Sequence Diagrams and Systems Architecture

CS6406

Cork Complex Systems Lab

Overview

- Model message flows using sequence diagrams.
- Relation of sequence diagrams to architecture
- Rules for defining system architectures

Use-Case Analysis

- Use-cases are a necessary starting point for systems development
- Formal models for use-cases
  - Sequence diagrams
  - Other sequence models (e.g., automata)
- Associated representation
  - Systems architecture
  - Functional models

Scenario Modeling Techniques – Interaction Diagram

- Scenario modeling describes how the objects in a system interact with each other in a scenario.
- A scenario is a sequence of events that occurs during one particular execution path within a use case of a system.
- Each event involves the interaction of objects passing messages between them.

Cork Complex Systems Lab

Scenario Modeling Techniques – Interaction Diagram (cont’d)

- An interaction diagram can be used to model the collaborating objects in scenarios, showing the objects involved in the scenario and the messages sent and received by them.
- These objects may be external or internal to the system.
- The messages represent the invocation of operations of the receiving objects.
- Sequence diagrams focus on the time sequencing of messages.

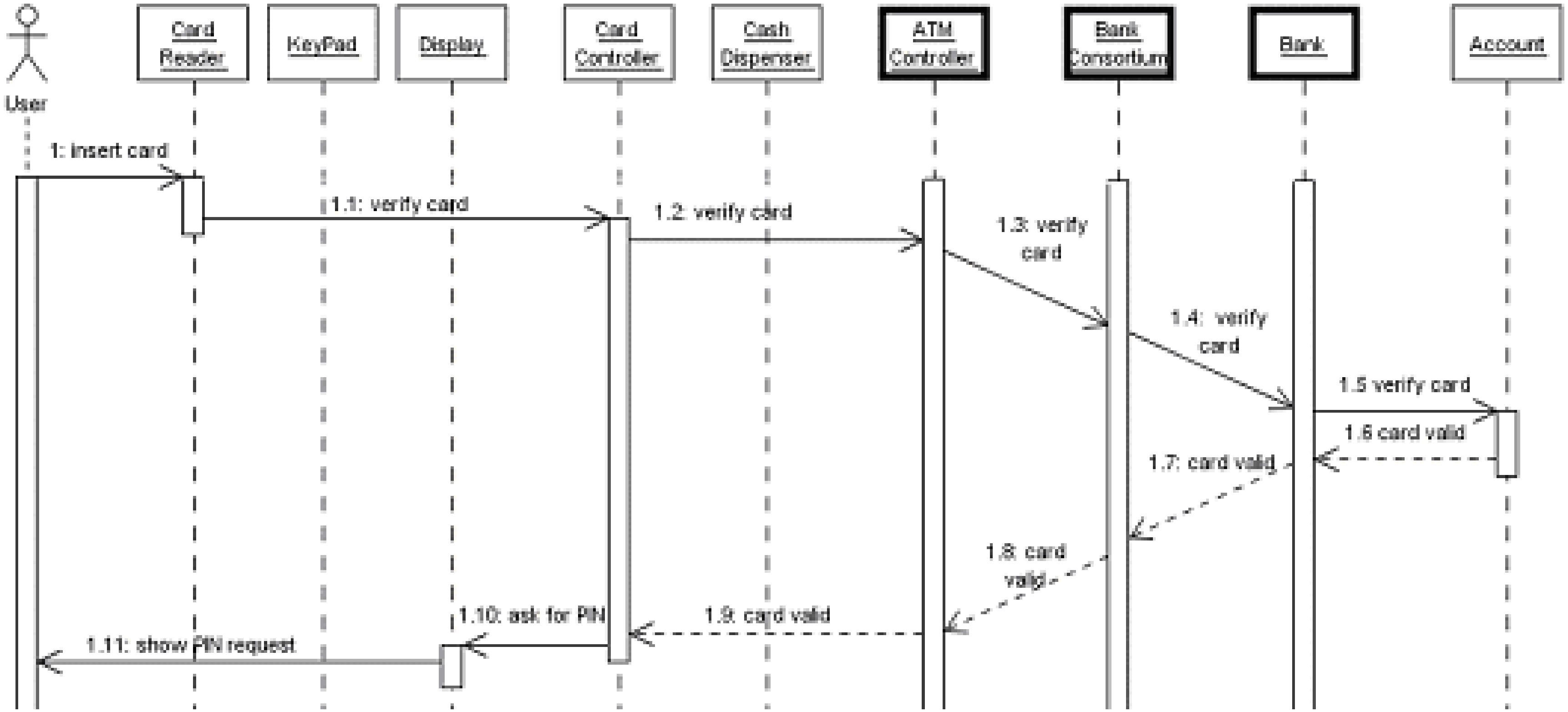
Use-Case 1– An Automatic Teller Machine (ATM)

- The ATM prompts the user to insert a card.
- The user inserts an ATM card.
- The ATM prompts the user to input the PIN.
- The user enters the PIN.
- The ATM asks the bank consortium to verify the ATM card number and PIN.
- The bank consortium verifies the ATM card number and PIN with bank.
- The bank notifies the bank consortium that the PIN is correct.
- The bank consortium notifies the ATM the PIN is correct.
- The ATM prompts the user to select a service.
- The user selects the withdraw cash service.
- The ATM prompts the user to enter the amount to withdraw.

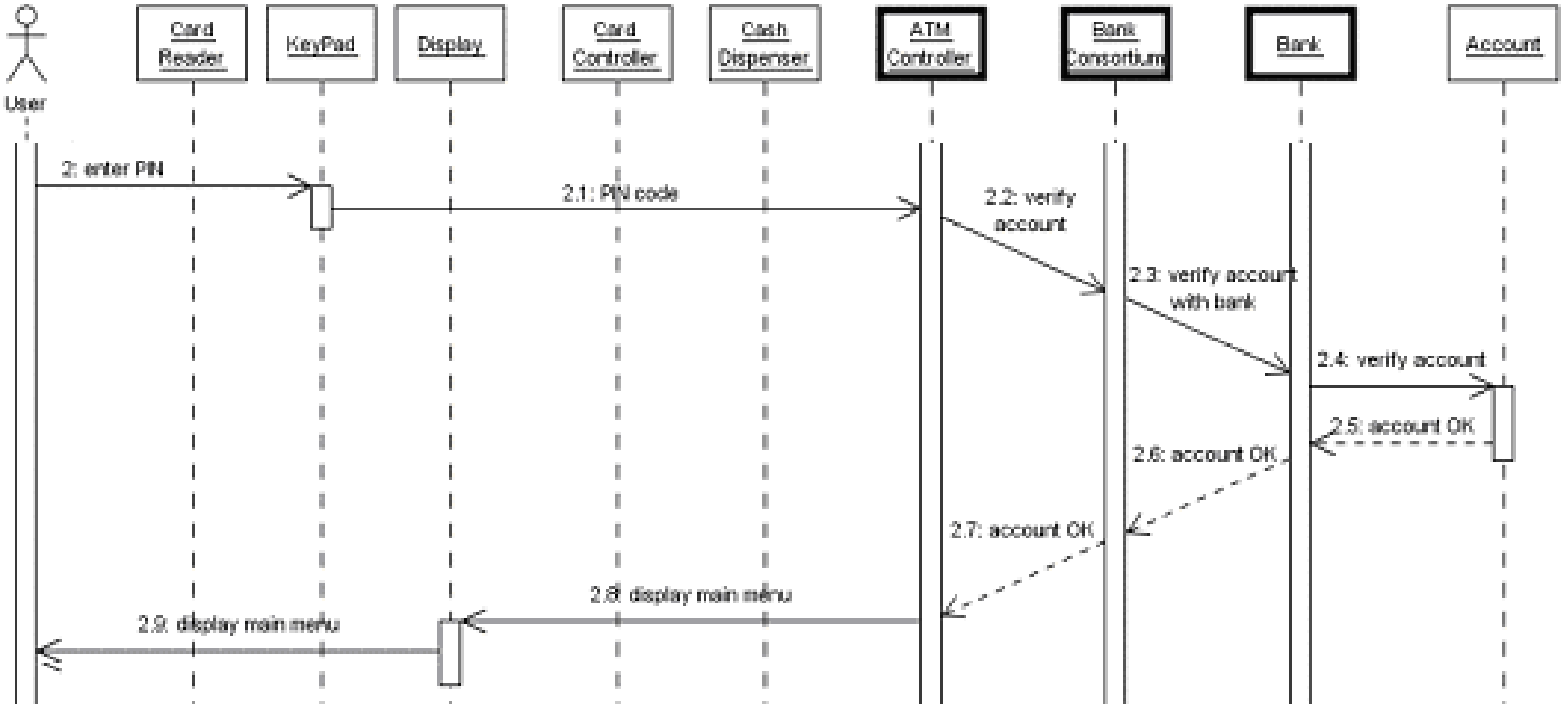
Example 1 – An Automatic Teller Machine (cont'd)

- The user enters the amount to withdraw.
- The ATM asks the bank consortium to process the request. The bank consortium forwards the request to bank.
- The bank confirms the successful execution of the request to the bank consortium which in turn notifies the ATM that the request has been approved.
- The ATM displays the successful transaction screen, ejects card and then dispenses cash requested.
- The ATM shows the main menu to the user for selecting the next service.

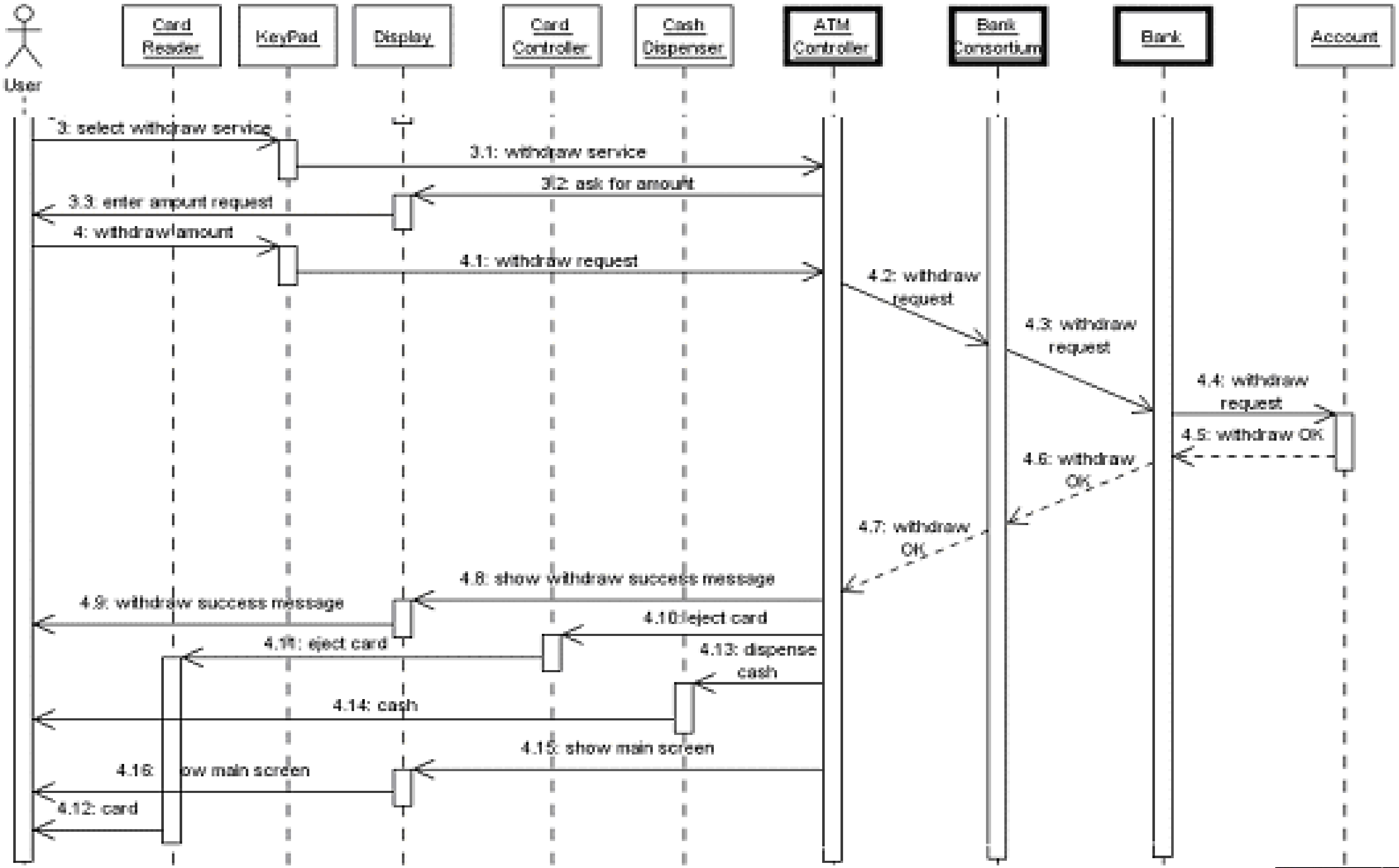
Example – An Automatic Teller Machine (cont'd)



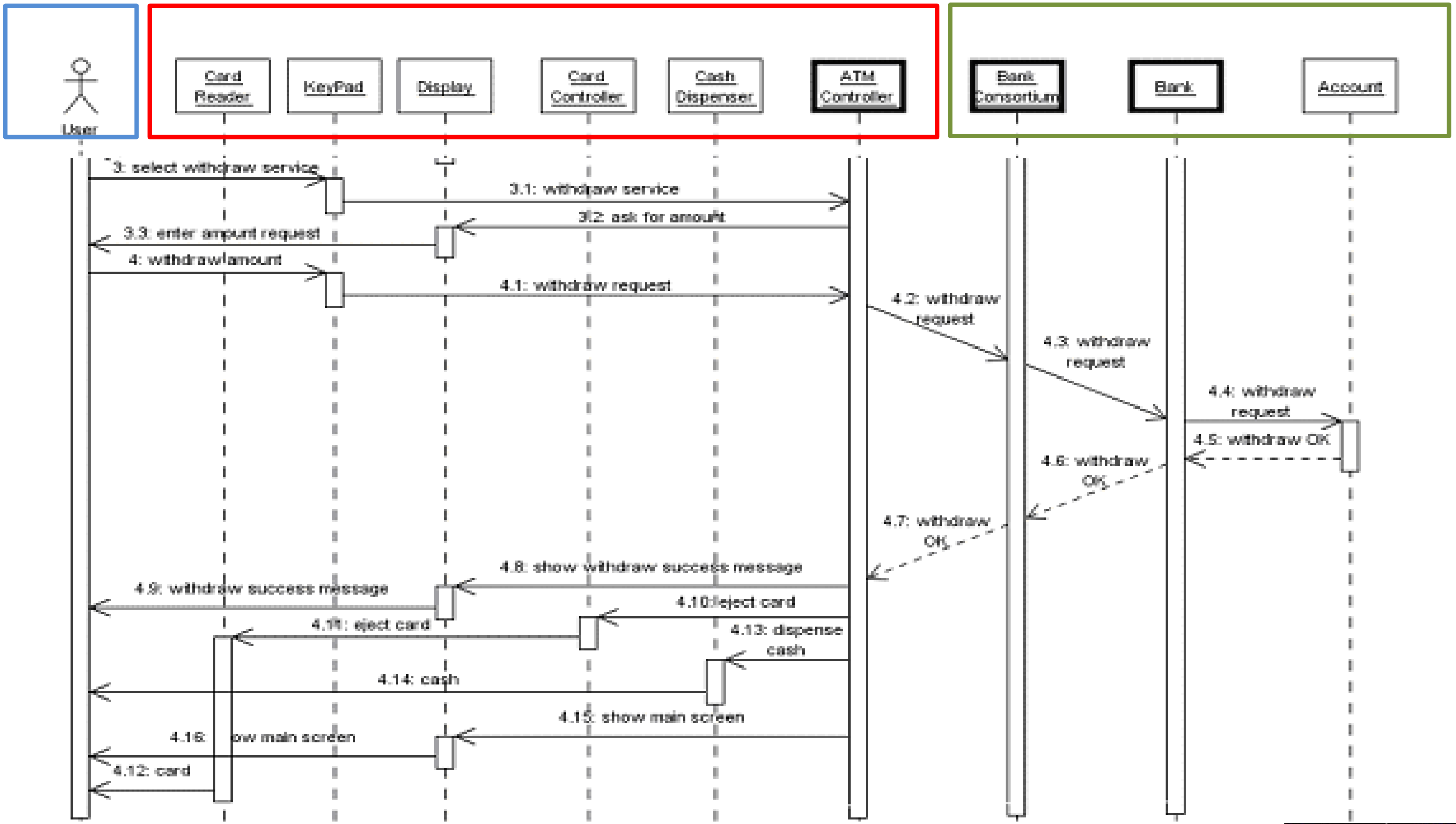
Example – An Automatic Teller Machine (cont'd)



Example – An Automatic Teller Machine (cont'd)



Architecture for Use-Case

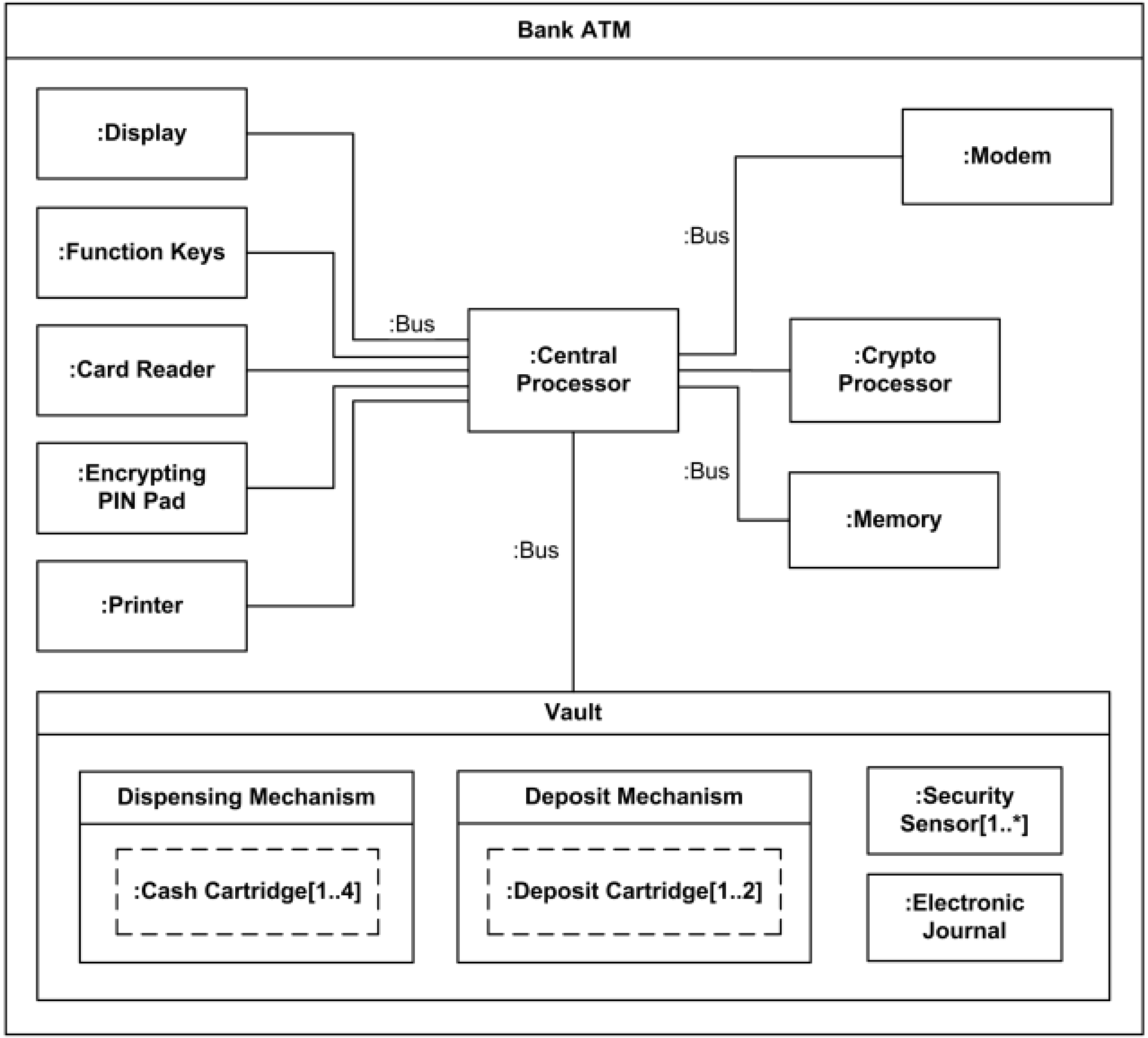


System Architecture

- Decomposition of system into core sub-systems
- Good decomposition is critical to any software project
  - Poor decomposition can lead to errors and/or inefficiency
- Example: security of messaging
  - Card verification vs. PIN verification

Cork Complex Systems Lab

ATM Sub-System Decomposition



System Architecture Definition

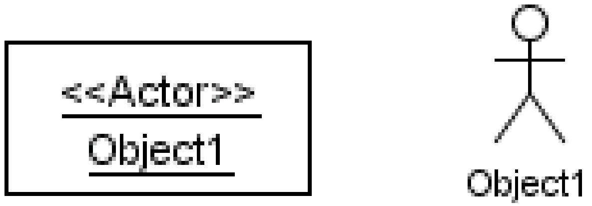
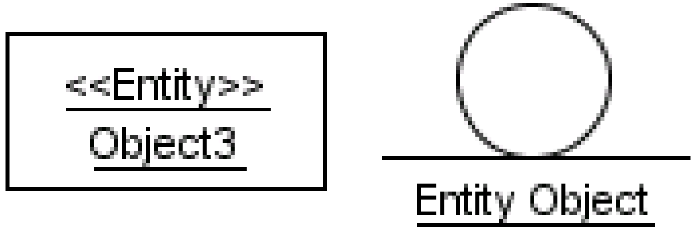
- Define a hierarchical structure
  - Trades off abstraction for “logical structure”
  - Example: logical structure of ATM is to separate ATM from main bank database
  - Ensure different functions are partitioned
  - Do not duplicate functionality
- Encode all critical actors for every use-case
  - If any use-case is omitted the requirements are not satisfied

Common UML Interaction Diagram Notation

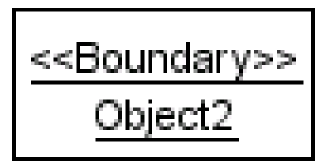
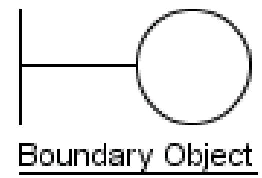
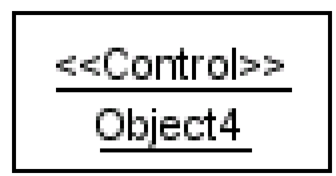
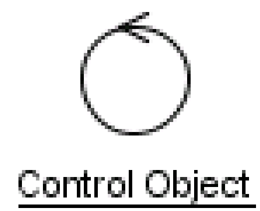
- **Object Symbol**

Naming Format	Notation
An object of an unspecified class.	object: object
A named object of a specified class.	objectX:Class
An unnamed object of a specified class.	:Class

Object Stereotype

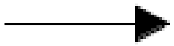
Object Category	Description	Graphical Notations
Actor Object	An external entity that interacts with the system.	
Entity Object	An object that models the data in the system. It often represents an object in the problem domain.	

Object Stereotype (cont'd)

Object Category	Description	Graphical Notations
Boundary Object	An object that handles the communication between actor objects and the system.	 
Control Object	An object that models the flow of control and functionality that do not naturally belong to entity objects or boundary objects.	 

Cork Complex Systems Lab

Messages

Message	Description	Notation
Procedure call or other nested flow of control	The message sender waits for the completion of the procedure call of the message receiver.	
Asynchronous communication	The sender dispatches a message and immediately continues with the next step of execution.	

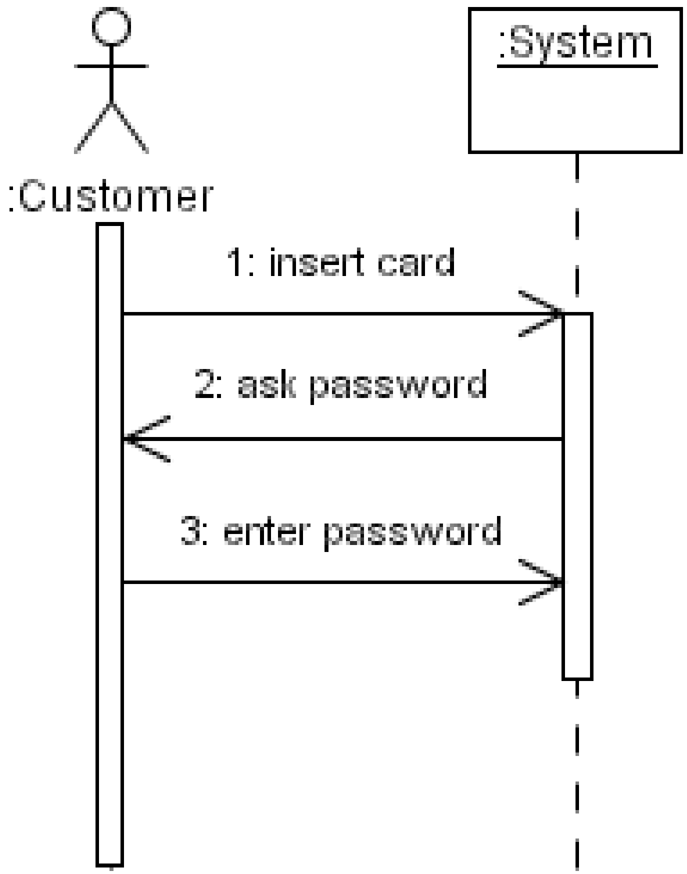
Messages (cont'd)

Message	Description	Notation
Return message	Message returned from the procedure call.	
Message with travel delay	The message will take a significant amount of time to arrive at the receiving object. (This is only used in sequence diagrams.)	

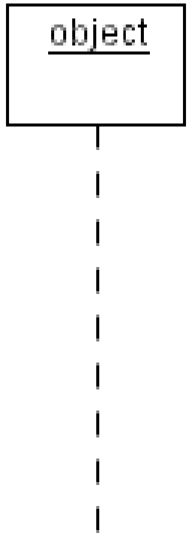
Sequence Diagrams

- An interaction diagram models the behavior of a group of objects that work together to achieve a user goal.
- A sequence diagram helps us identify a set of collaborating objects involved in a scenario of a use case.
- A sequence diagram has two dimensions: the vertical dimension and the horizontal dimension, respectively representing the passage of time and the objects involved in the interaction.
- Object icons are placed horizontally at the top of the sequence diagram, and messages are passed between them.

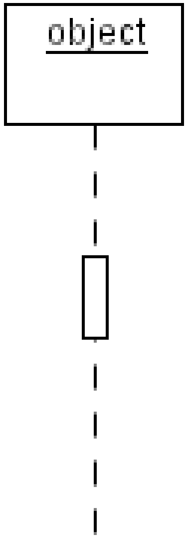
Sequence Diagrams (cont'd)



Life Line & Activation

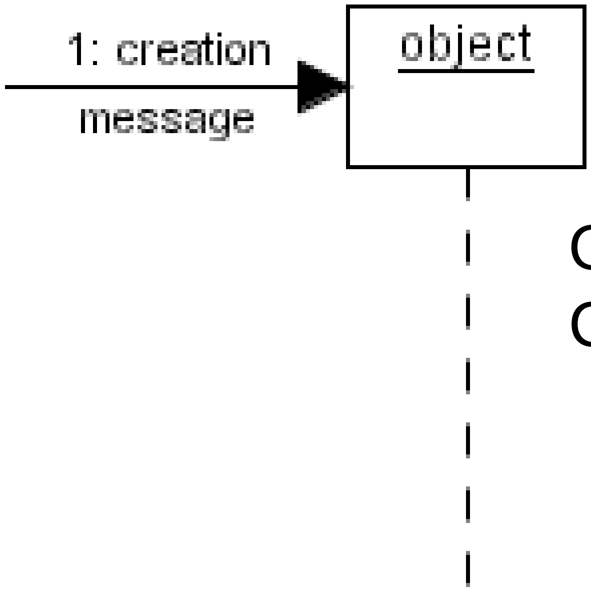


Object with
Lifeline

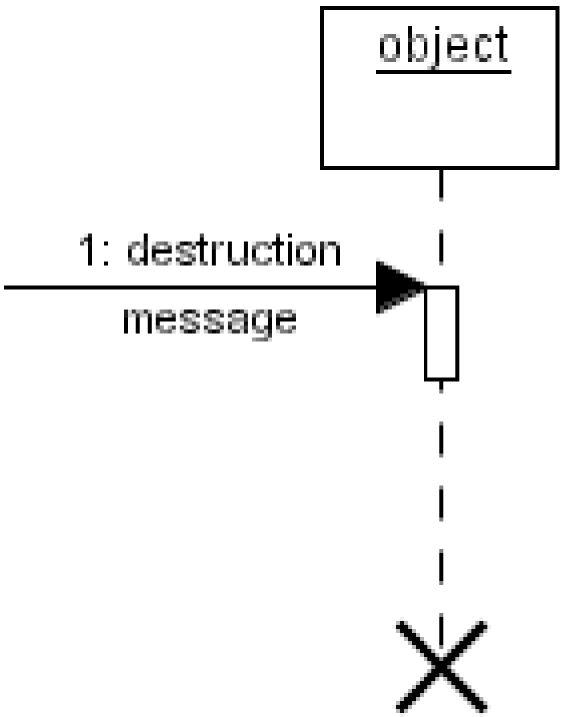


Object with
Activation

Creation & Destruction

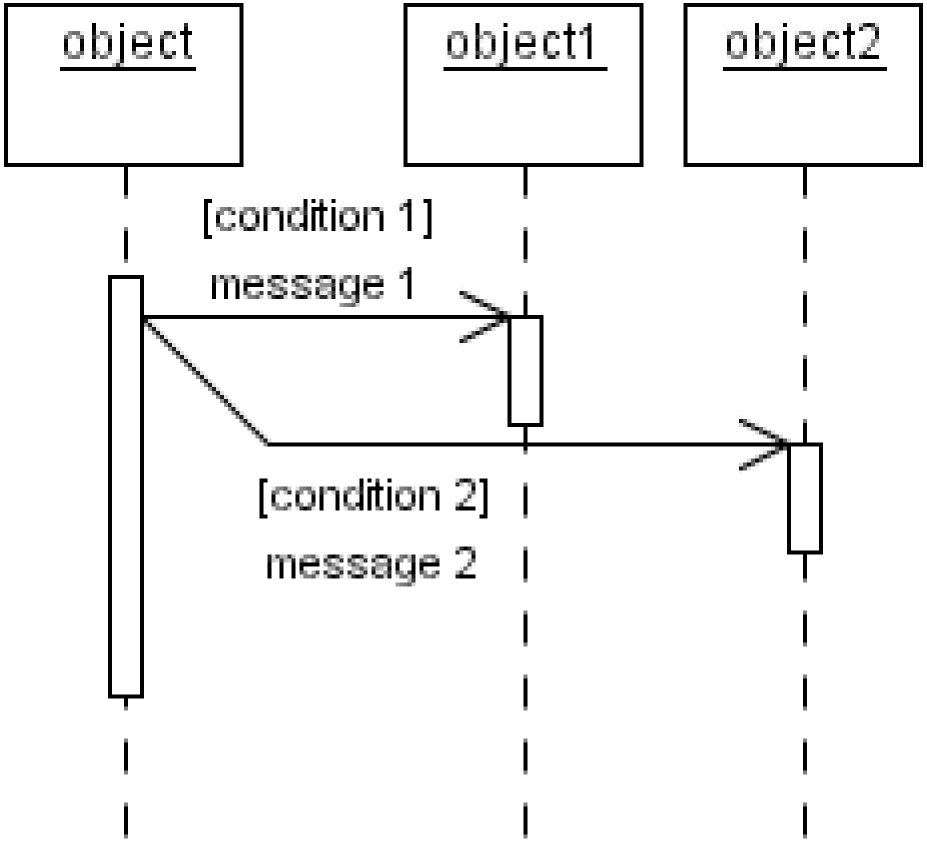


Object Creation



Object Destruction

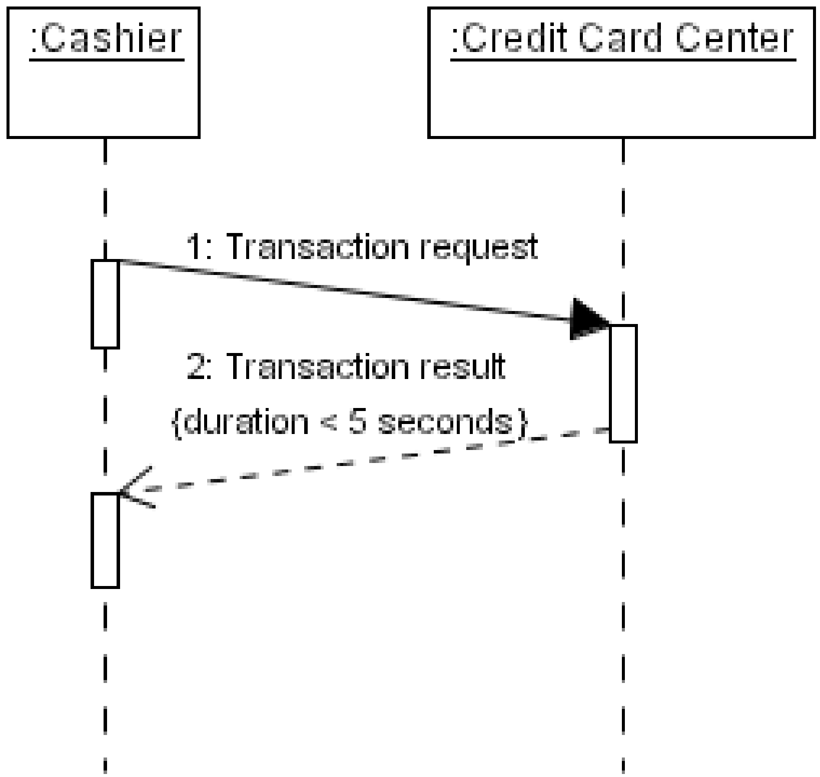
Branching



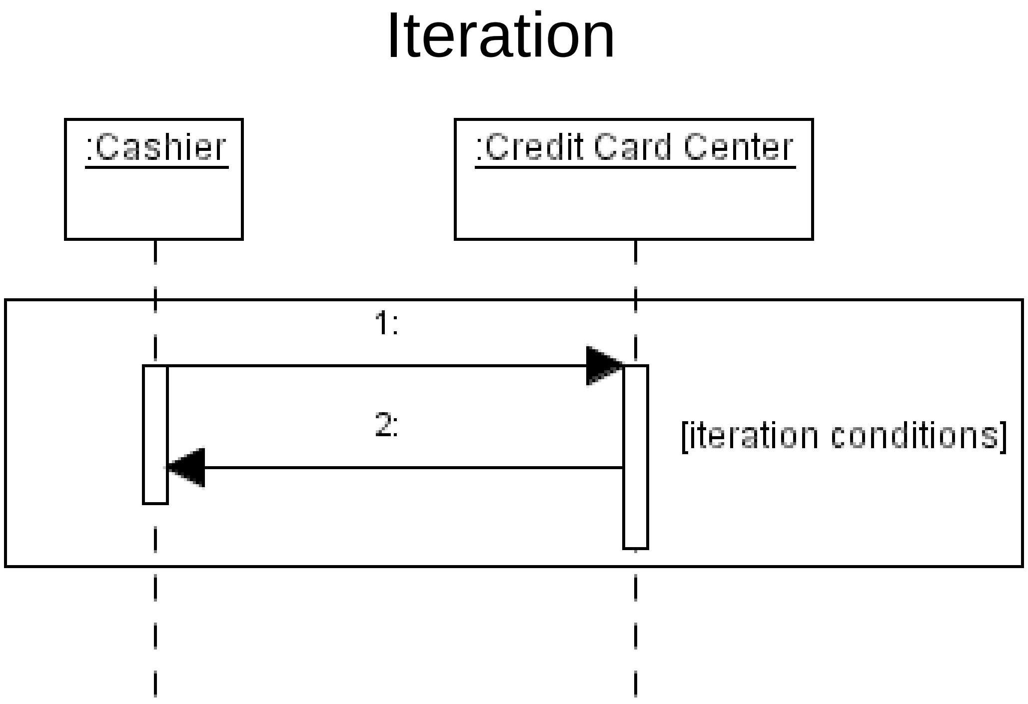
Conditional Message Transmission

Message that Takes Time

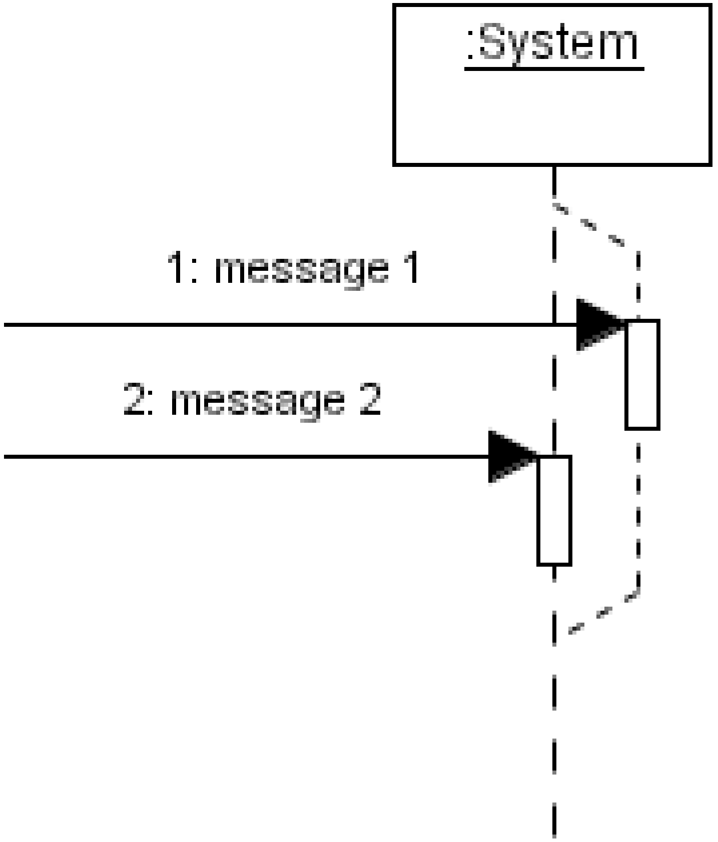
Message Transmission that Takes Time



Iteration

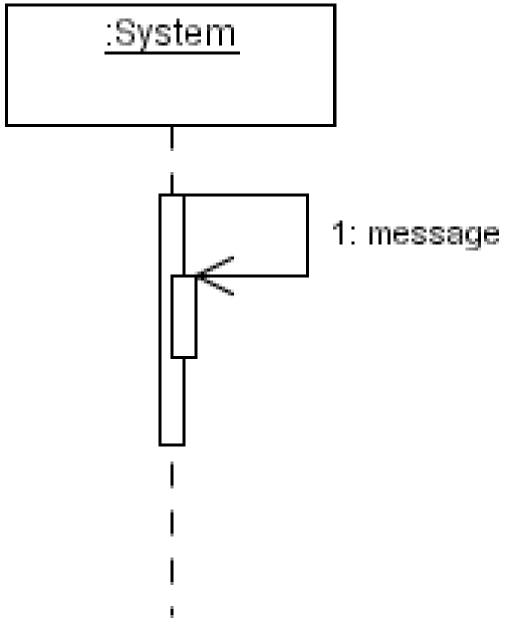

Iteration
:Cashier
:Credit Card Center
1:
2:
[iteration conditions]

Alternate Message Reception



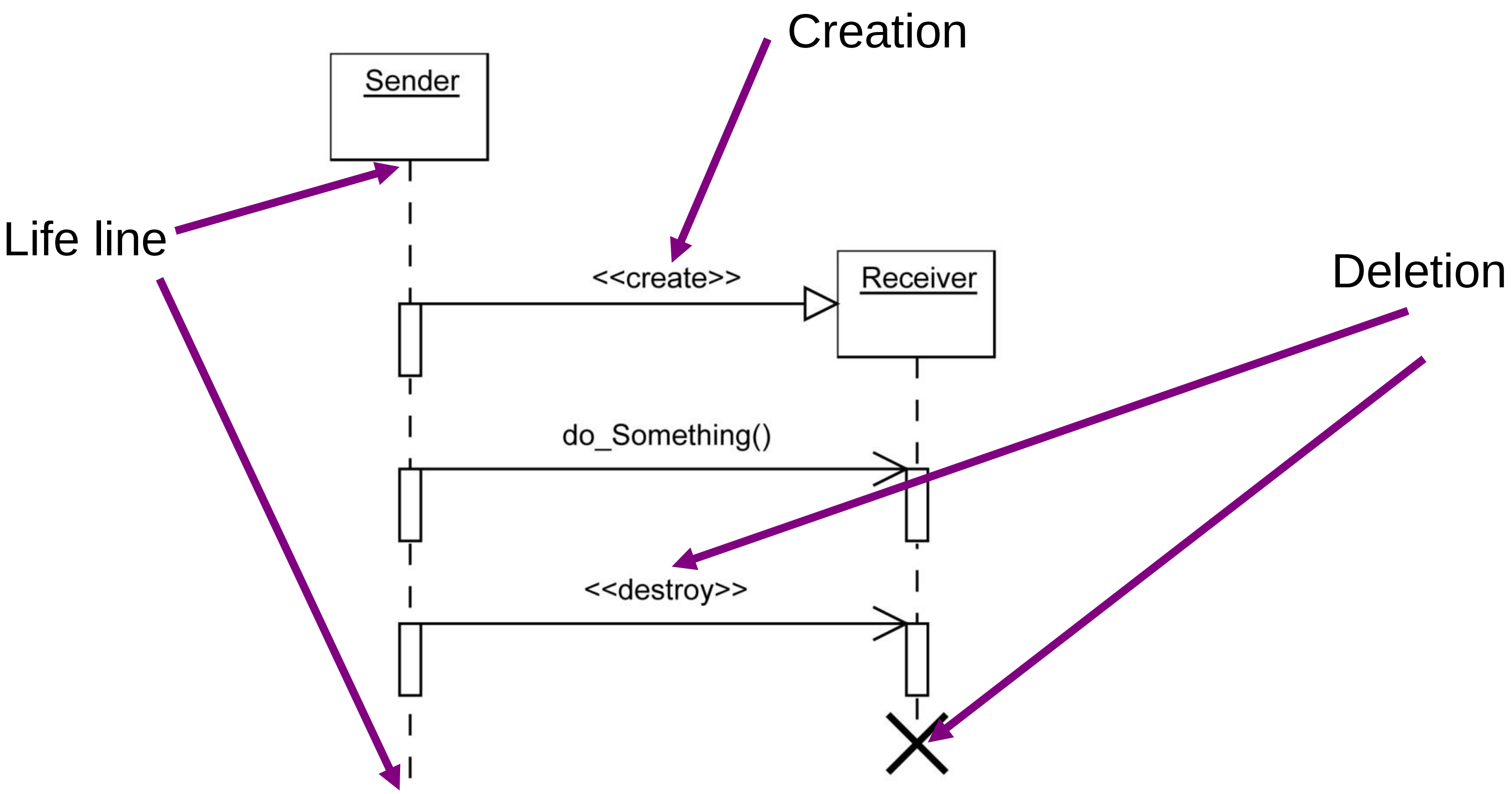
Alternate Message Reception

Recursion

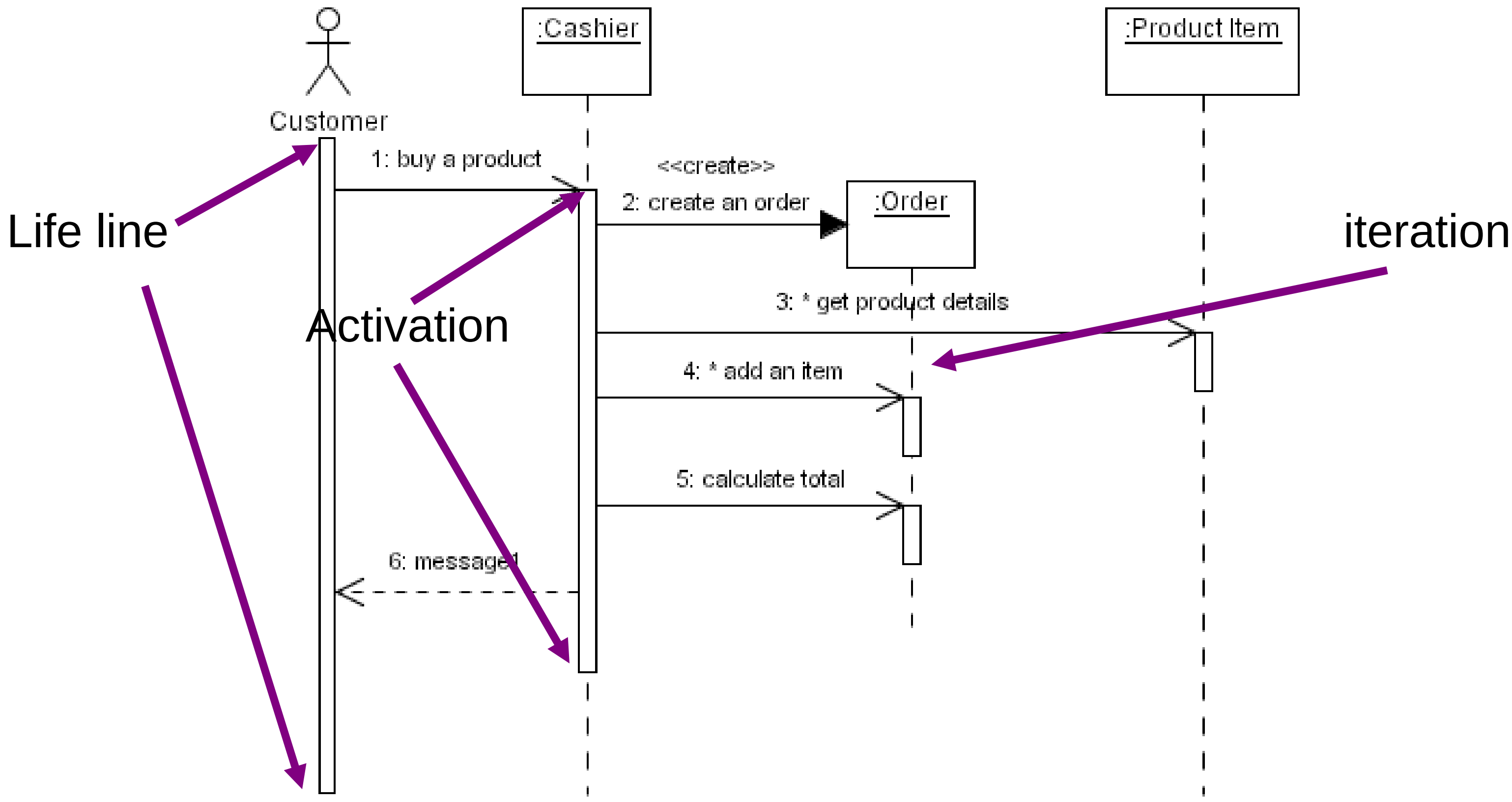


Recursion

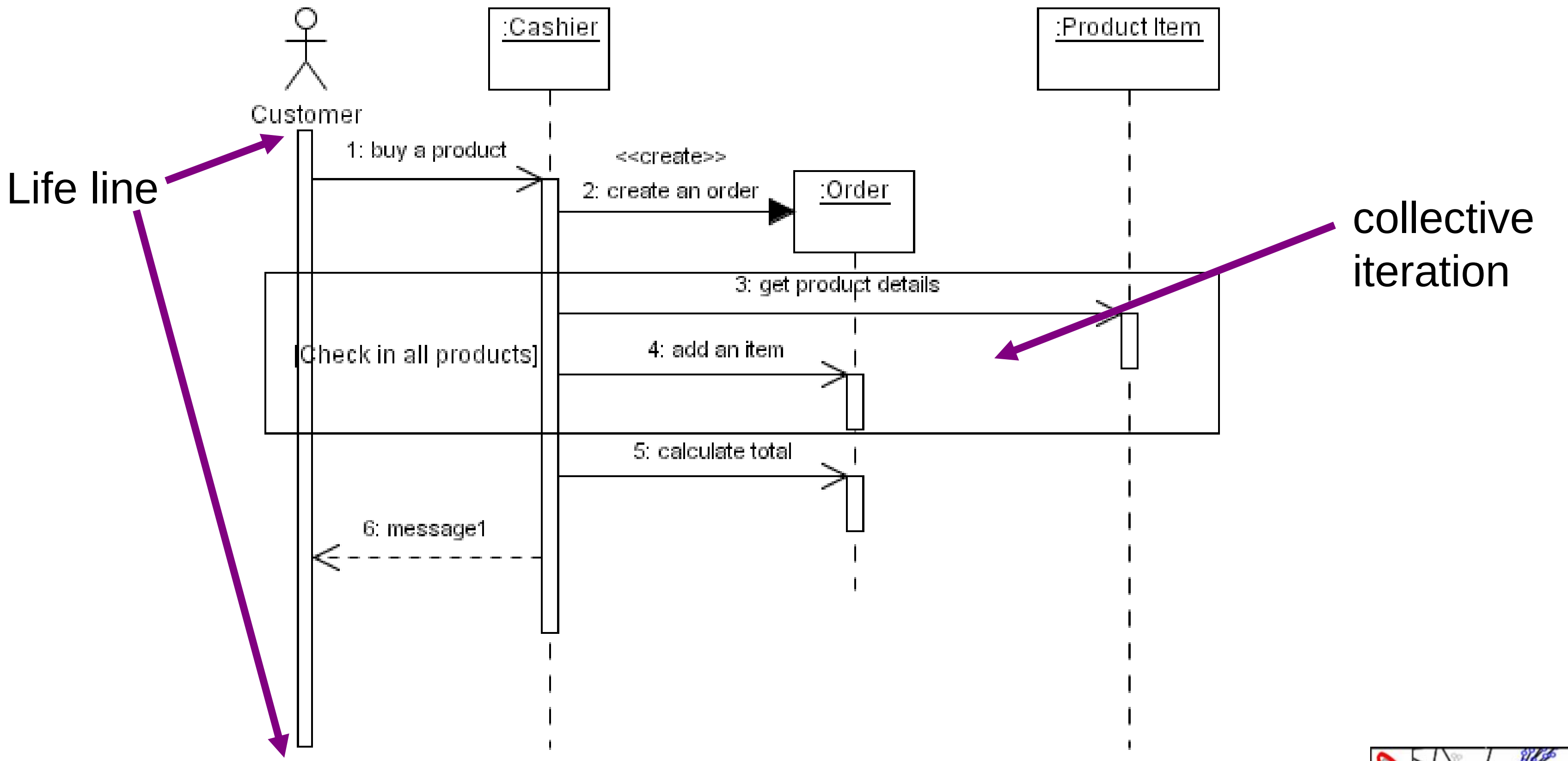
Example


Creation
Sender
Life line
<<create>>
Receiver
Deletion
do_Something()
<<destroy>>

Example

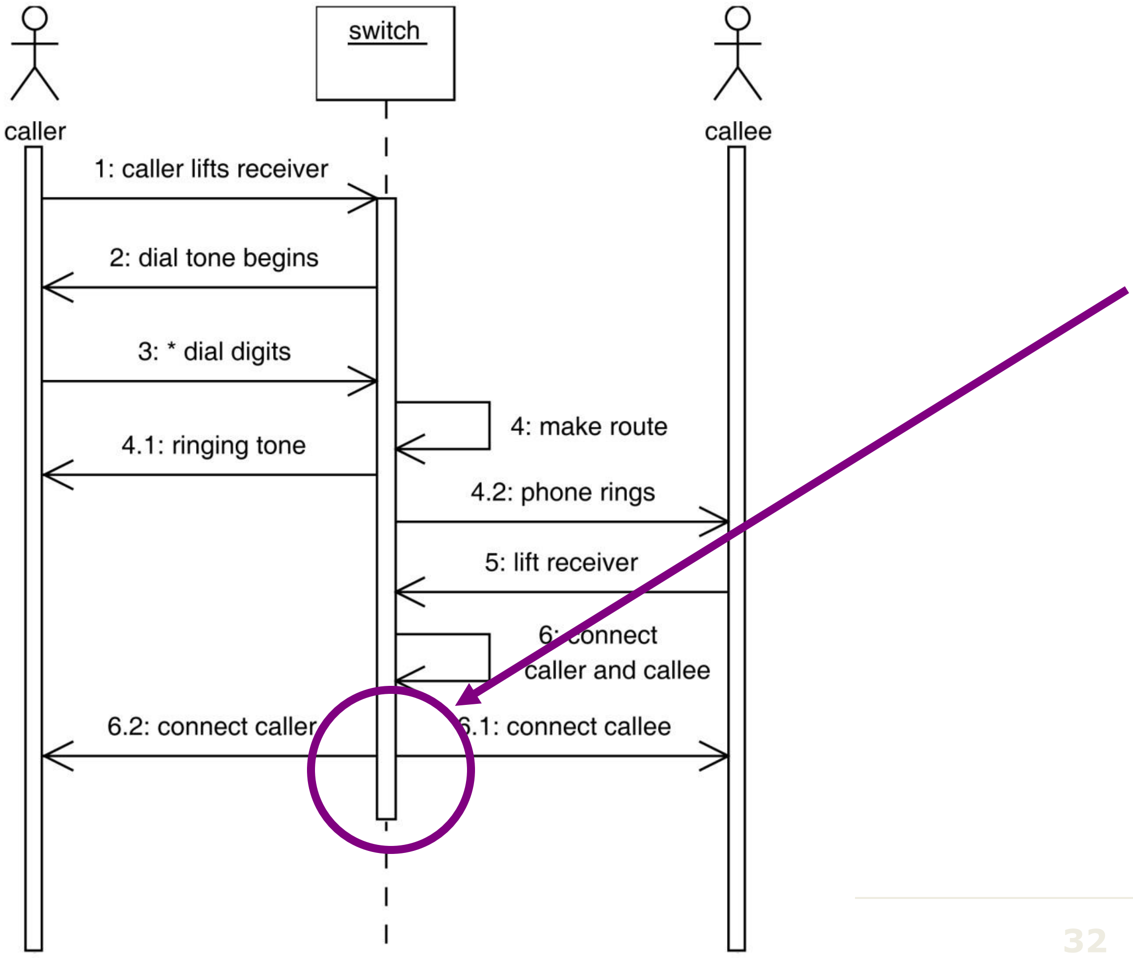

:Cashier
:Product Item
Customer
1: buy a product
<<create>>
2: create an order
:Order
Life line
iteration
3: * get product details
Activation
4: * add an item
5: calculate total
6: message1

Example



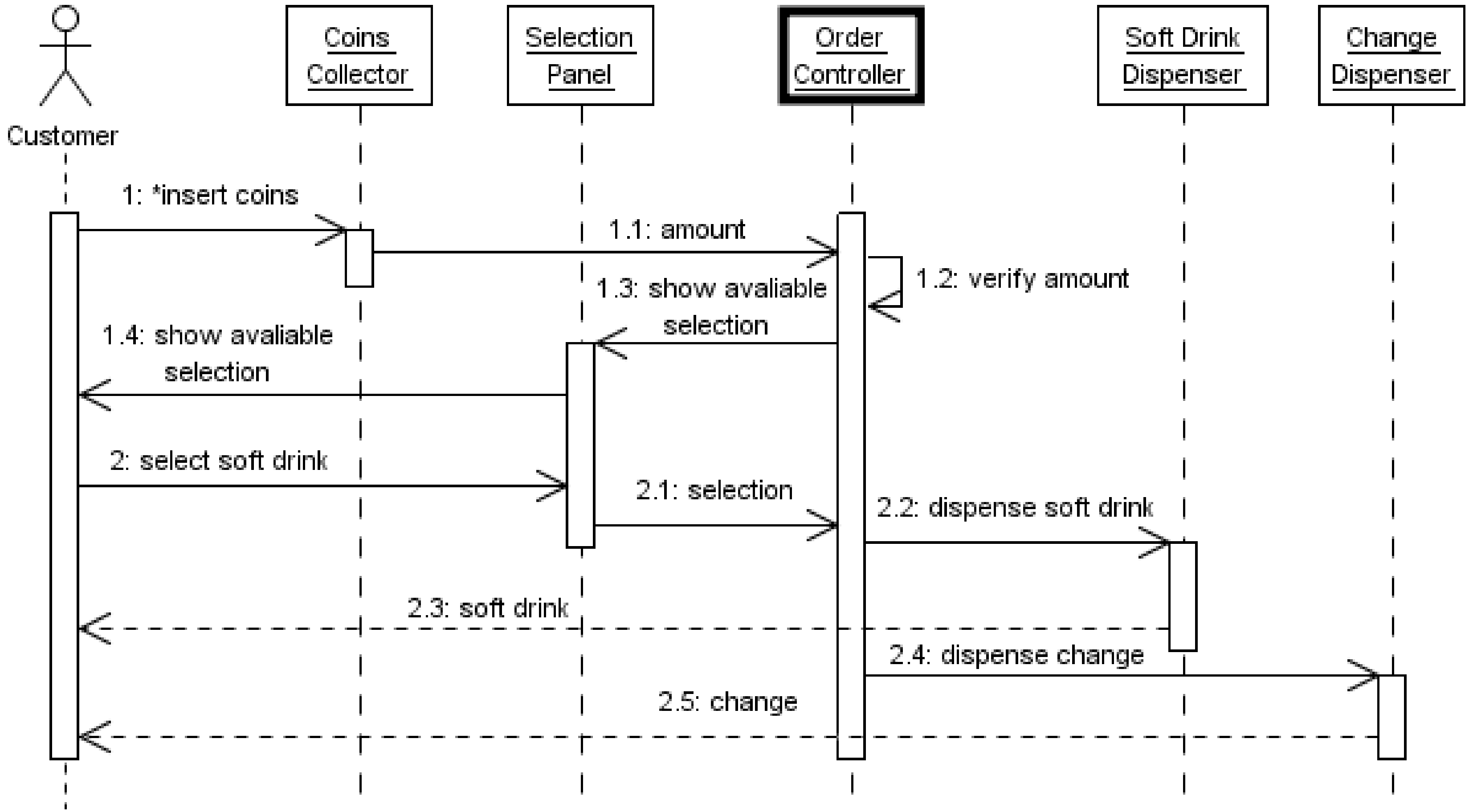
Customer
:Cashier
:Product Item
:Order
1: buy a product
<<create>>
2: create an order
3: get product details
[Check in all products]
4: add an item
5: calculate total
6: message1
Life line
collective
iteration

Example



Concurrent
Branch

Example - A Soft Drink Vending Machine



System Decomposition Principles

- It is important to decompose a system to improve
  - Model understandability
  - Inference complexity
  - Easy of implementation
- General guidelines
  - Based on “engineering practice”

Rules for System Decomposition

- All non-trivial systems are hierarchical
  - E.g., bio-systems
  - Galaxies, super-clusters, solar systems
- Are there rules for system decomposition?
  - Mathematics are only now being developed
  - Topology is understood
  - Functional decomposition is not understood

"The Prime Directive"

Partition software so that:

- each component is **cohesive** - does only one operation
- each component has **narrow coupling** with other components
- each component has **low complexity**
- each component can be nearly **exhaustively tested**
- each component is **easily understood**
- correct operation is based on satisfaction of, at most, **a few assertions**

CCSL Cork Complex Systems Lab

Prime Directive

Keep it Short and Simple - the KISS principle

Applying the KISS principle is the most important step in developing correct components.

Applying the Prime Directive

- We often deal with very large, complex systems in our professional careers. How do we apply the KISS Principle?

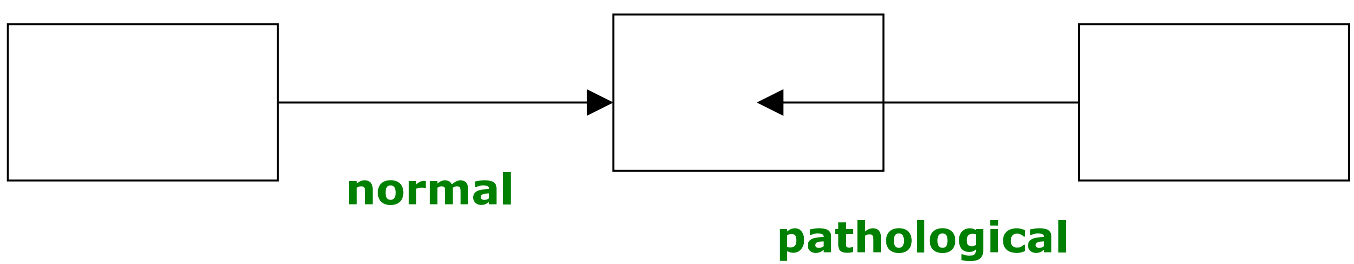
- Divide and Conquer!
  - Partition into an executive an a set of server modules.
  - Each server is focused on a single activity.
  - Higher level modules can use the services of lower level modules.
  - Higher level modules implement the required behavior of the system and so are not likely to be reusable. They are application artifacts.
  - Lower level modules implement solution-side functionality and can be widely reused when we design with foresight.

Structured Design

- Early work of software design (from 1979) that presented concepts such as cohesion, coupling, and encapsulation.
  - "Fundamentals of a Discipline of Computer Program and Systems Design"
    - by Edward Yourdon and Larry Constantine

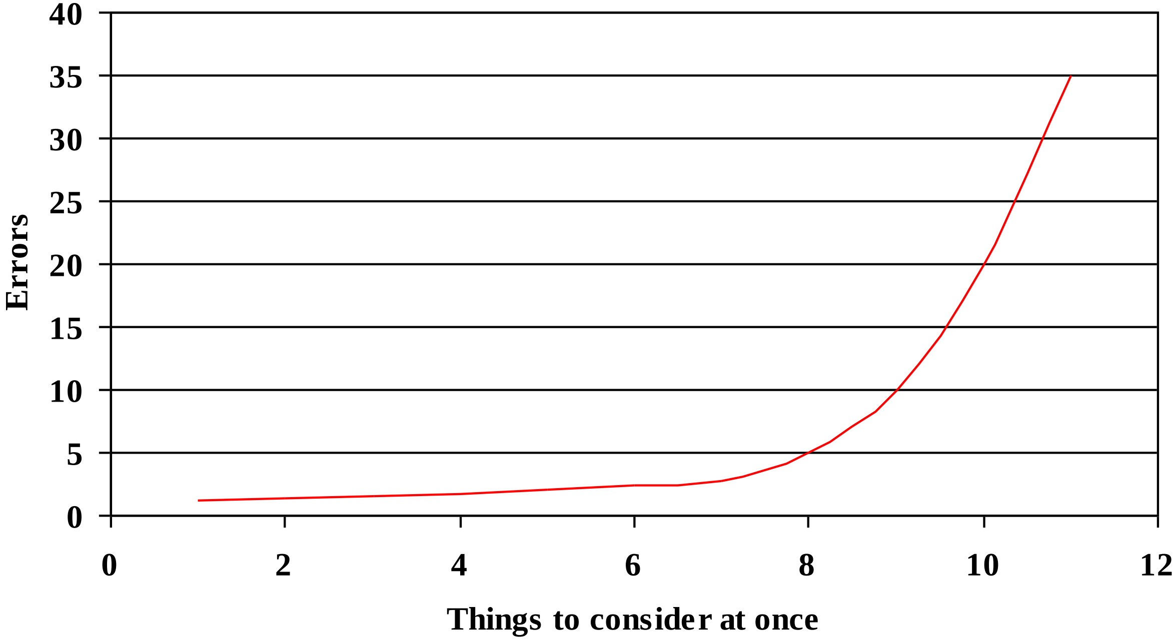
- Modules are not the same as for Parnas:
  - Module: A lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier.
    - A function, a procedure, a method

- **Normal** and **pathological** connections between modules:



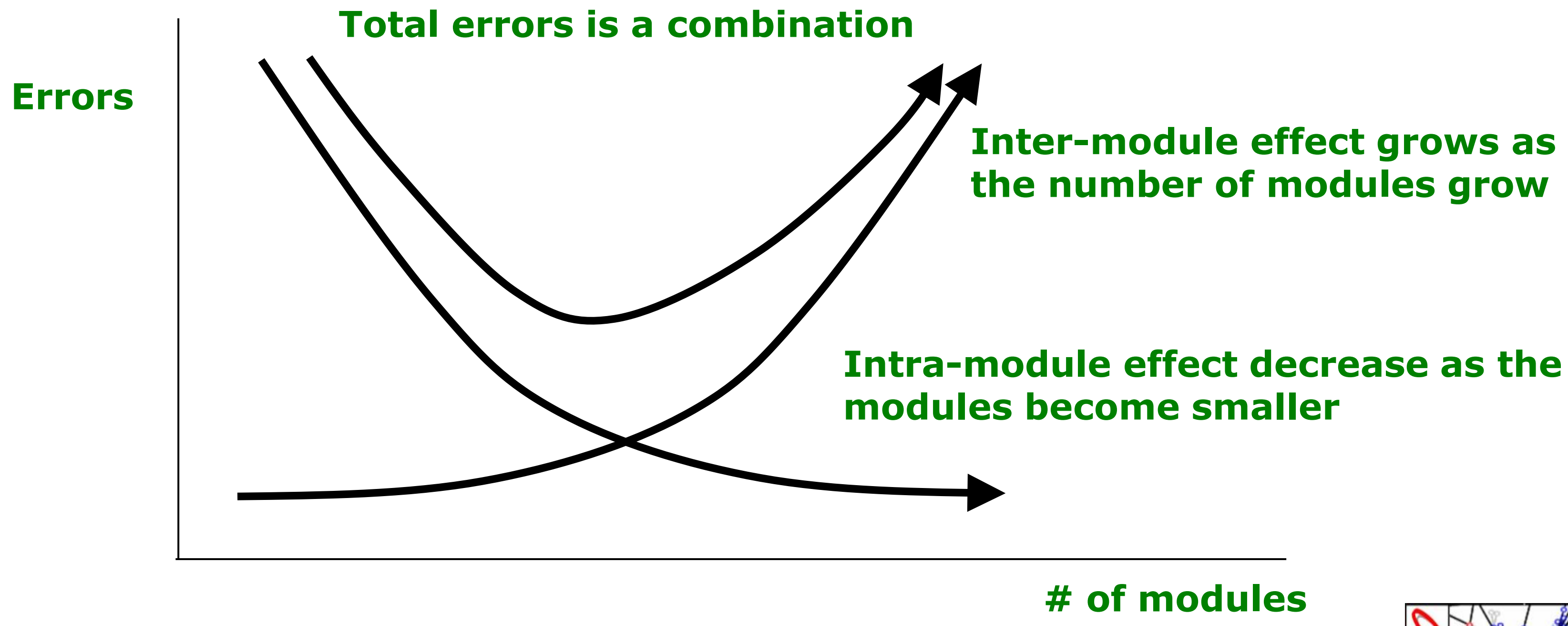
Human limitations on dealing with complexity

- George Miller: *The Magical Number Seven, Plus or Minus Two*
  - Can't keep track of too many things at the same time
  - Yourdon: Maximum number of subroutines called by a routine should be 5-9.



Two kinds of complexity

- Intra-module complexity
  - Complexity within one module
- Inter-module complexity
  - Complexity of modules interacting with one another



Overall cost

- The overall cost of a system depends on both:
  - The cost of production (and debugging)
  - And the cost of maintenance
    - Both are approximately equal for a typical system

- These costs are directly related to the complexity of the code
  - Complexity injects more errors and makes them harder to fix
  - Complexity requires more changes and makes them harder to effect

- Complexity can be reduced by breaking the problem into smaller pieces
  - (So long as the pieces are relatively independent of one another)

- But eventually the process of breaking pieces into smaller pieces creates more complexity than it eliminates.
  - 1970’s: Happens later than most designers would like to believe
  - 2000’s: Happens sooner than most designers would like to believe

Design approach

- Therefore, there is some optimal level of sub-division that minimizes complexity
  - But to reach it you need your judgment

- Once you know the right level, the key decision is to choose **how** to divide:
  - Minimize coupling between modules
    - Reduces complexity of interaction
  - Maximize cohesion within modules
    - Keeps changes from propagating
  - Duals of one another

Coupling

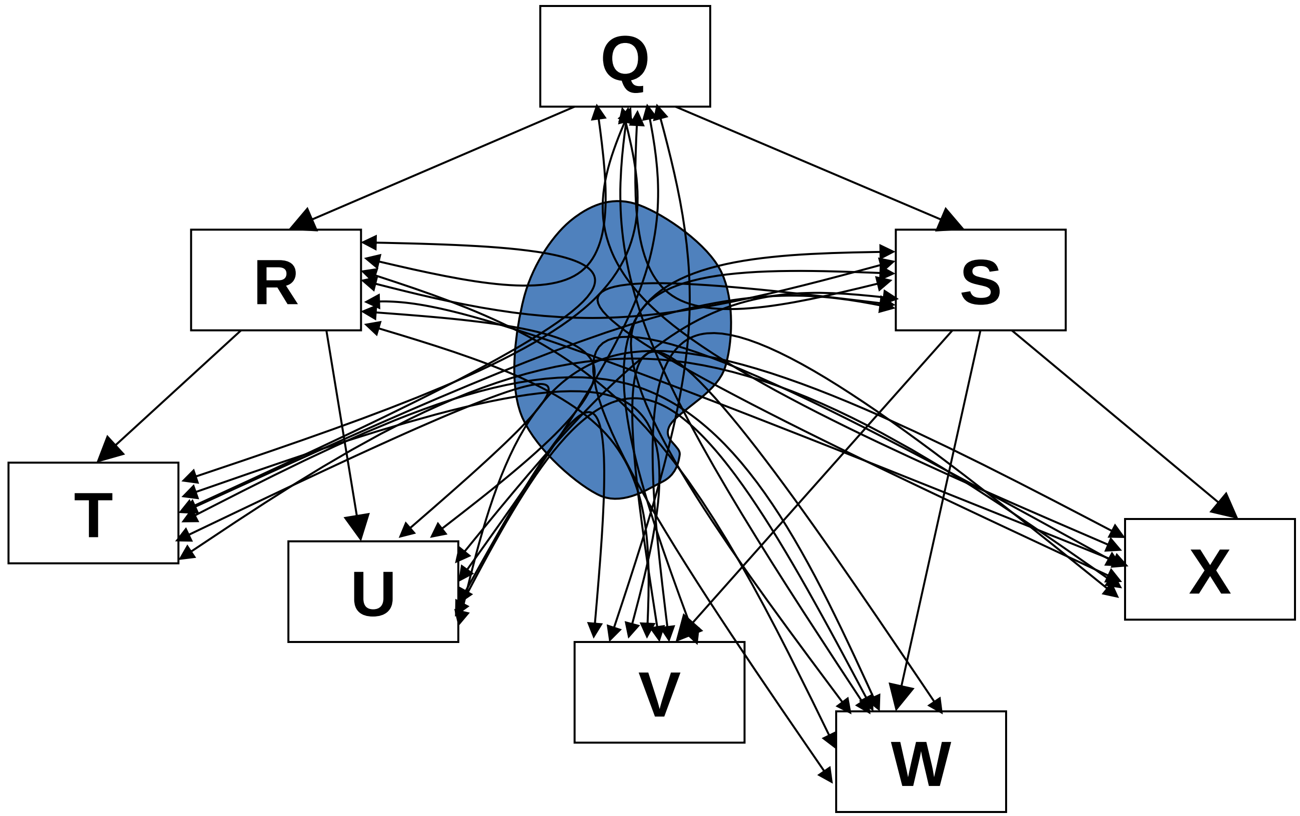
- Two modules are **independent** if each can function completely without the presence of the other
  - They are decoupled, or uncoupled
- Highly coupled modules are joined by many interconnections and dependencies
  - And loosely coupled modules have a few interconnections and dependencies
- Goal: Minimize coupling between modules in a system
  - Coupling translates into "the probability that in coding/modifying/debugging module A we will have to take into account something from module B"
- Note that a system that has only one module (function) is absolutely uncoupled
  - But that's not what we want!
  - (We'll analyze *cohesion*, coupling's complement, later)

Influences on coupling

- Type of connection
  - Minimally connected: parameters to a subroutine
  - Pathologically connected: non-parameter data references
- Interface complexity
  - Number of parameters/returns
  - Difficulty of usage
- Information flow
  - Data flow: Passing data is handled uniformly
  - Control flow: Passing of flags governs how data is processed
- Binding time
  - More static = more complex
    - E.g., literal “30” vs. pervasive constant N_STUDENTS, vs.

Common-environment coupling

- A module writes into global data
- A different module reads from it (data or, worse, control)



Cohesion

- While minimizing coupling, we must also maximize cohesion
  - How well a particular module "holds together"
    - The cement that holds a module together
  - Answer the questions:
    - Does this make sense as a distinct module?
    - Do these things belong together?

- Best cohesion is when it comes from the problem space, not the solution space
  - Echoed years later in OOA/OOD

Levels of lack of cohesion (roughly from worst to best)

- Coincidental
  - No reason for doing two things in the same routine
    - double computeAndRead(double x, char c);
- Logical
  - Similar class of things that still should be separated
    - char input(bool fromFile, bool fromStdIn);
- Temporal
  - The fact that things happen one after the other is no excuse to put them in the same routine
    - void initSimulationAndPrepareFirst();
- Procedural
  - Operations are together because they are in the same loop or decision process, but no higher cohesion exists
    - typeDecide(m); // Decide type of plant being simulated and perform simulation part 1

Cork Complex Systems Lab

Levels of lack of cohesion (roughly from worst to best) (cont)

- Communicational
  - Procedures that access the same data are kept together
    - void printReports(data x); // Outputs day report and monthly summary
- Sequential
  - A sequence of steps that take the output from the previous step as input for the next step
    - string compile(String program) {parse, semantic analysis, code generation}
- Functional
  - That which is none of the above
  - Does one and only one conceptual thing
  - Equivalent to information hiding
    - double sqrt(double x);

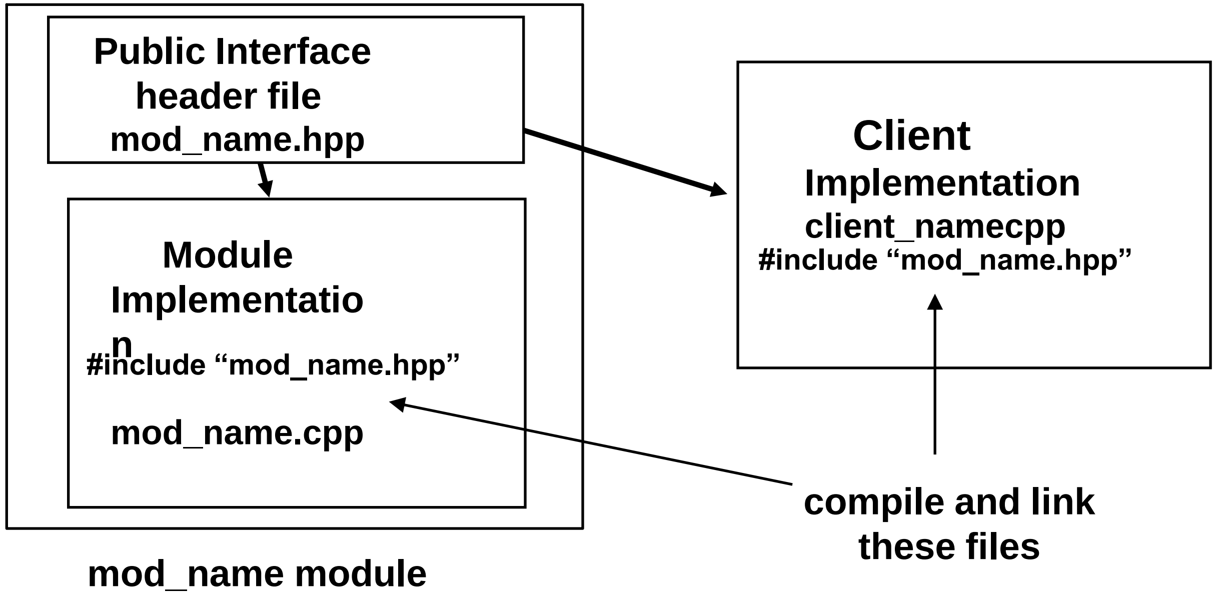
Practical Issues with Modularity

- Subdividing code
- Interface specification given modules
- Modular component design and reUse

Modularity

- The purpose of a module or class is to implement a small, simple logical model.
- The purpose of modularization is to build a software system out of cohesive, reliable modules.
- Modularization consists of dividing a program into modules which can be compiled separately. C++ performs type checking across module boundaries.

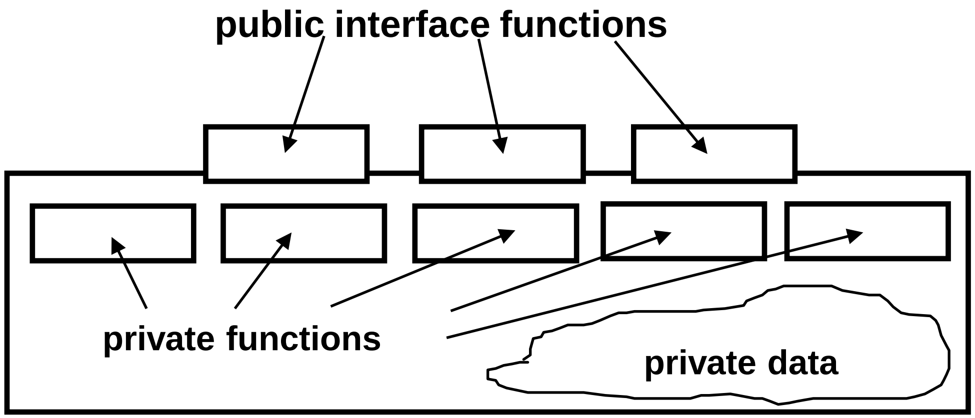
- Modules in C# and C++ are simply separately compiled files.
- We place module interface declarations in header files.
- Module implementations are placed in separate files which include the header file at compilation time via a preprocessor #include "mod_name.hpp" directive.



Encapsulation and the Information Cluster

"An information cluster is a set of [functions] used for every access to data that has a complex structure, sensitive security, or device dependence."

Meilir Page-Jones, The Practical Guide to Structured Systems Design, Yourdon Press, 1988



Information Clustering

- The major benefit of this organization is that knowledge of specific layout and implementation details is hidden from clients, who have access only to a public interface.
-

  The internal data could be reorganized, to improve performance say, without adversely affecting any of its clients provided that the public interface remains fixed.

- Classes are simply patterns for information clusters. Objects are their instances, defined in memory.
- Modules are information clusters with only one instance.