electronics

MDPI

Article

VoiceJava: A Syntax-Directed Voice Programming Language for Java

Tao Zan [1,*] and Zhenjiang Hu [2]

[1] School of Mathematics and Information Engineering, Longyan University, Longyan 364012, China
[2] Key Lab of High Confidence Software Technologies, Ministry of Education, Department of Computer Science and Technology, EECS, Peking University, Beijing 100091, China
* Correspondence: zan@lyun.edu.cn

Abstract: About 5–10% of software engineers suffer from repetitive strain injury, and it would be better to provide an alternative way to write code instead of using a mouse and keyboard and sitting on a chair the whole day. Coding by voice is an attractive approach, and quite a bit of work has been done in that direction. At the same time, dictating plain Java text with low accuracy through the existing voice recognition engines or providing complex panels controlled by the voice makes the coding process even more complex. We argue that current programming languages are suitable for programming by hand, not by mouth. We try to solve this problem by designing a new programming language, VoiceJava, suitable for dictating. A Java program is constructed in a syntax-directed way through a sequence of VoiceJava commands. As a result, users do not need to dictate spaces, parentheses, and commas, reducing the vocal load.

Keywords: voice coding; syntax-directed editing; code generation

Citation: Zan, T.; Hu, Z. VoiceJava: A Syntax-Directed Voice Programming Language for Java. *Electronics* **2023**, *12*, 250. https://doi.org/10.3390/electronics12010250

Academic Editors: Sanjay Misra, Robertas Damaševičius and Bharti Suri

Received: 28 November 2022
Revised: 27 December 2022
Accepted: 29 December 2022
Published: 3 Juanuary 2023

1. Introduction

A famous venture investor, Marc Andreessen, claimed that "software is eating the world", which is true that nowadays, as almost everything depends on software in our daily life. Software is written by software engineers. Unluckily, 5–10% or even more of them suffer from repetitive strain injury (RSI) [1]. RSI describes discomfort and pain in muscles, nerves, and tendonitis of the wrist or hand. Prolonged repetitive movements of the same body parts are eventually leading to an overuse of those body parts.

RSI is common among software engineers primarily because their wrists are either flexed or extended during keyboarding, and they spend extended hours sitting in front of a computer day in and day out. For programmers who suffer from RSI, it would be better to provide an alternative way to write code instead of a mouse and keyboard. Coding by voice sounds like an attractive approach, and quite a bit of work has been done in that direction [2]. At the same time, it is still difficult to be applied in the actual software development process since dictating in plain text has a low accuracy [3–5] and providing complex panels increases the coding complexity [6–8].

For example, suppose a programmer wants to dictate a simple statement such as:

```
( i < 1)
```

In VocalIDE [7], the programmer has to say: *"type open parentheses i space less than space one close parentheses"*

Coding by voice in this way takes work since it involves vocal and cognitive loads. When writing code by keyboard, only 7 keystrokes are needed, while it involves 11 words when speaking. It is worth mentioning that programmers have to speak the word *space* explicitly to indicate a space between words.

Desilets proposed a speech interface named VoiceCode [9], which lets users dictate pseudocode in a natural way, and then it is translated into actual code. For example, suppose the programmer wants to declare a method in Java [10]:

```
double m(int i){
  return i;
}
```

The programmer just needs to say: *"double define method m with arguments integer i body return i semi"*

Compared with VocalIDE, the programmer does not have to dictate *parentheses* and *space*. The sentence is matched with a method definition rule to generate Java code. On the one hand, if one sentence contains too much information, the programmer has to dictate the method name, arguments, return type, and body at the same time. They can dictate the simple sentence *"define method m"*, and the system generates a default value for the unmentioned parts. Then, they have to dictate many navigation commands to jump to the correct place for selecting and modifying them. On the other hand, dictating in that way is still unnatural as the whole sentence is not so meaningful. We prefer to say: *"define method m, whose return type is double, and it has one argument named i whose type is integer. In the body part, return variable i."* However, if we dictate this way, it involves too much of a vocal load.

We argue that current programming languages are suitable for programming by hand but not for programming by mouth. We try to solve this problem from the language perspective by designing a new programming language, VoiceJava, suitable for dictating. Unlike existing approaches that simply translate from text to text, we translate from text to an abstract syntax tree (AST). Program (from partial to full) written in our new language is parsed into an abstract syntax tree with possible holes for insertion (we call this AST VAST for short), which can be used to guide the voice coding process in a syntax-directed way.

In this paper, we mainly focus on the backend for voice programming, leaving AI-based voice recognition as future work. We believe that the better design of the language will lead to a better accuracy of voice recognition.

Our contributions can be summarized as follows:

- Much work has been done for dictating code, but existing methods all follow text-to-text patterns. Since they require a high voice load and due to the complexity and flexibility of dictating, they are still not good enough for actual usage. We argue that current programming languages are suitable for writing, not dictating. We try to tackle the voice-coding problem from the language perspective by designing a language called VoiceJava that is suitable for dictating.
- We introduce the idea of syntax-directed editing into voice coding to make the coding process structured. As a result, a program can be constructed step by step, which reduces the complexity of dictating.
- Our system has been implemented in Java, which contains more than 8000 lines of code and includes more than 200 test cases that guarantee the expressiveness and correctness of our system. VoiceJava's source code can be accessed online (https://bitbucket.org/lyun-prl/voicejava/) (accessed on 25 December 2022). To further prove the practicality of our system, we also implemented Java2VoiceJava (https://bitbucket.org/lyun-prl/java2voicejava/) (accessed on 25 December 2022), which translates all Java files in a project into a VoiceJava command sequence file by file. We translated the VoiceJava project from Java to VoiceJava, and running all the VoiceJava files can generate the VoiceJava project.

2. Motivation Example

Figure 1 illustrates the use of VoiceJava through syntax-directed programming. It shows the first 12 steps in developing a single Java program. In each step, the underscore shows the cursor position that indicates the insertion point, and a bubble on the right shows the command in VoiceJava.

- Step 1: Initially, there is an empty file with a cursor pointing to the first line. For a Java program, we could either define a package name, import a package, or define a class. For simplicity, we define a public class by command *define public class puppy*.

- Step 2: A class named `Puppy` is generated, and the cursor is pointed inside the class. Now, we can define a private attribute `puppyAge` by the command *define private variable puppy age*.
- Step 3: By dictating *type int*, we define `puppyAge`'s type as `int`. Since we may assign an initial value to `puppyAge`, the cursor is located at the right side of the equal sign.
- Step 4: If we do not want to give a value, we can move the cursor to the following line by dictating *move next*.
- Step 5: Now, we can define a setter for `puppyAge` by dictating *define public function set age*. A function named `setAge` is generated, and we need to define the return type for this function.
- Step 6: By dictating *type void*, we define `setAge`'s return type as `void`. The cursor moves to the arguments part.
- Step 7: We define an argument named `age` by the command *variable age*. The cursor is on the left of variable *age*, waiting to define the type.
- Step 8: We use the command *type int* to define its type as *int*. Since one function may need more than one argument, the cursor still sits inside the parentheses.
- Step 9: By dictating *move next*, we jump to the function's body part.
- Step 10: Now, we can assign an expression to variable `puppyAge` by dictating *let puppy age equal expression*, and the expression will be filled later.
- Step 11: By dictating *variable age*, we fill the expression part with a variable `age`
- Step 12: Since the definition for `setAge` is done, we just *move next* to jump out of the function.

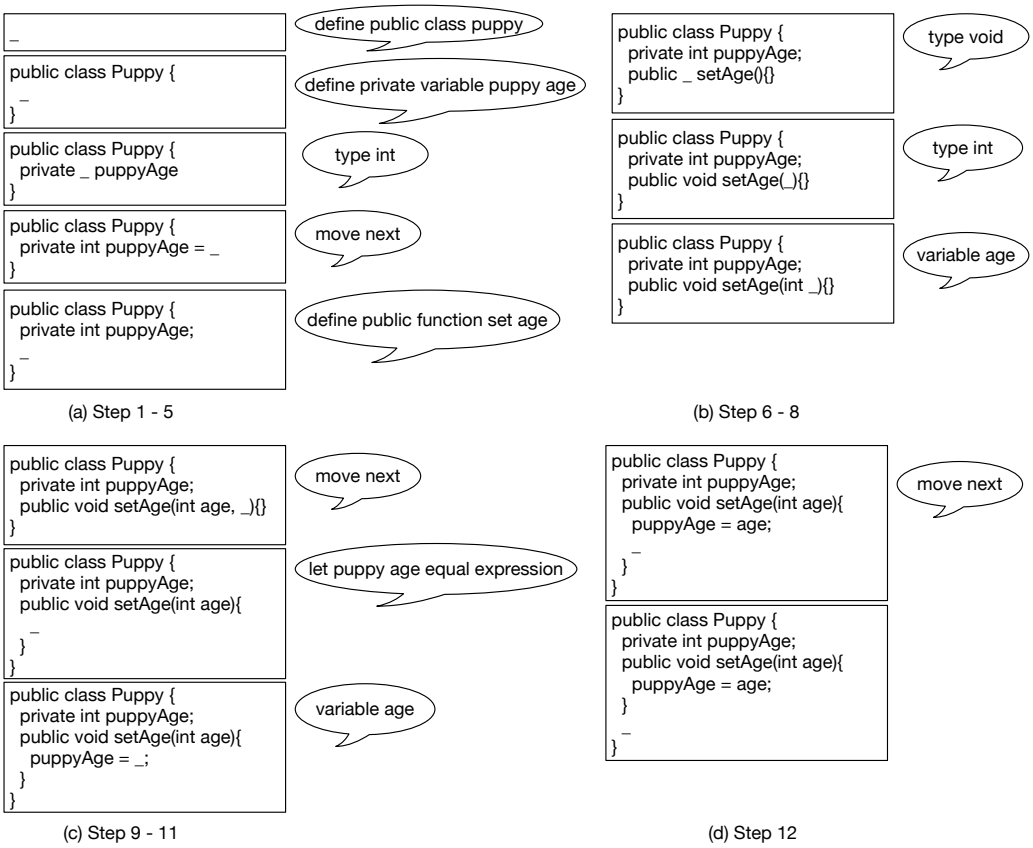

Figure 1. An example of the use of VoiceJava through syntax-directed programming.

Using VoiceJava helps us dictate actual code instead of dictating plain text. Our system handles miscellaneous details, including braces, parentheses, spaces, commas, and semicolons. The programming style in VoiceJava is well-structured, so that each time, we focus on one thing instead of mixing everything. For example, we do not support defining function name, return type, and arguments in one long command [10], which

makes dictating hard; we can dictate them step by step. The cursor can intelligently jump to the next insertion point through syntax-directed jumping, reducing the navigation load.

3. Overview of Language VoiceJava

In this section, we give a detailed explanation of our language constructs. However, due to page limitations, some of them are omitted.

3.1. Name

A name is a sequence of one or more words separated by space, and this is because a class/attribute/function/variable name is usually constructed by concatenated words. For example, if we want to define a function to say hello, we may name it `sayHello`. In this case, the name is a sequence of two words: `say` and `Hello`. When translating to Java, it will be concatenated in camel style. According to different contexts, we use different concatenating strategies. When it is a class or an interface name, the first letter of the first word is also capitalized; When it is an attribute, a variable, or a method name, it is not.

3.2. Package

In Java, normally, we need to define packages or import packages.

Example 1. For example, we can define a package by using the following command:

```
define package hello dot world
```

resulting in package hello.world in Java. The word `dot` represents the symbol dot (.) in Java.

Importing packages follows a similar syntax, except that `star` corresponds to the $*$ mark. For example, import hello dot world dot star will be translated to import hello.world.$*$;

3.3. Interface and Class

Interface and class declarations are similar, except for the use of different keywords (one uses `interface`, and the other uses `class`).

Example 2. For example, we can define a public class named `HelloWorld` by using the following command:

```
define public class hello world
```

It will be translated into Java code as public class HelloWorld {_}. The words `hello` and `world` are concatenated into `HelloWorld` in upper camelCase. The cursor is located between curly braces.

3.4. Basic Constructs

3.4.1. Method Declaration

A method can be declared by using the method declaration syntax.

Example 3. For example, we can define a public method named `sayHello` by using the following command:

```
define public method say hello
```

It generates Java code as: public _ sayHello(){}. The words `say` and `hello` are concatenated into `sayHello` in lower camelCase. The cursor is located at the return type of the method, so the programmer needs to dictate the return type for method `sayHello`.

Note that when defining a method inside an interface, the method has no body part.

3.4.2. Field Declaration

A variable or a field in a class can be declared by using a field declaration syntax.

Example 4. Suppose we want to define a private field named `count`; we can dictate the following command:

```
define private variable count
```

Then, it will generate the following Java code. The cursor will be at the type declaration part, and we need to define the attribute's type.

```
private _ count;
```

Example 5. If we simply want to define a variable, we can omit dictating the word `private`. Then, it will generate a variable declaration.

```
_ count;
```

3.4.3. Type Declaration

Type declaration defines the attribute/variable's type, the function's argument's type, or the function's return type.

Example 6. Primitive types can be defined easily by simply dictating the type's name. For example, we can set `count`'s type to int by type int.

```
private int count = _;
```

After setting the type, the cursor will move to the right side of the assignment operator since we may initialize the variable `count`.

Example 7. Types can be complex, and the following example defines a new type named `NodeList` parameterized with `Name` and `Integer` (NodeList<Name, Integer>).

```
type node list with name and integer
```

Note that the words `node` and `list` are concatenated into a new type `NodeList`. What if we want to define a node list? The following example shows the syntax.

Example 8. We can define a node list (Node[]) by using the `list of` keyword.

```
type list of node
```

What is more, we can even parameterize `Node` with arguments (Node<Name, Integer >[]) by the command: type list of node with name and integer.

3.5. Expression

An expression contains a null expression, an attribute access expression (expression a dot b represents a.b), a function call expression (expression call sum, expression a call sum represent sum(_) and a.sum(_), respectively), a primitive value (expression int five represents 5), a self-increase/decrease expression (expression a plus plus represents a++;), a subexpression which creates a pair of parentheses, a conditional expression (*?:*), a not expression, an array index expression, a binary operation expression, a let expression, and a return expression.

3.5.1. Let Expression

The let expression generates an assignment expression using the `let-equal` syntax. The `expression` represents a hole that will be filled later by an expression.

Example 9. For example, if we want to assign zero to variable `counter`, we can use the following commands:

```
let counter equal expression
int zero
```

It first generates a Java expression (counter = _;), and then zero is inserted into the cursor position as counter = 0;.

3.5.2. Return Expression

The command return expression generates a return statement (return _;).

3.5.3. Binary Expression

A binary expression is constructed by dictating the binary operator and the structure of the expression first and then dictating the left/right expression consecutively.

Example 10. For example, suppose we want to define a Java expression 2*(sum()+counter ++). We define a binary expression with a multiply operator (times), then set the left expression as two. After that, the cursor jumps to the right side of the multiply operator. It generates a pair of parentheses, and the cursor is located between them using subexpression. Again, we define a binary operation with the (plus) operator. We set its left-hand expression as `sum()` and right-hand expression as `counter++`.

```
expression times expression
int two
subexpression
expression plus expression
call sum
move next
counter plus plus
```

Note that we use move next after call sum. Because `sum` may need arguments, the cursor is located between the function call's parentheses. Thus, we can use move next to jump to the next insertion point.

3.6. If Statement

An if statement is defined by dictating define if, and a full if statement is constructed by a combination of other constructs. Let us illustrate how to define an if statement by an example.

Example 11. We first define an if expression with only one branch.

```
define if
expression greater than expression
variable i
int ninety
expression counter plus plus
```

It will be translated into the following Java program. The cursor is at one new line below `counter++;`, which means we can insert more expressions.

```
if(i > 90 ){
  counter++;
  _
}
```

Now, if we are dictating move next, the cursor jumps out of the body part, and a new else branch is generated. The cursor is located at the condition part, which expects a Boolean expression.

```
if(i > 90 ){
  counter++;
} else if(_){}
```

Now, if we again dictate move next, the cursor skips the condition and jumps to the body part.

```
if(i > 90 ){
  counter++;
} else { _ }
```

Finally, if we dictate move next once more, the cursor jumps out of the body part, resulting in an if statement with only one branch.

```
if(i > 90 ){
  counter++;
}
```

Here, we omit the explanation of the for/while/switch statements as they all follow a similar pattern.

4. Implementation

Figure 2 illustrates the architecture overview, i.e., the computation flow from the VoiceJava command text to the final Java code. A valid command text will always be matched to a command pattern defined in Figure 3, resulting in a command pattern instance. A command pattern instance generates a corresponding AST fragment, which is inserted into a Java AST in a syntax-directed manner. Finally, Java code is generated from the Java AST.

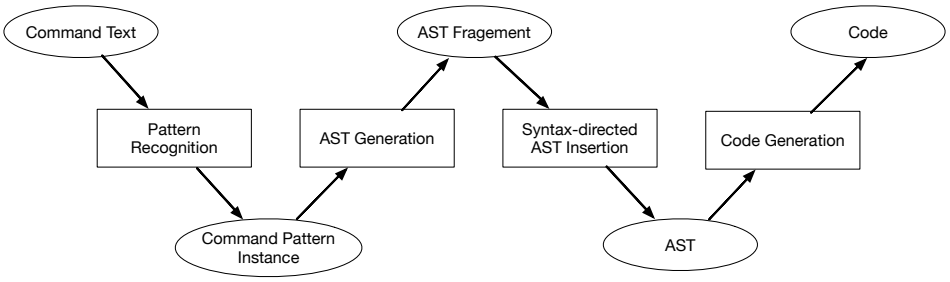

Figure 2. Architecture overview.

```
CommandPattern ::=
  | define package Name [dot Name]*
  | import static? Name [dot [Name|star]]*
  | define ClassKeyword interface Name
  | define ClassKeyword class Name [extends Name]? [implements Name]?
  | define MethodKeyword method Name [throws Name]?
  | define AttributeKeyword? variable Name
  | type (list of Name| Name [dot Name]* [with Name [and Name]*]?) [extends
      Name]?
  | define [enhanced]? for | define [do]? while | define if | define switch |
       break | continue
  | define try catch |throw new Name | break | continue | new instance Name [
      dot Name]*
  | let Name [dot Name]* equal expression | return expression | subexpression
  | expression? null | expression? call Name | expression? Name [dot Name]* [
      call Name]+
  | expression? Name [dot Name]? | expression? variable Name
  | expression? (int | byte | short | long | char | float | double | boolean)
       Name
  | expression? Name plus plus | expression? Name minus minus | expression?
      plus plus Name
  | expression? minus minus Name | not expression
  | expression? expression (OP | Compare) expression
  | expression? variable Name index Name | expression? string Name
  | conditional expression | lambda expression | cast expression
  | move next | move next body | move next statement
Name ::= [word]+
ClassKeyword ::= annotation | public | protected | private | static | final |
     abstract | strictfp
MethodKeyword ::= ClassKeyword | synchronized | native
AttributeKeyword ::= Annotation | public | protected | private | static |
    final | transient | volatile
OP ::= plus | minus | times | divide | mod
Compare ::= less than | less equal | greater than | greater equal | double
    equal | and | double and
```

Figure 3. Overview of language syntax.

4.1. Pattern Recognition

A command pattern contains a sequence of units. A unit can be a keyword that is used to match with a specified keyword such as `define, package`; A unit can be any that is matched with an arbitrary word that corresponds to an underscore (_) in a regular expression; A unit can be a slightly complex regular expression with type or (|), plus (+), question (?), or an asterisk(∗); A unit can be composed of a product of units; a unit can be a list of units.

```
Unit ::= KeyWord keyword | Any
       | Or Unit Unit | Plus Unit
       | Asterisk Unit | Question Unit
       | Product Unit Unit | List [Unit]
```

For example, the command pattern for defining a package define package Name [dot Name]∗ is represented by a unit list:

```
{
 name: "package",
 units: [Keyword "define", Keyword "package", Plus Any, Asterisk
     (Product (Keyword dot) (Plus Any))]
}
```

Note that the name, a list of words, is represented by `plus Any`.

Each command pattern is translated into an NFA graph, and all command patterns form a set of NFA graphs. A given command text is matched with the set of NFA graphs from one to the last until it is successfully matched with one NFA graph. A command pattern instance is constructed through matching, which is used in the next step for generating the AST fragment.

For example, the command text define package hello dot world generates a command pattern instance

```
{
 name: "package",
 units: [Keyword "define", Keyword "package", Any, Keyword "dot",
      Any]
}
```

The attribute `name` indicates which pattern has been matched, and `units` is used for generating AST fragments. `Any` unit stores the word information, the first `Any` stores `hello` and the second `Any` stores `world`.

Note that the order of patterns is essential during matching since a name can contain the keyword. The command pattern expression? variable Name index Name shall be put before expression? variable Name. For example, we want to produce hello[i] by dictating the command text variable hello index i. If it matches the second command pattern, we get helloIndexI. If the programmer wants a variable named helloIndexI, they can dictate hello first, then change it to helloIndexI by editing. We can improve this part by inferring context. For example, if a variable hello exists and it has an index functionality, then we prefer to generate hello[i]; otherwise, we generate helloIndexI.

4.2. AST Fragment Generation

For a given pattern instance, a corresponding AST fragment is created. We use an open-source project named JavaParser [11] to construct the AST. JavaParser is a tool that can be used to parse/analyze/generate Java source code, which provides specific APIs for our usage. Most of the AST fragments construction is relatively straightforward, so we just explain some essential constructs.

4.2.1. Name Construction

In Java code, a name can be a simple name such as `hello`, or a qualified name such as `hello.world`, or even using a star such as `hello.world.*`, which is used in import declaration. At the AST level, a name [12] can be a simple string or a string qualified by a qualifier which is also a name.

For a given unit list, we remove all units whose keyword is `dot`, reverse the list, and finally create a name recursively. For example, `hello.world.*` is represented as Name(Name(Name("hello"),"world"),"*").

Note that a preprocessing that concatenates words has been done before this step. For example, for the command my puppy dot age, `my` and `puppy` have been concatenated into `myPuppy` before the AST fragment generation.

4.2.2. Type Construction

Types can be a simple primitive type, list type, name, qualified name, or parameterized type. For primitive types, we directly call JavaParser's builtin function *parseType* to return the exact type from a string; For a list type (by command type list of name), we return an ArrayType whose item type is constructed from `name`; For a name, qualified name, and parameterized type, we first create a string representation of the type, then we call JavaParser's built-in function *parseClassOrInterfaceType* to return the exact type.

For example, if we dictate type grade dot node with integer and name, we manually create a string representation `Grade.Node<Integer, Name>` and then pass it to function *parseClassOrInterfaceType* to return a type that can be used in the AST.

4.2.3. FieldDeclaration and VariableDeclarator

The programmer can dictate a public field counter by define public variable counter or a variable counter by define variable counter. FieldDeclaration and VariableDeclarator share the same dictating command pattern, except VariableDeclarator does not need modifiers(`AttributeKeyword`). A FieldDeclaration is constructed based on VariableDeclarator plus modifiers.

4.3. Syntax-Directed AST Insertion

A generated AST fragment is inserted into a Java AST in a proper place guided by a special AST, which we call a VoiceJava AST that contains a special tree node to represent the next insertion point, and a path can be computed from a VoiceJava AST to traverse on a real Java AST to find the insertion point. JavaParser [11] provides useful APIs for us to insert AST fragments into an AST conveniently, so we can focus on essential points such as finding insertion points in a Java AST, implementing different insertion strategies according to the context and clever navigation.

4.3.1. Construction of VoiceJava AST

In VoiceJava AST, a node stores its type (`NodeType`), points to its parent, and a list where each item points to a child node. As described in the previous section, a VoiceJava AST always contains a node (holeNode) that is the next insertion point. When the attribute `isHole` is true, it indicates the node is a hole. `nodeTypeOptions` is used to store possible `NodeTypes` for this node that will be inserted. `NodeType` corresponds to AST fragment classes and command patterns. We also define two special `NodeTypes` named `Undefined` and `Wrapper`. `Wrapper` is used to mark an intermediate node, and `Undefined` is used to mark the holeNode. `parent` is helpful when we need to traverse back from a child to a parent.

```
public class VNode {
  private NodeType nodeType;
  private VNode parent;
  private List<VNode> childList;
  private NodeType[] nodeTypeOptions;
  private boolean isHole;
}
```

From the AST's point of view, initially, we have an empty Java AST denoted as a tree node with nodeType *CompilationUnit* (Each Java file denotes a compilation unit.). The tree node only has one child that is a hole (we call it holeNode) whose nodeType has yet to be decided. This child node can be either PackageDeclaration, ImportDeclarations, or TypeDeclarations denoted by nodeTypeOptions shown in Figure 4 (1).

Therefore, we can dictate a voice command to declare a package, import a package, define an interface, or define a Java class.

After dictating define package hello, the AST fragment for this voice command is inserted into a Java AST by calling the *setPackageDeclaration* function from the JavaParser library. At the same time, the first child's type is set as *PackageDeclaration*, and a new holeNode is created as shown in Figure 4 (2).

Then, the system expects the programmer to input either an ImportDeclaration or a TypeDeclaration. They can dictate import java dot until dot star, and the AST fragment for this voice command will be inserted into the AST by calling *setImport* function. The second child's type is set as *ImportDeclarations* in the VoiceJava AST, and another node whose type is *ImportDeclaration* is created under the second child. Since the programmer can import more than one package, the system still expects them to input an ImportDeclaration, and a new holeNode is created under the second child, as shown in Figure 4 (3).

Instead, if the programmer wants to define a class, they can dictate move next to move to TypeDeclarations as shown in Figure 4 (4). This voice command only affects the VoiceJava AST by deleting the possible insertion node under the ImportDeclarations node and setting the TypeDeclarations node as the next possible insertion node.

Finally, the programmer can create a public class `HelloWorld` by dictating define public class hello world, which creates a TypeDeclaration node under a TypeDeclarations node in the VoiceJava AST. As class can be further defined and a BodyDeclaration Node is created in the VoiceJava AST, which indicates the next insertion point as shown in Figure 4 (5).

A VoiceJava AST is constructed step-by-step through voice commands. A VoiceJava AST looks similar to a Java AST but is more abstract, as it hides many details such as package name and class name.

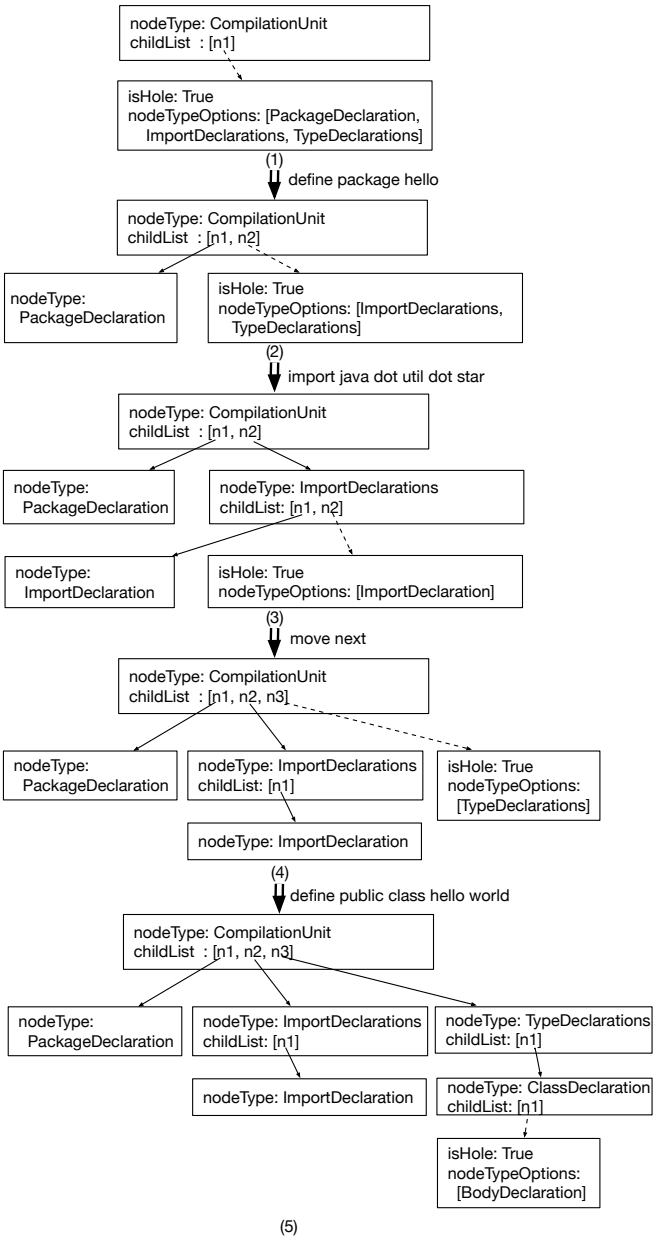

Figure 4. An example of implementation for a VoiceJava AST.

4.3.2. Path Computation

In a VoiceJava AST, a holeNode indicates the next insertion point. We first compute the path from the root node to the holeNode in the VoiceJava AST, which is used when traversing the Java AST. For example, in Figure 4 (5), the path is `ComputationUnit -> TypeDeclarations -> ClassDeclaration -> BodyDeclaration(holeNode)`.

The path is represented by a list that stores all on-path nodes' indexes in its parent's child list so that it is `[2,0,0]`. Node `TypeDeclarations` can be computed from node `CompilationUnit` by retrieving the third child from `childList`, and so on.

4.3.3. Insertion Point Retrieval

We traverse down from the root to the parent node of the insertion point according to the path in the Java AST. Function *getParent* accepts a Java AST (`parent`), a VoiceJava AST

(`vParent`), and a `path`, computes a pair whose first element is an Either-typed value, and the second element is the insertion point's index under its parent. An Either type means the parent node in the Java AST may be one of two possible types, and it can be either a left type `Node`, or a right type `NodeList<?>`.

```
public Pair<Either<Node, NodeList<?>>, Integer> getParent(
    CompilationUnit parent, VoiceJavaAST vParent, List<Integer>
    path){
  int index;
  for(index = 0; index < path.size() - 1; index++) {
    VNode vParent = vParent.getIthChild(path.get(index));
    NodeType nodeType = vParent.getNodeType();
    int indexOfHole = path.get(index + 1);
    if (nodeType.equals(NodeType.Wrapper)) {
      continue;
    }
    String name = TypeNameMap.map.get(nodeType);
    Class parentClass = parent.getClass();
    Method method;
    try {
      method = parentClass.getMethod(name);
      try {
        NodeList nodeList = (NodeList) method.invoke(parent);
        if (indexOfHole < nodeList.size()) {
          parent = nodeList.get(indexOfHole);
        } else {
          return new Pair<Either<Node, NodeList<?>>, Integer>(
              Either.right(nodeList), indexOfHole);
        }
      } catch (Exception e) {
          try {
            Optional<?> optionalData = (Optional<?>) method.
                invoke(parent);
            parent = (Node) optionalData.get();
          } catch (Exception e1) {
            parent = (Node) method.invoke(parent);
          }
        }
      }
    ...
  }
  return new Pair<Either<Node, NodeList<?>>, Integer>(Either.left
      (parent), path.get(index));
}
```

Initially, variable `vParent` points to the VoiceJava AST's root node, and we get its on-path child by *getIthchild* function. Each vnode has a corresponding `NodeType`, and we maintain a mapping from `NodeType` to a JavaParser function so that we can work on a Java AST by JavaParser's functions. For example, `PackageDeclaration` maps to function *getPackageDeclaration*. Since *getPackageDeclaration* is just a name in string, we cannot directly call it. Function `getMethod` returns a callable method if it indeed exists in the class, then, by calling the `invoke` function, we can execute the method.

Since function `invoke` needs to specify a return type, we allow the return type to be one of three types: `NodeList`, `Optional<?>`, and `Node`, so we explicitly convert the result into one of three types. A try-catch statement wraps the conversion code in case the conversion fails.

We need to mention that `Wrapper` is a type used for skipping computation on a Java AST. When there is no mapping for the node's type, we use the type `Wrapper` to indicate that.

Using a for-loop to traversing down from the root to the insertion point, the following path returns the parent node in the AST and the child index under this parent node.

For example, giving a path `[2,0,0]`, it will first get a vnode with type `TypeDeclarations`, which maps to function *getTypes* in JavaParser. Executing function *getTypes* results in a list (`NodeList`) with only one element (`ClassDeclaration instance`); we get this element as the new parent. Since its nodeType is `Wrapper`, the latter computation is skipped, and the for-loop is also finished. The final result contains a pair: a class instance and index zero.

4.3.4. Move Next

The semantics behind move next is simple: delete the current holeNode, and insert a new holeNode at the parent of the parent node. However, sometimes the strategy differs and involves the Java AST manipulation. Let us recap Example 11, the first move next command triggers generation of an else-if branch, the second move next command triggers the adaptation of the else-if branch to a default else branch, the third move next command triggers the removal of the default else branch. We also provide move next body and move next statement for quick movements. For example, after defining a method, if we want to directly move to the body part as the method has no arguments, we can call move next body to jump to the body part directly.

4.3.5. Clever Move

Every time we add a new subtree to a VoiceJava AST, we insert a holeNode at the proper place. It is not as simple as a move next command that normally deletes the current holeNode and inserts a new holeNode at the parent of the parent node.

Let us look at an example first. The following Java program corresponds to the left VoiceJava AST in Figure 5. The holeNode is at the bottom of the AST, whose parent is a Wrapper, and the parent of the parent is a VariableInitializer.

```
public class HelloWorld {
  public void sayHello(String name){
    String greeting = name + _;
  }
}
```

After we dictate string hello, the holeNode under Wrapper is replaced by an Expr node. Now, we cannot simply set a holeNode under Wrapper, VariableInitializer, or even VariableDeclarator. We have to keep going back to the Statements node and then add a new holeNode as its second child.

This keep-going-back strategy is implemented by a *cleverMove* function. The idea behind it is pretty simple: a parent node is not insertable if it can have only one child or has already reached the maximum number of children.

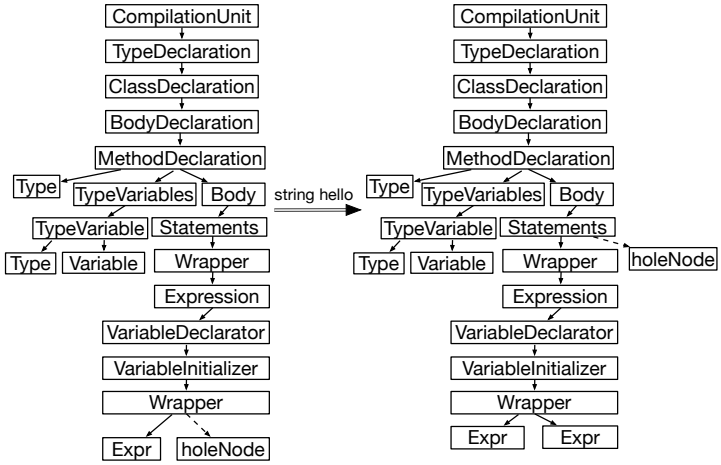

Figure 5. An example of clever move next.

For example, the VariableInitializer node, VariableDeclarator node, Expression node, and Wrapper node under Statements node (denotes a statement) can only have one child, the Wrapper node under VariableInitializer node (denotes a binary expression) can only have two children, so it will keep going back until reaching the Statements node. A new holeNode is inserted under this Statements node since it can have a list of Statements.

4.4. Code Generation

Each time an AST fragment is inserted into an AST, we call JavaParser's lexical preserving printer to print Java code into a file. In order to make voice coding practical, it would be better to design a specialized IDE to support cursor hinting, voice-based navigation, editing, and compilation. We leave this as our future work.

5. Experiments

To check the correctness of the implementation, we analyze each voice command pattern's possibilities and try to enumerate them all when writing test cases. For example, given the command pattern define package Name [dot Name]$*$ which is used for defining a package when starting coding in a new Java file, a command instance for this pattern can have zero or more dot's in the command text. Two test cases are used to represent each case: Analyzing possibilities for the command pattern import static? Name [dot [Name | star]]$*$ used for importing packages is a bit complex; Analyzing the command pattern itself is not enough. The context also needs to be considered since a programmer may import packages with/without defining the package. What is more, a programmer usually imports more than one package, which also needs to be tested. Thus, there are eight test cases in total.

The command define for outputs a sketch of a for-loop shown as follows:

for (; ;) return;

Generating test cases for this pattern needs to consider all the possibilities of a for-loop's AST (i.e., the structure of the for-loop). The for initialization expression, conditional expression, and increment expression can be omitted, and the for initialization expression can be an assignment expression when the variable is already defined, so there are twelve cases in total. What is more, a for-loop statement can be inside a while/if/else-if/else/for statement as shown in Table 1, so there are 5 cases. In summary, there are 17 test cases.

Following this strategy, we write test cases for each command pattern. We have written more than 200 test cases, which also show the expressiveness of VoiceJava. For each test case, the input is a file suffixed with .voiceJava that contains a valid command sequence, and the expected output is a file with the same name but suffixed with .out as shown in Figure 6. We check the equality of the generated Java code from the input file with the expected output file by comparing the plain text after removing all spaces and newlines.

Table 1. The possibilities for a for-loop.

while() for(;;) return
if() for(;;) return
if() else if() for(;;) return
if() else for(;;) return
for(;;) for(;;) return

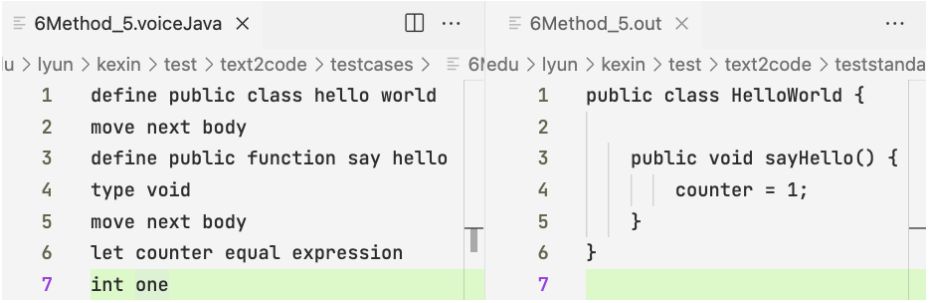

Figure 6. An example of .voiceJava and .out files.

Table 2 gives a summary of all test cases. The Cmd type means command type, and nums represent the number of test cases. For example, the package means a command related to defining packages. Since there are 18 types of expressions, the number of test cases for expression is more, up to 46.

Table 2. A summary of test cases.

cmd Type	nums	cmd Type	nums	cmd Type	nums
package	2	import	8	interface	3
class	8	method	5	field	9
type	7	argument	6	for	17
while	12	if	17	switch	16
subexpr	6	let	18	return	20
expression	46	instance	4	trycatch	2
move next	2	dot chain	12	continue	3

To check the practicality of our system, we implemented Java2VoiceJava, which translates all Java files in a project into a VoiceJava command sequence file by file. Then, we checked whether the Java files generated from the VoiceJava command sequence through our VoiceJava system were correct.

We used the VoiceJava project source code as the input and ran the generated .voiceJava files in the VoiceJava system. For example, Text2CompilationUnit.voiceJava was generated from Text2CompilationUnit.java by the Java2VoiceJava system. The Java file contained 4144 lines of code and the generated VoiceJava file contained 19,541 lines of commands. We ran all the commands every 0.5 seconds and output a Java file. A video that shows the generations of the first 1000 lines of code can be seen at https://www.youtube.com/watch?v=ex1vOSOFjG4 (accessed on 25 December 2022).

Table 3 shows the number of lines for the top 10 Java files in the VoiceJava project and the number of commands of its .voiceJava file, sorted in descending order according to the number of Java code's lines. Usually, the number of VoiceJava commands is three to five times as much as the number of code lines for the same Java file, which is sensible as one line of Java code is accomplished by more than one VoiceJava command.

Table 3. A comparison between Java and VoiceJava.

Filename	nums (.java)	nums (.voiceJava)
Text2CompilationUnit.java	4144	19541
Regex.java	286	654
HoleAST.java	261	905
Unit.java	193	512
Pattern.java	190	573
PrimitiveTypeAST.java	173	216
HoleNode.java	139	188
ListHelper.java	134	452
TypeAST.java	116	305
ASTManager.java	106	331

During the experiment, we found that the current naming convention in Java was not suitable for dictating, such as i, i1, Bi1, and a_1. We could support dictating variable i by using a meaningful word instead, which is a common practice in voice coding. For example, word air represented character a, bat represented b, and sit represented i. However, the dictating process was tedious and inefficient, so we used meaningful words instead of abbreviations, for example, initial instead of i and name instead of n.

6. Related Work

6.1. Plain Voice Coding

Desilets [13] proposed VoiceGrip for programming by voice, which let users dictate code using a pseudocode syntax that was easier to utter. However, the translation process from voice text to code was still at the text level. For example, if the user dictated *if current record number less than max offset then*, then *if* would be translated to *if(*, and *then* would be translated to *)\n{*. The most exciting idea was that directly dictating *currRecNum* was difficult; by dictating *current record number*, it was be translated into *currRecNum* in VoiceGrip. Desilets proposed an improved version named VoiceCode [9] in a short paper where the system was intelligent, but we could not find any detailed explanation materials or source code. Masuoka [10] added basic Java support to the VoiceCode system by defining command rules for Java.

Rosenblatt [6,7] developed a prototype system, VocalIDE, a web application that allowed users to write and edit programs using vocal commands. The voice recognition used the browser's built-in WebKitSpeechRecognition. As we mentioned in the introduction section, voice commands for text entries were plain text, and users had to explicitly dictate spaces and parentheses. The author proposed context color editing (CCE) that highlighted different parts with different colors, so the user could simply speak *select green* to select the code marked in green color.

Damien [5] implemented a browser-based system for dictating Java that recognized voice commands using PocketSphinx.js with a custom grammar. In order to increase accuracy, the system used two phases for the recognition: a structuring phase and a naming phase. In the structuring phase, the user only needed to dictate the structure of the code using a specific set of keywords, even for class/function/variable names and strings. The naming phase allowed the user to change the class/function/variable names and strings with real names.

Augusto [14] implemented a speech2code system for TypeScript. Voice commands were transformed into text using the Azure Speech to Text service, and the text was parsed to command, which was used to generate the corresponding code in text form. Our approach was different in that we generated AST fragments instead of plain text, which guaranteed the correctness of both AST fragments and full Java code at the AST level.

Lee [4] used an open-source voice recognition engine, Sphinx 4, with a custom grammar to increase the accuracy from 65% to 80–95%, and the recognized text was processed through JFlex/Cup software to judge whether it was valid Java code. While it was just a proof-of-concept implementation, the grammar was simple, and the programmer could only write a hello world Java program. Moreover, the user still needed to dictate parentheses.

A couple of Indian researchers have done much work by using speech recognition to write code. Patel [3] used Windows Speech Recognition (WSR) to convert voice to text. When facing a similar word, reserved keywords were converted into related keywords. Users did not have to dictate spaces between words but still had to dictate *function open/close* and *body open/close* to indicate parentheses. Several researchers [15–19] have also tried to design a Voice IDE for voice coding. For example, Parthasarathy [15] developed a web-based IDE for python and used the Houndify SDK [20] to generate text from voice. At the same time, they only showed server commands and did not mention how to dictate a complete Python program.

6.2. Syntax-Direct Editor for Voice Coding

Arnold [21] designed a generator for voice-recognition syntax-directed programming environments, which took as input a context-free grammar for a programming language. The syntax-directed editor provided code navigation based on the program's structure and supported the automatic completion of program constructs. Thus, users did not have to dictate spaces or parentheses. Our VoiceJava is a syntax-directed programming language motivated by Arnold's work.

Hubell [8,22] proposed a syntax-directed editor VASDE that popped up a dialog. Then, the user interacted with the dialog via speech instead of a mouse and keyboard. On the dialog, the button labels were speakable commands, and checkboxes were labeled with a unique character by speaking *select label* to select it. However, entering expressions was not supported in voice commands in that paper. Langan [23,24] proposed a Voice Expression Editor (VEE) model for expression editing that automatically numbered all the slots in the expression and highlighted the currently selected slot. When selecting a slot, simply dictating *select 2* selected slot number 2. This approach was similar to CCE proposed in Rosenblatt's VocalIDE [6].

6.3. Language Design

Begel designed a language called Spoken Java [25], along with a speech editor called SPEED [26], which used Nuance's Dragon NaturallySpeaking [27] for voice recognition. Spoken Java was syntactically easier to say than Java, which allowed users to speak in a naturally verbalizable way. For example, when defining a constructor *LinkedList*, the programmer could simply dictate *public linked list int value comma linked list next*, which is similar to Java. However, they did not need to consider concatenating words, parentheses, and function declarations. This sentence could be interpreted with six possible choices. If the user forgot to dictate *comma*, there would be 2,654,208 possible interpretations [26]. This flexibility caused much ambiguity. Begel presented an XGLR parsing algorithm to resolve the ambiguity and provide a list of possibilities for the user to select.

7. Conclusions

We argue that current programming languages are suitable for programming by hand, not by mouth. We proposed a new syntax-directed voice programming framework combining syntax-directed editing and programming language design. A new voice programming language could be generated for an existing programming language suitable for dictating. Compared with existing approaches, the construction of a program in a syntax-directed manner does not need the user to dictate spaces, parentheses, and commas anymore.

For future work, our research team has scheduled the following tasks. First, to reduce the number of dictated voice commands, compound commands can be defined. For ex-

ample, a let expression counter=0; uses two commands let counter equal expression; int zero; we can use one command let counter equal int zero instead. A variable declaration expression int counter also uses two commands define variable counter; type int, and we can also use one command define int variable counter instead. Second, we found that sometimes we wanted to dictate the following meaningful command directly instead of dictating the move next command. We could provide an intelligent insertion-point-finding algorithm by identifying the correct insertion point according to both HoleAST and the input command. Third, a voice-programming IDE that provides project management, code navigation, code editing, and code execution is also needed. We plan to explore the Zipper [28] structure for code navigation and editing. Fourth, in this paper, we mainly focused on the backend of syntax-directed voice programming, leaving AI-based voice recognition as future work. The recognition result can further be improved by inferring the context, such as defined local variables, library functions, and so on.

Author Contributions: Conceptualization, T.Z. and Z.H.; methodology, T.Z.; software, T.Z.; validation, T.Z. and Z.H.; investigation, T.Z.; writing—original draft preparation, T.Z. and Z.H.; writing—review and editing, T.Z. and Z.H.; All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Longyan Qimai Technology Innovation Fund grant number 2020LYF10007 and Ph.D. Start-up Foundation of Longyan University grant number LB2020010.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Van Tulder, M.; Malmivaara, A.; Koes, B. Repetitive Strain Injury. *Lancet* **2007**, *369*, 1815–1822.
2. Rajaselvi, M.; Jane Gloria, F.; Mohitha, V.; Selvarajan, G. A Survey of Programming Editors for the Visually Impaired. *Accessed Aug* **2021**, *12*, 01–08.
3. Patel, R.; Patel, M. Hands Free JAVA (Through Speech Recognition). **2014**, *5*, 4436–4439. Available online: https://ijcsit.com/docs/Volume205/vol5issue03/ijcsit20140503384.pdf (accessed on 25 December 2022).
4. Lee, H.; Fenwick, J.B., Jr.; Klima, R.E.; McRae, A.A.; Vahlbusch, J. Disability Assistive Programming: Using Voice Input to Write Code. Ph.D. Thesis, Appalachian State University, Boone, NC, USA, 2019.
5. Gonze, D.; Bonaventure, O. Coding with the Voice. Master's Thesis, Ecole Polytechnique de Louvain, Louvain-la-Neuve, Belgium, 2020.
6. Rosenblatt, L. VocalIDE: An IDE for Programming via Speech Recognition. In Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility, Baltimore, MD, USA, 29 October–1 November 2017; pp. 417–418.
7. Rosenblatt, L.; Carrington, P.; Hara, K.; Bigham, J.P. Vocal Programming for People with Upper-body Motor Impairments. In Proceedings of the 15th International Web for All Conference, Barcelona, Spain, 6–8 August 2018; pp. 1–10.
8. Hubbell, T.J.; Langan, D.D.; Hain, T.F. A Voice-activated Syntax-directed Editor for Manually Disabled Programmers. In Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility 2006, Portland, OR, USA, 23–25 October 2006; pp. 205–212.
9. Désilets, A.; Fox, D.C.; Norton, S. Voicecode: An Innovative Speech Interface for Programming-by-voice. In Proceedings of the CHI'06 Extended Abstracts on Human Factors in Computing Systems, Denver, CO, USA, 6–11 May 2006; pp. 239–242.
10. Masuoka, C. *Java Programming Using Voice Input: Adding Java Support to Voicecode*; University of Maryland at College Park: College Park, MD, USA, 2008.
11. JavaParser. Available online: https://javaparser.org (accessed on 26 March 2022).
12. Javaparser-Core. Available online: https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.2.5/com/github/javaparser/ast/expr/Name.html (accessed on 26 March 2022).
13. Desilets, A. VoiceGrip: A Tool for Programming-by-Voice. *Int. J. Speech Technol.* **2001**, *4*, 103–116. https://doi.org/10.1023/A:1011323308477.
14. Augusto, P. Speech-to-Code. Available online: https://pedrooaugusto.github.io/speech-to-code (accessed on 1 June 2022).
15. Parthasarathy, G.; Rangeesh, A.; Kishowr, V.S.S.; Sriram, R.; Vijay, S. An Approach to Accept Voice in Code Editor through Speech Recognition. *Int. J. Eng. Res. Technol.* **2018**, *6*. Available online: https://www.ijert.org/research/an-approach-to-accept-voice-in-code-editor-through-speech-recognition-IJERTCONV6IS07001.pdf (accessed on 26 March 2022).

16. Bhagavathsingh, B.; Srinivasan, K.; Natrajan, M.; et al. Real Time Speech Based Integrated Development Environment for C Program. *Circuits Syst.* **2016**, *7*, 69.
17. Shahane, A.; Kahate, A.; Gorad, T.; Sonawane, P. V-IDE: Voice Controlled IDE using Natural Language Processing and Artificial Intelligence. *Int. Res. J. Eng. Technol.* **2019**, *6*, 195–197.
18. Singh, A.; Tambatkar, G.; Hanwante, S.; Agrawal, N.; Hajare, R.; Khante, K. Voice to Code Editor Using Speech Recognition. *Int. Res. J. Eng. Technol.* **2018**, *5*, 389–390.
19. Borkar, S.; Kabra, R.; Kevadiya, S. SpeakEasy - Vocal Coding Online Platform. *Int. Res. J. Eng. Technol.* **2021**, *8*, 2204–2207.
20. Houndify. Available online: https://www.houndify.com/ (accessed on 26 March 2022).
21. Arnold, S.C.; Mark, L.; Goldthwaite, J. Programming by Voice, VocalProgramming. In Proceedings of the Fourth International ACM Conference on Assistive Technologies, Arlington, VA, USA, 13–15 November 2000; Association for Computing Machinery: New York, NY, USA, 2000; pp. 149–155. https://doi.org/10.1145/354324.354362.
22. Hubbell, T. *Voice-Activated Syntax-Directed Editing*; University of South Alabama: Mobile, AL, USA, 2005.
23. Langan, D.D.; Hain, T.; Hubbell, T.J.; Frøseth, J. A Voice-activated Integrated Development Environment for Manually Disabled Programmers. *Disabil. Rehabil. Assist. Technol.* **2008**, 3, 82–92. Available online: https://10.1080/17483100701343459 (accessed on 3 January 2023).
24. Langan, D.D.; Hain, T.F.; Camery, W.C. A Model for Voice-activated Expression Editing. *Comput. Inf. Sci.* **2008**, *1*, 2–11.
25. Begel, A. Spoken Language Support for Software Development. In Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing, Washington, DC, USA, 26–29 September 2004; pp. 271–272. https://doi.org/10.1109/VLHCC.2004.49.
26. Begel, A. Spoken Language Support for Software Development. Ph.D. Thesis, Electrical Engineering and Computer Sciences University of California at Berkeley, Berkeley, CA, USA, 2006.
27. Nuance Communications, I. Dragon NaturallySpeaking. Available online: https://www.nuance.com/dragon.html (accessed on 26 March 2022).
28. Huet, G. The Zipper. *J. Funct. Program.* **1997**, *7*, 549–554.