8

Universal Antipatterns

An ***antipattern*** is a characterization of a common software flaw. When you find an antipattern, substitute the correction. ***Universal antipatterns*** are antipatterns that you should avoid for all applications.

8.1 Symmetric Relationship Antipattern

8.1.1 Observation

An entity type has a self relationship with the same multiplicity and role names on each end. Typically this is a many-to-many self relationship. Symmetric relationships can be troublesome for programming and are always troublesome for relational databases.

8.1.2 Exceptions

There are no exceptions for relational database designs. Avoid symmetric relationships.

8.1.3 Resolution

Promote the relationship to an entity type in its own right. The improved model not only resolves the symmetry but is often more expressive.

8.1.4 Examples

Consider Figure 8.1a and Figure 8.2a. *RelatedContract* involves two contracts but the symmetry is troublesome. If each pairing is entered once, it is not clear which contract should be first and which is second. If each pairing is entered twice, the amount of storage increases and any change requires double update. If more than two contracts are related, the situation is messier yet (for three contracts that are double stored: C1-C2, C2-C1, C1-C3, C3-C1, C2-

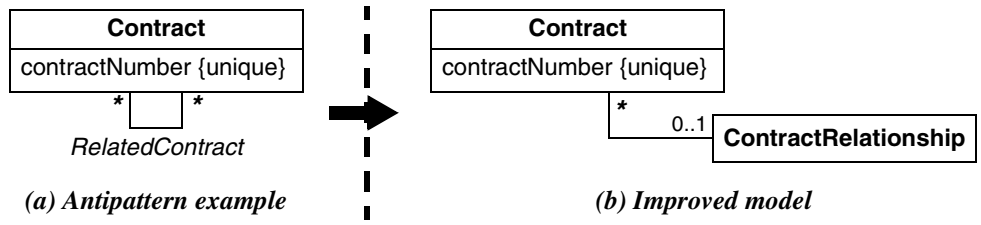


Figure 8.1 Symmetric relationship: UML contract model. Promote symmetric relationships to an entity type.

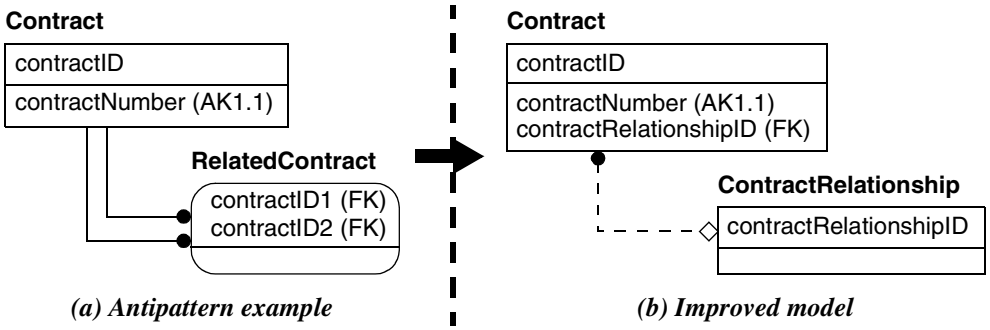


Figure 8.2 Symmetric relationship: IDEF1X contract model.

C3, C3-C2). Furthermore, the antipattern does not require that the related contracts be different. None of this is desirable.

The improved model (Figure 8.1b and Figure 8.2b) breaks the symmetry. To find related contracts traverse as follows: start with a *Contract*, find the possible *ContractRelationship*, then traverse back to *Contract* (excluding the initial contract) to obtain the related *Contracts*. Figure 8.3 shows the corresponding SQL Server code; the code is efficient if the join fields are indexed. (The SQL code presumes existence-based identity; see Chapter 16.)

The revised model has further advantages. The coupling is no longer binary and can readily support three or more related contracts. The model could be extended to make *Contract* to *ContractRelationship* many-to-many with different relationship types. For example, one relationship type could be successor contracts (one contract replacing another). A second relationship type could be alternative contracts (several contracts being considered as alternatives for purchase).

For another example, consider the words in a dictionary (Figure 8.4). An inferior model relates word meanings directly. Also the inferior model cannot handle a group of interchangeable words. Looking in the Framemaker 8 online thesaurus, the first definition of "account" has four synonyms (chronicle, history, annals, and report). The *SynonymSet* supports a group of word meanings.

```
SELECT C2.contractNumber
FROM Contract AS C1
   INNER JOIN ContractRelationship AS CR
       ON C1.contractRelationshipID =
          CR.contractRelationshipID
   INNER JOIN Contract AS C2
       ON CR.contractRelationshipID =
          C2.contractRelationshipID
WHERE C1.contractNumber = :aContractNumber AND
      C2.contractID <> C1.contractID
ORDER BY C2.contractNumber;
```

Figure 8.3 Symmetric relationship: Sample SQL traversal code. The code is efficient if the join fields are indexed.

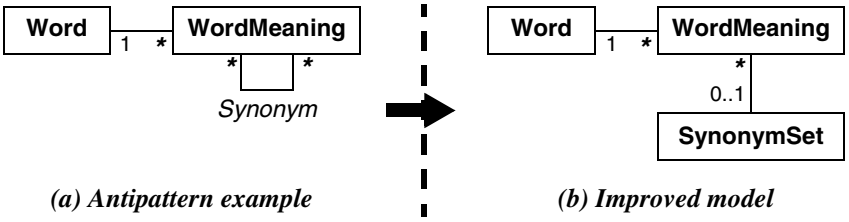


Figure 8.4 Symmetric relationship: UML synonym model. The improved model is more expressive.

8.2 Dead Elements Antipattern

8.2.1 Observation

A model has obsolete elements (entity types, relationships, attributes). They may have been relevant in the past but are extraneous now.

8.2.2 Exceptions

It is acceptable for a model to have small amounts (no more than a few percent of the total) of dead elements. Large amounts of junk cause confusion and complicate maintenance.

8.2.3 Resolution

Either cut the dead elements from the model or place them in isolation. For example, some commercial products have a special documentation section for deprecated database tables that will be removed in future releases.

8.2.4 Examples

Some databases have relic tables from past releases. It is acceptable to keep deprecated tables for a while, but eventually they should be removed. You should be suspicious of tables with zero records.

8.3 Disguised Fields Antipattern

8.3.1 Observation

The name and documentation for a field do not indicate the kind of data that is stored.

8.3.2 Exceptions

A few user-defined fields as well as miscellaneous comments are acceptable as an extensibility mechanism. Otherwise disguised fields are seldom justified.

8.3.3 Resolution

A relational database is supposed to be declarative. A field name should be informative and describe the data that is stored.

8.3.4 Examples

Disguised fields can arise in several ways.

- **User defined fields**. Many vendor packages have ***user-defined fields***—anonymous fields for miscellaneous data. Vendors cannot anticipate all customer needs and user-defined fields provide flexibility.
- **Mislabeled fields**. Software is constructed with an original purpose that meets business needs. With subsequent releases, developers may store different data without updating the schema. With user-defined fields, data lacks a description of its meaning. Mislabeled fields are worse, as the description is misleading.
- **Binary fields**. Some databases have binary fields whose interpretation is left to programming code. For example, the MS-Access system catalog has binary fields, such as the *Lv*, *LvExtra*, and *LvProp* fields in *MSysObjects*. These are rarely a good idea.
- **Anonymous fields**. Figure 8.5 shows an excerpt from a legacy application with anonymous address fields. Figure 8.6 shows some corresponding data. To find a city, you must search multiple fields. Worse yet, it could be difficult to distinguish the city of *Chicago* from *Chicago* street. You may need to parse a field to separate city, state, and postal code. It would be much better to put address information in distinct fields that are clearly named.
- **Overloaded fields**. A column of a table can store alternative kinds of values. Sometimes the kind of value is indicated by a switch in another column. Other times the values are distinguished by their format or contextual knowledge buried in programming code.

```
CREATE TABLE Location
  (location_num                  DECIMAL(3)
  ,location_name                 VARCHAR(15)
  ,location_address_1            VARCHAR(30)
  ,location_address_2            VARCHAR(30)
  ,location_address_3            VARCHAR(30)
  ,location_address_4            VARCHAR(30)
  ,location_address_5            VARCHAR(30)
  ,location_group_code           DECIMAL(2)
  ,location_business_type        VARCHAR(1)
  ,location_tot_bus_sales_dol    DECIMAL(11,2)
  ,location_gross_profit_dol     DECIMAL(11,2)
  ,CONSTRAINT PK_Location PRIMARY KEY (location_num ) ) ;
```

Figure 8.5 Disguised fields: Sample SQL code. Creating a table with anonymous address fields.

location_address_1	location_address_2	location_address_3
456 Chicago Street	Decatur, IL xxxxx	
198 Broadway Dr.	Suite 201	Chicago, IL xxxxx
123 Main Street	Cairo, IL xxxxx	
Chicago, IL xxxxx		

Figure 8.6 Disguised fields: Sample data. Anonymous address fields.

8.4 Artificial Hardcoded Levels Antipattern

8.4.1 Observation

Chapter 2 presented hardcoded trees with a different entity type for each level. Such an approach can be justified for models where the structure does not vary and it is important to enforce the sequence of types.

The antipattern also involves a fixed hierarchy but one with little difference between the entity types. Such a model is brittle, permits duplicate and contradictory data, and is difficult to maintain and extend.

8.4.2 Exceptions

Sometimes the hardcoding of artificial levels is desirable for its simplicity. For example, I needed to convert bill-of-material formats for a past project. The source was a hierarchical

indented list and the target was parent–child pairings. One program generated the hierarchy as output and another required the pairings as input. I did not want to program the recursive descent of a tree. Instead I hardcoded a fixed number of levels and quickly wrote a SQL query. Hardcoded levels can be acceptable for a prototype or throwaway code.

8.4.3 Resolution

Abstract and consolidate the levels. Use one of the tree patterns to relate the levels.

8.4.4 Example

Figure 8.7a shows a three-level hierarchy from a legacy application where an individual contributor has a supervisor who in turn has a manager. The limitation to three levels is arbitrary. Many questions come to mind. How should the software deal with an individual contributor who becomes a supervisor and then becomes a manager? Should there be three different records? Does the user multiply enter data, such as names, phone numbers, and addresses (omitted in the example)?

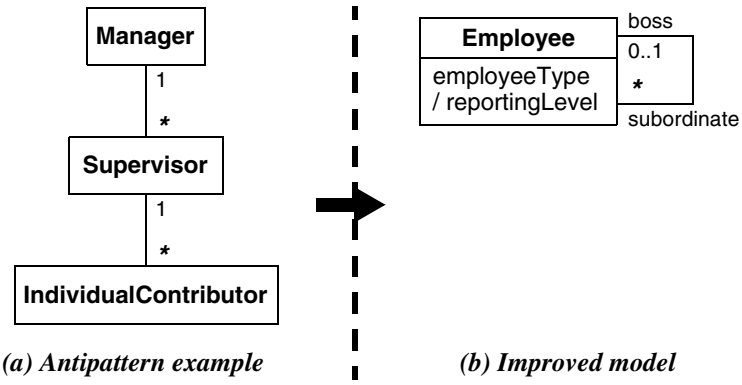


Figure 8.7 Artificial hardcoded levels: UML management hierarchy model. Abstract and consolidate the levels.

The improved model (Figure 8.7b) is simpler, more expressive, and avoids these issues. There can be an arbitrary number of management levels. An employee reports to a boss who is also an employee. The boss reports to his or her boss continuing up the reporting hierarchy. The field *employeeType* is an enumeration with the values of "Manager," "Supervisor," and "IndividualContributor." A boss can manage many subordinates and a subordinate has at most one boss. The highest ranking employee in the database has no boss. The '/' prefix is UML notation for derived data (see the Appendix for further explanation).

8.5 Excessive Generalization Antipattern

8.5.1 Observation

A model has a deep generalization. In many cases extensive taxonomies are motivated by object-oriented programming and are inadvisable.

8.5.2 Exceptions

If there is a formal standard for a taxonomy (such as for biological organisms) you should use it. Otherwise I cannot think of a justification. Normally, it is best to avoid deep generalization.

As an example, [Americazoo] presents a taxonomy for mammals, including Figure 8.8 for wolves. Each indentation is a lower level in the taxonomy. Figure 8.8 corresponds to a generalization that is seven levels deep.

```
Class: Mammalia
   Subclass: Theria
       Infraclass: Eutheria
          Order: Carnivora
             Family: Canidae
                Genus: Canis
                   Species: lupus
```

Figure 8.8 Excessive generalization: Taxonomy for wolves. Normally it is best to avoid deep generalization; formal standards are an exception.

8.5.3 Resolution

As a guideline a database taxonomy should be no more than four levels deep. (A programming taxonomy should be no more than five levels deep.)

8.5.4 Examples

Many years ago at GE Global Research, we built a software modeling tool, called OMTool. The tool was based on a metamodel with a taxonomy that was seven levels deep. Even though the taxonomy was sound, we found it difficult to remember all the levels complicating development. In retrospect we regretted using such a deep taxonomy.

On another project we prepared a large equipment taxonomy (50 pages long!). [Blaha-2003] We understood the problem well and had access to domain experts. Nevertheless, we had trouble with modeling. The taxonomy was so extensive that it was difficult to determine where to place equipment. Also the various types of equipment had many fields and we kept discovering additional data. For this project, it would have been better to forego a hardcoded taxonomy and instead use a softcoding approach. [Blaha-2006]

Section 10.2 shows a sound taxonomy that is three levels deep.

8.6 Disconnected Entity Types Antipattern

8.6.1 Observation

A model has a number of free-standing entity types. From the problem understanding, it would seem that they should be related.

8.6.2 Exceptions

Some disconnected entity types can be acceptable (as a guideline, no more than 10% of the entity types). But it is suspicious when a model has many of them.

8.6.3 Resolution

Recognize that the model is likely to be incomplete. Determine the missing relationships and add them.

8.6.4 Example

Many Eclipse (www.eclipse.org) applications generate XML files for storing persistent data. An application may also have a database that developers populate apart from Eclipse. If you reverse engineer the database, the resulting model can appear to be incomplete. Ideally the database should store both the added data and the Eclipse data.

8.7 Modeling Errors Antipattern

8.7.1 Observations

A model has one or more serious conceptual flaws. Modeling errors lead to bugs in the finished software, complicate development and maintenance, and can impair performance.

8.7.2 Exceptions

Errors in models become errors in the finished software. Errors are never acceptable, but sometimes you have to live with them, such as with legacy systems and vendor software.

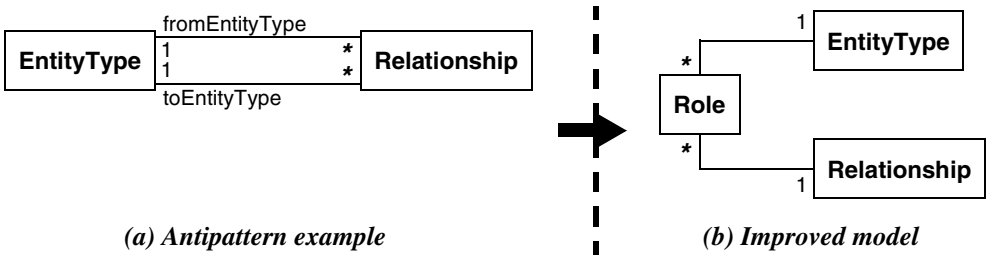
8.7.3 Resolution

Understand the flaw and how it may have come about. If possible, correct the model.

8.7.4 Examples

Over the years I have reverse engineered several modeling tools and inspected their internal metamodels. One would expect tool developers to have excellent models, but that is not always the case. Some data modeling tools have the deep flaw of directed relationships.

As Figure 8.9 shows, a directed relationship has “from” and “to” entity types. Using such a tool to construct Figure 8.1b, *Contract* would be “from” and *ContractRelationship*



(a) Antipattern example *(b) Improved model*

Figure 8.9 Modeling error: UML relationship model. Repair modeling errors.

would be "to" or vice versa. This is nonsense. The model simply states that *Contract* and *ContractRelationship* are related. An improved model (Figure 8.9b) introduces the notion of a role which is the intersection of an entity type and a relationship. As a side benefit, the improved model can support ternary relationships.

For another example, I reverse engineered a vendor product for a library catalog system and found that the database implemented a linked list. There is no such thing as a "linked list" in a conceptual model of a library. Someone did not abstract properly and misguidedly put implementation concepts in a conceptual model.

8.8 Multiple Inheritance Antipattern

8.8.1 Observation

A model has multiple inheritance. ***Multiple inheritance*** is a generalization for which an entity type inherits information from multiple supertypes.

8.8.2 Exceptions

Multiple inheritance is not appropriate for data models. It can be acceptable as a mechanism for programming reuse and for other kinds of models.

8.8.3 Resolutions

Avoid multiple inheritance in data models. Degrade the model if necessary. There is no clean way to implement multiple inheritance with a database. In practice I have found that multiple inheritance seldom occurs with databases and is not worth the bother.

8.8.4 Example

In Figure 8.10a and Figure 8.11a *FullTimeUnionEmployee* is both *FullTimeEmployee* and *UnionEmployee*. The model does not show it, but an extended model could define three additional combinations: *FullTimeNonUnionEmployee*, *PartTimeUnionEmployee*, and *PartTimeNonUnionEmployee*. Figure 8.10b and Figure 8.11b use a workaround (others are possible) to eliminate the multiple inheritance.

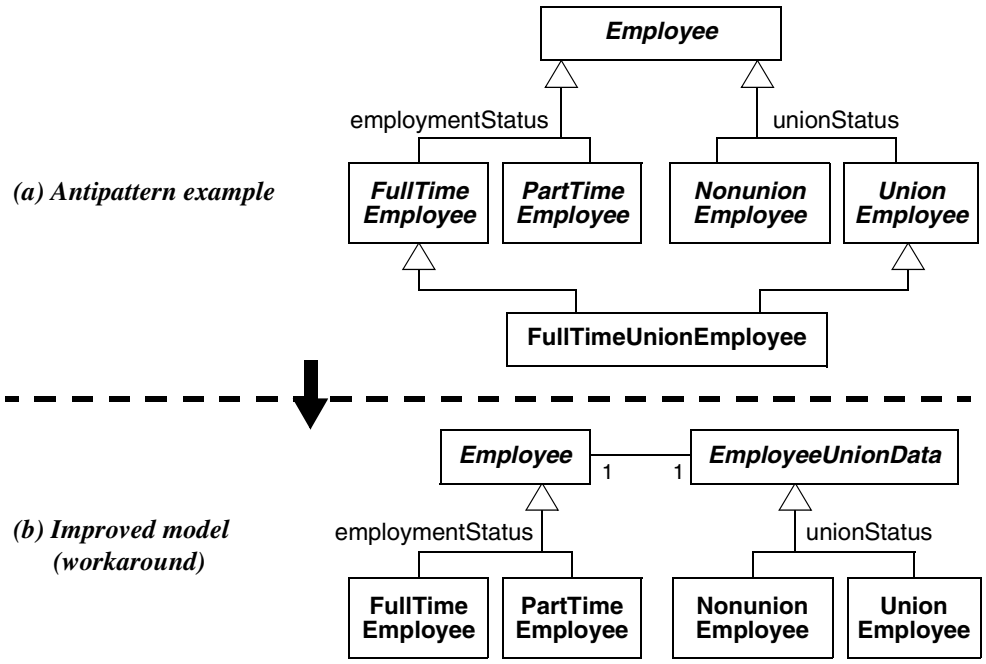


Figure 8.10 Multiple inheritance: UML employee model. Avoid multiple inheritance in conceptual data models.

8.9 Paradigm Degradation Antipattern

8.9.1 Observation

A relational database is degraded to fit some other paradigm.

8.9.2 Exceptions

Such a technique is highly questionable.

8.9.3 Resolution

Rework the model and architecture to avoid such degradation.

8.9.4 Examples

Many years ago, I was reverse engineering the database of a commercial product and was perplexed. The resulting model had many disconnected entity types with only a smattering of relationships. I could not understand how so much information could be missing and suspected that many relationships were disguised. I decided to cross check the schema with the

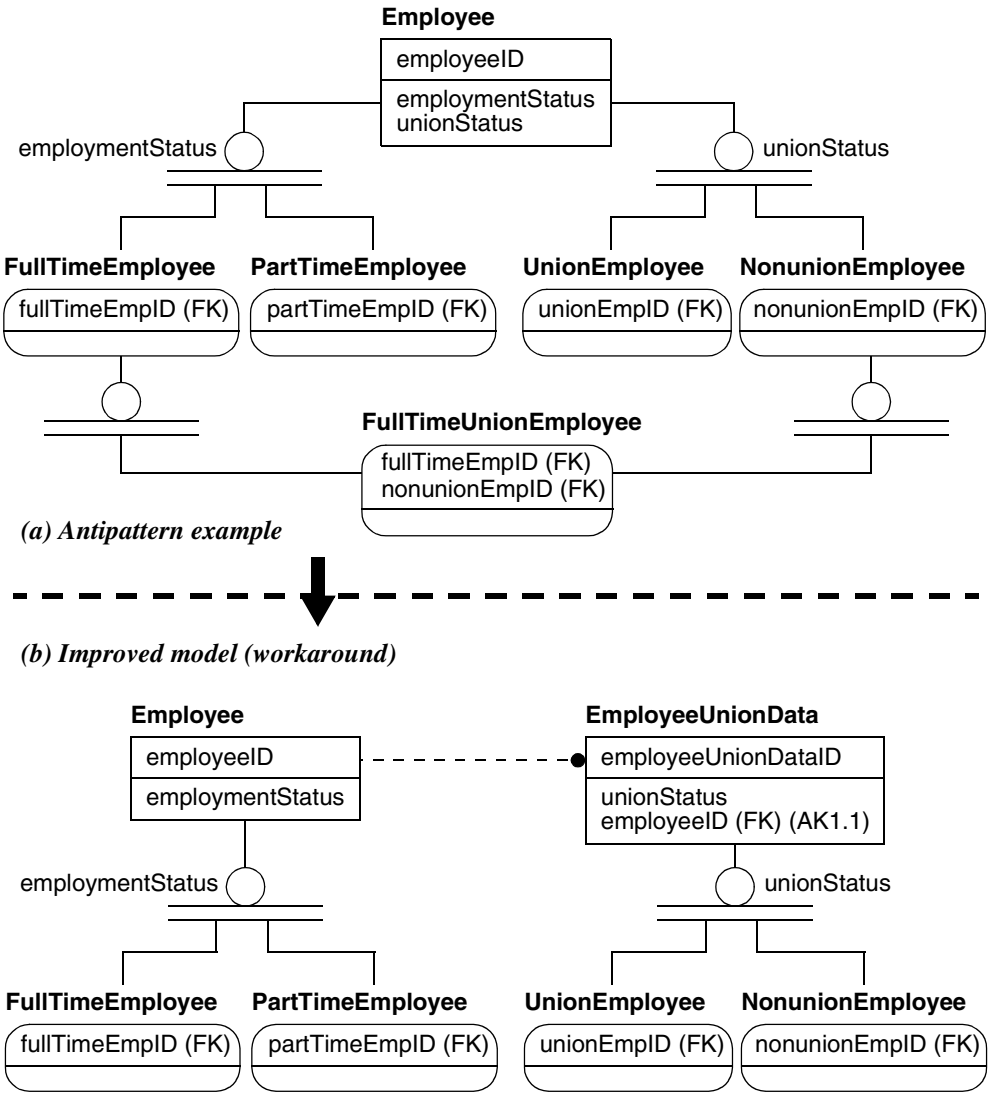


Figure 8.11 Multiple inheritance: IDEF1X employee model.

hierarchical screen layout and discovered the missing relationships. The software vendor confirmed my understanding.

The product supported a fixed hierarchy of depth three. Apparently the vendor created the original product with a proprietary hierarchical database. Then the vendor migrated to client-server technology and devised an isomorphic hierarchical database using a relational database for the server. Figure 8.12 shows the structure for level 1, 2, and 3 tables. The pointer fields are hidden parent pointers.

Antipattern example

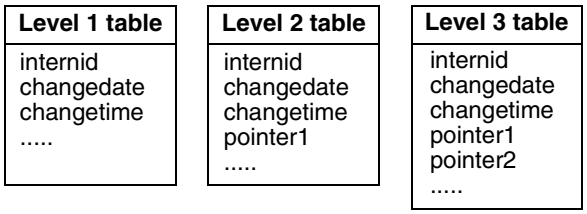


Figure 8.12 Paradigm degradation: Fixed three-level hierarchy.

Figure 8.13 shows a table from another application. I am not sure what the attributes mean. My best guess is that this table was being used for populating a spreadsheet.

Antipattern example

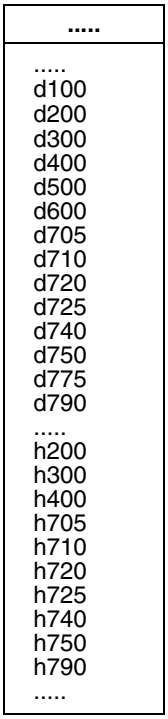


Figure 8.13 Paradigm degradation: Spreadsheet as a giant table.
Avoid distortions of a relational database.

8.10 Chapter Summary

An ***antipattern*** is a characterization of a common software flaw. As you construct models, you should be alert for antipatterns and correct them. Table 8.1 summarizes universal antipatterns—antipatterns to always avoid—along with their exceptions and resolution.

Table 8.1 Summary of Universal Antipatterns

Antipattern name	Observation	Exceptions	Resolution	Frequency
Symmetric relationship	A self relationship has the same multiplicity and roles on each end.	None	Promote the relationship to an entity type.	Common
Dead elements	A model has unused elements.	Acceptable in small amounts.	Delete them or isolate them.	Common
Disguised fields	Field names do not describe data.	A few user-defined fields.	Use meaningful names.	Common
Artificial hardcoded levels	There is a fixed hierarchy of similar entity types.	Use only with great caution.	Consolidate the levels and use a tree pattern.	Occasional
Excessive generalization	There is a deep generalization.	None	A db taxonomy should be at most four levels deep.	Occasional
Disconnected entity types	A model has free-standing entity types.	A few can be acceptable.	Add the missing relationships.	Occasional
Modeling errors	There is a serious conceptual flaw.	None	Fix the model.	Occasional
Multiple inheritance	A model has multiple inheritance.	Avoid for data models.	Use a workaround.	Seldom
Paradigm degradation	A relational db is degraded to some other paradigm.	Highly questionable.	Rework the model and architecture.	Seldom

Bibliographic Notes

[Brown-1998] discusses antipatterns for programming, architecture, and management. The book is informative, but the authors oversell the technology—antipatterns are not a panacea

for the difficulties of software development. As [Brooks-1987] notes, there is no silver bullet for improving software quality. [Laplante-2006] builds on [Brown-1998] and adds further management antipatterns as well as cultural antipatterns.

Many of the examples in this chapter came from my experiences with database reverse engineering — starting with existing database structures and inferring the underlying models. [Premerlani-1994] and [Blaha-1995] present unusual database designs that were found during reverse engineering.

References

[Americazoo] http://www.americazoo.com/goto/index/mammals/classification.htm

[Blaha-1995] Michael Blaha and William Premerlani. Observed idiosyncrasies of relational database designs. *Second Working Conference on Reverse Engineering*, July 1995, Toronto, Ontario, 116–125.

[Blaha-2003] Michael Blaha. Data store models are different than data interchange models. *Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003)*, November 2003, Victoria, BC.

[Blaha-2006] Michael Blaha. *Designing and Implementing Softcoded Values*. IEEE Computer Society ReadyNote, 2006.

[Brown-1998] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: John Wiley & Sons, Ltd, 1998.

[Brooks-1987] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer 20*, 4 (April 1987), 10–19.

[Laplante-2006] Phillip A. Laplante and Colin J. Neill. *Antipatterns: Identification, Refactoring, and Management*. Boca Raton, FL: Auerbach Publications, 2006.

[Premerlani-1994] William Premerlani and Michael Blaha. An approach for reverse engineering of relational databases. *Communications ACM 37*, 5 (May 1994), 42–49.