Design Patterns & Software Architecture

Memento

dr. Joost Schalken-Pinkster

Windesheim University of Applied Science

The Netherlands

The contents of these course slides is (in great part) based on:
Chris Loftus, *Course on Design Patterns & Software Architecture for NEU*. Aberystwyth University, 2013.
Jeroen Weber & Christian Köppe, *Course on Patterns and Frameworks*. Hogeschool Utrecht, 2013.
Leo Pruijt, *Course on Software Architecture*. Hogeschool Utrecht, 2010-2013.

Session overview

- Memento

Memento design pattern

Let’s find a design pattern

Will now present, on the board,
and using Eclipse,
a solution that utilises
the memento design pattern…

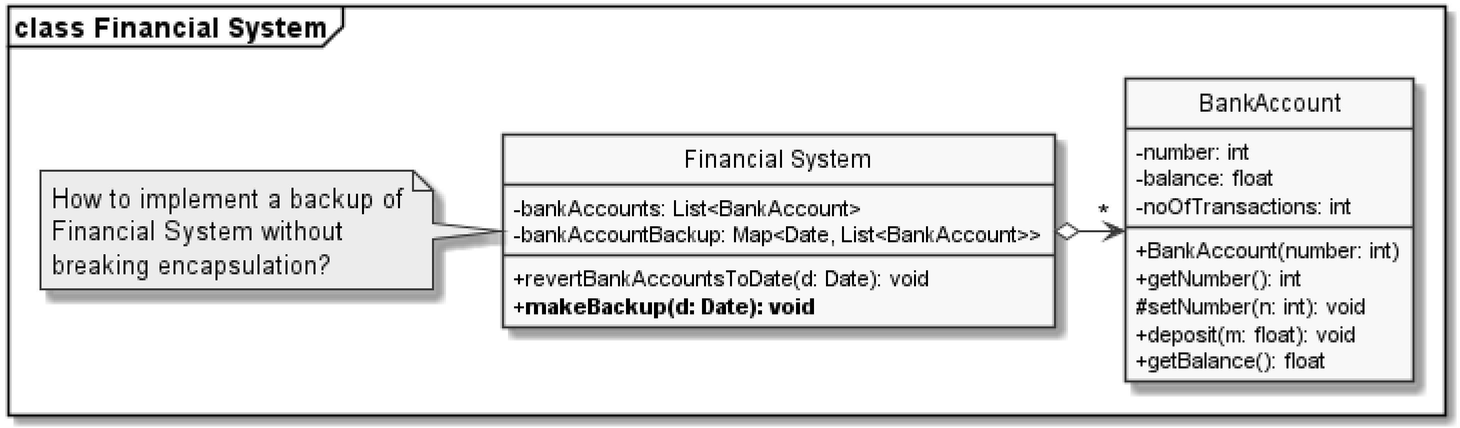
Case: Financial System Requirement

You have been asked to make a financial transaction system

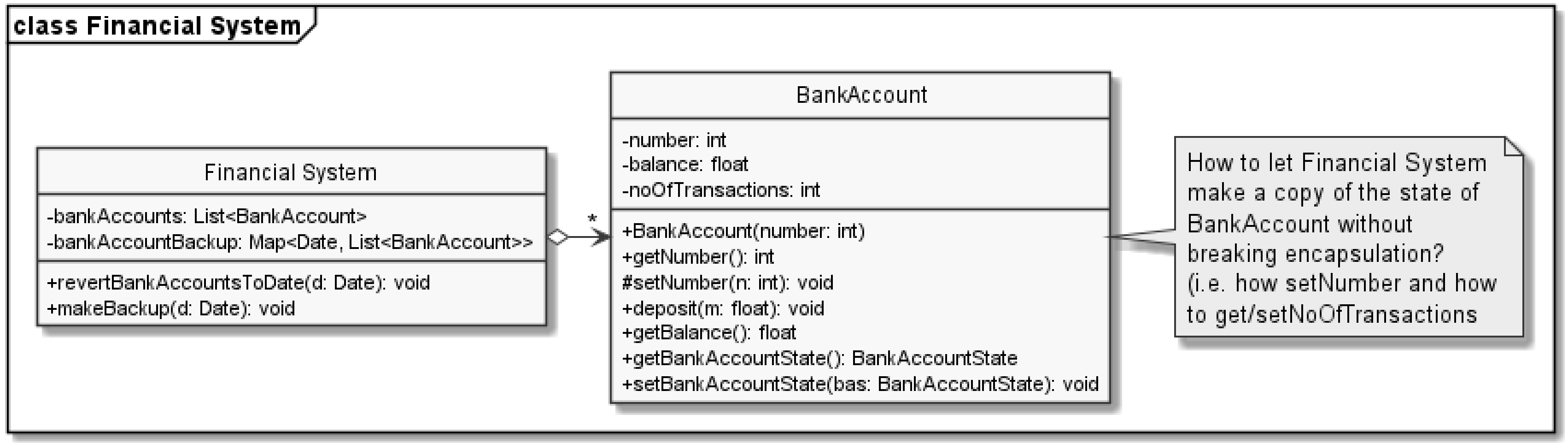
- The system should have an overall record of all BankAccounts
- BankAccounts should have a number, balance and transaction number count.

- It should be possible to let the financial transaction system rollback all changes to bank accounts (as part of a backup procedure).
- The backup procudure should not break the encapsulation of the BankAccount objects, as this would be a security risk.

Case: Financial System
Design 1

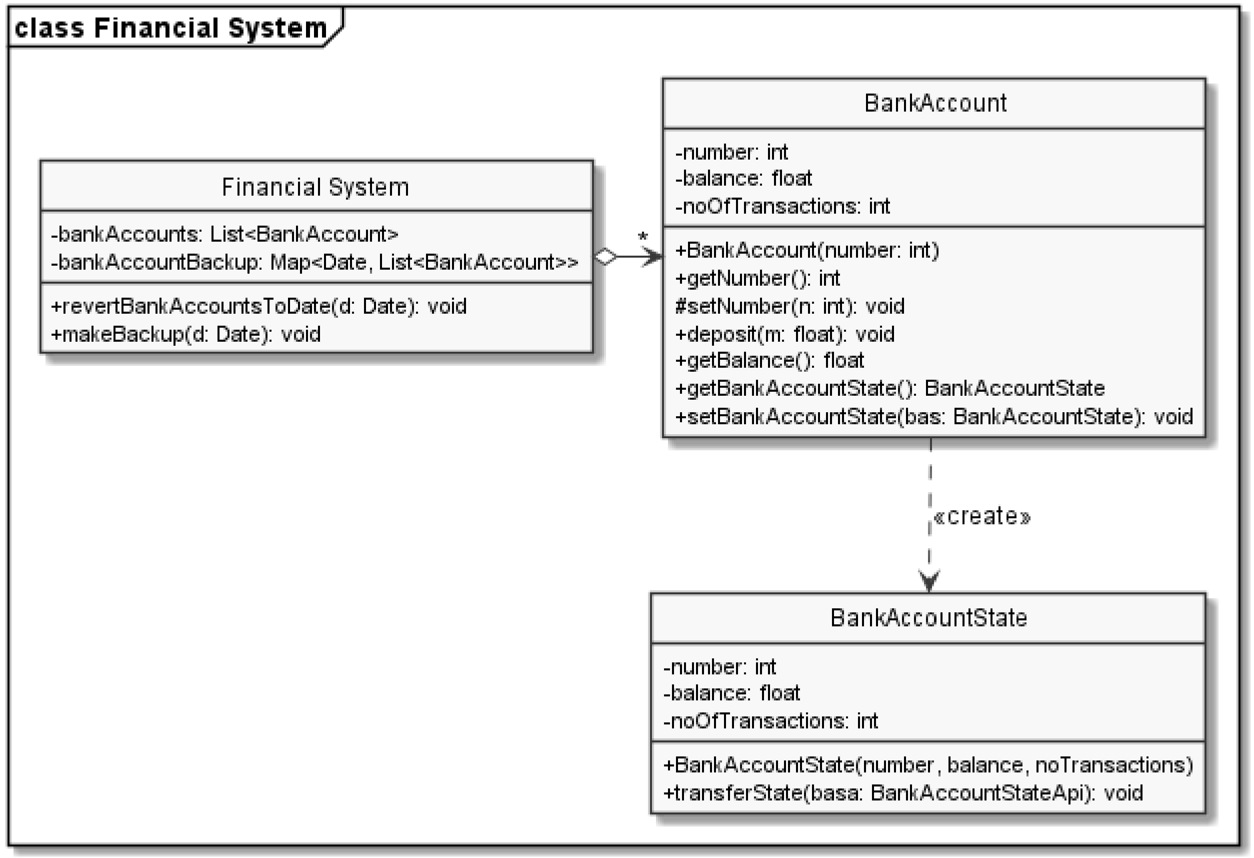

class Financial System
How to implement a backup of Financial System without breaking encapsulation?
Financial System
-bankAccounts: List<BankAccount>
-bankAccountBackup: Map<Date, List<BankAccount>>
+revertBankAccountsToDate(d: Date): void
+makeBackup(d: Date): void
*
BankAccount
-number: int
-balance: float
-noOfTransactions: int
+BankAccount(number: int)
+getNumber(): int
#setNumber(n: int): void
+deposit(m: float): void
+getBalance(): float

Case: Financial System
Design 1

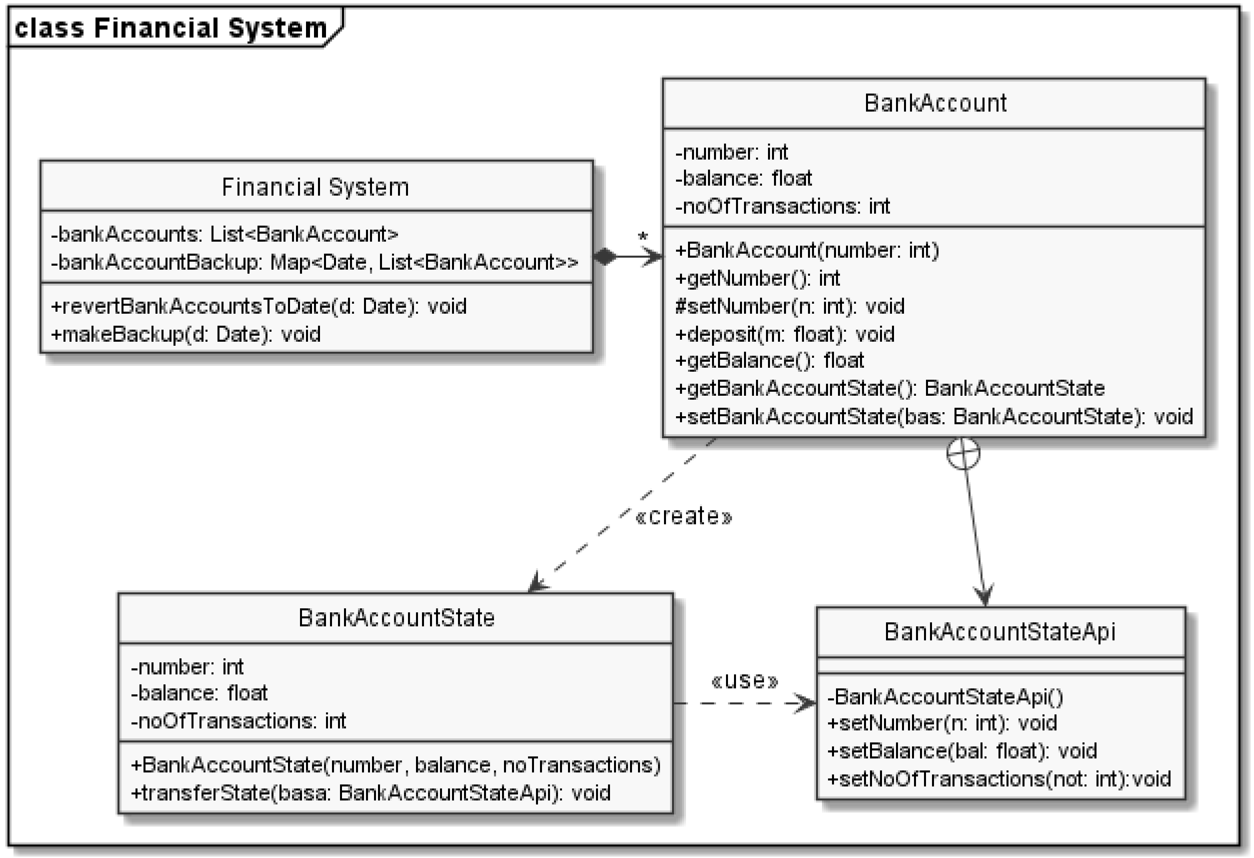


class Financial System
Financial System
-bankAccounts: List<BankAccount>
-bankAccountBackup: Map<Date, List<BankAccount>>
+revertBankAccountsToDate(d: Date): void
+makeBackup(d: Date): void
*
BankAccount
-number: int
-balance: float
-noOfTransactions: int
+BankAccount(number: int)
+getNumber(): int
#setNumber(n: int): void
+deposit(m: float): void
+getBalance(): float
+getBankAccountState(): BankAccountState
+setBankAccountState(bas: BankAccountState): void
How to let Financial System make a copy of the state of BankAccount without breaking encapsulation? (i.e. how setNumber and how to get/setNoOfTransactions

Case: Financial System
Design 2: now no encapsulation break…
But how to access the BankAccountState?


class Financial System
Financial System
-bankAccounts: List<BankAccount>
-bankAccountBackup: Map<Date, List<BankAccount>>
+revertBankAccountsToDate(d: Date): void
+makeBackup(d: Date): void
*
BankAccount
-number: int
-balance: float
-noOfTransactions: int
+BankAccount(number: int)
+getNumber(): int
#setNumber(n: int): void
+deposit(m: float): void
+getBalance(): float
+getBankAccountState(): BankAccountState
+setBankAccountState(bas: BankAccountState): void
«create»
BankAccountState
-number: int
-balance: float
-noOfTransactions: int
+BankAccountState(number, balance, noTransactions)
+transferState(basa: BankAccountStateApi): void

Case: Financial System
Design 3: narrow interface to BankAccountState


class Financial System
Financial System
-bankAccounts: List<BankAccount>
-bankAccountBackup: Map<Date, List<BankAccount>>
+revertBankAccountsToDate(d: Date): void
+makeBackup(d: Date): void
*
BankAccount
-number: int
-balance: float
-noOfTransactions: int
+BankAccount(number: int)
+getNumber(): int
#setNumber(n: int): void
+deposit(m: float): void
+getBalance(): float
+getBankAccountState(): BankAccountState
+setBankAccountState(bas: BankAccountState): void
«create»
BankAccountState
-number: int
-balance: float
-noOfTransactions: int
+BankAccountState(number, balance, noTransactions)
+transferState(basa: BankAccountStateApi): void
«use»
BankAccountStateApi
-BankAccountStateApi()
+setNumber(n: int): void
+setBalance(bal: float): void
+setNoOfTransactions(not: int):void

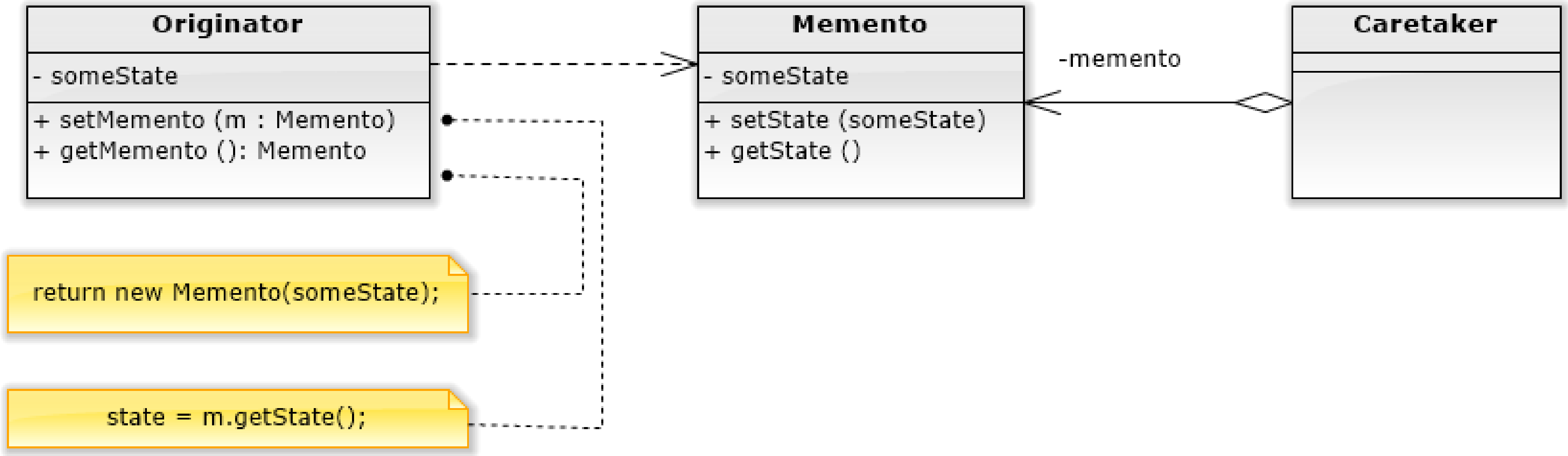
Memento pattern definition

- **Intent**: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

- **Motivation**: Sometimes it's necessary to record the internal state of an object. (for checkpoints or undo mechanisms or persistence), so you must save state information somewhere so that you can restore objects to their previous states. But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally.
  This is solved by the memento without breaking encapsulation

- **Applicability**: Use the Memento pattern when:
  - a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
  - a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure:


Originator
- someState
+ setMemento (m : Memento)
+ getMemento (): Memento
Memento
- someState
+ setState (someState)
+ getState ()
Caretaker
-memento
return new Memento(someState);
state = m.getState();

- **Participants:**
  - Memento
    - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
    - protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a *narrow interface* to the Memento—it can only pass the memento to other objects. Originator, in contrast, sees a *wide interface*, one that lets it access all the data necessary to restore itself to its previous state.
  - Originator
    - creates a memento containing a snapshot of its current internal state.
    - uses the memento to restore its internal state.
  - Caretaker
    - is responsible for the memento's safekeeping.
    - never operates on or examines the contents of a memento.

- **Consequences**: The memento pattern leads to:
  - Preserving encapsulation boundaries
  - It simplifies the Originator
  - Using mementos might be expensive
  - Defining narrow and wide interfaces could be difficult in a language
  - Hidden costs in caring for mementos

- **Implementation:** Some issues implementing a memento.
  - Language support for narrow and wide interfaces
  - Storing incremental changes vs full storage

Reading

For this lesson please read:

- Chapter Memento of Design Patterns: Elements of Reusable Object-Oriented Software, pp 316-325.