LECTURE 12

THREE-LAYER ARCHITECTURE DESIGN

Ian Stavness, CMPT 270, 2017-18 Term 1

Abstraction

- eliminate irrelevant details
- ignore relevant details that aren't needed for the current task
- generalize the problem to handle more situations and allow its reuse

Decomposition

- Most software systems are very complex.
- Often the key to solving a complex problem is **decomposition**, i.e., divide-and-conquer approach
  - Divide the complex problem into simpler subproblems.
  - Solve the subproblems, i.e., for each subproblem, develop a module to handle it.
  - Combine the results of the subproblem solutions to solve the original problem.

  Note that the subproblems are solved by recursively applying the approach, until problems are obtained that are simple enough to directly solve.

Decomposition

- Use ***abstraction*** to ignore the details about how a subproblem is solved
- Need a precise **specification** of **what** the subproblem does/should do
  - develop the specification for each subproblem when subdividing
  - reason about the subdivision using the tentative specifications
  - use the final specifications to combine the subproblem solutions
  - use the final specification of a subproblem when solving the subproblem

Decomposition

- For a method, the specification consists of
  - signature
  - precondition, including conditions that can throw an exception
  - postcondition, including the return value (if one)

Guidelines for decomposition

- Locality
- Strong cohesion
- Weak coupling

Locality

- Keep related things as close together as possible, usually in the same module
  - related things should be local, close by

  e.g., Control variable for a loop declared in the loop

  e.g., local variables of a method or constructor

  e.g., a class encloses data and the operations on that data

Strong Cohesion

- Everything in a module should be related to the same purpose, i.e., work together
- If not, split up the module

  e.g., no method should be both an accessor and a mutator
- Note that there are different types of cohesion according to how the parts work together

Weak Coupling

- Minimize interaction between modules

  e.g., no method should have a large number of parameters,

  although a constructor may have many parameters

- If too much interaction between modules, consider merging them or parts of them
- There are different types of coupling according to different types of interactions that can occur

How to decompose?

1. read the specification for what command/operation to do
   - e.g., an item is selected from a menu
2. determine the command to be done
3. if necessary, read in the data for the command
4. execute the command
5. output any results

How to decompose?

- Execution of a command consists of a sequence of steps
- Some possible actions for a step:
  - access an object in the system
  - access information from the object
  - calculate some new values
  - modify something in the object
  - remove the object from the system
  - create a new object and save it in the system
  - return something

How to decompose?

- What classification of objects/classes would a command be?

  entity container control interface

- What classifications of objects/classes might be used by a command?

How to decompose?

- Separate the I/O from the rest of the system
  - Different types of devices have very different ways of doing I/O
  - Forms the interface for the system
- Separate the command determination from the rest of the system

How to decompose?

- Separate the steps for one command from those for another command
  - traditionally, use a different **method** for each command
  - recent OO approach, use a different **class** for each command
    - each command is in a clearly identified location, its own class
    - easy to add new commands or remove unwanted commands; add or remove a class
    - in addition to a “do” method for a command, might also have “undo”, “redo”, etc
    - possible to save commands for later execution
    - possible to send a command to a remote location for execution there
  - The approach of using an object for a command is the basic idea behind the *Command Pattern*

Three-Layer Architecture

- Presentation Layer
- Domain Layer (*or* Logic Layer)
- System Layer (*or* Data Layer)

  - Each entity/subsystem only knows about certain other ones (minimize coupling).
  - Each entity should have its own abstract specification that specifies what it can do.
  - How an entity does its job is hidden from the rest of the system.

Presentation Layer

- userInterfaces
  - Handles input/output with the users
- startup
  - Initializes the system:
    - Activates the creation and initialization containers
    - Activates the login of users
- controller
  - Determines the next operation to be done
  - Obtains parameter values for the operation
  - Executes the operation
  - Handles the result of the operation

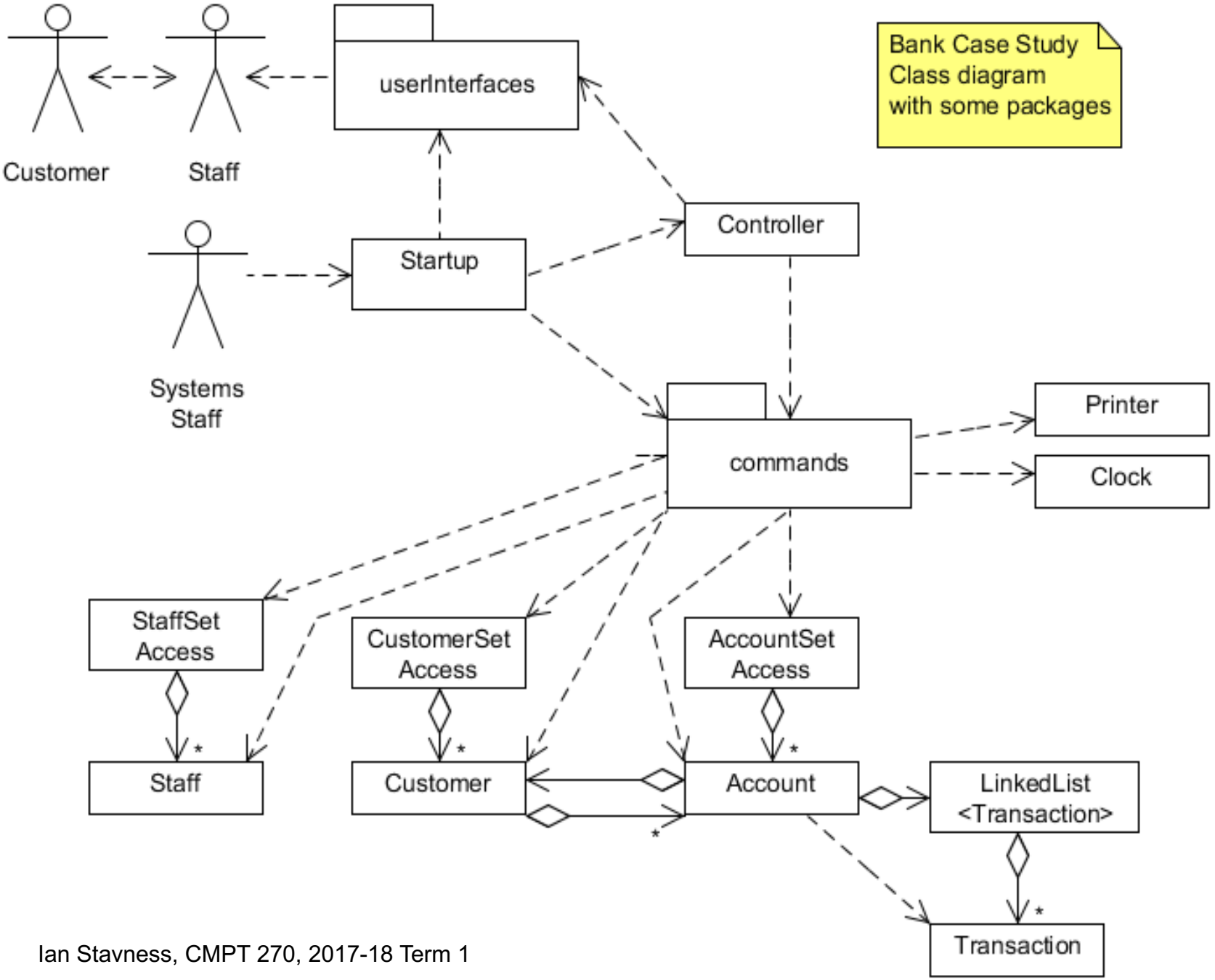
Domain Layer

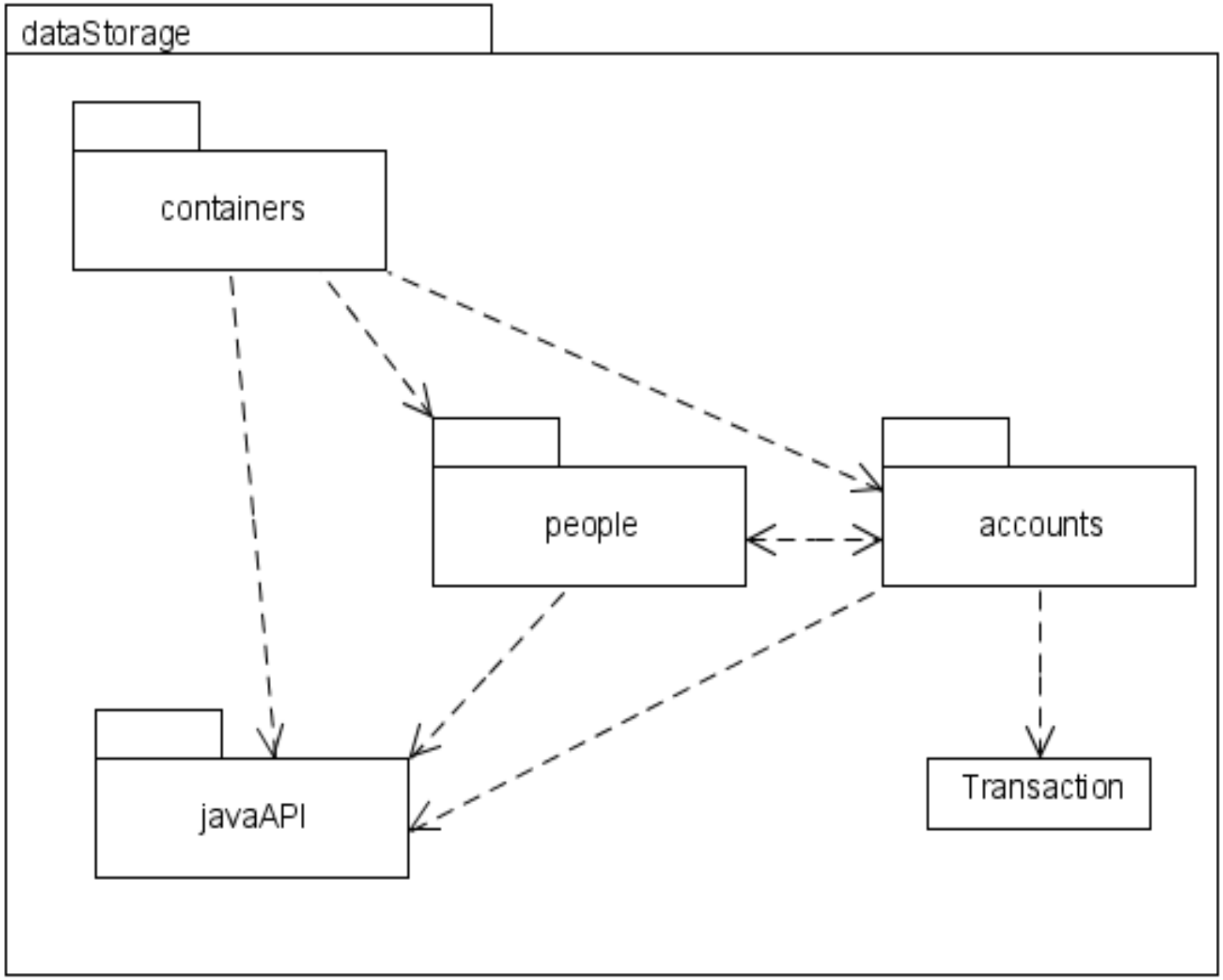
- commands
  - The operations of the application with a separate class for each operation
- dataStorage
  - The enities of the system
  - The containers storing the entities

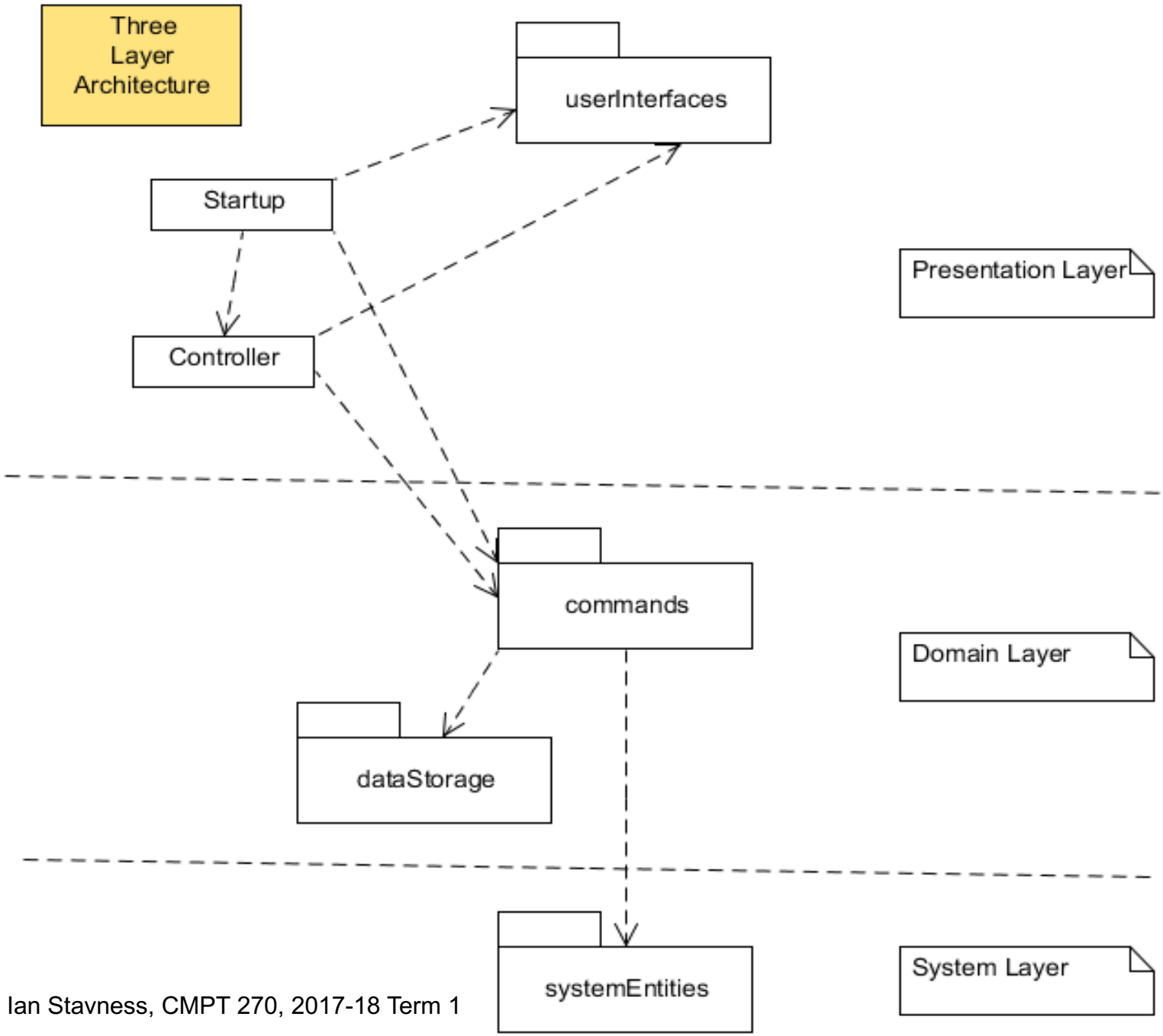
System Layer

- systemEntities
  - Interfaces to the facilities of the operating system

    e.g., clock, printers, files, databases


Bank Case Study
Class diagram
with some packages
Customer
Staff
userInterfaces
Controller
Startup
Systems
Staff
commands
Printer
Clock
StaffSet
Access
CustomerSet
Access
AccountSet
Access
Staff
Customer
Account
LinkedList
<Transaction>
Transaction


dataStorage
containers
people
accounts
javaAPI
Transaction


Three Layer Architecture
userInterfaces
Startup
Presentation Layer
Controller
commands
Domain Layer
dataStorage
systemEntities
System Layer

Controller – Command interaction

- Normally, use call-return interaction
- Problem, how can errors be handled? Standard Java approach, use exceptions.
- But, we want to allow remote or delayed invocation.
- Solution:
  - construct the command object
  - invoke a mutator method to do the command
    - pass in any parameters
    - Upon completion, the results or error message are stored in fields of the command object
  - use accessors/observers to access the results or an error message from the command object

  Note: define a special command class, CommandStatus, to record the success state of a command execution. It will be the ancestor of any command that can fail.

Controller main loop

- Using the interface obtain the id of the next command
- Determine the command selected
- Using the interface obtain any needed parameter values of the correct type
- Create the command
- Execute the command
  - If successful

    access the results from the command object

    handle the results, e.g., display the results using the interface
  - Else

    access the error message, and display it using the interface

Command

- Check the parameters for valid values
- Carry out the command
  - If successful

    store the results in appropriate fields
  - Else

    store the error message in its field

```
public class CommandStatus
{
        /** Specification of whether or not the command was successfully executed. */
        protected boolean successful = false;

        /** If the command was not successful, an appropriate error message. */
        protected String errorMessage;

        public boolean wasSuccessful()
        {
                return successful;
        }

        /** @precond ! wasSuccessful()
          * @return the errorMessage  */
        public String getErrorMessage()
        {
                if (wasSuccessful())
                    throw new RuntimeException("The last execution must have been "
                                + "unsuccessful in order to retrieve its error message.");
                return errorMessage;
        }
}
```

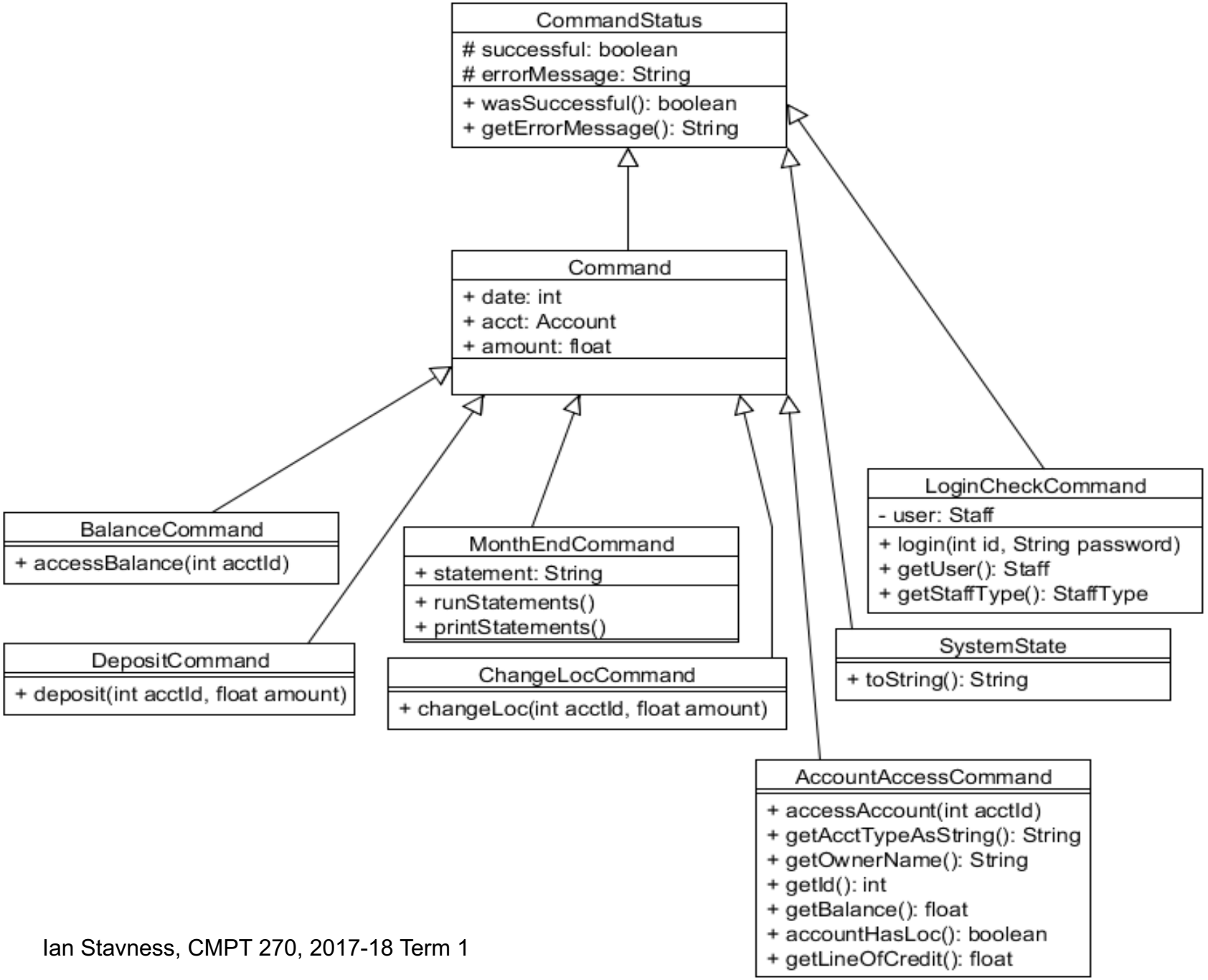
```
package commands;

import accounts.Account;

/**
 * This class contains attributes that are common
 * to all commands, except LoginCheckCommand.
 */
public class Command extends CommandStatus
{
        /** Date of the command. */
        public int date;

        /** The main account pertaining to the command. */
        public Account acct;

        /** The amount that the command involves. */
        public float amount;
}
```


CommandStatus
successful: boolean
errorMessage: String
+ wasSuccessful(): boolean
+ getErrorMessage(): String
Command
+ date: int
+ acct: Account
+ amount: float
BalanceCommand
+ accessBalance(int acctId)
MonthEndCommand
+ statement: String
+ runStatements()
+ printStatements()
LoginCheckCommand
- user: Staff
+ login(int id, String password)
+ getUser(): Staff
+ getStaffType(): StaffType
DepositCommand
+ deposit(int acctId, float amount)
ChangeLocCommand
+ changeLoc(int acctId, float amount)
SystemState
+ toString(): String
AccountAccessCommand
+ accessAccount(int acctId)
+ getAcctTypeAsString(): String
+ getOwnerName(): String
+ getId(): int
+ getBalance(): float
+ accountHasLoc(): boolean
+ getLineOfCredit(): float

```
package commands;
import containers.AccountSetAccess;
import systemEntities.Clock;
import systemEntities.Printer;
public class BalanceCommand extends Command
{
        public void accessBalance(int acctId)
        {
                date = Clock.getDate();
                acct = AccountSetAccess.dictionary().get(acctId);
                if (acct != null)
                {
                        successful = true;
                        amount = acct.getBalance();
                        // print receipt
                        Printer.printBalance(acct.getNumber(), amount, date);  // print receipt
                }
                else
                {
                        successful = false;
                        errorMessage = "Account number " + acctId + " not found.";
                }
        }
}
```

Small part of Controller class:

```
switch (cmdId)
{
        case 11:    balanceInquiry();          break;
        case 12:    deposit();                 break;
        case 21:    changeLoc();               break;
        case 22:    doMonthEnd();              break;
}

protected void balanceInquiry()
{
        int acctId = staffInterface.getAcctID();
        BalanceCommand balCommand = new BalanceCommand();
        balCommand.accessBalance(acctId);
        if (balCommand.wasSuccessful())
        {
                staffInterface.sendMessage("The balance is "
                    + NumberFormat.getCurrencyInstance().
                                                format(balCommand.amount)
                    + ".\nReceipt being printed.");
        }
        else
                staffInterface.sendMessage(balCommand.getErrorMessage());
}
```