2

Tree Template

The tree is a term from graph theory. A ***tree*** is a set of nodes that connect from child to parent. A node can have many child nodes; each node in a tree has one parent node except for the node at the tree's top. There are no cycles — that means at most one path connects any two nodes.

Figure 2.1 shows an example of a tree. *A* is at the top of the tree and has no parent node. *A* is the parent for *B*, *C*, and *D*. *B* is the parent for *E* and *C* is the parent for *F*.

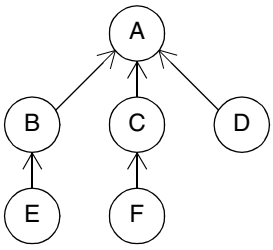


Figure 2.1 Sample tree. A tree organizes data into a hierarchy.

There are six templates for trees. The first hardcodes an entity type for each level of the tree. The others are more abstract and use the same entity types for all levels.

- **Hardcoded tree**. Hardcodes entity types, one for each level of the tree. Use when the structure of a tree is well known and it is important to enforce the sequence of types in the levels of the hierarchy.
- **Simple tree**. Restricts nodes to a single tree. Treats nodes the same. Use when tree decomposition is merely a matter of data structure.
- **Structured tree**. Restricts nodes to a single tree. Differentiates leaf nodes from branch nodes. Use when branch nodes and leaf nodes have different attributes, relationships, and/or semantics.

- **Overlapping trees**. Permits a node to belong to multiple trees. Treats nodes the same. Use when a node can belong to multiple trees.
- **Tree changing over time**. Stores multiple variants of a tree. A particular tree can be extracted by specifying a time. Restricts nodes to a single tree. Treats nodes the same. Use when the history of a tree must be recorded.
- **Degenerate node and edge**. Groups a parent with its children. The grouping itself can be described with attributes and relationships. Restricts nodes to a single tree. Treats nodes the same. Use when the grouping of a parent and its children must be described.

2.1 Hardcoded Tree Template

2.1.1 UML Template

Figure 2.2 shows the UML template for hardcoded trees. (The Appendix explains UML notation.) The diagram has three levels, but in practice there can be any number of levels. The angle brackets denote parameters that require substitution. A ***Tree*** is a hierarchy of entities with the entities of each level having the same entity type. You need not show *Tree* in a use of the template.

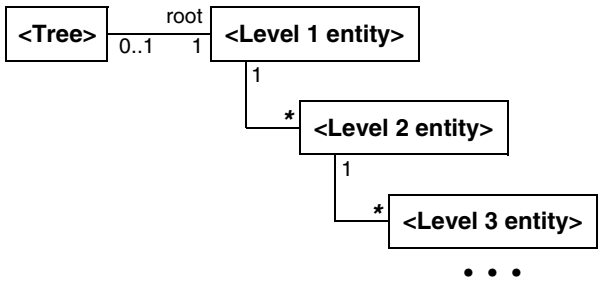


Figure 2.2 Hardcoded tree: UML template. Use when the structure is well known and it is important to enforce the sequence of types in the levels of the hierarchy.

The hardcoded representation is easy to understand, but it is fragile. The hardcoded template is only appropriate when the data structure is well known and unlikely to change. Otherwise a change to the hierarchy of types breaks the model and database structure necessitating rework of application code.

In practice I seldom build applications with the hardcoded tree template. Nevertheless, this is a good template to keep in mind. It is often helpful to model data using a hardcoded template — so that business persons and other stakeholders can understand the model. Then you can build the application with an abstract representation (Section 2.2 – Section 2.6) and populate it with the content of the hardcoded model.

2.1.2 IDEF1X Template

Figure 2.3 restates Figure 2.2 with the IDEF1X notation. (The Appendix explains IDEF1X notation.) The following are foreign keys: *rootID* references *Level1entity*, *level1ID* references *Level1entity*, and *level2ID* references *Level2entity*. Although it is not shown, the levels would have fields for application data.

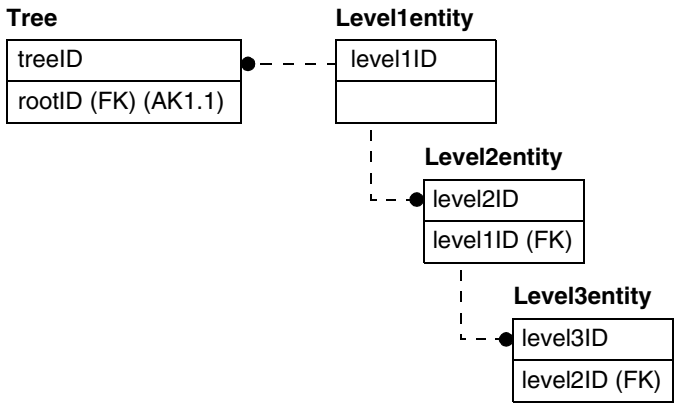


Figure 2.3 Hardcoded tree: IDEF1X template.

2.1.3 SQL Queries

Figure 2.4 and Figure 2.5 show representative SQL queries for the template. These queries are very simple which is an attraction of the hardcoded approach. The colon prefix denotes variable values that must be provided. The subsequent levels of the hierarchy (level 3, level 4, and so forth) have similar queries.

```
SELECT level1ID
FROM Level2entity AS L2
WHERE level2ID = :aLevel2ID;
```

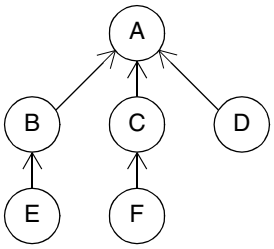
Figure 2.4 Hardcoded tree: SQL query. Find the parent for a child.

```
SELECT level2ID
FROM Level2entity
WHERE level1ID = :aLevel1ID
ORDER BY level2ID;
```

Figure 2.5 Hardcoded tree: SQL query. Find the children for a parent.

2.1.4 Sample Populated Tables

Figure 2.6 shows hardcoded tables populated with data. A is at level 1; B, C, and D are at level 2; E and F are at level 3. The ID values are arbitrary but internally consistent.



Level_1 table

level1ID	name	treeID
1	A	1

Level_2 table

level2ID	name	level1ID
1	B	1
2	C	1
3	D	1

Level_3 table

level3ID	name	level2ID
1	E	1
2	F	2

Figure 2.6 Hardcoded tree: Populated tables.

2.1.5 Examples

Figure 2.7 shows a fixed organization chart. Of course, more levels could be added. This simple model could be appropriate for a small company but is risky for a large company as it is rigid and does not accommodate change.

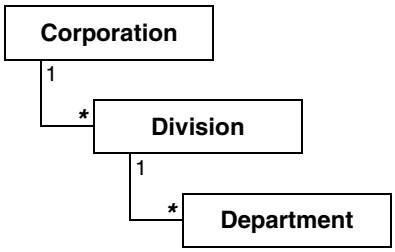


Figure 2.7 Hardcoded tree: Organizational chart model.

Figure 2.8 has another example, the division of a book into chapters, sections, and subsections.

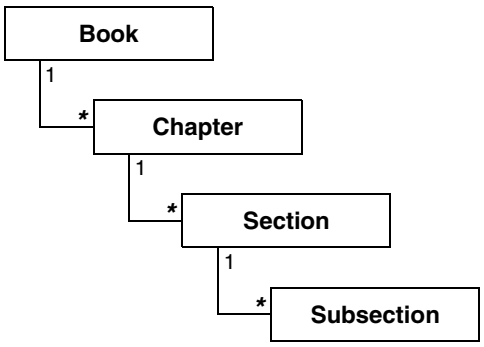


Figure 2.8 Hardcoded tree: Book structure model.

2.2 Simple Tree Template

2.2.1 UML Template

In the simple tree template (Figure 2.9) all nodes are the same and decomposition is merely a matter of data structure. A ***Tree*** is a hierarchy of nodes and has one node as the root. A ***Node*** is an entity type whose records are organized as a *Tree*. An individual node may, or may not, be the root. You need not show *Tree* in a use of the template.

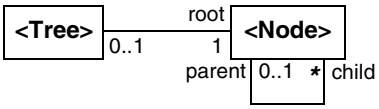
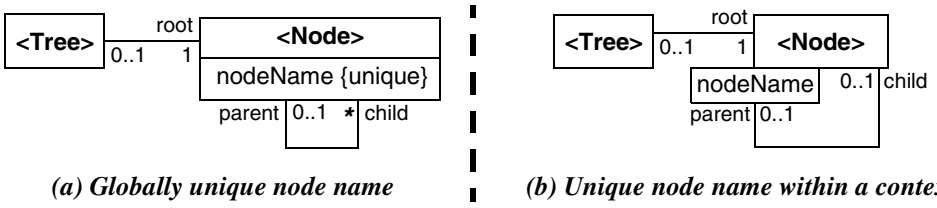


Figure 2.9 Simple tree: UML template. Use when tree decomposition is merely a matter of data structure.

Figure 2.9 adds the constraint that the tree cannot have any cycles. (Consider Figure 2.1. *A* cannot have *F* as a parent even though the template in Figure 2.9 does not prevent it. Traversing parent relationships, this would cause a cycle from *F* to *C* to *A* back to *F*.) Similarly each node must have a parent, except for the root node — the template alone is more lax.

Each node may have a name. As Figure 2.10 shows, node names can be globally unique (left template) or unique within the context of a parent node (right template). The box under *Node* in Figure 2.10b is a UML qualifier (see the Appendix for an explanation).

In Figure 2.1 node names are globally unique. In contrast, a node name can be unique within a context. Figure 2.11 shows an excerpt of this book's file structure where I have kept old copies of files from reviews. File names are unique within the context of their directory.



(a) Globally unique node name *(b) Unique node name within a context*

Figure 2.10 Simple tree: UML template, with node names. There are two variations of the template—globally unique names and names within a context.

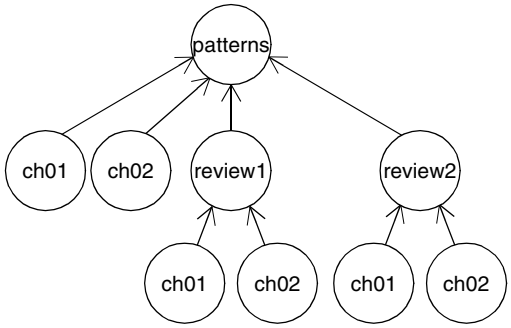
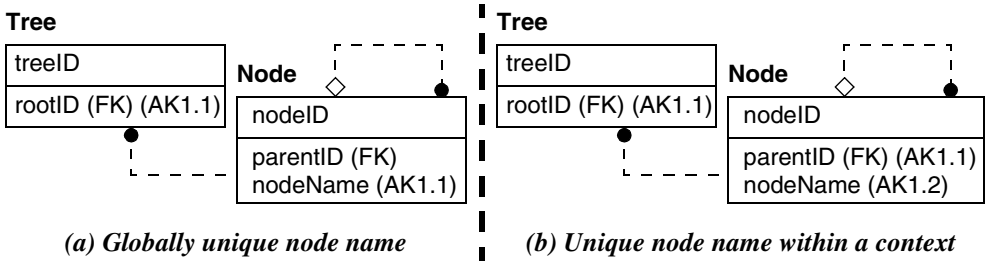


Figure 2.11 Sample tree with node names that are unique within a context.

2.2.2 IDEF1X Template

Figure 2.12 restates Figure 2.10 with the IDEF1X notation. The following are foreign keys: *rootID* references *Node* and *parentID* references *Node*.



(a) Globally unique node name *(b) Unique node name within a context*

Figure 2.12 Simple tree: IDEF1X template.

Figure 2.12 uses existence-based identity which is my preferred approach for database design. (See Chapter 16.) ***Existence-based identity*** means that each entity has its own artificial identifier (such as an Oracle sequence or a SQL Server identity field) as a primary key. Thus the primary key of *Node* is *nodeID* and not, for example, *nodeName* (Figure 2.12a) or

parentID + *nodeName* (Figure 2.12b). Although I favor existence-based identity, the templates in this book do not require it.

In Figure 2.12 the *AK* notation denotes a candidate key, that is a combination of attributes that is unique for each record in a table. No attribute in a candidate key can be null. All candidate key attributes are required for uniqueness.

Figure 2.12b defines *parentID* + *nodeName* as a candidate key, but one record is an exception. The root node for a tree has a null *parentID* and a candidate key cannot involve a null. Most relational DBMSs treat NULL as just another value and do not strictly enforce candidate keys, so the definition of a unique key for *parentID* + *nodeName* does work. (I verified this for SQL Server and expect that it would work for most relational DBMSs.) Alternatively, you can forego the unique key and check the constraint with application code.

2.2.3 SQL Queries

Figure 2.13 and Figure 2.14 show SQL queries for common traversals of the template. The colon prefix denotes variable values that must be provided.

```
SELECT Parent.nodeID AS parentNodeID,
   Parent.nodeName AS parentNodeName
FROM Node AS Child
   INNER JOIN Node AS Parent ON Child.parentID = Parent.nodeID
WHERE Child.nodeID = :aChildNodeID;
```

Figure 2.13 Simple tree: SQL query. Find the parent for a child node.

```
SELECT Child.nodeID AS childNodeID,
   Child.nodeName AS childNodeName
FROM Node AS Child
WHERE Child.parentID = :aParentNodeID
ORDER BY Child.nodeName;
```

Figure 2.14 Simple tree: SQL query. Find the children for a parent node.

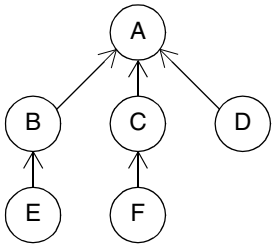
2.2.4 Sample Populated Tables

Figure 2.15 shows sample simple tree tables populated with data. The ID values are arbitrary, but internally consistent.

2.2.5 Examples

Simple trees also arise in many applications.

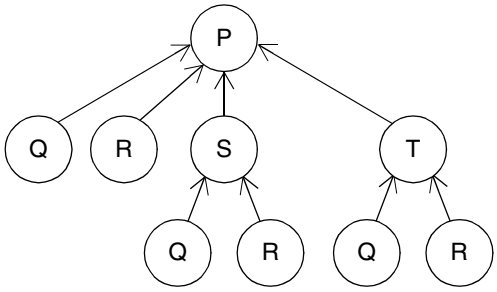
In Figure 2.16 a *manager* has many *subordinates*. Each *subordinate* reports to at most one *manager*: The CEO reports to no *manager*, and all others report to one *manager*. The management hierarchy can be arbitrarily deep.



Node table

nodeID	parentID	nodeName
1		A
2	1	B
3	1	C
4	1	D
5	2	E
6	3	F

(a) Globally unique node name



Node table

nodeID	parentID	nodeName
1		P
2	1	Q
3	1	R
4	1	S
5	1	T
6	4	Q
7	4	R
8	5	Q
9	5	R

(b) Unique node name within a context

Figure 2.15 Simple tree: Populated tables.

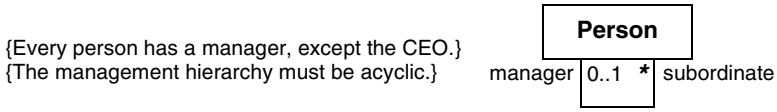


Figure 2.16 Simple tree: Management hierarchy model.

In Figure 2.17 formal requests for proposals (RFPs), such as government projects, often involve extensive requirements that are captured with an indented list. Level 0 is the requirement for the RFP as a whole that elaborates into levels 1, 2, 3, and so forth. A level 1 requirement can have sub requirements such as 1.1 and 1.2. These sub requirements can have further detail such as 1.1.1, 1.1.2, and 1.2.1. The nesting can be arbitrarily deep depending on the desired level of detail. The RFP yields a tree of requirements.

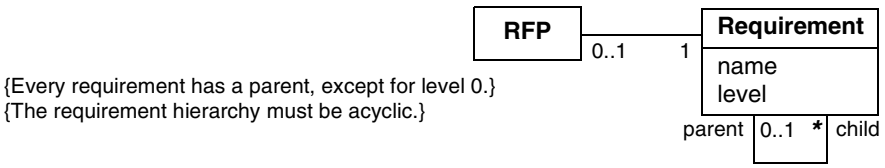


Figure 2.17 Simple tree: Nested requirements for RFPs model.

2.3 Structured Tree Template

2.3.1 UML Template

Figure 2.18 shows the UML template for structured trees when there is a need to differentiate leaf nodes from branch nodes. A ***Tree*** is a hierarchy of nodes and has one node as the root. A particular node may, or may not, be the root. You need not show *Tree* in a use of the template.

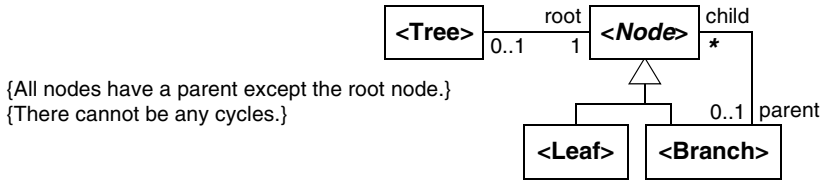


Figure 2.18 Structured tree: UML template. Use when branch nodes and leaf nodes have different attributes, relationships, and/or semantics.

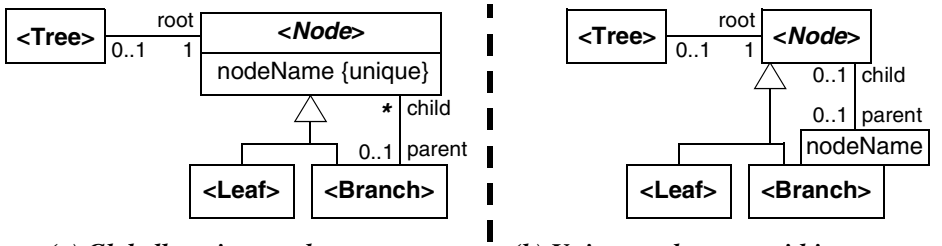
A ***Node*** is either a leaf node or a branch node. A ***Leaf*** node (such as *D*, *E*, and *F* in Figure 2.1) terminates the tree recursion. A ***Branch*** node (such as *A*, *B*, and *C* in Figure 2.1) can have child nodes each of which, in turn, can be a leaf node or a further branch node.

Figure 2.18 adds the constraint that a tree cannot have any cycles. (See Section 2.2.1 for an explanation of cycles.) Similarly each node must have a parent, except for the root node — the template alone is more lax.

As with simple trees, the node names in Figure 2.19 can be globally unique (left template) or unique within a context (right template).

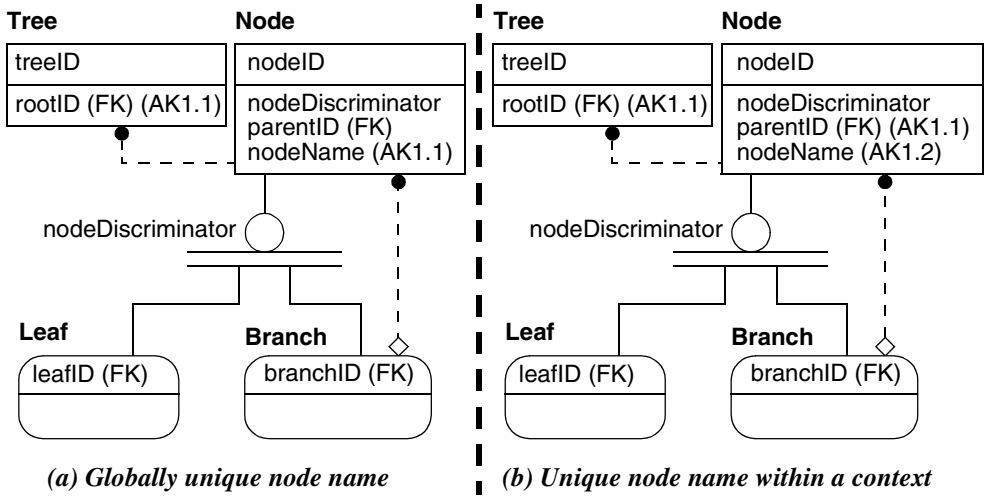
2.3.2 IDEF1X Template

Figure 2.20 restates Figure 2.19 with the IDEF1X notation. The following are foreign keys: *rootID* references *Node*, *parentID* references *Branch*, *leafID* references *Node*, and *branchID* references *Node*. The generalization is exhaustive—every *Node* record must have a corresponding *Leaf* record or *Branch* record. The *nodeDiscriminator* field is an enumeration with values “Leaf” and “Branch” indicating the appropriate subtype record.



(a) Globally unique node name *(b) Unique node name within a context*

Figure 2.19 Structured tree: UML template, with node names. There are two variations of the template—globally unique names and names within a context.



(a) Globally unique node name *(b) Unique node name within a context*

Figure 2.20 Structured tree: IDEF1X template.

As with Figure 2.12b, Figure 2.20b defines *parentID* + *nodeName* as a candidate key, but one record is an exception. The root node for a tree has a null *parentID* and a candidate key cannot involve a null. Since most relational DBMSs are lax and treat NULL as just another value, the definition of a unique key for *parentID* + *nodeName* does work. Although it is not shown, the *Leaf* and *Branch* tables would have additional fields for application data.

2.3.3 SQL Queries

Figure 2.21 and Figure 2.22 show SQL queries for common traversals of the template. The queries could omit the *Branch* table, but I wrote them according to template traversal—then it is easy to retrieve any data that is added to the template. The colon prefix denotes variable values that must be provided.

```
SELECT Parent.nodeID AS parentNodeID,
   Parent.nodeName AS parentNodeName
FROM Node AS Child
   INNER JOIN Branch AS B ON Child.parentID = B.branchID
   INNER JOIN Node AS Parent ON B.branchID = Parent.nodeID
WHERE Child.nodeID = :aChildNodeID;
```

Figure 2.21 Structured tree: SQL query. Find the parent for a child node.

```
SELECT Child.nodeID AS childNodeID,
   Child.nodeName AS childNodeName
FROM Node AS Child
   INNER JOIN Branch AS B ON Child.parentID = B.branchID
   INNER JOIN Node AS Parent ON B.branchID = Parent.nodeID
WHERE Parent.nodeID = :aParentNodeID
ORDER BY Child.nodeName;
```

Figure 2.22 Structured tree: SQL query. Find the children for a parent node.

2.3.4 Sample Populated Tables

Figure 2.23 shows sample structured tree tables populated with data. The ID values are arbitrary, but internally consistent.

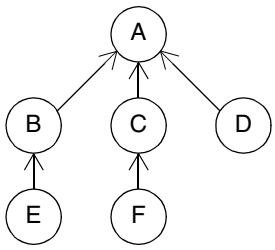
2.3.5 Examples

Trees often arise in applications and sometimes the structured tree template is the best choice.

Many drawing applications and user interfaces have the notion of a group. In Figure 2.24 a *DrawingObject* is *Text*, a *GeometricObject*, or a *Group*. A *Group* has two or more lesser *DrawingObjects*; the resulting recursion yields trees of *DrawingObjects*. Note the further refinement from Figure 2.18—a group must include at least two *DrawingObjects*.

Figure 2.25 is the analog to Figure 2.16. Figure 2.16 suffices if you merely need to record the reporting structure. In Figure 2.25 a *Person* can be a *Manager* or an *IndividualContributor*. Except for the CEO, each *Person* reports to a *Manager*. The management hierarchy can be arbitrarily deep. There are material differences between *Managers* and *IndividualContributors* necessitating the use of subtypes. For example, only *Managers* can be in charge of *Departments*.

Many years ago, the original Microsoft PC-DOS file structure was a hierarchy. Each file belonged to at most one directory. (Modern operating systems permit files to belong to multiple directories as the next chapter explains.) In Figure 2.26 a *File* may be a *DataFile* or a *DirectoryFile*. Directories contain multiple files, some or all of which may be subdirectories. The combination of a *DirectoryFile* and a *fileName* yields a specific *File*—file names are unique within the context of their directory. All *Files* belong to a single directory except for



Node table

node ID	node Discrim	parent ID	node Name
1	branch		A
2	branch	1	B
3	branch	1	C
4	leaf	1	D
5	leaf	2	E
6	leaf	3	F

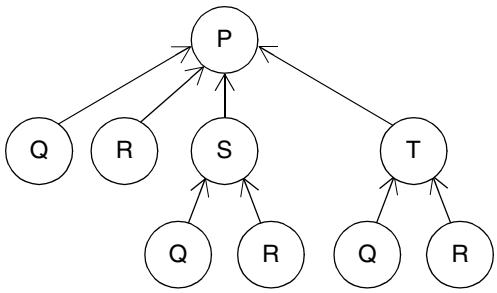
Branch table

branchID	. . .
1	
2	
3	

Leaf table

leafID	. . .
4	
5	
6	

(a) Globally unique node name



Node table

node ID	node Discrim	parent ID	node Name
1	branch		P
2	leaf	1	Q
3	leaf	1	R
4	branch	1	S
5	branch	1	T
6	leaf	4	Q
7	leaf	4	R
8	leaf	5	Q
9	leaf	5	R

Branch table

branchID	. . .
1	
4	
5	

Leaf table

leafID	. . .
2	
3	
6	

Leaf table (cont.)

leafID	. . .
7	
8	
9	

(b) Unique node name within a context

Figure 2.23 Structured tree: Populated tables.

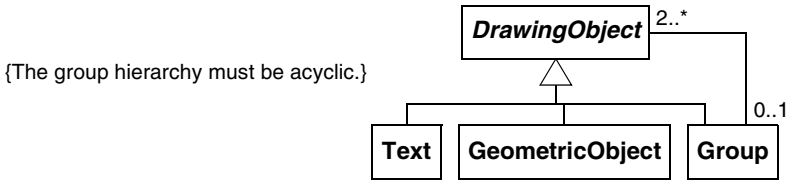


Figure 2.24 Structured tree: Graphical editor model.

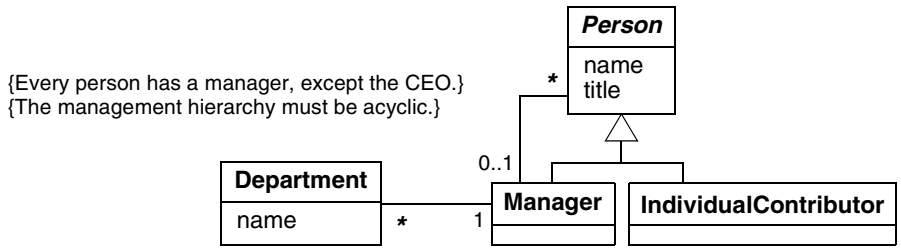


Figure 2.25 Structured tree: Management hierarchy model.

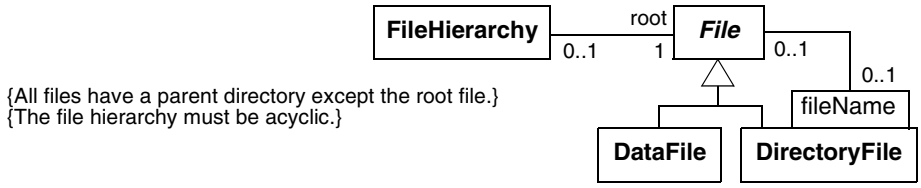


Figure 2.26 Structured tree: File hierarchy directory model.

the *root File*, which belongs to none. Directories can be nested to an arbitrary depth, with *DataFiles* and empty *DirectoryFiles* terminating the recursion.

2.4 Overlapping Trees Template

2.4.1 UML Template

Figure 2.27 permits a node to belong to multiple trees. A ***Tree*** is a hierarchy of nodes and has one node as the root. A ***Node*** is an entity type whose records are organized as a *Tree*. A node may be the root of multiple trees. You should include *Tree* when using this template, so that you can distinguish the multiple trees. The dotted line and attached box is UML notation for an entity type that is also a relationship (see the Appendix for an explanation.)

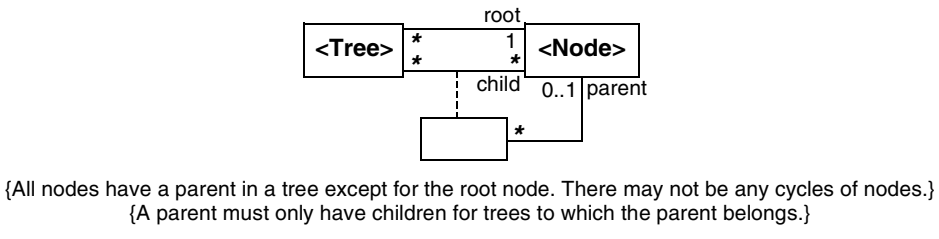


Figure 2.27 Overlapping tree: UML template. Use when a node can belong to more than one tree.

You can retrieve a tree by starting with a tree record and retrieving the node that is the root of the tree. Traverse the parent relationship to retrieve the collection of children for the root node. You can recursively expand the tree, level by level, traversing parent relationships to get the next lower level of children. As you traverse nodes, filter records and only consider children of the tree under consideration.

Figure 2.27 treats nodes uniformly like the simple tree template. It would be confusing to distinguish between branches and leaves as with the structured tree template because the distinction could vary across the different trees for a node. All the overlapping–tree examples I have seen to date treat nodes uniformly.

As with the other tree templates, Figure 2.27 adds a constraint that forbids cycles, as the template alone cannot prevent them. Each node in a tree must have a parent except for the root node. Another constraint is that a parent must only have children for trees to which the parent belongs.

This template is already complex, so it is best to handle node names in a simple manner. Each node has a globally unique name and there is no provision to vary node name by context.

2.4.2 IDEF1X Template

Figure 2.28 restates Figure 2.27 with the IDEF1X notation. The following are foreign keys: *rootID*, *treeID*, *childID,* and *parentID*.

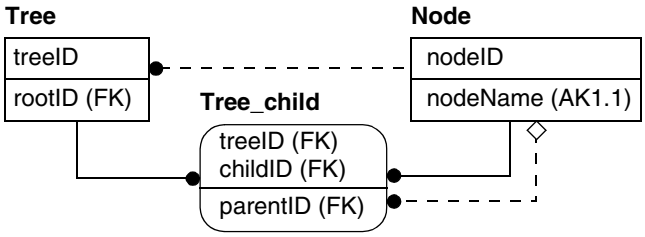


Figure 2.28 Overlapping tree: IDEF1X template.

2.4.3 SQL Queries

Figure 2.29 and Figure 2.30 show SQL queries for common traversals of the template. The colon prefix denotes variable values that must be provided for each query.

```
SELECT N.nodeID AS parentNodeID, N.nodeName AS parentNodeName
FROM Tree_child Tc
   INNER JOIN Node AS N ON Tc.parentID = N.nodeID
WHERE Tc.treeID = :aTreeID AND Tc.childID = :aChildNodeID;
```

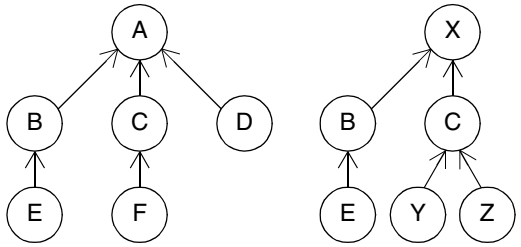
Figure 2.29 Overlapping tree: SQL query. Find the parent for a child node.

```
SELECT N.nodeID AS childNodeID, N.nodeName AS childNodeName
FROM Tree_child Tc
   INNER JOIN Node AS N ON Tc.childID = N.nodeID
WHERE Tc.treeID = :aTreeID AND Tc.parentID = :aParentNodeID
ORDER BY N.nodeName;
```

Figure 2.30 Overlapping tree: SQL query. Find the children for a parent node.

2.4.4 Sample Populated Tables

Figure 2.31 shows sample overlapping tree tables populated with data using globally unique node names. The ID values are arbitrary, but internally consistent.



Node table

node ID	node Name
1	A
2	B
3	C
4	D
5	E
6	F
7	X
8	Y
9	Z

Tree_child table

tree ID	**child ID**	parent ID
1	**1**	
1	**2**	1
1	**3**	1
1	**4**	1
1	**5**	2
1	**6**	3

Tree_child table (cont)

tree ID	**child ID**	parent ID
2	**7**	
2	**2**	7
2	**3**	7
2	**5**	2
2	**8**	3
2	**9**	3

Figure 2.31 Overlapping tree: Populated tables.

2.4.5 Example

Overlapping trees occur less often than structured and simple trees.

Mechanical parts provide a compelling example. In Figure 2.32 a *PartRole* can be the *root* of a *BOM* (bill-of-material) and have multiple children, successively forming a tree. A

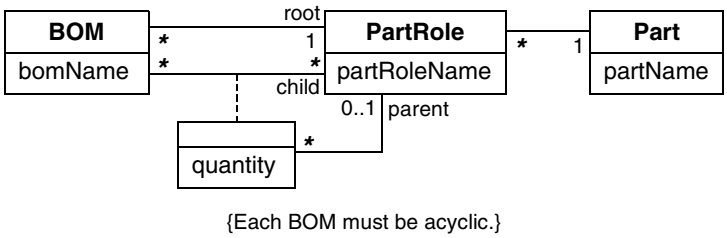


Figure 2.32 Overlapping tree: Mechanical parts model.

PartRole is the usage of a *Part*. The same part may have different usages within the same *BOM* and across *BOMs*. There are overlapping trees because a *PartRole* can belong to multiple *BOMs*, such as one for a product's design (engineering BOM), another for how it is built (manufacturing BOM), and another for how it is maintained (service BOM).

Figure 2.33 shows engineering and manufacturing BOMs for a lamp.

Engineering BOM for a Lamp

Level	PartName	PartRoleName	Qty
01	L101	Lamp	1
02	SA1	Socket asm	1
03	WA1	Wiring assembly	1
04	Plg2	Plug	1
04	C2	Cord	1
04	PS1	Power switch	1
04	Scr1	Switch screw	2
03	Skt1	Socket	3
02	SS1	Shade support	1
02	Shd5	Shade	1
02	BA1	Base assembly	1
03	M2	Mat	1
03	B1	Base	1
03	Sft1	Shaft	1
03	Scr25	Shaft-base screw	1
02	Scr25	Sft-shd supp scr	1

Manufacturing BOM for a Lamp

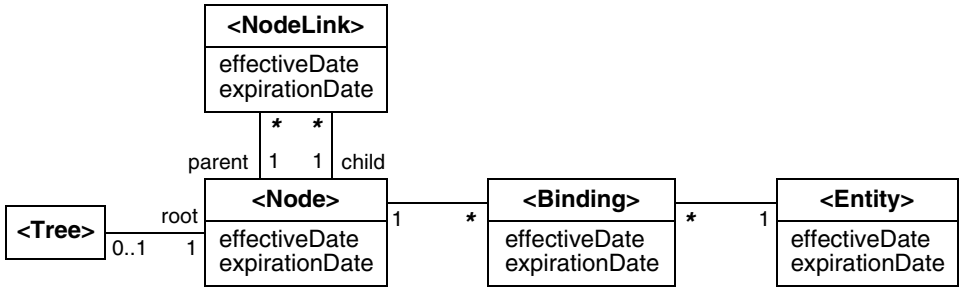
Level	PartName	PartRoleName	Qty
01	L101	Lamp	1
02	SA1	Socket asm	1
03	WA1	Wiring assembly	1
04	WAC1	Wiring asm core	1
05	Plg2	Plug	1
05	C2	Cord	1
05	PS1	Power switch	1
04	Scr1	Switch screw	2
03	Skt1	Socket	3
02	SS1	Shade support	1
02	Shd5	Shade	1
02	BA1	Base assembly	1
03	BAC1	Base asm core	1
04	B1	Base	1
04	Sft1	Shaft	1
04	Scr25	Shaft-base screw	1
03	M2	Mat	1
02	Scr25	Sft-shd supp scr	1

Figure 2.33 Overlapping tree: Sample data for mechanical parts model.

2.5 Tree Changing over Time Template

2.5.1 UML Template

For some applications, there is a need not only to store trees, but also to store the history of trees as they evolve over time. Figure 2.34 shows such a template. To represent change over time, the template separates an entity from its position in a tree (node). The timeline for an entity can differ from that of its various positions.



{All nodes have a parent except the root node. There may not be any cycles of nodes.}
{A child has at most one parent at a time.}

Figure 2.34 Tree changing over time: UML template. Use when you must store the history of a tree as it changes over time.

A ***Tree*** is a hierarchy of nodes and has one node as the root. A ***Node*** is a position within a *Tree*. An ***Entity*** is something with identity and data. A ***Binding*** is the coupling of an *Entity* to a *Node*. A ***NodeLink*** is the parent–child relationship between the *Nodes* of a *Tree*.

Strictly speaking, the template does not enforce a tree. The intent is that a child may have multiple parents over time, but only one parent at any time. The template implies that if an application changes the node at the root of a tree, it must start a new tree.

You can access a *Tree* by retrieving its *root* node and then retrieving the descendants for the desired time. Start with the *root Node* and then traverse from *Node* to *NodeLink* to *child Nodes* for the desired time. Repeat this traversal for each level to obtain the full tree. Each *Node* can be dereferenced to an *Entity* via *Binding*. (Even though the template does not enforce the constraint, each *Node* should bind to one *Entity* at a time.) Applications must carefully check to ensure that a tree is returned and catch any illegal data. The template itself permits nonsensical combinations of dates that do not yield a proper tree.

This template is already complex, so it is best to handle node names in a simple manner. Each node has a globally unique name and there is no provision to vary node name by context. The effective and expiration dates permit *Node* data and *Entity* data to vary over time.

2.5.2 IDEF1X Template

Figure 2.35 shows the template for trees that change over time using the IDEF1X notation. The following are foreign keys: *rootID* references *Node*, *nodeID* references *Node*, *entityID*

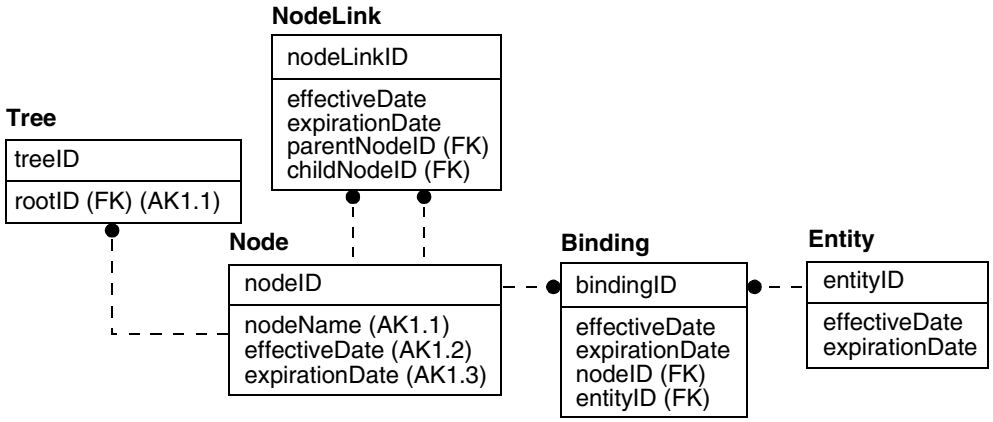


Figure 2.35 Tree changing over time: IDEF1X template.

references *Entity*, *parentNodeID* references *Node*, and *childNodeID* references *Node*. In Figure 2.35 the node name can change over time (three part candidate key—*nodeName* + *effectiveDate* + *expirationDate*), but the node name could also be invariant over time (candidate key of *nodeName* alone). Note that the handling of time reflects a limitation of relational DBMSs. It would be better to use time intervals but most relational DBMSs only support points in time.

2.5.3 SQL Queries

Figure 2.36 and Figure 2.37 show SQL queries for common traversals of the template. The colon prefix denotes variable values that must be provided for each query. A null *effectiveDate* means that a *Node* applies indefinitely from the past. A null *expirationDate* means that a *Node* applies indefinitely into the future.

2.5.4 Sample Populated Tables

Figure 2.38 shows sample tables for trees that change over time populated with data. For brevity Figure 2.38 only shows the *Node* and *NodeLink* tables. The ID values are arbitrary, but internally consistent. A null *effectiveDate* means that a *Node* applies indefinitely from the past. A null *expirationDate* means that a *Node* applies indefinitely into the future.

2.5.5 Example

This template can be difficult to follow, but it is powerful and has a compelling example.

Figure 2.16 shows a model of a management hierarchy where there is no change in structure over time. Figure 2.39 extends the model to track the evolution over time. The model can store the current reporting hierarchy, reporting hierarchies of the past, and planned hierarchies of the future. The hierarchy changes as persons join and leave a company. The hierarchy also changes due to promotions and demotions and management restructuring.

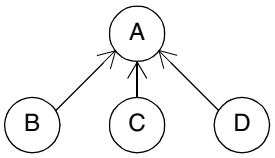
```
SELECT Parent.nodeID AS parentNodeID,
   Parent.nodeName AS parentNodeName
FROM Node AS Child
   INNER JOIN NodeLink AS NL ON Child.nodeID = NL.childNodeID
   INNER JOIN Node AS Parent ON NL.parentNodeID = Parent.nodeID
WHERE Child.nodeID = :aChildNodeID AND
   (Child.effectiveDate IS NULL OR
       :aDate >= Child.effectiveDate) AND
   (Child.expirationDate IS NULL OR
       :aDate <= Child.expirationDate) AND
   (NL.effectiveDate IS NULL OR
       :aDate >= NL.effectiveDate) AND
   (NL.expirationDate IS NULL OR
       :aDate <= NL.expirationDate) AND
   (Parent.effectiveDate IS NULL OR
       :aDate >= Parent.effectiveDate) AND
   (Parent.expirationDate IS NULL OR
       :aDate <= Parent.expirationDate);
```

Figure 2.36 Tree changing over time: SQL query. Find the parent for a child node.

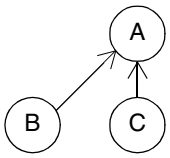
```
SELECT Child.nodeID AS childNodeID,
   Child.nodeName AS childNodeName
FROM Node AS Child
   INNER JOIN NodeLink AS NL ON Child.nodeID = NL.childNodeID
   INNER JOIN Node AS Parent ON NL.parentNodeID = Parent.nodeID
WHERE Parent.nodeID = :aParentNodeID AND
   (Child.effectiveDate IS NULL OR
       :aDate >= Child.effectiveDate) AND
   (Child.expirationDate IS NULL OR
       :aDate <= Child.expirationDate) AND
   (NL.effectiveDate IS NULL OR
       :aDate >= NL.effectiveDate) AND
   (NL.expirationDate IS NULL OR
       :aDate <= NL.expirationDate) AND
   (Parent.effectiveDate IS NULL OR
       :aDate >= Parent.effectiveDate) AND
   (Parent.expirationDate IS NULL OR
       :aDate <= Parent.expirationDate)
ORDER BY Child.nodeName;
```

Figure 2.37 Tree changing over time: SQL query. Find the children for a parent node.

A person may hold multiple positions over time and even two positions at the same time. Thus a person can be manager of one group and an acting manager of another group. The same person may be an individual contributor and a manager during different portions of a



Tree, 1 July 2000 0100



Tree, 1 July 2000 0300

Node table

node ID	node Name	effective Date	expiration Date
1	A		
2	B		
3	C		
4	D		1 July 2000 0100

NodeLink table

node link ID	effective Date	expiration Date	parent Node ID	child Node ID
1			1	2
2			1	3
3		1 July 2000 0100	1	4

Figure 2.38 Tree changing over time: Populated tables.

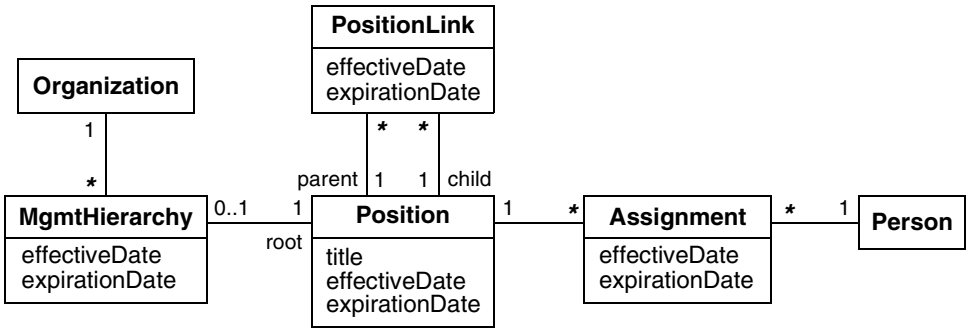


Figure 2.39 Tree changing over time: Evolving management hierarchy model.

career. The actual positions that are available also evolve. *Assignment* is an entity type because a person may hold the same position several times in a career.

The model provides matrix management (whether or not it is desired). This is because the model does not enforce a tree—that a child can only have a single parent at a time. Application code would need to provide such a constraint if it was desired.

2.6 Degenerate Node and Edge Template

2.6.1 UML Template

The degenerate node and edge template (Figure 2.40) is useful when there is a need to store data about the parent–child grouping. I call it *degenerate node and edge* because it is based on the *node and edge* directed graph template presented in the next chapter. This template rarely occurs.

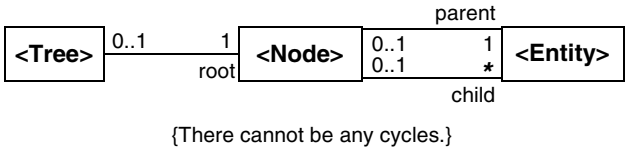


Figure 2.40 Degenerate node and edge: UML template. Use when you need to store data about the parent–child coupling.

A ***Tree*** is a hierarchy of entities and has one entity as the root. A ***Node*** is a position within a *Tree* and groups one parent *Entity* with all of its child *Entities*. An ***Entity*** is something with identity and data. The sequencing of the *Nodes* of a *Tree* occurs via the couplings to *Entities*. You need not show *Tree* in a use of the template.

In this template, *Nodes* have globally unique names as there is no context for defining the scope of uniqueness.

2.6.2 IDEF1X Template

Figure 2.41 restates Figure 2.40 with the IDEF1X notation. The following are foreign keys: *rootID* references *Node*, *nodeID* references *Node*, and *parentID* references *Entity*.

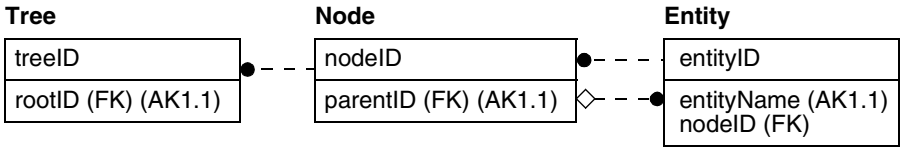


Figure 2.41 Degenerate node and edge: IDEF1X template.

2.6.3 SQL Queries

Figure 2.42 and Figure 2.43 show SQL queries for common traversals of the template. The colon prefix denotes variable values that must be provided for each query.

```
SELECT Parent.entityID AS parentEntityID,
   Parent.entityName AS parentEntityName
FROM Entity AS Child
   INNER JOIN Node AS N ON Child.nodeID = N.nodeID
   INNER JOIN Entity AS Parent ON N.parentID = Parent.entityID
WHERE Child.entityID = :aChildEntityID;
```

Figure 2.42 Degenerate node and edge: SQL query. Find the parent for a child node.

```
SELECT Child.entityID AS childEntityID,
   Child.entityName AS childEntityName
FROM Entity AS Child
   INNER JOIN Node AS N ON Child.nodeID = N.nodeID
   INNER JOIN Entity AS Parent ON N.parentID = Parent.entityID
WHERE Parent.entityID = :aParentEntityID
ORDER BY Child.entityName;
```

Figure 2.43 Degenerate node and edge: SQL query. Find the children for a parent node.

2.6.4 Sample Populated Tables

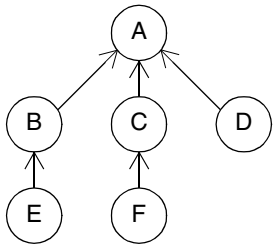
Figure 2.44 shows sample tables for the degenerate node and edge template populated with data. The ID values are arbitrary, but internally consistent.

2.6.5 Example

Figure 2.45 uses the degenerate node and edge template in a metamodel of a generalization tree for single inheritance. Each *Generalization* involves one *supertype* and one or more *subtypes*. An *EntityType* may participate in *Generalization* at most once as a *supertype* and at most once as a *subtype*. A *Generalization* may or may not be exhaustive—indicating whether or not every *supertype* record has a corresponding *subtype* record. The *discriminator* is a special *Attribute* that indicates the appropriate *subtype* record for each *supertype* record and may be implicit or explicitly noted in an application model.

In Figure 2.45 there is a need to store data about the parent–child grouping. The diagram notes whether each generalization level is exhaustive and its optional discriminator.

Figure 2.46 shows an excerpt of an application model with one level of generalization illustrating the metamodel. In Figure 2.46 a *ScheduleEntry* can be a *Meeting*, *Appointment*, *Task*, or *Holiday*. The generalization is exhaustive—each *ScheduleEntry* must be exactly one of these four possibilities. *ScheduleEntry* is the supertype and *Meeting*, *Appointment*, *Task*, and *Holiday* are subtypes.



Entity table

entityID	entityName	nodeID
1	A	
2	B	11
3	C	11
4	D	11
5	E	12
6	F	13

Node table

nodeID	parentID
11	1
12	2
13	3

Figure 2.44 Degenerate node and edge: Populated tables.

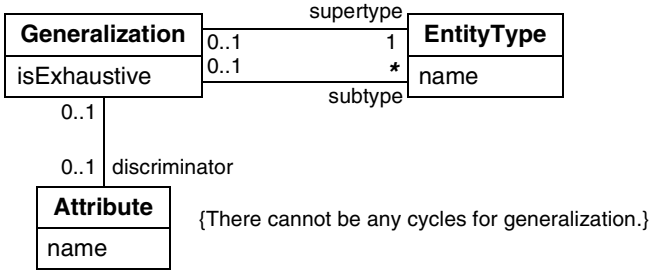


Figure 2.45 Degenerate node and edge: Metamodel for single inheritance.

2.7 Chapter Summary

Trees occur in many application models and are often a critical issue for representation. There are six templates for trees with different trade-offs.

- **Hardcoded tree**. Use when each level of a tree has a different entity type and the sequence of entity types is well known and unlikely to change.
- **Simple tree**. Suffices when tree decomposition is merely a matter of data structure and all nodes are the same.

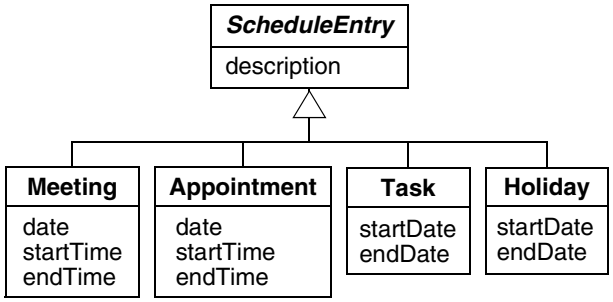


Figure 2.46 Excerpt of data model for calendar application.

- **Structured tree**. Use when branch nodes differ from leaf nodes. For example, the command *dir directoryFileName* elicits a different response from *dir dataFileName*. The structured tree is preferred when branch nodes and leaf nodes have different attributes, relationships, and/or semantics.
- **Overlapping trees**. Use when there are multiple trees and a node can belong to more than one tree.
- **Tree changing over time**. Records the history of a tree. This template permits storing of the past, present, and future content of trees.
- **Degenerate node and edge**. Use when there is data for the parent–child grouping.

Table 2.1 summarizes the tree templates.

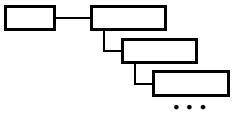
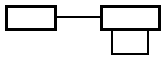
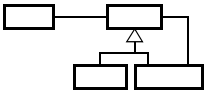
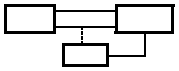
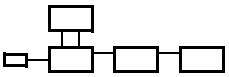
Bibliographic Notes

Figure 2.27 was partially motivated by [Fowler-1997], pages 21–22 but is a more powerful template capturing the constraint that a child has one parent for a tree.

References

[Fowler-1997] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Boston, Massachusetts: Addison-Wesley, 1997.

Table 2.1 Summary of the Tree Templates

Template name	Synopsis	UML template	Use when	Frequency
Hardcoded tree	Specifies a sequence of entity types, one for each level of the hierarchy.		A tree's structure is known and the types in the hierarchy are ordered.	Seldom
Simple tree	Restricts nodes to a single tree. Treats all nodes the same.		Tree decomposition is merely a matter of data structure.	Common
Structured tree	Restricts nodes to a single tree. Differentiates leaf nodes from branch nodes.		Branches and leaves have different attributes, relationships, and/or semantics.	Common
Overlapping trees	Permits a node to belong to multiple trees. Treats all nodes the same.		A node can belong to more than one tree.	Occasional
Tree changing over time	Stores multiple variants of a tree. Extract a particular tree by specifying a time.		A tree changes over time and you must store the history.	Occasional
Degenerate node and edge	Groups a parent with its children. The grouping itself can be described.		There is data for the parent–child grouping.	Rare

Note: This table can help you choose among the tree templates.