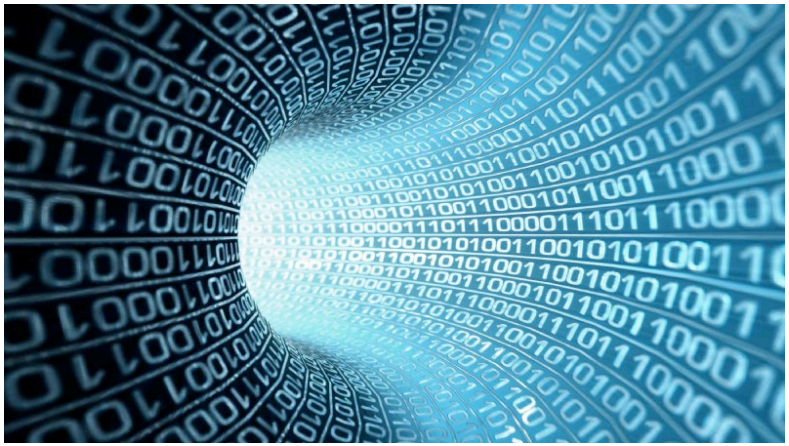
Object Oriented Design

COSC346

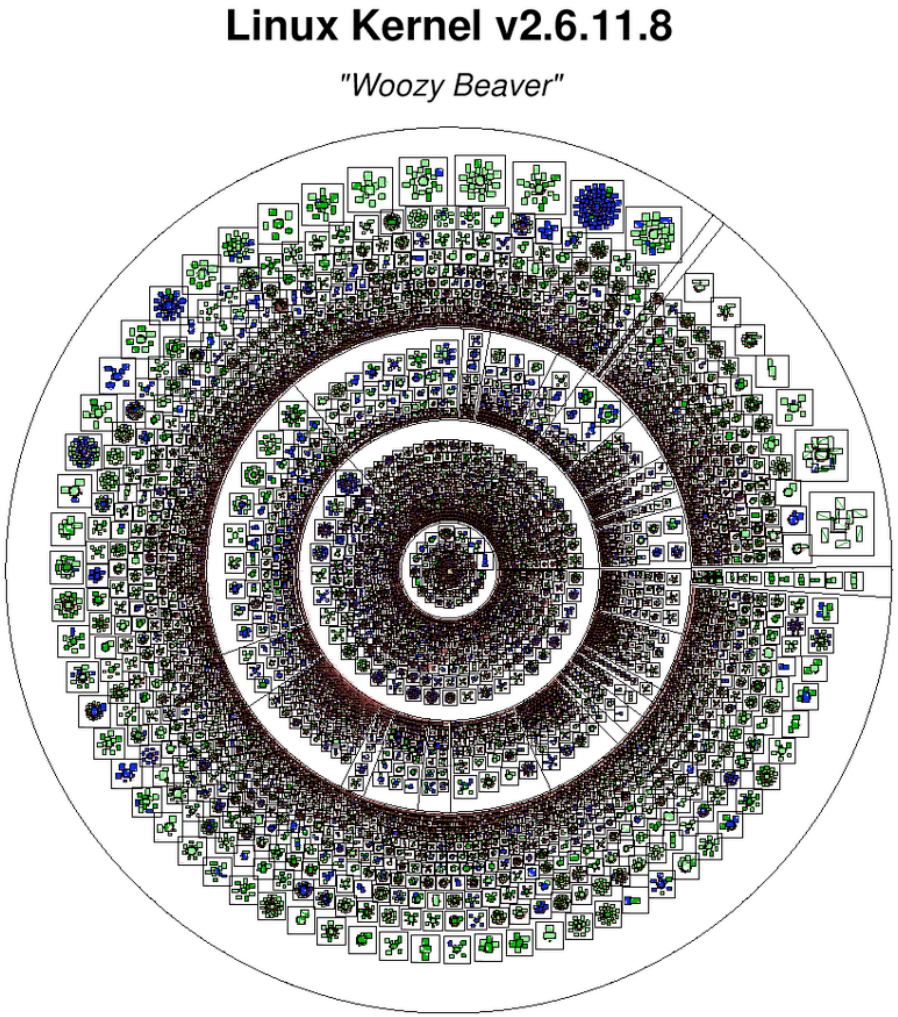
Programming in the large

- One of the main advantages of OOP is its usefulness for "Programming in the Large"
  - *i.e.*, for building large software systems (*e.g.*, Photoshop, Word, Grand Theft Auto)
  - Large development team (from 10s to 100s of people, GTA IV = 1,000 people + $100M)
  - No individual is responsible for whole project or even understands all aspects of project.
  - Major challenge is management of details and communication between different subsystems



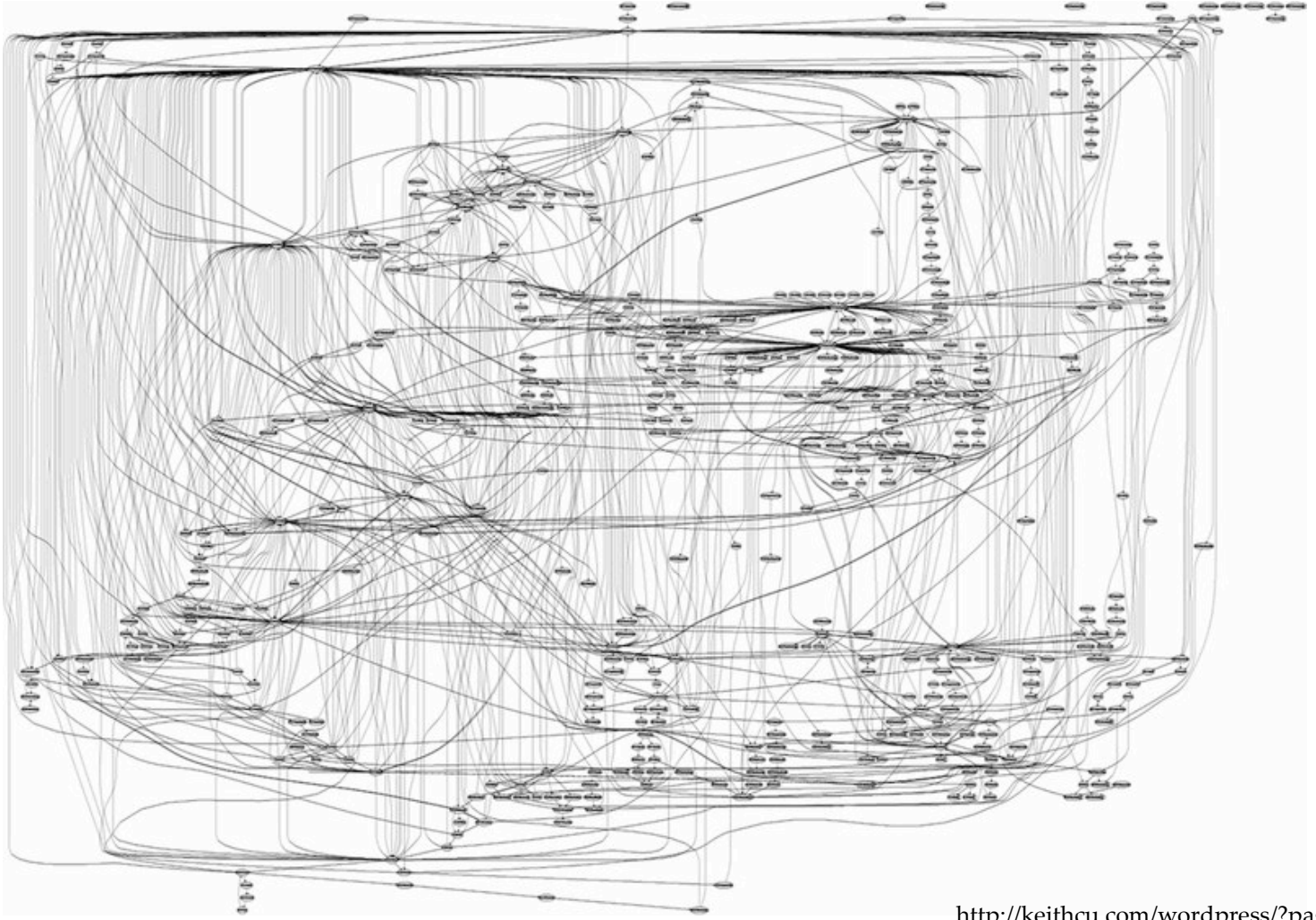
http://www.hw.ac.uk

Programming in the large



http://keithcu.com/wordpress/?page_id=599

Programming in the large



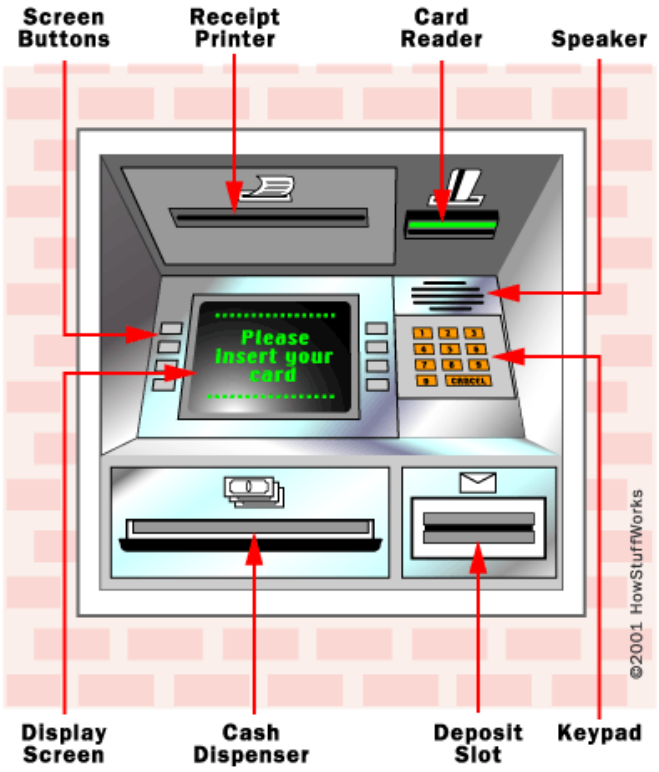
http://keithcu.com/wordpress/?page_id=599

Overview

1. **Design**—first understand a problem in terms of requirements (w/o reference to software)
2. **Use Cases**—next understand how the system might be used to perform a particular task
3. **Determine Classes**—often suggested by use cases
4. **CRC Cards**—how the classes interact
5. **Assigning Responsibilities** to classes
6. **Sequence Diagrams**—determine the dynamic interaction between classes
7. **Class Diagrams**—static interaction between classes

1. Design

- Design is the process of understanding the requirements of a system
  - *e.g.*, an ATM must verify identity, securely store cash, dispense money, and accept deposits
- For object-oriented software, we need to determine the entities (or classes) inherent in the system
- Candidates often emerge from how the system is to be used, which can be described as a series of use cases



2. Use cases

- A use case is a narrative that describes the sequence of events of an actor (an external agent) using a system to complete a single goal or task
  - *e.g.*, a person withdrawing money from ATM
- To describe the use case, we describe the actions of the actor and how the system responds
  - The actor might be a human user or it might be another system

2. Use cases

- The use case describes both the desired actions and what might go wrong (the error behaviours) in that single task
- To describe a whole system it is often necessary to include many use cases
- Use cases are described in varying level of detail
  - Brief: a few summarising sentences
  - Casual: a few paragraphs of text
  - Fully dressed: "a formal document based on a detailed template with fields for various sections" (Wikipedia)

ATM example

- Here we present a "fully dressed" use case

Use Case

Withdraw money from ATM

Actors

Customer

Brief Description

Customer wishes to withdraw money from an ATM

ATM example

Step-by-Step

1. User inserts card into machine
2. System validates card and requests PIN
3. User enters PIN
4. System validates PIN and awaits action
5. User selects withdraw cash
6. System asks "how much?"
7. User enters amount
8. System check user's balance and its available cash
9. System dispenses cash
10. System ejects card
11. User takes money and card

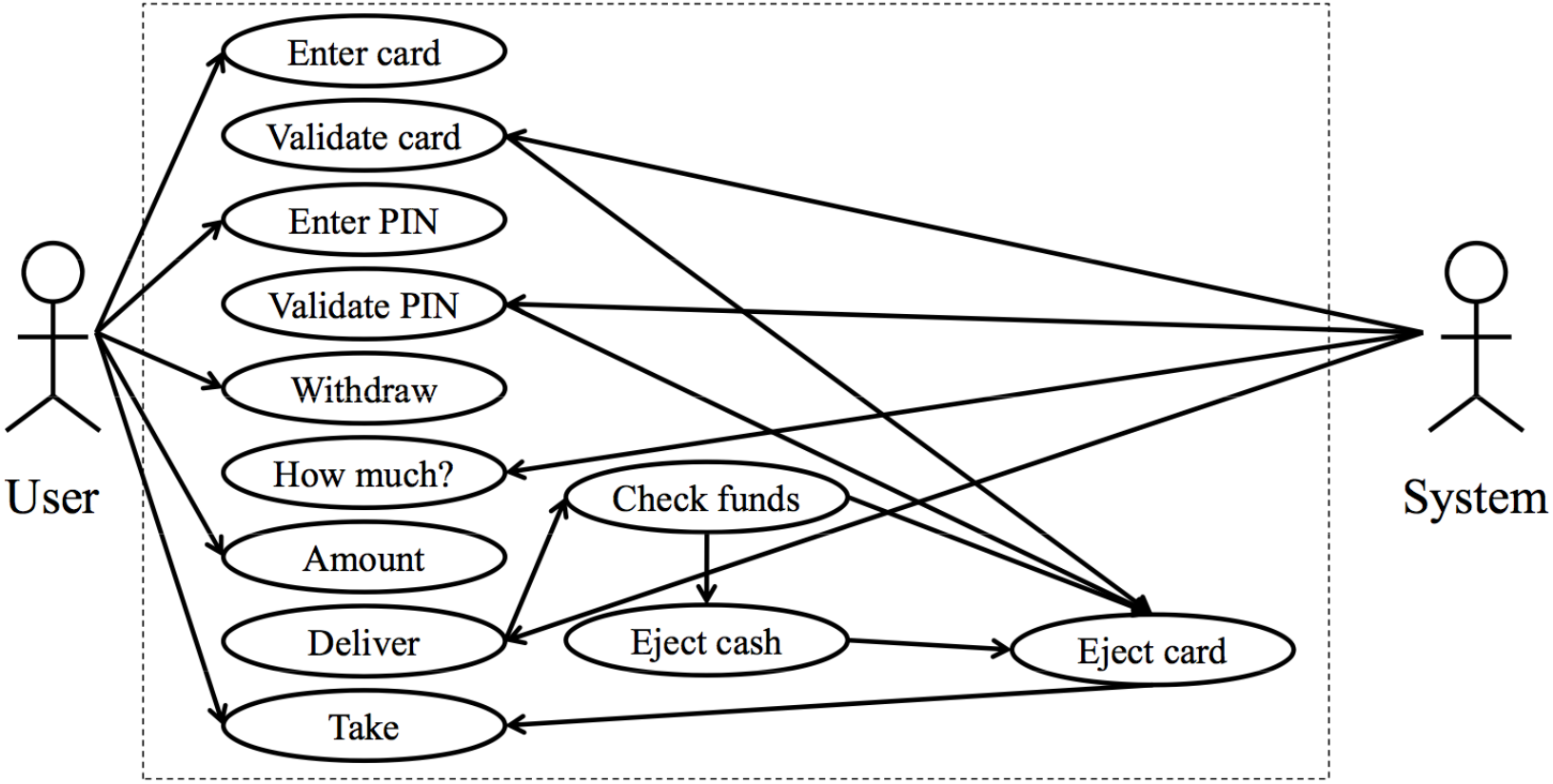
ATM example

Errors

- Errors can happen between any two steps
  - In all cases respond the same way: back out of transaction, eject card, restart
- This is like handling optionals

ATM example

- Use case diagrams show the actors, their actions, and how they interact

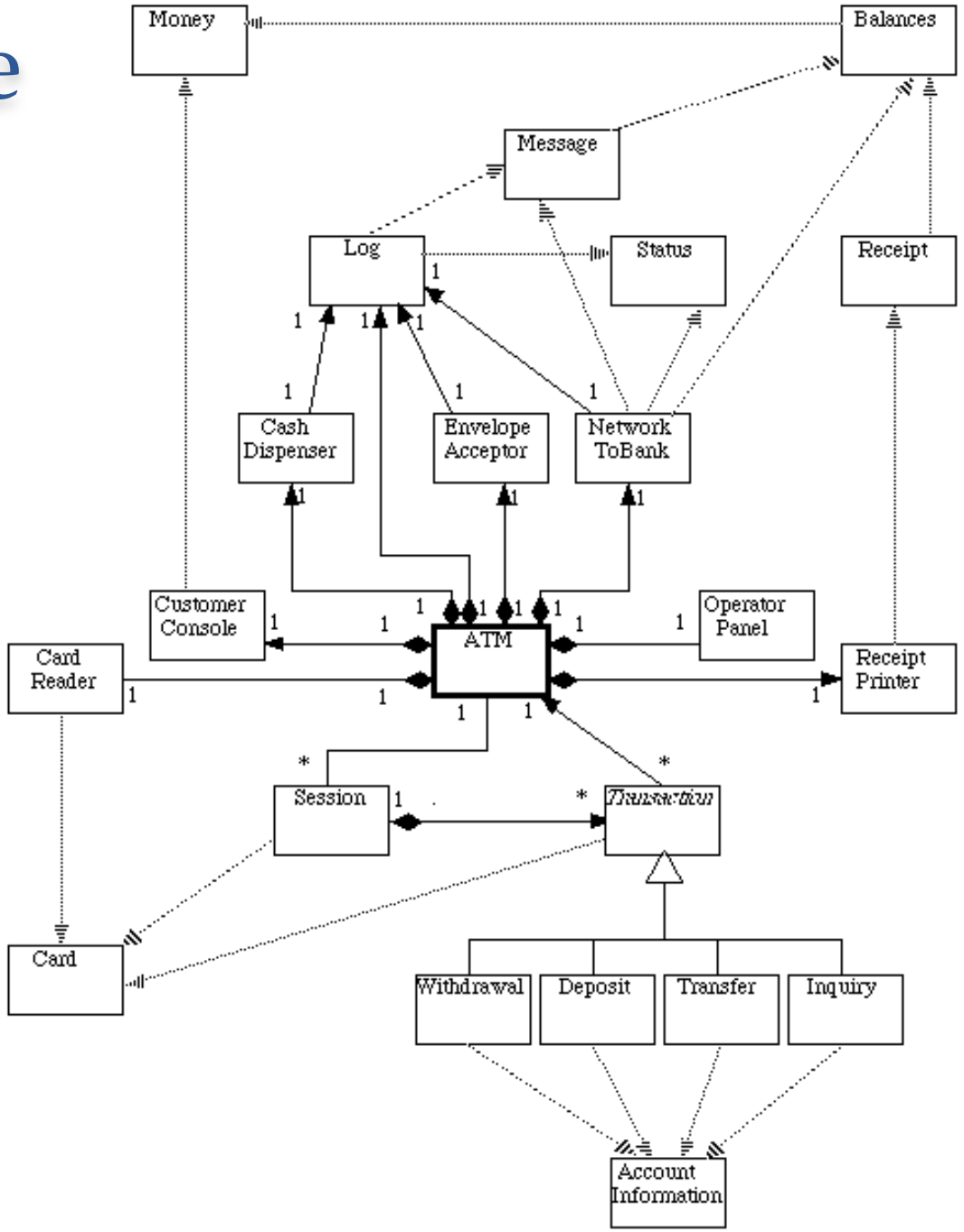


3. Determine classes

- From our analysis, we need to determine the classes in the system

- Choose a subset of use cases (could be just 1) and try to determine classes and interactions necessary to realise that use case:
  - Could nouns be classes?
  - Could verbs be actions (methods)?

3. Determine classes



4. CRC card

- A **CRC** card is a "**Class Responsibility Collaboration**" card

- On a small paper CRC card write:
  - The class name
  - Its super- and sub-classes (if any)
  - The class responsibilities
  - The names of other classes communicating with this class
  - The author of the class

4. CRC card

Class: Card_reader	
Subclasses: None	Superclasses: None
Responsibilities	Collaborators
Wake ATM on card insertion	ATM
Read card	Card
Eject card	
Swallow card	

- Lay out the CRC cards and simulate the running of the program as a conversation between them
- Cards send messages to each other
  - Objects invoke method on each other

5. Assigning responsibilities

- Two types of responsibilities for an object:
  - **Knowing** (state)
    - about data
    - about related objects
  - Knowledge is usually stored in **instance variables**
  - **Doing** (behaviour)
    - Deriving or calculating something
    - Knowing which other objects can do things
    - How to coordinate with other object that do things
  - These abilities are usually performed as **methods**

5. Assigning responsibilities

- Try to assign responsibilities that minimise coupling and maximise cohesion
  - **Coupling**—Assign responsibilities that lower dependency of objects on each other
  - **Cohesion**—Assign responsibilities that increase the independence of an object

Image by Steve Easterbrook, University of Toronto

Coupling and cohesion

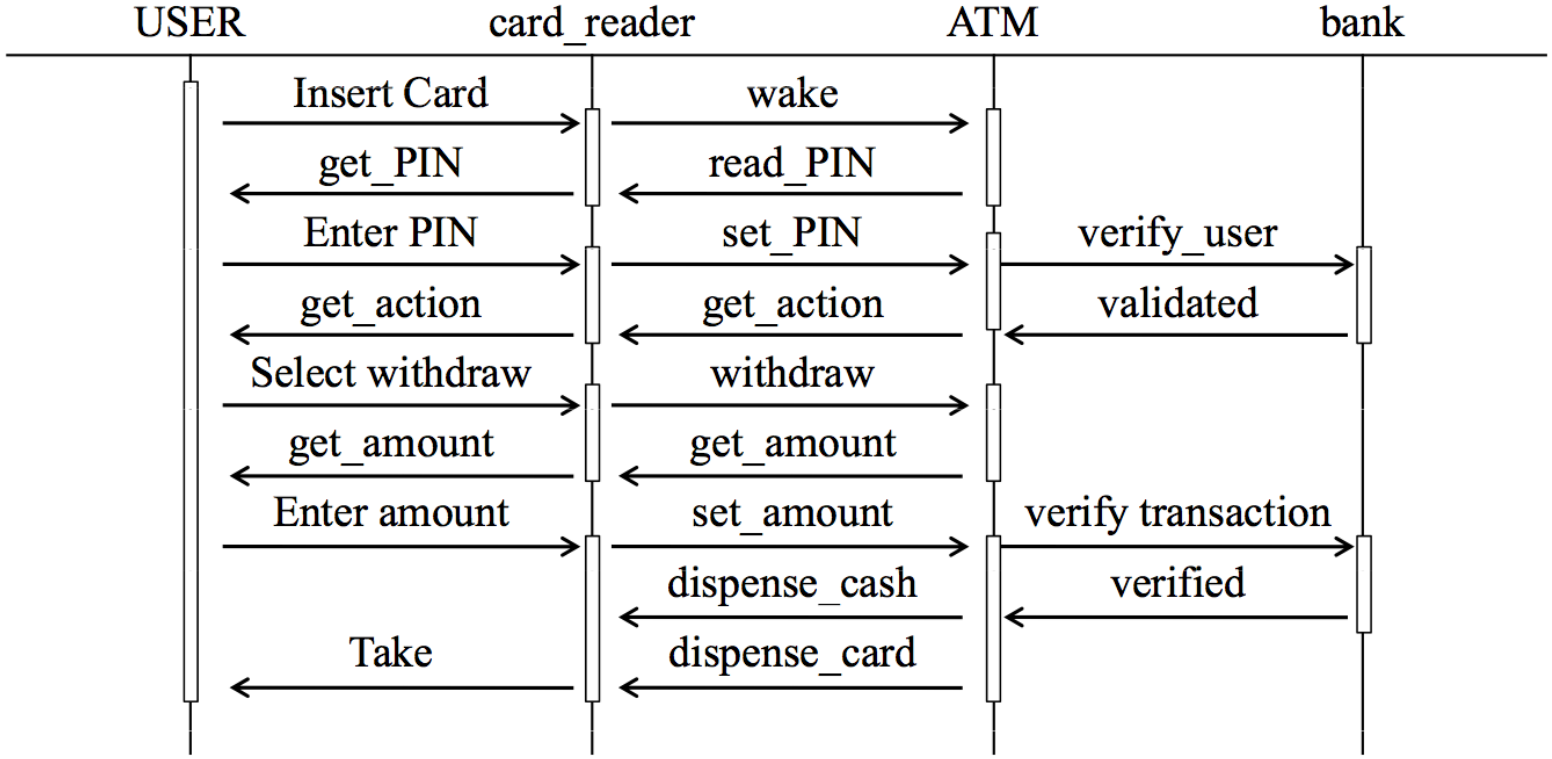
- You want *low* coupling and *high* cohesion. How do you accomplish this goal?
  - **Law of Demeter**: talk to your friends, but not your friend's friends
    - Avoid compound messages
  - **Respect encapsulation**: don't ask for personal details
    - Don't rely on internal details of another object
  - **Avoid code duplication**: don't use the same logic in multiple places
  - **Group code by function**: Try to give each class a well-defined task

Code smell

- Code smell refers to signs that your code might need reorganisation
  - **Divergent Changes**: one class requires multiple changes for different reasons (low cohesion)
  - **Feature Envy**: one class is too interested in workings of another class (high coupling)
  - **Shotgun Surgery**: changing one class requires changes in other classes (high coupling)
- Other general code smells:
  - Large classes, large methods, long parameter lists
  - Unclear naming, too few comments, too many comments
  - Uncalled code, overly general code
  - Many others …

6. Sequence diagrams

- Sequence diagrams illustrate how objects interact (via calls) to fulfil tasks
  - Time goes from top to bottom

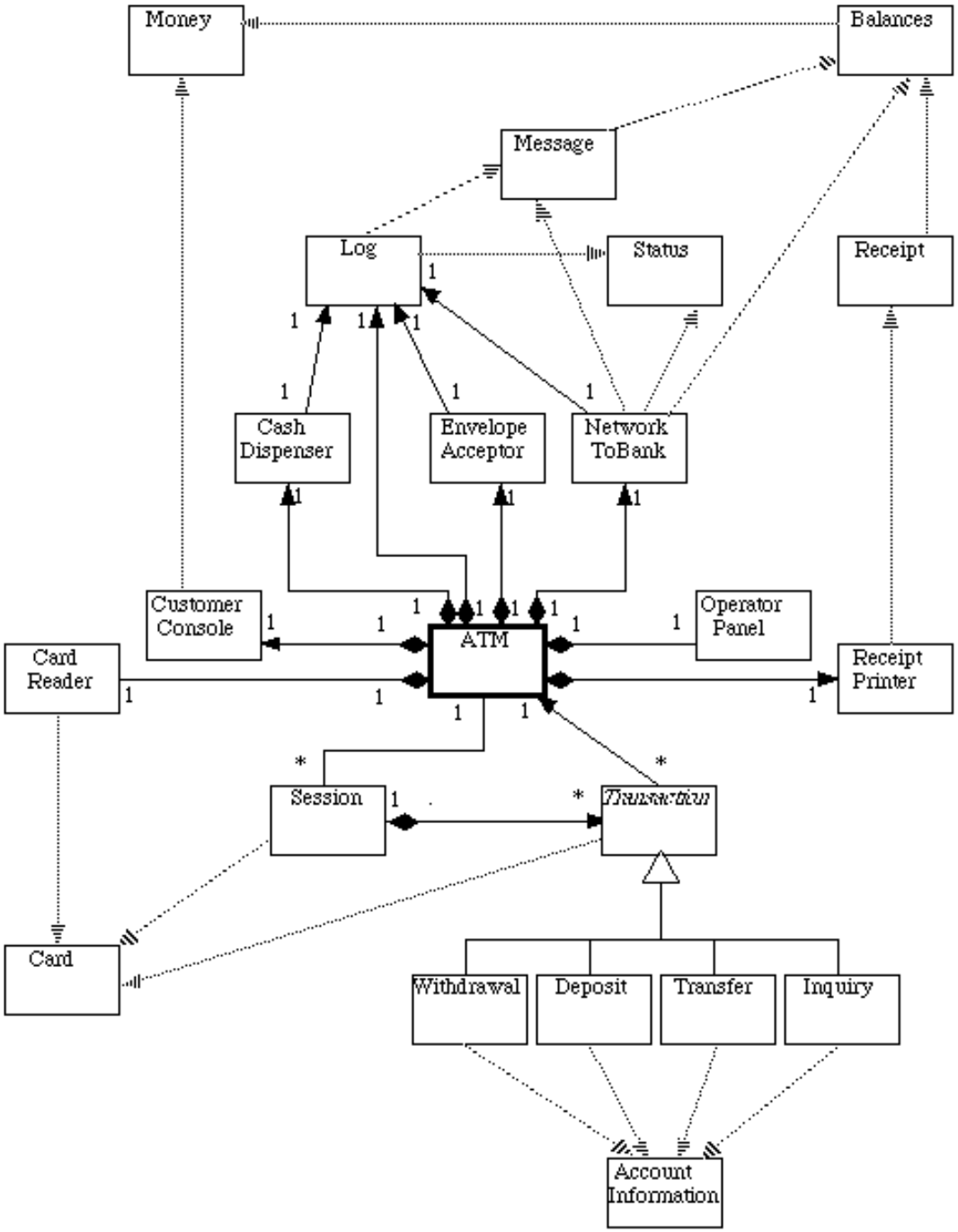


7. Class diagrams

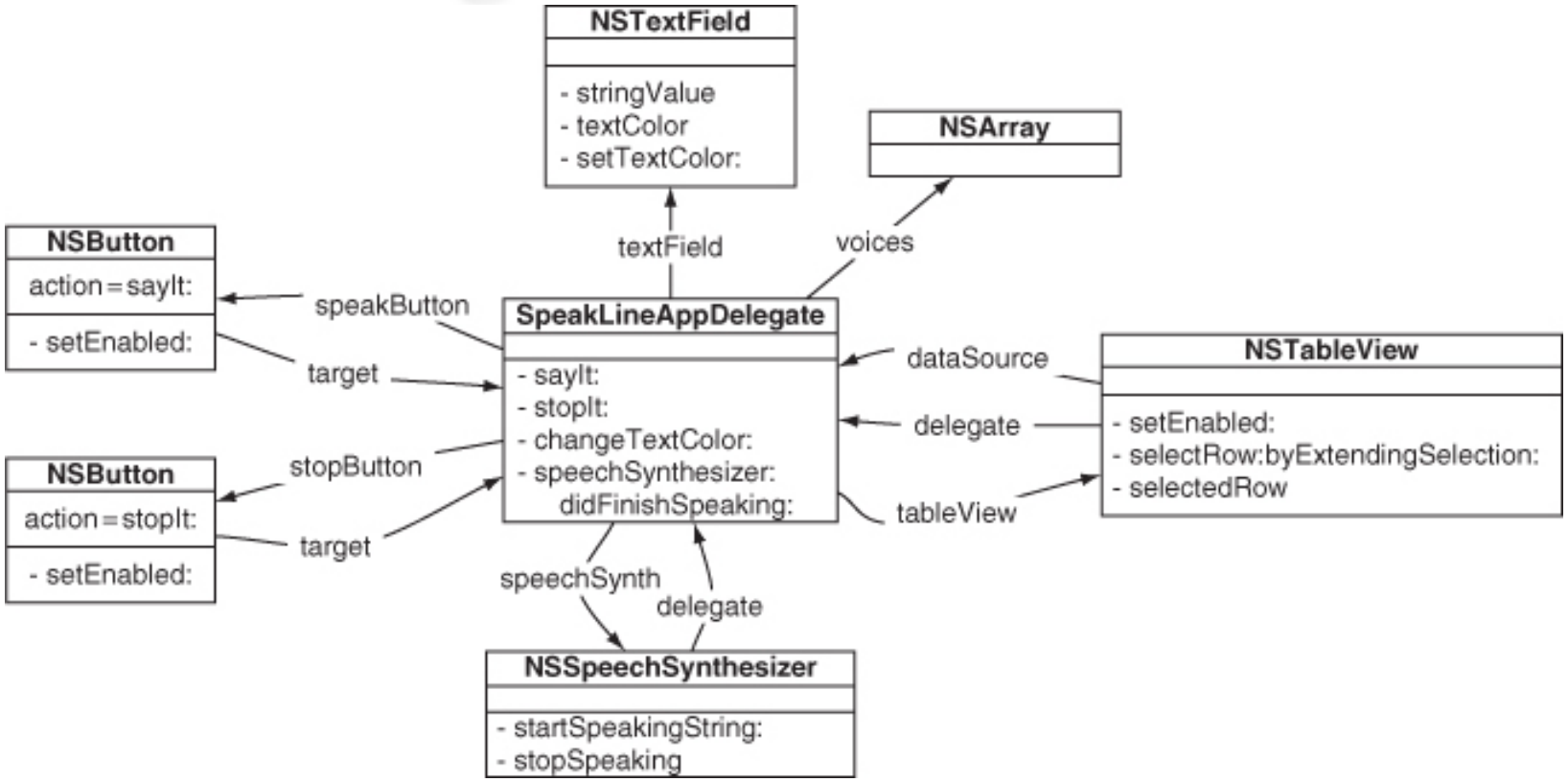
- A class diagram (AKA an object diagram) illustrates how a program's classes interact
  - They are useful for design when a program is actually being implemented

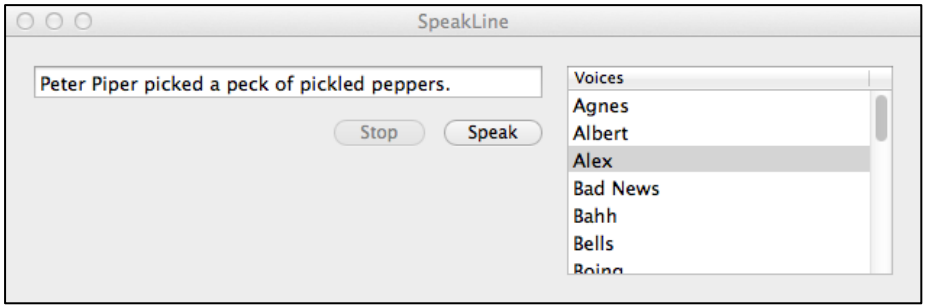
- Class diagrams include details about:
  - Classes, associations, and attributes;
  - Interfaces and operations;
  - methods;
  - attribute type information;
  - dependencies

7. Class diagrams



7. Class diagrams





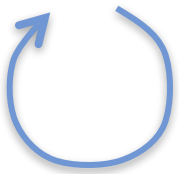
Conclusion

- Design diagrams are useful for planning out your object-oriented project
  - Think of them like as a way to organise your thoughts before you start coding
  - Like an outline of an essay
- Design diagrams include:
  - use case diagrams
  - CRC cards
  - sequence diagrams
  - class (or object) diagrams

Unofficial design guide from Lech

Preliminary stage:

- Describe functionality and requirements—**WRITE IT DOWN**
- How are you going to test the software?
- How are you going to maintain it?
- How are you going to support it?
- How are you going to distribute it?



Development stage:

- Decide on the representation—what is the underlying model?
- Implement all the interface logic with empty functions/ methods
- Write test scripts
- Fill in the code for implementation—keep testing as you develop

Design Pattern - Memento

- Capture an objects internal state without exposing internal structure
  - maintain encapsulation
- Undo/Redo
  - e.g. Editor undo action, ctrl-z
- Three components
  - memento—basic state storage/retrieval
  - originator—creates new mementos
  - caretaker—holds all mementos

Memento Example

Toolmaker

```
// Memento
typealias Memento = NSDictionary

// Originator
protocol MementoConvertible {
    var memento: Memento { get }
    init?(memento: Memento)
}
struct GameState: MementoConvertible{
    var chapter: String

    init(chapter: String) {
        self.chapter = chapter
    }

    init?(memento: Memento) {
        guard let mementoChapter =
                memento["chapter"] as? String else {
                 return nil
        }
        chapter = mementoChapter
    }

    var memento: Memento {
        return ["chapter": chapter ]
    }
}
```

```
// Caretaker
enum CheckPoint {
    static func save(_ state: MementoConvertible,
         saveName: String) {

        let defaults = UserDefaults.standard
        defaults.set(state.memento, forKey: saveName)
        defaults.synchronize()
    }

    static func restore(saveName: String) -> Memento? {
        let defaults = UserDefaults.standard
        return defaults.object(forKey: saveName) as? Memento
    }
}
```

Toolmaker

Builder

```
var gameState = GameState(chapter: "Chapter 1")

gameState.chapter = "Chapter 2"
CheckPoint.save(gameState, saveName: "gameState1")

gameState.chapter = "Chapter 3"
CheckPoint.save(gameState, saveName: "gameState2")

if let memento = CheckPoint.restore(saveName: "gameState1") {
    let finalState = GameState(memento: memento)
    print(finalState ?? "no state :(")
}
```

Memento in the real world?

Summary?