EVOLUTION COMPLEXITY:
INVESTIGATIONS INTO SOFTWARE FLEXIBILITY

Technical report CSM-424, ISSN 1744-8050 (Aug. 2005 revision)
Department of Computer Science, University of Essex

Amnon H. Eden ([1])
Department of Computer Science
University of Essex
and
Center For Inquiry

Tom Mens ([2])
Software Engineering Lab
Université de Mons-Hainaut

Abstract. *Flexibility* has been hailed as a desirable quality since the earliest days of software engineering. Classic and modern literature suggest that particular programming paradigms, architectural styles and design patterns are more "flexible" than others but stop short of suggesting objective criteria for measuring such claims.

We suggest that *flexibility* can be measured by applying notions of measurement from computational complexity to the software evolution process. We define *evolution complexity* (EC) metrics, and demonstrate that—

(a) EC can be used to establish informal claims on software flexibility;
(b) EC can be constant $\mathcal{O}(1)$ or linear $\mathcal{O}(n)$ in the size of the change;
(c) EC can be used to choose the most flexible software design policy.

We describe a small-scale experiment designed to test these claims.

Keywords: Software evolution, flexibility, science of software design.

Related terms: Computational complexity, software complexity, design patterns, software architecture.

([1]) Email: eden@essex.ac.uk, postal address: Colchester, Essex CO4 3SQ, United Kingdom, phone +44 (1206) 872677, fax +44 (1206) 872788

([2]) Email: tom.mens@umh.ac.be, postal address: Av. du champ de Mars 6, 7000 Mons, Belgium, phone +32 (65) 373453, fax +32 (65) 373459

1 Introduction

Ever since the earliest days of software engineering, from the "software crisis" [19] through "software's chronic crisis" [10], evolution (or "maintenance") of industrial software systems has remained notoriously expensive, the cost of which often exceeds that of the development phase [1]. Classic and contemporary software design literature argue that "flexibility" is a major concern in determining the maintainability of software. Textbooks about software design emphasize the flexibility of particular choices, thereby implying their superiority. In particular, object-oriented programming, architectural styles, and design patterns are presented as more "flexible" than their alternatives.

We observe two problems with the current notion of *software flexibility*. The first problem is the absence of reliable metrics thereof. No formal criteria for flexibility were offered ([3]), 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. For example, such misconception is reflected 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* [12]

We find this surprising because hardly any artefact is "flexible" in absolute terms. The RAM capacity of a notebook or a desktop computer, for example, can be expanded only if the hardware and the operating system were specifically designed to accommodate for such changes (and even then only to the extent to which it was specifically designed for). 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 purpose. Other examples can be drawn from consumer appliances to urban architecture, each of which is only flexible towards a particular class of changes. ([4])

The same rule applies to software, as we will demonstrate below, and yet 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

([3]) With the exception of the works discussed in Section 5.

([4]) The same rule also applies to biological species. Every organism can only adapt to a particular class of changes in its environment but is sensitive to others. For example birds can adapt to changes in the amount of sunlight but are sensitive to changes in the temperature, whereas the reverse is true for many botanical genus. Some bacteria can tolerate changes in the ph value (acidity) whereas others (*extremophiles*) are resistant to extreme temperatures.

modular decomposition, Parnas [22] claimed that the Abstract Data Type architecture is "flexible". Twenty years later Garlan, Kaiser and Notkin [9] 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.

Such reservations regarding the flexibility of design patterns were also made by Gamma et. al:

> *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***. ([8], p. 24, our emphasis)

To summarize, we argue that "flexibility" is a quality that is relative the class of changes; suggesting that a particular software is flexible (or not) in absolute terms is misleading. We seek to remedy both these problems in this paper.

Contributions

The contributions we seek can be summarized as follows:

(a) To corroborate and make precise informal claims on the flexibility of particular programming paradigms, architectural styles and design patterns.
(b) To provide means for measuring flexibility with varying degrees of accuracy.
(c) To provide means for choosing the most flexible design policy.

Outline

The remainder of this paper is organized as follows: In Section 2, we formulate our terminology and illustrate it with an example. In Section 3, we discuss the evolution complexity (EC) of four problems, illustrating how EC metrics can be used to establish informal claims on software flexibility. In Section 4 we describe a small-scale experiment that corroborates the prediction made in the preceding section. In Section 5 we discuss related work, including the notion of software complexity. In Section 6 we summarize the conclusions that can be drawn from our discussion. We conclude in Section 7 with a discussion of future directions and of the possible tradeoffs between EC and other concepts.

2 Definitions

In this section, we define the terminology used in our discussion and motivate our definitions with an example.

We use the term *design policy* with reference to any sensible collection of design decisions pertaining to a particular implementation, such as programming paradigms, architectural styles and design patterns. We generalize the implementations we discuss in terms of design policies, thereby expanding the scope of our discussion. We also use the terms *implementation* and *program* interchangeably with reference to a unit of source code that is used as the ultimate specification of the software system.

2.1 Co–evolution

Software evolves as the result of a host of possible changes in its operational environment [15], such as changes in the client's expectations and changes in the hardware or other software. In software engineering terms we also say that implementations are *adjusted* to changes in the definition of the *problem*, namely the functional and/or non-functional requirements. To facilitate our definitions, we unpack each problem as a set of elements (e.g., states or variables). This approach will enable us to define evolution complexity as a function of the change in the problem.

We focus on the correlation between changes in the problem and the corresponding changes in the implementation. To distinguish between these two kinds of changes, we refer to the former as *shifts* and to the latter as *adjustments*, hence the term *co-evolution*, which 'packages' the two. Co-evolution and related terminology is illustrated in Figure 1. The glossary in Table 1 summarizes these terms.

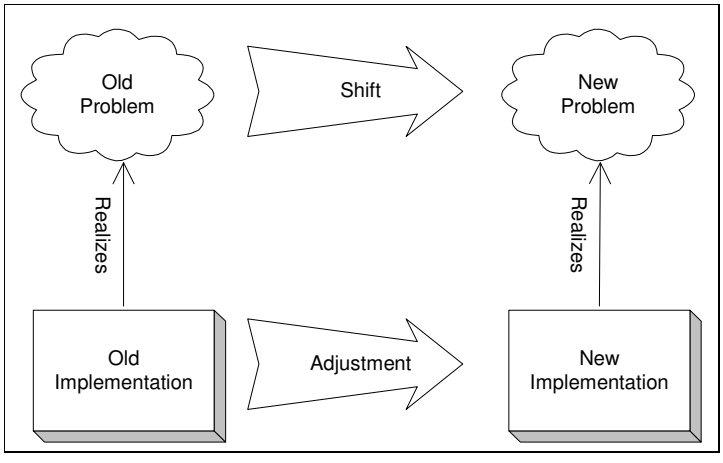

Figure 1. A co-evolution step

Table 1. Glossary of terms (illustrated in Figure 1)

Problem. A well-defined description of the program's observable behaviour ("functional requirements") or organization ("non-functional requirements").

Implementation. The subject of the evolution process; a program (unit of source code) which realizes a particular *problem*.

Shift. A specific change in a given *problem*, encoded as a pair $\sigma = \langle p_{old}, p_{new} \rangle$ where p_{old} designates the *problem* before the shift and p_{new} designates the revised *problem*.

Adjustment. A specific change in a given *implementation*, encoded as a pair $\alpha = \langle i_{old}, i_{new} \rangle$ where i_{old} designates the *implementation* before the adjustment and i_{new} designates the revised *implementation*.

Co-evolution step. A pair $\varepsilon = \langle \sigma, \alpha \rangle$ consisting of a *shift* $\sigma = \langle p_{old}, p_{new} \rangle$ and an *adjustment* $\alpha = \langle i_{old}, i_{new} \rangle$, such that i_{old} realizes p_{old} and i_{new} realizes p_{new}.

2.2 Example: Java's `Collection` interface

Let us illustrate our terminology using the problem of data structures and its implementations using the Collection interface in the `java.util` package (accompanying the Java™ class library, version 1.4.2). The design policy that guided the authors of `java.util` is also known as (part of) the Iterator design pattern [8].

Problem

Provide several concrete data structures and operations thereon. We may unpack an instance of this problem as two sets such as the following:

- $p_{DS/Iter} \triangleq$ 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.

Implementation

Clients of Java's data structures are made flexible by means of the `Collection` interface. The interface hides the particulars of each data structure and exposes only the common operations thereon. Figure 2 illustrates how this policy was realized in the `java.util` class library. A Java program demonstrating how the `Collection` interface can be used for the purpose of representing the audition process is given in Table 2. We encode this implementation as $i_{Java.util}$.

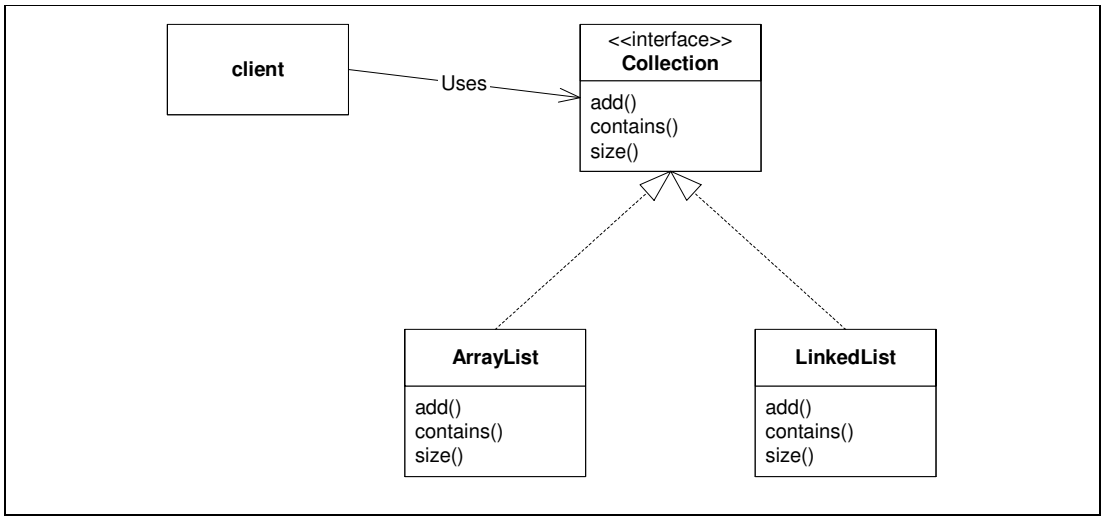

Figure 2. Data structures and their interface in `java.util`.

Table 2. Java program illustrating the use of `Collection`. Class `Director` is resilient to (changes in) the choice of data structure representing the cast because it communicates therewith solely via the interface.

```
public class Director {
  public void audition(Collection movieCast, Actor candidate) {
     if (! movieCast.contains(candidate) && CanAct(candidate))
       movieCast.add(candidate);
  }
  ...
}

public class Movie {
  private ArrayList theCast;  // Data structure chosen here
  ...
}
```

The encoding of evolution steps

The following detailed encoding of the *shift*, *adjustments* and *co-evolution steps* is only necessary for formulating specific units ("steps") in the evolution process. It can be skipped without reducing the readability of much of this paper.

Consider two ways in which this example can evolve:

- Change the implementation of the movie's "cast" from `ArrayList` into `LinkedList`
- Use operation $remove$ from the movie's "cast" (for example because an actor has resigned)

We may encode the two revised *problems* as follows, respectively:

- $p_{\overline{DS}/Op} \triangleq$ Use `LinkedList` where `ArrayList` was used before
- $p_{DS/\overline{Op}} \triangleq$ Include also the operation $remove$

We may encode each *shift* as a pair of *problems* (before and after the change) as follows:

- $\sigma_{DS} \triangleq \langle p_{DS/Op},\ p_{\overline{DS}/Op} \rangle$ (change data structure)
- $\sigma_{Op} \triangleq \langle p_{DS/Op},\ p_{DS/\overline{Op}} \rangle$ (add $remove$ operation)

We may encode the two revised *implementations* respectively:

- $i_{\overline{DS}} \triangleq i_{Java.util}$ with `LinkedList` where `ArrayList` was used
- $i_{\overline{Op}} \triangleq i_{Java.util}$ with method `remove` in `Collection` and in all data structures

We may also encode the respective *adjustments* as pairs of *implementations* as follows:

- $\alpha_{DS} \triangleq \langle i_{Java.util}, i_{\overline{DS}} \rangle$
- $\alpha_{Op} \triangleq \langle i_{Java.util}, i_{\overline{Op}} \rangle$

In conclusion, we may define two *co-evolution steps*, each representing a pair of changes (a *shift* and an *adjustment*) as follows:

- $\varepsilon_{DS} \triangleq \langle \sigma_{DS}, \alpha_{DS} \rangle$
- $\varepsilon_{Op} \triangleq \langle \sigma_{Op}, \alpha_{Op} \rangle$

We may summarize at this point our intuition regarding the flexibility of the example described above as follows:

- It is "flexible" towards changing the data structure (ε_{DS}) in the sense that the resources required to realize ε_{DS} do not depend on the number of data structures (the change is local to class `Movie` in Table 2);
- It is "inflexible" towards adding an operation (ε_{Op}) in the sense that the resources required to realize ε_{Op} *grow* with the number of the data structures (every data structure would have to be modified).

In the following section we demonstrate how EC establishes these intuitions.

2.3 Evolution complexity

From our discussion in the notion of flexibility and from its definitions (Section 1) we understand that particular software is considered "flexible towards evolution step ε" if the resources required for realizing it are "fixed". In actuality, the "effort" required to carry out a particular change, whether measured in terms of time or money, 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 financial factors that are difficult to measure. Given the complexity of this problem we ask, How can we quantify flexibility? A possible solution to this problem can be obtained by combining notions of measurement in the theory of computational complexity with what can be summarized as the principle "software evolution is software too".

In his award-winning paper "Software Processes are Software Too" [20] ([5]), Osterweil suggests to treat human and executable processes uniformly. In other words, that every aspect of the software process can be conceptualized as software, thereby allowing it to be subjected to the same techniques and formalisms we use to analyze software. The reason is 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." [21]

Evolution complexity can be defined as the application of Osterweil's principle to software evolution. We conceptualize the evolution process as software (or technically, as a metaprogram ([6])), thereby subjecting it to the same techniques and formalisms used for analyzing software. For example, we may use computational complexity to measure the complexity of the evolution process, leading to following definition:

([5]) In a 10 year retrospective, Osterweil's paper of 1987 was awarded in 1997 a prize as the most influential paper of ICSE 9.

([6]) A metaprogram is a program that manipulates other programs. The canonical example is a compiler. Metaprogramming functions are built into many programming languages.

Definition 1 (*evolution complexity*). The complexity of a co-evolution step ε is the complexity of the metaprogramming process that realizes ε.

Computational complexity [27] is concerned with estimating how the resources that computational processes require grow with changes in the problem. Similarly, evolution complexity is concerned with estimating how the resources that evolution processes require grow with changes in the problem. In the remainder of this paper we demonstrate Definition 1 and illustrate how it can be used to corroborate and quantify informal claims regarding the (hitherto elusive) quality of flexibility of specific programming paradigms, architectural styles and design patterns.

Caveat

We identify the following possible misconceptions of Definition 1:

- We do not claim that software evolution can, should, or eventually will be fully automated. Rather, we argue that treating the evolution process *as if* it were automated is a metaphor that is useful for the purpose of quantifying its complexity. ([7])
- Computational complexity does not measure the *actual* time that computational processes require. Likewise, EC does not measure the *actual* resources evolution processes requires, but its *complexity*. "Complexity" measures not the magnitude of these resources but how they *grow* as a function of the change in the problem.

The `Collection` interface revisited

The complexity of a particular computational process is measured by breaking it into commensurable sub-steps. Similarly, the complexity of a particular co-evolution step $\varepsilon=\langle\sigma,\alpha\rangle$ can be measured by breaking it into commensurable sub-steps. A first approximation to this measure is provided by the number of modules in adjustment α that are affected by shift σ. Thus, a rudimentary EC metric is obtained by calculating the number of modules that were changed, added or removed as a result of the respective shift. This is obtained by calculating the symmetric set difference between the sets of modules in the "before" and "after" implementations. In a class-based programming language such as Java, classes are such modular units. This metric is encoded by the following definition:

([7]) Similar misconceptions of Osterweil's general principle are discussed in [21].

Definition 2. The metric $\mathcal{C}^{1}_{Classes}$ measures the complexity of a co-evolution step $\varepsilon = \langle \sigma, \langle i_{old}, i_{new} \rangle \rangle$ as follows:

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

Where $\Delta\, Classes(i_{old}, i_{new})$ designates the symmetric set difference ([8]) between the set of class definitions in i_{old} and the set of class definitions in i_{new}.

This metric relies on the simplifying assumption that the resources required for adding, removing, or changing each modular unit commensurate. We discuss this assumption in the next subsection.

Let us demonstrate how $\mathcal{C}^{1}_{Classes}$ can be used to compare the complexity of the two co-evolution steps described in Collection example (Section 2.2). Consider the number of classes affected by changing the data structure (ε_{DS}) with the number of classes affected by adding an operation (ε_{Op},). The results of this comparison are summarized in Table 3.

Table 3. $\mathcal{C}^{1}_{Classes}$-complexity of evolving the Collection interface

Shift / Design policy	Change data structure	Add operation
Collection interface	$\mathcal{O}(1)$	$\mathcal{O}(\lvert DS \rvert)$ ([9])

This suggests that the complexity of changing the data structure (ε_{DS}) is independent of the size of the implementation (constant complexity) whereas the complexity of adding an operation (ε_{op}) is proportional to the number of data structures in the implementation (linear complexity). It implies, for example, that adding an operation to a library with 20 data structures is twice as hard as making the same adjustment to a library with only 10 data structures. This establishes the intuitions expressed in Section 2.2. It also corroborates our more general claim regarding the notion of flexibility, namely that each design policy is only flexible towards the class of changes it was specifically tailored to accommodate.

([8]) The symmetric set difference between sets S and T is defined as $S \Delta\, T \triangleq (S \setminus T) \cup (T \setminus S)$

([9]) 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.

The generalized metric for EC

Early in the software lifecycle, for example during the design phase, only very general information on the software is available (it's "architecture") but a detailed implementation is not. In such situations, a coarse metric such as $\mathcal{C}^{1}_{Classes}$ can provide us with a first approximation that can help the software designer to choose the "most flexible" design policy.

However, $\mathcal{C}^{1}_{Classes}$ is inadequate in at least three situations:

(a) *Where the evolution of different class definitions do not commensurate*. During late phases in the software lifecycle, it may become evident that the evolutions of two classes do not commensurate. For example, evolving a class definition of 2-lines does not commensurate with the evolution of a class definition that is 200-lines long. In such cases more refined EC metrics, which possibly take into account the size of each class, can give better approximations.

(b) *Where classes were not yet defined*. During even earlier phases of architectural design, a complete list of classes has not been compiled yet. In such cases, less refined approximations are required. For example, a metric incorporating a less refined notion of modularity such as Java's *package* structure may be more appropriate.

(c) *Where the programming language does not support classes*. Evolution complexity should also be defined for programming languages that are not class-based.

We conclude that EC must accommodate both for varying degrees of modular granularity, as well as for varying degrees of information on each module. This intuition is relayed in the generalized metric for evolution complexity:

Definition 3. The generalized metric $\mathcal{C}^{\mu}_{Modules}$ measures the complexity of a co-evolution step $\varepsilon = \langle \sigma, \langle i_{old}, i_{new} \rangle \rangle$ as follows:

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

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

The generalized metric is parameterised by the variables $Modules$ and μ:

- $Modules$ represents any notion of "module", including classes, procedures, methods, and packages. For example, $\mathcal{C}^{1}_{Packages}$ is a coarse metric which measures EC in terms of number of packages in the implementation.
- μ represents any metric of software complexity that is meaningful w.r.t. 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, whereas fixing $\mu = CC$ (Cyclomatic Complexity) incorporates information about the flow control.

Evidently, Definition 3 gives rise to any number of EC metrics. Various such metrics are demonstrated in the remainder of this paper.

2.4 Discussion

Why assume that the resources required for *adding*, *removing*, and *changing* a module commensurate? Because experience teaches that the cost of software evolution exceeds that of the development process [1]. (This is probably due to the inherent complexity of large systems, where every change in an existing module can potentially have a "domino effect", which is precisely the reason why quantifying flexibility is of interest.) It is therefore not unreasonable to assume that *removing* and *changing* a module requires on average resources of the same magnitude as *adding* a new module.

Nonetheless, a situation may arise where only *adding* a module is demanding and where the effort it involves is, say, proportional to its size (which can be measured in terms of "Lines of Code"). Let us assume that this is the case, and that *removing* and *changing* a module are inconsequential. To model evolution complexity after these concerns, we may construct an ad-hoc a metric $\mathcal{C}^{Add/LoC}_{Modules}$ can as follows:

Definition 4. The metric $\mathcal{C}^{Add/LoC}_{Modules}$ measures the complexity of a co-evolution step $\varepsilon = \langle \sigma, \langle i_{old}, i_{new} \rangle \rangle$ as follows:

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

Alternatively to this ad-hoc approach, we may specialize the generalized metric for this purpose. Observe that $\mathcal{C}^{Add/LoC}_{Modules}$ can also be derived from Definition 3 by replacing μ in with the software complexity 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 example demonstrates that the generalized metric for evolution complexity can be used for measuring any well-defined claim on flexibility, namely whenever such a claim is made in the context of a concrete co-evolution step. Further demonstrations to this claim are given in the following section.

3 Case studies

In this section, we demonstrate the use of EC metrics to establish and quantify informal claims on the flexibility of various programming paradigms, architectural styles, and design patterns.

3.1 Architectural styles

In 1972, Parnas [22] presented the problem of Key Word in Context (KWIC) to discuss the cons and pros of two modular decomposition policies. The problem has become a classic in software design literature, and in 1993 Garlan and Shaw [24] used it to demonstrate the flexibility of architectural styles. In this section, we use EC to establish and to quantify the informal claims made in [24].

Problem

Parnas describes the KWIC problem as follows:

> *The KWIC [Key Word in Context] 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. [22]*

The description can be encoded as follows:

- $p_{KWIC} \triangleq$ *Represent an indexing system with data*

$$Lines=\{Line_1 \ldots Line_L\}; Words=\{Word_1 \ldots Word_W\}; Chars=\{Char_1 \ldots Char_C\}$$

and algorithms

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

Implementations

Parnas discusses several design ("modular decomposition") policies that can possibly guide the solution to the KWIC problem. In their analysis of the same problem, Garlan and Shaw [24] describe Parnas' solutions as three architectural styles: Shared Data, Abstract Data Structure and Pipes and Filters. Below we describe each one of these solutions. ([10]) Note that they differ not in the number of modules, which is $|Alg|$ in all architectural styles, but in the way data and functionality are distributed:

- $i_{Shared} \triangleq$ Implement p_{KWIC} as *Shared Data*. Traditional modular decomposition ("functional decomposition" according to Parnas) yields one module-per-functionality, all of which operate on some shared representation of the data. i_{Shared} is illustrated in Figure 3.
- $i_{ADT} \triangleq$ Implement p_{KWIC} as *Abstract Data Type*. Alternatively, Parnas describes an implementation that conforms to the principles of data abstraction (a.k.a. *information hiding*), where operations over data are only allowed via an abstract interface. i_{ADT} is illustrated in Figure 4.
- $i_{P\&F} \triangleq$ Implement p_{KWIC} as *Pipes and Filters*. Finally, Garlan and Shaw suggest a modular decomposition which encapsulates each algorithm in an independent module, or a "filter", which is a stateless process that accepts an input stream and produces an output stream. $i_{P\&F}$ is illustrated in Figure 5.

([10]) In addition, Garlan and Show 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.

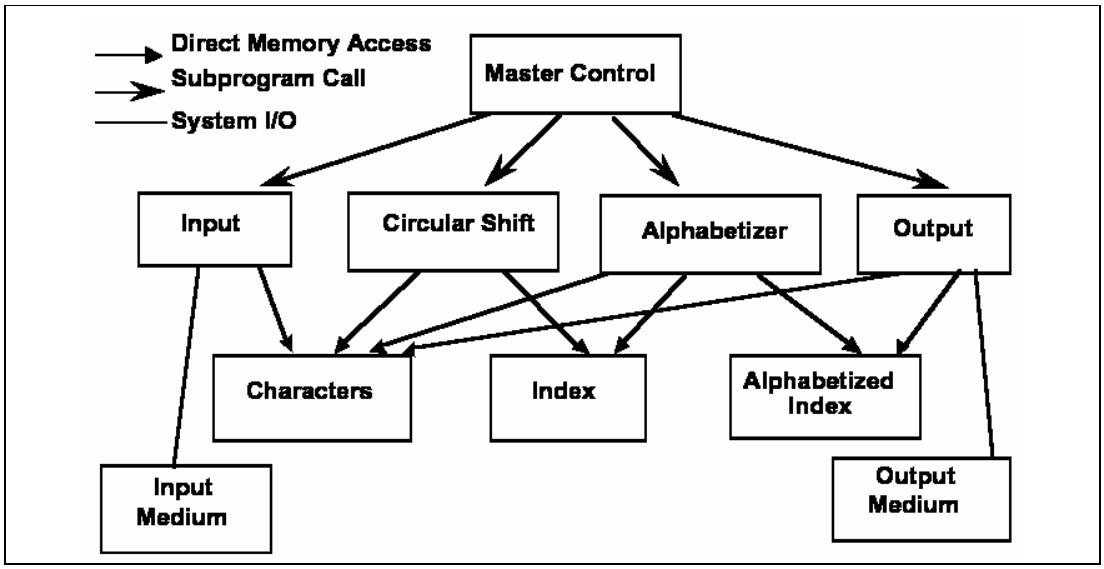

Figure 3. i_{Shared}, a Shared Data implementation to the KWIC problem (adapted from [24]).

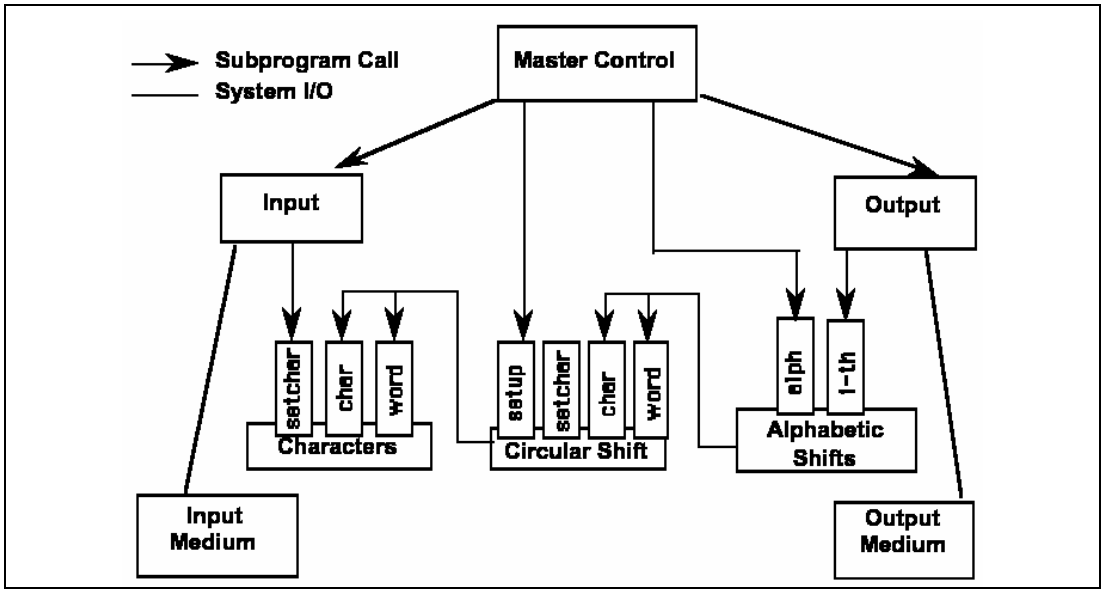

Figure 4. i_{ADT}, an Abstract Data Type implementation to the KWIC problem (adapted from [24]).

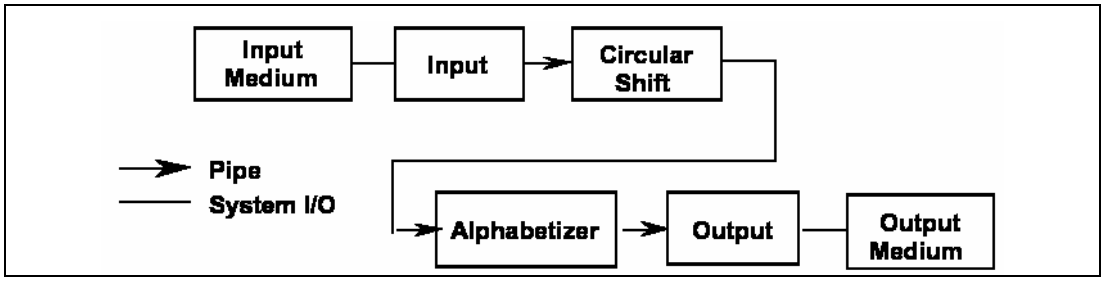

Figure 5. $i_{P\&F}$, a Pipes and Filters implementation to the KWIC problem (adapted from [24]).

Shifts

Consider the following shifts to p_{KWIC}:

- $\sigma_{Data} \triangleq$ Compact the data representation to an efficient format, e.g., by packing four letters to a byte.
- $\sigma_{paral} \triangleq$ Parallelize the processes, thereby allowing simultaneous, distributed processing of multiple documents.

Analysis

Informal claims about the flexibility of each implementation were made by Parnas [22], later refined by Shaw and Garlan [24].

Regarding i_{SR} 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 accomodated."

Regarding i_{ADT} 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 $i_{P\&F}$ 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 summarized in comparative matrix, cited in Table 4.

We may use the metric $\mathcal{C}^{1}_{Modules}$ to corroborate these claims and to make them precise. The results of this analysis are summarized in Table 5, whose structure follows the informal claims as cited in Table 4.

Table 4. Informal claims about the flexibility of three implementations towards shifts in the KWIC problem (adapted from [24]).

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

Table 5. $\mathcal{C}^{1}_{Modules}$-complexity of evolving three implementations towards shifts in the KWIC problem.

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

3.2 Programming paradigms

Object-oriented programming (OOP) is hailed, among other reasons, for promoting flexibility. Experienced programmers, however, observe that object-oriented mechanisms such as inheritance and dynamic binding make programs more flexible only towards the particular changes 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" [26] and yield highly inflexible systems.

The purpose of this subsection is to establish this intuition. We describe an example to the problem of representing a deterministic finite-state automaton (DFSA) and

analyze the complexity of evolving object-oriented vs. procedural implementations towards shifts in this problem.

Problem

We make the DFSA problem concrete using a digital clock as an example: Consider a 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 change between states or to perform a specific action depending on the current state. The clock's behaviour can be modelled as a deterministic finite state automaton (DFSA), illustrated in Figure 6.

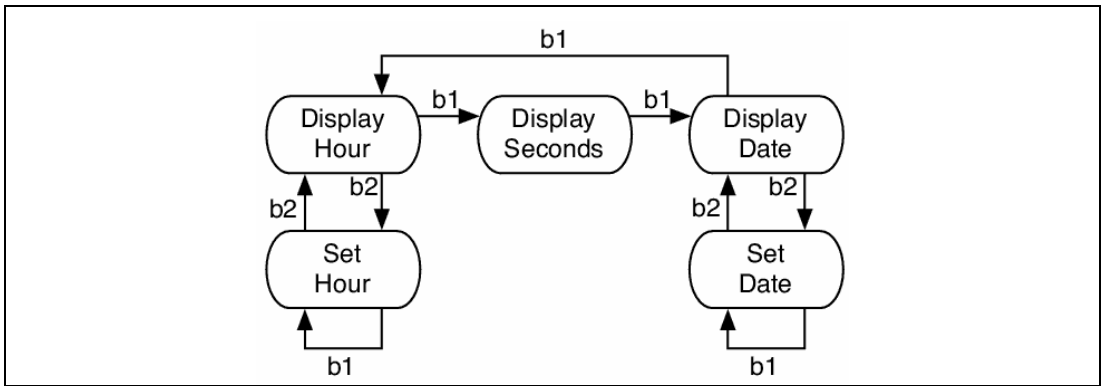

Figure 6. DFSA representation of a digital clock

We may encode this problem as follows:

- $p_{DFSA} \triangleq$ Represent a DFSA with a set of states

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

  and alphabet

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

Shifts

Consider two shifts to this problem:

- $\sigma_{+\zeta} \triangleq$ add letter ζ to Σ
- $\sigma_{+s} \triangleq$ add state s to S

Implementations

Consider the following two implementations of r_{DFSA}:

- $i_{OO} \triangleq$ *Implement* p_{DFSA} *in OOP*. The State pattern [8] manifests a classical object-oriented solution to the representation of a DFSA. Overall, there are $|S|$ classes and $|S| \times |\Sigma|$ methods in this implementation, sketched in Table 6.
- $i_{PROC} \triangleq$ *Implement* p_{DFSA} *in procedural style*. In the same chapter, the authors describe the "anti-pattern" as follows:

  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. ([8], p. 307)

  Overall, there are $|\Sigma|$ functions in this implementation, each consisting of a "switch" state with $|S|$ "cases", sketched in Table 7.

Table 6. i_{OO}, an object-oriented (Java™) implementation to the Clock problem

```
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 */}
}
```

Table 7. i_{proc}, a procedural (ANSI C) implementation of the Clock problem

```
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:          /*...*/; }
}
```

Analysis

Informal claims about the flexibility of each co-evolution step are made in [8].

Regarding i_{OO} they claim:

- "new states ... can be added easily"
- "decentralizing the transition logic in this way makes it easy to modify or extend the logic" (*ibid.*, pp. 307–308)

Regarding i_{PROC} they claim:

- "Adding a new state ... complicates maintenance."

The difficulty at this point lies in comparing adjustments in a Java™ program with adjustments in an ANSI C program. The question is, Which commensurable units of modularity can be used to compare such adjustments? As the simplest solution, we define the function $Class/Func(i)$:

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

Fixing also the software complexity metric $\mu = 1$ yields the metric $\mathcal{C}^{1}_{Class/Func}$. The results of analyzing the complexity of two co-evolution steps in this example using $\mathcal{C}^{1}_{Class/Func}$ are summarized in Table 8.

Table 8. $\mathcal{C}^{1}_{Class/Func}$-complexity of evolving object-oriented vs. procedural implementations towards shifts in the DFSA problem.

Shift / Design policy	Add letter	Add state
O-O programming	$\mathcal{O}(\lvert S \rvert)$	$\mathcal{O}(\lvert 1 \rvert)$
Procedural programming	$\mathcal{O}(\lvert 1 \rvert)$	$\mathcal{O}(\lvert \Sigma \rvert)$

It may be (justly) argued however that evolving ANSI C functions and Java™ classes do not commensurate, and that a more refined approach may be in place. A more sophisticated notion of software complexity, for example as suggested by the cyclomatic complexity metric [17], may provide a more faithful measure of the relative complexity of different modules. This yields the following metric:

Definition 5. The metric $\mathcal{C}^{CC}_{Class/Func}$ measures the complexity of a co-evolution step $\varepsilon = \langle \sigma, \langle i_{old}, i_{new} \rangle \rangle$ as follows:

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

where $CC(m)$ of module m is its Cyclomatic Complexity, namely—

$$CC(m) = \begin{cases} \text{\#nodes in the flow graph of } m & \text{when } m \text{ is an ANSI C function} \\ \text{Total sum of \#nodes in the flow graphs of all methods in } m & \text{when } m \text{ is a Java class} \end{cases}$$

Analyzing the complexity of the same co-evolution steps using the metric $\mathcal{C}^{CC}_{Class/Func}$ yields slightly different results, summarized in Table 9.

Table 9. $\mathcal{C}^{CC}_{Class/Func}$ -complexity of evolving an object-oriented vs. procedural implementation of DFSA problem.

Design policy \ Shift	Add letter	Add state
O-O programming	$\mathcal{O}(\lvert S\rvert \times \lvert \Sigma\rvert)$	$\mathcal{O}(\lvert \Sigma\rvert)$
Procedural programming	$\mathcal{O}(\lvert S\rvert)$	$\mathcal{O}(\lvert \Sigma\rvert \times \lvert S\rvert)$

Discussion

We may conclude from the results from applying different EC metrics the following:

- That neither programming paradigm is "flexible" in absolute terms, regardless of the metric chosen.
- That the answer to the question, "Which programming paradigm is more flexible?" can be reduced to the question, "Which changes to the problem are most likely?".
- That the accuracy of the EC metric varies with the amount of information available about the implementation.
- That metrics of different granularity levels are useful during both early and late phases in the software lifecycle: Unrefined metrics, such as $\mathcal{C}^{1}_{Class/Func}$, are useful during the design process, whereas refined metrics, such as $\mathcal{C}^{CC}_{Class/Func}$, become useful during software evolution.

A third solution to the DFSA problem has not been discussed: States and alphabet 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.

3.3 Design patterns

In [18], we analyzed the evolution complexity of two design patterns. In this section, we summarize the conclusions drawn from our analysis.

Visitor

The patterns catalogue [8] discusses the problem of representing abstract syntax trees and operations thereon. It is argued that an "ideally" flexible implementation is one

written in a programming language that supports *double dispatch* [4]. Since such mechanism is not provided by the programming languages at the focus of the authors (C++ and Smalltalk in [8], but also neither in most other O-O programming languages), the Visitor pattern is presented. The pattern consists of two class hierarchies, representing (a) the set of elements in abstract syntax (e.g., a *variable*) and (b) the set of operations thereof (e.g., "print this tree").

Regarding the Visitor they claim:

- "[it] makes adding new operations easy"
- "Adding new concrete element classes is hard"

We may use $\mathcal{C}^{1}_{Classes}$ to corroborate these claims and to make them precise. The results of this analysis are summarized in Table 10.

Table 10. $\mathcal{C}^{1}_{Classes}$-complexity of evolving the Visitor pattern

Shift / Design policy	Add operation	Add element
$i_{Visitor}$	*1*	$\mathcal{O}(\lvert Ops \rvert)$

Abstract Factory

The patterns catalogue [8] 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 GUI (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 each.

The authors discuss two design policies in implementing a solution to this problem: In the first (the "anti-pattern"), a simple compound "switch" statement (multiple conditional branching) is used to determine which object to create. As an alternative, the Abstract Factory pattern conforms to the general spirit of the OOP paradigm, employing the standard combination of subtyping and dynamic binding to hide the concrete class of the object created behind a uniform interface. In [18], we analyze the complexity of evolving each design policy, the results of which are summarized in Table 11.

Table 11. $\mathcal{C}^{1}_{Classes}$-complexity of evolving the Abstract Factory vs. Switch implementations towards shifts in the Object Creation problem (with K clients, P products, C configurations).

Shift / Design policy	Add configuration	Add state
Abstract Factory	$\mathcal{O}(\|P\|)$	$\mathcal{O}(\|C\|+\|P\|)$
"Switch"	$\mathcal{O}(\|K\|)$	$\mathcal{O}(\|K\|)$

These results provide further corroboration to the more general claims we made, namely that the flexibility of each implementation depends on which shifts are most likely. This suggests that a software architect must weigh carefully the question which shifts exactly are most likely to occur before choosing the appropriate design policy. For example, because the 'switch' design policy would be the most flexible choice for an application with considerably more clients than products and configurations.

4 An experiment in EC

A metric is validated when it can be shown to measure what it is supposed to measure. The most obvious test to the metrics we suggested is to investigate how the actual resources required for realizing a particular evolution step grow as a function of the respective shift in a controlled environment. Below, we describe the initial results obtained from consolidating the results obtained from conducting several small-scale experiments to this extent at the University of Essex and at the Université de Mons-Hainaut. Further empirical evidence will require larger sample groups.

This experiment was designed to establish some of the specific predictions made in Section 3.2 regarding the complexity of evolving an object-oriented implementation towards shifts in the DFSA problem. The detailed instructions as handed to the subjects were made available in [6]. Below, we summarize the relevant tasks the conclusions drawn.

In the first part of the experiment, subjects were required to implement a rudimentary clock, represented as a state machine (p_{DFSA}) with three states ($|S|=3$) and one letter in the alphabet ($|\Sigma|=1$). 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 4, subjects added a second letter to Σ (shift $\sigma_{+\zeta}$)
- In task 5, subjects added three more states to S (shift σ_{+s})
- In task 6, subjects added a third letter to Σ

In Section 3.2, we suggest the metrics $\mathcal{C}^{1}_{Class/Func}$ and $\mathcal{C}^{CC}_{Class/Func}$ for measuring the complexity of these evolution steps. Both metrics predict that the complexity of adding a letter to the alphabet grows proportionally with the number of states ($|S|$). In terms of this experiment, both metrics predict that task 6 will require twice the time that task 4 requires. In reality, task 6 took on average 1.56 times as much as task 4. We believe that these initial results, collected from 4 subjects (standard deviation: 1.8), corroborate our prediction.

5 Related work

Software complexity and evolution complexity

Curtis suggests that "In the maintenance phase [software] complexity determines … how much effort will be required to modify program modules to incorporate specific changes." [5] Zuse [28] counts over two hundred metrics for software complexity in literature. Three prominent examples are the following:

- *Lines of Code* (LoC) counts the number of lines in the program's text.
- McCabe's *Cyclomatic Complexity* (CC) [17] measures the number of nodes in the flow graph of the program (demonstrated in Section 3.2).
- Halstead's *Volume* [11] 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.

In Sections 2.4 and 3.2, we demonstrated how various software complexity metrics can be used for measuring flexibility. Unfortunately, neither simple nor sophisticated software complexity metrics have been proven accurate as indicators of productivity, comprehensibility or maintainability [28]. This suggests that the accuracy of an EC metric is only limited by the accuracy of the software complexity. We hope that the future will bring accurate software complexity metrics.

Metrics for software evolution

Quantifying the actual resources required for software evolution remains a relatively unexplored problem. Jørgensen [14] used several models to predict the effort that ran-

domly selected software maintenance tasks require. The size of individual maintenance tasks was measured in LOC. Sneed [25] proposed a number of ways to extend existing cost estimation methods to the estimation of maintenance costs. Ramil et al. [23] 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 ways in which it is tied to the notion of software flexibility.

Metrics for software flexibility

We are unaware of alternative approaches for measuring 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 [16]. For example, Chaumun et al. [3] 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 [2].

6 Summary and conclusions

We argued that "flexibility" is a quality that is relative to the change in the problem. We corroborated this claim by examining the flexibility of recognized programming paradigms, architectural styles and design patterns. We concluded that neither design policy is flexible in absolute terms.

We suggested that, if a software process such as evolution or development is treated as an executable (meta-)program, its complexity can be measured by borrowing notions from computational complexity. This motivated our definition of evolution complexity and of various metrics thereof. We suggested that "flexibility" can be quantified in these terms.

We demonstrated that the benefits gained from this approach are manifold:

(a) EC can be used to corroborate and quantify informal claims on the flexibility of particular programming paradigms, architectural styles, and design patterns.
(b) EC can be used to measure "flexibility" with varying degrees of accuracy.
(c) EC can be used to choose the most "flexible" design policy, given the most likely changes to the problem.

7 Future directions

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

Evolution complexity can be used to analyze 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 vs. .NET). In particular, EC can be used in supporting the decision whether to apply a particular refactoring [7], possibly by incorporating a range of EC metrics into integrated development environments which support *refactoring*, such as IBM Eclipse and Borland JBuilder.

An examination of the relation between evolution complexity and actual resources consumed by the evolution effort 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, evolution complexity 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.

Evolution complexity tradeoffs

Studying the flexibility of different programming paradigms towards shifts in the DFSA problem (Section 3.2) suggests that a trade-off that may exist between EC and computational complexity, a relation that is analogous to the trade-off between space and time (computational) complexities. While 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 measuring the complexity of the software development process are yet to be proposed.

Acknowledgements

This research has been carried out in the context of the scientific network RELEASE financed by the European Science Foundation (ESF). The authors wish to thank Jeff

Reynolds and Mehdi Jazayeri for their comments and suggestions. We also wish to thank Nguyen Long, Jon (Mac) Nicholson, Peter Ebraert, and K. Allem for taking part in our experiment. The authors wish to thank Naomi Draaijer and Mary J. Anna for their inspiration.

References

[1] B. Boehm. *Software Engineering Economics*. Englewood Cliffs: Prentice-Hall, 1981.

[2] N. Chapin, J. Hale, K. Khan, J. Ramil, W.G. Than. "Types of software evolution and software maintenance". *J. Software Maintenance and Evolution*, Vol. 13 (2001), pp. 3–30. New York: John Wiley & Sons.

[3] M.A. Chaumun, H. Kabaili, R.K. Keller, F. Lustman. "A change impact model for changeability assessment in Object-Oriented Software Systems." *Science of Computer Programming*, Vol. 45, Nos. 2–3 (2002), pp. 155–174. Amsterdam: Elsevier.

[4] I. Craig. *The Interpretation of Object-Oriented Programming Languages*. New York: Springer-Verlag, 2000.

[5] B. Curtis. "In search of software complexity." *Proc. Workshop on Qualitative Software Models for Reliability, Complexity and Cost* (Oct. 1979), pp. 95–106.

[6] A.H. Eden. "An experiment in evolution complexity: instructions to subjects." Technical report CSM-431 (Jun. 2005), Department of Computer Science, University of Essex, ISSN 1744-8050.

[7] M. Fowler. *Refactoring*. Addison-Wesley, 2003.

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

[9] D. Garlan, G.E. Kaiser, D. Notkin. "Using tool abstraction to compose systems." *IEEE Computer*, Vol. 25, No. 6 (June 1992), pp. 30–38. Los Alamitos: IEEE Computer Society.

[10] W.W. Gibbs. "Software's chronic crisis". *Scientific American* (Sep. 1994), p. 86–95.

[11] M.H. Halstead. *Elements of Software Science*. New York: Elsevier, 1977.

[12] IEEE. *Standard Glossary of Software Engineering Terminology* 610.12-1990, Vol. 1. Los Alamitos: IEEE Press, 1999.

[13] ISO. *Information Technology—Software Product Evaluation—Quality Characteristics and Guidelines for their Use.* ISO/IEC 9126. ISO/IEC, 1991.

[14] M. Jørgensen. "Experience with the accuracy of software maintenance task effort prediction models." *IEEE Trans. Software Engineering*, Vol. 21, No. 8 (1995), pp. 674–681. Los Alamitos: IEEE Computer Society Press.

[15] M.M. Lehman, J.F. Ramil, P. Wernick, D.E. Perry, W.M. Turski. "Metrics and laws of software evolution—the nineties view." *Proc. Int'l Symposium Software Metrics* (5–7 Nov., 1997), Albuquerque, NM, pp. 20–32 . Los Alamitos: IEEE Computer Society Press.

[16] L. Li, A. Offutt. "Algorithmic analysis of the impact of changes to object-oriented software." *Proc. Int'l Conf. Software Maintenance—ICSM* (1996), pp. 171–184. Los Alamitos: IEEE Computer Society Press.

[17] T. McCabe. "A software complexity measure." *Trans. Software Engineering* Vol. 2 (1976), pp. 308–320. Los Alamitos: IEEE Computer Society Press.

[18] T. Mens, A.H. Eden. "On the evolution complexity of design patterns". *Electronic Lecture Notes in Computer Science*, Vol. 127, No. 3, pp. 147–163. Amsterdam: Elsevier, 2004.

[19] P. Naur, B. Randell, (eds.) *Software Engineering: Report of a conference sponsored by the NATO Science Committee* (7–11 Oct. 1968), Garmisch, Germany. Brussels, Scientific Affairs Division, NATO, 1969.

[20] L. Osterweil. "Software processes are software too" *Proc. 9th Int'l Conference Software Engineering—ICSE* (Mar. 1987), pp. 2–13. Los Alamitos: IEEE Computer Society.

[21] L. Osterweil. "Software processes are software too revisited." *Proc. 19th Int'l Conference Software Engineering—ICSE* (May 1997), pp. 540–548. Los Alamitos: IEEE Computer Society.

[22] D.L. Parnas. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, Vol. 15, No. 12 (Dec. 1972), pp. 1053–1058. New York: ACM Computing Society.

[23] J.F. Ramil, M.M. Lehman. "Metrics of software evolution as effort predictors—a case study". *Proc. Int'l Conf. Software Maintenance* (Oct. 2000), pp. 163–172. Los Alamitos: IEEE Computer Society Press.

[24] M. Shaw, D. Garlan. *Software Architecture—Perspectives on an Emerging Discipline*. Upper Saddle River: Prentice Hall, 1996.

[25] H. Sneed. "Estimating the costs of software maintenance tasks." *Proc. Int'l Conf. Software Maintenance* (1995), pp. 168–181. Los Alamitos: IEEE Computer Society Press.

[26] A. Taivalsaari. "On the notion of inheritance". *ACM Computing Surveys*, Vol. 28, No. 3 (Sep. 1996), pp. 438–479. New York: ACM Computing Society.

[27] A. Urquhart. "Complexity". Ch. in L. Floridi (ed.), *The Blackwell Guide to Philosophy of Computing and Information*. Oxford: Blackwell, 2004.

[28] H. Zuse. *Software Complexity*. Berlin: Walter de Gruyter, 1998.