Patterns of Data Modeling

Michael R. Blaha
Modelsoft consulting Corp
E-mail: blaha@computer.org
www.modelsoftcorp.com

ICOODB
September 2010

Section 1: Introduction

Definitions

- ***Model***: an abstraction of some aspect of a problem.
- ***Data model***: a model that lets you understand the structure of data.
  - Do not model problems literally
  - Instead search for the deep, inner essence of a problem.
  - Such a model accommodates change and is less costly to develop.
  - It is straightforward to implement a data model.
- ***Pattern***: a model fragment that is profound and recurring.
  - Patterns focus on structure (classes and relationships). Attributes provide fine details that vary for specific applications.

Our focus here is on patterns for data models and databases.

Pattern Definitions from the Literature

The definition of ***pattern*** varies in the literature.

- [Alexander-1979]. A solution to a problem in context.
- [Buschmann-1996]. Describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution.
- [Coad-1995]. A template of interacting objects, one that may be used again and again by analogy.
- [Erl-2009] A proven solution to a common problem individually documented in a consistent format and usually as part of a larger collection.
- [Fowler-1997]. An idea that has been useful in one practical context and will probably be useful in others.
- [Gamma-1995] Systematically names, motivates, and explains a general design that addresses a recurring design problem... It describes the problem, the solution, when to apply the solution, and its consequences.

Why are Patterns Important?

- **Enriched modeling language**. Patterns provide a higher level of building blocks than modeling primitives. Patterns are prototypical modeling fragments that distill the knowledge of experts.
- **Improved documentation**. Patterns offer standard forms that improve modeling uniformity.
- **Reduced modeling difficulty**. Many developers find modeling difficult because of the intrinsic abstraction. Patterns are all about abstraction and give developers a better place to start.
- **Faster modeling**. Developers do not have to create everything from scratch and can build on the accomplishments of others.
- **Better models**. Patterns reduce mistakes and rework. Carefully considered patterns are more likely to be correct and robust than an untested, custom solution.

Drawbacks of Patterns

- **Sporadic coverage**. You cannot build a model by just combining patterns. Typically you will use only a few patterns, but they often embody key insights.
- **Pattern discovery**. It can be difficult to find a pattern, especially if your idea is ill-formed.
- **Complexity**. Patterns are an advanced topic and can be difficult to understand.
- **Inconsistencies**. There has been a real effort in the literature to cross reference other work and build on it. However, inconsistencies still happen.
- **Immature technology**. The patterns literature is active but the field is still evolving.

Pattern vs. Seed Model

Most of the database literature confuses patterns with seed models.

- ***Seed model***: a model that is specific to a problem domain.
  - Provides a starting point for applications from its problem domain.

	Pattern	Seed model
Applicability	Application independent	Application dependent
Scope	An excerpt of a model	Intended to be the starting point for an application
Model size	Typically <10 classes	Typically 10-50 classes
Abstraction	More abstract	Less abstract
Model type	Can be described with a data model	Can be described with a data model

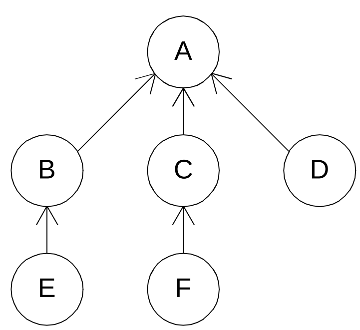
Section 2: Aspects of Pattern Technology

- ***Mathematical template***: an abstract model fragment that is devoid of application content.
  - Driven by deep data structures that often arise in database models.
  - Notation: Angle brackets denote parameters that are placeholders.
- ***Antipattern***: a characterization of a common software flaw.
  - Shows what not to do and how to fix it
- ***Archetype***: a deep concept that is prominent and cuts across problem domains.
- ***Identity***: the means for denoting individual objects, so that they can be found.
- ***Canonical model***: a submodel that provides a useful service for many applications

The remaining lecture will partially cover the above topics.

Section 3: Mathematical Template — Tree

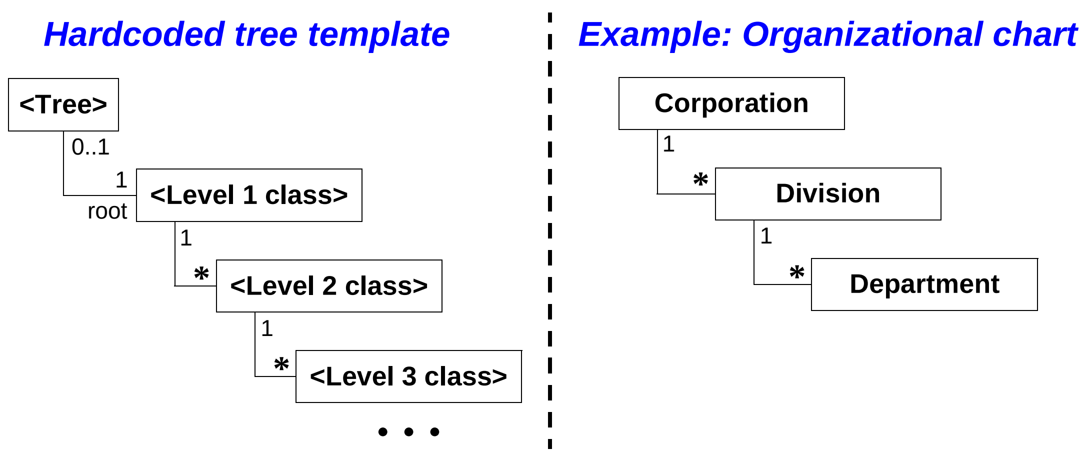
- ***Tree***: a term from graph theory.
  - A tree is a set of nodes that connect from child to parent. Each node has one parent node except for the node at the tree's top.
  - A node can have many (zero or more) child nodes.
  - There are no cycles — at most one path connects any two nodes.
- An example of a tree...



Six Tree Templates

- **Hardcoded tree**. Hardcodes types, one for each level of the tree.
- **Simple tree**. Restricts nodes to a single tree. Treats nodes the same.
- **Structured tree**. Restricts nodes to a single tree. Differentiates leaf nodes from branch nodes.
- **Overlapping trees**. Permits a node to belong to multiple trees. Treats nodes the same.
- **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.
- **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.

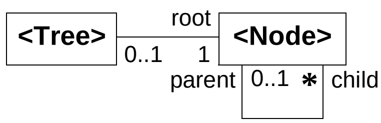
Hardcoded Tree



- 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.
  - In practice, used for examples, but seldom for code.

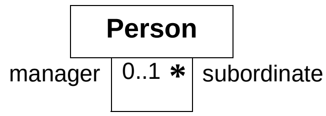
Simple Tree

Simple tree template



{All nodes have a parent except the root node.}
{There cannot be any cycles.}

Example: Management hierarchy

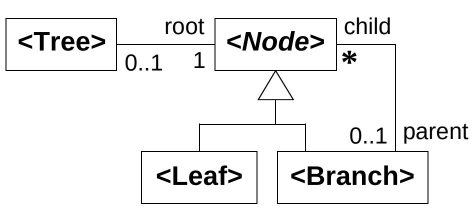


{Every person has a manager, except the CEO.}
{The management hierarchy must be acyclic.}

- Use when:
  - Tree decomposition is merely a matter of data structure.
- Node names can be globally unique or unique within the context of a parent.

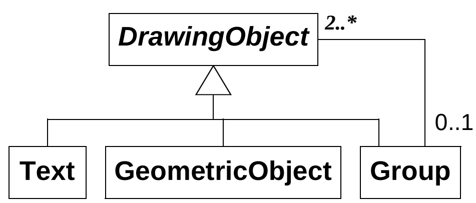
Structured Tree

Structured tree template



{All nodes have a parent except the root node.}
{There cannot be any cycles.}

Example: Graphical editor

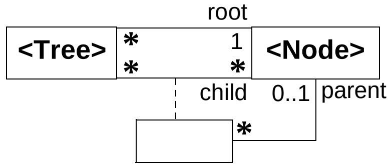


{The group hierarchy must be acyclic.}

- Use when:
  - Branch nodes and leaf nodes have different attributes, relationships, and/or behavior.
- Node names can be globally unique or unique within the context of a parent.

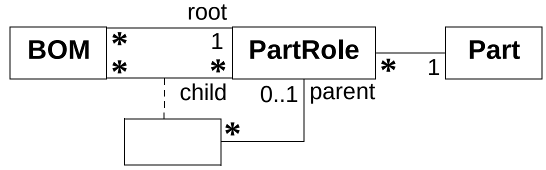
Overlapping Trees

Overlapping trees template



{All nodes have a parent except the root node.}
{There cannot be any cycles.}
{A parent must only have children for trees to which the parent belongs.}

Example: Mechanical parts

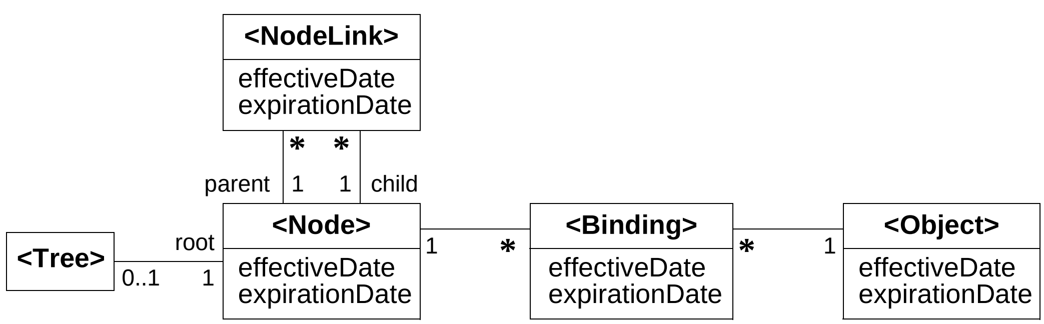


{Each BOM must be acyclic.}

- Use when:
  - A node can belong to multiple trees.
  - Example: A part can have several bill-of-materials, such as one for manufacturing, another for engineering, and another for service.
- Motivated by [Fowler, page 21] but a more powerful template capturing the constraint that a child has at most one parent for a tree.

Tree Changing Over Time

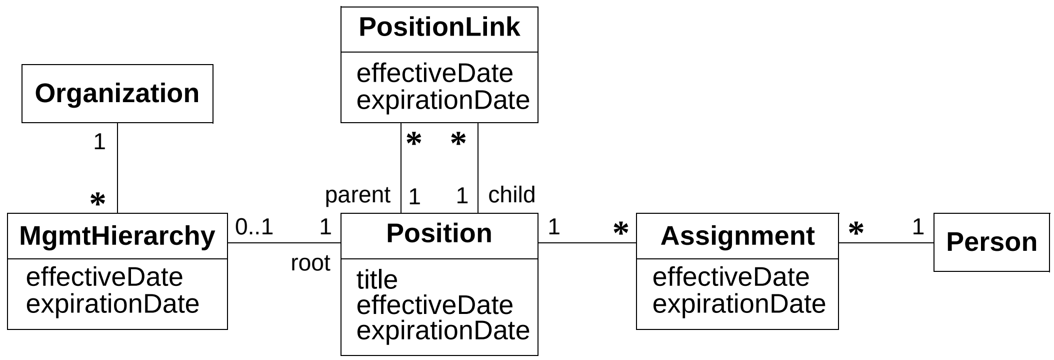
Tree changing over time template



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

Tree Changing Over Time (continued)

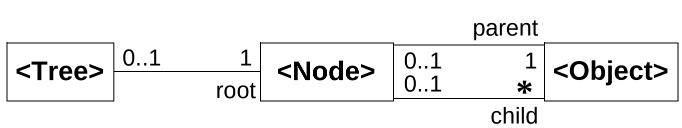
Example: management hierarchy



- Use when:
  - The history of a tree must be recorded.

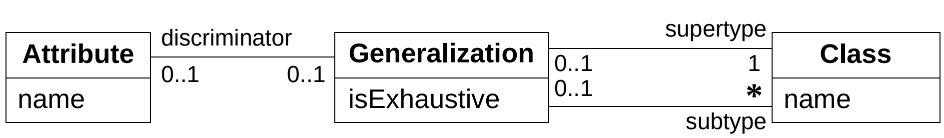
Degenerate Node and Edge

Degenerate node and edge template



{There cannot be any cycles.}

Example: Single inheritance



{There cannot be any cycles.}

- Use when:
  - The grouping of a parent and its children must be described.

Additional Templates

There are additional templates.

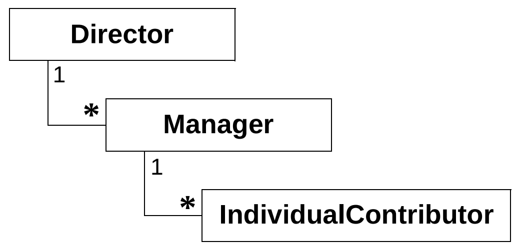
- Directed graph.
  - **Simple DG**. Treats all nodes the same.
  - **Structured DG**. Differentiates leaf nodes from branch nodes.
  - **Node-edge DG**. Treats nodes and edges as peers.
  - **Connection DG**. Promotes a node-edge connection to a class.
  - **Simple DG changing over time**. Stores variants of a simple DG over time.
  - **Node-edge DG changing over time**. Stores variants of a node-edge DG over time.

Additional Templates (continued)

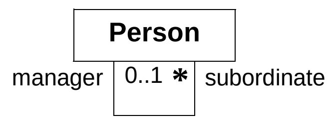
- Undirected graph.
  - **Node-edge UDG**. Treats nodes and edges as peers.
  - **Connection UDG**. Promotes a node-edge connection to a class.
  - **UDG changing over time**. Stores variants of a node-edge UDG over time.
- Item description.
  - **Item description**. Relates data and metadata in the same model.
  - **Homomorphism**. Expresses an analogy between two item description templates.
- Star schema.
  - **Star schema**. Represents data as facts that are bound to dimensions.

Section 5: Mathematical Template — Example

Management Template — Hardcoded Tree



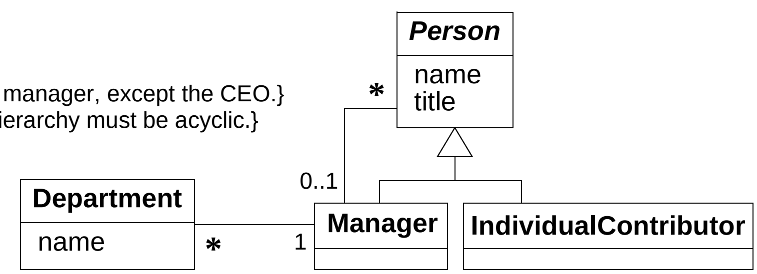
Management Template — Simple Tree



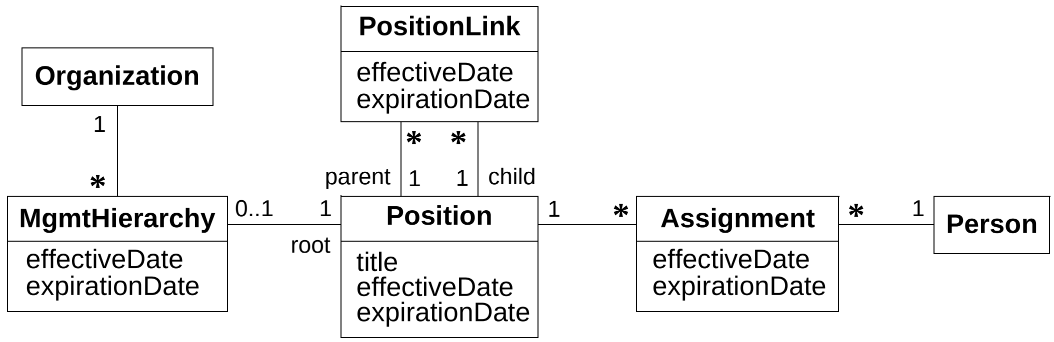
{Every person has a manager, except the CEO.}
{The management hierarchy must be acyclic.}

Management Template — Structured Tree

{Every person has a manager, except the CEO.}
{The management hierarchy must be acyclic.}



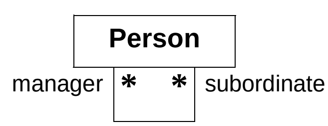
Mgmt Template — Tree Changing Over Time



The model provides matrix management. 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.

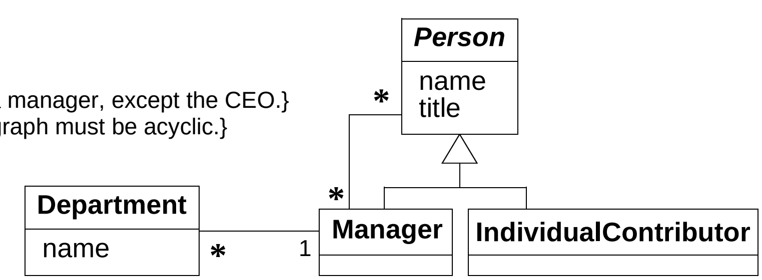
Management Template — Simple Directed Graph



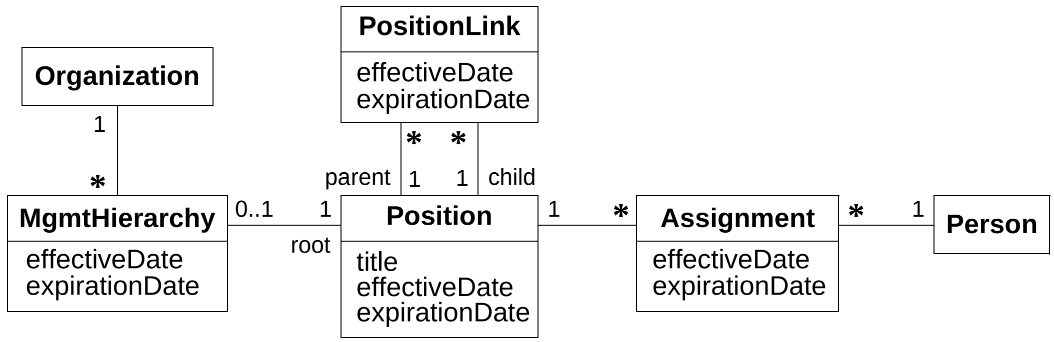
{Every person has a manager, except the CEO.}
{The management graph must be acyclic.}

Mgmt Template — Structured Directed Graph

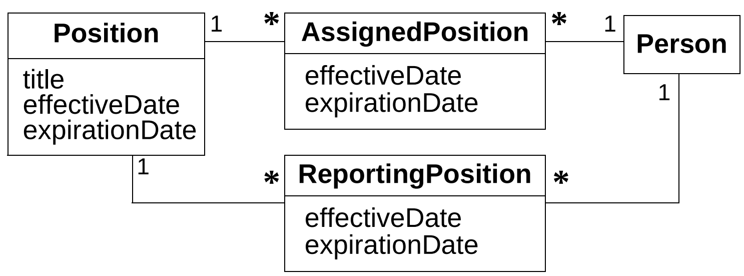
{Every person has a manager, except the CEO.}
{The management graph must be acyclic.}



Mgmt Template — Simple DG Changing Over Time



Inferior Mgmt Model — Changing Over Time

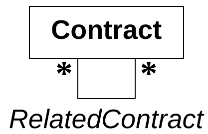


Section 6: Antipatterns

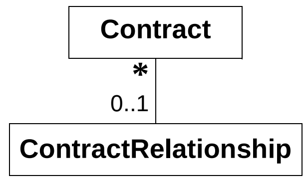
- ***Antipattern***: a characterization of a software flaw. When you find an antipattern, substitute the correction.
  - ***Universal antipattern*** — avoid for all applications.
  - ***Non-data-warehouse antipattern*** — acceptable for data warehouses, but avoid them otherwise.
- Patterns are good ideas that can be reused.
  In contrast, antipatterns look at what can go wrong.
- The literature focuses on antipatterns for programming code, but antipatterns also apply to data models.
- [Brown-98]. An antipattern is some repeated practice that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results.

Universal Antipattern: Symmetric Relationship

Antipattern example



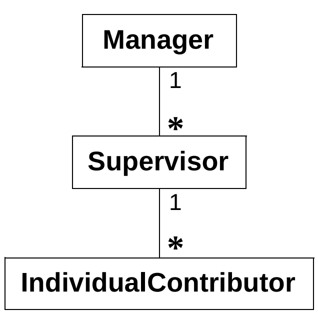
Improved model



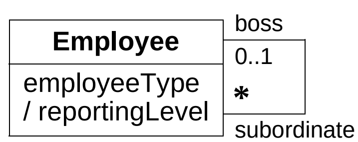
- ***Observation***: There is a self relationship with the same multiplicity and role names on each end.
  - Symmetric relationships are always troublesome for relational databases.
    - Which column is first? Which column is second?
    - Double entry or double searching of data.
- ***Improved model***: Promote the relationship to a class in its own right. The improved model is often more expressive.

Universal Antipattern: Artificial Hardcoded Levels

Antipattern example



Improved model



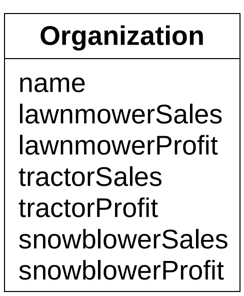
- ***Observation***: There is a fixed hierarchy with little difference between the levels.
  - Contrast with the hardcoded tree template where there is a material difference between the levels.
- ***Improved model***: Abstract and consolidate the levels. Use one of the tree patterns to relate the levels.

Additional Universal AntiPatterns

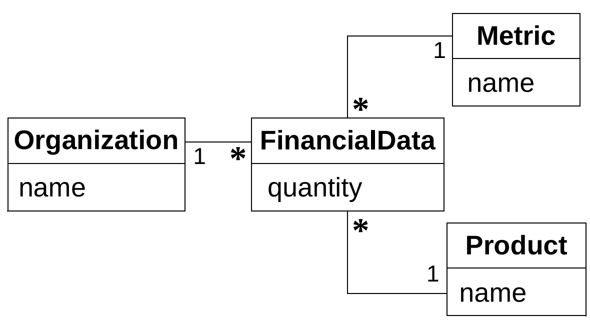
- **Dead elements**. A model has unused elements (classes, relationships, attributes).
- **Disguised fields**. Field names do not describe data.
  - User-defined fields, mislabeled fields, binary fields, anonymous fields, overloaded fields.
- **Excessive generalization**. There is a deep generalization. Extensive taxonomies are often motivated by object-oriented programming and are inadvisable.
- **Disconnected classes**. A model has free-standing classes. From the problem understanding, it would seem that they should be related.
- **Modeling errors**. A model has one or more serious conceptual flaws.
- **Multiple inheritance**. A model has multiple inheritance. Multiple inheritance is not appropriate for data models.
- **Paradigm degradation**. A database is degraded to some other paradigm.

Non-DW Antipattern: Parallel Attributes

Antipattern example



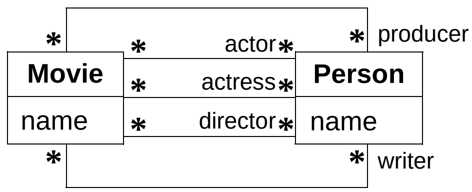
Improved model



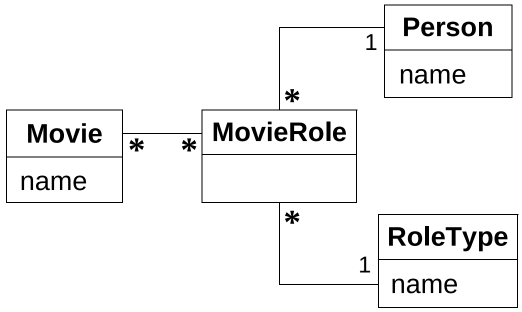
- ***Observation***: A class has groups of similar attributes. Such a model can be brittle, verbose, and awkward to extend.
- ***Exceptions***: OK for data warehouses.
- ***Improved model***: Abstract and factor out commonality.
  - The improved model can handle new products and financial metrics.

Non-DW Antipattern: Parallel Relationships

Antipattern example



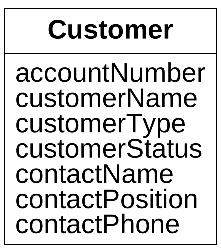
Improved model



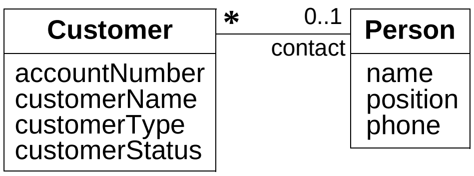
- ***Observation***: Two classes have several (at least three) similar relationships.
- ***Exceptions***: OK for data warehouses.
- ***Improved model***: Abstract and factor out commonality.

Non-DW Antipattern: Combined Classes

Antipattern example



Improved model



- ***Observation***: A class has disparate attributes and lacks cohesion.
  - The contact position and contact phone depend on the contact name which in turn depends on the customer.
  - Several customer records could have the same contact name with inconsistent positions and phones.
- ***Exceptions***: OK for data warehouses.
- ***Improved model***: Make each concept its own class.

Section 7: Antipattern Example

Reverse Engineering the LDAP Standard

- LDAP = Lightweight Directory Access Protocol
  - LDAP is a public standard that has two primary purposes: user authentication and sharing basic data across applications.
  - LDAP was originally implemented with files, but we will study a product with a database implementation.
  - The LDAP schema is by intent a meta-schema that stores both a model and the model's data.
- My motive was to reverse engineer the database so that my client could better understand the product.
- Available inputs.
  - Schema: tables, attributes, data types, nullability, and primary keys.
  - Data.
  - A book explaining LDAP concepts.

LDAP Reverse Engineering: Original Schema

I was given a printout of a SQL server schema.

DsTimestamp

	Column Name	Datatype	Length	Precision	Scale	Allow Nulls	Identity
	i_Replication_Key	int	4	10	0		✓
	dt_SchemaTimestamp	datetime	8	0	0	✓	
	dt_DitTimestamp	datetime	8	0	0	✓	
	dt_ReplicationTimestamp	datetime	8	0	0	✓	
	dt_GroupTimestamp	datetime	8	0	0	✓	

There were a total of 11 tables. *DsTimestamp* is one of the tables.

LDAP Reverse Engineering: Original Schema

First, I typed the schema into a modeling tool (three slides).

Configuration

replicationKey[1..1]:int(4) {pk}
id[1..1]:int(4)
containerPartitionID:int(4)
containerDbID:int(4)
peKey:varchar(255)

AttributeContainers

replicationKey[1..1]:int(4) {pk}
aid[1..1]:int(4)
containerClsID[1..1]:int(4)
required[1..1]:bit

ObjectAttributes

dsID[1..1]:int(4) {pk}
sequence[1..1]:int(4) {pk}
aid[1..1]:int(4) {pk}
vcVal:varchar(255)
iVal:int(4)
vbVal:varbinary(255)
imgVal:image
dtVal:datetime
expiresTime:datetime

DsTimestamp

replicationKey[1..1]:int(4) {pk}
schemaTimestamp:datetime
ditTimestamp:datetime
replicationTimestamp:datetime
groupTimestamp:datetime

ObjectLookup

dsID[1..1]:int(4) {pk}
entryName[1..1]:varchar(255)
objectClass[1..1]:int(4)
containerDsID:int(4)
dseType[1..1]:int(4)
creatorsName:varchar(255)
createTimestamp:datetime
modifiersName:varchar(255)
modifyTimestamp:datetime
acl:image
expiresTime:datetime

Attributes have a name, nullability, datatype, and primary key flag.

LDAP Rev Engr: Original Schema (continued)

ClassContainers

replicationKey[1..1]:int(4) {pk}
clsID[1..1]:int(4)
containerClsID[1..1]:int(4)

Classes

clsID[1..1]:int(4) {pk}
name[1..1]:varchar(255)
oid:varchar(255)
description:varchar(255)
rdnAid[1..1]:int(4)
guid[..1]:char(39)
dseDitType[1..1]:int(4)
displayName:varchar(255)
isSecurityPrincipal[1..1]:bit
containerType[1..1]:int(4)
defaultSecurityDecriptor:image
acl:image

DsConfiguration

serverID[1..1]:int(4) {pk}
instanceID[1..1]:int(4)
serverName[1..1]:varchar(255)
dynamicDbFlags[1..1]:int(4)
replicationFlags[1..1]:int(4)
replicationProto[1..1]:varchar(255)
replicationEndP[1..1]:varchar(255)
replicationQSize[1..1]:int(4)
replicationLagTime[1..1]:int(4)
replicationBuffSize[1..1]:int(4)
replicationSyncTime[1..1]:int(4)
replicationInfo[1..1]:varchar(255)

Subrefs

namespacePartitionID[1..1]:int(4) {pk}
subrefEntry:varchar(255)
subrefPrentID:int(4)
valuePartitionCount[1..1]:int(4)

LDAP Rev Engr: Original Schema (continued)

DsoGrid
serverID[1..1]:int(4) {pk}
namespacePartitionID[1..1]:int(4)
valuePartitionID[1..1]:int(4)
dsoType[1..1]:int(4)
datasource[1..1]:varchar(255)
database[1..1]:varchar(255)
login:varchar(255)
password:varchar(255)
maxCnx:int(4)
timeout:int(4)
replicationType:int(4)

Attributes
aid[1..1]:int(4) {pk}
name[1..1]:varchar(255)
oid:varchar(255)
description:varchar(255)
dataType[1..1]:int(4)
multiValued[1..1]:bit
searchble[1..1]:bit
guid[1..1]:char(39)
syntax[1..1]:int(4)
displayName:varchar(255)
constraints:varchar(255)
acl:image

LDAP Reverse Engineering: Observations

- The schema has a strong and uniform style.
  - Primary key fields are IDs, *replicationKey*, and *sequence*.
  - All primary key fields are int(4).
- ***Antipattern***: Parallel attributes.
  - *ObjectAttributes* has parallel attributes: *vcVal*, *iVal*, *vbVal*, *imgVal*, and *dtVal*. Apparently, each record fills in the one field with the appropriate data type.
  - The usage is very limited and seems OK here.
- ***Antipattern***: Disguised (overloaded) fields.
  - *ObjectAttributes*.*iVal* is used to store both integers and IDs (essentially pointers to objects). I determined this by inspecting data.
  - Thus the LDAP standard subverts referential integrity. This is largely a consequence of LDAP's heritage of being designed for files.

LDAP Reverse Engineering: Observations (cont.)

- ***Antipattern***: Modeling error,
  - There can be many *ClassContainers* for the same contained *Classes* and container *Classes*.
  - This lets a class contain multiple copies of a class.
  - Apparently, the multiple copies do not have different roles. This is odd. There is no way to distinguish the multiple copies.
- ***Antipattern***: Paradigm degradation.
  - The LDAP standard forces data into a hierarchical structure. A hierarchy is adequate for simple data. It distorts a complex data structure (unlike the neutral structure of relational databases).
  - LDAP degrades use of a relational database. It foregoes referential integrity and uses pointers that programming code must handle.

Section 8: Archetypes

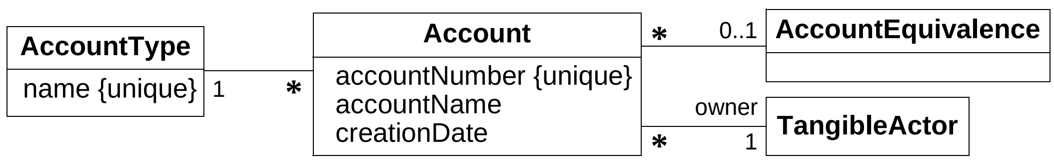
Archetype: an abstraction that often occurs and transcends individual applications.

- Archetypes are similar in style to other data pattern books and their emphasis on seed models.
  - The difference is that archetypes emphasize the core concepts and omit application details.
- The term archetype is taken from [Arlow-2004].

Archetype: Account

Account — a label for recording and reporting a quantity of something.

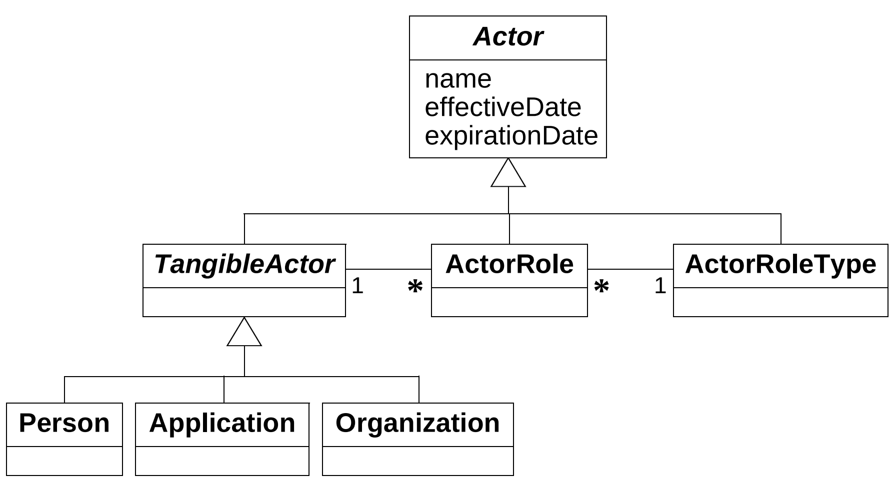
- An owner can have multiple accounts for an account type.
- Some accounts can be unwanted duplicates and remain undetected.
- *AccountEquivalence* can logically combine accounts without having to move data (see the ***Symmetric Relationship Antipattern***).



Archetype: Actor

Actor — someone or something that is notable in terms of data or relationships.

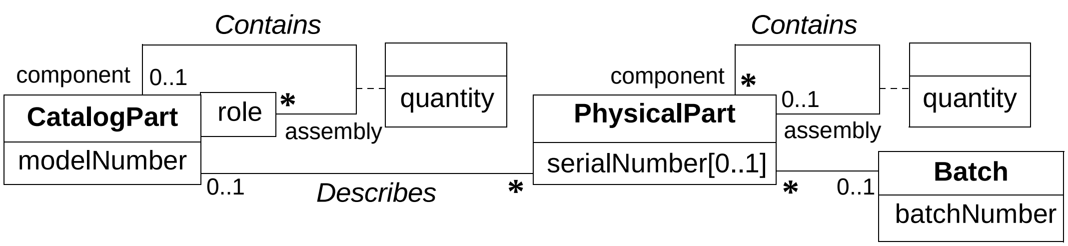
- Useful as a “hook” for permissions, approvals, logging...
- This archetype is consistent with the literature but is more robust, adding roles and applications.



Archetype: Part

Part — a specific good that can be described.

- ***PhysicalPart*** — a tangible thing.
  – Customer service records refer to physical cars.
  – Occurrences form a collection of trees.
- ***CatalogPart*** — a description of a similar group of things.
  – Design documents describe car models.
  – Occurrences form a directed acyclic graph.



Additional Archetypes

- ***Address*** — a means for communicating with an actor.
- ***Asset*** — something of value.
- ***Contract*** — an agreement for the supply of products.
- ***Course*** — a series of lessons about a subject.
- ***Customer*** — someone involved in the purchase of products.
- ***Document*** — a physical or electronic representation of a body of information.
- ***Event*** — an occurrence at some point in time.
- ***Flight*** — the travel by an airplane between airports.
- ***Item*** — a part or a service.
- ***Location*** — a physical place in space.

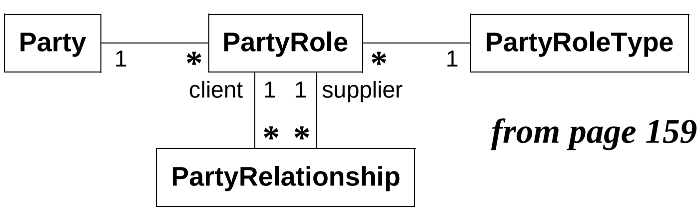
Additional Archetypes (continued)

- ***Opportunity*** — an inquiry that can result in business.
- ***Payment*** — the assignment of money in return for something of value.
- ***Position*** — a job held by someone in an organization.
- ***Product*** — the packaging of a physical item for a particular marketplace.
- ***Role*** — a function played by someone or something.
- ***Transaction*** — an exchange that must be completed in its entirety or not at all.
- ***Vendor*** — someone involved in the sale of products.

Section 9: Pattern Literature

Jim Arlow and Ila Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*. Boston: Addison-Wesley, 2004.

- Their archetype models are large and more like seed models.
  - Small archetype models are more likely to be application independent and reusable.
- They distinguish between client and supplier. This is a modeling error. This is completely unnecessary, given that they have roles.



from page 159

- The book focuses on design and programming.
- Data modeling notation: UML class model.

Pattern Literature (continued)

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

- Fowler discusses different application domains and gradually elaborates the seed models, explaining important abstractions along the way.
  - Most of his examples are from health care, finance, accounting, and the stock market.
- Data modeling notation: IE-like notation with object-oriented jargon.
- This is an excellent book.

Pattern Literature (continued)

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995.

- Focuses on issues of programming design.
  - They don't cover databases.
- Discusses abstract patterns that transcend individual programs.
  - This stands in contrast to most of the database pattern books.
- Data modeling notation: OMT class model notation (a precursor to the UML).
- This is a seminal work.

Pattern Literature (continued)

David C. Hay. *Data Model Patterns: Conventions of Thought*. New York, New York: Dorset House, 1996.

- Presents seed models for a wide variety of applications areas.
  - Person and Organization
  - Product
  - Procedure
  - Contract
  - Laboratory
  - Material planning
  - Process manufacturing
  - Document
- Data modeling notation: Richard Barker et al's (Oracle notation).
- This is an excellent book.

Pattern Literature (continued)

Len Silverston. *The Data Model Resource Book, Volume 1*. New York, New York: Wiley, 2001.

Len Silverston. *The Data Model Resource Book, Volume 2*. New York, New York: Wiley, 2001.

- Vol 1 presents seed models for a wide variety of applications areas.
  - Person and Organization
  - Product, Order, Shipment
  - Work effort
  - Invoice, Accounting, Budgeting
  - Human Resources
- Vol 2 presents seed models for a variety of industries.
- Data modeling notation: Richard Barker et al's (Oracle notation).

Pattern Literature (continued)

Len Silverston and Paul Agnew. *The Data Model Resource Book, Volume 3*. New York, New York: Wiley, 2009.

- Chapters 2 and 3 have an excellent discussion of *party* (comparable to *actor* in this book). They distinguish between a declarative role (a role that a person or organization plays within an entire enterprise) and a contextual role (a role in a specific relationship).
- Volume 3 is an excellent book. The scope is limited, but the book is abstract and incisive.
- Data modeling notation: Richard Barker et al's (Oracle notation).
  - Uses this notation for consistency with earlier books, even though the notation is dated.