Databases 2

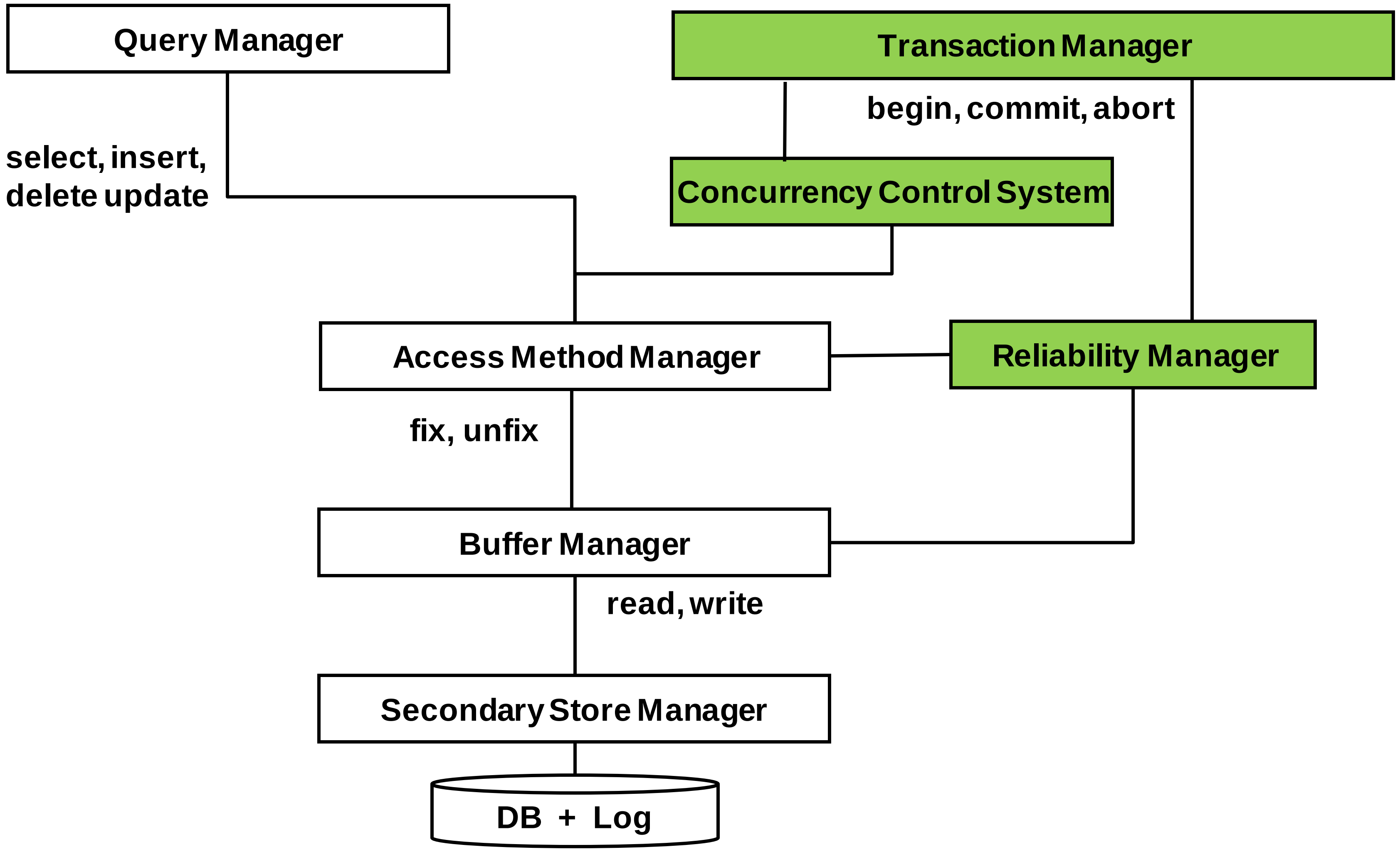
1

Transactional Systems

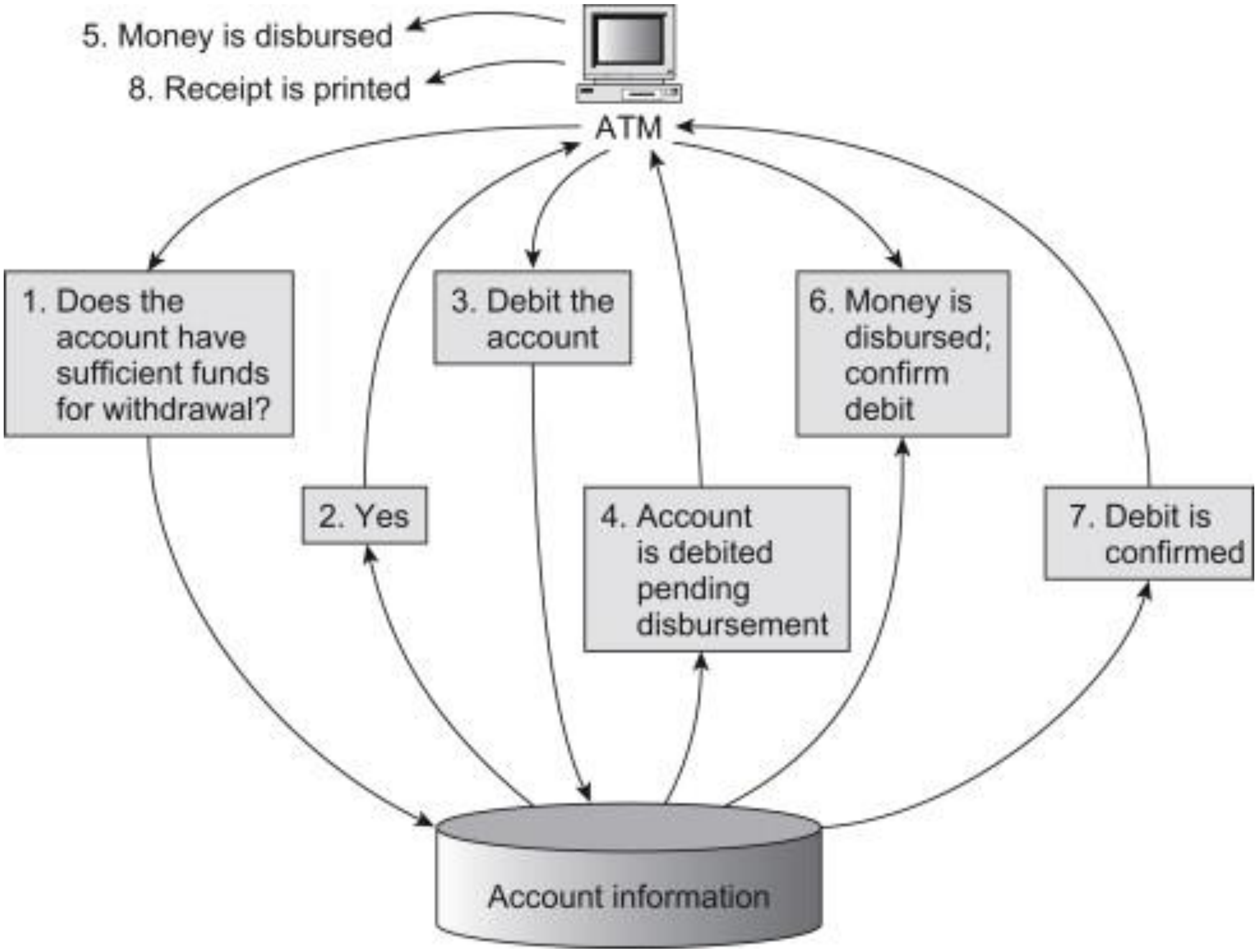
In this lecture

- The architecture of the DDBMS
- Introduction to the concept of *transaction*
- *Properties* of a transaction
- Modules of the DBMS responsible of guaranteeing such properties

The architecture of the DBMS



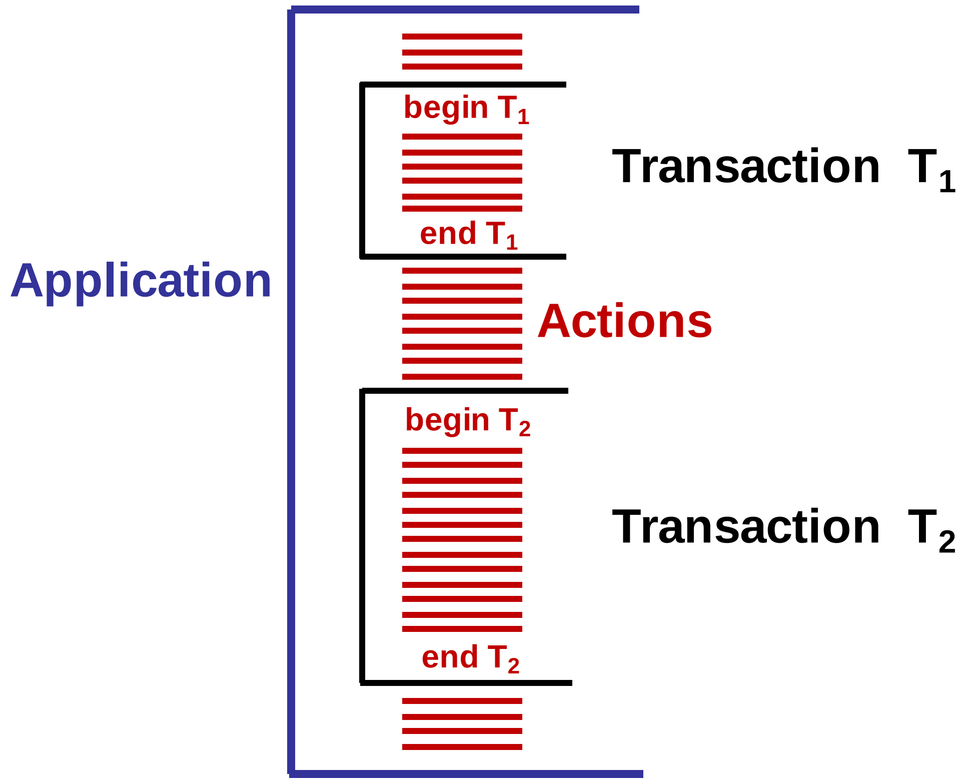
The need of transactions: ATM example


5. Money is disbursed
8. Receipt is printed
ATM
1. Does the account have sufficient funds for withdrawal?
2. Yes
3. Debit the account
4. Account is debited pending disbursement
6. Money is disbursed; confirm debit
7. Debit is confirmed
Account information

Definition of Transaction

- An elementary, atomic unit of work performed by an application
- Each transaction is conceptually encapsulated within two commands:
  - `begin transaction` (bot)
  - `end transaction` (eot)
- Within a transaction, **one** of the commands below is executed (exactly **once**) to signal the end of the transaction:
  - `commit-work` (commit)
  - `rollback-work` (abort)
- **Transactional System** (OLTP): a system that supports the execution of transactions on behalf of concurrent applications

Difference between Application and Transaction



Transactions: an example

```
begin transaction;
update Account
   set Balance = Balance + 10 where AccNum = 12202;
update Account
   set Balance = Balance – 10 where AccNum = 42177;
commit-work; // end transaction
```

Transactions: an example with alternatives

```
begin transaction;
update Account
   set Balance = Balance + 10 where AccNum = 12202;
update Account
   set Balance = Balance – 10 where AccNum = 42177;
select Balance into A from Account
   where AccNum = 42177;
if ( A >= 0 ) then
   commit-work; // end transaction with success
else
  rollback-work; // end transaction with failure
```

ACID Properties of Transactions

- A transaction is a unit of work enjoying the following properties:
    - Atomicity
    - Consistency
    - Isolation
    - Durability

1 Transactional Systems

Atomicity

- A transaction is an indivisible unit of execution
  - **Either all** the operations in the transaction are executed **or none** is executed
- In the case of the bank transfer, the execution of a single update statement would be disastrous
- The time in which COMMIT is executed marks the instant in which the transaction ends successfully:
  - An error before should cause the rollback of the work
  - An error afterwards should not alter the effect of the transaction
- The ROLLBACK of the work performed can be caused
  - **By the application** with a ROLLBACK statement
  - **By the DBMS**, for example for the violation of integrity constraints or for concurrency management
- In case of a rollback, the work performed must be **undone**, bringing the database to the state it had before the start of the transaction
- It is the application's responsibility to decide whether an aborted transaction must be redone or not

Consistency

- A transaction must satisfy the DB **integrity constraints**
  - **if** the initial state S_0 is consistent
  - **then** the final state S_f is also consistent
  - *This is not necessarily true for the intermediate states S_i*
- Example
  - The sum of the worked hours per task should equal the planned work hours of the project
    - If the constraint holds before the transaction it must hold also after its execution
    - The constraint can be temporarily violated during the execution of the transaction, e.g., when shifting work from a task to another one, but must be satisfied at the end

Isolation

- The execution of a transaction must be independent from the concurrent execution of other transactions
- In particular, the concurrent execution of a number of transaction **must produce the same result as the execution of the same transactions in a sequence**
- E.g., the concurrent execution of T1 and T2 must produce the same results that can be obtained by executing one of these sequences
  - T1,T2
  - T2,T1
- Isolation impacts performance and trade-offs can be defined between isolation and performance

Transactional Systems

Durability

- The effect of a transaction that has successfully committed will last "**forever**"
  - Independently of any system fault
  - Sounds obvious… but every DBMS manipulates data in *main memory*

Transaction Properties and related mechanisms

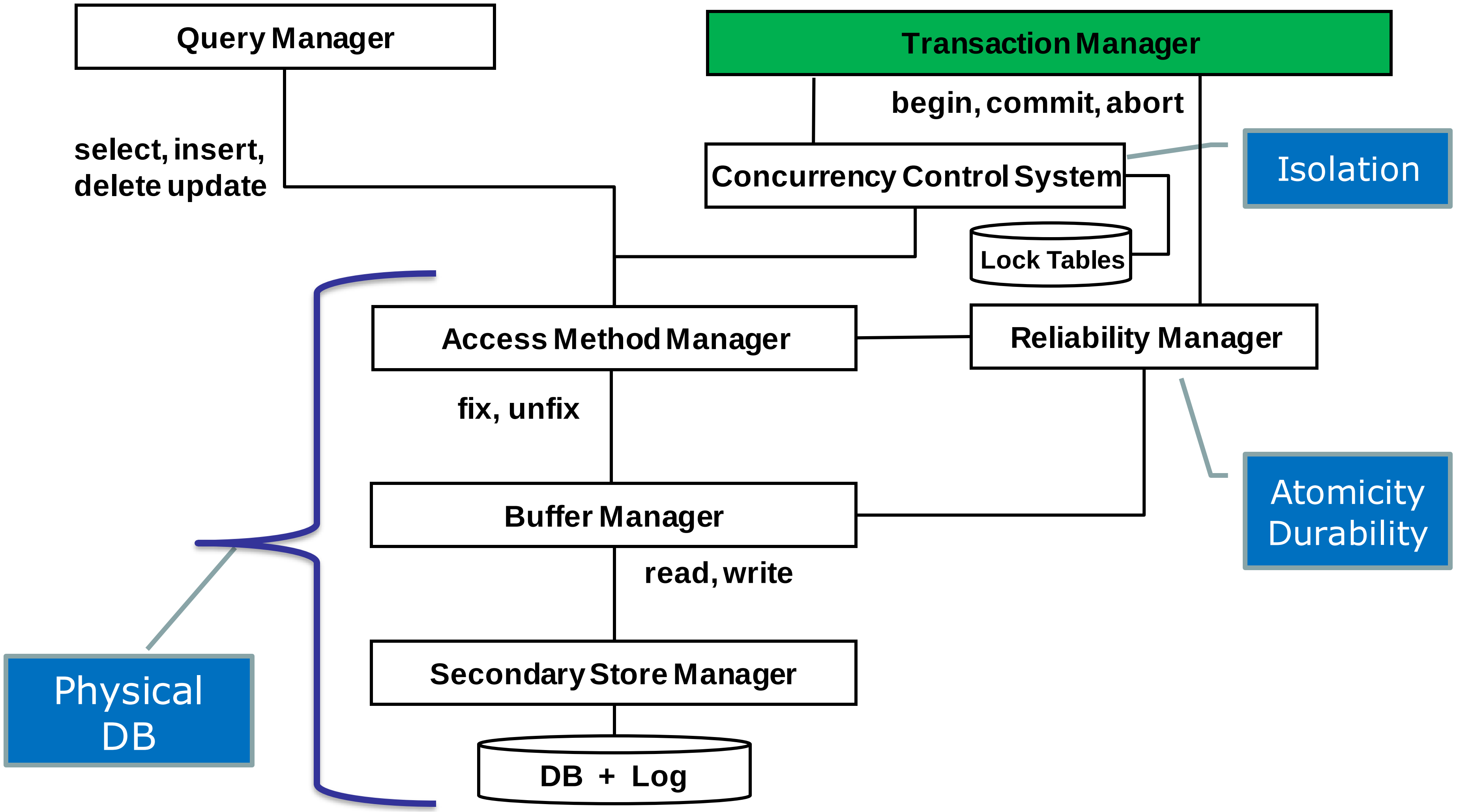
- **A**tomicity
  - Abort-rollback-restart,   Commit protocols
- **C**onsistency
  - Integrity checking of the DBMS
- **I**solation
  - Concurrency control
- **D**urability
  - Recovery management

ACID properties

Transactions and DBMS modules

- Atomicity and Durability
  - Reliability Manager
- Isolation
  - Concurrency Control System
- Consistency
  - Integrity Control System at query execution time (with the support of the DDL compiler)

Transactions in the DBMS architecture



Transaction management in Java & JDBC

```
public void addExpenseReport(ExpenseReport expenseReport, Mission mission)
                              throws SQLException, BadMissionForExpReport {
//Check that the mission exists and is in OPEN state
  if (mission == null | mission.getStatus() != MissionStatus.OPEN) {
          throw new BadMissionForExpReport("Mission cannot introduce expense report"); }
   MissionsDAO missionDAO = new MissionsDAO(connection);
   String query = "INSERT into expenses (food, accom, transp, mission) VALUES(?, ?, ?, ?)";
   // Delimit the transaction explicitly
    connection.setAutoCommit(false);
    try (PreparedStatement pstatement = connection.prepareStatement(query);) {
           pstatement.setDouble(1, expenseReport.getFood());
           pstatement.setDouble(2, expenseReport.getAccomodation());
           pstatement.setDouble(3, expenseReport.getTransportation());
           pstatement.setInt(4, expenseReport.getMissionId());
           pstatement.executeUpdate(); // 1st update
   // 2nd update to change the status of the mission, in a separate component
   missionDAO.changeMissionStatus(expenseReport.getMissioId(), MissionStatus.REPORTED);
   connection.commit();
    } catch (SQLException e) {
            connection.rollback(); // if update 1 OR 2 fails, roll back all work
                              throw e;
          }
 }   finally { connection.setAutoCommit(true); }
}
```

Transaction management in JDBC: issues

- This style of programming transaction is known as “programmatic demarcation”
- The programmer writes explicit code to start / end the transaction
- Issues:
  - What happens if I forget to handle an exception?
  - What happens if a component A starts a transaction and then calls a component B that also starts a transaction?
    - Error? Nested Transactions? Joined transaction?
  - How can I determine if a transaction is active when a component is executed?