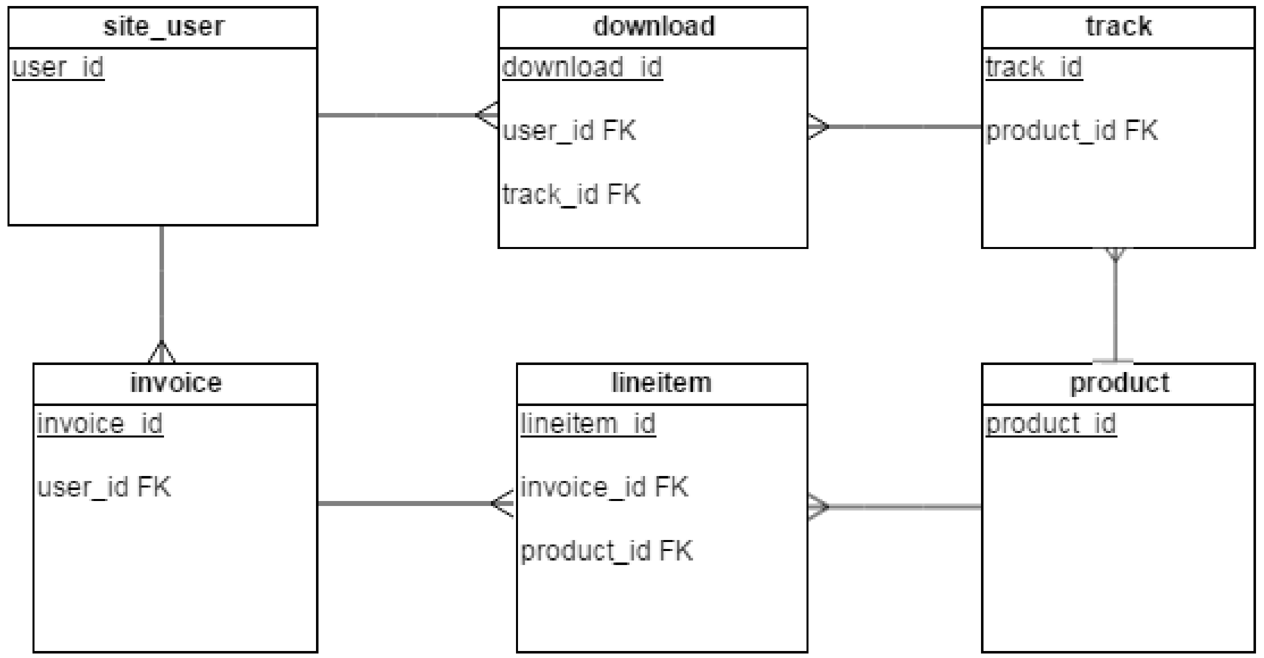
Intro to the Music Project

Last time: Music DB
E-R diagram in crow's foot notation



- This diagram shows the FKs determining how the tables are related.
- It shows a "crows foot" at the "many" end of the one-to-many relationship, where we would put a star in the UML-style diagram. This is another common notation.

The Music project

Last time, defining domain data, designing the service API to allow a stateless service layer by making each service method "self-contained".

"self-contained" method: all the domain data needed for the method's code is provided in the method's arguments. Nothing is built incrementally in the service layer.

So we expect this in music1’s service API, which you will specify as part of the project.

The domain classes are given in music1-setup, and some of the needed DAOs.

Idea of app: selling CDs for a band. CD=Product

Directory/Package Structure: parallel to pizza1

```
├───database
│   └───lib-for-ant
├───sound
│   ├───8601
│   ├───jr01
│   ├───pf01
│   └───pf02
├───src
│   ├───main
│   │   └───java
│   │       └───cs636
│   │           └───music
│   │               ├───config
│   │               ├───dao
│   │               ├───domain
│   │               ├───presentation
│   │               └───service
│   │                   └───data
│   └───test
│       └───java
│           └───cs636
│               └───music
│                   └───dao
└───target
```

The database createdb.sql, etc.

The sound files, .mp3's

Source packages as in pizza1, plus subpackage of service for transfer objects

A few DAO Junit5 tests

Sound files: each album has a web page

There is one sound.html file for each CD. Example:

https://www.cs.umb.edu/cs636/music1-setup/sound/pf01/sound.html

Note that the sound.html files list more tracks than we have in the database, but we have the ones with mp3s. See Murach, pg. 657 to see his directory structure: we’re taking his sound directory...

From Murach’s distribution: one directory for each CD, using the filesystem for decent organization of data.

Note that this is all read-only data, so doesn’t need to be in the database as much as changeable data does.

In music1, not easy to actually play these from Java (need special library), so just print out the filename as "playing".

Music1 service API

PA1: you need to design the service API. The presentation code is sketched out, but not how it calls down. This is a crucial part of database app design. Each call should be “self-contained” in the sense we discussed last time. You can use the service API for pizza as a model. HW3 has some practice on API design.

Note that Murach talks about the layers (see pg. 17), but in fact does not supply a service API, so you need to design one yourself.

Lack of service API in Murach: Look at Murach, pg. 654 in CatalogController, presentation code. This project is now available to you in ../murachMusicStore from your cs636 directory. From outside, you could use the full filespec of the zip file:

/courses/cs636/s19/eoneil/murachMusicStore.zip to get the project.

Murach's CatalogController is at src/music/controllers/CatalogController.java in that project. Here the email of a user is found in a cookie, then calls directly into UserDB, a DAO (implemented by a static class rather than a singleton).

```
            user = UserDB.selectUser(emailAddress);  //call from presentation code directly into UserDB, a static
class
```

We want to have presentation code call a service API, and let the service method call the DAO.

Music1 domain and transfer objects

Recall PizzaOrder vs. PizzaOrderData

in music:

User for core code, UserData for transfer to presentation.

In this case, User.java has many more fields than UserData. User.java is really meant to have all the fields shown on pg. 663, but we've simplified it down to the basics.

Clearly fields like creditCardNumber should be kept as secret as possible.

Although User is immutable in pa1, it has the potential to be mutable as a user address is added to a record of an already-registered user, so we will treat it as mutable.

Music1 domain and transfer objects

- Invoice for inside service API (where this object is mutable: we mark an invoice as "processed"), InvoiceData for returning to presentation: this is much simpler than Invoice with referenced User and LineItems.
- Download vs. DownloadData: also simpler.
- Product, Track: invariant, "reference data", use for both inside and outside. This app does not support adding or changing a Product or a Track.
- Of course in reality we could bring the app down, add a Product to the database, and bring it up again. But during operation, the set of Products and the Product details remain fixed. So it's OK to let these objects flow into the presentation layer--note there is no Product-related transfer object defined.
- Note that Product and Track have getters but no setters for their properties, as consistent with their invariant status. So objects returned to presentation are protected against modification.

User Interface

- music1 = client-server implementation, like pizza1, so boring line-oriented UI.
- But aimed at web site, with proper web pages, as shown in Murach. Let's look at them. See **Music Project UI and Page Flow** (for user pages)
- Look at a page in the Page Flow: Catalog page for example

From pa1.html:
The Catalog page, also shown on pg. 645, displays the catalog, i.e., the list of CDs. The user can choose a particular CD (for example product code 8601) and get more info on it, or display the current cart (Cart page, see below) or go back to the Home page.

- This means the user is given choices of what to do next: get more info on one CD (chosen), see the current Cart, or go back to the beginning state. We can do this in simple line-oriented UI, as done in the provided skeleton UserApp.java.

UI in supplied UserApp.java

- Note that in general, it is difficult to convert a web UI into a line-oriented UI because a web UI has an arbitrary graph connecting different executable units, whereas a normal program has a tree of calls of different executable units.
- On the other hand, a tree-like web site is user-friendly, so many websites are basically treelike.
- The Music UI is organized in a tree-like way: Look at the page flow for music and see the tree.
- UserApp: supplied code, you just add calls to your service layer.
  - Note that it combines the Product and Sound pages into one method processProduct.
  - This makes sense when you note that both these pages need to allow the user to add to cart and display cart and get back to Catalog.
  - You should not have to change the control structure of the supplied UserApp.java.

What about the Cart for music?
It’s domain data not held in the database…

- First note that a Cart is specific to a user, i.e., user-private data, not shared data among users.
- Following Murach, we’re not saving carts to the database.
- They just go away when the user buys some CDs or abandons the process.
- The resulting Invoice object from the purchase is of course put in the DB, along with its LineItems.
- But carts are domain data. They could be saved to the DB, and many actual sites do this, for logged-in users (we need the userid of the login to be a key in the database).
- In our case, the Cart, private to the current user, needs to be saved in the presentation layer
- That is, there is a *field* holding a Cart object in the presentation layer class but no such field in the service layer class (that would be state in the service layer, against the Stateless Service Layer principle)
- See UserApp.java for field cart (commented out for now.)

Carts in action

The service layer can still act on Carts: we just have a method with a Cart argument.

We want to put all the important code for the app in the service layer, not hidden in the presentation layer, because different UIs could talk to the user, but they all should work with Carts the same way.

Note we want to do the needed “new Cart()” in the service layer, since that action is needed whatever the UI is, by the Thin Presentation Layer rule.

Yet we need to *hold* the resulting Cart object in a field in the presentation code.

No problem: Cart createCart() in the service API, result saved in presentation. In UserApp, we would have cart = xxService.createCart() (or whatever you want to call it)

Similarly, code to add an item to a Cart should be in the service layer.

Cart details

- Carts need to record user selections: 2 copies of CD pf02 for example and 1 copy of jr0.
- So need a set of CartItems, where CartItem has product info and quantity.
- Note that the Cart is examined in the presentation layer and printed out to the user.
- Various possibilities exist here: see current setup uses product code and quantity, but it would be possible to use a Product object given that it is immutable now.
- Note that the product code is a “natural key”, a unique id for the product with a meaningful name. It would be possible to use the PK integer id, but it is less meaningful.
- If the presentation layer wants more details about the product, they should be able to call a service layer method to get them, using an argument of the key.

Presentation variables in Music holding domain data across calls to the service layer

- Currently commented out in UserApp.java: fields holding domain data, all specific to the current user. You will need to uncomment these when filling out UserApp.java.

```
private UserData user;  // once registered, non-null

private Cart cart;  // the CDs selected so far by the user (registered or not),
                    // but not yet bought
```

- We see that the Cart object belongs to the presentation layer, but the *action of creating a Cart* should be done by the service layer, in a "Cart createCart(…)" call. We want the service layer to do all the important actions of an app.
- Also, inside processProduct, local variable product holds the current Product being examined by the user, across calls to the service layer.
- So the Product object saved in presentation layer does live a while. It represents a *specific user's* choice, i.e., user-private data. We could switch it to key value instead, especially recommended if we change Product to be mutable.

Relationships among the domain objects for Music

Last time we looked at relationships among the domain objects for Music, and found mostly unidirectional relationships (in terms of inter-object refs), many of which follow a FK in the DB.

The exception was a bidirectional relationship between Product and Track.

We found state held in the presentation layer across multiple calls to the service API : Cart, User, Product objects. Cart cart and UserData user are fields of UserApp, Product is a local variable of processProduct that holds state *across multiple calls to the service API*, so is "longer term" state in the system. Note that these are all user-private variables, as are all variables defined in the presentation code.

Two kinds of domain objects

- Persistent domain objects, whose data is tied to database data. For example a Product object is tied to a certain row of the products table.
- Memory-only domain objects, so non-persistent, like Cart. There is no cart table in the database (but there could be). If a user abandons a cart, it is eventually garbage-collected and forgotten by the system.

Summary on domain objects

- They hold “domain data”, though not the only way domain data can be held (ex: int currentDay in Pizza)
- Domain objects move across the layers, carrying domain data where it’s needed.
- As domain data, they are not stored (held in fields) in the service layer (or DAO) from one service call to another (“stateless service layer”)
- Domain objects (persistent ones) hold data from DB as “scratch copy”, or data bound for DB.
- Domain objects can hold domain data not from the DB, or only partially from the DB. The other data source is the user.

Summary on domain objects

- Domain objects are normally not long-lived : they carry fresh data from DB, whenever service layer needs data. Most of them live only for one service call.
- Exception: Product object in UserApp, representing user choice, but note that Product objects are invariant. OK for presentation layer to hold (invariant) domain data for a user.
- Domain objects can be mutable, like PizzaOrder, or immutable/invariant, like PizzaTopping. It depends on what the app is designed to do.
- Invariant domain objects are OK to return to presentation code as is.
- When the app wants to return the data of a mutable object to the presentation layer, it’s best practice for it to create another “transfer” object, e.g., PizzaOrderData for PizzaOrder. PizzaOrderData has all the data of PizzaOrder, but no methods to change the data. Some transfer objects have only a subset of the data.
- Domain objects are POJOs, informal “Java Beans”, with “properties”, available via getters and possibly setters. Reference:Java tutorial at http://docs.oracle.com/javase/tutorial/javabeans

DAO design: problem of inter-related objects

- How much should we fill in of the objects hanging off the one we're focused on?
- Recall all the things that can hang off an Invoice object: User, several LineItems, each LineItem refs a Product. Each Product refs all its Tracks
- Example: need to return a PizzaOrder from DAO, should it have all its Toppings hanging off of it?
- We do have the option of leaving the ref to the Set<Topping> null, saying no details are available
  - This should be made clear in the header comment of the DAO method.

See 2 finders for PizzaOrders

```
        Set<PizzaOrder> find OrdersByRoom(…) – full details: Toppings, PizzaSize
        Set<PizzaOrder> find OrdersByDays(…) – no details! P.O. object has null size ref, null
                                                 Set<Toppings>
                                               (only used by code that doesn't need details)
```

You might say “what a kludge!”

- Clearly this lack of clarity on what exactly to return from the DAO muddles the separation of the layers’ responsibilities.
- Wouldn’t it be nice to not have to decide between full information (with all that database access, slowing things down), and fast access of basic info?
- That’s part of what object-relational mapping offers. You only retrieve the PizzaOrder info and that PizzaOrder object is returned by the DAO.
- But that PizzaOrder object cleverly detects accesses to its ref to Toppings (while within the service layer) and fills them in then, only when needed. “lazy loading”
- The domain objects seem to be the same old POJOs, but in fact they are “managed” by the runtime system of the object-relational system.
- This is possible by code-rewriting of the getters of the POJO by “bytecode enhancement”.

Music project object graphs

Last time we worked on this--

A Download object has a User object and a Track object (providing the song info), and that points to a Product

Download → Track → Product

            → User

Put ovals around these class names to make a better picture.

Let’s Check the source files to see these refs set up:

Download.java:

```
public class Download implements Serializable {
private static final long serialVersionUID = 1L;
private long id;
private User user;  ← ref to User  (later see getUser())
private Track track;  ← ref to Track (later see getTrack())
…
```

Checking out the domain relationships

Working on object graph:

Download → Track → Product

→ User

Track.java:

```
public class Track {
private long id;
private Product product;   ← ref to Product
private int trackNumber;
private String title;
private String sampleFilename;
```

Product is just being ref'd, doesn't have any relevant refs for this. Similarly User.

Invoice object graph

```
Invoice -> User
        -> Set<LineItem> → LineItem -> Product -> Set<Track> -> Track
                                                               -> Track
                         → LineItem -> Product -> Set<Track> -> Track
                                                               -> Track
```

Note that the Set is represented by the HashSet or TreeSet object in the actual object graph

Part of Invoice.java:

```
private User user;   ← ref to User
private Date invoiceDate;  ←Date is an object, but not a domain object
private BigDecimal totalAmount;
private boolean isProcessed;
private Set<LineItem> lineItems; ← Set of LineItems
```

Null refs to indicate missing data

- Earlier we saw how pizza1’s DAO in some cases returns incomplete object graphs, e.g. PizzaOrders without PizzaToppings, to avoid some processing costs.
- Similarly you can return Invoice objects with null refs to LineItems if that suffices for some action needed by the service layer, or LineItems with null refs to Product.
- Note that if we had bidirectional relationships everywhere, we would have even more objects ref’d from each object, and it would be hard to know where to draw the line and null-out further refs.
- So that’s an additional “cost” to additional relationships in the domain classes: needing to decide where to cut them off.

Building new domain objects

Look at Download: We don't see new User() or new Track()

- Instead, see setters for User and Track: DI for Download creation

Look at Invoice: We don't see new HashSet() or new TreeSet(), or new User()

- Instead see constructor:

Accept User object

```
public Invoice(long id, User u, Date d, boolean isProc,
                          Set<LineItem> items, BigDecimal totAmount) {
              invoiceId = id;
              user = u;
              invoiceDate = d;
              isProcessed = isProc;
              lineItems = items;
              totalAmount = totAmount;
       }
```

Accept already-formed Set

There are setters as well. Again see proper DI (dependency injection)

Check for news in domain

```
 F:\cs\cs636\music1-setup\src\main\java\cs636\music\domain>grep new *.java
 Cart.java:     * Construct a new Cart to hold items
 Cart.java:                     items = new HashSet<CartItem>();
 Cart.java:                             items = new HashSet<CartItem>();
 Download.java:                 downloadDate = new Date();
 LineItem.java:                 BigDecimal total = product.getPrice().multiply(new
BigDecimal(quantity));
```

- Only see one creation of a domain object: CartItem
- Could be refactored.
- Could argue that it's a completely private detail of Cart, not a dependency that ever would be needed to unit-test Cart.

Do we need to destroy domain objects?

- No, they don’t have any refs to long-lived objects like Connection or file handles.
- We just let Java garbage-collect them, typically at the end of a service call.

Typical lifetime of a domain object (mutable ones anyway):

1. Creation in DAO in a DAO finder, representing a scratch copy of database data
2. Use in service layer to do the current action, i.e., service call
3. Loss of inbound refs at end of service call leads to garbage-collection

Or

1. Creation in service layer, argued to DAO create-call or update call or delete call
2. Later return to service layer, lose inbound refs at end of service call, get GC’d.