Measuring software flexibility

A.H. Eden and T. Mens

Abstract: Flexibility has been recognised as a desirable quality of software since the earliest days of software engineering. Classic and contemporary software design literature suggests that particular implementations are more flexible than others, but stops short of suggesting objective criteria for quantifying such claims. To measure software flexibility in precise terms, we introduce the notion of evolution complexity and demonstrate how it can be used to measure and compare the flexibility of (1) programming paradigms (Object-Oriented against Procedural programs), (2) architectural styles (Shared Data, Pipes and Filters, and Abstract Data Type) and (3) design patterns (Visitor and the Abstract Factory). We also demonstrate how evolution complexity can be used to choose the most flexible design policy. We conclude with experimental results corroborating our claims.

1 Introduction

Rapid technological developments pervade every aspect of daily life, having a direct effect on the software we use. Every element of the software's operational environment [1] is in a state of constant flux. Frequent changes in the hardware, operating system, cooperating software and client's expectations are motivated by performance improvements, bug-fixes, security breaches and attempts to assemble synergistically even more sophisticated software systems. Classic and contemporary literature in software design recognise the central role of flexibility in software design and implementation. Structured design, modular design, object-oriented design, software architecture, design patterns and component-based software engineering, among others, seek to maximise flexibility. Textbooks about software design emphasise the flexibility of particular choices, thereby implying the superiority of the design policy they advocate. But despite the progress made since the earliest days of software engineering, from the 'software crisis' [2] through 'software's chronic crisis' [3], evolution (formerly 'maintenance') of industrial software systems has remained unpredictable and notoriously expensive, often exceeding the cost of the development phase [4]. Flexibility has therefore become a central concern in software design and in many related aspects in software engineering research. The purpose of this paper is to contribute to our understanding of this quality and to examine ways in which it can be quantified ('metrics').

We begin by observing two problems in the current notion of 'software flexibility'. The first is the absence of reliable metrics thereof. No formal criteria for flexibility have so far been offered [Note 1] and no metrics for quantifying it are known to us.

The second problem we observe in the current notion of flexibility is that it is misconceived as an absolute quality. Such a misconception is reflected for example in IEEE's definition of software flexibility.

> **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 [5].

We find this surprising because hardly any artefact is 'flexible' in absolute terms. The RAM capacity of a desktop computer, for example, can be expanded only if the hardware and the operating system were specifically designed to accommodate for such changes [Note 2]. The same applies to articles of clothing (such as trousers and skirts), which cannot be expanded in size unless specific provisions were made for this explicit purpose. Other examples can be drawn from consumer appliances to urban architecture, each of which is only flexible towards a particular class of changes. However, claims on the flexibility of particular programming paradigms, architectural styles and design patterns ('design policies') are rarely qualified. For example, in his seminal paper on modular decomposition, Parnas [6] claimed that the Abstract Data Type architecture is 'flexible'. Twenty years later Garlan *et al.* [7] qualified this claim by demonstrating a specific class of changes towards which the same architectural style is not flexible. (In Section 3.1, we corroborate and make precise Garlan *et al.*'s observation.) More recently, Parnas observed that predicting the class of changes is the key to understanding software flexibility [Note 3]:

> To apply this principle ('design for change'), one begins by trying to characterise the changes that are likely to occur over the 'lifetime' of the product. As

IEE Proceedings online no. 20050045

doi:10.1049/ip-sen:20050045

Paper first received 22nd October 2004 and in final revised form 30th March 2006

A.H. Eden is with the Department of Computer Science, University of Essex, Colchester, Essex, CO4 3SQ, UK and is also with the Center For Inquiry, Amherst, New York, USA

T. Mens is with the Software Engineering Lab, Université de Mons-Hainaut, Av. du champ de Mars 6, 7000 Mons, Belgium

Note 1: With the exception of the works discussed in Section 5.
Note 2: And even then, memory can only be extended to the extent to which it was specifically designed for.
Note 3: Or as Parnas puts it, the key to understanding 'software aging'.

> we cannot predict the actual changes, the predictions will be about classes of changes ([8], p. 281).

Similar reservations regarding the flexibility of design patterns were also made by Gamma *et al.* ([9], p. 24, our emphasis):

> 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**.

1.1 Contributions

Computational complexity [10] allows us to measure how the cost of computation grows as a function of the size of the input. The big Oh notation indicates an approximation to the number of steps taken by the algorithm. For example, an algorithm traversing a matrix of size *Columns* × *Rows*, which can also be expressed in pseudocode as follows:

```
procedure traverse(matrix: aMatrix, integer: columns, integer: rows) is
   for (integer c:= 1) until c == columns
      for (integer r:= 1) until r == rows
         traverse(aMatrix[c,r]);
```

has the complexity $\mathcal{O}(Columns \times Rows)$. This means that the number of steps in traversing a matrix of size *Columns* × *Rows* grows proportionately to the product of the number of columns and rows. In our example, this means that traversing a matrix of size 4×5 will take about twice as much as traversing a matrix of size 2×5 [Note 4].

We borrow this notion of measurement for estimating software flexibility. We say that *a* is more flexible than *b* towards a particular evolution step because the number of changes required for *a* is smaller than the number of changes required for *b*. For example, Shaw and Garlan [11] say that the Shared Data architectural style is inflexible towards changes in the data storage format because such a change 'will affect almost all of the modules'. In contrast, the Abstract Data architectural style is flexible towards the same change because it 'can be changed in individual modules without affecting others'. Evolution complexity allows us to establish this claim (Section 3.1), showing that the complexity of evolving Shared Data is linear in the number of modules in the implementation $\mathcal{O}(|SharedData|)$, whereas the complexity of same evolution step in Abstract Data Type is constant $\mathcal{O}(1)$. Thus, the complexity of an evolution step measures how inflexible is the implementation towards a particular class of changes: the less changes are required, the more flexible it is.

In this paper, we formulate the notion of evolution complexity to achieve the following.

1. To provide means for quantifying flexibility.
2. To corroborate and make precise informal claims on the flexibility of particular programming paradigms, architectural styles and design patterns.
3. To provide means for choosing the most flexible design policy.

We define 'evolution step' as the unit of evolution with relation to a particular change in the implementation, and demonstrate the following.

- When a particular evolution step is evidently 'easy', we demonstrate its complexity is fixed (constant complexity).
- When a different evolution step is evidently 'difficult', we demonstrate its complexity grows as a function of the size of the implementation (linear complexity).

1.2 Outline

The rest of this paper is organised as follows. In Section 2, we formulate our terminology and illustrate it with an example. In Section 3, we examine metrics for quantifying the flexibility of four recognised design policies. In Section 4, we describe a small-scale experiment that corroborates our predictions. In Section 5, we discuss related work, including the notion of 'software complexity'. In Section 6, we summarise the conclusions that can be drawn from our discussion.

2 Definitions

In this section, we define the terminology used in our discussion and motivate our definitions with an example. We formulate the term 'evolution step' as a unit of measuring the cost of the evolution process using the notion of 'evolution cost metric'.

We use the term **design policy** with reference to any sensible collection of software design decisions, such as programming paradigms, architectural styles and design patterns. Design policies allow us to expand the scope of our discussion from a particular implementation into a class of such. We also use the terms 'implementation' and 'program' interchangeably with reference to a unit of source code.

2.1 Evolution step

Consider the operational environment [12] of a particular program. It consists of the hardware, the operating system, other programs and other sources of input to the program. We abstract the operational environment as a set of functional and non-functional requirements, which we call the **problem**. Software evolution can be described as the process during which changes occur in the problem, which entail changes in the **implementation**. To distinguish between changes in the problem and changes in the implementation, we refer to the former as **shifts** and the latter as **adjustments**, jointly represented as an 'evolution step'. This terminology is illustrated in Fig. 1.

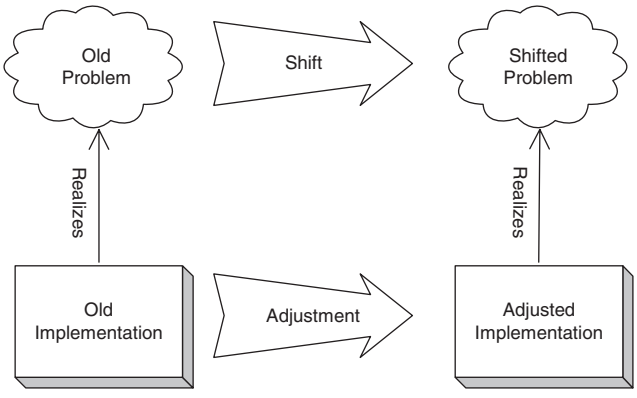

Fig. 1 *Evolution step*

Note 4: It is customary to omit constants in the big Oh notation.

Definition. **Evolution function** is a functional relation $\mathcal{E}$ such that

$$\mathcal{E} : \mathbb{P} \times \mathbb{P} \times \mathbb{I} \rightarrow \mathbb{I}$$

and $\mathcal{E}(p_{old}, p_{shifted}, i_{old}) = i_{adjusted}$ only if i_{old} realizes p_{old} and $i_{adjusted}$ realizes $p_{shifted}$.

Evolution step is a pair $\varepsilon = \langle \langle p_{old}, p_{shifted}, i_{old} \rangle, \mathcal{E}(p_{old}, p_{shifted}, i_{old}) \rangle$

Fig. 2 *Evolution function, evolution step*

Let us represent the set of problems as $\mathbb{P}$, the set of implementations as $\mathbb{I}$ [Note 5]. A step in the process of software evolution can be represented as a mapping of the combination of the old problem $p_{old} \in \mathbb{P}$, the shifted problem $p_{shifted} \in \mathbb{P}$ and the old implementation $i_{old} \in \mathbb{I}$ into the adjusted implementation $i_{adjusted} \in \mathbb{I}$. This mapping can thus be represented as the **evolution function** (Fig. 2), a mathematical function $\mathcal{E}$ which maps each tuple $\langle p_{old}, p_{shifted}, i_{old} \rangle$ to the adjusted implementation $i_{adjusted}$.

2.2 Example: Java's `Collection` interface

Let us illustrate our terminology using the problem of representing data structures and their implementations using the `Collection` interface in the `java.util` package [Note 6]. The design policy that guided the authors of `java.util` is also known as (part of) the Iterator design pattern [9]. These definitions are given to demonstrate how to make precise claims on the flexibility of particular implementations, but the reader may skip the technical detail included in this subsection without adversely affecting the readability of the rest of this paper.

2.2.1 Problem: Provide several concrete data structures and operations thereon. We may unpack a concrete instance of this problem as follows:

- $p_{DS/OP} \triangleq$ Provide a client with data structures

$$DS = \{\texttt{LinkedList}, \texttt{ArrayList}\}$$

and a collection of operations shared by them

$$OP = \{\texttt{add}, \texttt{contains}, \texttt{size}\}$$

To make this example concrete, we use the data structures and their respective operations to represent a movie cast and the audition process.

2.2.2 Implementation: The package `java.util` provides the uniform interface to all data structures, the `Collection` interface. The purpose of this interface is to hide the particulars of each data structure and thereby provide a fixed point of reference for clients of the data structures. Flexibility is supported by allowing the programmer to replace one data structure with another without requiring any adjustments to the remainder of the implementation. Fig. 3 depicts such an implementation, designated *Collection*, which uses the data structures to represent the collection of actors (`movieCast`) as determined by the director. Fig. 4 depicts the UML class diagram of *Collection*.

Note 5: In this context, the constitution of which in terms of Zermelo–Fraenkel's set theory is of no relevant consequence.
Note 6: Accompanying the Java™ class library, version 1.4.2.

2.2.3 Encoding evolution steps: Consider two possible shifts to the problem $p_{DS/OP}$:

- `movieCast` should be represented as a `LinkedList`
- Operation `remove` over `movieCast` is also required (for example because an actor has resigned)

We may encode the new ('shifted') problems as follows:

- $p_{Shifted\text{-}DS} \triangleq$ Use `LinkedList` where `ArrayList` was used;
- $p_{Shifted\text{-}OP} \triangleq$ Add the operation `remove` over the movie's cast.

We may encode the two revised implementations, respectively:

- $Collection_{Adjusted\text{-}DS} \triangleq Collection$ with `LinkedList` where `ArrayList` was used
- $Collection_{Adjusted\text{-}OP} \triangleq Collection$ with method `remove` in `Collection` and interface and in all data structures implementing it

The scenarios following each one of these combinations of shifts and adjustments can be represented as an evolution step as follows.

- Evolve the data structure: $\varepsilon_{DS} \triangleq \langle\langle p_{DS/OP}, p_{Shifted\text{-}DS}, Collection\rangle, Collection_{Adjusted\text{-}DS}\rangle$
- Evolve the operation: $\varepsilon_{OP} \triangleq \langle\langle p_{DS/OP}, p_{Shifted\text{-}OP}, Collection\rangle, Collection_{Adjusted\text{-}OP}\rangle$

These evolution steps can also be expressed in terms of the evolution function (Fig. 2):

- $\mathcal{E}(p_{DS/OP}, p_{Shift\text{-}DS}, Collection) = Collection_{Adjusted\text{-}DS}$
- $\mathcal{E}(p_{DS/OP}, p_{Shift\text{-}OP}, Collection) = Collection_{Adjusted\text{-}OP}$

Detailed encoding of shifts, adjustments and evolution steps can be used to define the class of changes concerned, but is not essential for the understanding of notion evolution complexity.

2.3 Evolution cost metrics

We may summarise at this point, our intuition on the flexibility of the Java `Collection` interface as follows:

- it is flexible towards changing the data structure (ε_{DS}) in the sense that the cost of executing ε_{DS} does not depend on the number of data structures;
- It is inflexible towards adding an operation (ε_{OP}) in the sense that cost of executing ε_{OP} grows with the number of the data structures, $|DS|$.

In other words, the flexibility of `Collection` interface towards each evolution step ε can be quantified in terms

```
public class Director {      // Flexible towards adjustments in movieCast:
  private Collection movieCast = new ArrayList;
  public void audition(Actor candidate) {
    if (CanAct(candidate))
      movieCast.add(candidate);
  }
  ...
}
```

Fig. 3 *Java program illustrating the use of* `Collection`*, which makes class* `Director` *flexible towards adjustments in* `movieCast`

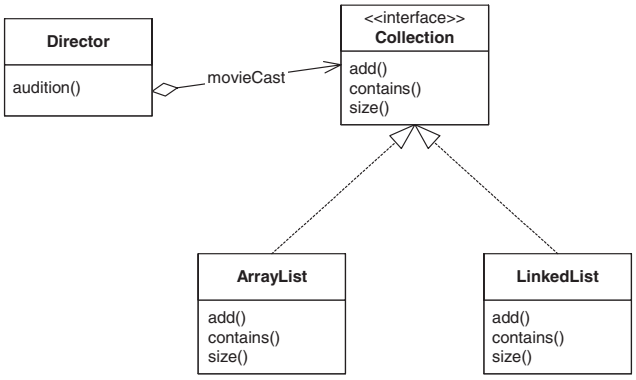

Fig. 4 *UML class diagram of the program in Fig. 3 and related modules*

of the 'cost' of the evolution process. In reality, however, the cost of an evolution process, whether measured in terms of time, equipment, labour and other resource required for executing the evolution step, depends on the size of the programming team, the relevant experience of its members and the software development tools at their disposal, the clarity and currency of the documentation and on a complex range of other cognitive, social and economic factors that are difficult to measure. Given the complexity of this problem we ask, how can we quantify the cost of the evolution process?

A hint as to the solution was given in a talk by Parnas on the subject of Software Aging:

> '... one organises the software so that the items that are most likely to change are 'confined' to a small amount of code, so that if those things do change, only a small amount of code would be affected' ([8], p. 281).

In other words, 'flexibility' (measured in terms of the cost of the evolution process) is directly linked to the amount of code that is affected. Thus, a first approximation to measuring the cost of executing an evolution step ε is given by the evolution cost metric which counts the number of modules that are affected by ε.

Let us also make the simplifying assumption that the costs of adding, removing or changing each modular unit commensurate. Thus, the evolution cost metric we propose is obtained by calculating the number of modules that were added, removed or adjusted [Note 7] as a result of the evolution. This number is obtained by calculating the symmetric set difference between the sets of classes in the old (i_{old}) against the adjusted ($i_{adjusted}$) implementations (Fig. 5).

Note 7: The notion 'a module that was changed' can be fixed in a number of ways, the simplest of which is to count the number of modules that require recompilation after the change.

Let us demonstrate how $\mathcal{C}^1_{Classes}$ can be used to compare the cost of changing the data structure (ε_{DS}) with the cost of adding an operation (ε_{OP},). For this purpose, imagine that a computer program is carrying out the evolution process: such a (meta-)program adjusts the old implementation. The complexity of this (meta-)program can be calculated using the evolution cost metric $\mathcal{C}^1_{Classes}$ (Fig. 5) as follows:

$$\mathcal{O}(\mathcal{C}^1_{Classes}(\varepsilon_{DS})) =$$
$$\mathcal{O}(\#\text{classes affected by replacing } \texttt{ArrayList} \text{ with } \texttt{LinkedList}) =$$
$$\mathcal{O}(1)$$

and

$$\mathcal{O}(\mathcal{C}^1_{Classes}(\varepsilon_{OP})) =$$
$$\mathcal{O}(\#\text{classes affected by adding operation } \texttt{remove} \text{ to } \texttt{Collection}) =$$
$$\mathcal{O}(|DS|)$$

The complexity of each evolution step is summarised in Fig. 6.

This establishes the intuitions expressed in Section 2.3 on the flexibility of the Collection design policy towards each evolution step as follows.

- We say that the `Collection` interface is flexible towards changing the data structure because the complexity of executing ε_{DS} is fixed and independent of the size of the implementation: $\mathcal{O}(\mathcal{C}^1_{Classes}(\varepsilon_{DS})) = 1$. This means, for example, that executing this evolution step in an implementation with 20 data structures cost about as much as executing the same in an implementation with only 10 data structures.
- We also say that the `Collection` interface is inflexible towards adding an operation because the complexity of executing ε_{OP} grows with the number of data structures in the implementation: $\mathcal{O}(\mathcal{C}^1_{Classes}(\varepsilon_{OP})) = |DS|$ (linear complexity). This means, for example, that the cost of executing ε_{OP} in an implementation with 20 data structures is twice as hard as much the cost of as executing the same in an implementation with only 10 data structures.

This result also corroborates the claim that the Collection design policy is not flexible in absolute terms. Instead, we demonstrated that it is flexible towards one class of evolution steps (adding data structures) and inflexible towards another (adding an operation.)

2.4 Evolution complexity

The flexibility of the `Collection` implementation towards each evolution step ε can be quantified by measuring the complexity of the metaprogramming process that executes ε. This approach can be best described

Definition. The **evolution cost metric** $\mathcal{C}^1_{Classes}$ measures the cost of executing evolution step $\varepsilon = \langle \langle p_{old}, p_{shifted}, i_{old} \rangle, i_{adjusted} \rangle$ in terms of the number of classes affected by the adjustment:

$$\mathcal{C}^1_{Classes}(\varepsilon) \triangleq |\Delta\, Classes(i_{old}, i_{adjusted})|$$

where $\Delta\, Classes(i_{old}, i_{adjusted})$ designates the symmetric set difference between the sets of classes in each program, namely: $(Classes(i_{old}) \setminus Classes(i_{shifted})) \cup (Classes(i_{shifted}) \setminus Classes(i_{old}))$.

Fig. 5 $\mathcal{C}^1_{Classes}$

Evolution step / Design policy	Change data structure (ε_{DS})	Add operation (ε_{Op})
`Collection` interface	$\mathcal{O}(1)$	$\mathcal{O}(\|DS\|)$

Fig. 6 $\mathcal{C}^1_{Classes}$*-complexity of evolving the* `Collection` *interface*

Add operation: see also [Note 8]

as an application of the principle of 'software evolution is software too'.

In his award-winning paper 'Software Processes are Software Too' [13], Osterweil suggests to treat human and executable processes uniformly. This is helpful because 'manual and automated processes are both executed, they both address requirements that need to be understood, both benefit from being modelled by a variety of ... models, both must evolve guided by measurement, and so forth' [14]. Evolution complexity can thus be defined as the application of Osterweil's principle to software evolution: The evolution process is conceptualised as a computational process executing a particular 'evolution step' (Fig. 2), specifically as a metaprogram [Note 9] which simulates the respective evolution process. In particular, we can use computational complexity [10] to measure the complexity of the evolution process with relation to a particular 'evolution step' (Fig. 7).

Specifically, this means that the complexity of evolution step $\varepsilon = \langle \langle p_{old}, p_{shifted}, i_{old} \rangle, i_{adjusted} \rangle$ is the complexity of the process which adjusts i_{old} into $i_{adjusted}$ such that $i_{adjusted} = \mathcal{E}(p_{old}, p_{shifted}, i_{old})$.

Computational complexity [10] is concerned with estimating how the 'cost' of the computational processes grows as a function of the size of the problem. Similarly, evolution complexity is concerned with estimating how the cost of the evolution process grows as a function of the size of the implementation. The complexity of a computational process is measured by breaking it into commensurable sub-steps. Similarly, the complexity of the evolution process can be measured by breaking it into commensurable sub-steps. The metric $\mathcal{C}^1_{Classes}$ demonstrates how this number can be calculated: the complexity of adjusting the implementation towards shifts in the data structure is constant $\mathcal{O}(1)$, whereas the complexity of adjusting the implementation towards shifts in the operations is linear in the number of data structures $\mathcal{O}(|DS|)$.

Note 8: The actual number of steps is $|DS| + 1$, whose complexity is equivalent to linear complexity by the abstraction conveyed by the conventional big Oh notation [10].

Note 9: A metaprogram is a program that manipulates other programs. The canonical example is a compiler. Metaprogramming functions are built into many programming languages.

2.4.1 Caveat: We identify the following possible misconceptions of evolution complexity (Fig. 7).

- We do not claim that software evolution can, should or eventually will be fully automated. Rather, we argue that conceptualising the evolution process *as if* it were automated is a metaphor that is useful for the purpose of quantifying the complexity of the evolution process [Note 10].
- *Complexity* is a measure of growth, not an absolute value. Computational complexity does not measure the *actual* time that computational processes require but how it *grows*. Likewise, evolution complexity does not measure the *actual* cost of the evolution processes but how it *grows*.

2.5 More evolution cost metrics

The evolution cost metric $\mathcal{C}^1_{Classes}$ is inadequate in at least three situations.

1. *When the evolution of different modules do not commensurate*. During late phases in the software lifecycle, it may become evident that the cost of evolving one module does not commensurate with the evolution of another. For example, the cost of changing a 'small' class [for example, a class defined in two lines of code (LoC)] does not commensurate with the evolution of a 'large' class (for example, a class defined in 200 LoC).
2. *When modules were not yet implemented*. During even earlier phases of architectural design, the implementation has not been completed yet.
3. *When the programming language does not support classes*. Evolution cost metrics should also be defined for programming languages other than class-based.

We conclude that the metric must accommodate for varying degrees of modular granularity, as well as for varying degrees of information on each module. This leads us to define the generalised evolution cost metric (Fig. 8).

The generalised metric is parameterised by the variables *Modules* and μ.

- *Modules* represent any notion of module that is appropriate for the circumstances, such as class, procedure, method and package. For example, $\mathcal{C}^1_{Packages}$ is a coarse metric counting the steps in the evolution process in terms of number of *packages* in the implementation.
- μ represents any *software complexity metric* that is meaningful with relation to a particular module m. For example, fixing $\mu = LoC$ yields the metric $\mathcal{C}^{LoC}_{Modules}$ which incorporates information about the size of each module,

Note 10: Similar misconceptions of Osterweil's general principle are discussed in the work of Osterweil [14].

Definition. The complexity of the evolution step $\varepsilon = \langle \langle p_{old}, p_{shifted}, i_{old} \rangle, i_{adjusted} \rangle$, also the **evolution complexity** of ε, is the complexity of a process executing ε.

Fig. 7 *Evolution complexity*

Definition. The **generalized evolution cost metric** $\mathcal{C}^{\mu}_{Modules}$ measures the cost of executing evolution step $\varepsilon = \langle \langle p_{old}, p_{shifted}, i_{old} \rangle, i_{adjusted} \rangle$ in terms of the *software complexity* $\mu(m)$ of each module m affected by the adjustment:

$$\mathcal{C}^{\mu}_{Modules}(\varepsilon) \triangleq \sum_{m \in \Delta Modules(i_{old}, i_{adjusted})} \mu(m)$$

where μ is any *software complexity* metric and $\Delta Modules(i_{old}, i_{adjusted})$ designates the symmetric set-difference between the set of modules in i_{old} and the set of modules in $i_{adjusted}$.

Fig. 8 *The generalised evolution cost metric*

whereas fixing $\mu = CC$ (cyclomatic complexity) incorporates information about the control flow.

Naturally, Fig. 8 gives rise to any number of evolution cost metrics. Various such metrics are demonstrated in the rest of this paper. More generally, an evolution cost metric $\mathcal{C}$ is a function which associates an evolution step ε with a non-negative number which attempts to quantify the cost of the process of executing ε. Formally

$$\mathcal{C} : \mathbb{P} \times \mathbb{P} \times \mathbb{I} \times \mathbb{I} \rightarrow \mathbb{R}^{+}$$

2.5.1 Discussion: In proposing the metric $\mathcal{C}^{1}_{Classes}$, we assumed that the costs of (1) adding, (2) removing and (3) changing a module commensurate. How can this assumption be justified? Experience teaches that the cost of evolving a module often exceeds that of developing it in the first place [Note 11]. [4]. It is therefore not unreasonable to assume that removing and changing a module costs about as much as adding a new module.

Alternatively, consider a situation where only adding a module is significant. For example, let us assume that

- the cost of evolution effort is proportional to the module's size, which is determined by the simplistic count of Lines of Code (LoC), and that
- removing and changing a module are inconsequential.

Under such circumstances, the appropriate evolution cost metric is $\mathcal{C}^{Add/Loc}_{Modules}$, which can be defined as follows (Fig. 9).

Let us demonstrate that the metric $\mathcal{C}^{Add/Loc}_{Modules}$ can be obtained by refining the generalised evolution cost metric. We fix the software complexity metric μ to be the metric *Add/LoC*, defined as follows

$$Add/LoC(m) \triangleq \begin{cases} LoC(m) & \text{when } m \text{ is a new module} \\ 0 & \text{otherwise} \end{cases}$$

This demonstrates that $\mathcal{C}^{Add/Loc}_{Modules}$ as formulated in Fig. 9 can be obtained by fixing $\mu(m) = Add/LoC(m)$ in the generalised evolution cost metric.

3 Case studies in evolution complexity

In this section, we use evolution cost metrics to corroborate informal claims on the flexibility of various programming paradigms, architectural styles and design patterns.

Note 11: The reason is usually because of the inherent complexity of large systems, where every change in an existing module can potentially have a 'domino effect', which is precisely the reason for quantifying flexibility.

3.1 Architectural styles

In 1972, Parnas [6] presented the problem of Key Word in Context (KWIC) for the purpose of demonstrating the flexibility of two 'modular decomposition policies'. The problem has become a classic in the software design literature, and in 1993 Garlan and Shaw [11] encoded each modular decomposition policy as an architectural style. In this section, we use evolution complexity to establish and to quantify the informal claims made by Garlan and Shaw [11] on these architectural styles.

3.1.1 Problem: The KWIC problem is defined as follows:

> The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be 'circularly shifted' by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order [6].

The description can be encoded as follows.

- $KWIC \triangleq$ Represent an indexing system with algorithms

$$Alg = \{Input, Shift, Alphabetize, Output\}$$

3.1.2 Implementations: Parnas discusses several design policies that can possibly guide the solution to the KWIC problem, encoded by Garlan and Shaw [11] as the following three architectural styles: Shared Data, Abstract Data Structure and Pipes and Filters [Note 12]. Below we briefly summarise each architectural style. Note that they differ not in the number of modules, which is $|Alg|$ in all implementations, but in the way data and functionality are distributed.

- Shared Data (Fig. 10): a functional decomposition policy yields one module-per-functionality, all of which operate on some shared representation of the data.
- Abstract Data Type (Fig. 11): a policy that conforms to the principles of data abstraction (a.k.a. information hiding), where operations over data are only allowed via an abstract interface.
- Pipes and Filters (Fig. 12): a modular decomposition policy of encapsulating each algorithm in an independent

Note 12: In addition, Garlan and Shaw [11] suggest other architectural styles that may also be used to solve the KWIC problem such as *blackboard architecture*, omitted from discussion for lack of space.

Definition. The **evolution cost metric** $\mathcal{C}^{Add/Loc}_{Modules}$ measures the cost of executing evolution step $\varepsilon = \langle \langle p_{old}, p_{shifted}, i_{old} \rangle, i_{adjusted} \rangle$ as the total sum of the lines of code in each one of the new modules:

$$\mathcal{C}^{Add/Loc}_{Modules}(\varepsilon) \triangleq \sum_{m \in Modules(i_{adjusted}) \setminus Modules(i_{old})} LoC(m)$$

where $Modules(i_{adjusted}) \setminus Modules(i_{old})$ returns the set of modules *added* to the implementation and $LoC(m)$ designates the number of 'lines of code' of module m.

Fig. 9 $\mathcal{C}^{Add/Loc}_{Modules}$

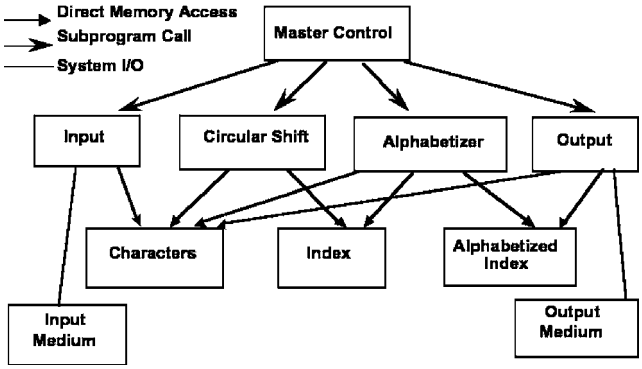

Fig. 10 *Shared Data implementation to the KWIC problem (adapted from Garlan and Shaw [11])*

Rectangles represent the elements of the implementation (modules, as chunks of procedures)

module, or a 'filter', which is a stateless process that accepts an input stream and produces an output stream.

3.1.3 Shifts: Consider the following shifts to the *KWIC* problem:

- Compact the data representation to an efficient format, for example, by packing four letters to a byte.
- Parallelise the processes, thereby allowing simultaneous, distributed processing of multiple documents.

3.1.4 Analysis: Informal claims about the flexibility of each design policy were made by Parnas [6], later refined by Shaw and Garlan [11], summarised in Fig. 13.

Regarding the Shared Data policy they claim:

- 'a change in data storage format will affect almost all of the modules,'
- 'changes in the overall processing algorithm and enhancements to system function are not easily accommodated.'

Regarding the Abstract Data Type policy they claim:

- 'both algorithms and data representations can be changed in individual modules without affecting others;'
- 'the solution is not particularly well-suited to (functional) enhancements.'

Regarding Pipes and Filters policy they claim:

- 'it supports ease of modification (of the algorithm)',
- 'it is virtually impossible to modify the design to support an interactive system (because) decisions about data representation will be wired into the assumptions about the kind of data that is transmitted along the pipes.'

These claims were summarised by Shaw and Garlan [11] using a comparative matrix, depicted in Fig. 13.

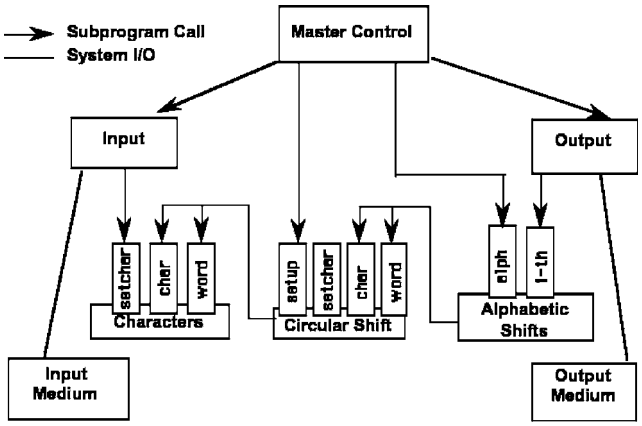

Fig. 11 *Abstract Data Type implementation to the KWIC problem (adapted from Garlan and Shaw [11])*

Horizontal rectangles represent the elements of the implementation (abstract data types) and vertical rectangles represent elements in the ADT's interface

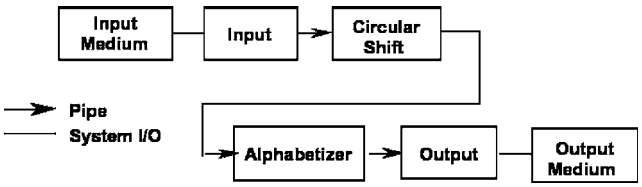

Fig. 12 *Pipes and Filters implementation to the KWIC problem (adapted from Garlan and Shaw [11])*

Rectangles represent the elements of the implementation (filters)

Evolution step / Design policy	Change Data Representation	Enhance functionality
Shared Data	—	—
Abstract Data Type	+	—
Pipes and Filters	—	+

Fig. 13 *Informal claims about the flexibility of three design policies towards shifts in the KWIC problem (adapted from Shaw and Garlan [11])*

Minus symbol stands for 'easy to evolve' and a plus symbol for 'difficult to evolve'

We may use the metric $\mathcal{C}^{1}_{Modules}$ to corroborate and quantify these claims. The results of this analysis are summarised in Fig. 14.

3.2 Programming paradigms

Object-oriented programming (OOP) is hailed, among other reasons, for promoting flexibility. Experienced programmers, however, observe that object-oriented mechanisms

Shift / Design policy	Compact Data Representation	Enhance functionality
Shared Data	$\mathcal{O}(\|Alg\|)$	$\mathcal{O}(\|Alg\|)$
Abstract Data Type	$\mathcal{O}(1)$	$\mathcal{O}(\|Alg\|)$
Pipes and Filters	$\mathcal{O}(\|Alg\|)$	$\mathcal{O}(1)$

Fig. 14 *$\mathcal{C}^1_{Modules}$-complexity of evolving the design policies in Fig. 13, corroborating the claims made by Shaw and Garlan [11]*

such as inheritance and dynamic binding make programs more flexible only towards the particular shifts they specifically were tailored to accommodate. For example, it has been established that gratuitous use of inheritance may lead to the problem of 'fragile base class' [15] and yield highly inflexible programs.

In this subsection, we establish the intuition regarding the flexibility of object-oriented programs expressed above. We formulate the problem of representing a deterministic finite-state automaton (DFSA) and analyse the complexity of evolving object-oriented against procedural implementations towards shifts in this problem.

3.2.1 Problem: Consider the problem of representing the behaviour of a digital clock with three display states: *DisplayHour*, *DisplaySeconds*, *DisplayDate* and two setting states *SetHour*, *SetDate*. The clock accepts input from two buttons b_1 and b_2, which are used to perform a specific action depending on the current state. The clock's behaviour can be modelled as a DFSA, illustrated in Fig. 15.

We may encode this problem as follows.

- *Clock* $\triangleq$ Represent a clock with a set of states

$$States = \{DisplayHour, DisplaySeconds, DisplayDate, SetHour, SetDate\}$$

and buttons

$$Buttons = \{b_1, b_2\}$$

3.2.2 Shifts: Consider two revisions to problem *Clock*:

- Add button b to *Buttons*
- Add state s to *States*

3.2.3 Implementations: Consider the following two implementations of *Clock*:

- *OO* $\triangleq$ Implement *Clock* in OOP policy. The State pattern [9] manifests a classical object-oriented solution to the representation of a clock. Overall, there are $|States|$ classes and $|States| \times |Buttons|$ methods in this implementation, sketched in Fig. 16.
- *PROC* $\triangleq$ Implement *Clock* in a procedural policy, described as follows:

> 'An alternative is to use data values to define internal states and have context operations check the data explicitly. But then we would have look-alike conditional or case statement scattered throughout the context's implementation.' ([9], p. 307)

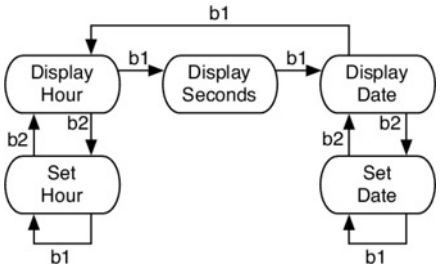

Fig. 15 *DFSA representation of a digital clock*

Overall, there are $|Buttons|$ procedures (ANSI C functions) in such an implementation, each consisting of a 'switch' state with $|States|$ 'cases', sketched in Fig. 17.

3.2.4 Analysis: Informal claims about the cost of each evolution step are made in the work of Gamma *et al.* [9].

Regarding the object-oriented implementation they claim:

- 'new states ... can be added easily',
- 'decentralising the transition logic in this way makes it easy to modify or extend the logic' ([9], pp. 307–308).

Regarding the procedural implementation they claim:

- 'adding a new state ... complicates maintenance.'

The difficulty at this point lies in comparing adjustments to an implementation in Java™ with adjustments to an implementation in ANSI C. The question is, Which commensurable units of modularity can be used to compare such adjustments? As the simplest answer, we define the function *Class/Func(i)*, which counts the number of modules (ANSI C functions or Java classes) in the implementation i:

$$Class/Func(i) \triangleq \begin{cases} \#Functions(i) & \text{When } i \text{ is written in Ansi C} \\ \#Classes(i) & \text{When } i \text{ is written in Java} \end{cases}$$

Fixing the software complexity metric $\mu(m) = 1$ and $Modules(i) = Class/Func(i)$ in the generalised evolution cost metric yields the metric $\mathcal{C}^1_{Class/Func}$. The results of calculating the complexity of each evolution step using $\mathcal{C}^1_{Class/Func}$ are summarised in Fig. 18.

However, it may be (justly) argued that adjusting ANSI C functions and Java™ classes do not commensurate, and that a more refined approach may be in place. An alternative (and possibly more sophisticated) software complexity metric, such as the CC metric [16], may provide a more faithful representation of the cost of evolving of different modules. Fixing $\mu(m) = CC(m)$ and $Modules(m) = Class/Func(m)$ in the generalised evolution cost metric yields the metric $\mathcal{C}^{CC}_{Class/Func}$, formulated as depicted in Fig. 19.

Analysing the complexity of the same evolution steps using the metric $\mathcal{C}^{CC}_{Class/Func}$ yields slightly different results, summarised in Fig. 20.

3.2.5 Conclusions: We may conclude the following from the results obtained from using different evolution cost metrics.

1. Neither programming paradigm is flexible in absolute terms, irrespective of the metric chosen.

```
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 */}
}
```

Fig. 16 *Object-oriented implementation to the clock problem (in Java™)*

```
enum states = {DisplayHour, DisplaySecond, DisplayDate, SetHour, SetDate};

struct clock_r {
  states state;            // Current state
} aClock;

void b1(aClock) {          // button 1 pressed
  switch (aClock.state) {
    case DisplayHour:      /*...*/;
    case DisplaySecond:    /*...*/;
    case DisplayDate:      /*...*/;
    case SetHour:          /*...*/;
    case SetDate:          /*...*/; }
}

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

Fig. 17 *Procedural implementation of the Clock problem (in ANSI C)*

2. The question *Which programming paradigm is more flexible?* can be reduced to the question *Which shifts to the problem are most likely to occur?*
3. The accuracy of an evolution cost metric varies with the amount of information available about the implementation.

More generally, we also conclude the following.

4. Metrics of different granularity levels are useful during both early and late phases in the software lifecycle. A coarse metric (such as $\mathcal{C}^{1}_{Class/Func}$) is useful during the design process, namely before the implementation is complete, whereas a refined metric (such as $\mathcal{C}^{CC}_{Class/Func}$) is useful in later stage, namely once the implementation phase has been completed.

A third solution to the clock problem has not been discussed. States and buttons can be represented in a data structure rather than being hard-coded as classes or functions. This solution is more flexible because the complexity of either evolution step is constant. This solution, however, does not affect the conclusions drawn above.

Evolution step / Design policy	Add letter	Add state
O-O programming	$\mathcal{O}(\lvert States \rvert)$	$\mathcal{O}(1)$
Procedural programming	$\mathcal{O}(1)$	$\mathcal{O}(\lvert Buttons \rvert)$

Fig. 18 *$\mathcal{C}^{1}_{Class/Func}$-complexity of evolving object-oriented against procedural implementations towards shifts in the clock problem*

3.3 Design patterns

In the work of Mens and Eden [17], we analysed the complexity of evolving two design patterns. In this section, we summarise the conclusions drawn from our analysis.

3.3.1 Visitor: The patterns catalogue [9] discusses the problem of representing abstract syntax trees and operations thereon. It is argued that, ideally, the most flexible implementation is one written in a programming language that supports *double dispatch* [18]. However, the programming languages at the focus of the authors (C++ and Smalltalk [9], but also neither in most other O–O programming languages such as Java and C#) do not support such a mechanism, which motivates the *Visitor* pattern as a design policy. The Visitor pattern consists of two class hierarchies, representing (1) the set of elements in abstract syntax (e.g. *constant*, *variable*, *addition expression*) and (2) the set of operations thereof (e.g. 'print this tree'). Regarding the Visitor design policy they claim:

Definition. The **evolution cost metric** $\mathcal{C}^{CC}_{Class/Func}$ measures the cost of evolution step $\varepsilon = \langle \langle p_{old}, p_{shifted}, i_{old} \rangle, i_{adjusted} \rangle$ in terms of the total sum of the cyclomatic complexity of the modules affected by the adjustment:

$$\mathcal{C}^{CC}_{Class/Func}(\varepsilon) \triangleq \sum_{m \in \Delta Class/Func(i_{adjusted}, i_{old})} CC(m)$$

where $CC(m)$ of module m is its Cyclomatic Complexity [18], which is defined as follows:

$$CC(m) \triangleq \begin{cases} \#\text{independent paths in the flow graph of } m & \text{When } m \text{ is an Ansi C function} \\ \text{Total sum of \#indendent paths in the flow graph of all methods of } m & \text{When } m \text{ is a Java class} \end{cases}$$

Fig. 19 $\mathcal{C}^{CC}_{Class/Func}$

Design policy \ Evolution step	Add letter	Add state
O-O programming	$\mathcal{O}(\lvert States\rvert \times \lvert Buttons\rvert)$	$\mathcal{O}(\lvert Buttons\rvert)$
Procedural programming	$\mathcal{O}(\lvert States\rvert)$	$\mathcal{O}(\lvert Buttons\rvert \times \lvert States\rvert)$

Fig. 20 $\mathcal{C}^{CC}_{Class/Func}$*-complexity of evolving an object-oriented against procedural implementation of clock problem*

Design policy \ Evolution step	Add operation	Add element
Visitor pattern	$\mathcal{O}(1)$	$\mathcal{O}(\lvert Operations\rvert)$

Fig. 21 $\mathcal{C}^{1}_{Classes}$*-complexity of evolving the Visitor pattern*
Operations stands for the number of 'visit' operations

- '(it) makes adding new operations easy';
- 'Adding new concrete element classes is hard'.

We may use $\mathcal{C}^{1}_{Classes}$ (Fig. 5) to corroborate these claims and to make them precise. The results of this analysis are summarised in Fig. 21.

3.3.2 Abstract factory: The same catalogue [9] also discusses the problem of providing 'an interface for creating families of related or dependent objects without specifying their concrete classes.' The actual object that need be created depends on the global context ('current configuration'). For example, when a Graphical User Interface client seeks to create a new 'dialogue box', considerations of flexibility determine that it must remain independent from the question, Which windowing systems have been implemented and from the particulars of how dialogue boxes are generated in windowing system.

The authors discuss two design policies solving this problem: The first (the 'anti-pattern') uses a 'switch' statement – multiple conditional branching – to determine which dialogue box to create. As an alternative which conforms to the general spirit of the OOP paradigm, the Abstract Factory pattern dictates that subtyping and dynamic binding should be used to hide the concrete class of the object created behind a uniform interface. In the work of Mens and Eden [17], we use the evolution cost metric $\mathcal{C}^{1}_{Classes}$ to analyse the flexibility of each design policy, the results of which are summarised in Fig. 22.

These results further corroborate the more general claims we made: that flexibility depends on which class of shifts that are most likely to occur. This suggests that a software architect must weigh carefully the question, Which shifts exactly are most likely to occur before he/she may choose the appropriate design policy.

4 Empirical support

A metric is validated when it can be shown to measure what it is supposed to measure. The most obvious test to the evolution cost metrics is to investigate how the actual costs of a particular evolution step grow in a controlled environment. We describe the consolidated results obtained from conducting several small-scale experiments to this extent at the University of Essex and at the Université de Mons-Hainaut. These experiments [19, 20] corroborate the predictions made in Section 3.2 regarding the complexity of evolving object-oriented and procedural implementations towards shifts in the clock problem.

In the first part of all experiments, subjects were presented with an implementation of a state machine (p_{Clock}) with three states ($|States| = 3$) and one button ($|Buttons| = 1$). Next, the subjects were asked to evolve the implementation in a series of tasks, and to measure the time required to complete each task.

- In task 2, subjects added a second button to *Buttons*
- In task 3, subjects added three more states to *States*
- In task 4, subjects added a third button to *Buttons*

Specifically, the experiment was designed to test the predictions made (Section 3.2) using the evolution cost metrics $\mathcal{C}^{1}_{Class/Func}$ and $\mathcal{C}^{CC}_{Class/Func}$ regarding the cost of evolving an object-oriented (i.e. along the list of the program in Fig. 16) against a procedural (Fig. 17) implementation. Fig. 23 contrasts the theoretical predictions with the empirical results obtained in the two experiment types.

Fig. 23 should be interpreted as follows, depending on the implementation to be evolved.

- Both metrics $\mathcal{C}^{1}_{Class/Func}$ and $\mathcal{C}^{CC}_{Class/Func}$ predict that the complexity of adding a letter to the alphabet in an object-oriented implementation grows proportionally with the number of states |*States*|, namely that task 4 (adding a letter to an O–O implementation of a DFSA with six states) will cost about twice as much as task 2 (adding a letter to an O–O implementation of a DFSA with three states). In practice, our results show that task 4 took 2.5 times as much as task 2 (calculated as the median of the ratios between the tasks). We believe that these initial results, collected from seven subjects (standard deviation: 1.4), strongly corroborate both metrics as predictors of flexibility.
- The metric $\mathcal{C}^{1}_{Class/Func}$ predicts that the complexity of adding a letter to the procedural implementation is fixed, namely that task 4 (adding a letter to a procedural implementation of a DFSA with six states) will cost about as much as task 2 (adding a letter to a procedural implementation of a DFSA with three states). The metric $\mathcal{C}^{CC}_{Class/Func}$ predicts that the complexity of the task of adding a letter to the

Evolution step / Design policy	Add configuration	Add state
Abstract Factory	$\mathcal{O}(\lvert Products \rvert)$	$\mathcal{O}(\lvert Configurations \rvert + \lvert Products \rvert)$
'Switch'	$\mathcal{O}(\lvert Clients \rvert)$	$\mathcal{O}(\lvert Clients \rvert)$

Fig. 22 *$\mathcal{C}^{1}_{Classes}$-complexity of evolving the Abstract Factory against Switch implementations towards shifts in the Object Creation problem*

alphabet in a procedural implementation grows proportionally with the number of states $|States|$, namely that task 4 will cost about twice as much as task 2. In practice, our results show that task 4 took 1.5 times as much as task 2 (calculated as the median of the ratios between the tasks). These initial results, collected from seven subjects (standard deviation: 0.47) indicate that the cost grew less than predicted by $\mathcal{C}^{CC}_{Class/Func}$ and more than predicted by $\mathcal{C}^{1}_{Class/Func}$.

Statistically sounder results should be obtained to explore the possibility that a different evolution cost metric may be in place.

5 Related work

Switching between design policies is an expensive and time-consuming activity, but tool support for automating it has the potential of reducing its complexity. For example Kerievsky [21] offers a catalogue of well-defined 'refactorings', such as switching for example from a procedural 'switch' policy to an object-oriented (e.g. the State pattern) design policy. Such catalogues of well-defined transformations constitute a first step towards automating changes in the design policy, a process which may greatly reduce the cost of executing the corresponding class of evolution steps.

5.1 Software complexity

Curtis [22] suggests that 'In the maintenance phase [software] complexity determines ... how much effort will be required to modify program modules to incorporate specific changes.' Zuse [23] counts over 200 metrics for software complexity in the literature. Three prominent examples are the following.

- LoC counts the number of lines in the source code of the implementation.
- McCabe's cyclomatic complexity [16] measures the number of nodes in the flow graph of the program (demonstrated in Section 3.2).
- Halstead's Volume [24] metric is given by the equation $(N_1 + N_2) \times lg_2(n_1 + n_2)$, where n_1 is the number of distinct operators, n_2 is the number of distinct operands, N_1 is the total number of operators and N_2 is the total number of operands in the respective module.

Each one of these software complexity measures can replace the parameter μ in the general metric for evolution complexity (Fig. 8).

- In Sections 2.5, we demonstrated the result of fixing $\mu = \text{LoC}$ in the generalised metric.
- In Section 3.2, we demonstrated the result of fixing $\mu = \text{CC}$ in the generalised metric.
- The result of fixing $\mu = Volume$ (Halstead's Volume metric) in the generalised metric yields the evolution cost metric $\mathcal{C}^{Volume}_{Modules}$, defined as follows

$$\mathcal{C}^{Volume}_{Modules}(\varepsilon) = \sum_{m \in \Delta Modules(i_{old}, i_{adjusted})} \times (N_1(m) + N_2(m)) \times lg_2(n_1(m) + n_2(m))$$

where $N_1(m)$ is the total number of operators in module m, $N_2(m)$ is the total number of operands in m, $n_1(m)$ is the number of distinct operators in m and $n_2(m)$ is the number of distinct operands in m.

Unfortunately, neither simple nor sophisticated software complexity metrics have been proved accurate as indicators of productivity, comprehensibility or maintainability [23]. This suggests that the accuracy of an evolution cost metric is only limited by the accuracy of the software complexity. We hope that the future will bring more accurate software complexity metrics.

5.2 Metrics for software evolution

Quantifying the actual cost of the software evolution remains a relatively unexplored problem. Jørgensen [25] used several models to predict the effort that randomly selected software maintenance tasks require. The size of individual maintenance tasks was measured in LOC. Sneed [26] proposed a number of ways to extend existing cost estimation methods to the estimation of maintenance costs. Ramil *et al.* [27] provided and validated six different models that predict software evolution effort as a function of software evolution metrics. None of these approaches however suggests an obvious way in which it is tied to the notion of software flexibility.

	Prediction by $\mathcal{C}^{1}_{Class/Func}$	Prediction by $\mathcal{C}^{CC}_{Class/Func}$	Median measured	Standard deviation
Object-oriented implementation	2	2	2.50	1.40
Procedural implementation	1	2	1.50	0.47

Fig. 23 *Theory against practice: summary comparing the predictions made against the empirical results obtained in experiments regarding the cost of evolving object-oriented and procedural implementations*

5.3 Metrics for software flexibility

We are unaware of alternative approaches for quantifying software flexibility, nor of any formal criteria for establishing this quality. It has been suggested that a more accurate way to measure flexibility relies on algorithms or measures that compute the impact of changes [28]. For example, Chaumun *et al.* [29] 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 finer granularity of recognition of types of software evolution activities. Such an attempt to make an objective classification of evolution activities was carried out in the work of Chapin *et al.* [30].

6 Conclusions

We proposed evolution cost metrics and suggested that flexibility can be measured as the complexity of executing particular evolution steps. We studied the complexity of evolving implementations of five recognised programming paradigms, architectural styles and design patterns, and demonstrated that evolution complexity corroborates intuitions and established observations on the flexibility of these design policies. As stipulated, we also demonstrated the *flexibility* of a particular implementation is relative to the change, and that while a particular implementation is flexible towards one class of changes, it is also inflexible towards another. In particular, the benefits from the measurements proposed are the following.

1. Evolution complexity can be used to corroborate and quantify informal claims on the flexibility of particular programming paradigms architectural styles and design patterns.
2. Evolution complexity can be used to measure flexibility with varying degrees of accuracy.
3. Evolution complexity can be used to choose the most flexible design policy, given the class of the most likely shifts to the problem.

6.1 Future directions

The small-scale experiment described in Section 4 should be expanded in all aspects, e.g. testing predictions made with respect to other problems, as well as for the purpose of establishing statistically sounder results, using larger sample groups and larger implementations. Of particular interest is to examine the validity of coarse (such as $\mathcal{C}^{1}_{Classes}$) against refined (such as $\mathcal{C}^{CC}_{Class/Func}$) evolution cost metrics.

EC can be used to analyse the flexibility of design policies beyond the examples given here. For example, it can be used to throw light on the claims made on the recent introduction of generics to Java and in comparing the flexibility of particular technologies (e.g., CORBA against .NET). In particular, EC can be used in supporting the decision whether to apply a particular refactoring [31], possibly by incorporating a range of evolution cost metrics into integrated development environments which support refactoring, such as IBM Eclipse and Borland JBuilder.

An investigation of the relation between EC and actual cost of the evolution process is also of interest, albeit more of socio-economic nature than from the software engineering perspective.

Note, however, that given the similarity between the concepts, EC is no more dependent on empirical validation than computational complexity. So defined, it remains to be examined whether polynomial, exponential and logarithmic complexity functions are meaningful in the context of software evolution.

6.2 EC tradeoffs

Studying the flexibility of different programming paradigms towards shifts in the DFSA problem (Section 3.2) suggests that a trade-off may exist between EC and computational complexity, a relation that is analogous to the trade-off between space and time (computational) complexities. Although evidence at this stage is anecdotal, it remains to be examined whether decreased EC (increased flexibility) leads to time and/or space penalties. Trade-off may also exist between the development effort (early design) and the evolution effort, in the spirit of the adage 'weeks of programming can save you hours of planning'. But metrics for quantifying the complexity of the software development process are yet to be proposed.

7 Acknowledgments

This research has been carried out in the context of the scientific network RELEASE financed by the European Science Foundation (ESF). We wish to thank Nguyen Long, Jon Nicholson, Peter Ebraert, Khalid Allem, Valérie Fiolet, Arista Benoit, Chevalier Jérémie, Michot Arnaudm, Pistone Bruno, Bastien Grégory, Hoste Michael, Jonckers Jérome, Lavry Aline, Salame Maroun and Scolas Laurent for taking part in our experiment. We also wish to thank Jeff Reynolds and Mehdi Jazayeri for their comments and suggestions, and Naomi Draaijer for her support, and Mary J. Anna for her inspiration.

8 References

1 Lehman, M.: 'Laws of software evolution revisited', *Lect. Notes Comput. Sci.*, **1149** (Proc. 5th European Workshop on Software Process Technology), pp. 108–124
2 Naur, P., and Randell, B.: 'Software engineering: report of a conference sponsored by the NATO science committee', 7–11 October 1968. Scientific Affairs Division, NATO, Garmisch, Brussels, Germany, 1969
3 Gibbs, W.W.: 'Software's chronic crisis', *Sci. Am.*, 1994, **271**, (3), pp. 86–95
4 Boehm, B.: 'Software engineering economics' (Prentice-Hall, Englewood Cliffs, 1981)
5 IEEE 'Standard glossary of software engineering terminology 610.12-1990' (IEEE Press, Los Alamitos, 1999)
6 Parnas, D.L.: 'On the criteria to be used in decomposing systems into modules', *Commun. ACM*, 1972, **15**, (12), pp. 1053–1058
7 Garlan, D., Kaiser, G.E., and Notkin, D.: 'Using tool abstraction to compose systems', *Computer*, 1992, **25**, (6), pp. 30–38
8 Parnas, D.L.: 'Software Aging'. Proc. Int. Conf. on Software Engineering – ICSE, May 1994 (IEEE Computer Society Press, Los Alamitos), pp. 279–287
9 Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: 'Design patterns: elements of reusable object-oriented software' (Addison-Wesley, Reading, 1995)
10 Urquhart, A.: 'Complexity', in Floridi, L. (Ed.): 'The Blackwell guide to philosophy of computing information' (Blackwell, Oxford, 2004)
11 Shaw, M., and Garlan, D.: 'Software architecture—perspectives on an emerging discipline' (Prentice-Hall, Upper Saddle River, 1996)
12 Lehman, M.M., Ramil, J.F., Wernick, P., Perry, D.E., and Turski, W.M.: 'Metrics and laws of software evolution – the nineties view'. Proc. Int. Symp. on Software Metrics, 5–7 November 1997 Albuquerque, NM, (IEEE Computer Society Press, Los Alamitos), pp. 20–32

13 Osterweil, L.: 'Software processes are software too'. Proc. 9th Int. Conf. on Software Engineering – ICSE, (IEEE Computer Society, Los Alamitos), pp. 2–13
14 Osterweil, L.: 'Software processes are software too revisited'. Proc. 19th Int. Conf. on Software Engineering – ICSE, May 1997, (IEEE Computer Society, Los Alamitos), pp. 540–548
15 Taivalsaari, A.: 'On the notion of inheritance', *ACM Comput. Surveys*, 1996, **28**, (3), pp. 438–479
16 McCabe, T.: 'A software complexity measure', *IEEE Trans. Soft. Eng.*, 1976, **2**, pp. 308–320
17 Mens, T., and Eden, A.H.: 'On the evolution complexity of design patterns', *Electron. Lecture Notes Comput. Sci.*, 2004, **127**, (3), pp. 147–163
18 Craig, I.: 'The interpretation of object-oriented programming languages' (Springer-Verlag, New York, 2000)
19 Eden, A.H.: 'An experiment in evolution complexity: instructions to subjects', Technical Report CSM-431 Department of Computer Science, University of Essex, ISSN 1744-8050, 2005
20 Mens, T., and Eden, A.H.: 'Revised experiment in evolution complexity: instructions to subjects'. Technical Report CSM-439 Department of Computer Science, University of Essex, ISSN 1744-8050, 2005
21 Kerievsky, J.: 'Refactoring to patterns' (Addison Wesley Professional, Reading, 2004)
22 Curtis, B.: 'In search of software complexity'. Proc. Workshop on Qualitative Software Models for Reliability, Complexity and Cost, October 1979, pp. 95–106
23 Zuse, H.: 'Software complexity' (Walter de Gruyter, Berlin, 1998)
24 Halstead, M.H.: 'Elements of software science' (Elsevier, New York, 1977)
25 Jørgensen, M.: 'Experience with the accuracy of software maintenance task effort prediction models', *IEEE Trans. Softw. Eng.*, 1995, **21**, (8), pp. 674–681
26 Sneed, H.: 'Estimating the costs of software maintenance tasks'. Proc. Int. Conf. on Software Maintenance, 1995, (IEEE Computer Society Press, Los Alamitos), pp. 168–181
27 Ramil, J.F., and Lehman, M.M.: 'Metrics of software evolution as effort predictors – a case study'. Proc. Int. Conf. on Software Maintenance, October 2000, (IEEE Computer Society Press, Los Alamitos), pp. 163–172
28 Li, L., and Offutt, A.: 'Algorithmic analysis of the impact of changes to object-oriented software'. Proc. Int. Conf. on Software Maintenance – ICSM, 1996, (IEEE Computer Society Press, Los Alamitos), pp. 171–184
29 Chaumun, M.A., Kabaili, H., Keller, R.K., and Lustman, F.: 'A change impact model for changeability assessment in object-oriented software systems', *Sci. Comput. Program.*, 2002, **45**, (2–3), pp. 155–174
30 Chapin, N., Hale, J., Khan, K., Ramil, J., and Than, W.G.: 'Types of software evolution and software maintenance', *J. Softw. Maint. Evol.*, 2001, **13**, pp. 3–30
31 Fowler, M.: 'Refactoring: improving the design of existing code' (Addison-Wesley, 2003)