Writing with Style

William Cook
based on
Style: Toward Clarity and Grace
by Joseph Williams

1

Encode
a complex web
of ideas…

2

…as a linear
stream of text

3

paper
organization
≠
research
process

4

Significance	Motivate why the research is important or useful. Explain what problem it addresses.
Clarity	Organize the paper well and write clearly. Make sure you support your claims.
Novelty	Extend the frontier of knowledge. Explicitly relate your research to previous work.
Correctness	Critically evaluate and support your claims with proofs, an implementation, examples, or experiments.

5

Clarity

6

- Subject of sentence names a character
- Verbs name action involving characters

Missing Subjects

"Termination occurred after 23 iterations"

Missing Subjects

character

"The program terminated after 23 iterations"

action

Missing Subjects

"Determination of policy occurs at the presidential level"

Subject = Actor

"The President determines policy"

Weak Verbs

"The algorithm supports effective garbage collection in distributed systems"

Stronger

"The algorithm collects garbage effectively in distributed systems"

13

NOM: Nominalization

Noun instead of verb/adjective

14

Verb NOM

Verb	*Nominalization*
discover	discovery
move	movement
collaborate	collaboration

15

Adjective NOM

Adjective	*Nominalization*
difficult	difficulty
applicable	applicability
different	difference

16

empty verb + NOM

"The police conducted an investigation of the matter"

17

Verb = Action

"The police investigated the matter"

18

"there is" + NOM

"There is a need for further study of this program"

19

Name the Actor

"The engineering staff must study this program futher"

20

NOM + empty verb

"The intention of the IRS is to audit our records"

weak

21

Verb = action

"The IRS intends to audit our records"

22

NOM + NOM

"There was a review of the evolution of the technique"

23

Find Actor

"She reviewed the evolution of the technique"

24

Using “how”

“She reviewed how the technique evolved”

25

NOM + verb + NOM

“Extensive rust damage to the hull prevented repairs to the ship”

26

Actors, Actions

“Because rust had damaged the hull, we could not repair the ship”

27

Useful Nominalizations

28

Reference to previous sentence

“these arguments all depend upon…”

“This decision has…”

29

Name a verb’s object

“I do not understand her meaning or his intention”

(what she means
what he intends)

30

Common concepts

"Taxation without representation was not the central cause of the revolution"

31

Be careful

compilation

dependency

inheritance

implementation

32

Cohesion

Managing the flow of information

33

Sentences

subject	•ideas already mentioned •familiar ideas
verb	•action
object	•new ideas

34

Topics form a logical sequence of ideas

old new old new old new old

35

Technique

Underline subjects

Do they flow?

36

Emphasis

Put important things at the end

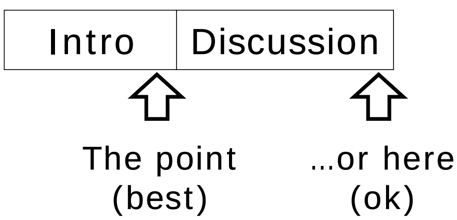
sentence	final words
paragraph	last sent.
section	last para.

37

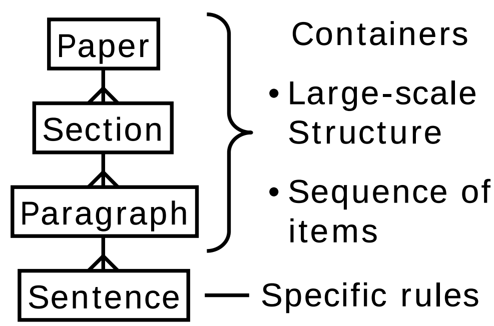
Coherence

38

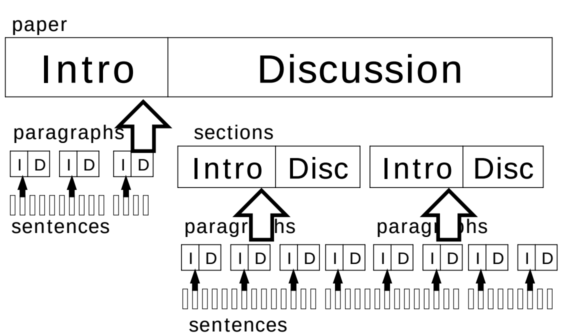
The Point



39



40



41

Themes

Strings of related words

Woven into the text

42

Active
Passive

43

Active

subject	The partners
verb	broke
object	the agreement

44

Passive

subject	The agreement
be + past participle	was broken
prepositional phrase	by the partners

45

Passive is fine, if it is more coherent

46

Active

"Our partners were old friends… but they let us down. The partners broke the agreement."

47

Passive

"We thought we had a good agreement. Then we found out who killed it. The agreement was broken by the partners."

48

Miscellaneous Rules

49

Section Title Rule

First sentence of every section: **Must include the section title**

(except intro/conclusion)

50

Little Piggy Rule

Avoid "we" as subject, unless it is something you, the author, actually did

51

"Our" Rule

Avoid "our", as in "our technique"

Give everything a <u>name</u> instead

52

"This" Rule

Avoid "this" as a subject.

Or qualify it: "this technique"..

53

Misc.

No parentheses (ever)

"fleshed" not "flushed"

54

Comma for clauses

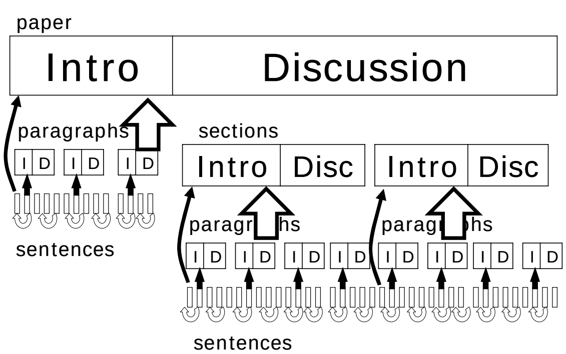
We went to the store
and bought some food.

We ate it,
and it was good.

55

Summary

56



57

Fixed	Topic	Stress
Variable	Old/Known	New/Unknown

Fixed	Issue	Discussion
Variable	Point	(Point)

Fixed	Subject	Verb	Complement
Variable	Character	Action	-

58

Exercises

59

This paper formalizes the notion of virtual classes, in the form of the language *vc*, an extension of Featherweight Java. We present its dynamic semantics and static type rules, and show that the type system is sound.

Let us introduce virtual classes by analogy. Mainstream object-oriented languages invariably enable (virtual) methods to mean different things in context of objects of different type, at the syntactic level by means of overriding definitions of methods in subclasses, and in the dynamic semantics by means of late binding in method invocations.

Virtual classes are class valued attributes of objects, and they can also be refined (to subclasses) in context of a subclass; at the syntactic level there are introductory and further-binding declarations, and at the dynamic level there is late binding. As a result, the actual, dynamic value of a virtual class is not known exactly at compile time, but it is known to be a particular class which is accessible as a specific attribute of a given object, and it is statically known to be a subclass of some compile-time constant class. Virtual classes give rise to covariance, which requires a strict treatment in order to be type safe, such as that of Caesar or gbeta. Other examples of a strict and safe treatment of covariance are the formalization of variant parametric types in [32], and the inclusion of wildcards into the J2SE 5 version of the Java platform. Note, however, that virtual classes is a different and in several respects more powerful mechanism than variant parametric types and wildcards.

A1

60

Virtual classes are class-valued attributes of objects. They are analogous to virtual *methods* in traditional object-oriented languages: they follow similar rules of definition, overriding and reference. In particular, virtual classes are defined within an object's class. They can be overridden and extended in subclasses, and they are accessed relative to an object instance, using late binding. This last characteristic is the key to virtual classes: it introduces a dependence between static types and dynamic instances, because dynamic instances contain classes that act as types. As a result, the actual, dynamic value of a virtual class is not known at compile time, but it is known to be a particular class which is accessible as a specific attribute of a given object, and some of its features may be statically known, whereas others are not.

61

A2

Proof. By induction on the structure of *c*. Rule A-Assign computes a safe approximation of the store by joining stores. Rule A-If computes a safe approximation of the program behavior by combining the results of the statement's two branches. …

Analyzing Traversal Conditions

The precision of the analysis can be significantly increased by analyzing the conditions under which a program traverses its paths. For example, the analysis in the previous section conservatively estimates that the program in Fig. 3 needs the *name* field for every employee even though the program traverses the *name* field only if the employee's salary is greater than $65,000. We extend our analysis to identify and include such conditions, so that they may be expressed in a database query.

62

B1

The analysis in this section is quite imprecise and can therefore lead to an excessive over-approximation of the database values a program requires. For example, the analysis conservatively estimates that the program in Fig.3 needs the *name* field for every employee even though the program traverses the *name* field only if the employee's salary is greater than $65,000.

Analyzing Traversal Conditions

The precision of the analysis can be significantly increased by considering the conditions under which a program traverses data paths. If a program condition can be expressed in a query language, then the analysis can incorporate that condition in the traversal summary.
…

63

B2

An understanding of inheritance as a general concept at the same level as recursion or iteration provides a basis for interpreting inheritance systems provided by particular programming languages. Just as iteration and recursion have many different forms, so does inheritance, and it is impossible to state exactly what inheritance is. One may merely identify when a form of inheritance is supported. Most of the confusion about inheritance stems not from the actual mechanism is provides but from the expectations of those who wish to use it. The first task of this paper is to extricate inheritance from the grip of these expectations.

We first examine the connection between inheritance and the subtype relation. We imagine a situation in which two classes implement the same type, but are not related by inheritance (subtypes without inheritance), and conversely a situation in which one class inherits from another but is not a subtype (inheritance without subtypes). Both situations are common enough in object-oriented programming so we reject the position that inheritance is necessarily equated with subtyping. We view inheritance as providing an implementation hierarchy, although type-checking does play a role in determining the correctness of inheritance.

64

C1

Inheritance is one of the central concepts in object-oriented programming. Despite its importance, there seems to be a lack of consensus on the proper way to describe inheritance. This is evident from the following review of various formalizations of inheritance that have been proposed.

The concept of prefixing in Simula (Dahl and Nygaard, 1970), which evolved into the modern concept of inheritance, was defined in terms of textual concatenation of program blocks. However, this definition was informal, and only partially accounted for more sophisticated aspects of prefixing like the pseudo-variable this and virtual operations.

The most precise and widely used definition of inheritance is given by the operational semantics of object-oriented languages. The canonical operational semantics is the “method lookup” algorithm of Smalltalk: (omitted)

Unfortunately, such operational definitions do not necessarily foster intuitive understanding. As a result, insight into the proper use and purpose of inheritance is often gained only through an “Aha!” experience (Borning and O’Shea, 1987).

65

C2

Direct instruction communication--in which instructions in a block send their operands directly to consumer instructions within the same block in a dataflow fashion--permits distributed execution by eliminating the need for any intervening shared, centralized structures such as an issue window or a register file between the producer and consumer.

As shown in Figure 5, the TRIPS ISA supports direct instruction communication by encoding the consumers of an instruction as targets within the producing instruction, allowing the microarchitecture to determine where the consumer resides and forward a produced operand directly to its target instruction(s). The nine-bit target fields (T0 and T1) shown in the encoding each specify the operand type (left, right, predicate) with two bits and the target instruction with the remaining seven. A microarchitecture supporting this ISA will determine where each of a block's 128 instructions is mapped, thereby determining the distributed flow of operands along the dataflow graph within each block. An instruction's number is implicitly determined by its position in the chunks shown in Figure 6.

66

D1

With *direct instruction communication*, instructions in a block send their results directly to intra-block, dependent consumers in a dataflow fashion. This model supports distributed execution by eliminating the need for any intervening shared, centralized structures (e.g. an issue window or register file) between intra-block producers and consumers.

Figure 5 shows that the TRIPS ISA supports direct instruction communication by encoding the consumers of an instruction's result as targets within the producing instruction. The microarchitecture can thus determine precisely where the consumer resides and forward a producer's result directly to its target instruction(s). The nine-bit target fields (T0 and T1) each specify the target instruction with seven bits and the operand type (left, right, predicate) with the remaining two. A microarchitecture supporting this ISA maps each of a block's 128 instructions to particular coordinates, thereby determining the distributed flow of operands along the block's dataflow graph. An instruction's coordinates are implicitly determined by its position its chunk.

67 D2

2 ISA Description

The PowerPC ISA has some features that make it different from the Alpha and PISA ISAs. For example, the Alpha ISA has 25 instructions with 4 formats and the PISA ISA has 135 instructions with 4 formats. Not all of these instructions are implemented in the simulator. In this section, we describe features of the ISA that are implemented in the simulator.

68

3 TRIPS Architecture

The TRIPS architecture is designed to address key challenges posed by next-generation technologies—power efficiency, high concurrency on a latency-dominated physical substrate, and adaptability to the demands of diverse applications [10, 12]. It uses an EDGE ISA [2], which has two defining characteristics: *block atomic execution* and *direct instruction communication*. The ISA aggregates large groups of instructions into blocks which are logically fetched, executed, and committed as an atomic unit by the hardware. This model amortizes the cost of per-instruction overheads such branch predictions over a large number of instructions. With direct instruction communication, instructions within a block send their results directly to the consumers without writing the value to the register file, enabling lightweight intra-block dataflow execution.

69

4 Exit Predictor Design

In this section we describe hyperblock-based exit predictors in detail. We design exit predictors based on both conventional schemes and neural techniques. Exit predictors based on conventional techniques have a simple and scalable design, and can make fast predictions, with accuracies close to some of the best traditional branch predictors. The perceptron-based exit predictor requires more time to make a single prediction, but provides higher accuracy than other high-bandwidth exit predictors.

70

3 Critical Path Model

The critical path model for the TRIPS architecture is heavily based on the dependence-graph model previously developed for superscalar architectures [5]. The model represents various microarchitectural events as nodes in a directed acyclic graph. Edges between the nodes represent dependence constraints among the events. Figure 2 shows a typical dependence graph constructed for a slice of four blocks seen during the program execution. In addition to representing the usual constraints such as data dependences, branch mispredictions, and finite instruction window sizes, the TRIPS model represents constraints imposed by block atomic execution and operand routing.

71