change-impact model for object-oriented systems.

Because the cost and complexity of software evolution may depend on the type of evolution activity, we also require a more objective and finer granularity recognition of types of software evolution activities. An attempt to make such an objective classification of evolution activities was carried out in [1].

6 Conclusion

In this paper we proposed a formal definition of evolution complexity to quantify the cost of adjusting a particular implementation policy to a change ("shift") in the requirements. As a case study, we used our formalism to formally validate the intuition that design patterns improve the evolvability of programs. We were able to determine precisely to which extent, and for which changes in the requirements, a particular design pattern is "more evolvable" than another implementation policy. Despite all this evidence, a more scientific empirical validation of our proposed evolution complexity metric remains to be done.

In general terms, our formalism allows us to precisely quantify the cost of implementing particular changes in the requirements, and to choose the most flexible implementation policy to implement these changes. Our formalism can be used at various levels of granularity (e.g., class level and method level). This implies that it can also be used during the design phase, when not all implementation details are known yet.

In the future we even plan to apply our ideas at an architectural level (as opposed to a design level), by considering changes to *architectural styles* [14] instead of design patterns. This will allow us to adapt existing architectural cost estimation models [9] to an evolution context.

7 Acknowledgements

This research was carried out in the context of the scientific network RELEASE financed by the ESF. We thank Jeff Reynolds for his useful comments and for his contribution to the definition of evolution complexity. We also thank Bart Dubois for proofreading this paper.

References

[1] Chapin, N., J. Hale, K. Khan, J. Ramil and W.-G. Than, *Types of software evolution and software maintenance*, Journal of software maintenance and evolution **13** (2001), pp. 3–30.

[2] Chaumun, M. A., H. Kabaili, R. K. Keller and F. Lustman, *A change impact model for changeability assessment in object-oriented software systems*, Science of Computer Programming **45** (2002), pp. 155–174.

[3] Chidamber, S. R. and C. F. Kemerer, *A metrics suite for object oriented design*, Transactions on Software Engineering **20** (1994).

[4] Fenton, N. E. and S. L. Pfleeger, "Software Metrics: A Rigorous and Practical Approach," PWS Publishing Company, 1997.

[5] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[6] IEEE 610.12-1990, "Standard Glossary of Software Engineering Terminology," IEEE Standards Software Engineering: Customer and Terminology Standards **1**, IEEE Press, 1999 .

[7] ISO 9126, "Information technology - software product evaluation - quality characteristics and guidelines for their use." ISO/IEC, 1991 .

[8] Jørgensen, M., *Experience with the accuracy of software maintenance task effort prediction models*, IEEE Trans. Software Engineering **21** (1995), pp. 674–681.

[9] Kazman, R., J. Asundi and M. Klein, *Quantifying the costs and benefits of architectural decisions*, in: *Proc. Int'l Conf. Software Engineering*, 2001, pp. 297–306.

[10] Lehman, M. M., J. F. Ramil, P. Wernick, D. E. Perry and W. M. Turski, *Metrics and laws of software evolution - the nineties view*, in: *Proc. Int'l Symp. Software Metrics* (1997), pp. 20–32.

[11] Li, L. and A. Offutt, *Algorithmic analysis of the impact of changes to object-oriented software*, in: *Proc. Int'l Conf. Software Maintenance* (1996), pp. 171–184.

[12] McCabe, T., *A software complexity measure*, Transactions on Software Engineering **2** (1976), pp. 308–320.

[13] Ramil, J. F. and M. M. Lehman, *Metrics of software evolution as effort predictors - a case study*, in: *Proc. Int'l Conf. Software Maintenance*, 2000, pp. 163–172.

[14] Shaw, M. and D. Garlan, "Software Architecture — Perspectives on an Emerging Discipline," Prentice Hall, 1996.

[15] Sneed, H., *Estimating the costs of software maintenance tasks*, in: *Proc. Int'l Conf. Software Maintenance* (1995), pp. 168–181.

[16] Urquhart, A., *Complexity*, in: L. Floridi, editor, *The Blackwell Guide to Philosophy of Computing and Information* (2004).

[17] Zuse, H., "Software Complexity: Measures and Methods," De Gruyter, 1991.