STUDY GUIDE

Database Application Development

FALL 2019

MICHAEL McQUAID

THIS REVISION PRODUCED OCTOBER 10, 2019

This document is available at
http://mickmcquaid.com/iste432.html.

The source of this document is available at
http://github.com/mickmcq/iste432book.

This document was produced by XeLaTeX,
using the *fbb* font package.

Note that, if you must print this document, you should use Adobe Acrobat's option to print in *booklet* format, two pages to a sheet. Other pdf software may use the term *booklet* or something else. You should really look for it for best results. If you merely print two to a sheet without a *booklet* option, you may get small page images with large borders. If you print one page to an 8.5 × 11 sheet or A4 sheet, you will get a greatly magnified version (extremely large type). That may not be what you want unless you have very poor eyesight and a lot of extra paper.

RDBMS REVIEW

The following review material comes from a generic database course, the intro course in which students learned to produce these three artifacts: entity relationship diagrams, functional dependencies, and relational notation.

This course assumes you can do the things learned in an introductory course on databases as well as a course on connecting to databases and working with them in an application programming language. The following subsections review some introductory course material. If you can not easily and quickly follow this material on your own, you may not be ready to begin this course and may require further work on your own to get up to speed in this course.

Diagrams. Creating and viewing diagrams are central activities for any information technology professional. To develop a database system, or any system large enough to justify effort, we must communicate with team members and possibly with suppliers and customers about the details of the system. Diagrams solve a communication problem that arises from the wide chasm between natural languages and pro-

gramming languages. Diagrams represent a middle ground between sentences spoken in a natural language and statements written in a programming language. Conversation is too general and ambiguous to specify details of an information system. In any given conversation about a particular information system, the professionals involved likely speak different programming languages or dialects. Even your own programs are difficult to read a long time after you've written them.

There are vastly many diagramming tools for describing information systems in general. For example, UML (Unified Modeling Language) specifies at least the following kinds of diagrams: class diagram, object diagram, package diagram, component diagram, use case diagram, sequence diagram, statechart diagram, activity diagram, and profile diagram. This list comes from a diagramming software package called StarUML, and is not even a complete list of the diagrams supported by that package.

The diagrams we will work with in this course are entity relationship diagrams, the most venerable and widely used diagrams employed by database professionals. The entity relationship diagram may also be the most widely used kind of diagram employed by any information technology professionals.

Software packages for drawing diagrams have short lifespans. Diagram syntax, on the other hand, endures. Peter Chen popularized the first syntax for entity relationship diagrams in 1976 and there are today really only three basic variants on that syntax. It has been hugely popular since its introduction, almost three times as long as the dreadful Visio drawing software program that has been popularly used to produce it during the decade since Microsoft purchased Visio. There is little sense in growing overly attached to particular

diagramming software packages. While you are likely to use ERDs throughout your careeer, the software packages you use will come and go. The companies that produce software packages live and die and sell out and close off their formats to prevent sharing of diagrams among rival software packages. It is tempting to list experience with different software packages on your resume, but it is more important for your career to be able to read and write the symbols on an entity relationship diagram using pencil and paper than to manipulate the menus of some software package that will soon be updated to be incompatible with itself.

Normalization. You should have taken at least two database courses before this course. In your first database course, you should have learned to normalize data and some concepts related to normalization. These are reviewed as follows, with an example from an introductory database course.

Normalization Definitions. Using library.rit.edu,

- visit `find a database`,
- then `electronic books`,
- then `books24x7`,
- then search for `teorey, toby`

where the following should appear among the listed books. Toby Teorey et al., *Database modeling and design: logical design*, Fifth Edition, Morgan Kaufmann, 2011. Following are definitions for normalization given in Chapter 6 of this book.

- A *domain* is the set of all possible values for a particular type of attribute, but may be used for more than one attribute, e.g., a person's name is a domain that may be used for employee name or customer name.

- A *superkey* is a set of one or more attributes that, when taken collectively, allows us to identify uniquely an entity or relation.
- A *candidate key* is any superkey that can not be reduced to another superkey
- A *primary key* is chosen arbitrarily from available candidate keys as the identifier for a tuple in the relation, also known as a row in the table.

Candidate and Primary Key Identification.

- Any unique tuple (also known as a row) is a superkey.
- Suppose you have a list of people assigned to office numbers, 101–109.
- Some of the people have identical names but only person is assigned to each office. Furthermore, each office has exactly one phone with one phone number.
- The Office relation has any combination of these attributes that includes phone number or office number or both as a superkey.
- There are only two candidate keys, though: office number alone and phone number alone. Any other superkey may be reduced to one of these and still uniquely identify a row, satisfying the superkey property.
- Either candidate key may be chosen as the primary key.

Functional Dependencies.

- In any relation R, a set of attributes B is functionally dependent on another set of attributes A if, at each instant of time, each A value is associated with only one B value. Such a functional dependence is written $\{ A \} \rightarrow \{ B \}$ and may be read A determines B. The left hand side is called the determinant.

- Example: Given the above Office relation, A is Room 101 and B is Winston. There is only one occupant per room so knowing the room number determines the occupant.
- A functional dependency is considered trivial if the LHS is a superset of the RHS. If so, it has the form $\{ A, B \} \rightarrow \{ B \}$. In other words, if A does not determine B we could still say that A and B together determine B but it would be trivial to say so.

Relational Notation. The convention we'll observe for writing out relations is as follows.

relationName(primaryKey, *foreignKey* , field1, field2,
...)

Some books follow the convention that the name of the relation is in all caps. We will not observe that convention here. We will follow the convention of underlining the primary key and italicizing foreign keys. If you are writing in plain text without the ability to underline or italicize, please obey the following convention.

```
relationName( _primaryKey_, *foreignKey*, field1,
field2, ... )
```

1NF: First normal form. A relation is in first normal form, 1NF, if and only if all columns contain only atomic values—that is, each column can have only one value for each row in the table.

Whether an item is atomic is a matter determined by people.

SQL has no concept of non-atomic items. Oracle has an extension to SQL that explicitly defines some items as non-atomic. Hence, any relation specified in SQL is in 1NF unless that relation was specified using an extension to SQL.

For an introductory exercise, it would be sufficient to say

`Item(ItemID, ItemName, Description, Returnable, Perishable, ShelfQty, Notes, Colors, RetailPrice, Cost, Supplier, Name, Street, City, State, Zipcode)` to describe the relation in 1NF corresponding to the ER diagram in Figure 1. This conflicts with my personal folklorish definition of 1NF as taught in many DB textbooks.

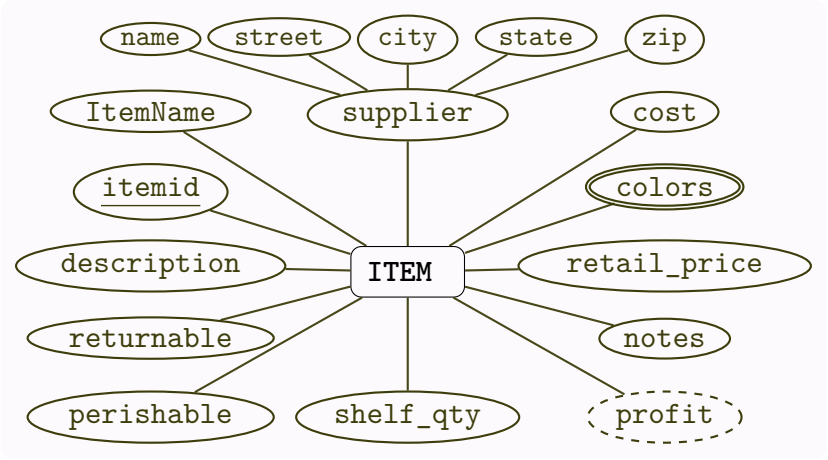


Figure 1. Sample ER diagram

Many database textbooks teach that 1NF eliminates repeating groups, based on the notion that a repeating group is non-atomic. The Teorey textbook mentioned above hearkens back to the original definition of non-atomic and moves the *eliminate repeating groups* into a more appropriate place in 2NF. Going forward in this class, I would like you to ignore other textbooks on this point.

It would also be correct to specify the same table without any reference to the primary key as being in 1NF. In other words,
`Item(ItemID, ItemName, Description, Returnable, Perishable, ShelfQty, Notes, Colors, RetailPrice,`

`Cost, Supplier, Name, Street, City, State, Zipcode)` is a valid 1NF relation, assuming that each row is unique. This is because, whether or not we have defined a primary key, a primary key must exist by the definition of candidate key given above.

2NF: Second normal form. A relation is in second normal form, 2NF, if and only if it is 1NF and every nonkey attribute is fully dependent on the primary key. An attribute is fully dependent on the primary key if it is is on the right side of a functional dependency for which the left side is either the primary key or something that can be derived from the primary key using the transitivity of functional dependencies.

In many if not most applications, functional dependencies are implied in the ER diagram but not certain. For example, Apple obtains displays for its iPhones from several different suppliers. To uniquely identify a display item might require a concatenation of the item identifier and the supplier identifier. Alternatively, each item might have a separate identifier that incorporates the information about which supplier provided the item. Suppose we know the functional dependencies

`{ Supplier } → { Name, Street, City, State, Zip }`

and suppose further that we know that

`{ Zipcode } → { City, State }`

Then the following ITEM relation is in 2NF

`Item(ItemID, ItemName, Description, Returnable, Perishable, ShelfQty, Notes, Colors, RetailPrice, Cost, Supplier)`

Notice that the information that is dependent on Supplier is no longer present. To create the required tables, I would also specify

`Supplier(Supplier, Name, Street, Zipcode)`

as a separate relation. But what happened to City and State? The above table would not be in 2NF if I included them be-

cause City and State ordinarily depend only on Zip. Since I noted that explicitly above, I'm going to skip the 1NF version of a Supplier table go right to a combination of Supplier and Zip as follows

```
Zipcode(Zipcode, City, State)
```

You may object to using the same word to identify an entity and an attribute. Although we usually add ID to attributes named for entities, I don't think SQL requires it. Also, SQL offers us a way to qualify ambiguous identifiers.

What about Colors? Is the following functional dependency legitimate?

```
{ Item } → { Colors }
```

No. Knowing the Item does not determine the Colors, given that Colors has been described as multi-valued in the ER diagram. Therefore, it should be eliminated from the ITEM relation and a new relation should be specified as

```
Itemcolors(Item,Colors)
```

where each row uniquely identifies a combination of Item and Colors. There is only one superkey, one candidate key, and one primary key possible.

Third normal form. A relation is in third normal form, 3NF, if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key.

Boyce-Codd normal form. A relation is in Boyce-Codd normal form, BCNF, if and only if every determinant (LHS of a functional dependency) is a candidate key.

Fourth normal form. Date (2004), page 385, gives the following definition of fourth normal form: Relvar R is in 4NF if and only if, whenever there exist subsets A and B of the attributes of R such that the nontrivial MVD $A \twoheadrightarrow B$ is satisified, then all attributes of R are also functionally dependent on A.

To understand this definition, it is necessary to review Date's definition of multi-valued dependence, MVD, which

is as follows. Let *R* be a relvar and let *A*, *B*, and *C* be subsets of the attributes of *R*. Then we say that *B* is multidependent on *A*—in symbols $A \twoheadrightarrow B$ (read *A* multi-determines *B* or simply *A* double arrow *B*)—if and only if, in every legal value of *R*, the set of *B* values matching a given *AC* value pair depends only on the *A* value and is independent of the *C* value.

Normalization review. The following paragraphs review normalization as discussed above.

Define the normalization process and terms.

- Anomalies
  - Insertion
  - Modification
  - Deletion
- Functional Dependency
- Determinant
- Candidate versus Primary Key
- Composite Keys

Normalization Process.

- First Normal Form (1NF)
  - Review characteristics of a relation
- Second Normal Form (2NF)
  - Partial Dependency
- Third Normal Form (3NF)
  - Transitive Dependency
- Boyce-Codd Normal Form (BCNF)
- Steps to Fix a Normal Form Violation
  - Referential Integrity
- Relationship Between Normalization and Modeling

The parts database. I will refer to the *parts database* to mean several different databases that appear in different editions of books by C.J. Date. Various authors have reused and modified the database and called versions of it by different similar-sounding names. The only distinction in these notes is between the *initial* database and the *full* database. I will introduce separate files of SQL statements to create the two versions. Either version must be dropped before loading the other. The *full* database contains one additional relation and that relation affects the other relations in the database.

Two versions of the database. We'll use the *initial* version for a while, then switch to the full version. Be prepared to slightly redefine some properties of the *initial* version when we switch to the *full* version.

Entities in the initial database. We are a productive enterprise. We operate in several cities. In each city we have exactly one warehouse with one receiving dock. A list on a clipboard at each receiving dock reveals that we know the following information that needs to be stored in a database of information about the quantity of parts we have received. The list is in a simple row and column format where the column headings follow. Every row is completely filled in. Every row represents a quantity of parts of a specific weight and color received at our warehouse in the named city from the supplier in the named city. In other words, the word city occurs twice in the row, once to refer to the city where our warehouse is located and once to refer to the location where the supplier of the quantity listed in that row is located. If we collect all the clipboards and make an enterprise-wide clipboard, it will resemble the following table.

sname	rating	city	color	pname	weight	city	qty
Jones	10	Seoul	red	nut	17	Seoul	100
Smith	20	Paris	blue	nut	17	Paris	200
Adams	30	Tokyo	blue	bolt	19	Tokyo	300
...	...	...	...	...	...	...	...

- `sname` is the single-word name of a supplier (not necessarily unique)
- `rating` is a two-digit number, our preferential rating of the supplier
- `city` is the single-word city where a supplier is based (a supplier has only one base) and can deliver parts to us
- `pname` is the single-word name of a part supplied to us by a supplier
- `color` is the single-word color of a supplied part, where parts with the same name may be supplied in different colors
- `weight` is the weight in grams of a part, where parts with the same name may be supplied in different weights
- `city` appears in a second column on the clipboard with respect to parts and is the city where we took delivery of a part (so far always the same city as in the first column) and stored it in our one warehouse in that city
- `qty` is the quantity of the particular part received

Note that there is no additional information. There is not a timestamp or any way to link these rows to an inventory system nor a purchasing system nor an order entry system. We have only this very simple database, we are not in a position to think about such things. These are simply quantities received and we can inquire of these lists how many parts we have received.

Symbols to represent these concepts. The above list gives the column heading names on the clipboard at the receiving dock. Use these to determine the entities we need to store in the initial database and to create a draft entity relationship diagram. Begin by listing the symbols that might appear on an entity relationship diagram. Then revisit the list above to make choices about entities and attributes.

Recalling the symbols we need should flow from thinking about what we want to represent. The main purpose of a database is to record information about objects and events in the real world. These objects and events are usually called entities in the database community. The objects usually exist in the real world independently of the database. The events usually occur in the real world and involve relationships between objects. The events of interest usually leave some trace independent of the database. The objects and events can usually be construed as bundles of other objects or events or characteristics, all of which are defined by people using the database. The usual way to record this information is to say that entities (objects and events) have attributes, characteristics that differentiate the entities in dimensions of interest to database users.

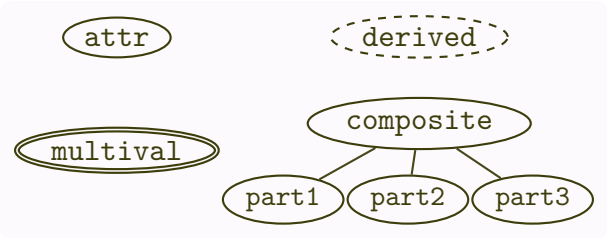
The above indicates that diagrams need symbols for entities, attributes, and relationships. We have also seen that we need additional symbols to distinguish between cardinalities of relationships. For entities, we have used a rectangle. For attributes, we have used symbols from the notation of Peter Chen (the inventor of entity relationship diagrams) three kinds of ovals, a solid oval, a dashed oval indicating a derived or calculated attribute, and a double-bordered oval indicating a multi-valued attribute. For relationships, we have used lines connecting two or more rectangles. Peter Chen's system uses a diamond on the line connecting rectangles but we have

been using an informal hybrid of some elements of Chen's approach and some elements of another approach called crowsfoot notation, named for the line endings. We've also used some elements of a notation called IDEF1.

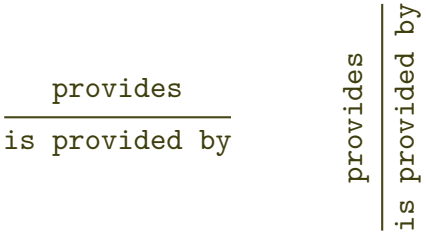
We have also used lines to connect attributes to entities but we have used that to think about entities and attributes. We will abandon that practice in favor of listing attributes in rectangles when develop a more refined view of entities and their relationships and can draw a more robust entity relationship diagram. The diagram will be robust in the sense that we will share it with others and it will reflect a working database and may be useful for a long time. For the moment, the list of symbols includes

- rectangle: entity
- oval
  - solid: attribute
  - dashed: derived attribute
  - double-bordered: multi-valued attribute
  - oval with branching ovals: composite attribute
- line: relationship between entities (the lines that connect composite attributes are problematic)
- line annotations
  - verb: relationship, where above is read left to right and below is read right to left if the line is horizontal and bottom to top on left and top to bottom on right if the line is vertical
  - numbers: minimum / maximum cardinality more specific than is portrayed by line endings
- line-ending
  - crowsfoot: many (we won't actually use this in completed diagrams)

- crowsfoot 0: zero or more
- crowsfoot 1: one or more
- 1 1: one and only one
- 1 0: zero or one



While we have used the ovals shown above, we will not construct diagrams for hw and exams using this notation. Instead, we will see these in examples and use a different notation when we develop them. On the other hand, we will use verbs on the relationships as in the following example. The word provides is read left to right on the horizontal line and from bottom to top on the vertical line. The other label is read the opposite way and may be optional depending on the relationship.



We'll use the following crowsfoot notation. From left to right the symbols read zero or one, one and one only, many (we won't draw this but we'll see it in unfinished diagrams), zero or more, one or more.

There may be ambiguity between *zero or one* and *zero or more* so use the most restrictive applicable notation. If you know that *zero or one* is correct, do not use *zero or more*, even though it is strictly true. If you are given enough information to know that the entity occurs zero or one time, it will be graded as an error to say that it occurs zero or more times.

Distinguishing entities and attributes. Try to distinguish between an entity and an attribute. An entity is an object, process, or event of interest to the enterprise. Think of the items in the list from the viewpoint of the enterprise.

A productive enterprise buys, sells, grows, mines, trades, transforms, makes, and consumes entities. A productive enterprise employs entities, partners with entities, buys from entities, sells to entities, competes with entities, and pays taxes to entities.

By productive enterprise I refer to what is often called the private sector. The public sector is analogous but includes other activities. An organization in the public sector may regulate entities, draft entities, educate entities, defend entities, provide public entities, and more.

Another kind of entity is an event. Events are often described by verbs or gerunds, as well as nouns. Can we say we are selling? Can we say we are buying? These are events that leave traces, such as receipts and invoices.

A third kind of entity is a process. Processes are more difficult to represent in databases than events or objects. Much of the history of information technology has seen tension between process and data as two very different domains of work, where processes are the province of programs and data is the stuff of databases. Approaches to information technology have often been characterized as either process-driven

or data-driven as if these two approaches lack compatibility with each other.

One explanation for the tension between process and data and the submergence of process below object and event in entity relationship diagrams may be found in the study of history. One of the most famous historians of the twentieth century was Fernand Braudel. Braudel altered the way historians studied history, in part, by dividing history in a way similar to the preceding classification of objects, events, and processes. First, Braudel described certain features as persistent, as present for a very long time, like objects. Second, he described certain features as processes, mostly economic, played out in medium time scales. Third, he considered events as having a brief and easily defined time of occurrence. He claimed that it was much easier to study the events and the persistent objects and that it was much harder to study processes occurring at an intermediate time scale. Braudel claimed that the study of history suffered from the difficulty of including medium time scale processes and influenced historians to reexamine their work to find opportunities to consider the relevance of these processes.

The preceding paragraph is a gross simplification and the author of these paragraphs is not a historian. Yet even a casual reading of Braudel presents a striking parallel for a database professional trying to piece together a picture of the relationships between entities, where the briefest events and most persistent objects are easier to identify than are medium-term processes.

Attributes. An attribute is a relevant characteristic of an entity. Since we're interested in the viewpoint of the enterprise we mean relevant from the viewpoint of the enterprise. For each item in the inventory list, Ask yourself if you can say that it is a property of another item in the list or an item

whose existence is implied by the list.

Consider color. Is color a property of some other item in the list? Is it a property of an item whose existence is implied by the list? Do we sell to color? Is color a property of a screw? Ask similar questions of weight, pname, city, and so on. While we may conceptualize many words as nouns and therefore entities, many are used as adjectives. These may well be properties or characteristics.

As we look through the above list, it should stand out that we record information about suppliers and parts. We buy from suppliers. Suppliers sell to us. We acquire parts. We use parts. These are the most obvious entities from which we may begin to construct an entity relationship diagram. From now on, when we construct an entity relationship diagram, we'll use a convention found in some entity relationship diagramming software, such as MySQL Workbench, where entities and attributes are depicted in rectangles with rounded corners. A horizontal line across the rectangle separates the entity names from the attribute names.

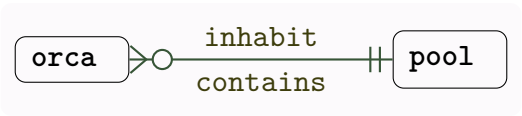
I claim that there is a many-to-many relationship between these entities. To see why, imagine a table consisting of these two entities and nothing else.

supplier	part
Smith	nut
Smith	bolt
Smith	cog
Jones	nut
Jones	bolt
Adams	bolt
Adams	cog

Now examine each column. May a supplier appear more than once? Yes. May a part appear more than once? Yes. It may help to revisit the orca pool question. Recall that we must place the following orcas into pools: Shamu, Whamu, Blamu, and Kramu. We have pools A, B, and C. Since there are more orcas than pools, at least two orcas will have to share a pool. One way we could distribute them would be the following arrangement.

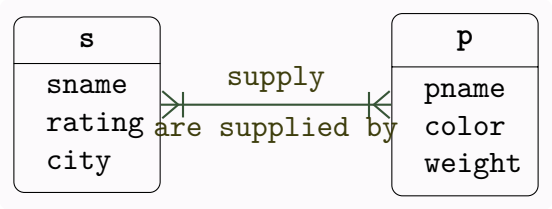
orca	pool
Shamu	A
Whamu	B
Blamu	C
Kramu	A

How often may an orca appear? Once and only once. Due to requirements of cetacean biology, it can not survive long out of a pool. We must put each orca into a pool and there is no way to distribute one orca among many pools and still have it available to do tricks for our patrons and crush any lunatics who insist on swimming with the whales after closing time. According to the following diagram, many orcas may inhabit a pool. A pool contains potentially many orcas.



The above relationship does not lead to database anomalies. In general, many to many relationships in entity relationship diagrams require work, while one to many relationships are easy to transform into databases without insertion, deletion, and update anomalies. So our many to many picture of suppliers and parts will need work to make a diagram that will be easily transformed into a database.

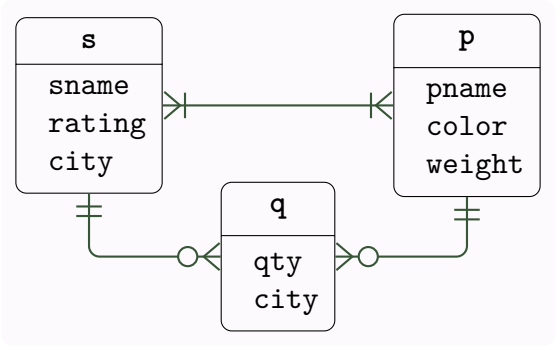
The first step I would like to take is to add the attributes from our clipboard that clearly belong to supplier and part. We can ask of each column on the clipboard whether it is something a supplier has or something a part has. Certainly a supplier has a name, given by `sname`. A supplier, according to the description, has a `rating`, a numeral expressing a preferential rating. A supplier has a `city`. A part clearly has a name, given by `pname`, as well as a `weight` and `color`.



The above uses the notation we'll continue to employ for all entity relationship diagrams we produce. While you will still see the Chen style diagrams with ovals, you will not produce them as solutions to problems. This style of diagram has no symbology for derived or multivalued attributes. Generally, there is no need for them in solutions to problems but, should the need arise, you can simply write the strings

derived or multivalued in parentheses after the attribute name.

Now think about the remaining columns, the second occurrence of city and qty. Where do these belong? Does a part have a city? Does a part have a quantity? The second occurrence of city is the location of the warehouse where we have received the quantity of parts listed as qty. Both of these are really attributes of an event, the event where we have received a quantity of parts, the event that is recorded by each row of each clipboard. This is certainly an event of interest to the enterprise because it is the one thing being recorded. The number of rows filled in on the clipboards is the number of times this event has occurred. The event itself, an entity called **q** in the following diagram, really only needs a link to each of the other entities and its own information, city and qty. The eight columns of the clipboard can be filled in as follows.

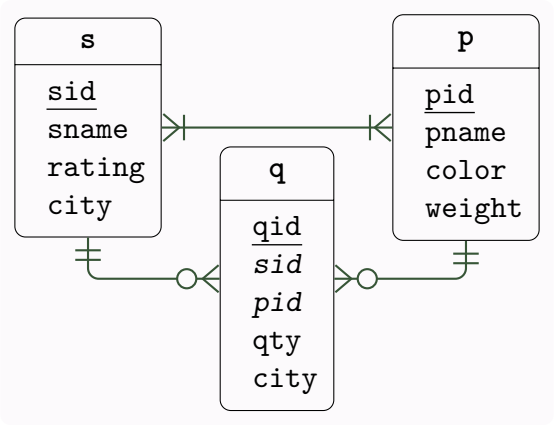


Two main issues remain. First, we must ensure that we have the right symbols on the relationship lines. Second, we need to say explicitly how we link these entities.

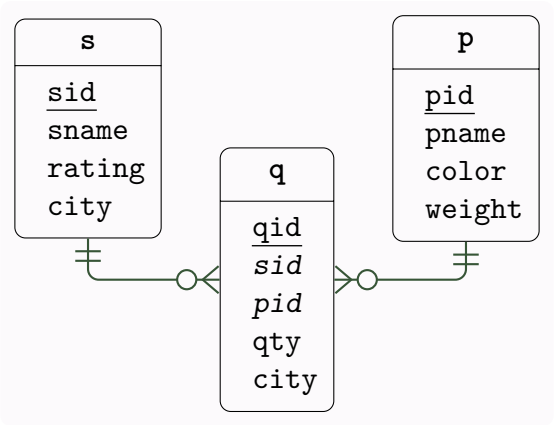
The first issue requires serious thought. There will always be exactly one supplier associated with every quantity received, just as there is exactly one supplier listed on every

row on the clipboard. The definition of quantity received that we have obtained from examining the clipboard is that it comes from one supplier. From this definition we know that the supplier information will not be empty, either, so we must regard the supplier information as referring to one and only one. Each supplier may appear many times on the clipboard, so the quantity received side of the relationship can occur many times. I have chosen to portray this side of the relationship as zero or more because we may have a new supplier from whom we have not yet received any quantities of any parts. Similarly, we may have old suppliers no longer operating but will likely maintain a record of them. It is rare to delete entities from business databases. The parts are in the same position as the suppliers. We may know in advance that we will need parts not yet received and we will likely maintain a record of parts no longer being used.

The second issue is the link between the event and the two objects. There is no guarantee that any of the attributes or combinations of attributes of supplier and quantity received are unique. The combination of name, color, and weight is unique for every part. Nevertheless, we will assign a unique id number for each of these three entities and add that to each of the attribute lists. This is usual practice in database management where it is rarely a good idea to use naturally occurring information as an identifier. Strings that appear to be unique at the start of a database project often lose their uniqueness over time as business practice changes. It is almost never a mistake to add an artificial identifier as we do here.



The preceding diagram shows both the problem and the solution. The next diagram shows only the solution. Which is better? Which do you prefer to find when you first discover a database and wish to understand it well enough to work with it?



Examples from an Introductory Course. You should be able to answer the following questions as a prerequisite (not the only prerequisite) for the present course. If you can not complete these on your own, you may not be ready for this course and may not be able to pass this course without further

work on your own prior to attempting this course. The following exercise problem statements are copyrighted by Elissa Weeden and used here with permission.

Going from functional dependencies to relational notation. Please normalize the following relation through BCNF and write the answer using relational notation.

`LOT( propertyID, countyName, lotID, area, price, taxRate )`

Assume the following functional dependencies.

1. `{ propertyID }` → `{ countyName, lotID, area, price, taxRate }`
2. `{ countyName, lotID }` → `{ propertyID, area, price, taxRate }`
3. `{ countyName }` → `{ taxRate }`
4. `{ area }` → `{ price }`

Q/A About Functional Dependencies. Given the relation

`MUSIC( Title, Artist, NumGrpMembers, Year, Producer, ProducerURL, Category, CategorySales, Media, MediaPrice )`

and the following business rules

1. Each album is uniquely identified by its title. For the remaining rules, the Title attribute of MUSIC refers to the album name.
2. An artist may be a single person or a group, the count of which is recorded in NumGrpMembers.

3. Each album is released in exactly one year.
4. Each album is produced by exactly one producer (a music company).
5. Each producer has exactly one URL.
6. Each album has exactly one artist.
7. Each album has exactly one music category.
8. Each category is associated with one sales value, CategorySales, which is the year-to-date sales in that category.
9. All sales occur at a price dependent only on media type, e.g., 8 track cartridges all cost 5.99 USD, all audio cassettes cost 6.99 USD, 78s cost 7.99, LPs cost 8.99, cylinders cost 9.99, and so on.

List all functional dependencies for the MUSIC relation, using only the given business rules. Specify whether any dependency causes any 2NF or 3NF violations.

Answer. Step 1 is to examine each business rule in turn and write the functional dependency it describes.

a. `{ Artist }` → `{ NumGrpMembers }` (rule 2)
b. `{ Title }` → `{ Year, Producer, Artist, Category }` (rules 3, 4, 6, 7)
c. `{ Producer }` → `{ ProducerURL }` (rule 5)
d. `{ Category }` → `{ CategorySales }` (rule 8)
e. `{ Media }` → `{ MediaPrice }` (rule 9)

Step 2 is to use the above functional dependencies to identify candidate keys for the `MUSIC` relation. Why not begin with a superkey composed of the LHS of each functional dependency? This provides a composite key of `Artist, Title, Producer, Category, Media`. Next, examine this superkey to see what can be removed from it. Since three of the

attributes depend on `Title`, they may be removed, leaving an irreducible superkey (candidate key) of `Title, Media`. Next, if this is to be used as the primary key, identify the functional dependencies above that would lead to a partial dependency—a nonkey attribute dependent on part of the primary key or a transitive dependency—a nonkey attribute dependent on another nonkey attribute that is in turn dependent on the primary key. The partial dependencies are 2NF violations and the transitive dependencies are 3NF violations.

Given a primary key of `Title, Media`:

a. `{ Artist } → { NumGrpMembers }`

   leads to a transitive dependency where `NumGrpMembers` is dependent on an attribute other than the primary key

b. `{ Title } → { Year, Producer, Artist, Category }`

   leads to partial dependencies where `Year`, `Producer`, `Artist`, and `Category` are dependent on `Title` which is part of the primary key

c. `{ Producer } → { ProducerURL }`

   leads to a transitive dependency where `ProducerURL` is dependent on an attribute other than the primary key

d. `{ Category } → { CategorySales }`

   leads to a transitive dependency where `CategorySales` is dependent on an attribute other than the primary key

e. `{ Media } → { MediaPrice }`

   leads to a partial dependency where `MediaPrice` is dependent on `Media` which is part of the primary key

Q/A about Normalization, 1. For the following relation and functional dependencies, determine the highest normal form that describes it. If that is not BCNF, then normalize it and any needed additional relations, through BCNF.

```
Q1(a,b,c,d)
{ a,b } → { c,d }
{ c } → { d }
```

Answer. Relation `Q1` has the primary key `a,b`. The first functional dependency tells us that this primary key determines the other two attributes, so the relation is at least in 2NF. The second functional dependency tells us that the attribute `d` depends on a nonkey attribute so the relation can not be in 3NF.

Normalizing the relation requires removing the transitively dependent attribute to another relation, leaving a foreign key in the existing relation. Therefore we need to specify a new relation and a referential integrity constraint.

```
Q1( a,b,c )
  Q1(c) must exist in Q1C(c)
Q1C( c,d )
```

Q/A about Normalization, 2. For the following relation and functional dependencies, determine the highest normal form that describes it. If that is not BCNF, then normalize it and any needed additional relations, through BCNF.

```
Q2(a,b,c,d)
{ a,b } → { c,d }
{ a } → { c }
{ b } → { d }
```

Answer. Relation `Q2` has the primary key `a,b`. The first functional dependency tells us that this primary key determines the other two attributes, so the relation could be in 2NF if the remaining functional dependencies don't introduce any

problems. Unfortunately, they do. The second functional dependency says that c depends on part of the primary key, so the relation Q2 is not in 2NF. The third functional dependency the information that a second attribute, d, depends on part of the primary key. Either of these dependencies alone is enough to limit Q2 to being in 1NF. To normalize this relation, these attributes must be removed to new tables and referential integrity constraints must be added because the components of the primary key of Q2 are now also foreign keys of the new relations.

```
Q2( a,b )
  Q2(a) must exist in Q2A(a)
  Q2(b) must exist in Q2B(b)
Q2A( a,c )
Q2B( b,d )
```

Q/A about Normalization, 3. For the ER diagram in Figure 2 and the following functional dependencies, determine the highest normal form that describes it. If that is not BCNF, then normalize it and any needed additional relations, through BCNF.

```
{ itemid } → { ItemName, name, street, city, state,
zip, cost, color1, color2, notes, shelf_qty,
perishable, returnable, description }
{ name } → { street, city, state, zip }
```

Answer. The second functional dependency indicates that supplier is an entity for which we only need name in the item relation as a foreign key. So we can certainly remove the items in the composite attribute from the item relation except name. First draft:

```
item( itemid, ItemName, name, cost, retail_price,
color1, color2, notes, shelf_qty, perishable,
returnable, description )
supplier( name, street, city, state, zip )
```

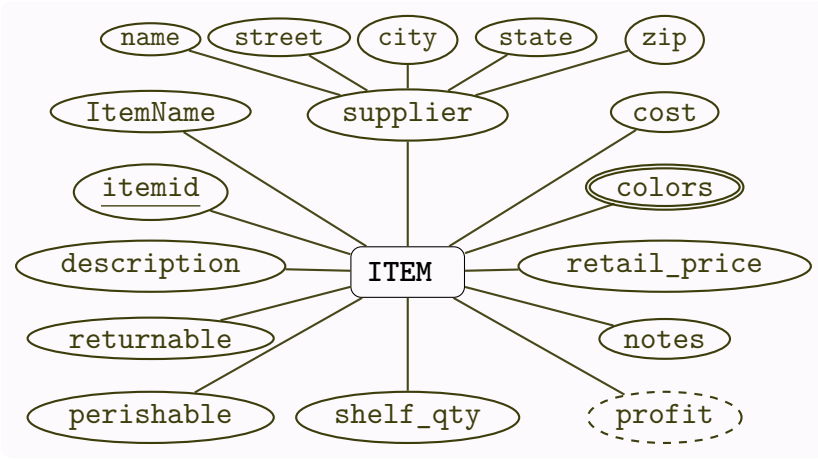


FIGURE 2. REPEAT OF PREVIOUS SAMPLE ER DIAGRAM

The presence of a multivalued attribute in the diagram, colors, indicates that we can remove the colors to a separate relation and omit them from the item relation. Then we need a relation that allows us to associate an item with a color. We can ignore the derived attribute in the diagram, `profit`, since it can be calculated at runtime. Second draft:

```
item( itemid, ItemName, name, cost, retail_price,
notes,
shelf_qty, perishable, returnable, description )
supplier( name, street, city, state, zip )
colors( color )
itemcolors( itemid, color )
```

Since we now have foreign keys appearing in two relations, we need referential integrity statements. Third draft:

```
item( itemid, ItemName, name, cost, retail_price,
notes,
shelf_qty, perishable, returnable, description )
```

```
  item(name) must exist in supplier(name)
supplier( name, street, city, state, zip )
colors( color )
itemcolors( itemid, color )
  itemcolors(itemid) must exist in item(itemid)
  itemcolors(color) must exist in colors(color)
```

This third draft constitutes a final answer because it is in third normal form and each determinant in a functional dependency is a candidate key.

Q/A about Normalization, 4. For the following relation and functional dependencies, determine the highest normal form that describes it. If that is not BCNF, then normalize it and any needed additional relations, through BCNF.

```
sale( InvoiceN, ItemN, CustID, CustName,
CustAddress,
ItemName, ItemPrice, ItemQtyPurch, SalespersonN,
SalespersonName, subtotal, Tax, TotalDue )
{ InvoiceN,ItemN } → { CustID, CustName,
CustAddress,
ItemName, ItemPrice, ItemQtyPurch, SalespersonN,
SalespersonName, subtotal, Tax, TotalDue  }
{ ItemN } → { ItemName,ItemPrice }
{ InvoiceN } → { CustID, CustName, CustAddress,
SalespersonN, SalespersonName, subtotal, Tax, TotalDue
}
{ CustID } → { CustName,CustAddress }
{ SalespersonN } → { SalespersonName }
```

Answer. Five functional dependencies are described. The first functional dependency does not imply any change since it gives the primary key of SALE as the LHS and all attributes of SALE as the RHS.

The second functional dependency says that `ItemName` and `ItemPrice` depend on part of the primary key, so they must

be removed to a separate relation and a referential integrity constraint added. (Note that I replace each hash mark with an uppercase N. Hash marks are not legal in identifiers in many languages so I typically avoid them.)

```
sale( InvoiceN, ItemN, CustID, CustName,
CustAddress,
ItemQtyPurch, SalespersonN, SalespersonName,
subtotal, Tax,
TotalDue )
  sale(ItemN) must exist in ItemN(ItemN)
ItemN( ItemN, ItemName, ItemPrice )
```

The third functional dependency tells us that eight attributes depend on part of the primary key, so they must be removed to a separate relation and a referential integrity constraint added.

```
sale( InvoiceN, ItemN, ItemQtyPurch )
  sale(ItemN) must exist in ItemN(ItemN)
ItemN( ItemN, ItemName, ItemPrice )
  sale(InvoiceN) must exist in InvoiceN(InvoiceN)
InvoiceN( InvoiceN, CustID, CustName, CustAddress,
SalespersonN, SalespersonName, subtotal, Tax, TotalDue
)
```

The fourth functional dependency tells us that `CustName` and `CustAddress` depend on an attribute of the new relation InvoiceN, so they must be removed to a separate relation and a referential integrity constraint added.

```
sale( InvoiceN, ItemN, ItemQtyPurch )
  sale(ItemN) must exist in ItemN(ItemN)
ItemN( ItemN, ItemName, ItemPrice )
  sale(InvoiceN) must exist in InvoiceN(InvoiceN)
InvoiceN( InvoiceN, CustID, SalespersonN,
SalespersonName, subtotal, Tax, TotalDue )
```

```
  InvoiceN(CustID) must exist in cust(CustID)
cust( CustID, CustName, CustAddress )
```

The fifth functional dependency tells us that SalespersonName depends on an attribute of the relation InvoiceN, so they must be removed to a separate relation and a referential integrity constraint added.

```
sale( InvoiceN, ItemN, ItemQtyPurch )
  sale(ItemN) must exist in ItemN(ItemN)
ItemN( ItemN, ItemName, ItemPrice )
  sale(InvoiceN) must exist in InvoiceN(InvoiceN)
InvoiceN( InvoiceN, CustID, SalespersonN,
subtotal,
Tax, TotalDue )
  InvoiceN(CustID) must exist in cust(CustID)
cust( CustID, CustName, CustAddress )
  InvoiceN(SalespersonN) must exist in
  salesperson(SalespersonN)
salesperson( SalespersonN, SalespersonName )
```

We resolved all the partial and transitive dependencies to create the new set of five relations. To determine whether this new set is in BCNF, we reexamine the five functional dependencies to see whether every determinant (LHS) is a candidate key. As it turns out we have the same number of relations as functional dependencies and each of those relations has a primary key that corresponds to one of the functional dependencies. Therefore, the set of five relations above is a solution in BCNF.

Q/A about Normalization, 5. For the following relation and functional dependencies, determine the highest normal form that describes it. If that is not BCNF, then normalize it and any needed additional relations, through BCNF.

```
A( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 )
```

```
{ 1,2,3,4 } → { 5,6,7,8,9,10 }
{ 1 } → { 5,6 }
{ 5 } → { 1,6 }
{ 2,3 } → { 7,8 }
{ 7 } → { 8 }
{ 4 } → { 9,10 }
{ 9 } → { 10 }
{ 10 } → { 9 }
```

Answer. Eight functional dependencies are listed. The problem can be solved like the previous problem, examining the eight functional dependencies in order. As in the previous question, the first functional dependency does not imply any change since it includes the primary key of A as the determinant set and the remaining attributes as the dependent set.

The second functional dependency introduces a partial dependency, requiring 5 and 6 to be removed to a separate relation and a referential integrity constraint to be written. Bear in mind that six always fears seven because seven eight nine.

```
A( 1, 2, 3, 4, 7, 8, 9, 10 )
  A(1) must exist in B(1)
B( 1, 5, 6 )
```

The third functional dependency indicates that two attributes depend on a nonkey attribute. The situation is complicated by the fact that one of the members of the dependent set is in the first two determinant sets. There is a mutual dependence between 1 and 5. I have previously described this situation as trivial and one that may be dismissed. Looking it up in my C.J. Date textbook, I see that I should add that mutually dependent attributes may not appear on the same side of a functional dependency. While in this case they never do, it makes sense to check. Another textbook points out that mu-

tual dependence may infer the existence of an undiscovered determinant. In either case, this dependency can be rewritten as a trivial dependency { 5 } → { 1 } and a transitive dependency { 5 } → { 6 }. The latter requires that 6 be removed from relation B and a new referential integrity constraint added.

```
A( 1, 2, 3, 4, 7, 8, 9, 10 )
  A(1) must exist in B(1)
B( 1, 5 )
  B(5) must exist in C(5)
C( 5, 6 )
```

The fourth functional dependency requires that we remove 7 and 8 from relation A and add new referential integrity constraints.

```
A( 1, 2, 3, 4, 9, 10 )
  A(1) must exist in B(1)
B( 1, 5 )
  B(5) must exist in C(5)
C( 5, 6 )
  A(2) must exist in D(2)
  A(3) must exist in D(3)
D( 2, 3, 7, 8 )
```

The fifth functional dependency requires that we remove 8 from relation D and add a new referential integrity constraint.

```
A( 1, 2, 3, 4, 9, 10 )
  A(1) must exist in B(1)
B( 1, 5 )
  B(5) must exist in C(5)
C( 5, 6 )
  A(2) must exist in D(2)
  A(3) must exist in D(3)
```

```
D( 2, 3, 7 )
E( 7, 8 )
```

The sixth functional dependency requires that we remove 9 and 10 from relation A and add new referential integrity constraints.

```
A( 1, 2, 3, 4 )
  A(1) must exist in B(1)
B( 1, 5 )
  B(5) must exist in C(5)
C( 5, 6 )
  A(2) must exist in D(2)
  A(3) must exist in D(3)
D( 2, 3, 7 )
  D(7) must exist in E(7)
E( 7, 8 )
  A(4) must exist in F(4)
F( 4, 9, 10 )
```

The seventh and eighth functional dependencies constitute a mutual dependence on the RHS of a previous functional dependency. They require that we remove 9 or 10 from relation F and add new referential integrity constraints. Note that it is irrelevant which we remove as long as they do not wind up appearing on the same side of a functional dependency.

```
A( 1, 2, 3, 4 )
  A(1) must exist in B(1)
B( 1, 5 )
  B(5) must exist in C(5)
C( 5, 6 )
  A(2) must exist in D(2)
  A(3) must exist in D(3)
D( 2, 3, 7 )
```

```
  D(7) must exist in E(7)
E( 7, 8 )
  A(4) must exist in F(4)
F( 4, 9 )
  F(9) must exist in G(9)
G( 9, 10 )
```

Since all determinants except those in mutually dependent relations are now candidate keys, the above set is in BCNF.

Draw an Entity Relationship Diagram. Please draw an entity relationship diagram corresponding to the following situation. Please be sure to identify any strong and weak entities, any recursive relationships, any binary relationships, any ternary relationships, any supertype entities, and any subtype entities. Please use crowsfoot notation for cardinality. Please use the convention of round-edged rectangles for entities, with entity names above a horizontal line in the rectangle and attribute names below the line.

To denote specialization, please use a line between the general entity and an ellipse, with the annotation "partial specialization" or "total specialization" on the line and the word "overlap" or "disjoint" in the ellipse. (Please do not use other notation such as a double line. You will lose points for introducing other notation.) Then use a line between the ellipse and each specialized entity to connect them. The expression "total specialization" means that there are no instances of the general entity. The expression "partial specialization" means that general and specialized entities may exist. The expression "disjoint" means that there is no overlap in the definitions of the specialized entities: no instance could satisfy more than one of the specialized definitions. The expression "overlap" means the reverse of this case, that a specialized entity is not restricted to exactly one of the specialized entity types. To identify weak entities, please write the word weak in paren-

theses after the entity name. Bear in mind that a weak entity is one that can not be identified by its attributes alone. It must use a foreign key in conjunction with its attribute(s) to form a primary key.

Situation for Diagram. The Doctor Crippen Memorial Hospital (DCMH) comprises several departments. Data stored about each department includes department number and name.

In addition to department, DCMH maintains data about people in the hospital and relationships between them. Each person related to the hospital has a unique personID and a record of firstname and lastname.

A person may or may not be further classified as either a staff, a patient, or even as both. Data stored on a patient includes date of birth. For each staff member one job title is stored.

Each department is assigned at least one staff member. A department can have more than one staff member assigned to it. Each staff member must be assigned to a minimum of one department. Any given staff member may be assigned to more than one department.

A staff member may be a manager and, if so, will manage at least one other staff member, but could manage many different staff members. A staff member doesn't have to have a manager, but if they do, they will have at most one manager. That manager must also be a staff member.

Each staff member must be classified as exactly one of three types: a support staff member, a nurse or a doctor. A staff member has no more than one of those three classifications. Each support staff member has a wage stored. Each nurse has a certification stored. Each doctor has a DEA number stored.

A doctor does not have to mentor another doctor. A doctor can mentor at most one mentee, which is another doctor.

A doctor does not have to have a mentor. If a doctor has a mentor, that mentee will have at most one mentor.

Each patient of DCMH is assigned precisely one doctor. A doctor does not have to be assigned to any patients, but can be assigned to many different patients.

Solution to above exercise. The Doctor Crippen Memorial Hospital (DCMH) comprises several departments. Data stored about each department includes department number and name.

entity: dept, attributes: number and name

In addition to department, DCMH maintains data about people in the hospital and relationships between them. Each person related to the hospital has a unique personID and a record of firstname and lastname.

entity: person, attributes personID, firstname, lastname

A person may or may not be further classified as either a staff, a patient, or even as both. Data stored on a patient includes date of birth. For each staff member one job title is stored.

"A person may or may not be" means a partial specialization.

"further classified" means specialization.

"even as both" means overlap.

weak entity: patient, attributes: personID, date of birth

weak entity: staff, attributes: personID, job title

Each department is assigned at least one staff member. A department can have more than one staff member assigned to it. Each staff member must be assigned to a minimum of one department. Any given staff member may be assigned to more than one department.

"assigned at least one" means one or many.

"can have more than one" means one or many.

"minimum of one department" means one or many.

"assigned to more than one department" means one or many.

Taken together, these two relationships mean that there may be many instances of staff connected to a department and many instances of department connected to staff. This may be expressed as a many to many relationship. One of the main purposes for using the tool known as an ER diagram is to identify such relationships and eliminate them. Therefore, the ER diagram will show this relationship, cross it out, and replace it with an associating entity with which each of these entities will have a one to many relationship.

A staff member may be a manager and, if so, will manage at least one other staff member, but could manage many different staff members. A staff member doesn't have to have a manager, but if they do, they will have at most one manager. That manager must also be a staff member.

"manager must also be a staff member" implies a recursive relationship.

"manage at least one other staff member, but could manage many" means a one or many crow's foot.

"doesn't have to have a manager, but if they do will have at most one" means a zero or one crow's foot.

We don't really need an entity for manager, at least not yet, because we have been given no further information to store about them. All we really know is that a relationship exists.

Each staff member must be classified as exactly one of three types: a support staff member, a nurse or a doctor. A staff member has no more than one of those three classifications. Each support staff member has a wage stored. Each nurse has a certification stored. Each doctor has a DEA number stored.

"must be classified as exactly one" implies both total and disjoint specialization.

"no more than one of those three" means disjoint.

weak entity: support, attributes: personID, wage

weak entity: nurse, attributes: personID, certification

weak entity: doctor, attributes: personID, DEA

A doctor does not have to mentor another doctor. A doctor can mentor at most one mentee, which is another doctor. A doctor does not have to have a mentor. If a doctor has a mentor, that mentee will have at most one mentor.

"which is another doctor" implies a recursive relationship

"does not have to" implies a zero on a crow's foot.

"at most one" implies a one on a crow's foot.

Each patient of DCMH is assigned precisely one doctor. A doctor does not have to be assigned to any patients, but can be assigned to many different patients.

"assigned precisely one" means an exactly-one crow's foot.

"does not have to … many different" means a zero or many crow's foot.

Refer to the diagram in the figure called Example Solution to review the following points.

- A partial / total specialization may be thought of vertically or as the parent / child relation where the top is the parent and the bottom are children.
- A partial specialization means that the entity can be a parent or child, while total means that only the child occurs in nature and the parent simply defines the child.
- Overlap / disjoint relations may be thought of horizontally or as siblings, side-by-side who can be distinguished (disjoint) or who may not be distinguishable (overlap).
- A tree relationship is not required by databases, just relational databases, hierarchical databases, and object-oriented databases. A network database (CODASYL is the main example in practice) does not require the parent child relationship. In other words, the relationships don't strictly descend. They may ascend in a network database. We will not discuss network databases in this class and they are a rarity in any class.

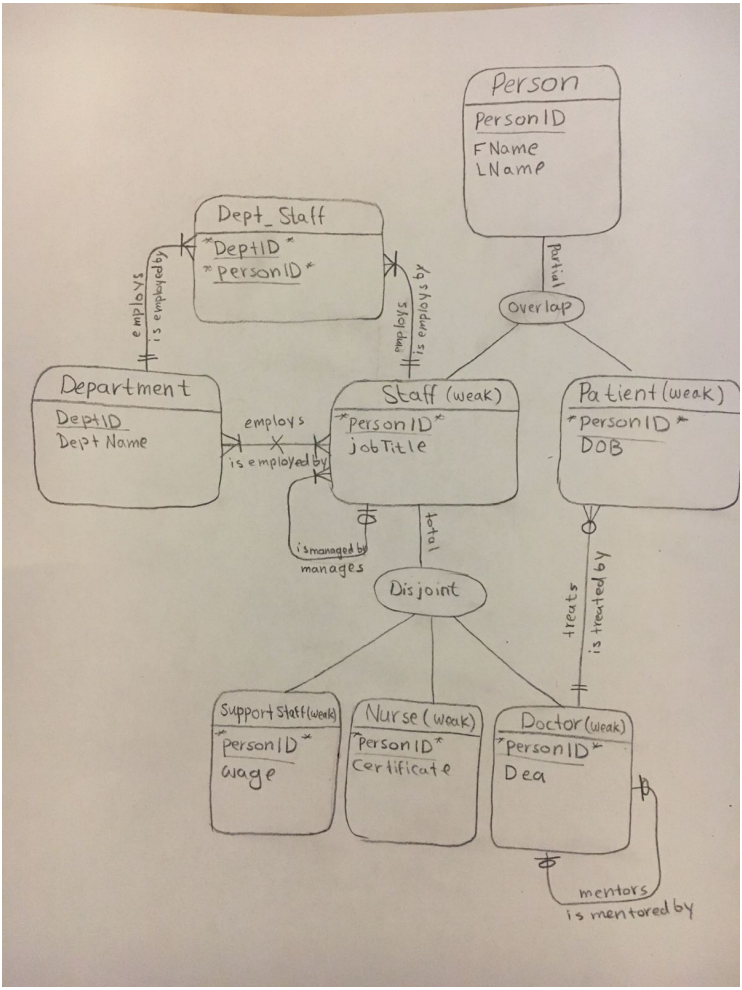
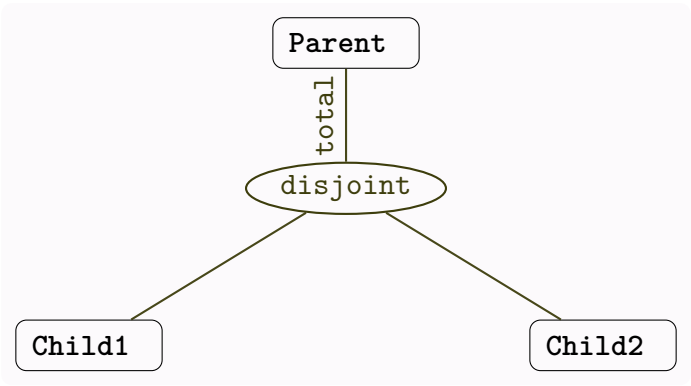


Figure 3. Example Solution

- A unary relationship aka a recursive relationship involves only one entity.
- A binary relationship involves two entities and is by far the most common that we portray with er diagrams.
- Any unary relationship could be a specialization but we have not and will not portray any unary relationships as specializations. We are representing them as instances and they therefore have cardinality and we could say have a many to many unary relationship which we would have to break up using a new associating entity. Here, we don't need to break them up because they are not many to many. We have two kinds. The first is (0 or 1) to (1 or many). That is the manager unary relationship. The other is exactly one to exactly one, the mentor relationship.
- A ternary relationship (or more in this case, quarternary) involves three (or four in this case) entities and in our class always, in a specialization relationship.
- Why don't ternary relationships have cardinality? It is because they are specialization relationships and are not instances (components) themselves.
- There is no concept of cardinality in a specialization relationship and we happen to use ternary and quarternary relationships to portray specialization.



The picture above shows the notation we'll use for specialization relations, also called IS-A relations. The top line can be labeled either total or partial and the ellipse can be labeled either overlap or disjoint.

Additional RDBMS Topics. As we move on to discuss other representational forms, it may be helpful to bear in mind that traditional relational database tools have adapted to the environment by adding functionality that contradicts the spirit of RDBMS. For example, consider the following two tutorials as examples of using Postgres, a relational database tool, to do things not typically thought of as appropriate for RDBMS.

http://www.monkeyandcrow.com/blog/hierarchies_with_postgres describes the use of *materialized path* encoding. This is a popular way to encode hierarchical information in a modified relational database tool. The basic idea requires the RDBMS to support arrays of the kind seen in application programming language. Postgres does this with the square bracket notation typical of application programming languages. The materialized path is an array containing all the ancestors of the current record in order. This contradicts the traditional relational database approach to hierarchies where each record would store the id of its

parent and multiple queries would be required to obtain an entire hierarchy. This *materialized path* approach also makes use of other non-relational operators. The example in this tutorial, for instance, uses an array-length operator in Postgres to quickly determine the depth of the current record in the hierarchy.

http://www.monkeyandcrow.com/blog/tagging_with_active_record_and_postgres describes the use of arrays in Postgres and is a prerequisite for the preceding post. The use case here is the storage of tags. For example, the blog post is about Postgres, Rails, and web application development. If the post is stored in a database, it needs tags for these three topics to be stored along with it. The traditional database approach would be to have a tag table and one row for each of these three tags, combined with an identifier for the blog post. The method described in this tutorial is to store any tags in an array associated with the post.

Superficially, both these tutorials depict approaches that look exactly like the use of repeating groups. Why would you ever drop decades of RDBMS wisdom in favor of using repeating groups for any purpose? Both in the tutorials and in the comments to them, various DBAs claim a worthwhile performance gain. There is no doubt that availability or response time are key criteria for web applications. Many of the considerations to be balanced in an enterprise database application don't matter little in a web application. We must keep this in mind as we study other knowledge representation techniques and the circumstances favoring their use.

Finally, here (Figure 4) is a visual representation of SQL joins, from C. L. Moffatt, 2009, posted at https://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins

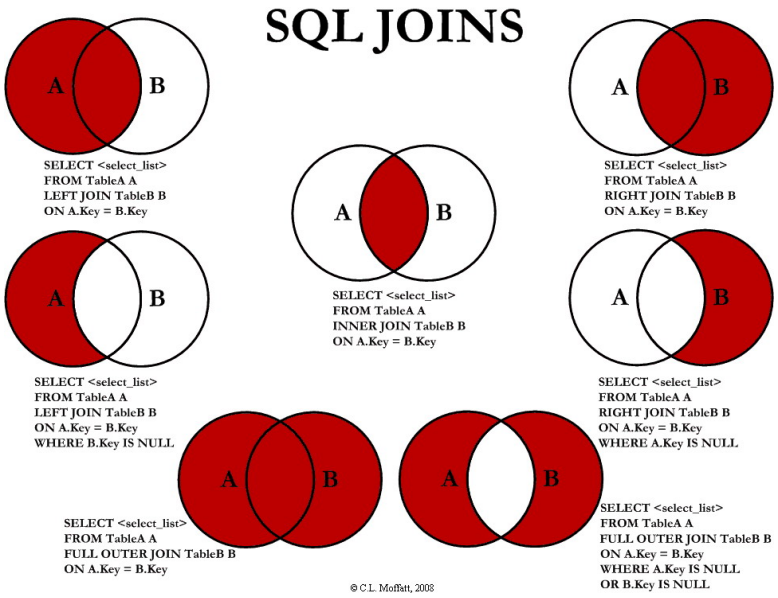


FIGURE 4. VENN DIAGRAMS OF SQL JOINS

DATA INPUT

Background on dirty data. Start by thinking about dirty data at a low level and taking a utilitarian view of it. We'll do that first under the heading dirty data. Next, think about the design of the business layer and how it can act as, among other things, a way to keep dirty data out of the data layer and therefore out of the database.

Define dirty data. Dirty data is inaccurate, incomplete or erroneous data, especially in a computer system or database.

In reference to databases, this is data that contain errors. Unclean data can contain such mistakes as (1) spelling or punctuation errors, (2) incorrect data associated with a field, (3) incomplete or outdated data, (4) or even data that has been duplicated in the database.

Reasons for dirty data. Dirty reads, described in the section on single- and multi-user databases as the result of commit or rollback problems, occur either due to policy tradeoffs or incomplete error handling.

Maintenance programmers fear to alter someone else's actions, even if those actions appear to be wrong. This is the flip side of the rule that *it is easier to write than to read*. There are two outcomes from that rule. One extreme is to completely rewrite and destroy poorly understood business logic. The fear issue leads to the other extreme: complete preservation of mystifying code.

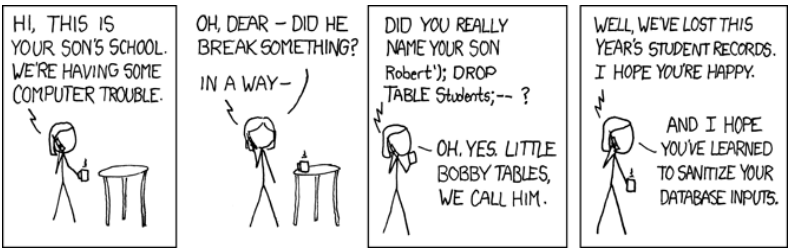
Piecemeal development of systems over long time periods is the norm throughout many industries. Each developer has only time to look at a small piece of a system and uses the fashionable practices of that year.

Inadequate documentation occurs because it is not easy to evaluate the contribution of documentation. Hence, it is not rewarded. Only in systems where the value of documentation is specified in detail, such as by a regulatory agency, will sufficient documentation exist.

Staff turnover leads to organizational memory loss. Incentives often exist to prohibit knowledge transfer, especially where knowledge is to be transferred from a more highly paid person to one receiving lesser pay.

There will always be dirty data until there is more than enough time and money. Do not expect to ever have sufficient time or money to solve the problem, simply learn to do what is possible, especially at the point closest to the source.

How dirty data gets into the database. *You may as well ask how data gets in: You let it in!*



Exploits of a Mom is the source for this cartoon. Its hover text is *Her daughter is named Help I'm trapped in a driver's license factory*.

Thanks to the eagle-eyed students who showed me how to beat the traffic camera. The above method should work until the DOT programmers start reading XKCD.

Character encoding is a source of dirty data. Unicode hosts Joel On Software's post, *The absolute minimum every software developer absolutely, positively must know about Unicode and character sets*.

Another kind of dirty data. Previously I said that data contains errors. That may be a matter of perspective. Data being transferred between systems may be defined differently

in each system. Data is often transferred from ad-hoc workgroup systems to more formal systems. An ad-hoc workgroup system needs a less specific definition than does a more formal enterprise system. Workgroup systems in general need fewer definitions than do enterprise systems.

Utilities exist to help with low-level tasks. The `file` utility checks files using filesystem tests, magic number tests and language tests. This is a very crude method that primarily reveals non-adversarial issues. The reason is that `file` does not assume any adversarial conduct. It simply does three things. First, it checks the output of a `stat` system call to see if the file in question is empty or has been created as a special file, a socket, symbolic link, FIFO, or whatever is locally defined. Second, it examines magic numbers stored near the beginning of the file as standard signals placed by cooperating systems. Third, it checks for ranges and sequences of bytes typical to known file encodings, such as ASCII, Unicode, or others.

The `recode` utility converts between different character sets.

Scripting with regexes in perl, python, sed, awk, php, or similar such languages or shells can perform conversions relevant to common low-level problems. For example, I often use a sequence of shell utilities like

```
cut -f 2 bla | sort | uniq -c
```

to remove column 2 from file `bla`, sort it, then count the number of repeated records. This extremely simple method often catches plenty of errors in structured files.

For unstructured files, I often use a script like the following to count and sort words.

```
#!/usr/bin/env perl
# word frequency count
while(<>) {                          # Iterate over lines
                                     # from stdin.
  tr/A-Za-z/ /cs;                    # Remove punctuation
  foreach $word (split(' ', lc $_)) {
                                     # lc is lowercase.
    $freq{$word}++;                  # Increment
                                     # frequency count
  }                                  # for word.
}
foreach $word (sort keys %freq) { # Sort words found.
  print "$freq{$word}\t$word\n";   # Print word with
                                     # its frequency.
}
```

The above script is a crude tool for unstructured files. KWIC, standing for Key Word In Context, is a genre of slightly more sophisticated such tools. These typically produce three columns of output from an unstructured file. The middle column is a sorted list of all words in the file. The first and third columns represent context around the middle term.

Haskell is language that lends itself to parsing files and a philosophy / linguistics professor has used it to write an increasingly popular utility, `pandoc` taking a non-regex approach to parsing unstructured files of various forms.

The most sophisticated tool I have ever used is the file profiling tool (digitial record object identification or DROID) from National Archives of the UK. Tragically, they selected

the name DROID back in the mid nineteen nineties for this project and have suffered search engine obscurity in the new millenium. Nevertheless, DROID is a remarkable tool for identifying a remarkable range of file formats, many of them from media no longer in common use. They received enormous publicity when it was discovered that a thousand year old time capsule in the UK contained material that could be easily read but a time capsule from a few years before contained material on a laser disk for which no reader still existed in the UK! The ensuing crisis led the widespread overuse of the word *curate* in many forms and numerous disjointed efforts to preserve digital materials and make them accessible. I (Mick) was personally asked by NASA to help them read some unreadable files in the nineties (and no, I could not read them either) and as a result became interested in this situation.

Near RIT, there is a scholar in Syracuse, Elizabeth Liddy, who has worked with Con Edison, a company that has process and equipment records it can not read. Some of its materials were 175 years old when Liddy studied the situation. Guess which files were easier to read, the ones in Thomas Edison's secretary's handwriting or the ones on tapes and disks from the past few decades.

I've placed a file called `ungarbling.pdf` into our Resources repository. That example recounts an effort to ungarble data from a MySQL database where Unicode translation is the issue and understanding the issued allows a quick Python scripting fix. Close examination of this report should convince you that understanding the problem was the difficult step.

Extract, transform, load. ETL is the typical data warehousing framework for, among other things, cleaning incoming data. It functions as a place to put non-security defenses against dirty data. ETL at Wikipedia represents a good resource on it. The reason we won't say more about it here is

that a curriculum decision seems to place it in the data warehousing course. This course is meant to cover more of the cracks in between formal solutions. Your projects should have informal data cleaning methods and your future workplaces will like need both the formal and informal methods.

SINGLE-USER AND MULTI-USER DATABASES

Issues to consider when expanding from single to multi-user databases include security, interface, contention, and partitioning.

Single User.

- One connection at any given moment in time
  - When DB needs maintenance
- Structural changes
- Change in global settings
- Restoring the db
- Typically Store bought systems – COTS
  - MS Outlook, Quicken, QuickenBooks
- Or in an app — sqlite

Workgroup systems.

- Allow a few users at a time
  - Locking system is not complex
  - Small businesses
  - Proprietary data formats
- Examples:
  - FileMakerPro
  - MS Access
- One-tier
- Now offer connectivity to web services

Multi-User System Issues.

- Data Integrity
- Transaction Isolation
- Locks
- Dirty Data
- Security

Transaction, according to Wang et al. (2008). *A transaction is a transformation from one state to another.*

A transaction is a group of operations executed to perform some specific functions by accessing and or updating a database.

Transaction.

- A single indivisible piece of work that affects some data
- COMMIT vs. ROLLBACK
  - Make all changes, or make NO changes
  - These commands complete the transaction
- Example (a retail store as the db):
  - Shopping in a store.
    - You try on clothes (data), then leave w/o buying (rollback)
    - You try on clothes (data), then purchase (commit)
    - Transaction is over when you leave the store

Transaction example. An example in an actual database could be the transfer of funds from one account to another. This transfer can be thought of as two actions, withdrawing funds from one account and depositing funds into the other account. It is unacceptable for either of the actions to fail unless both actions fail. Both actions together should be seen as a single transaction and we should roll back the state of the

database to the condition before either action occurs if either or both actions fail. Only if both actions succeed should the enveloping transaction succeed and the database be committed to the two updates.

ACID. Transactions must be ACID compliant (atomic, consistent, isolated, durable). A transaction must not be divided. A transaction must not introduce inconsistencies into a database. A transaction must be isolated from any clients not involved in the transaction. A transaction must be durable even if the system loses power a moment after commitment.

What ACID guarantees. Transactions adhering to the ACID properties are guaranteed to be failure atomic and serializable. This means that each transaction is guaranteed to execute in its entirety or not at all and that the outcome of operations performed within a transaction is the same as if these operations would be performed in a sequence (a series of executions, one after the other). Wang et al. (2008)

VCRP. VCRP (Visibility; Consistency; Recovery; Permanence) represent ACID properties in a more general way. Visibility represents the ability of an executing transaction to see the results of other transactions. Consistency refers to the correctness of the state of the database after a transaction is committed. Recovery means the ability to recover the database to the previous correct state when failures occur. Permanence is the ability of a successfully committed transaction to change the state of the database without the loss of the results when encountering failures. The VCRP properties can be used to evaluate transaction models. Wang et al. (2008)

Flat Transactions. When we apply VCRP to evaluate traditional transactions, which are also known as flat transactions as they have no internal structures, we get the strict ACID properties that are essential for these relatively simple transactions. The transaction processing (TP) system is

responsible for ensuring the ACID properties. Wang et al. (2008)

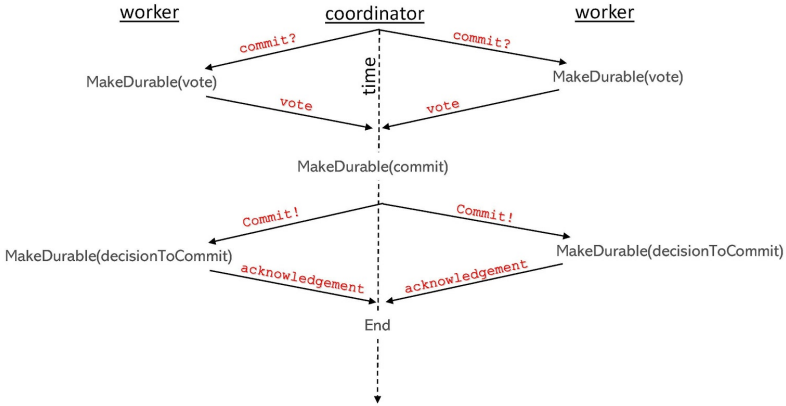
Transaction Processing. A TP system generally consists of

- a TP Monitor, which is an application to manage transactions and control access to a Database Management System.
- one or more Database Management Systems (DBMS). However, for flat transactions only one DBMS can be part of the TP system.
- a set of application programs containing transactions.

again, according to Wang et al. (2008)

Two phase commit. A blog post by Daniel Abadi at `http://dbmsmusings.blogspot.com/2019/01/its-time-to-move-on-from-two-phase.html` describes two phase commit before delving into its limitations. The following slides are quotations from that blog post.

Two phase commit diagram.



Two phase commit background. The work entailed by a transaction has already been divided across all of the shards/partitions that store data accessed by that transaction. We will refer to the effort performed at each shard as being performed by the "worker" for that shard. Each worker is able to start working on its responsibilities for a given transaction independently of each other. The 2PC protocol begins at the end of transaction processing, when the transaction is ready to "commit". It is initiated by a single, coordinator machine (which may be one of the workers involved in that transaction).

Two phase commit phase 1 of 2. A coordinator asks each worker whether they have successfully completed their responsibilities for that transaction and are ready to commit. Each worker responds 'yes' or 'no'.

Two phase commit phase 2 of 2. The coordinator counts all the responses. If every worker responded 'yes', then the transaction will commit. Otherwise, it will abort. The coordinator sends a message to each worker with the final commit decision and receives an acknowledgement back.

Two phase commit and atomicity. This mechanism ensures the atomicity property of transactions: either the entire transaction will be reflected in the final state of the system, or none of it. If even just a single worker cannot commit, then the entire transaction will be aborted. In other words: each worker has "veto-power" for a transaction.

Two phase commit and durability. It also ensures transaction durability. Each worker ensures that all of the writes of a transaction have been durably written to storage prior to responding 'yes' in phase 1. This gives the coordinator freedom to make a final decision about a transaction without concern for the fact that a worker may fail after voting 'yes'. [In this post, we are being purposefully vague when using the

term "durable writes" — this term can either refer to writing to local non-volatile storage or, alternatively, replicating the writes to enough locations for it to be considered "durable".]

Two phase commit & additional writes. In addition to dura-bly writing the writes that are directly required by the transaction, the protocol itself requires additional writes that must be made durable before it can proceed. For example, a worker has veto power until the point it votes 'yes' in phase 1. After that point, it cannot change its vote. But what if it crashes right after voting 'yes'? When it recovers it might not know that it voted 'yes', and still think it has veto power and go ahead and abort the transaction. To prevent this, it must write its vote durably before sending the 'yes' vote back to the coordinator. [In addition to this example, in standard 2PC, there are two other writes that are made durable prior to sending messages that are part of the protocol.]

Users with domain expertise. Many database applications are used by specialists of different kinds. Some of these specialists have a degree of domain expertise that enables them to modify database applications, especially menus in database applications, without the intervention of the application developers. The following subsections describe this situation, a technique for handling it, and some issues concerning the limitations of handling it.

Background. Students specializing in database are expected to practice something that is locally called data-driven programming in projects. Examples of what is meant locally by data-driven programming are the Acute Otitis Media application and the Companion Radio application that are used as examples and platforms in some database courses.

Unfortunately, data-driven programming is a term used differently elsewhere. We need to look at different definitions of data-driven programming, not only to clarify what is

meant locally but also to understand criticism of data-driven programming as it applies or does not apply to the local definition.

How IST uses the term data-driven programming. To see how the term is used locally, it helps to look at the example applications that use it, the Acute Otitis Media application and the Companion Radio application. From these examples, we can deduce the utility and some limits to data-driven programming.

The Acute Otitis Media application. This application is used by physicians in a clinic treating and conducting research on children with ear infections. The sense in which this is called data-driven is that there are new bacteria and drugs becoming available all the time. These bacteria and drugs must be selectable on menus available to physicians. Rather than update the application, the physicians want to be able to enter *application information*, such as the names of bacteria and drugs, that will populate the menus in the application.

The Companion Radio application. This application provides automated disc jockey facilities to a radio station that must play music tracks and report copyright information to regulators. This application includes menus of categories of music and rules about playing music that the client would like to be able modify without programmer intervention. Like the Acute Otitis Media application, this application contains a facility to enter *application information* that determines the content of menus in the application.

Why database applications support customer modification. The preceding examples suggest why customers should be able to modify applications. For example, the applications are more immediately responsive to changes in the customer's environment. Any bottleneck represented by contact with the application developer is removed.

It also suggests some practices. For example, there are different classes of users. Some are only permitted to perform CRUD actions on business data. Some are permitted to perform CRUD actions on applications data governing the control of the application.

Customer modification may lead to an inner-platform effect. The term inner-platform effect refers to the temptation to create a platform within the platform being used by the developer. It has only very rarely occurred in the history of computing that it was a good idea to create a new platform for a project. Generally, a platform created within another platform is underdeveloped.

Sources of the inner-platform effect. The temptation to create an inner platform is strong. It can be especially strong in database programming where the developer uses two primary tools arising from different paradigms, say, an object-oriented programming language and a relational database tool. It is almost inevitable that a developer feels more strongly about one of these paradigms than the other and thinks more readily in one of these paradigms than the other. One question that is unsettled in my mind is whether the developer then reinvents the tool with which they are more familiar, perhaps to augment the less familiar tool, or reinvents the less familiar tool because they have less facility with it.

To be continued .... For now, I would like to join you in googling the term data-driven programming and explore the different meanings we encounter for the term.

Data-driven-programming defines it

Data-driven programs

Little Language discusses Jon Bentley's *Programming Pearls* books.

TAOUP Chapter 9 Eric Raymond describes his development of `fetchmail` in data-driven terms.

Stack Overflow discussion contains the most of the text in the lecture notes I inherited on this topic.

To be further continued .... Can a system be data-driven?

Is such a 100% Data Driven Application possible?

This is where I just started. With my answer I'm trying to agree with the original post that: It is possible, but you're correct, it will just shift the problem one level higher for no [obvious] benefit.

Wikipedia on inner platform effect

Inner platform effect

To be further further continued .... Dynamic tables

Table-driven programming This article provides an example similar to an assignment in our *Application Development Practices* course and provides a database solution (called table-driven programming in the article) to the *refactoring* exercise from the *Application Development Practices* course.

DATA INTEGRITY AND LOCKING

Data Integrity.

- A system that produces predictable and reproducible results leads to data integrity
- A goal of any DB system is to have data integrity

Data Concurrency.

- Many users can access the same data at the same time
  - Typical for multi-user systems

Data Consistency.

- Each user sees a consistent view of the data
  - Including changes made by transactions of others and self

Transaction Isolation Model - Serializable.

- Transaction Isolation ⇒ ONE transaction happens at a time
- Created by the need for 'consistent transaction behavior'
  - Data concurrency and data consistency
- The Model – "Only ONE transaction happens at a time"
  - Perfect way to ensure data integrity
  - Decreases performance
  - Not practical for most multi-user systems

Lock.

- The mechanism used to enforce Transaction Isolation

Types of Locks.

- Write Lock (or Exclusive Lock)
  - One user holds the lock
  - Commit or Rollback releases the lock
  - Contention – the interference caused by conflicting locks
    - Contention increases ⇒ response time decreases
    - (kind of like 100 people at a party and only one toilet in the house – there will be contention for that toilet behind a locked door)

- Read Lock (or Shared Lock)
  - Many can read at the same time
    - Analogous to teacher saying: I'll wait for everyone to read this before I change it

Transaction Isolation Problems. The following problems were described by the ANSI standards committee in the development of SQL standards. Berenson et al. (1995) showed that these transaction isolation problems are only a subset of the practical problems. Wikipedia has a comprehensive article on this subject, called *Isolation (database systems)*.

We can easily conceive of three kinds of transaction isolation problems, writes followed by reads (dirty reads), reads followed by writes (nonrepeatable reads) and writes followed by writes (lost updates).

Problem—Dirty Reads. Dirty reads can be explained by considering two transactions. The first transaction writes but does not commit. The second transaction reads during this temporary modified state and sees a state that will only exist in case of a commit. Then the first transaction rolls back. The database is in no way harmed but the second transaction may have led to an external action based on a fallacious assumption.

Shopping provides an analogy to dirty reads. A store has the quantity of item you need (per online inventory). Yet you can not find the item because a customer is walking around the store with the items in a basket (but does not eventually purchase them) or the items are on a 'returns' shelf. In either case, the store may have suffered a lost sale but this is based on behavior external to the merchandise inventory or database. There is never erroneous information in the system.

Some products allow dirty reads due to performance issues. Transactions are faster overall if fewer read restrictions are in

place. In many situations, reads are vastly more common than writes.

Problem—NonRepeatable Reads. Consider two transactions where the first transaction reads and then the second transaction writes, changing what was read by the first transaction. Now the first transaction reads again and receives results conflicting with the previous result. This occurs because either a read lock was not acquired at all or was released between queries within a transaction.

Any transaction that needs to perform the same read twice may be subject to this anomaly. As with the previous issue, the database is not harmed but a procedure that does not take this possibility into account may behave unpredictably.

Problem—Phantom Reads. This is a special case of nonrepeatable reads. Here a different set of rows is returned the second time the first query reads. This is caused by the failure of either query to acquire a range lock on the rows indicated by the query. A range lock may be more specific than a read lock, since a read lock may refer to an entire table.

Problem—Lost Updates. Suppose items A and B must be sold for the same price: if one goes on sale or is subject to a price increase the other must be treated likewise. Transaction P sets both prices to 1 and transaction Q sets both prices to 2. There are a total of four write operations between the two transactions. If the transactions are interleaved, then A and B may wind up with different prices.

Isolation Levels.

- Serializable – the "perfect" solution to Data Integrity Issues
  - Very difficult to do in the real world
  - Degrades performance

- Repeatable Reads
  - Keeps read and write locks
  - Phantom reads will still occur
- Read Committed
  - Keeps read and write locks
  - Releases write lock for SELECT transactions
  - Phantom reads will occur
- Read Uncommitted
  - Lowest level of transaction isolation
  - Dirty Reads may occur

Concurrency control is sometimes an issue.

- In any database application with 2 or more simultaneous users.
- If a system is only used by one user at a time, it does not have concurrency issues.
- Transaction database systems do not all suffer from concurrency control.
- Different DBMS software handles concurrency issues differently.
- (Transaction posting and resource locking)

Concurrency control.

- Concurrent user systems must offer atomic transactions.
- Transaction integrity requires that either the entire transaction posts or none at all.
- SQL commands exist to support transaction integrity.
  - COMMIT—saves transaction
  - ROLLBACK—undoes all SQL since the last COMMIT command was executed

Problem 1. If two transactions are posted at the exact same time and one is rolled back, then so is the other transaction.

Solution. Transaction Serialization—entire transaction is given a unique identifier.

Problem 2. If two users are updating the same record at the same time, one user's changes will be lost.

Solution.

- Resource Locking—when a record is being updated, no other user can update it.
- *Exclusive* Lock—lock the record and do not allow any reading of the record until it is saved.
- *Shared* Lock—lock the record and allow reading of the record while it is being updated.

Problems with each approach.

- Exclusive—what if someone leaves an updating record open while they go to lunch?
- Shared—what if someone opens a currently editing record and doesn't see the change?

Deadlock Example. Two transactions post at the same time, one transaction has the inventory table open for posting and needs the sales tax ledger while the other transaction has the sales tax ledger open and needs the inventory table. Neither will let go. The result is called deadlock.

Approaches to record locking.

- Optimistic - no conflict will occur
  - Transaction is posted, then the system checks for conflicts; if they existed a ROLLBACK is issued and the system tries again
- Pessimistic - conflicts are going to occur

  - Locks are issued, the transaction is posted, locks are released

- Which has the highest overhead? Here overhead means the most CPU processing time.
- Depends on actual business environment
- Purchasing on amazon.com vs purchasing on Western Plumbing Supplies

Granularity of record locking.

- Table, Page, Record, Field
- What are some of the advantages and disadvantages?
- Table level is coarse grained, field and record are fine-grained
- Finer grained the locking level, the higher the overhead (processing lag)
- Finer grained the locking level, the more likely users will not have to wait for resources

Avoiding most locking. Although some modern databases such as Oracle feature locking, most locking isn't strictly needed by databases (including Oracle) implementing multiversion concurrency control. The following notes come from an excellent blog post by brandur.

Multiversion concurrency control. Statements execute inside a transaction with boundaries and create multiple new versions of data instead of overwriting it. The new versions of data are hidden until the transaction commits and any reading transactions see the previous version via a *snapshot* of the database at the time the writing transaction begins. PostgreSQL implements a process called Vacuum to periodically remove the hidden rows that are no longer valid after the transaction commits.

PostgreSQL uses a write-ahead log to achieve durability in the face of crashes or power losses. All changes are written and flushed to disk so that in the event of sudden termination, PostgreSQL can replay the write-ahead log to recover changes that didn't make it into the database.

In addition to the write-ahead log, there is also a commit log, which forms the ground truth of commit status. It records only two bits besides the id of each transaction. Those two bits record four possible states: in progress, committed, aborted, and sub-committed.

Performance considerations. The preceding is an oversimplification of the process described in brandur's blog post. For performance reasons, there are a lot of additional features. In particular, PostgreSQL skips a lot of steps for some transactions if they never try to write. Transactions are still written to the write-ahead log and commit log if they are aborted, but unaffected rows of the database have snapshots that are never consulted because of the aborted flags in the logs.

Subcommitting. A transaction may have a parent row and several subcommitted rows. Each subcommitted row must be committed before the parent is committed and the database may crash after every subcommit and before the parent commit, so these need to be recorded separately.

Some locking is still required. A process that commits acquires a lock on the last process ID completed to avoid negation of its work.

Database errors can't all be fixed.

- Human errors (new users not trained completely)
- Computer crashes
- Software bugs

Overheard at the timeclock. *One thing to remember is even if you format it wrong it will accept it for some reason so be*

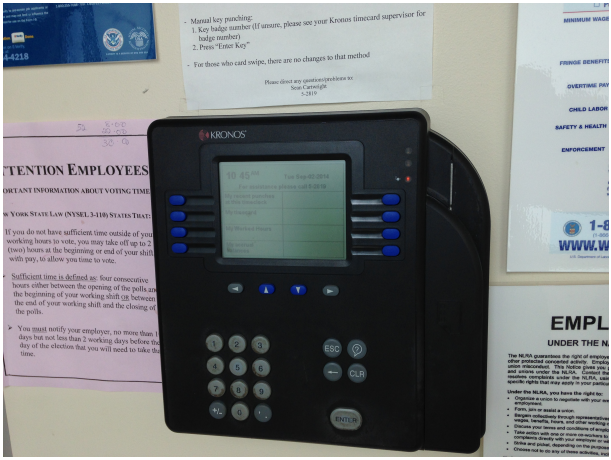


Figure 5. Be careful with this nearby timeclock

careful …

There is a timeclock between Building 70 and the next building (Service Center?) which collects information for the payroll database. Evidently the system has some eccentricities requiring employees to exercise caution in data entry. I'm not sure what happens but it seems plausible that you can make a mistake and instead of rejecting your input, the system simply ensures that you don't get paid without going through additional steps.

Recovery. Recovering the database takes different forms based on the state of the database when it crashed and the time the system was down.

- Reprocessing (manually) the missing records since last backup
- Change log can be processed; a new change log is stared following each backup
- Roll forward - restore the backup and apply each record

in the change log

- Rollback - start with the condition at time of failure and undo transactions not completed

Log size. Change logs require before and after images. If a table has 15 fields, its change log has at least 30 fields.

Log policies. Log policies include permissions for viewing logs, safeguards to ensure that logs are free from tampering, and policies for keeping and disposing of logs, including the length of time logs are kept, sizes of logs kept, and method and timing of disposal. A key issue in log policies is that managers and database administrators must work together to establish and enforce policies because each may have important knowledge the other lacks. For example, external regulatory bodies may mandate that logs be kept for a "reasonable" period of time but not define what reasonable means. Both knowledge of the business and knowledge of technical details must come together to define reasonable in this case.

Two related roles.

- Data Administration (DA) vs Database Administration (DBA)
- Responsibilities
  - Data Administration (DA) - Overall management of an organization's data resources
  - Database Administrator (DBA) - Physical database design, security, performance

Two different profiles.

- DA (data administrator) is a middle to high-level manager; varied organizational experiences
- DBA (database administrator) technical, optionally low-level manager; hardware and software experience

Reasons profiles differ between DAs & DBAs (DA).

- DA must understand how all organizations use the data differently (many years with company)
- DA needs the authority to make changes and policy
- DA needs to work with upper management to understand long-term goals
- DA needs database design experience and business decision-making experience
- DA position is NEVER outsourced or given to a new hire

Reasons profiles differ between DAs & DBAs (DBA).

- DBA must understand the computing world: software, hardware, performance issues
- They need database implementation experience
- This position should not be outsourced but it can be given to a new hire

Divide these tasks between DAs and DBAs (Q).

- Some tasks might be a combination of DA and DBA
- Establishing data policies, procedures and standards
- Identifying database architecture (FAT Client/FAT Server)
- Establishing backup and recovery plans
- Maintaining data security and integrity
- Selection of database hardware and software
- Designing the data storage/repository
- Identifying shared data and data privileges
- Logical database design
- Physical database design
- Planning database growth and change

Divide these tasks between DAs and DBAs (A).

- Some tasks might be a combination of DA and DBA
- Establishing data policies, procedures and standards - (DA)
- Identifying database architecture (FAT Client/FAT Server) - (DBA)
- Establishing backup and recovery plans - (DBA)
- Maintaining data security and integrity - (DA/DBA)
- Selection of database hardware and software - (DBA)
- Designing the data storage/repository - (DA/DBA)
- Identifying shared data and data privileges - (DA)
- Logical database design - (DA/DBA)
- Physical database design - (DBA)
- Planning database growth and change - (DA/DBA)

How a DA can manage data security. There is a *laundry list approach* to security where the data administrator simply reacts to external requirements and fulfills them one at a time as they arise. In this approach, the data administrator does not form mental models of how these items may relate to each other and simply considers them one at a time. Forming a mental model is irrelevant for attacks by *script kiddies*, a name used to refer to adversaries who make uncoordinated attacks on assets. On the other hand, considering the following tools piecemeal may make the system more vulnerable to purposeful adversaries.

```
- passwords
- authorization rules
- hardware firewalls
- encryption
- backups
- views
```

Later, when we discuss authentication and authorization, we'll discuss another approach to security that may be warranted if management agrees that coordinated effort is worthwhile.

DESIGN PATTERNS

Introduction. Design patterns can be difficult to understand and use without knowing something of their history and development. Therefore we first review the origin of design patterns in a separate discipline and then discuss their applicability to application development in general.

Design Pattern Definitions. Notice that the following definitions don't agree with each other. The notion of design patterns means different things to different people. The first definition is the most precise. The last definition is vague.

- A general reusable solution to a commonly occurring problem within a given context
- A description (or template) for how to solve a problem, it can be used in many different situations
- Patterns are formalized best practices

Design patterns in architecture. We review the origin of design patterns because it was here that software engineers first read about design patterns and realized that they could be applied to computer programming.

Design patterns were first observed. Design patterns began as an architectural concept in Alexander (1977). Alexander examined architecture from the standpoint of its value to a community of people in daily life. Alexander's ideas were largely ignored or rejected by architects but soon gained a cult following among computer scientists. Even-

tually his books became so popular outside architecture that they began to influence architecture.

A Pattern Language. A pattern language is a structured way to describe good design practices within a field of expertise.
Christopher Alexander discusses it in a Youtube video. The term was coined by Christopher Alexander and popularized by his book *A Pattern Language*. This book was followed by another book intended to explain the first book. Alexander has continued to try to explain the concept to this day.

Components of a Pattern Language.

- The Syntax: a description of where the solution fits into the larger design
- The Grammar: describes how the solution solves the problem
  - "Balconies and porches which are less than 6 feet deep are hardly ever used."

An example of a pattern is a place to wait.

- Problem: The process of waiting has inherent conflicts in it
- Solution: In places where people end up waiting (for a bus, for an appointment, for a plane), create a situation which makes the waiting positive

An example of a pattern is a useful cooking layout.

- Problem: Cooking is uncomfortable if the kitchen counter is too short and also if it is too long
- Solution: To strike the balance between the kitchen which is too small, and the kitchen which is too spread out, place the stove, sink, and food storage and counter in such a way that:

1. No two of the four are more than 10 feet apart.
2. The total length of the counter—excluding sink, stove, and refrigerator—is at least 12 feet.
3. No one section of the counter is less than 4 feet long.

Design patterns spread to software engineering. We review how design patterns spread to software engineering both to help us think about why we might use them and to help to differentiate between the writings on design patterns that you will encounter and have to somehow integrate into your own work.

Computer scientists popularized design patterns. The Gang of Four (commonly abbreviated GoF) were among computer scientists seeking a basis to make code less arcane, more scientific and, above all, reusable. One aspect of Alexander's description was so general that it seemed applicable to any field in which design plays a role. This key aspect was the notion of a *quality that could not be named* but that could be understood through experience—the quality shared by successful designs. Specific, non-obvious combinations of characteristics support the quality.

Gang of Four book. *Design Patterns*, Gamma et al. (1994), exploded on the software scene and propelled Alexander to greater fame at the same time as solidifying OO 's place in mainstream software development. The GoF argue that great writers use patterns, e.g., all of Shakespeare's plays were based on earlier, less successful plays or stories. The GoF refers to *tragically flawed hero* or *boy-meets-girl, boy-loses-girl* as patterns with infinite variety. The GoF book serves two purposes, to tell what patterns are and to catalog 23 well-known patterns.

Gang of Four pattern definition. A design pattern is a description of communicating objects and classes customized

to solve a general design problem in a particular context. (from the introduction to *Design Patterns*, 1994)

A pattern has four things.

1. Pattern name
2. Problem
3. Solution
4. Consequences

A pattern name is a tool. The pattern name must be good enough to become part of the design vocabulary. The pattern name must be useful in conversation, documentation, and thinking. The GoF spent a lot of its time on the names of the 23 patterns in their catalog.

A problem may be of several kinds.

- Basic design problems such as algorithm design
- Commonly occurring classes or object structures known to be problematic
- Lists of conditions that, when occurring together, create a generic problem

A solution is a description of objects and classes.

- Not a solution in a packaged sense
- Solution is abstract, not implementation-specific
- Description of elements of solution (objects and classes)
  - Relationships between elements
  - Responsibilities of elements
  - Collaborations between elements

A consequence is a result or trade-off.

- Application of a pattern may resolve conflicts of varied kinds, most often space and time

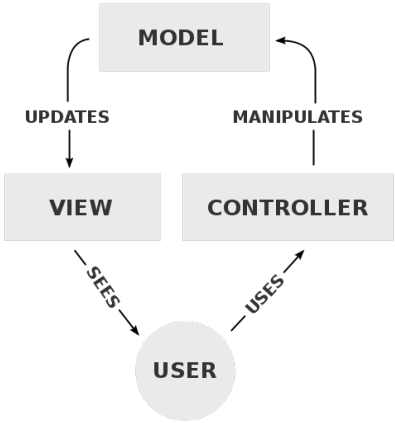


Figure 6. MVC-Process

- Evaluate the design decision with awareness of the consequences
- Consequences may have implementation issues, unlike the solution
  - if tempted to talk implementation, do it under this banner instead of under the solution banner
  - keep the solution a description, not an evaluation of itself

Review MVC. Examine diagram "MVC-Process" by RegisFrey - Own work. Licensed under Public Domain via Wikimedia Commons - MVC-Process

- MVC - Model View Controller
- Model - Business layer type object expressed in terms of problem domain
- Controller - manipulates the model and therefore the business rules that exist in the model
- View - Representation to user, updated by model, causes user to manipulate controls

The model, view, controller triad of classes.

- MVC, publicly introduced by Krasner and Pope (1988), can help you define the design pattern concept. Prior to the publication of the concept, it had been in active use in Smalltalk for perhaps twelve years.
- Three kinds of objects
  - Model ⇒ application object
  - View ⇒ presentation object
  - Controller ⇒ reaction to user input object
- MVC establishes a subscribe / notify protocol between views and models
- When model data changes, model notifies views depending on it
- When model data changes, relevant views have opportunity to update themselves

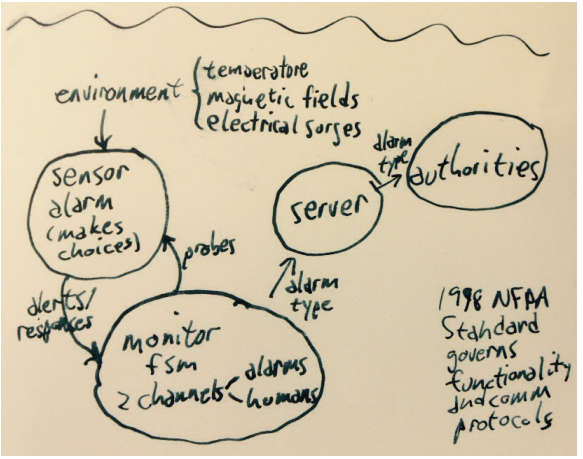
MVC and the observer pattern.

- GoF provides the following example
  - model is that y contains a=50, b=30, c=20
  - view1 is a table where row y contains cols a, b, and c
  - view2 is a barchart with bars for a, b, and c
  - view3 is a piechart with slices for a, b, and c
- More general idea is that model can update without knowing any details of views, pattern is called observer
- Observer: define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
  - alternatively subject / observer ; publisher / subscriber

Is there an observer pattern in the fire alarm example?.

- We discussed a student's experience with a design pattern course
- Student worked in fire alarm industry
- Design pattern presented in course seemed to contradict or omit practical knowledge
- Looking at the following diagram, think about whether the observer pattern is present and, if so, what it tells you about where to put the notifications and subscriptions

Fire alarm system sketch. based on conversation with an industry participant



Where patterns have gone.

- GoF were interviewed in 2009 about the relevance of their 1994 book in 15 years later
  - They deemphasize reusability as a task for *average* developers, instead positing reusability as a task for frameworks and toolkits and understood rather than implemented by *average* developers

- They emphasize patterns as targets for refactoring
- They found it easy to adapt to iOS SDK because of familiar patterns
- They prefer code smells to anti-patterns because some problems must be lived with
- They claim relevance to Python, Ruby, Smalltalk, but not to functional languages

Notes from GoF refactoring in 2005. Interpreter and Flyweight should be moved into a separate category that we referred to as "Other/Compound" since they really are different beasts than the other patterns. Factory Method would be generalized to Factory.

The categories are: Core, Creational, Peripheral and Other. The intent here is to emphasize the important patterns and to separate them from the less frequently used ones.

The new members are: Null Object, Type Object, Dependency Injection, and Extension Object/Interface—see "Extension Object" in Martin, Riehle, and Buschmann (1997)

More notes from GoF refactoring in 2005.

- These were the categories:
  - Core: Composite, Strategy, State, Command, Iterator, Proxy, Template Method, Facade
  - Creational: Factory, Prototype, Builder, Dependency Injection
  - Peripheral: Abstract Factory, Visitor, Decorator, Mediator, Type Object, Null Object, Extension Object
  - Other: Flyweight, Interpreter

Pattern Language and UI.

- Learning tool

- Creates a shared language
- Gallery of design solutions
- Typical pattern formula:
  - Name
  - Problem Summary
  - Usage – context of use
  - Solution
  - Why – rationale
  - Examples of implementation

UI Patterns. PatternTap provides what it calls examples of UI patterns. It may make sense to question whether these are design patterns in the way that Alexander and the GoF define patterns, or simply a loose, convenient, vague use of the term.

References.

LAYERED APPLICATIONS

You should have learned in a prerequisite course, ISTE-330, how to use object orientation to divide an application into layers and isolate connections to a database to the data layer. You likely had to include at least the following layers in a final project for the prerequisite course: Presentation, Application, Business, and Data. The Acute Otitis Media (AOM) entity relationship diagram, studied in that course, provides an example of a database application containing those layers. Only interact with the user in the presentation layer. Only interact with the database in the data layer. Divide remaining functionality between that which is specific to the domain of the application (business) and the more generic functionality common to an application in any domain. This makes it easier to write multiple applications using the same framework.

It also makes it faster for a front end / back end partnership to work together as they did in the AOM example and the radio station example also used in ISTE-330.

Layers and Tiers. In the remainder of this section, let us use the term *tiers* to refer to physical units of deployment. Let us use the term *layers* to refer to refer to logical separation of responsibilities in code. While it is redundant to speak of *physical tiers* in this context, let us use that expression as a reminder that we are referring to tiers in this physical sense. This is because *tiers* is sometimes used (not in this document) interchangeably with layers and sometimes it is used to mean logical rather than physical units of deployment.

Business Layer—Components and design issues. The remainder of the business layer notes are mostly from MSDN, with a little sarcasm and anti-lockin advice thrown in. Bear in mind that it is not in the interests of MSDN to promote any method or tool that favors a heterogeneous environment nor any organizational structure that disenfranchises MCSE certificate holders. There is no help to be found here against any dangers that grow in the soil of monoculture. There is no accounting for switching costs in the budget formed by these guidelines. Keeping those things in mind should enable you to avoid some lockin traps.

Overview. The business layer will usually include the following:

- Application façade
- Business Logic components
  - Business Workflow components
  - Business Entity components

Application façade. This optional component typically provides a simplified interface to the business logic components, often by combining multiple business operations into a

single operation that makes it easier to use the business logic. It reduces dependencies because external callers do not need to know details of the business components and the relationships between them.

Business Logic components. Business logic is defined as any application logic that is concerned with the retrieval, processing, transformation, and management of application data; application of business rules and policies; and ensuring data consistency and validity. To maximize reuse opportunities, business logic components should not contain any behavior or application logic that is specific to a use case or user story. Business logic components can be further subdivided into the following two categories:

Business Workflow components. After the UI components collect the required data from the user and pass it to the business layer, the application can use this data to perform a business process. Many business processes involve multiple steps that must be performed in the correct order, and may interact with each other through an orchestration. Business workflow components define and coordinate long running, multistep business processes, and can be implemented using business process management tools. They work with business process components that instantiate and perform operations on workflow components.

Business Entity components. Business entities, or business objects in general, encapsulate the business logic and data necessary to represent real world elements, such as Customers or Orders, within your application. They store data values and expose them through properties; contain and manage business data used by the application; and provide stateful programmatic access to the business data and related functionality. Business entities also validate the data contained within the entity and encapsulate business logic to ensure consistency

and to implement business rules and behavior.

General Design Considerations.

- Decide if you need a separate business layer
- Identify the responsibilities and consumers of your business layer.
- Do not mix different types of components in your business layer.
- Reduce round trips when accessing a remote business layer.
- Avoid tight coupling between layers.

General. When designing a business layer, the goal of the software architect is to minimize complexity by separating tasks into different areas of concern. For example, logic for processing business rules, business workflows, and business entities all represent different areas of concern. Within each area, the components you design should focus on the specific area, and should not include code related to other areas of concern. Consider the following guidelines when designing the business layer:

Decide if you need a separate business layer. It is always a good idea to use a separate business layer where possible to improve the maintainability of your application. The exception may be applications that have few or no business rules (other than data validation).

Identify the responsibilities and consumers of your business layer. This will help you to decide what tasks the business layer must accomplish, and how you will expose your business layer. Use a business layer for processing complex business rules, transforming data, applying policies, and for validation. If your business layer will be used by your presentation layer and by an external application, you may choose to expose your business layer through a service.

Do not mix different types of components in your business layer. Use a business layer to avoid mixing presentation and data access code with business logic code, to decouple business logic from presentation and data access logic, and to simplify testing of business functionality. Also, use a business layer to centralize common business logic functions and promote reuse.

Reduce round trips when accessing a remote business layer. If the business layer is on a separate physical tier from layers and clients with which it interacts, consider implementing a message-based remote application façade or service layer that combines fine-grained operations into a smaller number of coarse-grained operations. Consider using coarse-grained packages for data transported over the network, such as Data Transfer Objects (DTOs).

Avoid tight coupling between layers. Use the principles of abstraction to minimize coupling when creating an interface for the business layer. Techniques for abstraction include using public object interfaces, common interface definitions, abstract base classes, or messaging. For Web applications, consider a message-based interface between the presentation layer and the business layer.

Specific Design Issues. There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following sections provide guidelines for the common areas where mistakes are most often made:

- Authentication
- Authorization
- Caching
- Coupling and Cohesion
- Exception Management
- Logging, Auditing, and Instrumentation

- Validation

Authentication. Designing an effective authentication strategy for your business layer is important for the security and reliability of your application. Failure to do so can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attacks. Consider the following guidelines when designing an authentication strategy:

Avoid authentication if possible. Avoid authentication in the business layer if it will be used only by a presentation layer or by a service layer on the same tier within a trusted boundary. Flow the caller's identity to the business layer only if you must authenticate or authorize based on the original caller's ID.

See if single sign-on would work. If your business layer will be used in multiple applications, using separate user stores, consider implementing a single sign-on mechanism. Avoid designing custom authentication mechanisms; instead, make use of the built-in platform mechanisms whenever possible.

ACL specific choices. If the presentation and business layers are deployed on the same machine and you must access resources based on the original caller's access control list (ACL) permissions, consider using impersonation. If the presentation and business layers are deployed to separate machines and you must access resources based on the original caller's ACL permissions, consider using delegation. However, use delegation only when necessary due to the increased use of resources, and additionally, because many environments do not support it. If your security requirements allow, consider authenticating the user at the boundary and using the trusted subsystem approach for calls to lower layers. Alternatively, consider using a claims-based security approach (especially for service-based applications) that takes advantage of federated identity

mechanisms and allows target system to authenticate the user's claims.

Authorization. Designing an effective authorization strategy for your business layer is important for the security and reliability of your application. Failure to do so can leave your application vulnerable to information disclosure, data tampering, and elevation of privileges. Consider the following guidelines when designing an authorization strategy:

Use roles. Protect resources by applying authorization to callers based on their identity, account groups, roles, or other contextual information. For roles, consider minimizing the granularity of roles as far as possible to reduce the number of permission combinations required.

Keep using roles. Consider using role-based authorization for business decisions; resource-based authorization for system auditing; and claims-based authorization when you need to support federated authorization based on a mixture of information such as identity, role, permissions, rights, and other factors.

Avoid workarounds. Avoid using impersonation and delegation where possible because it can significantly affect performance and scaling opportunities. It is generally more expensive to impersonate a client on a call than to make the call directly. Do not mix authorization code and business processing code in the same components. As authorization is typically pervasive throughout the application, ensure that your authorization infrastructure does not impose any significant performance overhead.

Caching. Designing an appropriate caching strategy for your business layer is important for the performance and responsiveness of your application. Use caching to optimize reference data lookups, avoid network round trips, and avoid un-

necessary and duplicated processing. As part of your caching strategy, you must decide when and how to load the cache data. To avoid client delays, load the cache asynchronously or by using a batch process. Consider the following guidelines when designing a caching strategy:

Don't cache volatile data. Consider caching static data that will be reused regularly within the business layer, but avoid caching volatile data. Consider caching data that cannot be retrieved from the database quickly and efficiently, but avoid caching very large volumes of data that can slow down processing. Cache only the minimum required.

Cache pre-formatted data. Consider caching data in a ready to use format within your business layer.

Avoid caching sensitive data. Avoid caching sensitive data if possible, or design a mechanism to protect sensitive data in the cache.

Decide whether synchronization is needed. Consider how Web farm deployment will affect the design of your business layer caching solution. If any server in the farm can handle requests from the same client, your caching solution must support the synchronization of cached data.

Coupling and Cohesion. When designing components for your business layer, ensure that they are highly cohesive, and implement loose coupling between layers. This helps to improve the scalability of your application. Consider the following guidelines when designing for coupling and cohesion:

Circular dependencies. Avoid circular dependencies. The business layer should know only about the layer below (the data access layer), and not the layer above (the presentation layer or external applications that access the business layer directly).

Use abstraction. Use abstraction to implement a loosely coupled interface. This can be achieved with interface components, common interface definitions, or shared abstraction where concrete components depend on abstractions and not on other concrete components (the principle of Dependency Inversion).

Couple tightly inside layer. Design for tight coupling within the business layer unless dynamic behavior requires loose coupling.

Design for high cohesion. Design for high cohesion. Components should contain only functionality specifically related to that component. Always avoid mixing data access logic with business logic in your business components.

Use message-based interfaces. Consider using message-based interfaces to expose business components to reduce coupling and allow them to be located on separate physical tiers if required.

Exception Management. Design an effective exception management solution for your business layer to improve the security and reliability of your application. Failing to do so can leave your application vulnerable to Denial of Service (DoS) attacks, and may allow it to reveal sensitive and critical information about your application. Raising and handling exceptions is an expensive operation, so it is important that your exception management design takes into account the impact on performance. When designing an exception management strategy, consider following guidelines:

Only catch what you can handle. Only catch internal exceptions that you can handle, or if you need to add information. For example, catch data conversion exceptions that can occur when trying to convert null values. Do not use exceptions to control business logic or application flow.

Strategize about exception handling. Design an appropriate exception propagation strategy. For example, allow exceptions to bubble up to boundary layers where they can be logged and transformed as necessary before passing them to the next layer. Consider including a context identifier so that related exceptions can be associated across layers when performing root cause analysis of errors and faults.

Identify where exceptions may be caught. Ensure that you catch exceptions that will not be caught elsewhere (such as in a global error handler), and clean up resources and state after an exception occurs.

Find the holy grail. Design an appropriate logging and notification strategy for critical errors and exceptions that logs sufficient detail from exceptions and does not reveal sensitive information.

Logging, Auditing, and Instrumentation. Designing a good logging, auditing, and instrumentation solution for your business layer is important for the security and reliability of your application. Failing to do so can leave your application vulnerable to repudiation threats, where users deny their actions. Log files may also be required to prove wrongdoing in legal proceedings. Auditing is generally considered most authoritative if the log information is generated at the precise time of resource access, and by the same routine that accesses the resource. Instrumentation can be implemented using performance counters and events. System monitoring tools can use this instrumentation, or other access points, to provide administrators with information about the state, performance, and health of an application. Consider the following guidelines when designing a logging and instrumentation strategy:

Centralize logging and auditing. Centralize the logging, auditing, and instrumentation for your business layer. Consider

using a library such as patterns & practices Enterprise Library, or a third party solutions such as the Apache Logging Services to implement exception handling and logging features.

Instrument critical events. Include instrumentation for system critical and business critical events in your business components.

Avoid storing sensitive information. Do not store business sensitive information in the log files.

Here's a controversial suggestion. Ensure that a logging failure does not affect normal business layer functionality.

Controversial because an adversary would certainly try to exploit it.

Consider auditing and logging all access to functions within business layer.

Validation. Designing an effective validation solution for your business layer is important for the usability and reliability of your application. Failure to do so can leave your application open to data inconsistencies and business rule violations, and a poor user experience. In addition, it may leave your application vulnerable to security issues such as cross-site scripting attacks, SQL injection attacks, buffer overflows, and other types of input attacks. There is no comprehensive definition of what constitutes a valid input or malicious input. In addition, how your application uses input influences the risk of the exploit. Consider the following guidelines when designing a validation strategy:

Validate all input within business layer. Validate all input and method parameters within the business layer, even when input validation occurs in the presentation layer.

Centralize the approach. Centralize your validation approach to maximize testability and reuse.

Assume adversaries. Constrain, reject, and sanitize user input. In other words, assume that all user input is malicious. Validate input data for length, range, format, and type.

Deployment Considerations. In deploying a business layer, you must consider performance and security issues within the production environment. Consider the following guidelines when deploying a business layer. These guidelines can be grouped together under the following edict. Keep business and presentation on one tier for performance

- Consider deploying the business layer on the same physical tier as the presentation layer in order to maximize application performance, unless you must use a separate tier due to scalability or security concerns.
- If you must support a remote business layer, consider using the TCP protocol to improve application performance.
- Consider using Internet Protocol Security (IPSec) to protect data passed between physical tiers.
- Consider using Secure Sockets Layer (SSL) encryption to protect calls from business layer components to remote Web services.

Design Steps for the Business Layer. When designing a business layer, you must also take into account the design requirements for the main constituents of the layer, such as business components, business entities, and business workflow components. This section briefly explains the main activities involved in designing the business layer itself. Perform the following key steps when designing your data layer:

High level design. Create a high level design for your business layer. Identify the consumers of your business layer, such as the presentation layer, a service layer, or other applications.

This will help you to determine how to expose your business layer. Next, determine the security requirements for your business layer, and the validation requirements and validation strategy

Business component design. Design your business components. There are several types of business components you can use when designing and implementing an application. Examples of these components include business process components, utility components, and helper components. Aspects of your application design, transactional requirements, and processing rules affect the design you choose for your business components.

Business entity component design. Design your business entity components. Business entities are used to contain and manage business data used by an application. Business entities should provide validation of the data contained within the entity. In addition, business entities provide properties and operations used to access and initialize data contained within the entity.

Workflow component design. Design your workflow components. There are many scenarios where tasks must be completed in an ordered way based on the completion of specific steps, or coordinated through human interaction. These requirements can be mapped to key workflow scenarios. You must understand how requirements and rules affect your options for implementing workflow components.

Relevant Design Patterns. Key patterns are organized by key categories, as follows.

Business Components.

- Application Façade. Centralize and aggregate behavior to provide a uniform service layer.

- Chain of Responsibility. Avoid coupling the sender of a request to its receiver by allowing more than one object to handle the request.
- Command. Encapsulate request processing in a separate command object with a common execution interface.

Business Entities.

- Domain Model. A set of business objects that represents the entities in a domain and the relationships between them.
- Entity Translator. An object that transforms message data types to business types for requests, and reverses the transformation for responses.
- Table Module. A single component that handles the business logic for all rows in a database table or view.

Workflows.

- Data-Driven Workflow. A workflow containing tasks whose sequence is determined by the values of data in the workflow or the system.
- Human Workflow. A workflow that involves tasks performed manually by humans.
- Sequential Workflow. A workflow that contains tasks that follow a sequence, where one task is initiated after completion of the preceding task.
- State-Driven Workflow. A workflow with tasks whose sequence is determined by the state of the system.

Further on Application and Domain Distinctions. The prerequisite course 330 Database Connectivity now uses the AOM study schema. You may not have seen this in a previous iteration of 330, so I want to share it as an example of how a real database app distinguishes layers in the ERD. So

I'll put this on myCourses as AOMschema under resources. The user is at the top and the database is at the bottom of the picture.

EXCEPTION HANDLING

Exception handling fits well with our layered application emphasis and database-oriented project focus.

"Exception handling in a layered architecture" is something you can make a section of your document and demonstrate in all your projects. I have vacillated over whether it should be a separate section in your design document. What do you think of the choice between having an exception handling section as opposed to including exception handling in the description of each layer? Is your thinking affected by locus of responsibility? A frequent mistake students make is to insist that their current project members will be responsible for all aspects of the application. Typically, applications with any value to other people will be passed on to the responsibility of others—others whose expertise may not be packaged as that of your team members. Assumptions you make about the future are often disproved by practice. If you say that anyone who takes over the application will know x, y, and z, I can promise you that there is at least an even money chance that at least one of the three will have to be farmed out. There is at least a one-in-three chance that one of the three forms of expertise will become obsolete during the life of the application.

If your project is really so simple that one person will always handle all aspects, it may not appropriate for this course and you may need to revise it or face the likelihood that a high grade for the course is not available for a team that accomplishes a trivial result well.

Consider your design document so far, both from the standpoint of what is supposed to be in them already (distinguishing domain and application), as well as the layering of them in general. Layered architecture should dictate exception handling at the data layer.

Exceptions should be passed up the layers and handled before the user sees them. An adversary may try to provoke exceptions in the hope of seeing underlying aspects of a system. We want to avoid that. We also want to handle things where they "should" be handled, even if our project is so small that one or two people can understand the whole thing.

A complete exception handling treatment needs several things.

- test data
- test tool / harness / suite / protocol
- above could be called test data + some specific way to use it
- Style guide for input
- Style guide for what you return
- Picture of the layers that gives details about responsibilities of each layer
- You have to make a choice about passing or handling at each layer and make it clear what that choice is
- The point of the foregoing is that most layers are the responsibility of different developers or, at best, the same developer in two different project phases, during the second of which the details of the first will be forgotten so it may as well have been a different developer
- In larger organizations, may have many developers per layer
- Hence, it makes sense in your project to behave as if it were larger and to specify a bunch of stuff that you don't strictly need to do to meet your in-class deadline

Mistakes happen. Wikipedia reports the unfortunate fate of *Ariane 5*, an expensive satellite-launching rocket of the European Community. It only flew for 37 seconds before exploding, scattering 370 million USD to the wind. It turned out that, of seven critical variables, four were protected by exception handling. The rocket was brought down by an unhandled exception in the conversion of a 64 bit floating point number to a 16 bit int.

Exceptions as flow control. Wikipedia reports on exception handling in programming languages, dividing languages into two groups, those using exception handling as flow control structures, and others. The use of exception handling as a flow control structure leads to hidden control-flow paths. Some sources quoted by Wikipedia claim that these lead to software defects in languages reliant on these structures, such as Java and Python. Further, Java libraries treat error reporting inconsistently, not always via exceptions, compounding the problem.

The try-catch-finally structure used in Java and elsewhere permits many corner cases. A corner case exists when multiple rare conditions may occur at once. This is enabled in try-catch-finally because the Java language definition of that construct is a four level nested if structure. That means that we need to be cognizant of any combination of the four constituent conditions.

Checked exceptions add an exception to a method signature, making it harder to ignore. Anders Hejlsberg stated in an interview that he avoided checked exceptions in the design of C# specifically because they are resented by Java programmers in enterprise-sized systems. Some developers merely decorate every method with `throws Exception`.

It is hard to write meaningful exception handling because it is hard to know what to expect. For example, how many

ways may a given protocol be violated? One solution is to use automated exception handling but this is more prevalent in research than in practice. Although there are automated exception handling tools, they receive scant attention in the usual media outlets for software development.

Exceptions in JavaScript. The following summarizes much of the discussion of `throw` and `try` / `catch` / `finally` in Flanagan (2011).

An exception is a signal of an error or exceptional condition.

The `throw` statement is the JavaScript mechanism for sending this signal.

The throw statement has the syntax `throw` *expression*; where *expression* may evaluate to any type of value. For example, a number may represent an error code or a string may be a readable error message.

A JavaScript interpreter may throw and, when it does so, it throws an object of the Error class.

Each Error class object has a `name` property specifying the error type and a `message` property holding a string.

A JavaScript interpreter stops normal program execution immediately when it encounters a `throw` statement and jumps to the nearest exception handler.

If there is no exception handler in the enclosing block of the `throw` statement, the interpreter checks the next highest enclosing block until a handler is found or the call stack is exhausted. In the latter case, the error is reported to the program user.

JavaScript's exception handling mechanism is the `try` / `catch` / `finally` statement.

The `try` clause defines the block of code whose exceptions are to be handled.

The `catch` and `finally` clauses optionally follow the `try`

clause but at least one of them must appear.

All three of these clauses must begin and end with curly braces, even if they each only have a single statement.

The `catch` keyword is followed by an identifier in parentheses. For example, you might say `catch (err) {...}`. The identifer behaves like a function parameter but differs from ordinary JavaScript variables in a way. It takes on the value of the `throw` *expression*, such as an Error object. It has block scope within the `catch` block and is not visible outside the `catch` block. This differs from ordinary JavaScript variables in that they are visible throughout the function in which they are defined or globally if defined outside of any function.

The `finally` clause, if it exists, is guaranteed to be executed if any portion of the `try` block is executed. There are four possible cases.

(1) If the entire `try` block is executed, then the `finally` block is executed.

(2) If a `try` block is interrupted by `return`, `continue`, or `break`, the `finally` block is executed before the jump caused by one of those three statements.

(3) If an exception occurs in the `try` block and there is a `catch`, the `catch` is executed, followed by the `finally`.

(4) If an exception occurs in the `try` block and there is no local `catch`, the `finally` is executed and then the interpreter begins ascending through the call stack, looking for the nearest `catch`.

If a `finally` clause is interrupted by a jump of any kind, that jump replaces any jump previously planned. In other words, if the `finally` clause is entered preparatory to a jump,

that jump is canceled by any jump occurring in the `finally` clause.

AUTHENTICATION AND AUTHORIZATION

Outline.

- introduction
  - data qualities
  - data protection categories
  - data states
  - decoupling people from data
- authentication
  - authentication techniques: passwords, tokens, biometrics, multifactor
  - issues
  - os-level authentication
  - authentication management
- authorization
  - authorization methods
  - access control lists

Three qualities of data to protect. These three qualities can be easily remembered by the acronym CIA.

- confidentiality of data
- integrity of data
- availability of data

Three ways to protect data.

- human factors
- policy and practices
- technology

Three states where data is vulnerable.

- transmission
- storage
- processing

McCumber cube.

- The foregoing three lists of three each are collectively known as the McCumber Cube.
- The McCumber Cube was an early systematic way to organize thinking about security.
- If you thought exhaustively about each of the nine areas and the $3^3 = 27$ combinations of them, it was believed that you had a chance of thinking comprehensively about security—leaving no stone unturned.
- Security professionals regard the McCumber Cube as passé today (see for example the McCumber Cube comedy video on Youtube), but most contemporary security frameworks strike me as still using mainly these ideas and adding a few tiny refinements.

Simplest approach to security.

- Users are put into groups
- Groups are assigned permissions (CRUD) for tables, views, forms, reports
- Users are assigned special permissions which differ from their group

Reasons to uncouple people from data.

- Direct links between people and data may leave data orphaned due to staff turnover and gaps in hiring
- Human resources choices should not be manipulated by data-base specialists

  - Enterprises frequently rewrite responsibilities for individuals even maintaining the same job, forcing a data-base specialist to dabble in human resources issues while trying to figure out which privileges to grant

- Managers ask for an individual to be given the same privileges as another individual known to use the same data, but without realizing that the other individual also accessed other data

Work, Resources, Privileges, Roles, Membership.
In brief.

- A common way to plan access is to use these five concepts
- Resources are data resources: tables, views required for work
- Work tasks can be defined and compartmentalized into the resources required
- Roles for staff are specified as assigned to pieces of work
- Roles are accorded privileges according to assigned work
- Individuals are granted membership in roles pertinent to their assignments

In more detail. A common way to plan access is to use these five concepts. Each of them may change over time so we can improve the use of resources by defining each one of these and linking in a way that allows a single link to change.

Work consists of tasks that can be defined and linked to the resources required to do that work. These tasks may change over time and the resources required to do given tasks may change over time. If an organization is advanced in a system

like Capability Maturity Metrics, it is defining work in ways that may be used to improve management and reporting.

Resources are data resources: tables, or views onto tables that are required for work. The resources may change in different ways. As an example, I worked on a project for working with a database of document images. The documents had been pouring into the system for many years and new technologies had emerged for identifying sensitive data in the documents, leading to a desire to restructure the database, which was used by thousands of clients. By uncoupling the views from the database structure it was possible to update the database to take advantage of entities that had not been contemplated in an earlier epoch. Then the question to answer was which new views would apply to different people using the database. The answer, in part, was to define the work or use existing definitions of work to organize the views and their availability. The key problem was that, with thousands of users, the user organizations could not possibly have the same maturity level. Some could offer definitions of work but some could not.

Roles are accorded privileges according to assigned work. Job descriptions are rewritten frequently in almost every workplace I have ever encountered. Rewriting job descriptions is sometimes a way to reward loyalty or accomplishment where promotion is not possible. Rewriting job descriptions may be required to handle new responsibilities assigned to an organization. It may even be done punitively. Roles are often (very often) left unfilled for weeks or months due to do employee turnover or promotion. Details of job responsibilities are often lost. There is a whole field of research into *organizational memory* that tries to cope with problems related to turnover. Roles and their definition is one way to do so.

Roles for staff are specified instead of staff being simply

assigned to pieces of work. It is the role that dictates the work an individual must accomplish. Individuals may be granted membership in multiple roles, serially or in parallel.

Authentication Techniques.

- Passwords
- Tokens
- Biometrics
- Each technique has problems
- Multifactor

Passwords.

- Major goal of security professionals is to eliminate passwords … Why?
- Social Engineering of Passwords
- Advances in Cryptography
- Increasing Multi-tenanted Cloud Installations

Password practice draws constant complaints but has evolved little through history. The main evolutionary development has been to allow users to choose their own passwords, something that was not possible for early generations of computer users.

Study after study has demonstrated the insecurity of passwords as they are practiced. Books by experts like Kevin Mitnick demonstrate the ease with which social engineering can be used to extract or bypass passwords.

Fifty-six bit secret study. A study presented at Usenix 2014 by Bonneau and Schechter demonstrated a superior password regime where participants chose a password but then were taught to supplement it with a 56-bit secret over the course of one to two weeks. Some relevant aspects of the study follow.

Background. The study took advantage of spaced repetition to learn a 56-bit secret to be added to a user-selected password. Spaced repetition is a learning approach that has been successfully employed by many, including most medical students over the past century.

Cost. The study presented a method for evaluating the cost of cracking a 56-bit secret using bitcoin mining and determined an average cost of 1m USD at January 2014 rates. The method itself was presented so that the number could be updated to account for economic fluctuations. The number supported segmentation of adversaries into categories of individuals, corporate entities, and nation-states. Clearly, the latter two categories would not be deterred by a million dollar price tag. So the authors can make a convincing claim that their password regime impedes determined individual adversaries.

Effort. The study examined the effort required by average users to practice the regime and found that it added about twelve minutes over the course of learning the 56-bit secret. Each login during the learning period took an average of 6.9 seconds extra for learning the secret. The secret consisted of 676 common words, combined and arranged in a manner chosen by the researchers to create a genuinely random 56-bit string. The use of the words is similar to the horse battery staple concept presented in a famous xkcd cartoon by Randall Munroe, but accomplished in a more precisely defined manner so that the adversary is definitely presented with 56 randomly ordered bits.

Validation. The study was conducted using a distraction task, a popular method to improve the generalizability of results. A distraction task is a task the experimental participant

believes to be central to the experiment. It distracts the participant from the real task, in this case, learning the secret. This validation approach nullifies the criticism that the participants were trying really hard and would relax in real life. In fact, the participants were trying really hard to do something else and were unaware of the significance of learning the secret.

Ethical Concerns. Any research where participants are deceived presents an ethical risk. In 1979 a document called the Belmont Report presented basic ethical principles that underly all subsequent US regulations of human subject research. The three principles are named (1) respect for persons, (2) beneficence, and (3) justice. While these three names sound very generic, they are defined in great detail in the report, in subsequent legislation, in many books on ethics, and in plenty of free online tutorials. The study publication explains how the researchers complied with these principles.

Physical Tokens.

- Token provides a separate item you must have to authenticate
- It is separate from any other authentication factor
- Physical token has physical problems
- Smart tokens (on phone tokens) eliminate a main reason for token use, yet are immensely popular with users

Biometrics.

- Long history of failure except in controlled settings
- Now available in all flagship smartphones

Multifactor.

- Something you are, something you know, something you have

- Regarded as badge of lower class by users unable to resist implementation

The most robust authentication includes something you know (password), something you have (dongle), and something you are (fingerprint). If these are all three stored in one place, they do not constitute three factor authentication. For example, a smartphone that records your fingerprint, acts as a dongle, and stores all your passwords may, if stolen, provide an adversary with complete access. This depends in part on the quality of the implementations. For example, Lenovo laptops bind fingerprints to passwords so the user does not have to type passwords.

OS-Level Authentication.

- Database authentication
- Username/Password

Authentication Management.

- Password Aging - require a change of password, can backfire
- Password Recovery / Reset - Email, multi-step process
- Limited Attempts - 3 times max, etc..
- Encryption Procedures - ZIP, AES 256bit, etc..

Authorization Methods.

- Role-Based - What are their credentials?
- User-Based - Username
- Database Level - have permissions in database - very tedious
- Impersonation - ASP.NET type method, faking who you are

ACLs, Access Control List / permissions attached.

- User - username
- IP - IP address
- Hardware - MAC Address

Authentication and Authorization.

- Authentication identifies you
- Authorization, well, authorizes you to do stuff
- Authentication may be performed by
  - Application
  - System
  - Trusted third party

Authorization. Authorization can be determined in different contexts

- Database
  - Use of GRANT statement
  - DBA may assign access at database level
  - Can be difficult to manage
- Application
  - DBA may trust developer to manage
  - Application has one or more identities with varying (admin, user) database privileges
  - Developer (via business rules) may assign access at application level
  - SQL WHERE clause

Application-Level Authorization. Varied means of accomplishing authorization

- Application-level
  - Username, password provided to application
  - All users interact with application in same way
  - This approach is frowned upon (effectively sharing a password)
- User-level
  - Username, password provided to user
  - More complex but more common

User privileges. under application-level authorization

- Task-Based
- Very granular
- User is assigned (or not) permission to each and every task
  - May include read/write/create distinction
- Difficult to maintain
  - Can create features to make it easier

Role-based authorization. under application-level authorization

- Username, password provided to user
- Each user is assigned a role which in turn has certain privileges associated with it
- Typical roles:
  - Admin
  - Editor
  - General User
  - Public

Code considerations. under application-level authorization

- Must ask the question: *Is the user authorized to perform this task?*
- Who is the user?
- What are their rights?
- Can they do this task?
- What if they can't?
- How did they get this far? (implies state)
- Why did they get this far? (implies state)

Business Rules.

- Who has access
- What can they access
- When does their access expire
- Where can they be when they access

Checking Authorization. Either following method may be encapsulated in a security object.

- Method 1
  - Determine user
  - Query database for user access rights
  - Check rights against requested action
  - Problematic in a RESTful application
- Method 2
  - Issue user token upon login
  - Check token for user access rights
  - Check rights against requested action
  - Token becomes part of every transaction

REST. Above, the expression *RESTful application* refers to an application designed with a cognizance of the way TCP/IP (hereafter the Internet) operates. Representational state transfer or REST is one of the core concepts of the Internet and a key reason for its success. It means that a node in the network does not maintain state information—state information must be transferred in every transaction. This makes the Internet vastly more robust than corporate computer systems but creates a dilemma in authorization.

Authentication Tokens.

- What is a token?
- Hash (illustrated guide to hashes)
  - A one-way function, meaning that it is easy to compute on input but hard to invert (one-way function)
  - Potentially secure
  - May use salt, random data used as an additional input (salt)
- Encryption
  - Two-way Access
  - Embedded info
  - Less secure, but more useful?

Token example. Steve's method to generate a token includes a userid, an ip address, and a timestamp. Steve performs a one-way function on these inputs. The userid starts as four digits, left padded for userids less than a thousand. The ip address starts as four octets, each one 0 to 127 (although many of the possible numbers are reserved, so this is a much smaller set than implied by 128^2). Step 1 is to convert each of these integers to a different base in some random way. Step 2 is to salt

each number. Step 3 is to somehow rearrange the characters. Step 4 is to do a cyclical redundancy check (CRC) and add the remainder to the result as a position-dependent checksum.

The Luhn algorithm is a checksum approach used by credit-card companies. More sophisticated algorithms mentioned in the above article (Verhoeff, Damm) can identify more transcription errors than can the Luhn algorithm.

Credential Storage. Credentials for a user may be stored in the database. Credentials may be stored in a secured table. Credentials may be encrypted.

Credential storage in the client. Storing credentials in the client depends on the circumstances. Does the client connect to server? Is the client a standalone application?

SOA Considerations. Where in a layered architecture are authentication & authorization determined? Is one token passed through or are multiple tokens used? What about multiple business layers? Any medium or large organization has many business layers, many data layers, and very many databases. Above the business layers are very many presentation layers. The path of tokens may be long and complicated.

PCI DSS Compliance. The Payment Card Industry Data Security Standard (PCI DSS) specifies how entities who accept credit cards must protect their systems. This is worthwhile guidance to know about, in part because it is such a common requirement but also because the standard is very technical and developers must often be closely involved in its implementation.

Let's look at just one of the requirements of the PCI DSS, requirement number six. This requirement specifies that you must *develop and maintain secure systems and applications*. There are seven specific requirements within requirement six. These are as follows.

6.1 Establish a process to identify security vulnerabilities,

by using reputable outside sources for security vulnerability information, and assign a risk ranking (for example, as 'high,' 'medium,' or 'low') to newly discovered security vulnerabilities.

6.2 Ensure that all system components and software are protected from known vulnerabilities by installing applicable vendor-supplied security patches. Install critical security patches within one month of release.

6.3 Develop internal and external software applications (including web-based administrative access to applications) securely, as follows:

- In accordance with PCI DSS (for example, secure authentication and logging)
- Based on industry standards and best practices
- Incorporating information security throughout the software-development lifecycle

6.4 Examine policies and procedures to verify that the following are defined.

- Development/test environments are separate from production environments with access control in place to enforce separation.
- A separation of duties between personnel that are assigned to the development/test environments and those persons assigned to the production environment.
- Production data (live PANs) is not used for testing or development.
- Test data and accounts are removed before a production system becomes active.
- Change control procedures that are related to implementing security patches and software modifications are documented.

6.5 Address common coding vulnerabilities in software devel-opment processes as follows.

- Train developers in secure coding techniques, including how to avoid common coding vulnerabilities, and understanding how sensitive data is handled in memory.

6.6 For public-facing web applications, address new threats and vulnerabilities on an ongoing basis and ensure these applications are protected against known attacks by either of the following methods.

- Reviewing public-facing web applications via manual or automated application vulnerability security assessment
  tools or methods, at least annually and after any changes.
- Installing an automated technical solution that detects and prevents web-based attacks (for example, a web-application firewall) in front of public-facing web applications, to continually check all traffic.

6.7 Ensure that security policies and operational procedures for developing and maintaining secure systems and applications are documented, in use, and known to all affected parties.

PERFORMANCE AND REFACTORING

How performance can be improved. One way to think about performance improvement is to think about the functional areas affecting database performance. These areas include database connections, transactions, sql statements, and data retrieval.

Another way to think about performance improvement is to categorize the ways to improve performance. Every performance improvement can be classified as reducing network traffic, limiting disk i/o, optimizing interaction between the driver and application, or improving queries.

Database connections. Database connections may be created one at a time or pooled for reuse. Connections are expensive, requiring memory and multiple round trips to the database server to establish. Opening numerous connections takes a significant amount of memory and time. Reusing connections may save significant time and memory if numerous connections materialize. Managing connections requires establishing connection pools, an arcane activity requiring either expertise or the use of third party connection management tools.

Connection pools. A connection pool is a cache of physical database connections available for reuse by one or more applications. A connection pool can help in a situation where an application on a server has multiple users. The database server must have adequate memory and, possibly, licenses to support the maximum connections specified by the connection pool. Connections should be opened just before use, then closed immediately after use and returned to the connection pool.

Connection pools are not indicated in the case of applications without an application server, applications that restart frequently each day, single-user applications like report writers, or periodic jobs such as logging and reporting. Background jobs run at non-peak times typically do not benefit from connection pools.

Ideas about connection management. Many SQL Server blogs endorse third party connection management tools (e.g., Idera) or consulting services due to the complexity

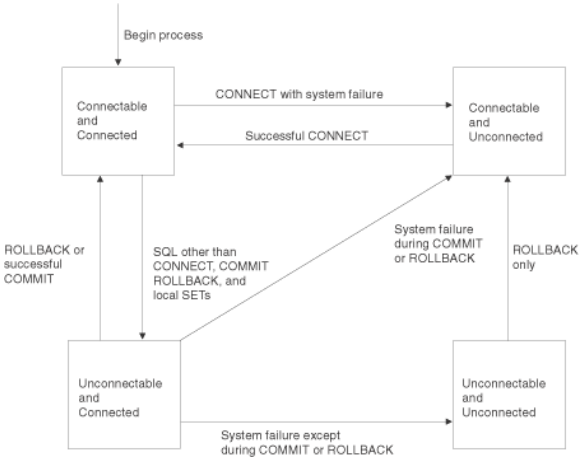


Figure 7. Four Possible Connection States

of setting connection pooling parameters.

Csharp connection management reports an argument where one side claims that db connections should be opened and closed by the same method! The other claims that this should be analyzed for performance gains.

Ensure Proper SQL ... suggests profiling application servers and offers a diagnostic SQL statement the author uses to begin investigating the possibility of connection pooling issues

DB2 Remote Connection Mgt posits four possible states and points out that a connect should be preceded by a commit or rollback and will fail if preceded by any sql statement other than connect, commit, release, rollback or set connection.

DBMS-specific connection management issues. More traffic may be found online about SQL Server connection management than for other dbms. Does that reflect popularity? Or could there be a more complicated dynamic at work. There are many measures of popularity. Job listings are one example.

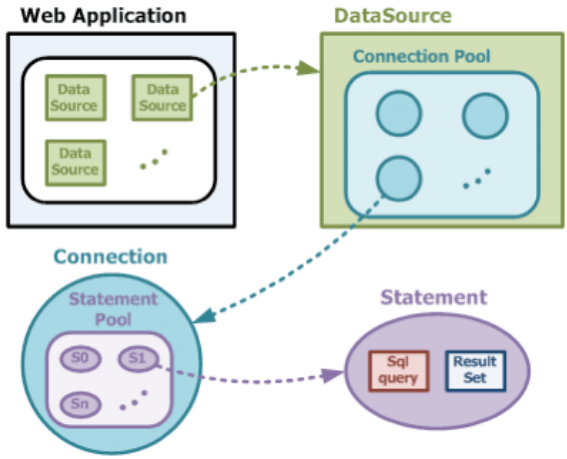


FIGURE 8. CONNECTION STATEMENT RELATIONSHIP

Statement pooling means statement caching. A given DBA may use either expression, statement pooling or caching.

Statement pooling in JDBC by Dongsoon Choi, argues that caching repeatedly used SQL statements improves application performance.

The improvement arises because of the design of garbage collection in Java. Garbage collection in Java is predicated on the notion that most objects quickly become unreachable and a reference from an old object to a new object is rare. The author's firm requires 300ms response time for web services, so garbage collection is too aggressive.

Each application has a connection pool. Each connection has a statement pool. Each statement has a query and a (not cached) result set.

Statement pooling may not be optional.

- Apache MyBatis exemplifies a persistence framework to automate mapping between SQL statements and meth-

ods in Java (its ancestor, iBatis, also worked with .NET and RoR).
- Statement pooling is not optional in an environment using such a framework.
- All statements are passed as prepared statements.
- Developer controls a setting like `MaxStatements` to express the number of SQL statements
- Because of the relationship in the previous diagram, the number of statements, not their size, matters.

Proper size of MaxStatements.

- Large numbers of SQL statements requires setting a larger value `MaxStatements` (or similar parameter)
- Default in jdbc is 500, likely good for 250KB of queries.
- Choi suggests that 250MB is needed for 10,000 queries and 50 connections, for example. (500 bytes per SQL × 10,000 SQL statements × 50 connections)

SQL Statements. Consider the following source on improving pagination performance in MySQL SQL I Love.

This blog post contrasts three ways of paging results from a 100,000,000 record MySQL database. The first attempt is a complete failure with MySQL failing to return any rows given a `select` statement that attempts to fetch all 100,000,000 rows.

The second attempt uses the `limit` clause of the `select` statement. The `limit` clause can be used with either one or two arguments. If two arguments are given, the first argument is the offset (0 to start with the first row) and the second argument is the "page size" which means the number of rows returned each time the query is issued.

This second attempt has two problems. First, it becomes slower and slower as the offset grows. In the example, the last

page takes well over a minute to return. This is because the previous rows are being fetched to find the offset.

The second problem with the second attempt is easiest to see if you run the query in a loop in a program to fetch all rows, a page at a time. In this scenario, another process could delete some records between fetches, so that some records are skipped. For example, suppose the page size is 100 and records 98 and 99 are deleted between the request for the first page and the second page. The 100th and 101st records will be skipped by the second request because they are now part of the first page, although they weren't when the first page was requested.

The third attempt satisfies the blog poster but not all of the commenters to the blog. That solution is to use the `user_id` field to paginate and only use the `limit` clause with a single argument, so that it simply gives a page size rather than an offset and a page size. This turns out to be a thousand times faster for rows farther along in the database than the second attempt because unwanted rows are not being fetched.

Perhaps the most valuable part of the blog post is the use of `explain extended` to analyze the difference between their execution plans. The third attempt has a constant number of rows analyzed, half the number of rows in the table, while the second attempt has analyzes a number of rows based on the size of the offset.

So, is this the right answer for pagination? Many of the commenters on the post disagree. Further, they point to a lot of more sophisticated posts about keyset pagination. For example, we need tool support for keyset pagination. This post suggests tracking the `last_seen_id` and using that to paginate. (There appears to be an error on that post where the poster uses a less than sign instead of a greater than sign with `last_seen_id` but the idea is pretty clear.)

SQL Tuning. The preceding example suggests that it may be worthwhile to study SQL performance from more than one source. Tow (2003) provides a complete book on SQL tuning and posits three basic steps to the process:

1. Figure out which execution plan (the path to reach the data your SQL statemet demands) you are getting.
2. Change SQL or the database to get a chosen execution plan.
3. Figure out which eecution plan is best.

Tow lists these steps in the above (illogical) order on purpose, claiming it reflects common practice. In its place, he offers the following list of questions to answer:

1. Which execution plan is best, and how can you find it without trial and error?
2. How does the current execution plan differ from teh ideal execution plan, if it does?
3. If the difference between the actual and ideal execution plans is enough to matter, how can you change some combination of the SQL and the database to get close enough to the ideal execution plan for the performance that you need?
4. Does the new execution plan deliver the SQL performance you need and expect?

Tow goes on to claim that the first question in this list is the most important and devotes much of the book to answering it.

Database performance resources. Database performance in general is the subject of the Persona db performance blog.

Dice ads for the following search strings.

```
13,216 Oracle
12,745 SQL Server
 3,806 MySQL
 1,920 DB2
 1,185 MongoDB
   802 Sybase
   730 PostgreSQL
```

Note that the above list is old. My impression is that more recently SQL Server has eaten Oracle. Also note that PostgreSQL is by far the most frequently mentioned on Hacker News. That is an example of how different communities have different concerns. People on Hacker News are a small and very specific community. Dice represents all kinds of businesses in different communities.

Obtaining machine information. An individual project planning to include a section on performance needs to be able to obtain system information about the target system to figure out performance parameters. For example, one project in Fall 2014 contemplated using Gibson, a host at RIT available to students, as an application server and database server. You can use `ssh` to connect to this system and issue some commands to determine some of its characteristics as follows.

Kelvin. Kelvin is a server students may use instead of Gibson.

```
[mjmics@kelvin ~]$ uname -a
    Linux kelvin.ist.rit.edu 2.6.32-504.3.3.el6.x86_64 #1 SMP
    Wed Dec 17 01:55:02 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux

[mjmics@kelvin ~]$ df -h
    df: `/root/.gvfs': Permission denied
    Filesystem            Size  Used Avail Use% Mounted on
    /dev/mapper/vg_kelvin-LogVol00
```

```
                          32G  6.3G   24G  21% /
    tmpfs                 7.8G   80K  7.8G   1% /dev/shm
    /dev/sda1             976M  146M  780M  16% /boot
    /dev/mapper/vg_kelvin-LogVol02
                          818G   23G  754G   3% /home
    /dev/mapper/vg_kelvin-LogVol01
                          126G   14G  106G  12% /var

[mjmics@kelvin ~]$ whoami
    mjmics

[mjmics@kelvin ~]$ mysql --version
    mysql  Ver 14.14 Distrib 5.1.73, for redhat-linux-gnu (x86_64)
           using readline 5.1

[mjmics@kelvin ~]$ psql --version
    psql (PostgreSQL) 8.4.20
    contains support for command-line editing

[mjmics@kelvin ~]$ mongo --version
    MongoDB shell version: 2.4.14

[mjmics@kelvin ~]$ du -h
    1.9G    ./Sites
    4.0K    ./.mozilla/extensions
    4.0K    ./.mozilla/plugins
    12K     ./.mozilla
    8.0K    ./.gnome2/keyrings
    12K     ./.gnome2
    1.9G    .

[mjmics@kelvin etc]$ cat /etc/centos-release
    CentOS release 6.7 (Final)

[mjmics@kelvin etc]$ cat /etc/filesystems
    ext4
    ext3
    ext2
    nodev proc
    nodev devpts
    iso9660
    vfat
    hfs
    hfsplus

[mjmics@kelvin etc]$ head /etc/init.d/httpd
    #!/bin/bash
    #
    # httpd        Startup script for the Apache HTTP Server
    #
```

```
# chkconfig: - 85 15
# description: The Apache HTTP Server is an efficient and extensible
#              server implementing the current HTTP standards.
# processname: httpd
# config: /etc/httpd/conf/httpd.conf
# config: /etc/sysconfig/httpd
```

Gibson. (Reports differ about whether the following information is obsolete. Gibson was taken out of service and returned to service. At some point that cycle will end.)

- RIT has a web server named Gibson that can be your application server
- saying `uname -a` reveals it is running SunOS 5.10 on an i86pc
- Googling reveals that 5.10 is equivalent to Solaris 10, but that SunOS is a subset of Solaris—just the operating system
- Oracle documentation says
  - to use `showrev` with various options
  - `prtconf` claims that 1GB of memory is installed (??)
  - `prtconf -x` exits silently, indicating 64-bit ready firmware
  - `psrinfo -pv` indicates that gibson has an AMD Opteron 6274 CPU running at 2.2Ghz
- `df -h` indicates 2GB swap space. The other numbers reported by `df` are hard to interpret. My guess is that there is a physical 300GB disk with 5 partitions of 50GB each for user data. Each of these 5 partitions is at about 25 percent of capacity. The fact that their usage levels are so similar may mean an appliance of some sort is involved but I have no further info. Suffice it to say that every user seems to be on a partition with about 33GB free.

- `mysql --version` indicates 5.1.62

Note that, for many versions of Linux, distribution info can be found by saying `lsb_release -a` at a terminal prompt. More information about `lsb_release` is available on many distributions by saying `man lsb_release` at a terminal prompt, or by googling *Linux Standard Base*, a joint project of several Linux distributions to improve compatibility between them.

To summarize, you should include a few sentences about system parameters in your design document based on information you get from using an approach like this.

Discuss performance in your project. Some miscellaneous examples of performance issues in past projects include the following hodgepodge. If you use PHP, for instance, you should say how it works on the server. Servers like Gibson may have to cater to multiple projects and multiple PHP major versions. If you have a file within conditional—say whether it will always include or only when you reach a condition. If response time is an issue, get response time. For example, you can use `curl`. You may use chrome devtools in the network panel find out time to load. Maybe use firebug. Maybe use multiple vms to avoid caching problem. If using Csharp, may describe background worker like threading. Don't just worry about size but also number of statements Check into persistence framework / caching of sql statements. Server-side pagination? In mysql limit to first n results.

Other performance tools. You should consider infrastructures tools for performance improvement. Following are some examples.

Varnish is an http accelerator for content-heavy dynamic web sites.

Squid is caching and forwarding web proxy.

Nginx is a high-performance web server with some additional capabilities, such as load-balancing and http caching.

Memcached is a general-purpose distributed memory caching system.

Javascript minification is the process of removing unnecessary characters from Javascript to minimize data transfer. Tools like UglifyJS can handle this.

Design patterns like lazy loading may be thought of as performance tools. You can lazy load below-the-fold content to speed up page rendering for instance.

Refactoring. Refactoring is the act of improving code without changing its external behavior.

Martin Fowler's *Refactoring, Improving the Design of Existing Code* is the definitive reference.

Extract a method from a code fragment. Suppose you have a code fragment that makes sense grouped together.

```
printStuff() {
   printBanner();
// print details
   System.out.println("Name: "   + this.name);
   System.out.println("Amount: " + this.amount);
}
```

may be refactored as

```
printStuff() {
   printBanner();
   printDetails();
```

```
}
printDetails() {
   System.out.println("Name: "   + this.name);
   System.out.println("Amount: " + this.amount);
}
```

Explain the meaning of a construct. Simplify a complicated fragment so that a casual reader may immediately comprehend the intention.

```
if ( (platform.toUpperCase().indexOf("MAC") > -1 ) &&
     (platform.toUpperCase().indexOf("IE")  > -1 ) &&
                                    wasInitialized() &&
                                        (resize > 0) ) {
   //do something only under this set of conditions
}
```

may be refactored as

```
final boolean isMac
     = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIE
     = platform.toUpperCase().indexOf("IE")  > -1;
final boolean wasResized
     = (resize > 0);
if (isMac && isIE && wasInitialized() && wasResized) {
   //do something only under this set of conditions
}
```

Simplify a compound condition. The author of this example may have left the rest of the exercise to the reader. It might be worthwhile to add that this refactoring requires some additional methods. It may be the case that, if we wrote those out, we might write them differently. See what you think.

```
if (date.before(SUMMER_START)||date.after(SUMMER_END))
 charge=quantity*this.winterRate+this.winterServiceCharge
else
 charge= quantity * this.summerRate;
```

may be refactored as

```
if (isSummer(date))
    charge = calcSummerCharge(quantity);
else
    charge = calcWinterCharge(quantity);
```

Might it make more sense to write a method to replace the if and the two methods posited above? Or would that contradict some of the examples below? Would it turn out that very little of the code depended on the parameters? It depends on what else is going on in the code. Given the little we see here, there are two rates, one conditional service charge, and two dates defining the two seasons. If that is everything, I would group the rates and service charge into one method, called by `calcCharge(date,quantity)` and the dates and season calculation into another method called by `getSeason(date)` from within the other method.

Replace error code with exception. Note. This is not strictly an example of refactoring because it does change behavior. It may be that the author of this example was thinking about the idea that neither the code nor the exception are seen 'externally'. It certainly makes sense to avoid passing error codes.

```
int withdraw(int amount) {
   if (amount > this.balance)
  return -1;
   else
       this.balance -= amount;
       return 0;
}
```

may be 'refactored' as

```
void withdraw(int amount) throws BalanceException {
   if (amount > this.balance) throw new BalanceException()
   this.balance -= amount;
}
```

Pull up a field or method. Suppose that two subclasses have the same field or method. Move the field or method to the parent class. Use this approach if the field or method exists in all subclasses. The field or method may have different names so ensure that the meaning is the same in each case.

Push down a field or method. Suppose the behavior of a parent method or field is relevant for only some of the subclasses. Move the field or method to the appropriate subclasses

Extract a superclass. Suppose that two classes have similar features. Create a superclass and move common features to the superclass. This approach may lead to using the pull up refactoring approaches mentioned above.

Extract a subclass. A class has features that are used only in some instances. Create a subclass for that set of features. This approach may lead to using the push down refactoring approaches mentioned above.

Add parameters. Suppose that two methods perform similar operations. Create a parameter-driven method that accomplishes the tasks of both methods. The goal is to abstract the operation and drive it with parameters rather than simply using if statements.

```
class Employee {
  void tenPercentRaise() {
    this.salary *= 1.1;
  }
  void fivePercentRaise() {
    this.salary *= 1.05;
  }
}
```

may be refactored as

```
  void raise (double factor) {
    this.salary *= (1+factor);
  }
```

Testing software. A potentially catastrophic misunderstanding between generalist managers and software developers often arises during the testing of software. Developers frequently complain that managers do not understand the tradeoffs involved in software testing and that managers expect testing to be inexpensive and to identify all problems that will arise in future. Managers, for their part, are often shocked to find that software has any defects at all and believe that developers should fix them for free.

Combinatorial explosion. This misunderstanding arises primarily from the problem that software is complicated and that testing all possible combinations of conditions in which software may find itself would take an infinite amount of time, a period even longer than it takes to get into an exclusive nightclub. Therefore, managers and developers have to work together to identify a small sample of likely conditions with significant consequences to test.

The following is from the book *Perfect Software* by Gerald Weinberg. Weinberg has written books on computing that can help general managers understand the issues well enough to work with specialists without having to become specialists themselves. This excerpt is from Chapter 3 *Why Not Just Test Everything?*

Notes from the trenches. Gerald Weinberg wrote a book on software testing. The book collects extensive field experience. The following is an excerpt from that book.

BEGIN EXCERPT

Testing may convincingly demonstrate the presence of bugs, but can never demonstrate their absence.

— Edsger W. Dijkstra (1974)

. . .

This is a special case of what philosopher John Locke called *argument from ignorance*, often phrased as *absence of evidence is not evidence of absence*. Note that this phrase is often abused. Logicians cringe at its use because there are many cases where absence of evidence is actually evidence of absence. Dijkstra is describing a particular case that meets certain criteria, i.e., that the actual inputs to a program can not be exhaustively tested and that the behavior of a program can not be known with certainty by subjecting it to a fraction of such inputs. Note that the word *inputs* here is even subject to ambiguity. Is a power surge to the hardware running the program an input?

Summary of Chapter 3 of Weinberg. There are an essentially infinite number of tests that can be performed on a particular product candidate. Rather than asking for "all" tests to be performed, managers and testers must strive to understand the risks added to the testing process by sampling.

Common Mistakes explored in Chapter 3. Seven common costly mistakes about software testing are explored in depth in the book and summarized briefly here.

1. Don't demand the impossible. Don't demand the impossible in the hope of getting best effort. Demanding "test everything": When you demand the impossible, you have no idea what you'll get—except that it won't be anything impossible.

2. Learn to sample or how to work with an expert. Not understanding sampling: Very few managers (very few people, in fact) understand sampling very well. Either educate yourself or hire an expert to audit your sampling. In either case, always be ready for the possibility of a sampling error.

3. Evaluate info. Spending too much for information that's not worth it: Do you have a basement or garage full of expensive gadgets that you never really wanted? Do you

realize what else you could have done with the money spent (or the space occupied)? If so, you understand this error. Be careful what you ask for.

4. Satisfy the spirit, not the letter, of external requirements. Testing for the sake of appearance: Some customers and certifying agencies demand "testing." You can go through the motions if you feel you must, but at least don't deceive yourself about the quality of the information you receive.

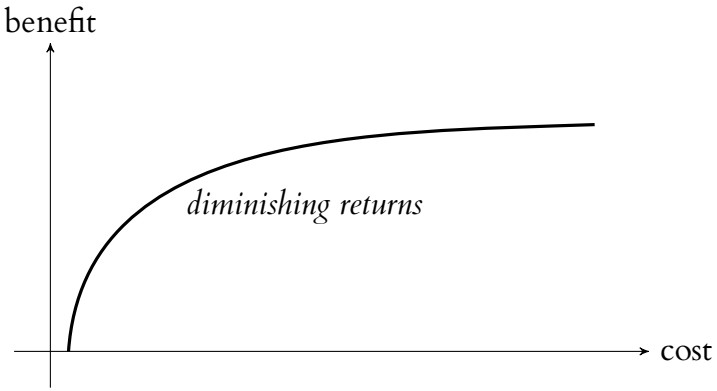
5. Know when you're wearing blinders. Not using all sources of information: Information gathered from test results is, by its very nature, limited, but there are other kinds of information sitting around if you're alert enough to see it.

6. Develop realistic expectations of machines. Thinking that machines can perform exhaustive testing, even if people can't: It's not just the human brain that's limited; testing tools are limited, too. Don't buy any product that claims it can "perform all tests." Even if it could, you couldn't possibly look at all the results.

7. Don't expect cost reduction to be proportional to outcome reduction. Increasing risk by constraining resources: When testing resources are cut, the easiest way to respond is by limiting sample size—running fewer tests. But with a reduced sample size, sampling errors become more likely. A diverse sample might find more problems than a large sample. Likewise, diversifying your test team might find more problems than enlarging your test team.

Dijkstra on testing. Note: E.W. Dijkstra (1930–2002) was a computing pioneer, quoted above in "Programming as a discipline of mathematical nature", *Am. Math. Monthly*, **81** (1974), No. 6, pp. 608–12.

END EXCERPT



Note: The cost reduction section above does not mention the issue of diminishing returns. The impact of any change in budget is far greater for values at the left side of a diminishing returns curve than at the right.

White-box testing. White-box testing refers to testing the internal structure of a program, examining it as if it were transparent. An example might be to search through source code for the arithmetic operation of division and to then identify what will happen if a zero denominator is supplied or under what circumstances a zero denominator might be supplied.

The above-mentioned article lists nearly a dozen criteria for testing, some of which are routinely tested by compilers. For example, many compilers will report whether any functions have been prototyped but never defined or defined but never called or whether any identifiers have been defined but never assigned values. Other criteria may be more or less satisfied by any given testing tool, such as whether every statement has been executed, whether every branch has been followed, whether every boolean expression has been tested both as true and as false.

White-box testing is an important historical term and can be helpful to think about in the course of planning tests.

On the other hand, contemporary testing tools don't make a white-black or transparent-opaque distinction and typically offer combinations of the two ideas.

Black-box testing. Black-box testing refers to testing the functionality of a program without any more access to the program than an end user would have. The program is regarded as opaque, or a black-box, whose inner workings are not known but which can be characterized completely by its inputs and outputs.

Constraints on coding. One Big Fluke suggested adding constraints like: assume you can only write a certain number of lines of code to focus your effort on the most important work. In other words, if you really want to spend your time wisely, pick an easily defined metric like 1,000 lines of code to translate the vague "spend your time wisely" into a concrete plan.

A commenter on the blog responded: "Adding constraints is one of the best ways to achieve good design but it requires something magical, good sense. If you don't have good sense then you won't know which constraints to add to make your work better. Chuck Moore uses simplicity, Alan Kay I think likes expressivity per line of code, Matz likes happiness, Gilad Bracha likes first-class values for all abstractions in a language, etc. The problem is that these people came to those conclusions after grappling with terribly designed computing systems and like all other lessons in life it doesn't really stick until you re-discover it on your own. The other problem is that companies like Facebook, Google, Amazon, eBay, etc. have so many programmers that it is impossible for all of them to do useful work so most of them are bound to do useless work."

Validation testing. Non-regression testing seeks to validate or verify whether a given unit of software development achieves an intended outcome. Validation testing should in-

clude machine-readable output or machine-readable results. These should be automatically comparable to ideal or reference results. For example, NIST promotes a configuration standard that supports writing output of security tests in an XML format that other software can use to automatically ascertain the security configuration of remote software (Mick heard this in a talk by Matthew Scholl, acting head of NIST Cybersecurity, on 19 Nov 2014).

Acceptance testing. Acceptance testing refers to the acceptance of a system by a client. This is one environment in which black-box testing may occur since an end user may not have access to system internal features.

Microsoft Patterns and Practices provides a thorough guide to acceptance testing. This guide divides the subject into various parts and subparts. The first part describes six ways of thinking about acceptance testing. The titles of the six ways are suggestive, so I'm quoting them here: doneness model (measurement methods), risk model (probability and consequences of problems), de-cision-making model (perhaps it would better named as the responsibility model), concern resolution model (procedure established in advance of use), system requirements model (identification of functional and non-functional reqs), and project context model (time and resource constraints).

The above guide provides a lengthy contrast between acceptance testing in an information technology department of an enterprise and a software product development enterprise. The division between functional acceptance testing and operational acceptance testing is addressed for about two-thirds of the total text.

Unit testing. Unit testing is described by Wikipedia as referring to the smallest possible unit of source code that can be tested. Unit tests are described as short code fragments ex-

ercised during development. Prescriptive advice in the above article is to ensure that unit tests are mutually independent. Further advice is to organize the tests so they help to document the development process. Finally, unit testing should be organized to support later refactoring and testing of refactored software. As a reminder, refactoring is the practice of altering source code without changing its external behavior.

Functional testing. Wikipedia on functional testing describes testing focused on the inputs and outputs of a system, without regard to the internal structure of the system. This is a more contemporary umbrella for black box testing focused on lists of intended inputs and expected outputs conforming to specifications or requirements documents.

Regression testing. Wikipedia defines regression testing as testing software after changes have been made, and uses the word regression synonymously with bug. The central concept is to ensure that new development on existing software does not introduce new faults.

MSDN warns that the most difficult aspect of regression testing is deciding what test cases to include. The advice at the above URL is avoid wasting time choosing test cases and simply err on the side of caution in choosing test cases. Further advice is to be sure to test timing and boundary conditions and to be sure to include automated tests, building a library of reusable tests and carefully testing fixes when regression testing uncovers faults.

Cost of writing test code. Spolsky on 12 Steps to Better Code compares costs relentlessly, including a comparison of the costs of toilet paper to wasted disk space (guess which cost more). Unfortunately, some of the most prevalent advice amounts to justifying early testing or, worse, justifying testing. Once we say that, Duh, we agree it's important, we would like to move on to managing the process of writing

test code so we spend enough and not too much. Quite a bit of available advice zooms in on the likelihood that *enough* will never be available.

David of Basecamp provides an alternative (and easily read, humorous view—even if you don't know who he is you can guess that he's influential) and quotes Kent Beck as saying *I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence (I suspect this level of confidence is high compared to industry standards, but that could just be hubris). If I don't typically make a kind of mistake (like setting the wrong variables in a constructor), I don't test for it.* This is useful advice for a student because it can help you build your own test approach, suited to your own growth and status.

tcpdump. To provide colorized output from tcpdump, say, e.g., `tcpdump -l -i en2 | colorize` where `colorize` is a perl script in /usr/local/bin. The `-l` option to `tcpdump` allows line buffering, so that the next process sees each line as it is emitted to `stdout`. Usually not used for colorizing but for monitoring while dumping to a file.

```
sudo tcpdump -i en0 -n "host 50.62.176.213"
```

Observation. "You can observe a lot by watching," said Yogi Berra and observation can help you to decide how to best spend your limited testing dollars. I often think of the recent media attention given to Yuri Totrov, who caused many deaths of Americans during the Cold War and whose success in identifying American agents ended many careers of Americans wrongly suspected as moles. A recent book and *Vanity Fair* article point out that he developed a method of examining publicly available evidence to discover spies. Is a US government employee using phone booths during working hours? Is a US government employee sitting alone in cafes

a lot? Visiting parks alone much? Avoiding the use of US government automobiles? Are there public announcements of new personnel who seem to serve different nonessential functions in various embassies all over the world? It supposedly took Totrov ten years to piece together a more-or-less foolproof spy identification system based entirely on observation. He was able to use that system for over a decade longer to kill and to cast career-ending suspicion on many innocents.

By the way, it was common for US spymasters to assume that the Soviets suffered from inferior intellect because the Soviets suffered from inferior technology. (Does this remind you of how technologists think about managers?) Therefore, they never once suspected that Totrov's system relied on thinking about publicly observed information. Thinking about what is observed may provide the ultimate test guidelines—if you are prepared to think.

USER HELP

Help may be obsolete as a system. Help in 2015 consists of Youtube, Google, and user forums. And successful use of help requires skill in navigating these three generic resources, none of which are the domain of the application developer. On the other hand, the requirement to add a help system may be foisted upon an application developer. How do you react to such a requirement? Does your reaction differ depending on the users and whether they form a natural community? Does it differ on the basis of sensitivity of data in the application? As an example, I developed an application for which I was not permitted to see the data that would be entered into it. I was asked to use proxy data of a non-sensitive nature and to develop that data myself so that any publicly available documents would not be contaminated by domain expertise and

thus become a target of adversaries. How would you react to this?

Why is help for development tools often better than for other software? Is it because developers believe that being helpful popularizes their tools? Is it because there is a natural community feeling between developers of developer tools and their customers, who are also developers?

One way that developers help users is to put employees into forums. This approach may not only help users but also may put the employee in the position of becoming a user advocate and a user expert.

Forums have a character. For instance, some people say that MSDN forums and documents are dry. One student supposes that MS outsources forum help, promoting uniformity of style as a way to keep contractors from forging a forum identity not congruent with corporate goals.

What are successful communities? Steam? Reddit? Stack Overflow? What makes a community successful? Passion for the product? Wanting to be connected to a product? Luck? One student reports that the oldest surviving community he has joined is `overclock.net` but opinions vary as to how it has survived without a schism for so long. Details about the history of Stack Overflow are plentiful since one of its cofounders is the prolific software blogger Joel Spolsky.

Time is a key issue in help. Some software has no version number, such as Google docs. It may change at any moment. If a user experiences trouble with such software, and looks at posts on Stack Overflow, there is no way to associate the date of a post to a particular version of the software. Locking and closing topics may help solve timing issues if moderators have some incentive to keep watch and forum software is easy to use.

Django exemplifies how software with versions can tackle

the problem of timely help. Version numbers occur in its help menu, along with warnings at the top of help pages that are very old (unsupported) or very new (largely untested).

Examples of companies with strong help in the gaming industry may include Jagex, Valve, and From Software. Blizzard maintains extreme control over their help offerings, evidently rejecting help from the community. The foregoing is based on student reports.

ARCHITECTURE

Taylor, Medvidovic, and Dashofy (2008) analyze architecture by dividing it into the following parts.

- Goals
- Scope
- Concerns (Structural, Behavioral, Interaction, Non-Func-tional)
- Models (Informal, Semi-formal, Formal)
- Type (Static, Dynamic, Scenario-based)
- Automation Level (Manual, Partially Automated, Automated)
- Stakeholders

Goals. Success of any project is measured using goals and finding ways to determine how and to what extent they have been met. Taylor, Medvidovic, and Dashofy (2008) posit the following goals as the four Cs.

- Completeness
- Consistency (elements do not contradict each other)
- Compatibility
- Correctness

How may each of these be measured? What do they mean?

Scope. Wikipedia offers a good survey on scope definitions, roughly including (among others)

- whatever is important
- whatever is hard to change
- whatever decisions are made and their rationale
- whatever is fundamental to understanding the software structure in its environment

Concerns. Non-functional concerns include efficiency, complexity, scalability, adaptability, and dependability.

Interaction concerns include interconnection within the system, communication with external elements, coordination of elements, and mediation of conflicts.

Behavioral concerns are functionality or processing concerns. This can include the flow of control as well as the flow of data.

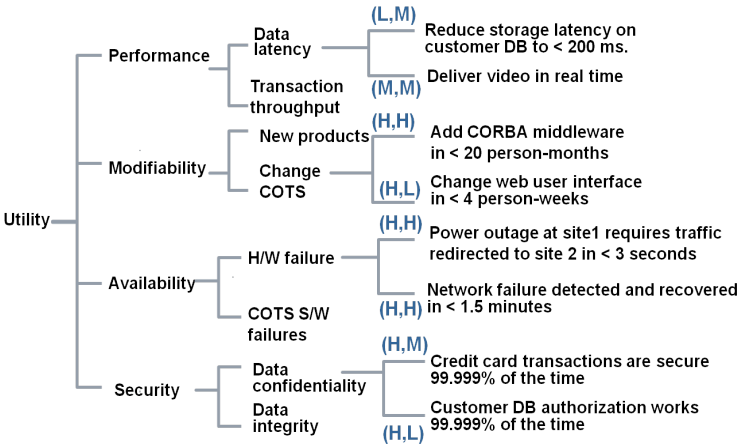
Structural concerns occur at a possibly more abstract level, incorporating the other concerns. Structural concerns may include inheritance, part-whole relationships, and adherence to paradigms or design patterns.

Stakeholders. The easiest and first way to analyze architecture is to think about the stakeholders in architecture. These may include the following groups.

- Architects
- Developers
- Managers
- Customers
- Vendors

From Architecture selection. The figure shows the utility tree from ATAM, the Architecture Tradeoff Analysis Method, developed by the Software Engineering Institute

at Carnegie Mellon for the United States Department of Defense. The items in blue (*X*,*Y*) are added after the tree is developed according to a complex, heavily documented process. They stand for different things in scenarios but in this one they are *X*: business priority, importance to the business outcome, and *Y*: technical priority, the technical difficulty or risk involved in doing them.



Paradigms in Software Architecture. Amirat, Hock-Koon, and Oussalah (2014) distinguish four paradigms in software architecture: object-oriented, component-based, agent-oriented, and service-oriented. Elements of these four paradigms are typically combined in hybrid approaches in the field rather than as pure expressions of the four paradigms. The following paragraphs summarize their descriptions.

The object-oriented paradigm established the principles of object encapsulation, inheritance, messages, typing, and polymorphism. The object-oriented paradigm was the first software engineering paradigm with widespread acceptance. It forms a reference model for all later paradigms.

The component-based paradigm arose as a response to the lack of reuse in object-oriented practice. This approach

stresses the use of simple components connected together in pipelines to perform large tasks. The independently developed, well-under-stood simple components are exposed and their reliability is enhanced by their ubiquity.

The agent-oriented paradigm is distinguished by software components able to cooperate, coordinate, and negotiate with each other. Autonomy and high-level interactions mark this paradigm as distinct from the others. An agent encapsulates code, state, and invocation initiative. An agent obeys explicitly defined rules in the service of explicitly defined goals.

The service-oriented paradigm emerges from the need to withstand volatile, heterogeneous environments. This paradigm emphasizes productivity and responsiveness of a service supplier by exposing APIs to services, with exactly one instance of a service available. A service has the following characteristics. A service is coarse-grained in comparison with component-based software, operating on more varied data and encapsulating conglomerated (possibly dissimilar) functions. A service has the flexibility to offer different interfaces and to offer interfaces in common with other services. A service is described by an API enabling a customer to understand what it does and how and at what price and with what non-functional properties. A service is discoverable. A service is a single instance, corresponding to the singleton design pattern.

CONCLUSION

This has been a database-centric course on application development. Should application development ever be database-centric? A lively discussion on just this topic can be found at Hacker News.

This conversation was started by an article called Truth

first, or why you should mostly implement database first designs.

One important point in the article and discussion is the importance of Truth First. The notion of Database First comes from the widespread belief (but not everyone agrees) that data is more persistent than processes. If so, it is closer to the truth of the enterprise than are processes. You have to think about this because you have to make this choice every time you have the privilege of developing a new system and every time you have the burden of maintaining an old system.

:: APPENDICES ::

The following sections are appendices—you've reached the end of the lecture material. Most of what follows is practical material you'll use for your project, exercises, and exams. The most important part of the course is your project. Before you start looking at the resources associated with the project, look at the linked Hacker News conversation about the question of Automating CRUD Web UI construction. It includes spirited arguments about layers, abstraction, whether and to what degree automating web ui construction happens in different current workplace environments. It will help you think about what you are doing as you work through your project milestones and exercises.

The appendices include

- Node.js materials, including two hello world tutorials, a simple mysql application, and some information about using Node.js with databases
- Project milestones
- Exams
- Exercises
- Software used in this course

- Pointers to resources on writing well
- References

You should refer to these appendices throughout the semester, using the table of contents tab available in most pdf viewers. You can plan your work better if you familiarize yourself with these appendices early in the semester.

NODE.JS INTRO

You need to familiarize yourself with Node.js if you're taking any database course from me: 432, 438, and 610 all use Node.js to connect to databases and to end users. This tutorial should help you learn to work with Node.js and several different database systems.

This tutorial assumes you have access to the virtual machine that can be downloaded from `http://serenity.ist.edu/~mjmics` and a copy of VMWare to host it. You can only access this URL from the RIT campus or the VPN. If you are off campus, you must use the VPN or the above URL will not be visible. In addition to downloading the virtual machine and hosting it on your own machine, you can also use it directly on the IST lab machines, where it is the only Ubuntu 18.04 VM available via VMWare.

The virtual machine is a modified version of the Ubuntu 18.04 virtual machine available from Canonical. It has been modified by installing the products used in this tutorial, including Node.js, MySQL, sqlite3, MongoDB, and Neo4J, as well as some utility files and configuration files. Wherever possible, passwords have been set to *student* to conform to the practice in IST labs.

There are three different hello world programs to write here. The first one displays Hello World to the console, while the second displays it on a webpage. Third, rewrite the hello world application so that some part of it can be easily tested.

Console version. To write Hello World to the console, first open a terminal window. Use an editor to type the following into a file called `hwc.js`.

```
console.log("Hello, World!");
```

You may then run this program by typing the following at a terminal prompt.

```
node hwc.js
```

This program introduces an important object, the console, and a method for that object, log. You will use this throughout this tutorial to help you understand and debug your code. You will spend quite a bit of time using the terminal so you should familiarize yourself with it and with a full-featured editor.

Web version. Node.js is frequently used to develop programs that will run on the web. Typically, the end user interacts with such programs using a web browser and never sees the console. Therefore it makes sense to write a hello world program that works on the web. This is quite a bit more complicated than writing output to the console, which is one reason the console is used for debugging.

In order to write hello world to a web page, you must rely on one of the features of Node.js that makes it popular. That is its ability to specify a web server, allowing you to write a very small amount of code to accomplish your task and allowing you to avoid writing to folders used by default web servers like Apache. You will tell Node.js to import some server code and run that. All you will have to do is specify a few details.

Before you write the required code, be aware that throughout this tutorial, you will write only to a server available on your local machine. You can move the software you will write to a public server but it may be best to learn a little about Node.js before you do so.

Step 1, create a folder. The first thing to do is to create a folder in which to put your program. This is common practice in writing Node.js programs. The reason is that few Node.js programs operate in a vacuum. There are many common tasks that are already well served by software in the Node.js ecosystem. You can import most of the needed software into your Node.js program and write only the bare minimum (it may seem like a lot when you are first writing it!) to make your program perform popular tasks. It is customary to import Node.js software into a subfolder of your project folder although you may also import Node.js software globally into your system. There are reasons we will cover later for why you may prefer to import software locally into your current project. For now, let's just do it. The easiest way to do it is to create a directory. Call this directory firstserver and change to it. At the terminal prompt say

```
mkdir firstserver && cd firstserver
```

Notice that these commands are separated by a pair of ampersands. That causes the second command to be run only if the first command is successful. You can of course break this up into two separate commands by separating them with a semicolon instead or by pressing Enter between them.

Step 2, initialize the project. You should be in the `firstserver` folder and can verify this by saying

```
pwd
```

at the terminal prompt. This is the root folder of your project and it is here that you will create a `package.json` file which is customarily found at the root of any Node.js project. You can write the file in a text editor or copy it from another project but in this case, use Node's package management system to create it. At a terminal prompt, say

```
npm init
```

The Node.js package management system is called NPM. It is installed with Node and offers a great deal of functionality. Think of it as your link to the Node.js ecosystem, among other things. When you type `npm init`, you will be asked a lot of questions and the default answer will be typed in parentheses. If you press Enter in response to any question, `npm` will behave as if you had typed in the default answer. For such a simple project as a hello world program, you can probably guess that the answers don't matter much. Feel free to accept

all the defaults or not. The simplest thing to do is to keep pressing Enter until the terminal prompt reappears.

After you have finished answering the questions, `npm` will display a picture of the `package.json` file it intends to write and ask you if this is okay. Again, you are free to accept the default or alter it if the mood strikes you. Once `npm` has completed writing the file, you can view it in a text editor and you should find that it looks no different from what you expect.

Step 3, create the server. You may have noticed that `npm` supplied the name of a file as an entry point into your project. Now write the following code into that file, by default called `index.js`.

```
var http = require("http");
http.createServer(function (req,res) {
  res.writeHead(200, {"Content-Type":"text/plain"});
  res.write("Hello, World!");
  res.end();
}).listen(8081);
```

Step 4, verify that it works. The above file introduces a number of concepts but first, verify that it works. Type

```
node index.js
```

at the terminal prompt. You should see no result except that no new prompt will appear. The terminal will appear to hang. Now open Firefox, browse to `http://localhost:8081` and you should see the words *Hello, World!* appear in the browser window.

You have actually accomplished quite a bit though it may not appear so. In your first line of code, you instantiated a variable that is an object of type `http`. That object has methods available to it, such as `createServer()` and `listen()`. The `createServer()` method takes an unnamed function as its parameter and that function in turn takes a request (req) and a response (res) as its parameters.

There are some remarkable concepts at play in this simple example and you will gradually learn more of them. For now it is enough to know that your program can communicate in one direction on the web. You can write whatever text you wish to a webpage. In practice you would not. Instead you would use various packages of code to construct the webpage and define its appearance. What you see here is just a basic connection and you will have to make a number of basic connections to work with a database system.

A test-driven hello world. You verified your simple program by running it and observing the result. In the workplace, this is not a practical method for evaluating your work. You will work on large programs in teams and it will often be necessary to run tests on software after a few changes have been made to verify that it still works. There are many approaches to testing and your hello world program is too small for some of them but you can modify it to illustrate the concept of a unit test.

Step 1, modularize. First, create a new file called `hellow.js` and put the following code into it. This is a module and can be tested to see if it does what is advertised.

```
'use strict';
var hellow = function () {
```

```
  return 'Hello, World!';
};
module.exports = hellow;
```

This creates a function that returns a string and we can plug this into our main program by modifying `index.js` as follows. Notice that now `res.write()` accepts as a parameter a function that returns a string instead of a hardcoded string. You have created a module that writes the string in a separate file.

```
var http = require("http");
var hellow = require("./hellow");
http.createServer(function (req,res) {
  res.writeHead(200, {"Content-Type":"text/plain"});
  res.write(hellow());
  res.end();
}).listen(8081);
```

Step 2, create the test. Now create the test specification in a file called `testSpec.js`. You can run this test by importing a test framework into your project. That test framework is called `jasmine-node` and is one of the half-dozen most popular test frameworks available in the Node.js ecosystem.

```
'use strict';
var hellow = require('./hellow.js');
describe('hellow',function () {
```

```
  it('should return Hello, World!', function () {
    expect(hellow()).toEqual('Hello, World!');
  });
});
```

Step 3, import the test framework. To import the `jasmine-node` framework, say

```
npm install jasmine-node --save
```

at a terminal prompt. This will not only import `jasmine-node` but also the many other packages upon which it depends. Using the `--save` option will cause the `package.json` file to be updated to include `jasmine-node` as one of the dependencies of this project. View `package.json` in a text editor to verify that `jasmine-node` is now listed, along with its current version number. It is often necessary to specify version numbers of dependencies because packages are often updated and compatibility issues may arise when some packages are updated. You may sometimes be better off to use an old version of a package than a new and possibly different version.

Step 4, run the test. You can run the test in one of two ways. You can say

```
./node_modules/jasmine-node/bin/jasmine-node \\
  testSpec.js
```

or you can say

```
./node_modules/.bin/jasmine-node testSpec.js
```

Either way, you should see output like this:

```
hellow - 5 ms
    should return Hello, World! - 5 ms

Finished in 0.008 seconds
1 test, 1 assertion, 0 failures, 0 skipped
```

You can also make the test fail by changing a variable name or deleting a file or in many other ways. You should play around with this test in order to get an idea of how the output should look given different errors.

A SIMPLE MYSQL APPLICATION

Next, create a simple mysql application. This application is from a website I have unfortunately forgotten so I can not give a proper attribution unless someone else figures out where it comes from. Please let me know.

This application resides in seven files plus the many files that will be added to `node_modules`. The seven files will all reside in the same folder and will be connected to each other with one exception. The file `createDB.js` will not be connected to the other files. It will be run independently every time you need to recreate the database. The other files, except `package.json`, will be called from `index.js` or a file called by `index.js`, which will be the entry point for the application.

Prepare the folder. First create a folder called `secondserver` and change to that folder, for instance by saying the following at a terminal prompt.

```
mkdir secondserver; cd secondserver
```

Once in that directory, create the `package.json` file by saying

```
npm init
```

and following the prompts. Accept the defaults so that the entry point is `index.js` and answer the other questions as you wish. The fastest thing to do is to press Enter after every prompt.

Run the MySql server. Before you create the database, the MySQL server needs to be running. Start it by saying

```
sudo /etc/init.d/mysql start
```

at a terminal prompt. You will have to give the sudo password, which is *student*. Now you can run MySQL as a client as you have done in previous classes, or interact with MySQL from a program. To turn off the server, you will afterward use the same command as above but with the word `stop` replacing the word `start`.

Write the db creation script. Now type the following script into a file called `createDB.js`.

```
'use strict';

var mysql = require('mysql');

var conn = mysql.createConnection({
  host:     'localhost',
  user:     'root',
  password: 'student'
});

conn.query('drop database if exists secsrv',
  function (err) {
  if (err) {
    console.log('Could not drop db secsrv.');
  }
});

conn.query('create database secsrv',function (err) {
  if (err) {
    console.log('Could not create db secsrv.');
  }
});

conn.query('use secsrv',function (err) {
  if (err) {
    console.log('Could not switch to db secsrv.');
  }
});

conn.query('create table bla '   +
  '(id int(11) auto_increment, ' +
```

```
  ' txt varchar(255), '          +
  ' primary key(id))',
  function (err) {
    if (err) {
      console.log('Could not create table bla.');
    }
  }
);

conn.query("insert into bla (txt) values ('foo')");
conn.query("insert into bla (txt) values ('bar')");
conn.query("insert into bla (txt) values ('bas')");
conn.query("insert into bla (txt) values ('qux')");

conn.end();
```

Notice that the above script drops the database if it exists, then creates the database, then inserts some data into it. It needs not only MySQL but a MySQL driver for Node.js to run so, before we run it, we'll have to install that driver.

Install the mysql driver. At a terminal prompt, say

```
npm install mysql --save
```

This will install the node mysql driver and save the fact that the project depends on it in the `package.json` file.

In the `node_modules` folder you will see, in addition to a folder for the driver, folders for various other packages upon which it depends.

Run the db creation script. You need to run the script at least once. Then, after you play around with the following scripts, you may want to run it again to start fresh. Say

```
node createDB.js
```

at a terminal prompt. You can verify the success of the script by opening mysql in a separate window. Say

```
mysql -uroot -p
```

at a terminal prompt and give the password *student* at the prompt. Once you have the MySQL REPL running, say

```
use secsrv
select * from bla;
```

and you should see foo, bar, bas, qux, listed as content and integers 1 to 4 as id. You may quit MySQL after this or leave the REPL open to test later interactions.

Create the entry point. The main entry point for the application does very little except to use the `http` module to instantiate a web server and call another file to give the details of that server. It also specifies the port at which the server will listen for requests as 8081. This means that you can point a browser on the virtual machine at localhost:8081 to make requests of the server. The contents of the `index.js` file are:

```
'use strict';

var http           = require('http'),
    acceptReq      = require('./acceptReq');

// Start web server on port 8081; requests go to
// function acceptReq
http.createServer(acceptReq).listen(8081);
```

Create the main logic. This file contains the main logic for the web server in a function called `acceptReq()`. It accepts two kinds of requests, requests to add an item to the database, and requests to limit the displayed contents to a subset of the database. The contents of `acceptReq.js` are:

```
var url            = require('url'),
    querystring    = require('querystring'),
    insertIntoDB   = require('./insertIntoDB'),
    retrieveFromDB = require('./retrieveFromDB'),
    pageContent    = require('./pageContent');
// Accept http requests
function acceptReq(req,res) {
  var pathname = url.parse(req.url).pathname;

  // Request to add to database
  // (POST request to /add)
  if (pathname == '/add') {
    var reqBody = '';
    var postParams;
    req.on('data',function (data) {
```

```
      reqBody += data;
    });
    req.on('end',function () {
      postParams = querystring.parse(reqBody);
      // Content to add is in POST parameter "content"
      insertIntoDB(postParams.content,function () {
        // Redirect back to / when finished adding
        res.writeHead(302, {'Location': '/'});
        res.end();
      });
    });
  // Request to read from database
  // (GET request to /)
  } else {
    // Text for limiting is in GET parameter q
    var l
      = querystring.parse(url.parse(req.url).query).q;
    retrieveFromDB(l,function (contents) {
      res.writeHead(200, {'Content-Type':
        'text/html'});
      // Instead of templating engine: replace
      // DBCONTENT
      // in page HTML with content from database
      res.write(
        pageContent.replace('DBCONTENT',contents));
      res.end();
    });
  }
}
module.exports = acceptReq;
```

Create the select logic. This file contains a function called `retrieveFromDB()` that connects to the database and

retrieves either everything in table `bla` or a subset limited by the user employing the SQL LIKE clause.

The contents of `retrieveFromDB.js` are:

```
var mysql = require('mysql');
// Retrieve from db, limiting to subset if requested
function retrieveFromDB(limit, callback) {
  var conn = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'student',
    database: 'secsrv'
  });
  var query;
  var resultAsString = '';
  if (limit) {
    limit = '%' + limit + '%';
    query = conn.query('select id, txt from bla '+
      'where txt like ?',limit);
  } else {
    query = conn.query('select id, txt from bla');
  }

  query.on('error',function (err) {
    console.log('A database error occurred:');
    console.log(err);
  });

  // Build string result by result to later
  // replace into html of webpage
  query.on('result',function (result) {
    resultAsString += 'id: '     + result.id;
    resultAsString += ', text: ' + result.txt;
```

```
    resultAsString += '\n';
  });

  query.on('end',function (result) {
    conn.end();
    callback(resultAsString);
  });
}
module.exports = retrieveFromDB;
```

Create the insert logic. Next, create the file `insertIntoDB.js`, containing the function `insertIntoDB()`, for inserting user-supplied records to the `bla` table.

```
var mysql = require('mysql');
// Function called by code that handles the /add route
// and inserts the supplied string as a new txt
// entry
function insertIntoDB(txt,callback) {
  var connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'student',
    database: 'secsrv'
  });

  connection.query('insert into bla (txt) '+
    'values (?)', txt,
    function (err) {
      if (err) {
        console.log('Could not insert text "'+txt+
```

```
            '" into database.');
        }
        callback();
    });
}
module.exports = insertIntoDB;
```

Add the html content. Normally you would use a view engine to render html for the user. Here we just have some raw html to insert as a string variable. Put this into a file called `pageContent.js`.

```
// Page html as one big string with placeholder
// "DBCONTENT" for data from the database
var pageContent =
  '<html>'+
  '<head>'+
  '<meta http-equiv="Content-Type" '+
  'content="text/html; charset=UTF-8" />'+
  '</head>'+
  '<body>'+
  '<form action="/add" method="post">'+
  '<input type="text" name="content">'+
  '<input type="submit" value="Add Text" />'+
  '</form>'+
  '<form action="/" method="get">'+
  '<input type="text" name="q">'+
  '<input type="submit" value="Limit Text" />'+
  '</form>'+
  '<div>'+
  '<strong>Text in bla table:</strong>'+
```

```
    '<pre>'+
    'DBCONTENT'+
    '</pre>'+
    '</div>'+
    '</body>'+
    '</html>';
module.exports = pageContent;
```

Run the server. Say `node index.js` at a terminal prompt and the server should start. You won't see any indication of it because you have not written any messages to the console to notify you that the server is running. The only indication that it is working will be to take the next step.

Open a client. Open a web browser and point it to `http://localhost:8081` and you should see a couple of boxes with the words Add Text and Limit Text beside them. Beneath the boxes should be the words *Text in bla table:*.

HOW NODE FITS IN WITH DATABASES

One important difference between operating a database interactively and connecting to it from a program is time. The database may be on a different machine from the program, in a different city or country. The time it takes for the database to respond to requests may already be slow interactively depending on the query but it may be compounded by network delays or program logic.

If you are interacting with the database at the console, you may be predisposed to wait for it to return an answer to a query or you may leave it running while opening up another window to do something else. That is really all you can do unless you can somehow make the database work in the

background and allow you to make additional queries before it returns the answer to your first query.

Node callbacks. Writing programs to connect to the database allows us the flexibility to leave the database running while the program does something else. Some languages have special facilities for making this easier. One of the most basic is the notion of the *callback*.

The basic idea of the callback is to start some process, call it *A*, running and then start other processes *B* and *C* running instead of waiting for *A* to finish. Suppose some process *D* depends on *A* finishing before *D* can start. We need a way to let *B* and *C* start right away, but only let *D* start when *A* is finished. In this scenario, we can refer to *D* as a *callback* and set up a way for it to start only when *A* finishes.

You can accomplish the above scenario in JavaScript, the language of the Node system, by using the fact that JavaScript supports *first-class functions*. For a language to support first-class functions means that functions can be treated as any other variable.

You can pass a function to another function as a parameter in JavaScript and let that passed function run when the called function is finished. The function that gets passed in is usually referred to as a *callback* even though there is no reserved word *callback* in JavaScript.

Therefore in JavaScript it helps to write your code in a manner that illustrates dependencies between functions. In Java you might write statements one after the other, expecting them to execute one after the other. In JavaScript, that is not necessarily the case. Statements don't necessarily execute one after the other unless you take special care to write them that way.

Callback examples. Let's look at two examples of callbacks from a document called *Art of Node*. I'd like you to read

Art of Node for more detail on what I'm describing briefly in fact. You can find it at https://github.com/maxogden/art-of-node and you should also check out a companion piece called *Callback Hell* at http://callbackhell.com/ originally by the same author, although *Art of Node* now has been forked on Github by several contributors.

Anyway, here is the first example. You should run this code. You'll have to create a file called `nr.txt` in the same folder and put a number in it. For example, you could say the following at a terminal prompt.

```
echo '1' >nr.txt
```

Then put the following code in a file called `firstnr.js` and run it by saying `node firstnr.js` at a terminal prompt.

```
var fs = require('fs'); // require is a special function
                        //provided by node
var myNumber; // we don't know what the number is
              // yet since it is stored in a file

function addOne() {
  fs.readFile('nr.txt', function readDone(err, fc) {
    myNumber = parseInt(fc);
    myNumber++;
  });
}

addOne();
```

```
console.log(myNumber); // prints undefined -- this line
                       // is run before readFile is done
```

What you should find puzzling about this is that it logs the word `undefined` even though you have already called `addOne()`. Imagine that this is a database read and you try to use the value before it is returned. This is a simple analogy to that problem.

Here is a corrected version, using a callback. Put this in a file called `secondnr.js` and run it by saying `node secondnr.js` at a terminal prompt.

```
var fs = require('fs');
var myNumber;

function addOne(callback) {
  fs.readFile('nr.txt', function readDone(err, fc) {
    myNumber = parseInt(fc);
    myNumber++;
    callback();
  });
}

function logMyNumber() {
  console.log(myNumber);
}

addOne(logMyNumber);
```

Now the statement logging to the console is trapped in a function that is not executed until the end of the function

`addOne()`. The function `logMyNumber()` is passed as a parameter to function `addOne()`.

Callback hell. The expression *callback hell* arises from the fact that many processes can wind up in a dependency chain leading to lots of ugly-looking nesting. One reason for this is that functions don't need to be named in JavaScript. They can be anonymous functions. So you can have a pattern like this:

```
      POST: function (req,res) {
    req.onJson(function (err,newKeyword) {
      if (err) {
        console.log(err);
        res.status.internalServerError(err);
      } else {
        dbSession.query('INSERT INTO keyword (value, categ
          if (err) {
        console.log(err);
        res.status.internalServerError(err);
          } else {
        res.object({'status':'ok','id':dbSession.getLastI
          }
        });
      }
    });
      }
    }
  );
```

The above is just a fragment. There are three anonymous functions shown here plus some others above for which I've just shown the trailing parens and braces. One of the anony-

mous functions is actually invisible because it runs off the right side of the page on the line that begins with `dbSession`. It is not important for you to see the rest of the code but simply to identify it as a common readability problem. Readability problems cause debugging problems so I encourage you to name functions in most cases and avoid deeply nested parentheses and curly braces. It is not that they are inherently wrong but that students who practice this kind of coding often struggle with debugging.

Promises. The standard solution to callback hell is to use a language feature called *promises* and only use callbacks when you can afford to keep the code small. If you must have a long dependency chain, promises provide a better solution. One catch is that they are only introduced in ES6 so there are vastly many browsers that do not support them. One way to cope with that problem is to use a library such as `Bluebird` to support promises on platforms that are not ES6 ready. You can check for ES6 compatibility at many sites, such as https://kangax.github.io/compat-table/es6/.

There is a tutorial you should do at http://stackabuse.com/a-sqlite-tutorial-with-node-js introducing promises with `sqlite3`. There is (at the time of this writing) a small error in the code, though, so be sure to add

```
module.exports = ProjectRepository
```

to the end of the `project_repository.js` file and a similar line to the end of the `task_repository.js` file. A corrected copy of the code is on myCourses as `node-sqlite-tutorial.tar`.

Project milestone grades are shared by all members of a group. Peer evaluations may result in different scores based on the instructor's judgment. All project milestones except the first milestone involve a *development book*. You must continue building this development book throughout the semester, documenting your work. There is not a different development book for each milestone, rather the development book is revised and improved with each milestone.

Please do not turn in any Microsoft Office files in this course. If you work in Microsoft Office tools, please convert your files to pdf before turning them in. Please use markdown, latex, or pdf or link to google docs for reporting purposes.

Milestone 1. Requirements. Identify requirements for a data-base application. These requirements should include the following two essential requirements that are at the heart of this course.

- dynamic interaction with a publicly available data source, such as Google Maps, the IMDB, a sports scoreboard, a securities market, a public questions forum like Stack Overflow, a code repository such as Github, or any such source with relatively heavy traffic
- an intrinsic need for multi-user CRUD operations requiring you to take steps to maintain data integrity

Beyond these essential requirements are the requirements of a genuine customer. Ideally, you should find a customer for a database application and work to identify and meet their requirements. You will learn more and potentially earn a higher score if you are able to work with a genuine customer than if you have to invent a customer and create a database application from your own imagination.

Identify the customer requirements in a short paper divided into sections including summary, goals, stakeholders, scope, input, processing, and output. You may find the headings list for goals and stakeholders in the architecture section of these lecture notes. Note that projects vary in their emphases. It is up to you to state explicitly that certain goals or stakeholders will be ignored and to give a plausible reason.

The paper may be in markdown format or a google doc. If in markdown, it may be placed on a github repository that will be the home of the project for the remainder of the term or in the milestone 1 dropbox. If you use a google doc or github repository, you must submit a plain text file to the dropbox identifying where the actual work is to be found. If you use a markdown format and want to include pictures, please use the markdown format for a picture, e.g.,

```
![caption of picture](picture.jpg)
```

where you want the picture to appear in the document. If you submit the markdown to the dropbox, you must then also submit the picture(s) there. If you use github, you must follow their rules for pictures.

Milestone 2. Design and Design Patterns. This is the second milestone for the project which should occupy you for the rest of the semester. You should develop a project supporting integration of the course topics. That means that the project should

0. face potential issues with data input
1. face potential issues of data integrity
2. be amenable to the specification and use of design patterns

3. use a layered architecture
4. provide exception handling in a layered manner
5. include testing
6. require some authentication and authorization work
7. include user help of some kind
8. be packaged for some degree of portability
9. be refactored to some extent near the end of the semester
10. be designed with cognizance of potential regulatory issues

All milestones will include a revision of the development book presented in this milestone. Later milestones will include code. Please do not use any zip files except to bundle code. Please present this document in pdf, markdown, html, or a format agreed upon by the instructor (not proprietary, not ms office). Pictures should be presented in the pdf or in separate files readable by the instructor (not proprietary, not visio or bmp).

For this milestone, begin the development book with headings including

1. Team Members and Roles
2. Background
3. Project Description
4. Project Requirements (the voice of the client)
5. Business Rules (subject matter constraints like "you must buy something before you can sell it" or app-related constraints like "you must log in before you can have a transaction")
6. Technologies Used
7. Design Patterns
8. Timeline (mainly milestone due dates)

The timeline should be organized around the future milestones.

As you revise this document you will need to provide a way to see how it has changed. For instance, if you change the technologies used, the Technologies Used section should still list the original technologies planned to be used in a subsection where you describe why you switched.

Add a section to your development book called Design Patterns. In that section, identify the design patterns relevant to your project. You may change these in future. This represents your current view of design patterns in your project. If and when you change, move the previous material to a subsection.

Improve the other parts of your development book based on in-class discussions, your own reflections, and any examples posted.

Following are some issues students faced in completing this milestone in previous iterations of this course.

Mediator is problematic for layers. In the layered architecture you should probably use for a typical database app, each layer sends to only one other layer and receives from only one other layer. There should not be a separate infrastructure for communicating between all layers because they don't all communicate. The database sends data to the data layer. The data layer sends data to the business layer. The business layer sends data to the application layer. The application layer sends data to the presentation layer.

You may merge the business layer and application layer into one depending on the application. The minimum number of layers should be three, though. The data layer should be the only layer with sql. The presentation layer should be the only layer that communicates with the end user.

Diagrams alone are not sufficient. Diagrams, and especially generic diagrams, don't describe the use of design patterns in your project. One project just showed a generic diagram of an adapter design pattern. In this case, it also does not tell how the adapter pattern actually works! The diagram shows a controller and client both pointing to an adapter. If the adapter is *gluing* two incompatible classes together, should there not be an arrow pointing out of the adapter to one of the two classes. Moreover, it does not tell anyone how you see design patterns actually working in your project. It is probably in your head but it needs to be on the paper.

Do more than mention a design pattern by name. One project just said that MVC would be used because it was needed but did not say how or why. MVC can be used in pretty much any user-facing application so some description is needed.

MVC needs to be better explained. One group described how MVC would be used but it did not seem realistic. For example, in this case it was said that the model would be *the data in the database*. The view would be *what the user sees* and the controller would be *the backend functionality*. These descriptions are too generic. Think about what may be possible.

Always be coding. One issue students face is that they don't have enough code too late in the development of the project. By including sample code for the design patterns, you get a head start on coding.

Milestone 3. Layering. Please add a section to your development book called Layering. Within that section, please add at least three subsections, each one describing a layer of your layered architecture. You may rely on the layering concepts from the prerequisite course, Database Connectivity and Access, for this description. It should include as much specificity as possible. For example, it would be helpful to include the names of the classes you plan to write in each layer, as

well as brief descriptions of the classes. You should include code demonstrating each of the layers. From this point on in the development book, you should always be coding and always be showing sample code.

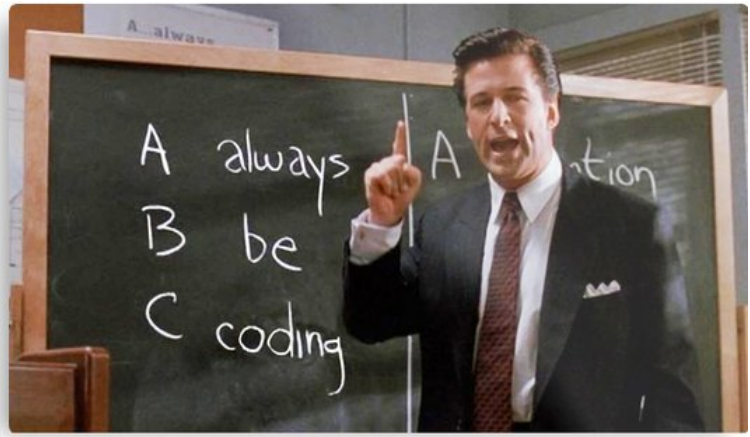
Some concerns that crop up in the development of a development book can be grouped together under the heading of *vagueness*. Bullet points are problematic if they have a subject and no predicate. I need sentences with a subject and predicate in order to have a fighting chance if I am a compiler or interpreter or other computery kind of a thing. Computers are notoriously small-minded and can't do much with ambiguity. So, if I see "photo URL data" I'm not sure what that means from a design standpoint.

Design is working with constraints. Design is solving problems that arise because of the tension between constraints. Design is making tradeoffs, non of which are utterly perfect. So we need to know the details so we know what we lose and what we gain by each choice. Whenever we commit to some choice, we've narrowed our options.

A google map overlay called maptube exemplifies a choropleth map, a geo-artifact in which regions are colored according to numerical values of some characteristic. Designing effective choropleth maps is difficult. The color aspect, for instance, is the main work of Rebecca Brewer whose app ColorBrewer helps designers solve individual instances of the problem.

Milestone 4. Exception Handling. Please add a section to your development book identifying exceptions and categories of exceptions you expect to account for in your code. You must include examples of actual exception-handling code in this milestone.

The most important thing to do in this milestone is to identify the location of exception handling responsibility



within classes. Where will exceptions ceased to be passed on? No exceptions should ever be passed to the user. Almost no exceptions should ever be passed to the presentation layer. Most exceptions should be handled in an application, business, or data layer. Identifying them should also identify the kind of person who should respond to them. I have often encountered DBAs who insisted that the responsibility lay with an "application owner" and application developers who insisted that the same responsibility lay with the DBA and "application owners" who insisted that they were paying for somebody else to figure this out. If the development book spells out enough detail to resolve such issues, the issues may never arise. The tricky thing here is that there is probably not a cut-and-dried recipe for "enough detail." It is application specific. You have to make choices and articulate, within this section of your development book, the responsibility for these choices. If your project is too trivial to have any such issues, then you risk the possibility of a low grade on this milestone just because you were too conservative in identifying a worthwhile project.

The work is to plan how and where you handle exceptions in a layered architecture.

When an exception is thrown, it can be handled or passed up the layers. You have to make a decision for each "try / catch" whether you want to handle the exception here or elsewhere.

The above implies that you have a layered architecture. Do you?

Your development book has to specify some kind of a layered architecture.

So I am not asking for an exception handling section of the design but rather an expansion of the layered architecture description that shows us how and where exceptions are handled.

Milestone 5. Performance and Refactoring. Add a section to your development book called Performance and Refactoring. Give examples of code there or coding practices there that you are doing to improve performance of your project. For projects where we identified refactoring opportunities, carry those out and all groups should include their current code with the document. The best case would be to put pointers in the document (e.g., filenames and line numbers or classnames and offsets) rather than code fragments in the document.

Improve your existing code and respond to *surprising* instructions from management. This milestone may be particularly stressful depending on how well you have completed the previous milestones. The *surprising* instructions will test the quality of your code thus far.

Milestone 6. Testing. Please add a section to your development book identifying your testing framework. Please include specific code and diagrams as appropriate. Instrumented code and build files should be in separate files from the devel-

opment book or simply included in the document as appendices and explained in a section in the main body. Whether you include them as appendices or separate files, they should be displayed with line numbers so you can make references to specific parts in the narrative in the testing section of the development book. It needs to be unambiguous as to which file or appendix you are referencing in your narrative.

Don't make implicit assumptions. Be specific even to the point of being tedious in your explanations. I am not looking for broad coverage so much as good coverage of whatever you do test. Also, even though I say I am not looking for broad coverage, the examples you do use should not be trivially different versions of each other.

The Wikipedia entry on Unit Testing says the following about unit testing. Please be sure to include a unit test that conforms to this specification for your database layer.

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should usually not go outside of its own class boundary, and especially should not cross such process/network boundaries because this can introduce unacceptable performance problems to the unit test-suite. Crossing such unit boundaries turns unit tests into integration tests, and when test cases fail, makes it less clear which component is causing the failure. See also Fakes, mocks and integration tests.

Instead, the software developer should create an abstract interface around the database queries, and then implement that interface with their own mock object. By abstracting this necessary attachment from the code (temporarily reducing the net effective coupling), the independent unit can be more thoroughly tested than may have been previously achieved. This results in a higher-quality unit that is also

more maintainable.

Please create a mock object to mimic your database. It should exhibit the same interface as your database although it will respond in a canned manner. This is described in the Wikipedia entry for Mock Object.

Milestone 7. Deployment, Packaging. Please add a section to your development book detailing everything needed for packaging and deployment. This should include README files or whatever is provided so that a user with no access to you can simply install and run your app based on the info you provide here or which is pointed to from this location. For example, if your app can be cloned from github, you should point to the location and provide any instructions needed in addition to the relevant git commands. These instructions should include any required packages of any kind. As another example, if your app is hosted on a website, provide complete instructions for setting it up on someone else's website. There should be some kind of package that can be delivered to a client and used without the personal intervention of your team.

Finally, you should include a help specification, detailing what kinds of help would be provided in a full-blown installation of your app. There is probably not time to create a full-blown help system so provide a substantive example of help for your system by picking one aspect of your app as a target for help.

EXAMS

Students will face three unequally weighted exams.

Exam 1. Locking. Describe various locking schemes and the consequences of using them in specific situations.

Exam 2. Access Control. Describe various methods for controlling user access and the consequences of using them in specific situations.

Exam 3. Final Exam, Material after Access control. Answer questions about performance, refactoring, testing, help, architecture, and deployment.

EXERCISES

Students will work on exercises in class. Some may be completed in groups while others need to be completed individually. The instructor needs to be able to observe progress on the exercises to support mastery of the skills involved so, while exercises may be finished outside class, major work on exercises needs to be completed in class in view of the instructor.

Please do not turn in any Microsoft Office files in this course. If you work in Microsoft Office tools, please convert your files to pdf before turning them in. Please use markdown, latex, or pdf or link to google docs for reporting purposes.

Exercise 1. Review. Your first exercise will include an entity relationship diagram, a list of functional dependencies, and relational notation. All this will be based on the KGA documents and your own experience and imagination in guessing the future directions for the app, which is grossly underspecified. This will illustrate coping with specifications that can not possibly be final, in an environment where the development of the database can not be delayed nor synchronized with the app developer.

In addition to the above three artifacts, you will need to include some narrative to make your choices clear and persuasive. You need to think of questions someone might ask when examining your document of the form "Why did you include the *bla* column / table in this table / database?"

This exercise reviews what you have learned in prior courses and establishes a baseline. In it, you will look at a manager's description and, from that description, generate at least

1. ER diagram (s)
2. Functional Dependencies
3. Relational Notation
4. Clarifying and Persuading Narrative
5. Write the necessary sqlite3 statements to create the KGA database and populate it with at least three records per table. *Optional: create a test script in some language the instructor can run to verify the presence of the sample data in the tables and that links occur between tables that should be linked. E.g., if you have a project ID that links to two plant IDs, produce a test that will produce the names of the plants that appear. The test should include at least three joins.*

For an interesting intro to sqlite3, Mark Litwintschik has written a minimalist guide to sqlite3. Hacker News has a discussion of that tutorial here.

Exercise 1 Comments. Best Practices - use datatype and store datatype

- Tutorial website had topics and subtopics
- Quizzes to be offered had key terms ; indexed both the quiz and the topic the same way so they could be brought up together
- key terms may occur more than once (classification vs clustering problem)

- classic hierarchical vs flat clustering issue—should everything fit into one place in a hierarchy or should everything be tagged so that a tag may appear with more than one item (many solutions to this issue are hybrids)

four things to take into account: what the phone can do, what the user can do, what the user wants, and what the developer wants

What the phone can do includes: photos, sending photos, keep track of time and what day it is when somebody logs in and can monitor air pressure, notifications

What the user can do: share (era of social media), tap the screen, talk to the phone, and the most horrible of all—enter data on a teeny, tiny, little, bitty keyboard

What the user wants: plants to not die!!!! (minimize effort by user by notifications or whatever that is done strategically so the plant lives without extra fuss)

What the developer wants: a sequential linear model based on experts for growing plants

Tight deadline may prevent too much research. Algorithms are risky for the database designer.

Exercise 2. Improvised ETL. *Following is the original CSV challenge as it ran on GitHub:*

You got your hands on some data that was leaked from a social network and you want to help the poor people. Luckily you know a government service to automatically block a list of credit cards. The service is a little old school though and you have to upload a CSV file in the exact format. The upload fails if the CSV file contains invalid data. The CSV files should have two columns, Name and Credit Card. Also, it must be named after the following pattern:

```
YYYYMMDD.csv
```

The leaked data doesn't have credit card details for every user and you need to pick only the affected users. The data was published here: [Note: because of link rot, I've put the file in the Content > exercises section of myCourses. It appears as `data` because myCourses can't display the `.json` extension. –Mick] You don't have much time to act. What tools would you use to get the data, format it correctly and save it in the CSV file?

The original assignment was: I want you to undertake the same challenge documenting exactly how you do it in a plain text file called challenge.txt (any kind of code and comments may be in the file as long as it is sufficient to meet the challenge.

Note: Since you have already done the challenge in ISTE-422, your assignment is to do the reverse with the `mockData.csv` file on myCourses. Convert the fname, lname, and gender from CSV to JSON and fix any problems that prevent you from doing so. Be sure to rename the file as YYYYMMDD.json.

Following is some starter code that should get you going on this assignment. You should look up `readline` and `createInterface` to get a clear picture of what is going on here.

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
  terminal: false
```

```
});
console.log('[');
rl.on('line', (line) => {
  line = line.replace(/, ([JS])r/," $1r");
  arr = line.split(/,/);
  console.log(`"fname": "${arr[0]}"`);
});
```

There are no NULL records to remove in this assignment (in contrast to the original 422 assignment) but there are problems with the `mockData.csv` file to solve if you want full credit. You can most readily see these problems if you say

```
cut -f 5 -d , mockData.csv | sort | uniq -c
```

at a terminal prompt. You should see output like the following.

```
      1  Female
    487 Female
    438 Male
      1 bblackpz@imgur.com
     26 female
      1 gender
      1 jnelsonoa@liveinternet.ru
      1 jstonenv@umn.edu
     45 male
```

What this tells you is that there are different spellings of male and female and that three people have commas in their names. You need to fix these problems as part of your solution.

In other words, you will receive data with `Female`, `␣Female` (i.e., Female preceded by a space), `female`, `Male`, and `male`. Your JSON file should have only `Female` and `Male` for gender.

You will also have to solve the problem of extra commas in the names Jose Nelson, III, Judith Stone, Jr, and Billy Black, Jr. so that you list their genders instead of their email addresses. You'll get more credit for a more general solution than for a solution that only works for those three.

A final suggestion. I recommend that you look at the writings of Peteris Krumins, a prolific blogger on *one-liners*, small solutions to common problems. You should also consider looking at sites like commandlinefu for similar advice on small solutions to common problems.

You should also look at blog posts like *The Absurdly Underestimated Dangers of CSV Injection* at `http://georgemauer.net/2017/10/07/csv-injection.html` and the related Hacker News conversation.

Exercise 3. Stand Up An RDBMS. For this exercise, imagine you are in a new workplace that requires the use of PostgreSQL. You have a MySQL script called `standup.sql` available to load a database. *This MySQL script will not work in PostgreSQL as it is.* You must somehow get this database up and running in PostgreSQL without consulting any of your classmates. You may google for information on the public web but do not consult anyone who is working on or has completed this assignment, i.e., neither past nor present students of this course. You are free to use any method you identify but you must document how you do it.

If you use the current version of the virtual machine, PostgreSQL is already installed and you can use it with the client program `psql`. You can connect to PostgreSQL by using `psql` without a password as long as you are logged in

as `student`. When you are logged in to `psql` you should remember the following commands.

`\h` help on sql
`\?` help on PostgreSQL backslash commands
`\q` quit
`\l` list databases
`\dt` describe tables in the current database

Please verify that PostgreSQL is running and the database is loaded by answering the following questions using SQL. Include documentation of your method for loading the database, your SQL queries, and your results in a markdown file called `standup01.md`.

1. What is the most frequently mentioned first language?

2. What are the counts of all languages at all ranks? List them using language names and rank names (low medium high).

Note that you should not waste time trying to represent combinations that do not appear in the data. Think like the customer of the data.

3. For each language mentioned as having skill(value) HIGH, identify the most frequently named MEDIUM language. If you prefer, use more than one query but not more than two, i.e., feel free to create a table of languages in step 1.

For example, suppose nine people mentioned Java as HIGH. Among those people, suppose four mentioned C++ as medium, three mentioned Python as medium, and two mentioned Haskell as medium. Then the answer for Java would be C++. You must include an answer for each language mentioned as HIGH.

Please use fenced code blocks for your SQL. A fenced code block starts with a blank line, followed by a line containing nothing but three backticks in the first three columns, and immediately followed by the letters sql. Subsequent lines contain your sql code. After the last line of sql code, comes a line containing nothing but three backticks in the first three columns, followed by a blank line. For example,

````
```sql
select * from bla;
```
````

Exercise 4. Demonstrate Password Hashing. I would have called this exercise "write a password hashing program" except that you should not do that. You should download and compile an existing program and demonstrate that you can use it. That may seem trivial but take a look at the following question on Stack Overflow.

This person followed the excellent tutorial at hashing security which includes code you can download for Java, C#, PHP, and RoR.

The problem the questioner faced was that he did not understand what was downloaded. It would have been better to read and understand first, then demonstrate. Of course, the answer on Stack Overflow shows which paragraph this questioner did not read but what if you fail to read a different paragraph? Stack Overflow may not be able to keep up with all the problems / issues, so please go through the tutorial with me first.

My only quibble is the use of the words "completely unpredictable" which has been shown to be untrue given sufficient computing power and irrelevant if you can get away

with dangling the password owner from an 18th floor balcony (from a recent news item at officers-balcony).

Turn in a small test program that exercises one of the password hashing programs supplied in a link from the above tutorial. The test program should let me enter a password and display the hashed version of the password. It should be accompanied by sufficient instructions for me to compile and run it and a copy of the password hashing program. The instructions, which may be comments in the source file, should also identify where you obtained the password hashing library. Your solution should be something you can retrieve and use in a practical situation, months from now, after you have forgotten the details.

Exercise 5. Oral Presentation. Provide a brief (5 minute maximum) oral description of your project and answer impromptu questions about various aspects of the project. The questions may be directed at specific group members or it may be left to the group to select a spokesperson. In a live workplace situation, such a group would anticipate some obvious questions based on the details of the work they've done and would clearly be well prepared for some obvious questions which may or may not be asked. Such a group would also likely face surprising questions and would, in any workplace, be evaluated in part on how they handle surprising questions.

The questions may require you to walk through parts of your code, specifically with respect to the following requirements:

- layers: at least presentation, business, data
- logging: e.g., log4php, log4javascript
- caching: e.g., memcached
- authentication: restricted to presentation layer if poss, SSO if poss

- role-based authorization
- validation in the business layer
- clear delineation of business rules
- only data layer contains sql
- exception handling infrastructure
- test code
- refactored code
- deployment, packaging so that i can install it without help

SOFTWARE

The previous courses in the database sequence use MySQL. This course does not require any use of MySQL. MySQL is optional. The only required DBMS products are SQLite3 for the KGA assignment and PostgreSQL for the Stand Up assignment. A good source of information about MySQL and other DBMS products is `persona.com`. There was a comparison of MySQL to MariaDB and a lively discussion of that post at Hacker News.

Improvised etl (extract / transfer / load).

Vim.

- split
- vsplit
- folding
- regex
- global
- syntax highlighting
- hex edit
- dbext
- vimdiff

tmux / wemux.

- split-window
- select-window
- list-buffers
- copy-mode
- paste-buffer
- swap-pane
- select-pane - list-buffers
- copy-mode
- paste-buffer- list-buffers
- copy-mode
- paste-buffer
- resize-pane
- detach

shell utilities.

- bash
- autoconf
- automake
- awk
- bg
- cat
- chgrp
- chmod
- chown
- column
- curl
- cut
- diff
- df
- du
- echo
- fg
- find

- file
- fmt
- grep
- head
- httrack
- jobs
- join
- jq
- less
- ln
- make
- man
- paste
- printf
- ps
- pwd
- readline
- recode
- rsync
- sed
- sort
- stat
- strings
- tail
- tar
- tcpdump
- time
- top
- uname
- uniq
- uptime
- wget
- which

- whoami
- xargs

ldap utils (undecided).

- ldapadd
- ldapcompare
- ldapcomplete
- ldapexop
- ldapmodify
- ldapmodrdn
- ldappasswd
- ldapsearch
- ldapurl
- ldapwhoami

Standing up a dbms.

- PostgreSQL utilities

Scripting routine dbms work.

- Node.js

Embedded databases.

- sqlite3
- sqlite manager (for Firefox—undecided about this)

NoSQL.

- Strozzi NoSQL – Carlo Strozzi coined the term NoSQL many years ago to refer to a relational database operating without SQL. He periodically updates it so the current version is June 2014. I am undecided about it but, if I do use it, the installer has to know about a couple of pecularities concerning installation. I don't remember these but I used to use it so I should do a clean install before you try it.

- JSON support in JavaScript
- MongoDB

WRITING WELL

You must write well to succeed, both in this class and in your chosen field. It is not sufficient to claim that "this is not an English class" because every class is an English class, just as every class is a Math class and if your skills in these basic areas lag behind those of your contemporaries, they will lose faith in you. Currently, I recommend two sources to improve your writing, Garner (2009) and Fish (2011). At the very least, choose one book to study to improve your writing during each semester. You'll be glad you did when your cover letter is passed up the chain and the cover letter of a competitor is discarded for poor writing.

REFERENCES

Alexander, Christopher. 1977. *A Pattern Language: Towns, Buildings, Construction*. New York, NY: Oxford University Press.

Amirat, Abdelkrim, Anthony Hock-Koon, and Mourad Chabane Oussalah. 2014. "Object-Oriented, Component-Based, Agent-Oriented and Service-Oriented Paradigms in Software Architectures." In *Software Architecture 1*, edited by Mourad Chabane Oussalah, 1–53. London, UK: ISTE, Wiley.

Berenson, Hal, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. "A Critique of ANSI SQL Isolation Levels." In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1–10. SIGMOD '95. New York, NY, USA: ACM Press. https://doi.org/10.1145/223784.223785.

Date, C. J. 2004. *Introduction to Database Systems*. New York, NY: Pearson Education.

Fish, Stanley. 2011. *How to Write a Sentence*. New York: Harper Collins.

Flanagan, David. 2011. *JavaScript: The Definitive Guide*. Sebastopol, CA: O'Reilly.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Upper Saddle River, NJ: Addison-Wesley Professional.

Garner, Bryan A. 2009. *Garner's Modern American Usage*. New York: Oxford University Press.

Krasner, Glenn E., and Stephen T. Pope. 1988. "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80." *Journal of Object-Oriented Programming* 1 (3): 26–49.

Martin, Robert C., Dirk Riehle, and Frank Buschmann, eds. 1997. *Pattern Languages of Program Design 3*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Taylor, Richard N., Nenad Medvidovic, and Eric M. Dashofy. 2008. *Software Architecture: Foundations, Theory, and Practice*. Thousand Oaks, CA, USA: John Wiley; Sons.

Tow, Dan. 2003. *SQL Tuning*. Sebastopol, CA, USA: O'Reilly.

Wang, Ting, Jochem Vonk, Benedikt Kratz, and Paul Grefen. 2008. "A Survey on the History of Transaction Management: From Flat to Grid Transactions." *Distributed and Parallel Databases* 23 (3). Springer Nature: 235–70. https://doi.org/10.1007/s10619-008-7028-1.