NB: pour gagner du temps, traitez les parties A, B et C directement sur l'énoncé et placez-le à l'intérieur de la copie qui vous a été remise

A. Structures de la langue (4 points – niveau B1)

1°) Usage des pronoms relatifs et des déterminants possessifs et démonstratifs (1,5 pt): *corrigez 6 erreurs dans ce petit texte (une erreur correctement détectée rapporte 0,25 pt, une correction erronée coûte 0,15 pt)*

Bertrand Meyer is a French computer scientist who developed ~~her~~ career in the USA and in Europe. He invented Eiffel who was designed as a contract-based object-oriented language. ~~These~~ was his major contribution to the development of object-oriented programming. The text proposed in this test is taken from one of ~~it's~~ book who has been translated ~~to~~ many languages. ~~Her~~ current position is at ETH Zurich where he holds the chair of Software Engineering. He has also written another important book ~~that~~ presents the theory of programming languages.

2°) Données quantitatives (1,5 pt): *écrivez les informations notées numériquement en toutes lettres comme lorsque vous les prononcez à voix haute:*

Bertrand Meyer was born in 1950 (.....). He has published in 2006 (.....) a revised version of his book on object-oriented programming. As of 12/31/2012 (.....) he had published 348 (.....) articles, reports and books . He recently received €2.5 m (.....) from the European Union to develop a new project.

¼ · 107,811 = 26,952.75 (.....)

.....

3°) Comparatifs et superlatifs (1 pt) : *corrigez 4 erreurs dans le texte qui suit:*

(une erreur correctement détectée rapporte 0,25 pt, une correction erronée coûte 0,15 pt)

Meyer is one of the more famous software engineering expert in Europe. Eiffel has not been as successful ~~than~~ Java but its contract-based approach was recognised as a more elegant and robuster approach than what most other languages were offering. Eiffel also includes an exception management mechanism which is much ~~more strong~~ than old-fashioned error reporting techniques based on error codes passed back up to the master program.

B. Décomposition et prononciation des mots (3 points – niveau B1)

Indiquer la catégorie (N=nom, V=verbe, A=adjectif, AV=adverbe) et la prononciation en notation API des mots suivants du texte (après les avoir éventuellement décomposés en écrivant les préfixes et suffixes EN MAJUSCULES et en parenthésant au besoin les étapes de la décomposition)
catégorie finale : mot = décomposition des préfixes et suffixes => /prononciation déduite/

be, A pour adjectif et AV pour adverbe
s au programme de travail seront évaluées ; sur l'exemple donné
:

+ -AL => /ˈgʌvənˈmentəl/

notez les catégories avec N pour nom, V pour ver
NB : seules les règles de prononciation indiquée
ci-dessous ce serait la réduction du suffixe -AL :
A : governmental = N: (V : govern) + -MENT) +

..... => /...../ . . . . reusable = ...

..... => /...../ . . . . : underestimated = ...

..... => /...../ . . . . : capturing = ...

..... => /...../ . . . . : unsuccessfully = ...

ve/ (comme dans «play»).

C. Compréhension écrite élémentaire (4 points – nivea

Les justifications doivent être de très courts extraits du texte judicieuseme

a) Quel est le principal problème que veut soulever l'auteur dans cet extrait
- Le manque de maturité technique de la notion de module réutilis
- la répugnance des programmeurs à utiliser des logiciels écrits pa
- la difficulté d'adapter le logiciel aux composants électroniques
- l'irréalisme économique de vouloir réutiliser du logiciel

b) Faux Vrai

☒ □ Le coût du logiciel n'était pas un souci pour les militaires

c) Faux Vrai

□ ☒ Le développement du logiciel était vu comme une cause de dé

d) donnez deux synonymes en anglais de : « weave » : . . . . .

Dans l'extrait qui suit, cerclez précisément 4 occurrences de la diphtongue
(chaque cerclage correct rapporte 0,25 pt, chaque erreur coûte 0,15 pt)

itive nature. Over and over again,

ng, comparing, traversing, allocating,

so characteristic of their trade.

Anyone who observes software development cannot but be impressed by its repet

programmers weave a number of basic patterns: sorting, searching, reading, writi

synchronizing.... Experienced programmers know this feeling of déjà vu which is

u B1)

ent choisis

?
sable
ar d'autres

. . . . . .

. . . . . .

passement de budget et de délai

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

e) Faux Vrai
□ ☒ Le principal problème du logiciel n'est pas de définir les bons algorithmes de base

. . . . . .

f) Faux Vrai
☒ □ Il suffit de gagner un tout petit peu de temps pour adopter des composants réutilisables

. . . . . .

g) Faux Vrai
□ ☒ Les programmeurs n'écrivent jamais exactement le même programme deux fois

. . . . . .

h) donnez un synonyme en anglais de « moot » : . . . . . .

D. Reformulation et synthèse (3 points – niveau B2)

Answer each question in English in 5 to 10 lines ; carefully summarise and rephrase, and don't just copy-paste whole sentences from the original text: any select citation should be short and explicit (enclosed within quotation marks)

1°) In what sense is programming repetitive?

2°) What business model is envisaged for software products?

3°) In the final example given by Meyer, what do the parenthesized words written in uppercase represent?

E. Production écrite élémentaire (3 points – niveau B1 - 150 mots)

NB : les étudiants des niveaux 2 & 3 doivent traiter ce sujet en priorité car l'évaluation de cette production portera davantage sur le contenu que sur la correction de la langue

Describe problems you experienced when you tried to reuse a piece of software you had written before.

F. Production écrite avancée (3 points – niveau B2 – 250 mots)

NB : l'évaluation de cette production portera à la fois sur le contenu, le lexique et la syntaxe

B. Meyer wrote this text some 25 years ago. How do you perceive today's software reusability issues in relation to the different obstacles he mentioned? Have any solutions and/or new problems emerged in between?

3. APPROACHES OF REUSABILITY

"Why isn't software more like hardware? Why must every new development start from scratch? There should be catalogs of software modules, as there are catalogs of VLSI devices: when we build a new system, we should be ordering components from these catalogs and combining them, rather than reinventing the wheel every time. We would write less software, and perhaps do a better job at that when we do get to develop. Wouldn't then some of the problems that ever overruns, the lack of reliability, just go away? Why isn't it so?"

You have probably heard this kind of remark before; perhaps you have uttered them yourself. famous NATO workshop on the software crisis, D. McIlroy was advocating "mass produced as a dream, is not new. This chapter discusses some of the progress that must be achieved if

3.1 REPETITION IN PROGRAMMING

Anyone who observes software development cannot but be impressed by its repetitive nature programmers weave a number of basic patterns: sorting, searching, reading, writing, compar synchronizing.... Experienced programmers know this feeling of *déjà vu* which is so characteristic of their trade.

3.1.1 A naive question

One good way to assess this situation is to honestly answer the following question — again assuming you develop software, or direct people who do. Consider the general problem of table searching: an element of some kind, say X, is given, as well as a set T of similar elements, and the program should determine whether or not x appears in T. The question is the following: How many times over the past six months did you, or people working for you, write some program fragment for table searching? Chances are the answer will be one or more. But what is truly remarkable is that, most likely, this fragment will have been written at the lowest reasonable level of abstraction, namely as code in some programming language, rather than by calling existing routines. Yet table searching is one of the best researched areas of computer science: excellent books describe the fundamental algorithms, and it would seem that nobody should need to code a searching algorithm any more, at least in standard case – in the same way that electronic engineers do not design standard inverters: they buy them.

3.1.2 Non-technical obstacles

Why then is reuse not more common?

Some of the obstacles are economic. If, as a contractor, you deliver to your customers software which is overly general and reusable, you will not be able to get the next job from them — because they will not need a next job! If reuse is to succeed on a large scale, incentives must be found to reward companies that develop reusable products and, within a company, to programmers whose work exhibits good reusability.

There are also serious organizational issues. The best reusable components in the world are useless if nobody knows they exist or if it takes a long time to obtain them, or if they cost too much. The practical success of reusability techniques requires the development of adequate databases of software components, which may be searched by appropriate keywords so that a potential user will find out quickly whether some existing component satisfies a particular need. Network services must also be available, allowing electronic ordering and immediate downloading of selected components. A price structure must be found to allow users to try out components at a low price, but reward component manufacturers fairly in case of widespread and repeated use.

The "Not Invented Here" complex is well known. What it means in practice for reusability is that reusable components should bring a significant advantage over home-brewed ones in terms of quality, ease of access and cost. A marginal advantage will not be enough to convince programmers to use somebody else's mousetrap rather than invent their own. […] Important as these questions are in the long term, it is a mistake, in my opinion, to concentrate on them in the current state of the technology. Organizational solutions to the reusability problem are only meaningful if you are starting from the appropriate basis. Today, however, the main roadblocks are technical: we simply do not have the appropriate notion of module. [...]

3.1.3 Change and constancy

The technical difficulties of reusability become apparent if one takes a closer look at the nature of repetition in software development. Such an observation leads to the realization that although programmers do tend to do the same things over and over again, these are not exactly the same things. If they were, the solution would be easy, at least on paper; but in practice so many details may change as to render moot any simple-minded attempt at capturing the commonality. [...]

Such is the software engineer's plight: time and time again composing a new variation that elaborates on the same basic themes.

Take table searching again. True, the general form of the code is going to look the same each time: start at some position in the table T, then start exploring the table from that position, each time checking whether the element found at the current position is the one that is being sought, and, if not, moving to another position. The process terminates when either the element has been found, or all the interesting positions of the table have been unsuccessfully probed. […] The problem of coming up with a general software element for searching is apparent here. Even though the pattern is fixed, there remains a considerable amount of variation: what type of table elements you are dealing with (ELEMENT), how the initial position is selected (INITIAL_POSITION), how you proceed from one position to the next (NEXT), and so forth.

Source: Bertrand MEYER *Object-oriented Software Construction* (p 27-29)
Prentice Hall International Series in Computer Science (1988)