Rahul Rajagopalan
6.170

Assignment 1: Object Modeling

A: Background Reading

Entity Relationship Model – Entity-relationship models are used to represent data in a way that provides a schema that can be used to organize the data in a way that allows it to be stored in a database. The model is designed during the requirements analysis stage, and later mapped to the model that the database uses.

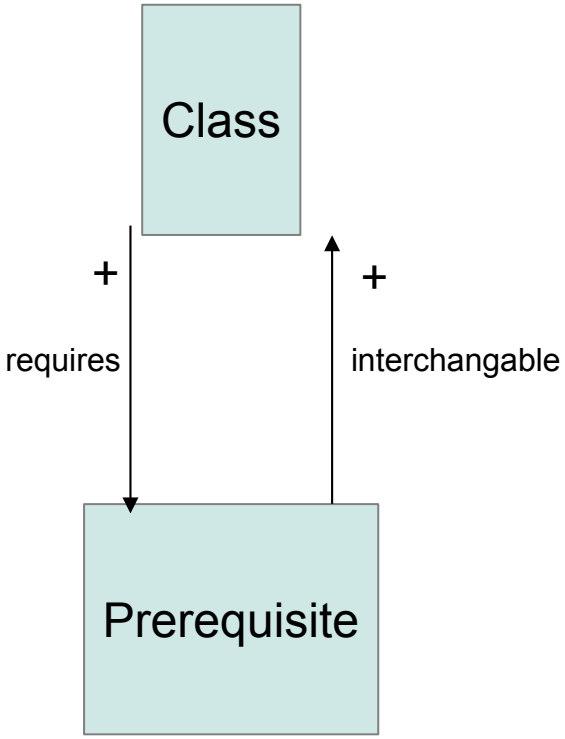
Object Modeling Technique – OMT is a language used to model the objects used in a software project written in an object-oriented style. The model is used to visualize the classes used in the program, and the messages (a.k.a. method calls) that they send to one another.

Software Analysis Patterns – The models used in software analysis patterns are meant to visualize the concepts involved in a software system (as opposed to the concrete objects). The pattern illustrates the proceesses in the project's logic and how they relate.

Unified Modeling Language – UML is a descendant of OMT and other modeling languages. It uses OM's to model the architecture of a software project, including objects, business processes, and the database's data model.

B. Conceptual Modeling Problems

1. Prerequisites



Ambiguities Resolved:

The term “prerequisites” wasn't defined; it was considered here to include not only classes but also other types of prerequisites (e.g. approval from a professor). So Prerequisite is not a subset of Class.

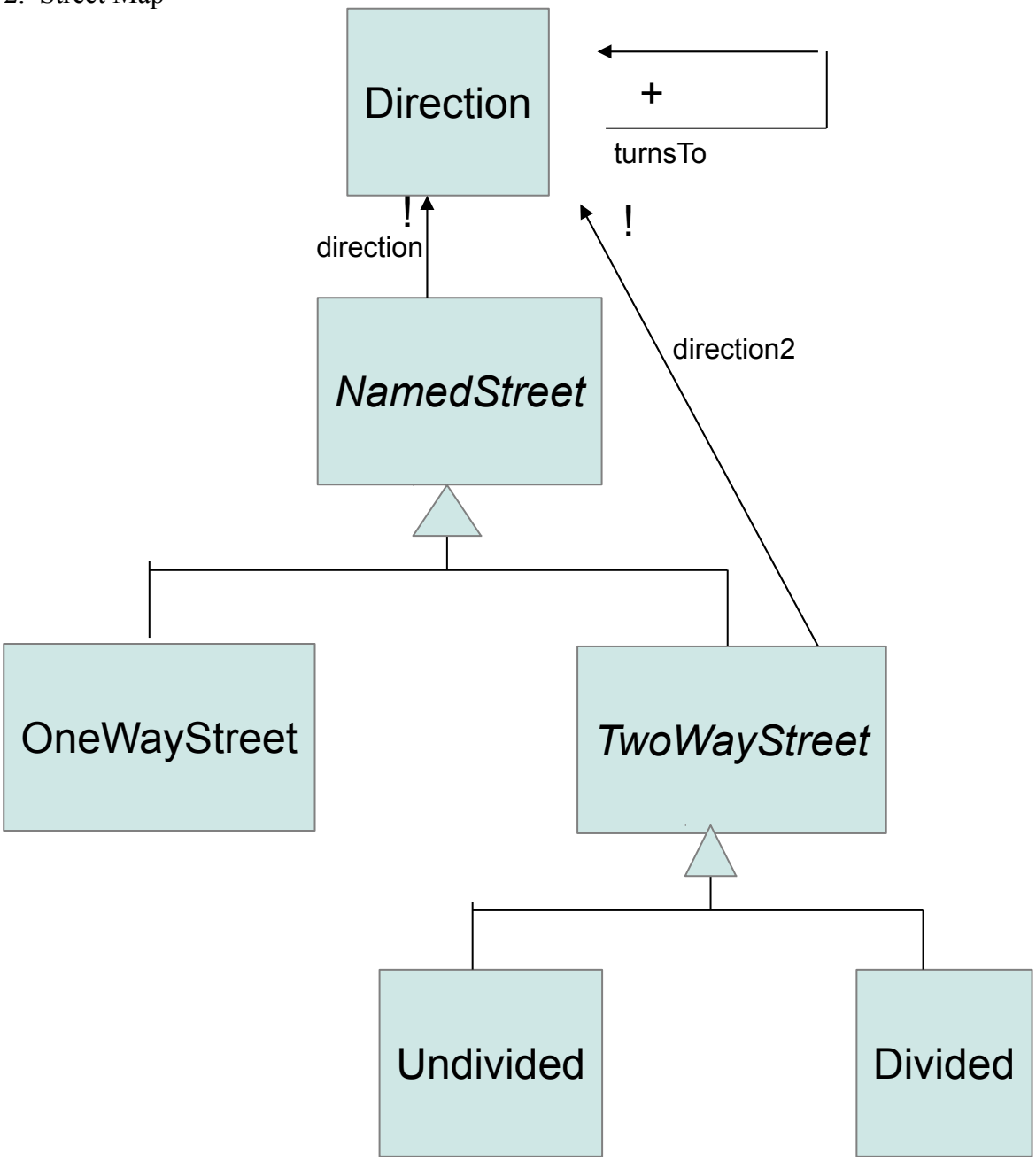
Complexities Ignored:

The concepts of corequisites and optional prerequisites were ignored since they weren't mentioned

in the question.

Designations:
"Class" refers to any course at a college that requires signing up for. "Prerequisites" can refer to any requirement that a given class has before a student will be allowed to take that class (other classes, recommendations, etc.)

2. Street Map

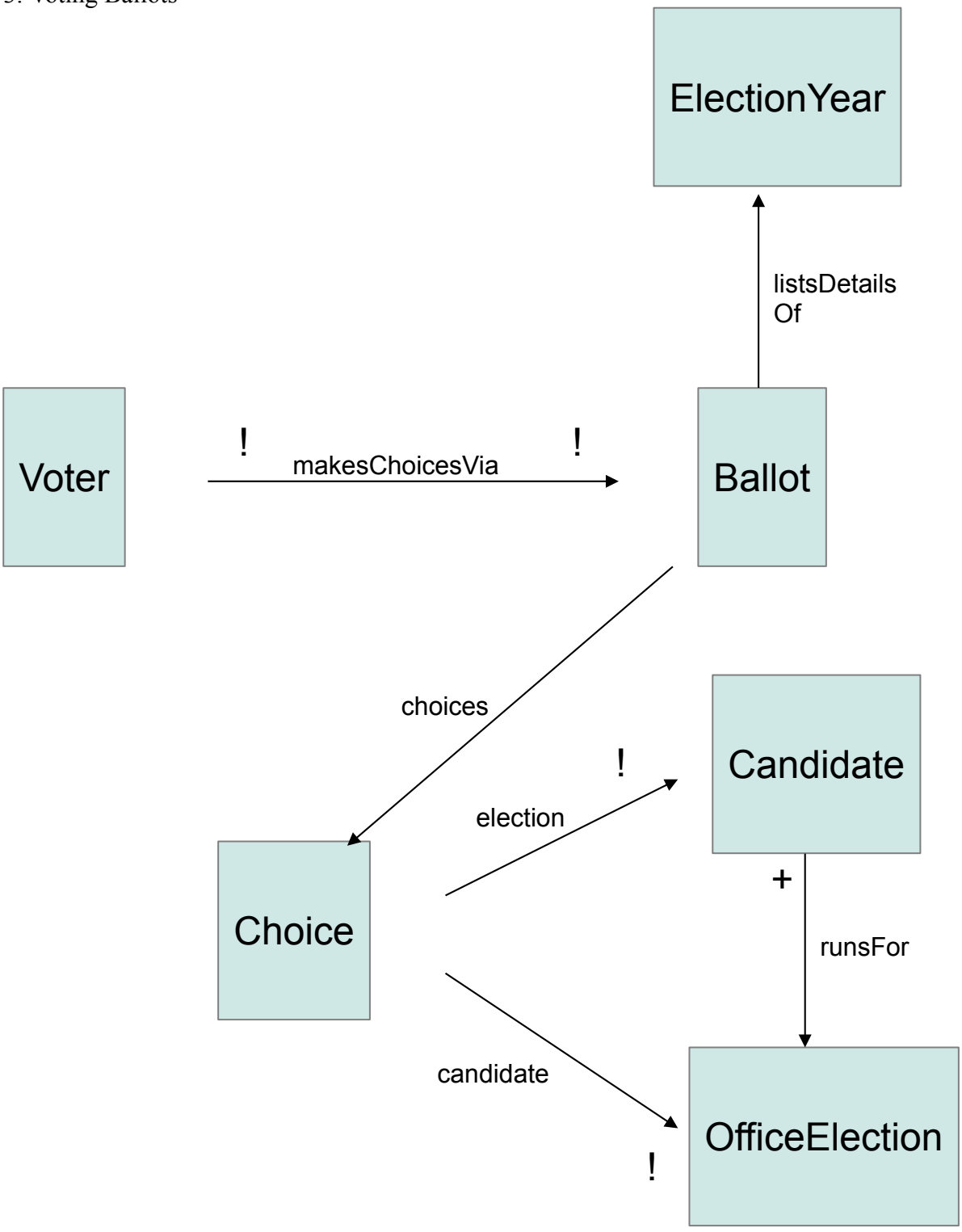


Ambiguities Resolved:
The term "intersection" was not explicitly defined. This model assumes that it is the connection between multiple directions, which may or may not be part of different named streets.

Complexities Ignored:
The possibility of unnamed streets (as seen in some countries) was ignored. This model also assumes that every direction has some intersection (there are no isolated dead ends; they would not be of practical use).

Designations:
"Divided" refers to the existence of a concrete barrier between the two directions of a two-way street (as opposed to a lane marking). Most highways fall under the Divided category.
"Directions" are the compass directions in which traffic on a street travels. All streets have at least one direction. Two-way streets have a second direction, which will be in the opposite direction. Traffic traveling in either direction will be considered to be inside the same named street.

3. Voting Ballots



Ambiguities Resolved:
Details about timing and the rules of the election were not given. This model assumes that there

are elections during various election years (even-numbered years in the US). It also assumes that a candidate can run for multiple offices during elections (e.g. Senators and governors running for President after getting more years of experience). Finally, it assumes that a person can act as both a voter and a candidate within the same election; a candidate can vote for themselves and also vote in elections for other offices.

Complexities Ignored:
This model considers the details of determining the results of an election to be out of the problem's scope. It also ignores the various implementation details involved in holding an election. Finally, it ignores referenda – elections where the population directly votes on whether a given law should be passed. (These could be considered a special case in which the candidates are Yes and No).

Designations:
"OfficeElection" refers to the contention of multiple candidates for a single office. A ballot typically displays multiple such elections being held in parallel.
"ElectionYear" refers to the year in which an election is occuring. Elections recur periodically.
"Choice" is a tuple referring to picking a certain candidate for a certain office, which is what a voter decides.

4 Java types
(Diagram on next page because of spacing constraints)

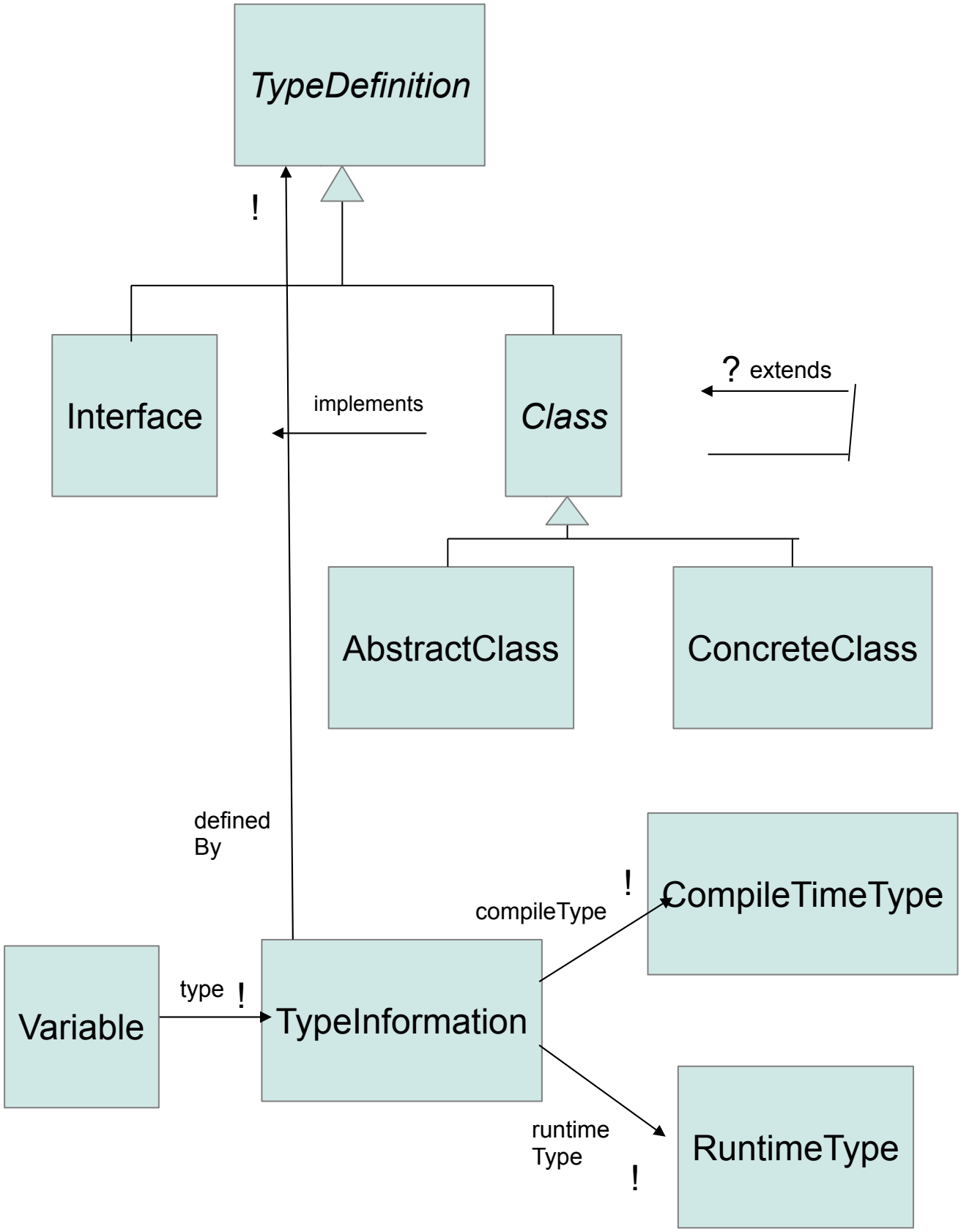
Ambiguities Resolved:
This model specification was less ambiguous than the previous ones because the Java type system has to be described exhaustively in the documentations so compilers can be written. The prompt did not specify whether the primitive types should be represented in the model; this model only depicts the system used for first-class objects.

Complexities Ignored:
The existence of primitive types were ignored; they are primarily an implementation detail used for optimization. Java provides object wrappers for them that do fit in the model, and allows them to be used almost interchangably with the primitives through autoboxing.
Generic types were also not discussed in the model, though for the most part they can be described similarly to other types in the type system. During runtime, they are treated exactly the same as non-generic types because of type erasure.
The various implementation details of the type system were also ignored; this model focuses on the interface and the aspects of the system that a Java programmer needs to be aware of.

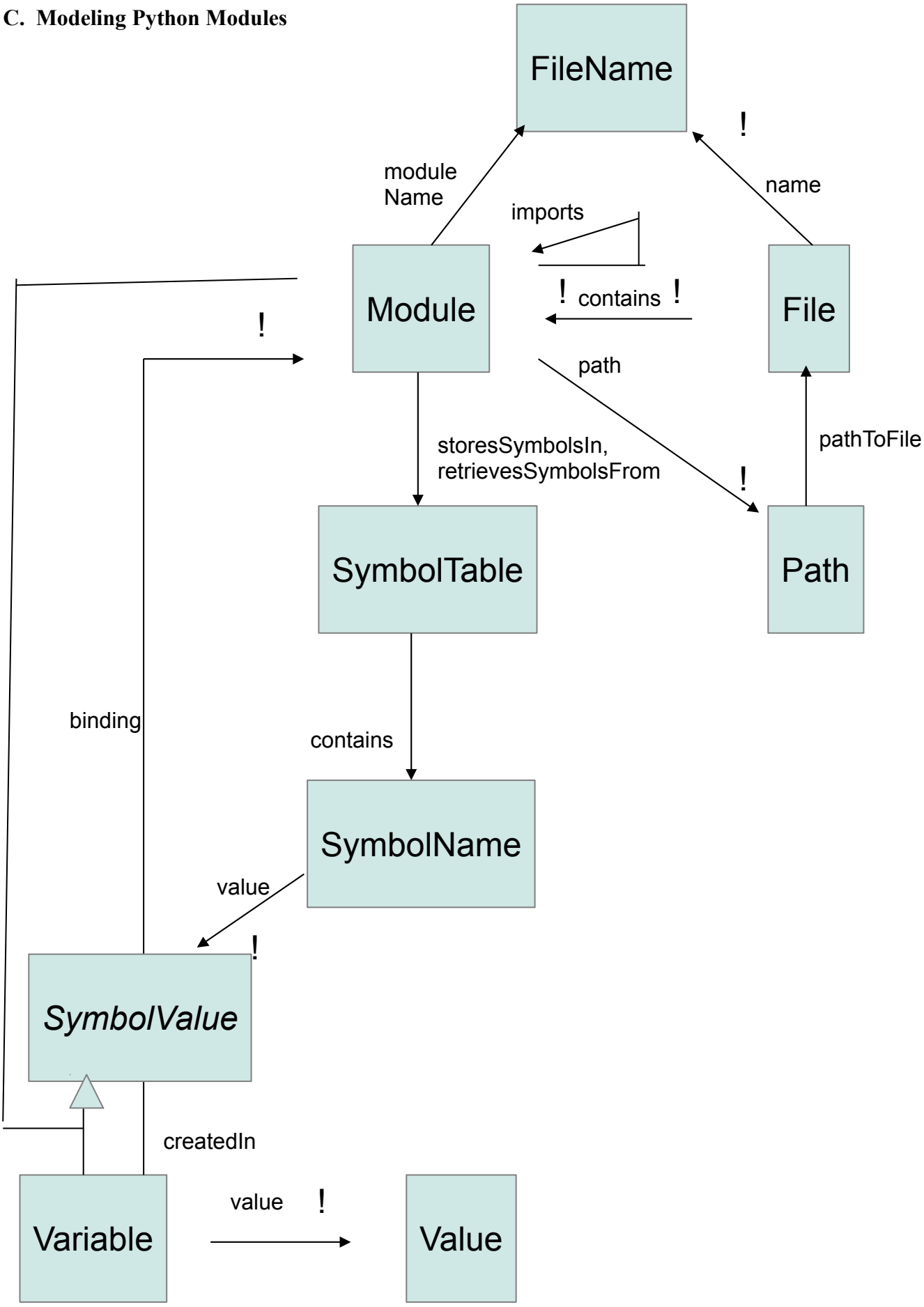
Designations:
"TypeDefinition" refers to the information a programmer provides when defining a new type and providing its available operations. A programmer can create a type using an interface or a class.
"TypeInformation" is a tuple combining "CompileTimeType" and "RuntimeType". In Java, variables often have different static and dynamic types, allowing programmers to use polymorphism. For example, in the following line of Java code:

```
final Foo foo = new FooImpl();
```

The variable foo has a CompileTimeType of Foo and a RuntimeType of FooImpl. Without typecasting, foo will respond only to methods defined Foo's interface, but will run the code from the versions of the methods defined in FooImpl.


TypeDefinition
!
Interface
implements
Class
? extends
AbstractClass
ConcreteClass
defined
By
!
compileType
CompileTimeType
Variable
type
!
TypeInformation
runtime
Type
!
RuntimeType

C. Modeling Python Modules



1. Bindings are added to a module's symbol table whenever a new variable is defined inside that module. The table maps the variable's name to its value. If a programmer tries to access a variable that is not in the table, Python will raise a NameError.
2. The name of the module will be bound to the variable __name__. The module's name is determined by taking the file's name and removing the .py extension. That is, the code in foo.py will be in the module named foo, and accessed from outside by executing “import foo”.
3. Import statements are used to access code from a different module. The statement “import m” will add only the symbol “m” to the current module's symbol table; m's value will be a reference to the module and the variables defined inside m can be accessed through m.variable_name. “from m import d” in contrast does not add “m” to the symbol table. It only adds “d”, and its value can be accessed without going through m. If an imported name is already bound, the new value from the imported module will overwrite the old one, which is why the “import m” syntax is usually to be preferred.

Ambiguities Resolved:

The system being modeled was relatively unambiguous because the specification has to clarify details so Python interpreters can be written.

Complexities Ignored:

Full details about bindings, the __name__ variable, and the different types of imports were not displayed in the diagram; they are explained in text instead.

The possible existence of multiple virtual Python paths with only one being active at a time (through the use of the virtualenv development tool) was ignored; this model assumes the case in which only the real environment is being used.

The implementation details of modules and imports were also ignored; the scope of this model is to focus on the interface that a Python programmer should know.

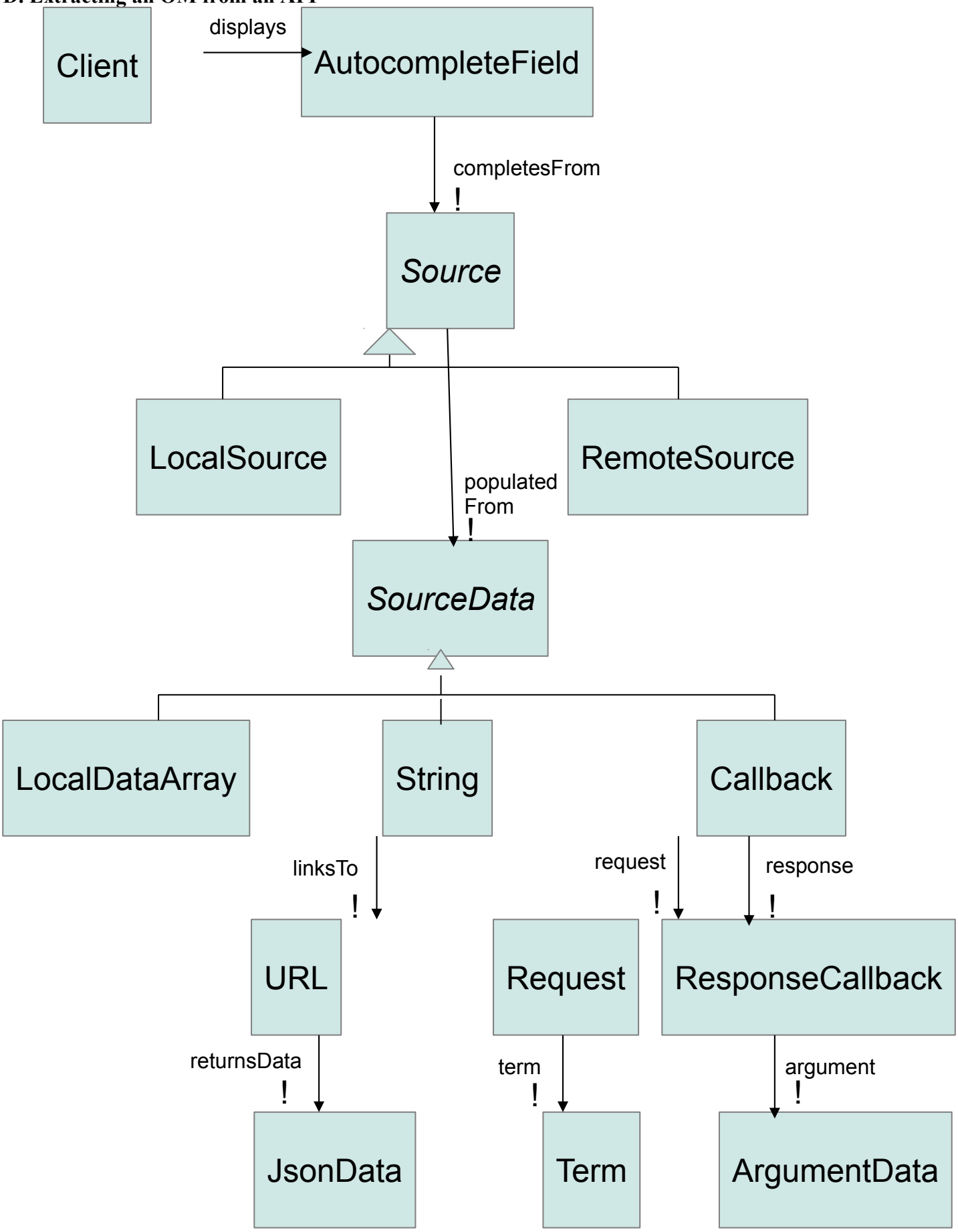
Designations:

“SymbolTable” refers to the dictionary that Python uses to bind variable names (“SymbolName”) to values. Each module has its own namespace, with import being the method of merging in symbols from other tables.

“Path” refers to the Python path, which is searched for modules whenever an import statement is run by the interpreter.

“SymbolValue” refers to the value associated with a symbol, usually a variable. Python treats functions the same way as scalar variables in this table; there will be a collision if both a scalar and a function share the same name. Note that modules can also be treated as SymbolValues. When the code “import foo” is executed, the name “foo” is bound to a reference to the foo module, and can be passed around like any other variable.

D. Extracting an OM from an API



Ambiguities Resolved:
The prompt did not specify how many levels of abstraction the model should cover. This model depicts the data that a user of the JQuery autocomplete API would need to know to write code that uses it.

Complexities Ignored:
This model ignores the implementation details of the autocomplete API. Also, it does not display the details about how the execution of the program proceeds given that callbacks are being used.

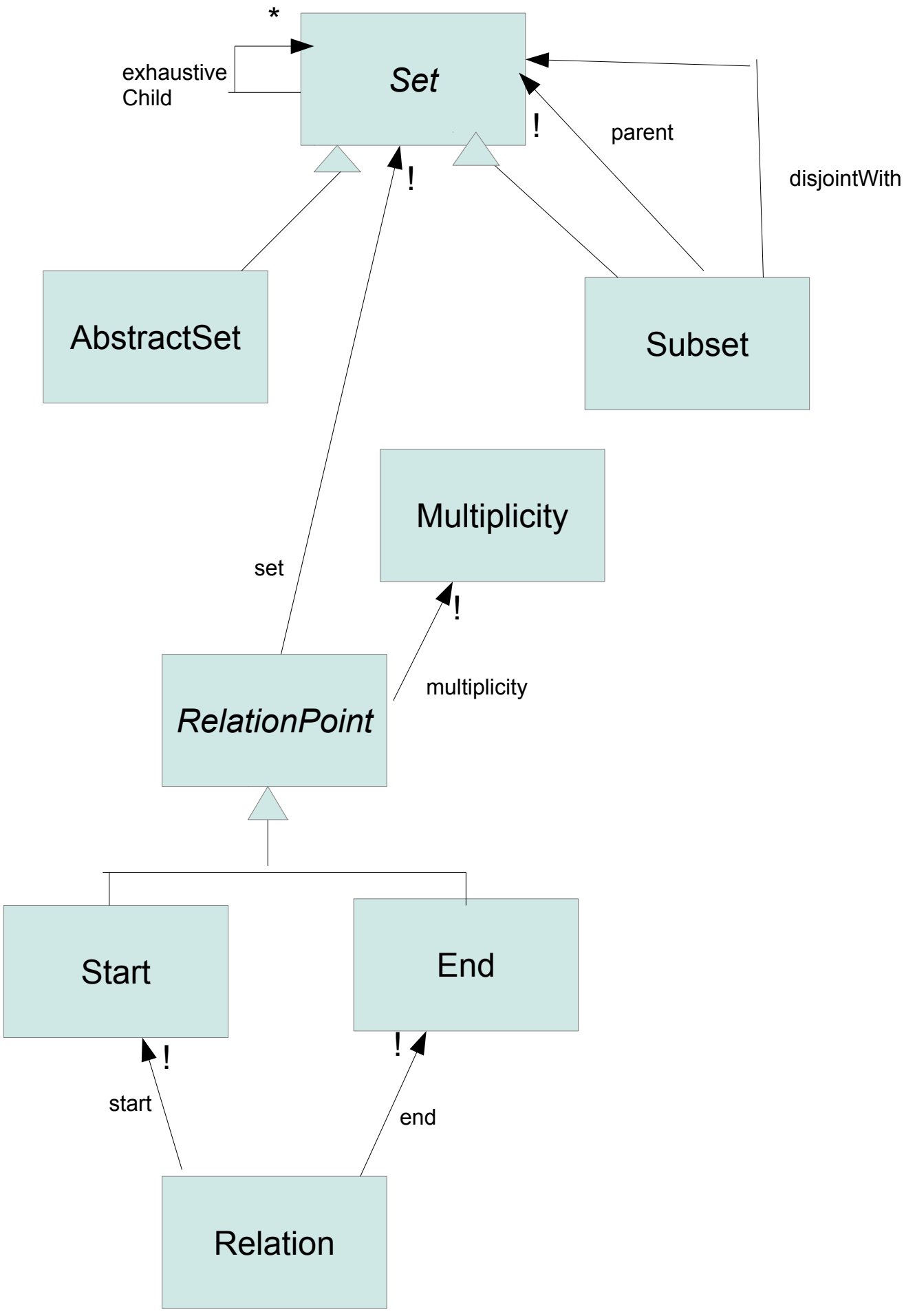
Designations:
The "Client" is the programmer using the API; he includes the JQuery code and calls the API's functions. The end-user whose text is autocompleted is not in the model because it is from the perspective of client code.
"Source" refers to the source of the information that can be used to autocomplete the text field; it can be either local or remote.
"SourceData" refers to how the data can be provided for autocompleting. It has been extracted from "Source" and connected by a relation (instead of directly making "LocalDataArray", "String", and "Callback" direct subsets of "Source") to avoid the ambiguity that results when multiple subsets in different disjoint trees exist and the superset is labeled as exhausted. Both SourceData and Source are exhausted by their subsets.
"String" is related to a URL because of the way the API works – data is retrieved from the URL specified by the String as JSON.
"Callback" is the callback option for getting data remotely. It takes as arguments a Request object with a field called Term (the term to be completed) and a Response callback whose argument contains the data.

E. Metamodeling
1. (Diagram on next page due to spacing constraints)

Ambiguities Resolved:
It was not given whether the model should contain a notion of the deficiency in the language's expressiveness that occurs when multiple disjoint subset trees exist for a set and we want to indicate that the set is exhausted by one or more of the subsets. This model simply describes the relation between Set and Subset as the prompt said to display the features of the notation, not features that aren't available.

Complexities Ignored:
The different types of multiplicites are not included in the diagram, because they are just syntax and the primary idea to convey is that it is possible for relation points to have multiplicity. This could easily be added to the model by making *, +, ?, and ! sets and making them exhaustive subsets of Multiplicity.

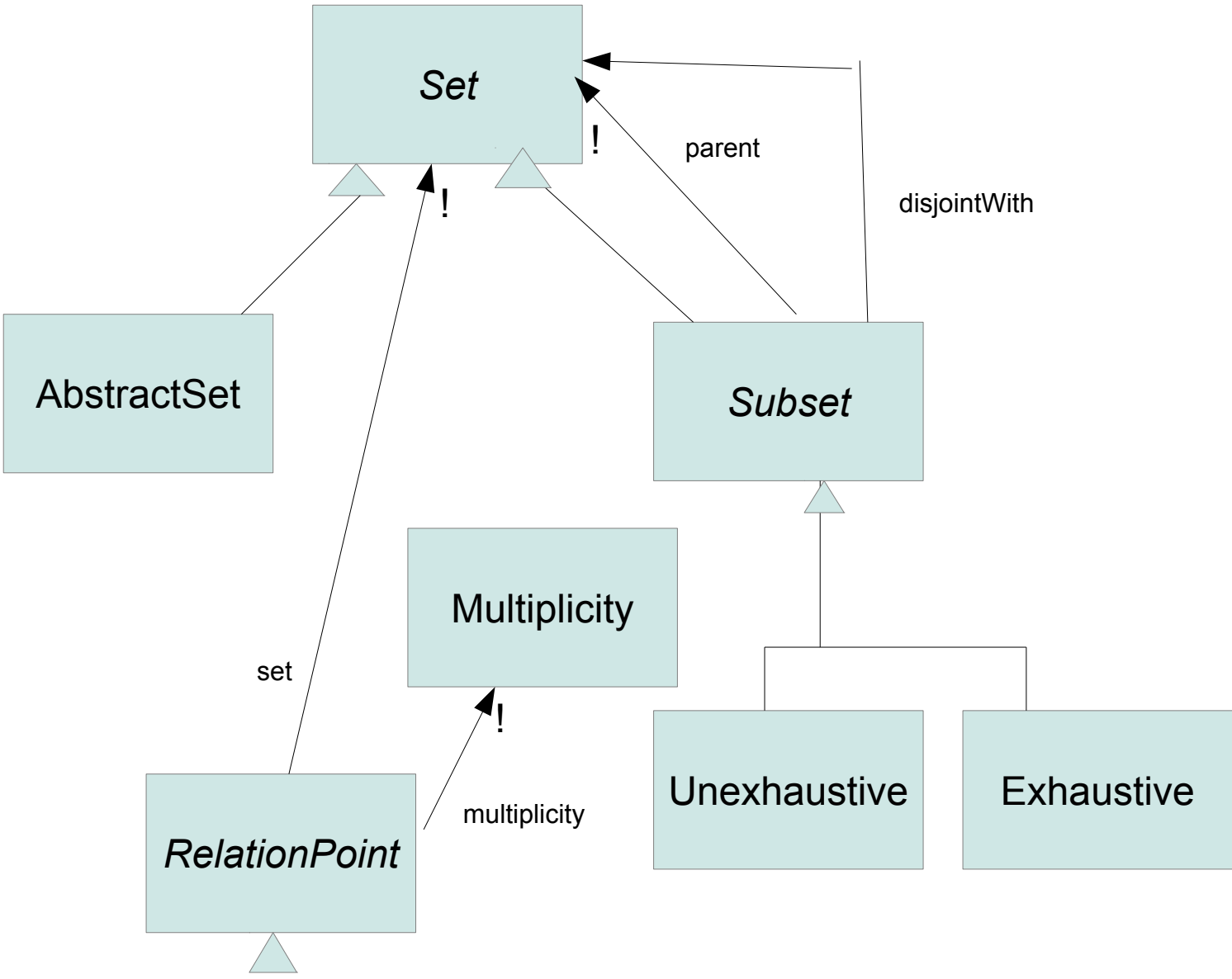
Designations:
If "Set's" exhaustiveChild relation exists for one child, then it must exist for all disjoint subsets of the original set. The notation as given can not express a more fine-grained version of this relation.
"AbstractSet" refers to any set which is marked in italics indicating that all options are exhausted by its subsets. "Subset" refers to any other type of set that has a Set as its parent. Note that it can be disjoint with other Sets (this includes AbstractSets and other Subsets).
"RelationPoint" is an abstract set refering to the two endpoints of a relation arrow, which can be labeled with multiplicity synbols. The set is exhausted by its subsets, "Start" and "End" which are the starting point and ending point of a relation arrow.
\


*
exhaustive
Child
Set
!
parent
disjointWith
!
AbstractSet
Subset
Multiplicity
set
!
RelationPoint
multiplicity
Start
End
!
!
start
end
Relation

2. Extension to the object model to overcome the stated deficiency

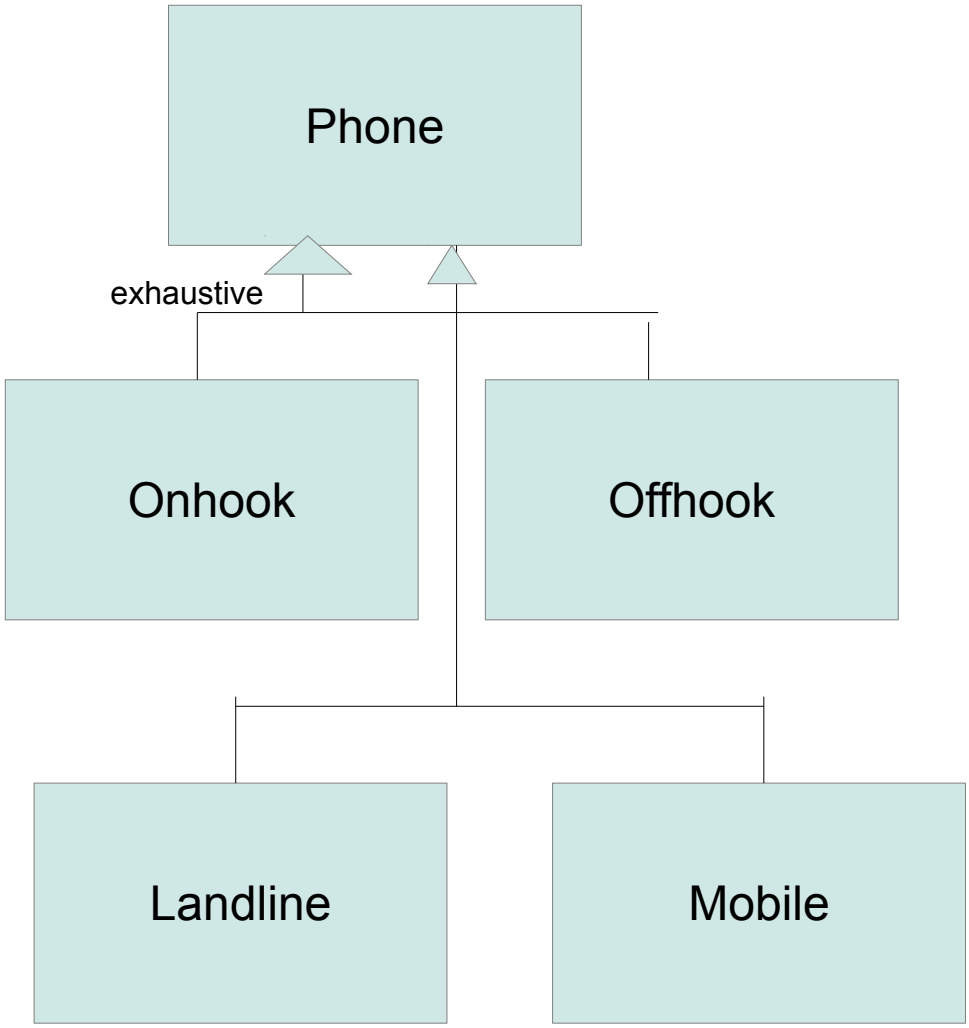
a) If a set is exhausted by some disjoint subsets, but not all, the arrow pointin to the superset from the exhaustive disjoint subsets are labeled as “exhaustive” (in the location where multiplicity would be labeled on the endpoint of a relation arrow). Nonexhaustive subset arrows are not labeled (implying “nonexhaustive”). If the set is exhausted by all its disjoint subsets, its name should be italicized (and “exhaustive” can be omitted in this case). However, if some subsets are not exhaustive, the name should remain unitalicized (and the “exhaustive” label is required to distinguish between subsets). (See c) for a graphical example)

b)
All subsets can be classified as either exhaustive or unexhaustive. The “exhaustive” label from a) puts all subsets associated with the labeled arrow into the Exhaustive category. If the superset is italicized, then all subsets are in the Exhaustive category. AbstractSets do not have an “exhaustive” or “unexhaustive” label, but their concrete children must have them so all leaves of a subset tree can be labeled in the end.



(The lower half of the diagram is the same as before)

c) Example



Onhook and Offhook are exhaustive – a phone must be engaged in a call or not engaged in a call at all times. However, Landline and Mobile are not exhaustive. The arrow label differentiates between them.

3. It is true that it is possible in many cases to work around the deficiency in the modeling notation. One can either extract additional subsets, rethink the sets in some other way, or if all else fails, explain which subsets are exhaustive in a footnote. Despite this, I think that this extension to the modeling notation is a good idea, because it is possible for the workarounds to become clumsy and make the model harder to understand. Building this feature into the model notation can resolve ambiguities in a more concise way, and those opposed to the feature can always fall back to the original workarounds.