Automatically Generating Consistent User Interfaces with ATOM3

Ahmet Can Buyukdemir

University of Antwerp

Abstract

A problem in today's world is learning process of a version of an application that we are familiar with the older user interfaces. With the increment of versions, there are new features are added in many application every day. However as an average user of these applications, finding what is looked for can be challenging to find. This paper represents a system which user can use to transform and create a user interface which is consistent with the ones which he is familiar with. By finding the similarities between 2 interfaces, the new interface is transformed into an appearance which looks like older user interface. While doing this, the system also preserves the functionality of the new system. By doing so, a new consistent user interface is presented to the users, where they can find the similar functions and controls in the same place like their older experiences.

Keywords: Domain specific modeling, Automatic interface generation, User interface consistency, AToM3

1. Introduction

The devices that we are using every day are becoming complex every single day. They have more features and therefore more complex user interfaces. Due to the fact that complexity of the applications brings new and challenging user interfaces, users are the main actors who are suffering from this situation. Even if it is a really simple application such as remote controller of an air conditioner, a new version of the application may create a situation where users cant find how to set the desired temperature.

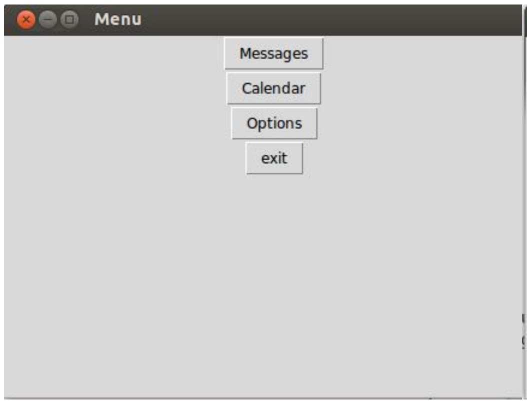


Figure 1: Version 1

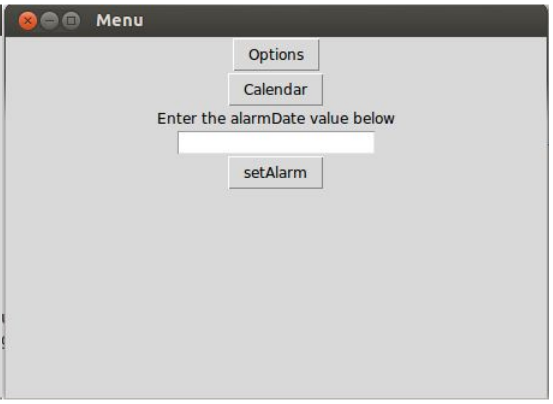


Figure 2: Version 2 without consistency

Considering the users previous experiences on the older versions of the same product, user may find the functionalities lot easier if the new version looks like old one. However this is not the case most of the times. Therefore answering the question "How is it possible to maintain the structure of the user interface even there are new functionalities in the new version of the application¿' may solve our problems from root. With the approach in this paper, it is going to be a big step on the solution of this problem.

In this paper, we are presenting a system, which is applied on AToM3, can create personally consistent user interfaces. Depending on the users previous experiences, our system takes the old interface as a base interface and modifying the new interface. After the modification, the new interface is going to be in the middle of new version and the old version. It is not possible to have a very exact appearance with the old version because of the different functionalities between the new and the old version, however the users are going to find the same functionalities in the same navigation step which it was used in the older versions.

The system of this paper presents is built on a domain specific language which defines functional and organizational structure of applications. By applying the specifications of each application with this language, similarities between the different versions are presented. Comparing these new specifications to the already experienced specifications, both the abstract and the concrete interface of the new application are modified. As it is looked like in

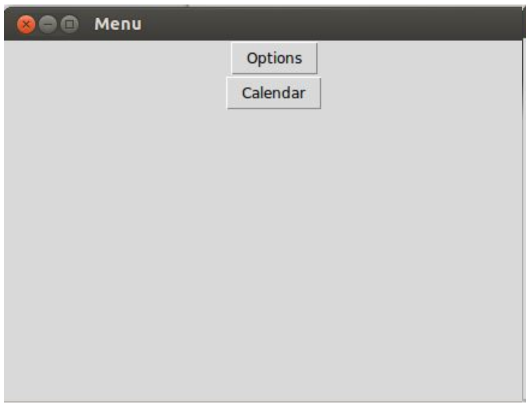


Figure 3: Version 2 with consistency

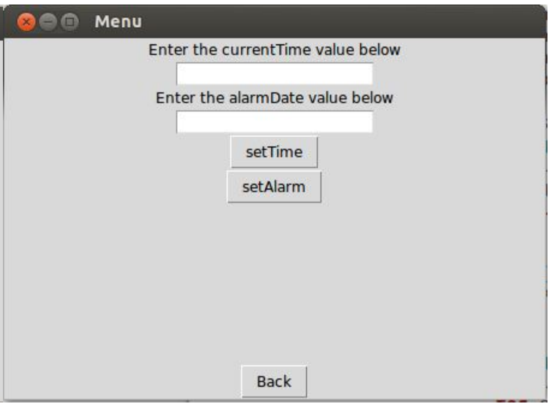

Menu
Enter the currentTime value below
Enter the alarmDate value below
setTime
setAlarm
Back

Figure 4: Version 2 with consistency:Options

Figure 1 and Figure 2, we can see two different versions of a same application. In the first application we can see that in menu we can find 4 different subgroup of the main menu. However in the second version, alarmDate variable and setAlarm command is in main menu, while it is not in version 1. In version 1 they are stored in the Options subgroup. So we can see an important consistency between two versions. This example is simple in order to show the consistency. However it might be more complex example than this one.

After we apply our approach, both abstract and concrete interface of version 2 is modified. (See Figure 3 and Figure 4) The alarmDate variable and setAlarm command in the main menu are replaced under Options subgroup. It also makes sense to ask the question Where is the Messages subgroup? just because it exists in Version 1. However as we can see, after version increases the application developers may have decided to remove this subgroup. Therefore it shows that the system is not making a comparison between different things in the application. The comparisons are done between similarities and then the necessary changes are mapped on the new interface. /newline

In order to use the user interface generation language which is applied in AToM3, the specifications of the application has to be clarified. After the progress of preparing the specifications of each version of the application is completed, then the user interface generation language presents an environment which the specifications can be applied on it. All these progress are going to end up with running the comparison of these two specifications and modifying one of them.

2. Related Work

There are many examples which can be given as a related works of this paper, because consistency is one of the greatest issues on interface generation. Every single day, new versions of the old applications or new applications are appearing. While designing their interface, interface designers should consider the potential users old experiences. By this way, they can provide a faster learning curve for the users comparing the users who are facing with inconsistent user interfaces.

One of the great examples and this papers source of inspiration is UNIFORM [1]. UNIFORM, is a system which creates personally consistent user interfaces for the applications. The consistency of the interfaces is provided by taking the users previous experiences as a base, just like the system of this paper.

The other example that can be given as a related work is, ITS [4]. ITS is a system which creates consistent user interfaces automatically, just like UNIFORM. ITS can work on different versions of the applications by using rule-based approach. Therefore, mapping operations are done by searching the patterns according to the rules. When the same condition is found in different versions , their approach is applied.

3. Design

3.1. Revealing the specifications

I have started on this project by understanding the specification format of the applications and how can be adapted into a domain specific language. The main role of this language is providing an environment to the user to implement the specifications by using it. By doing so user can translate the specification into a language where our system can understand. The very first step of this progress is, creating the specifications or finding it from internet. The specification creation process is handled by opening the application and navigating to ever single page, and finding its features. After it is completed, it can be implemented by using the language of our system.

3.2. The specification language

The specification language which is used is a simple language which has 3 usable classes. These classes are given as an attribute of the language, which

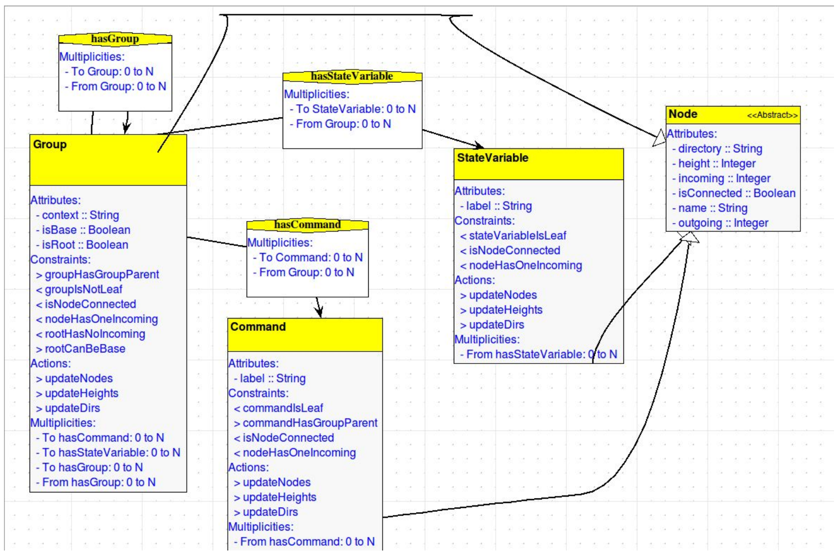


Figure 5: Class Diagram Formalism

user can use them while applying the specification which is prepared before. These 3 attributes are, Group, Command and StateVariable. For instance, a digital watch has alarm setting screen. Each different screen is defined as a Group object. Inside this screen we can set the alarm by setting the exact time. In order to do this, we need variables such as day, hour and minute. All these variables can be implemented as a state variable. The last object, Command is used for defining the functionalities of the application in this very specific screen. In this case we can say the button which is finishing setting alarm is a function call. So it can be defined as a Command in the language. Therefore by using these classes user can define the user interface very simple but in an effective and practical way.

In Figure 5, we can see the class diagram formalism of the language (abstract syntax). Here there is an extra class which is an abstract class Node. The subclasses are the classes which are defined in the paragraph above.

The language works in a tree structured way. Each root can be tree or selected as a base. Under the constraints of the language, there can be no more than 2 roots and only 1 root which represents the base user interface can be a base. Each object in the class has label and name attribute. Label

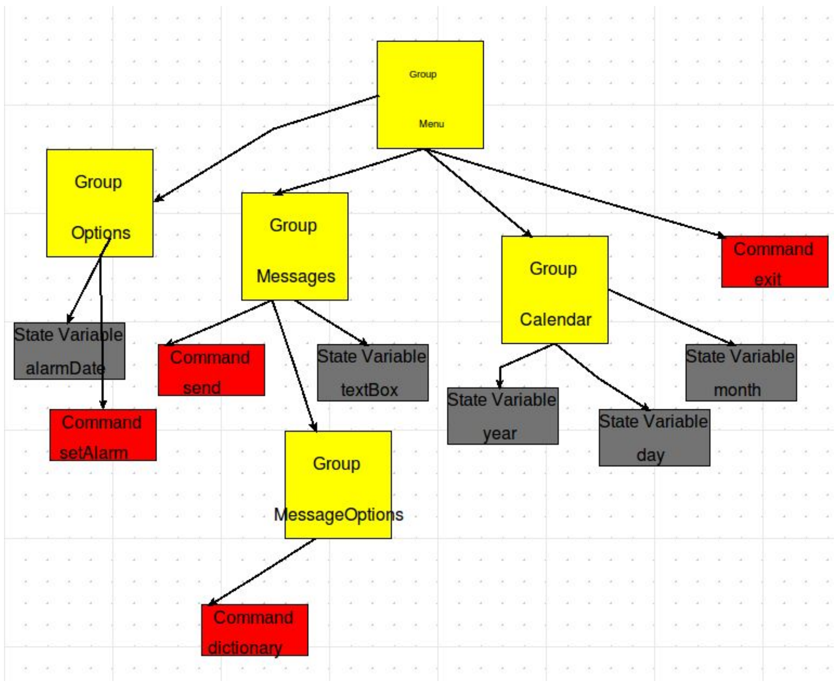


Figure 6:

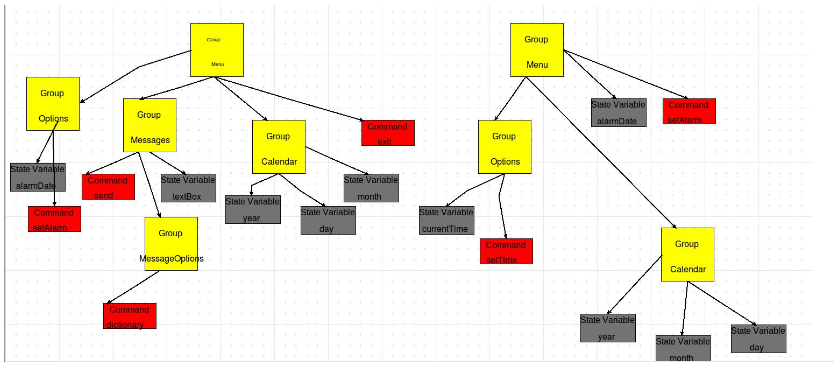


Figure 7:

attribute is for abstract interface and name attribute is for concrete interface. In Figure 6, there are 2 examples of different implementations. In Figure 6, user interface generation language is only used to create interface for 1 tree(1 application), however in Figure 7, we can see two different trees. One of these trees is base tree while the other one is not.

3.3. Constraints and Actions

3.3.1. Constraints

While designing this domain specific language, it was necessary to create constraints which draw some borders between language and user. By creating constraints, users are prevented to do some non-logical implementations. The following constraints are used:

- **atLeastOneRoot** This constraint is to check there is at least one root.
- **atLeastOneBase** This constraint to check there is at least one base root.
- **maximumTwoRoot** This constraint is to check if there is maximum two root in the system
- **atLeastOneGroup** This constraint is to check if there is at least one group in the system, because we need at least one screen such as an empty menu.
- **atMostOneBaseRoot** This constraint is to check if there is max 1 base root.
- **isNodeConnected** This constraint is to check if all nodes are connected
- **nodeHasOneIncoming** This constraint is to check if there is only 1 incoming edge to current node. If not, it returns False and prevents the current event.
- **rootCanBeBase** This constraint is to check if there are some groups who are assigned as base but not root.
- **rootHasNoIncoming** This constraint is to check whether the root nodes dont have incoming or not.
- **groupIsNotLeaf** This constraint is to check if groups are leaf nodes in the tree. If so, it returns false.

They are triggered in the event of such as SAVE or CONNECT. By doing so in almost every step, the abstract syntax is checked in the sense of correctness. If one of them returns False, it prevents the event to complete its action.

3.3.2. Actions

Actions functions are helping to maintain and update the object attributes during the editing period. By this way, when user completed the object creation and edition process, all the attributes of the objects are updated and ready to be simulated.
The following action functions are used:

- **updateNodes** This action function updates number of incoming and outgoing links of each node. The attributes incoming and outgoing are not used effectively during the programming. However they are ready to be used for the future. It is more than likely; they are going to play and important role in the maintenance of the specification trees.
- **updateHeights** This action function updates height attribute of each node. This attribute is used during the interface creation process. Navigation is handled by considering the height of the tree objects.
- **updateDirs** This action function updates and maintains the directory of each node. This directory attribute is used to satisfy the desired navigation. After each editing operation in the specification tree, directory attributes are checked if they need to be updated.
- **moveNodeUnderAnotherNode** This action function handling the moving operations in the specification tree. During the simulation, depends of the base tree, it may be necessary to change the placement of the nodes in the non-base tree. In this case, this action is called to reshape the non-base tree.
- **deleteEmptyGroups** : This action function handling the deletion of the Group objects which becomes a leaf node after some updates on the tree. This may be the case after each time an object is moved under another node.

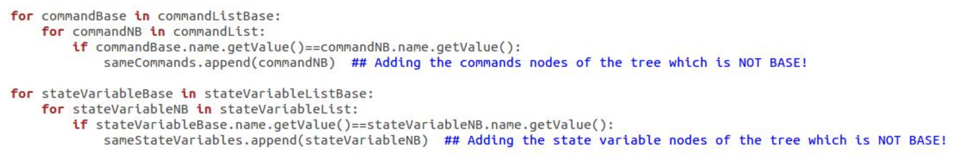
3.4. Architecture

As it explained in the Specification Language part, the design tree(s) has to be implemented by using the language. Depending on how many trees are there in the simulation, the output is going to differ. In the case of 1 design tree, the output is going to be representing a user interface which is based on the only design tree which was implemented alone. In the case of 2 design trees, the output is going to represent the non-base design tree by modifying it. There are 2 different python files that are playing the main role in the simulation process. These are Simulation.py and InterGen.py. Simulaton.py is the controller of the whole interface generation process. InterGen.py is another python file that generates the output file. The output file contains the generated user interface.
The architecture of the system has 3 different phases. These phases are called:

- **Mapping Phase** : This is the phase where similar functions and variables are identified.
- **Abstract Structural Phase** : This is the phase where necessary updates are done in the abstract representation of the new interface; such as moving nodes under another node.
- **Concrete Phase** : This is the phase where the concrete user interface is generated. This interface is going to have similar appearance with the base specification tree, depends on the results of first two phases.

3.4.1. Mapping Phase

The mapping phase is done comparing two different specification trees. Base tree functions and state variables are considered as base functions and state variables. Each of non-base tree functions and state variables are compared with the base group and state variables. If same functions or state variables are found, the system stores them inside of different lists. Most of the times, when there are some state variables are found similar between these two trees, then the commands under the same group is also a similar. However there are some cases which it doesnt happen always this way. For instance; sometimes variables in the software are not used after the command is invoked, but they may invoke related commands in case of their update. Such as, when they are changed, an event is raised and command is invoked. Therefore in these cases, our system only maps the state variables but not



```
for commandBase in commandListBase:
    for commandNB in commandList:
        if commandBase.name.getValue()==commandNB.name.getValue():
            sameCommands.append(commandNB)  ## Adding the commands nodes of the tree which is NOT BASE!

for stateVariableBase in stateVariableListBase:
    for stateVariableNB in stateVariableList:
        if stateVariableBase.name.getValue()==stateVariableNB.name.getValue():
            sameStateVariables.append(stateVariableNB)  ## Adding the state variable nodes of the tree which is NOT BASE!
```

Figure 8: The algorithm which is used in order to identify the similar functions and state variables

the commands.

3.4.2. Mapping Phase

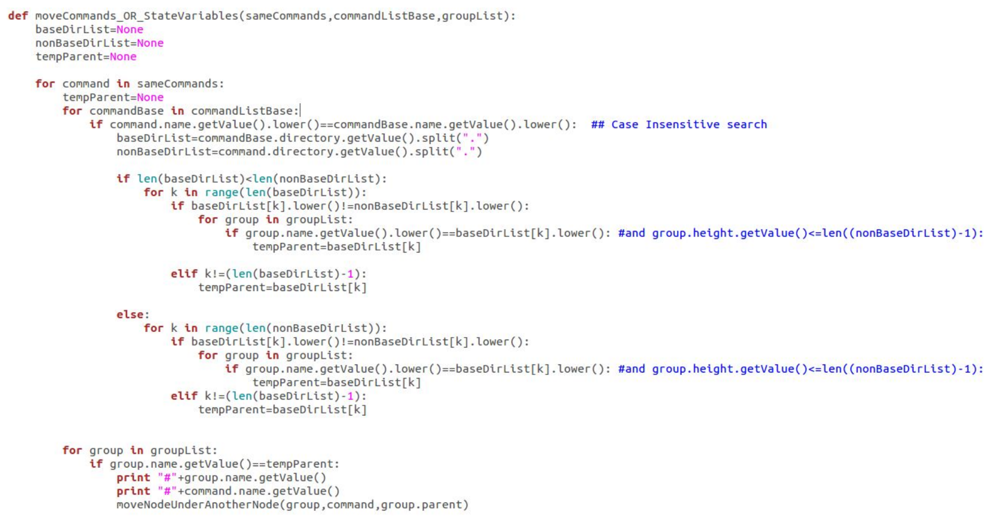
The goal of this phase is to ensure that similar functions and state variables are located in the same place in new interface. This goal is satisfied by using the information base that we achieved from the mapping phase. By using the lists of same commands and same functions, each of their directories are split into another lists called baseDirList and nonBaseDirList, so directories become easier to be compared. Here is an example how it works: Consider a radio player interface which is set as a base. Time options has the directory: Menu.Options.General.TimeOptions. However in another version of the radio player TimeOptions has the directory: Menu.Options.TimeOptions. In this case our algorithm is going to check whether there is a Group called General under Menu. Options in the second example. If it is found, then TimeOptions is moved under Menu.Options.General.

3.4.3. Concrete Phase

In this phase, InterGen.py plays the most important role. Depends on the abstract user interface, the parameters of the function start(), which is located in InterGen.py , changes. After start() function is invoked, another python file is created as an output. For visual elements, the library of tkinter is used. Using tkinter brings easiness to the interface design because it is simple to use and effective.

3.5. The Progress of User Interface Generation

After the building the specification trees are completed, users are allowed to press RUN. This button is invoking the simulate function in the Simulator class. This function is the controller of the whole simulation progress. First of all it puts every single node into related lists. Then it makes it ready for



```python
def moveCommands_OR_StateVariables(sameCommands,commandListBase,groupList):
    baseDirList=None
    nonBaseDirList=None
    tempParent=None

    for command in sameCommands:
        tempParent=None
        for commandBase in commandListBase:
            if command.name.getValue().lower()==commandBase.name.getValue().lower():  ## Case Insensitive search
                baseDirList=commandBase.directory.getValue().split(".")
                nonBaseDirList=command.directory.getValue().split(".")

                if len(baseDirList)<len(nonBaseDirList):
                    for k in range(len(baseDirList)):
                        if baseDirList[k].lower()!=nonBaseDirList[k].lower():
                            for group in groupList:
                                if group.name.getValue().lower()==baseDirList[k].lower(): #and group.height.getValue()<=len((nonBaseDirList)-1):
                                    tempParent=baseDirList[k]

                        elif k!=(len(baseDirList)-1):
                            tempParent=baseDirList[k]

                else:
                    for k in range(len(nonBaseDirList)):
                        if baseDirList[k].lower()!=nonBaseDirList[k].lower():
                            for group in groupList:
                                if group.name.getValue().lower()==baseDirList[k].lower(): #and group.height.getValue()<=len((nonBaseDirList)-1):
                                    tempParent=baseDirList[k]
                        elif k!=(len(baseDirList)-1):
                            tempParent=baseDirList[k]


        for group in groupList:
            if group.name.getValue()==tempParent:
                print "#"+group.name.getValue()
                print "#"+command.name.getValue()
                moveNodeUnderAnotherNode(group,command,group.parent)
```

Figure 9: The algorithm which handles the moving operations

the other functions and classes to use it.
The following is the summary of the whole UI generation process:

∗ Simulate function is invoked by pressing the Run button.

∗ Simulate function checks whether there are 2 different trees or not. Simulation is not going to start with more than 2 trees. There should be at least 1 tree and maximum 2 trees.

∗ If there is only 1 tree, start function in the InterGen.py is invoked with the objects of only tree.

∗ If there are 2 different trees, same commands and state variables are found and moving operations in the abstract user interface is done. After the modifications in the new user interface is completed, the new user interfaces node information are passed to the start of function of the InterGen.py

∗ InterGen.py generates another python file which contains the concrete user interface of the modified version of the non-base application. The output is ready to be run in python compiler.

4. Comparisons

Although the consistency of user interfaces is one of the biggest problems, there are not a lot of approaches to this solution. As it is mentioned under the Related Works section, UNIFORM[1] is one of the most successful approaches. Most of the things are very similar and this project is based on it. Even though the approaches are same, the tools are different. Such as, in order to create a domain specific language AToM3[2] is used , however in the project of UNIFORM, there is another language called PUC Specication Language[1]. However both of these languages have very common part such as the way that representating of abstract user interface. There are some other approaches which use rule-based transformations. By not doing to similar way, we could also indicate our difference between other approaches.

5. Future work

Although the goals of this project are fulfilled, there are some other features that can be added in the future. One of these goals is improving the capabilities of the language by adding new usable object to it, such as: one of the goals in future is, adding a class called ListGroup. Because the state variable representation in the current version of the language is little loose, by adding ListGroup, state variables can be grouped under ListGroup objects. One of the other goals is creating better looking interfaces. In this project, the appearance of the user interface was not the priority. Both concrete and abstract user interfaces can be improved and become better looking.

6. Conclusion

The goal of this project is satisfied by generating consistent user interfaces. With the older other projects, this project is going to be a resource for future researches about consistent user interface generation. By improving the skills of the current version, the project can be used in many areas where the applications have user interface.

7. References

[1] Nichols, J.,Myers Brad A.,Rothrock B. 2006. UNIFORM: Automatically Generating Consistent Remote Control User Interfaces, Carnegie Mellon University

[2] AToM^3, *AToM^3 AToM3Programming WebSite*,
http://atom3.cs.mcgill.ca/people/jlara/AToM3Programming/index.dtml

[3] Python, *Python Website*,
http://python.org

[4] Wiecha, C., Bennett, W., Boies, S., Gould, J., and Greene, S., 1990. ITS: A Tool for Rapidly Developing Interactive Applications. ACM Transactions on Information Systems