Section 1: Introduction

Pattern Definitions From 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.
- [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] Explains a general design that addresses a recurring design problem. Describes the problem, the solution, when to apply the solution, and its consequences.
- [Blaha-2010] A model fragment that is profound and recurring.
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.

<table>
<thead>
<tr>
<th></th>
<th>Pattern</th>
<th>Seed model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Applicability</strong></td>
<td>Application independent</td>
<td>Application dependent</td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>An excerpt of a model</td>
<td>Intended to be the starting point for an application</td>
</tr>
<tr>
<td><strong>Model size</strong></td>
<td>Typically &lt;10 classes</td>
<td>Typically 10-50 classes</td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td>More abstract</td>
<td>Less abstract</td>
</tr>
<tr>
<td><strong>Model type</strong></td>
<td>Can be described with a data model</td>
<td>Can be described with a data model</td>
</tr>
</tbody>
</table>

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 focus on the first two 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...

![Tree Diagram]

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

**Hardcoded tree template**

```
<Tree>
  0..1
  1
  root
  <Level 1 class>
    0..1
    1
    *<Level 2 class>
      1
      *
      <Level 3 class>
      ...
```

**Example: Organizational chart**

- Corporation
- Division
- Department

**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**

```
<Tree>
  root
  0..1
  1
  parent
  <Node>
    0..1
    *
    child

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

**Example: Management hierarchy**

- Person
- Manager
- Subordinate

**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**

![Structured tree template diagram]

- **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.

- **Example: Graphical editor**
  - DrawingObject
  - Text
  - GeometricObject
  - Group

- **Constraints:**
  - All nodes have a parent except the root node.
  - There cannot be any cycles.

### Overlapping Trees

**Overlapping trees template**

![Overlapping trees template diagram]

- **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.

- **Example: Mechanical parts**
  - BOM
  - PartRole
  - Part

- **Constraints:**
  - Each BOM must be acyclic.
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.}

- Note that the data structure does not enforce that a Node has at most one parent at any time. Application code would need to enforce this constraint.

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**

![Diagram of a tree data model]

{There cannot be any cycles.}

**Example: Single inheritance**

![Diagram of a class hierarchy]

{There cannot be any cycles.}

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

---

**Section 4: Mathematical Template — Additional Templates**

**Additional Templates**

There are templates for additional data structures...

- Directed graph.
- Undirected graph.
- Item description.
- Star schema.

*I welcome suggestions for other important data structures that can be characterized with templates.*
Section 5: Mathematical Template — Example

Here are alternative patterns for expressing the data structure of a corporate management hierarchy.

Management Template — Simple Tree

```
Person
 manager 0..1 subordinate

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

Management Template — Structured Tree

```
Person
   name
   title

Department
   name

Manager
   0..1

IndividualContributor

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

Mgmt Template — Tree Changing Over Time

```
Organization
   1
   *

MgmtHierarchy
   effectiveDate
   expirationDate
   0..1
   root

PositionLink
   effectiveDate
   expirationDate
   *
   *
   parent
   1
   1
   child

Position
   title
   effectiveDate
   expirationDate
   1

Assignment
   effectiveDate
   expirationDate
   *
   1

Person
```

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

```
Person
manager * * subordinate

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

Mgmt Template — Structured Directed Graph

```
Person
  name
  title

Manager

IndividualContributor

Department
  name

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

Mgmt Template — Simple DG Changing Over Time

```
Organization

MgmtHierarchy
  effectiveDate
  expirationDate

Position
  title
  effectiveDate
  expirationDate

PositionLink
  effectiveDate
  expirationDate

Assignment

Person
```

©2011 Michael R. Blaha
Patterns of Data Modeling
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**
  - `Contract` ★
  - `RelatedContract` ★

- **Improved model**
  - `Contract` ★
  - `ContractRelationship` 0..1

- **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**

- **Manager**
  - 1
  - *

- **Supervisor**
  - 1
  - *

- **IndividualContributor**

**Improved model**

- **Employee**
  - boss
  - 0..1
  - *
  - subordinate

  - **employeeType**
  - / reportingLevel

**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.

Non-DW Antipattern: Parallel Attributes

**Antipattern example**

- **Organization**
  - name
  - lawnmowerSales
  - lawnmowerProfit
  - tractorSales
  - tractorProfit
  - snowblowerSales
  - snowblowerProfit

**Improved model**

- **Metric**
  - name

- **FinancialData**
  - name
  - quantity

- **Product**
  - name

**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: Combined Classes

**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.

<table>
<thead>
<tr>
<th>Customer</th>
<th>0..1</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>accountNumber</td>
<td>contact</td>
<td></td>
</tr>
<tr>
<td>customerName</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>customerType</td>
<td>position</td>
<td></td>
</tr>
<tr>
<td>customerStatus</td>
<td>phone</td>
<td></td>
</tr>
<tr>
<td>contactPosition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>contactPhone</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Datatype</th>
<th>Length</th>
<th>Precision</th>
<th>Scale</th>
<th>Allow Nulls</th>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_Repl{</td>
<td>[1..1]}:int(4) {pk}</td>
<td>int</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dt_SchemaTimestamp</td>
<td>datetime</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>dt_DitTimestamp</td>
<td>datetime</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>dt_Repl{[1..1]}:int(4)</td>
<td>datetime</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>dt_GroupTimestamp</td>
<td>datetime</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>Configuration</th>
<th>AttributeContainers</th>
<th>ObjectAttributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>replicationKey[1..1]:int(4)</td>
<td>replicationKey[1..1]:int(4) {pk}</td>
<td>dsID[1..1]:int(4) {pk}</td>
</tr>
<tr>
<td>id[1..1]:int(4)</td>
<td>aid[1..1]:int(4)</td>
<td>sequence[1..1]:int(4)</td>
</tr>
<tr>
<td>containerPartitionID:int(4)</td>
<td>containerClsID[1..1]:int(4)</td>
<td>aid[1..1]:int(4) {pk}</td>
</tr>
<tr>
<td>containerDbID:int(4)</td>
<td>required[1..1]:bit</td>
<td>vcVal:varchar(255)</td>
</tr>
<tr>
<td>peKey:varchar(255)</td>
<td></td>
<td>iVal:int(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vbVal:varbinary(255)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>imgVal:image</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dtVal:datetime</td>
</tr>
<tr>
<td>DsTimestamp</td>
<td></td>
<td>expiresIn:datetime</td>
</tr>
<tr>
<td>replicationKey[1..1]:int(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>schemaTimestamp:datetime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ditTimestamp:datetime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>replicationTimestamp:datetime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>groupTimestamp:datetime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>entryName[1..1]:varchar(255)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>objectClass[1..1]:int(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>containerDsID:int(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dseType[1..1]:int(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>creatorsName:varchar(255)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>createTimestamp:datetime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>modifiersName:varchar(255)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>modifyTimestamp:datetime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>acl:image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expiresTime:datetime</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

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)

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)
- dsseDitType[1..1]:int(4)
- displayName:varchar(255)
- isSecurityPrincipal[1..1]:bit
- containerType[1..1]:int(4)
- defaultSecurityDescriptor:image
- acl:image

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

Attributes
- aid[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)
- aid:varchar(255)
- name:varchar(255)
- oid:varchar(255)
- description:varchar(255)
- dataType[1..1]:int(4)
- multiValued[1..1]:bit
- searchable[1..1]:bit
- guid[..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: Pattern Literature


- 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.

![Diagram](image.png)

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

Pattern Literature (continued)


- 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)


- 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. (Hays has just come out with a new book.)
Pattern Literature (continued)


- 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)


- 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.