BETA Exam Questions

Martin Geisler ⟨gimpster@gimpster.com⟩

January 20, 2003

Contents

These are notes for the questions for the aural exam in dPaSS, University of Aarhus, January 2003. The page numbers refer to

- *Object-Oriented Programming in the BETA Programming Language* ASIN: 0201624303 in questions 1–9,
- *Design Patterns* ISBN: 0201633612 in question 10,
- *Software Engineering* ASIN: 007050783X in question 11, and
- *Database Systems – The Complete book* ISBN: 0130319953 in questions 12–15.

1 The BETA object model

Including patterns vs. objects, object kinds, part objects, object references, pattern references (i.e. pattern variables) and computed references.

Plan

Cover the syntax of how patterns are declared in BETA and explain how BETA generalizes classes and procedures into the concept of a *pattern*. A pattern has a part with attributes which comes in four kinds: static and dynamic references, pattern declarations, and pattern variables. After the attributes you'll find an action part with `enter`, `do`, and `exit` parts. Explain how one accesses attributes of an object with *remote access* and *computed remote access*.

Describe how *static* objects are created when the surrounding object is created and belongs to that object from that point on, whereas *dynamic* objects are created explicitly by the program, see **page 31**. Combine this with *whole-part* and *reference* composition.

Chapter 11 from **page 155** deals with *pattern variables*. Show how they can be used to dispatch events to different procedure patterns as in Fig. 11.1, **page 160**, and how they can be used to infer subtype relationships between objects.

Describe the concept of object-oriented programming, how it is different from procedural programming, and the benefits you gain from the modularity and extensibility of the model. Also, explain how the BETA language is heavily based on a conceptual framework — the idea that a program should try and model the concepts in the real world as closely as possible. Fig. 18.1 on **page 286** explains how the modeling process should work.

With regard to concepts, talk about the *extension* (the set of phenomena covered by this concept), the *intension* (the set of common attributes for all phenomena covered by this concept), and the *designation* (the set of names that one can use to refer to the concept) of a concept, **page 19**.

Talk about ways to model the real world: *part objects* are used when we have an object hierarchy where each part belongs to some object, like when we model a stick man as Fig. 10.1, **page 146**.

2 The BETA execution model

Including execution control, parameter mechanisms, evaluations and equality, the inner mechanism and virtual procedure patterns.

Plan

Start by describing how all BETA programs are wrapped in an object descriptor and that it's the `do` part of this descriptor that is executed when the program is started. This generalizes to all object descriptors. Execution control flows from one `do` part to another, and passes through `enter` and `exit` parts in-between.

Describe the control structures in BETA: the `if` statement, the `for` loop, and the label commands. Combine this with the inner mechanism that allows you to build new control structures using singular inserted items that specialize a procedure pattern, **page 90**. These new control structures makes the built-in `for` loop unnecessary.

The concept of *virtual procedure patterns* is important to support specialization through inheritance, **page 107**. By making a pattern virtual in a superclass, you'll be able to *further-bind* the pattern in subclasses. This extends the functionality of the original pattern but keeps the name intact which is the key to general interfaces.

Classify evaluations as being either an *imperative* (`do X`), a *measurement* (`do X -> ...`), an *assignment* (`do ... -> X`), or a *transformation* (`do ... -> X -> ...`), **page 69**. Likewise for lists which are evaluated by evaluating each entry is some order. The comparison of lists define *value equality* in BETA: two objects are considered to be equal when the components in the lists from their `exit` parts are pair-wise equal, **page 72**. This is different from *reference equality* where you compare references to objects. Two objects can very well have the same *value* but different *references*.

3 The BETA inheritance model

Including attributes, qualification, polymorphisms, scope rules and specialization of action parts.

Plan

Short summary of how patterns are declared in BETA with attributes and procedure patterns, **page 28**. Then explain how to declare object inheritance in BETA.

Talk about how object-oriented modeling with inheritance is essential to the conceptual framework that BETA is based on because inheritance is used to model specialization in object hierarchies, **page 118** and Fig. 7.4.

We say that an attribute `V` is *qualified* by a pattern `P` if it's declared as `V:< P`. This indicates that the `V` attribute is a `P` or a subtype of `P`. This leads to *polymorphism* because we can now use an object of type `V` instead of type `P` at all places in our program, **page 121**.

The scope rules in BETA follow the block structure so that an attribute `V` declared in a pattern `P` is to all other objects declared within the object descriptor for `P`. Also, attributes declared in a superclass is visible in all subclasses. If the names collide, then it's the innermost attribute that is visible. The outer attributes can be access using the implicit `origin` attribute that always denotes the enclosing pattern, **page 74**.

If we have a dynamic reference to a superclass, then we can only access attributes of the superclass through that reference, even if reference points of a subclass with more attributes. The compiler will only allow you to call procedure patterns that it is sure exists in the object at runtime.

The action part of a pattern can be extended using the inner mechanism and subclasses. This makes it possible to build new control structures. An example with a `Cycle` pattern:

```
Cycle: (# do Loop: (# inner Cycle; repeat Loop; #) #);
```

4 The BETA virtual patterns

Including virtual procedure and class patterns, qualification and direct qualification.

Plan

BETA has a very general and powerful concept of *virtual patterns*. The same mechanism is used for both *class* and *procedure* patterns — it's only the way we use a pattern that indicate if it's a class or procedure pattern as it's really the same thing in BETA.

The concept of *virtual procedure patterns* is important to support specialization through inheritance, **page 107**. By making a pattern virtual in a superclass, you'll be able to *further-bind* the pattern in subclasses. This extends the functionality of the original pattern but keeps the name intact which is the key to general interfaces.

Another use of virtual patterns is in *virtual class patterns*. These are virtual patterns that are used more as a class instead of a procedure, **page 141**. Virtual class patterns can be used to implement generic data-structures such as stacks, queues, search trees, etc. The trick is to write a class that works using a virtual attribute qualified with a general type, preferably `Object`. Since every pattern in BETA is a subtype of `Object`, this ensures that the class will work with every pattern.

One can then take the generic class pattern and make a subtype that fixes the virtual type to the desired subtype of `Object` through a *final binding*. The result is a data-structure for which the compiler will enforce strict type-checking.

All virtual patterns are qualified by another pattern. It is this pattern that defines the minimum capabilities of the virtual pattern. Using a very general pattern like `Object` means that the virtual pattern can be further bound to anything, but also that you cannot be sure of it's capabilities. It's always the qualification that counts when the compiler is type-checks assignments.

The qualifying pattern can either be a singular pattern (*direct qualification*) as in `V:< (# ... #)` or another pattern as in `V:< P`, **page 113**.

Make an example with an *abstract factory* design pattern that can be specialized into making widgets for different operating system. This illustrates how one can use an abstract super pattern (perhaps called `WidgetFactory`) as an interface and then made concrete implementations as `GTKWidgetFactory`.

5 Deterministic Alternation and Coroutines

Including execution stacks.

Plan

For many simple problems a single thread of execution is sufficient, but when simulating problems that involve concurrency, or alternation between several tasks, it's important that the language has support for this.

In BETA this is realized through the use of several *execution stacks* or *coroutines*, which corresponds to activities that we want to alternate between. Each coroutine keeps track of it's own set of procedure invocations. The scheduling between the coroutines is non-preemptive, which means that control only goes from one coroutine to another with an explicit `suspend` imperative. This gives *deterministic alternation* because the successor is explicitly chosen by the programmer, **page 178**. It's also called multi-sequential execution.

The coroutines are associated with objects of kind *component*. An object of kind component can server as the base of an execution stack whereas objects of kind item cannot. An object of kind component is created with a `obj: @| P` or `obj: ^| P` declaration for static and dynamic references.

The execution of a coroutine `R1` is started when it is attached to the active stack of a running program, `P`. In doing this, the *return-link* of the component `R1` is switched with the return-link from the already running coroutine, and the *dynamic reference* now points to the top of the stack in `R1`. A coroutine is detached again when `suspend` is called, and this is also done by switching the return-links.

Example with a generator for pseudo-random numbers:

```
Random: (#
  r, min, max: @Integer;
  enter (min, max)
  do Cycle(#
    do ((12345*r + 17) mod (max - min)) + min -> r;
       suspend;
  #)
  exit r
#)

rand: @| Random;

do (10, 30) -> rand;
   rand -> putInt;
```

One can supply a list of evaluations for the `enter` part of a component upon attachment, these values will be available in the `do` part of the component. Likewise one can do a measurement on a component which will result in an evaluation of the `exit` part each time the component is detached. So each time we attach to `rand` and measure it, we'll execute the `do` part which results in a new random number and a call to `suspend` which in turn results in the execution of the `exit` part, giving us the new random number.

Finish off by talking about the antisymmetry in the way the coroutines are selected: it's the routine that attached another routine that gets to decide the next routine to attach. It's possible (with a little work) to implement symmetric coroutines where each coroutine selects the next routine to take over, **page 191**.

6 Concurrency: Semaphores, Monitors and Ports

Including basic concurrency and synchronization issues.

Plan

When modeling the real world, one often has to model actions that take place concurrently with other actions. This is supported in BETA using the `fork` imperative: if `S` is an object of kind component, then `S.fork` will start execution of `S` concurrent with the execution of the current component.

The concurrent execution of several components leads to problems with *synchronization* when they all access and modify a shared resource. The problem is that the state of the program can change at any time, so one cannot be certain that a condition holds, even if it has just been tested with an `if` statement. One needs to get exclusive access to the shared resource — this is handled through a *semaphore*.

Semaphores are implemented in BETA in a level below the normal language, they are a special construct for which the compiler guarantees certain behaviors. To gain access to a shared resource protected by a semaphore `S`, one must first call `S.P`. This will block the current component until the semaphore is unlocked by a call to `S.V`, **page 203**.

Describe how one uses a *monitor* to protect access to several procedure patterns in an object. Monitors can be extended with a `wait` pattern that describes a condition one will wait for, **page 208**.

For more advanced synchronization between components, one can use a *port* from the `System` pattern. A port has associated an `Entry` and an `Accept` pattern. A component makes a *request* by calling a procedure protected by the `Entry` pattern. This block until the receiver calls `Accept` on the corresponding port. This is called a *rendezvous*. The `Access` pattern in a port can be restricted to only allow access for a specific object, or from objects with a specific qualification.

When one wants to execute several components simultaneously but still wait for them all to finish, then one uses the `conc` pattern in `System` together with the `start` imperative, **page 216**.

To solve the problem with having *readers* and *writers*, one can define a `readerEntry` in the `Monitor` pattern. This is an entry that gives access to multiple objects as long as no one is executing another procedure pattern protected by an `Entry`, **page 220**.

7 Exception Handling

Including basic exception handling issues and specific BETA support.

Plan

Start by motivating the concept of exceptions. Exceptions occur when a program reaches a state where it cannot continue with normal execution. This could be a division with zero, popping off an item from an empty stack.

There's three goals when we want to handle exceptions: *meaningful errors*, possibly *recovery*, and *separation of control flow*, **page 239**.

These goals are reached in BETA through the use of virtual patterns. The predefined pattern `Exception` that helps here, **page 242**. This makes it easy to give a proper error message in the event of an exception, and it also forces a separation of the exception handling code from the normal control flow. Because exceptions in BETA are implemented as nothing more than procedure invocations, then it's possible for the handler to try and fix the problem. When the procedure returns, then the original code can move on as if the exception never occurred.

The default action of a handler that inherits from `Exception` is to terminate the program, but this can be changed by calling `Continue` in the handler. Exception handlers can be associated with both class patterns and procedure patterns or both. If the procedure level handler cannot handle the exception alone, then it can *propagate* the exception to the class level handler by invoking it, **page 245**. Automatic propagation is possible with the use of pattern variables:

```
A: (#
   Error: Exception(#   #)
   B: (#
      Error:< RealError;
      RealError: (#
         do (if Error## = RealError## then
                this(A).Error
            if)
      #)
   #)
#)
```

The use of virtual patterns makes the exception handling *static* compared to the more dynamic exception handling in language like C++, Java, and Pascal where the handler for an exception is determined dynamically at runtime.

8 Modularization

Including abstract datatype specification, program variants, and visibility and binding rules for fragment forms.

Plan

Start by explaining how large programs makes modularization necessary so that one can keep working with bits in a manageable size. Splitting a large program up into smaller pieces has a number of advantages as listen on **page 255**.

In BETA we work with *fragments* which together with a *name* and a *syntactic category* makes a *fragment-form*. These fragments are combined into *fragment groups*. A fragment group has an *origin* that specifies a fragment group with free *slots* for the fragment forms to be inserted into, **page 261**. It's possible to *include* other fragment groups in the current fragment group which gives full access to the new fragment forms, only restricted by the normal scope rules. If the implementation should be hidden, one can use `body` instead.

The scope for a fragment form is called it's *domain*. This includes the fragment itself, the domain of the origin and the domain of all included fragments. The domain will always lie within the *extent* of an fragment because body fragments also count in the calculation of the extent, **page 280**.

The relationships between the different fragments can be displayed in a graph where each fragment is a node. Each fragment has an thick arrow to it's origin, a normal arrow to included fragments, and a dotted arrow to body fragments. The domain for a fragment is now everything that can be reached by traversing thick or normal arrows, and the extent is everything that can be reached using any kind of arrows starting with the fragment.

The fragment system makes it possible to hide the implementation details in a body fragment. This fragment will be outside the domain, and can therefore not be manipulated by the program, except through the interface. By using different body fragments, it's possible to have several different *program variants* that all do the same, but with different implementations, e.g. a linked list instead of an array for a general container, **page 275**.

Every program in BETA is wrapped in an object descriptor like the one in `~beta/basiclib/betaenv` which provides access to the most often used patterns. One can see that the `Lib: Attributes` slot is in the domain of the program which means that library routines can be inserted here, **page 263**.

9 Conceptual Framework for Object-Oriented Programming

Including its relation to the BETA programming language.

Plan

The BETA language is built on the premise that the object-oriented way of writing programs is the best way to model the physical world. When we're writing programs that deal with entities from the real world, then it's important that these entities can be represented in the program. The goal is to end up with a program that is a faithful simulation of the problem-domain.

The modeling process starts when we identify the *phenomena* that makeup our *referent system*, that is, the real world. As we classify and group the phenomena in the referent system, we create concepts specific to our problem. It is these concepts that we model as *realized concepts* in our *model system*. In BETA this is done with patterns and hierarchies of patterns. The realized concepts (patterns) are finally turned into objects in our model. These objects should correspond closely to the phenomena in our referent system, see Fig. 18.1 on **page 286**.

The concepts formed from the referent system can be characterized by their *extension* (the collection of phenomena covered by the concept), their *intension* (the common properties of phenomena in the extension of the concept), and their *designation* which is the collection of other names by which the concept is known, **page 291**.

When deciding whether or not a given phenomena belongs to a certain concept, one can use several views. The two extremes are the *Aristotelian* and *prototypical* views.

The Aristotelian view says that properties in the intension of the concept can be divided into two groups: *defining* and *characteristic* properties and that one objectively can decide whether or not a phenomena has a given property, **page 292**.

The prototypical view says that the properties in the intension are merely *prototypes* and *examples* of properties that a typical phenomena covered by the concept can be expected to have. This gives a more fuzzy definition where one cannot determine objectively whether or not a given phenomena belongs to a given concept.

BETA has strong support for realizing the concepts using patterns. BETA supports whole-part and reference composition through static and dynamic objects, it has support for classification through inheritance.

The modeling process goes through three fases: analysis, design, and implementation. We start by finding and analyzing the phenomena that make up the referent system. The phenomena is often grouped based on a prototypical view because this is what we normally use when we generalize. These prototypical concepts has to be transformed into Aristotelian concepts that we can handle in a programming language like BETA. This is the design fase, where we turn the fuzzy prototypical concepts into concepts based on defining properties. The design in then realized through an implementation where we have to deal with the technical details necessary to get a running program.

10 Design Patterns

Including design principles.

Plan

Start by talking about what design patterns are and why people are using them. Design patterns give you a way to deal with situations that occur all the time, and for which people have found good, general solutions. A design pattern isn't meant to be something that you would copy directly, instead it's something that tells you how you can solve a problem in general terms, so you have to adapt it a little for each situation you use it in.

The design patterns in use today reflect the programming languages we have at our disposal. If we haven't had object-oriented languages, then there would have been a pattern called "Inheritance," but since this is part of those languages, there isn't such a pattern. So the design patterns are really just a collection of healthy *design principles*, then sum up how to do things, and thereby how not to do them.

There are a number of design principles that the design patterns will help one to use. First and foremost, then it's important to *program to an interface, not an implementation*. The reason is that the implementation should be allowed to change without affecting the rest of the program. This can only be done if the implementation is properly *encapsulated* and access is restricted to the interface.

The second principle is that one should *favor object composition over class inheritance*. When classes inherit from one another they become *tightly coupled* because a change in the implementation of a class can affect all subclasses. This is because the subclasses have access to (and therefore might depend on) the way the class implements it's interface. The subclasses aren't restricted to the narrow interface and this breaks encapsulation.

We have met a number of design patterns. The *Abstract Factory* design pattern illustrates the principle of programming to an interface. The idea of an abstract factory is that instead of creating objects from concrete classes all over the program, this job is handed over to the factory. This will then give you an object that is a subtype of an abstract interface — you don't need to know the exact subtype, all you need to know is the interface that the object implements. It can be difficult to extend the abstract factory with another product family, because this involves changing all the concrete factory classes since they have to implement the new method.

The *Proxy* design pattern describes how one can avoid the cost of creating large objects by using a place-holder instead. This is an example of *delegation* where the proxy steps in and takes over some of the responsibility of the object it's a proxy for. The proxy can do various optimizations such as delaying copying until a write is performed, or caching the object and enforce access policies. Give an example with a virtual proxy and a large playlist. It's easy for the proxy to read the header of each music file to determine the length and artist so the proxy only creates a real object when the file has to be played.

To illustrate the principle of decoupling the interaction between objects, we have the *Observer* design pattern. This shows how a number of objects can communicate with each other without knowing precisely who they're talking to. Use the Model/View/Controller architecture as an example.

11 Software Testing

Including test types and test programs.

Plan

Describe how it's necessary to test programs, especially with object-oriented design where most objects are created dynamically at runtime.

The different software tests can be divided into two broad categories: white- and black-box tests. A *white-box test* is a test where you have access to the implementation of a program. This means that you can design the tests so that every branch of the program is tested, and that you can make tests that deliberately tries to overflow buffers and other fixed-size areas of memory. These tests are very tightly coupled to the implementation so if that changes, then the tests will have to change as well.

A *black-box test* is the opposite: you only have access to the interface. These tests will try out all methods in the interface with both legal and illegal input to see if they behave correctly. These tests doesn't have to change if the implementation of the interface changes because the interface is the same, so it's easier to make and maintain these tests.

There are a number of different tests, depending upon which level the test is made on. A *unit test* is at the lowest level where you test a single unit (a single class) to ensure that it has the necessary functionality. This is done in isolation from other classes so that the programmer only needs to deal with his own code. These tests can be made in parallel by the different programmers, which speeds up development.

When the classes have passed a unit test, it's time to make an *integration test*. Here the classes are tested to ensure that they work properly together. This can be done by taking all the classes and see if everything works — this is called a *big bang* integration test — but this will usually lead to chaos. It's much better to test classes with each other as they are finished.

After the integration test it's time for a *system test*. Here the system is tested in a realistic environment to see if it conforms to it's specifications.

Throughout the development of the program, data should be gathered. This constitutes the *regression test*. This can be as simple as running all tests at night and then making a graph each morning that shows the number of failed tests, or it can be more detailed. By keeping results from previous tests, it's easier to see when something broke.

When the program is nearing it's completion it should be tested with the customer. An *acceptance test* should be made where the customer tries the program to see if it fulfills his list of requirements. It's important that this list is made early in the development process, preferably before any code has been written. The developer knows that his job is done when the program lives up to the demands on the list, and the customer knows that he will get a program according to the list. The customer isn't allowed to change his mind midways, and the developer isn't allowed to stop midway.

To get the best results the developers shouldn't test their own code, a *clean room* policy should be made. This means that the developers and the testers shouldn't discuss their respective problems with each other, because that can make them miss bugs if they start thinking the same way.

12 Data Models in Database Systems — Data Definition

Including entity-relationship diagrams, ODL.

Plan

Summary at **page 59** in the database book. Start by describing databases model groups data and the relations between these groups. In the *entity-relationship model* (E/R model) we group the data in *entity sets* which can have any number of *attributes*. These entity sets are related to each other with *relationships* which might also have attributes associated with them. A special relation is the *is-a* relation between super- and subclasses.

The attributes are normally simple data types such as integers, text strings and booleans. They describe the entity sets and thereby describe the data hold in the sets. The relationships shows how the sets relate to each other. Relationships can be both one-many, many-many, and many-one, where a one-many relationship between E and F indicates that you can find many entities in F that are related to a given entity in E.

We make *entity-relationship diagrams* when we design the structure or *schema* of our database. These diagrams use *rectangles* for entity sets, *diamonds* for relationships, *ovals* for attributes, *triangles* for is-a relations, and *lines* to connect relationships with entity sets. Some attributes of an entity set are special because they determine the identity of the entities in the set. These*key attributes* are underlined and entity sets that depends on keys from other entity sets, called *weak entity sets*, are indicated by a double line. The entities in these sets get their identity from the keys of the set itself together with keys from other entity sets reached through *supporting relationships* which also uses a double line, **page 54**.

A *normal arrowhead* from a relationship to an entity set E means that there's no more than one entity in E for each set of entities in the other sets in the relation. A *rounded arrowhead* means that there is exactly one matching entity, this is used to denote *referential integrity constraints*, **page 52**. We might also give the entities in a relationship *roles*, this is denoted by a label next to the line to/from the relationship.

When designing a database system using E/R diagrams, then it's important to be *faithful to the data* and to *avoid redundancy*. Redundancy is avoided by *normalizing* the relations, that is, splitting them up into several relations so that data is only represented once.

Another data definition model is the *Object Definition Language*. In this language, data is grouped into *classes* instead of entity sets. A class can have *attributes*, *relations*, and *methods*, **page 143**. Attributes are not limited to simple data types, but can also be *enumerations*, *structures*, or *collections*. The objects created from a class lie in the *extent* of the class which is typically named by putting the class name in plural: the extent of the class `Movie` is called `Movies`.

ODL only supports *binary (two-way) relationships* but any multi-way relationship can be turned into a two-way relationship by turning the relationship into a entity set and then make two-way relationships from this set to the other entity sets, **page 33**.

Methods are only declared in ODL, the implementation is handles in a host language. The declarations indicate the return type and arguments for the method. The arguments are specified as being either `in`, `out`, or `inout` where the last two types are passed by reference.

Object-oriented databases will understand schemata written in ODL directly but normal relational databases wont. The OQL language complements ODL as a query language.

13 Relational Database System

Including the relational model, normalization.

Plan

Start by describing how we normally use E/R diagrams to design our databases, but that no database can understand a E/R diagram directly, most databases use the *relational model* when dealing with data.

The core of this model is a *relation* which is a two-dimensional table with columns for *attributes* and rows for *tuples*. There are no ordering among the attributes and tuples because they're both *sets*. This implies that one cannot have two equal tuples in a relation because the tuples don't have any intrinsic identity. This is different from object-oriented databases where each entity has a unique identity.

The relational model is extremely popular because it's possible to implement it efficiently. Part of this efficiency comes from the fact that most databases treat the tuples as *bags* instead of sets, so that there can be several tuples with identical values in all attributes.

We normally want to *normalize* our relations to avoid redundancy and problems with our model. If the relations aren't properly normalized, then we will probably waste space and also see deletion abnormalities, where we cannot represent a studio if we don't have any films for that studio in the database.

We need the concept of a *functional dependency*, **page 83**. We write $A_1 \cdots A_n \rightarrow B_1 \cdots B_m$ when $B_1, B_2, \ldots, B_m$ is determined by $A_1, \ldots, A_n$. A *key* is a set of attributes that functionally determine the other attributes, **page 85**. A *superkey* is just a set of attributes that contain a key.

We have redundancy in our data if and only if we can find a subset $A_i \cdots A_j$ of $A_1 \cdots A_n$ which contains a key that isn't a key for the whole set of attributes. The problem is the data in attributes $A_i \cdots A_j$ can be duplicated because $A_i \cdots A_j$ isn't a key $A_1 \cdots A_n$, but at the same time, we know that some subset of $A_i \cdots A_j$ determines all of $A_i \cdots A_j$. So we can remove all attributes which isn't part of the key for $A_i \cdots A_j$ and then make another relation with A1···An where each tuple is unique. The attributes $A_i \cdots A_j$ cannot be removed from the original relation for they are used to identify the correct tuple in the new relation. To recreate the original tuple, a *natural join* is used between the new relations.

We say that a relation is on *Boyce-Codd normal form* if and only if the left side of every nontrivial functional dependencies contains a key, **page 105**. This means that a relation on Boyce-Codd normal form doesn't have any redundancy.

14 Query Languages in Database Systems — Data Manipulation

Including SQL, OQL.

Plan

Start by describing how we model our data using E/R diagrams and how these diagrams are translated into something that fits the relational model which our database can understand. Define the term *relation* as a two-dimensional structure with *tuples* as rows and *attributes* as columns. It is this structure that we manipulate using the *Structured Query Language*.

When manipulating data using we get the advantage of using a very *high-level language*. We don't have to worry about how the system stores the (potentially) many gigabytes of data, all we know is that it guarantees consistent and persistent storage. We would also like to be able to run many queries in parallel, and also be able to group queries into indivisible parts called *transactions*.

To fetch data with SQL we use the `SELECT` keyword:

```
SELECT a1, a2 ..., an
FROM R1, R2, ..., Rm
WHERE condition
```

One can think of the execution of a `SELECT` statement as first making a Cartesian product between all the relations mentioned in the `FROM` clause. This gives us a gigantic relation with all attributes found in the other relations. These tuples are then tested against the condition in the `WHERE` clause and finally the matching tuples are *projected* onto the attributes listed in the `SELECT` clause. If necessary, the attributes can be *qualified* with the name of the relation, or with a *tuple variable* declared in the `FROM` clause.

There are a number of different kinds of joins available. One can use a *natural join* to join two or more relations on their common attributes. If the attributes have different names, one uses `R1 JOIN R2 ON ⟨condition⟩` instead. These basic joins removes tuples without matches from either relation. To preserve these tuples one has to use an *outer join*. This kind of join comes three variants: *full*, *left*, and *right* outer joins. The first form inserts `NULL` values whenever a tuple doesn't match which means that it doesn't throw any tuples away. A left outer join will only insert `NULL` values for the right relation whereas a right outer join inserts `NULL` values there's no matching tuple in the left relation.

The tuples in a relation can be put into groups using a `GROUP BY` clause. When grouping on attributes a1, a2, ..., an, then they're the only attributes that we can use *unaggregated* in our `SELECT` clause. An attribute is aggregated when we use one of `COUNT`, `SUM`, `MIN`, `MAX`, or `AVG` on it. We're allowed to use any attribute in the `SELECT` clause in an aggregated form. The aggregation is done separately for each group of tuples.

Finally, we can specify a condition that the groups must meet with a `HAVING` clause. Here we can use attributes from the `GROUP BY` clause in unaggregated form and all attributes in aggregated form, just as with `SELECT`.

The `INSERT` statement is used to insert data into a relation. The values can be given as an explicit parenthesized list, or it can be produced by a subquery.

Values already inserted into a relation are updated with the `UPDATE` statement. This changes an attribute in a number of tuples selected by a `WHERE` clause.

Deletions are made with the `DELETE` statement. The tuples to be deleted are selected using a `WHERE` clause. This clause can involve subqueries or constants.

The query language used with object-oriented databases is *Object Query Language* or OQL for short. OQL looks and tries to behave very much like SQL does, **page 429**. In OQL we work with *collections* of *objects* instead of relations. The attributes of an object can be accessed using either a dot or an arrow. The objects created from a class lie in the *extent* of the class which is typically named by putting the class name in plural: the extent of the class `Movie` is called `Movies`. The extent is what we called the *extension* in the conceptual framework for BETA.

OQL looks very much like SQL with the change that it's now mandatory to declare variables to represent the objects in each relation:

```
SELECT s.name
FROM Movies m, m.stars s
```

Most SQL can be mapped directly into OQL with the exception of aggregation and grouping. OQL has the same five *aggregation* operators as SQL, but they now work on a bag of values. The `COUNT` operator accepts any bag, but the other operators requires a bag where each element is a single value.

Data can be *grouped* in a rather complex way. We can get a listing with the maximum budgets for each studio per year with a query like this:

```
SELECT year, studio,
  budget: MAX(SELECT p.m.budget FROM partition p)
FROM Movies m
GROUP BY year: m.year, studio: m.studio
```

The general structure of a query with a `GROUP BY` clause is this:

```
SELECT fi, ..., fj,
  avg: AVG(SELECT p.rx.ay FROM partition p)
FROM R1 r1, R2, r2, ..., Rn rn
GROUP BY f1: e1, f2: e2, ..., fm: em
```

Here the `GROUP BY` clause says that we build a structure with fields named $f_1, f_2, \ldots, f_m$. The fields get their values from the expressions $e_1, e_2, \ldots, e_m$. The structure has an extra field called *partition* which is a bag of structures with fields $r_1, r_2, \ldots r_n$. Each structure in the bag contains the objects that gives the same result when evaluated on $e_1, e_2, \ldots, e_m$.

The result of the whole expression is a bag of structures. We create the fields in the `SELECT` clause: each structure has fields $f_i, \ldots, f_j$ and `avg`. The `avg` field is an aggregation of values from objects from the partition bag. All the objects $p.r_1, \ldots, p.r_n$ gave the same result (stored in $f_1, \ldots, f_m$) when the expression $e_1, \ldots, e_m$ were evaluated. We can now aggregate attributes of these objects.

OQL supports some additional constructs for dealing with collections. The `FOR ALL` and `EXISTS` boolean operators works like their mathematical counterparts, **page 437**.

When using OQL with an object-oriented host language we no longer have to deal with the *impedance mismatch* problems that the relational model suffer from. The `SELECT` statements in OQL return collections of objects, and with the proper extension of the host language, we should be able to assign the result of such a query directly to a variable in the host language, **page 444**.

15 Using Relational Database Systems in Object-oriented programs

Including embedded SQL and mapping from E/R to relations

Plan

Databases are typically designed using E/R diagrams, so it's important to convert these into a relational model. This is done by creating a relation for each entity set with the necessary attributes. E/R notation has the is-a relationship to make subclasses. There are three different techniques for converting these into relations. The *E/R-style conversion* uses a relation for each subclass, **page 77**. These relations will then have all the keys from a class and it's parent classes as key attributes along with any new attributes. To fetch all data for a given entity, one has to visit several relations. This can be done with several *natural joins*.

The *object-oriented approach* creates relations for each possible subtree of a root class, and stores the data for a given tuple in a just one of these relations. This gives a lot of relations and makes it difficult to find a tuple when the type is unknown. But we only use a single tuple for each entity in our database so this uses the least amount of space, **page 78**.

The third option is to use *null values to combine relations*. This means that there will only be a single relation, but it will be filled with holes. The advantage is, that it's faster to select data from just one relation. The type of a tuple is determined from the amount of null values present, **page 79**.

Relationships are converted into relations where the keys from the entity sets in the relationships are used as attributes in the relation. With something-one relationships it's often possible to combine the relations into a single relation. This is important when dealing with *weak entity sets*.

These are always connected with a many-one supporting relationship to another entity set. We don't have to make a relation for the supporting relationship, instead we can place it in the same relation as the weak entity set, **page 71**.

When we want to use a database system form a conventional *host language*, we're faced with an *impedance mismatch* meaning that the relational model doesn't fit the way we normally manipulate data in our programming languages. Our host languages deal with data in terms of integers, chars, arrays, records, and so on whereas the relational model deals with tuples and relationships. This problem is solved when using object-oriented languages with ODL and OQL.

The connection from our host language to the database server is made through the use of *call-level interface*. One can use this these library routines in two ways. We can use a preprocessor which allows us to embed SQL statements in our code and use shared variables to move data back and forth or we can use the functions directly to send statements to the server.

When using *embedded SQL*, a preprocessor is necessary to make the code correct in the host language, Fig. 8.1 on **page 351**. The SQL statements are wrapped in suitable functions from a SQL library supplied by the DBMS vendor. Part of this process is the handling of *shared variables*. These variables are accessible in the host language, but also in the SQL statements, because the preprocessor takes care of expanding them when they occur in embedded SQL statements, **page 353**. The preprocessor is probably turning embedded SQL like this

```
BEGIN EXEC SQL
INSERT INTO :table VALUES(:a, :b)
END EXEC SQL
```

into something like this for a program written in C:

```
if (snprintf(buf, size, "INSERT INTO %s VALUES(%i, %i)",
             table, a, b))
  SQLSTATE = SQLExecute(buf)
```

To move through large result sets a *cursor* is used. This is an iterator that can traverse through a list of tuples. Scrolling cursors can more freely through the tuples, **page 361**. Data is inserted and fetched from the relation using the cursor and shared variables with special calls, **page 356**. Some DBMS support *server-side cursors* which means that the server only sends data back to the client as it scrolls through it. If this isn't supported, then the client could select the data in small bites using a `LIMIT` clause in the SQL statements. In BETA we scroll through result sets like this:

```
query.execute -> result[];
result.Scan(#
```

```
  current.s -> ...;
  current.i -> ...;
  current.i -> ...;
#);
```

An alternative to using embedded SQL is to access the database directly through the *call-level interface*, i.e. using functions from a SQL library. This lets the programmer do the job of the preprocessor. By using the standard SQL/CLI library and using standard SQL, our program should be able to work with different databases as long as they all implement SQL properly, **page 385**.