COMP201 – Software Engineering I
Revision Lecture

Lecturer: Dr. T. Carroll

Office: G.14
Email: Thomas.Carroll2@Liverpool.ac.uk
See Vital for all notes

A Whistle-Stop Tour…

- Software Process Models
- Requirements
- System Models
  - UML
  - Petri Nets
  - Mealy and Moor Machines
- Characteristics of Good Software Design
- System Structure
- Testing

Software Process Models

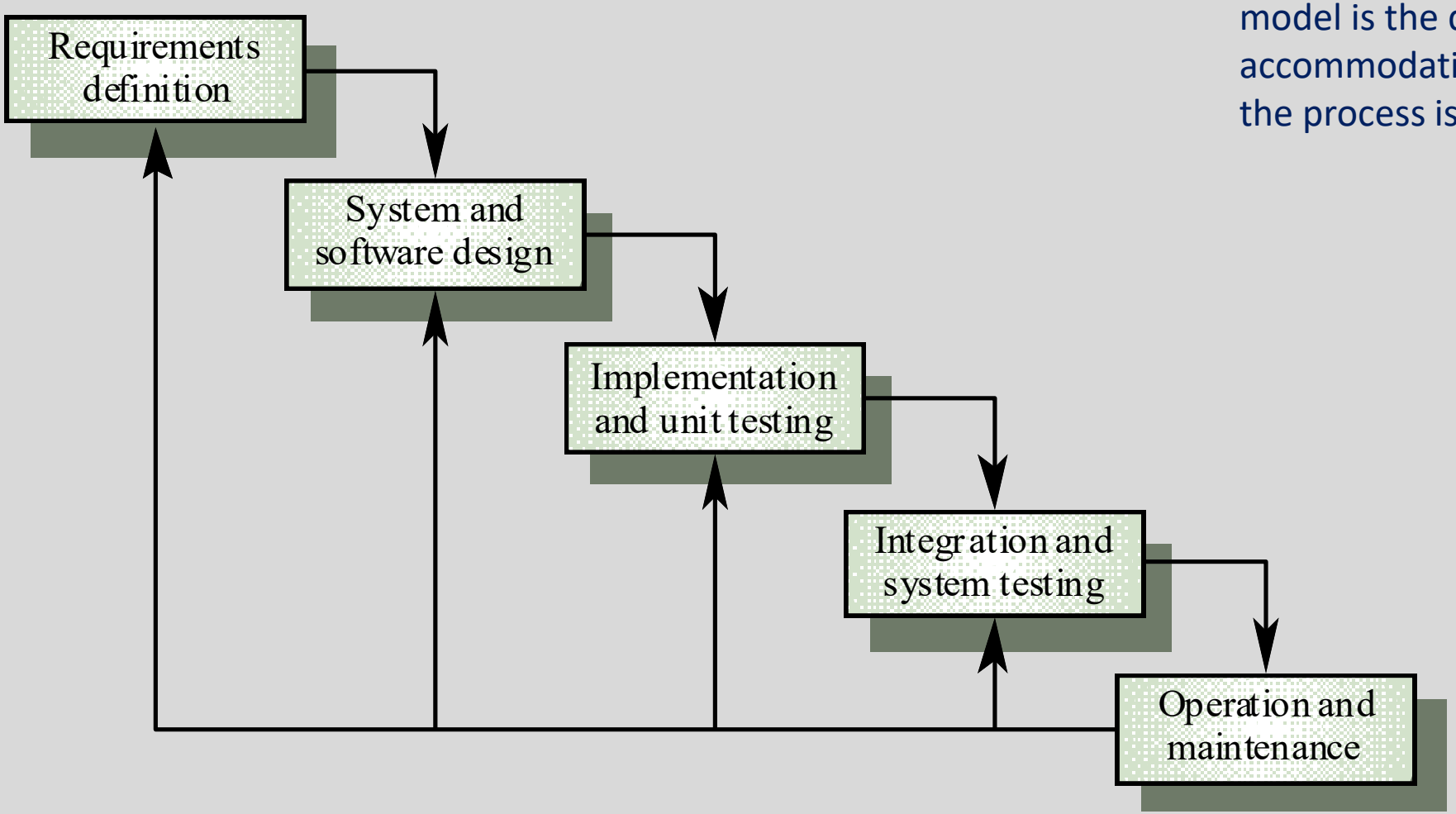
Software Processes

- Life Cycle – the process of building a product
- Coherent sets of activities for
  - Specifying,
  - Designing,
  - Implementing and
  - Testing software systems

Software Process Models

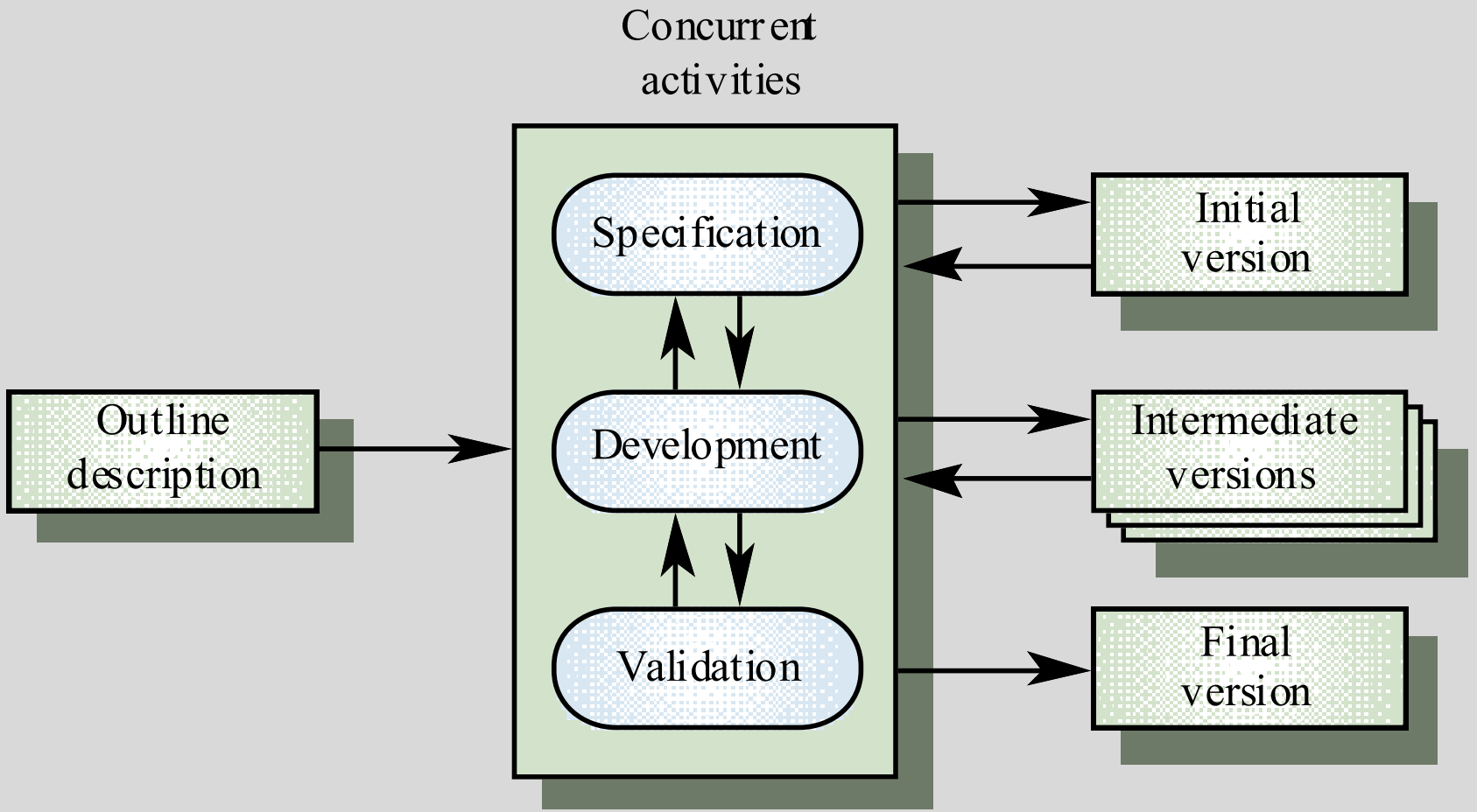
- **The Waterfall Model**
  - Separate and distinct phases of specification and development
- **Evolutionary Development**
  - Specification and development are interleaved
- **Formal Systems Development**
  - A mathematical system model is formally transformed to an implementation
- **Iterative development (most widely used)**
  - The system is built up in a series of steps

Waterfall Model

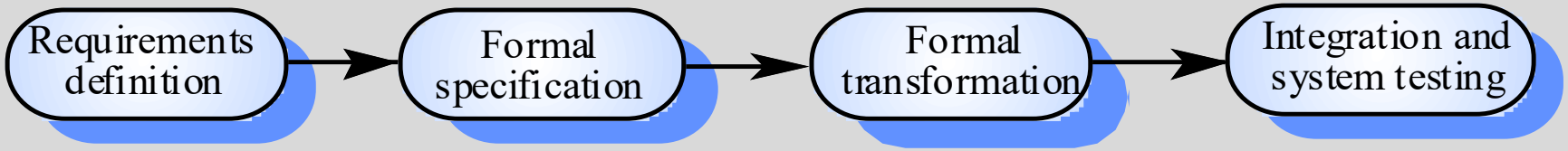


The drawback of the waterfall model is the difficulty of accommodating change after the process is underway

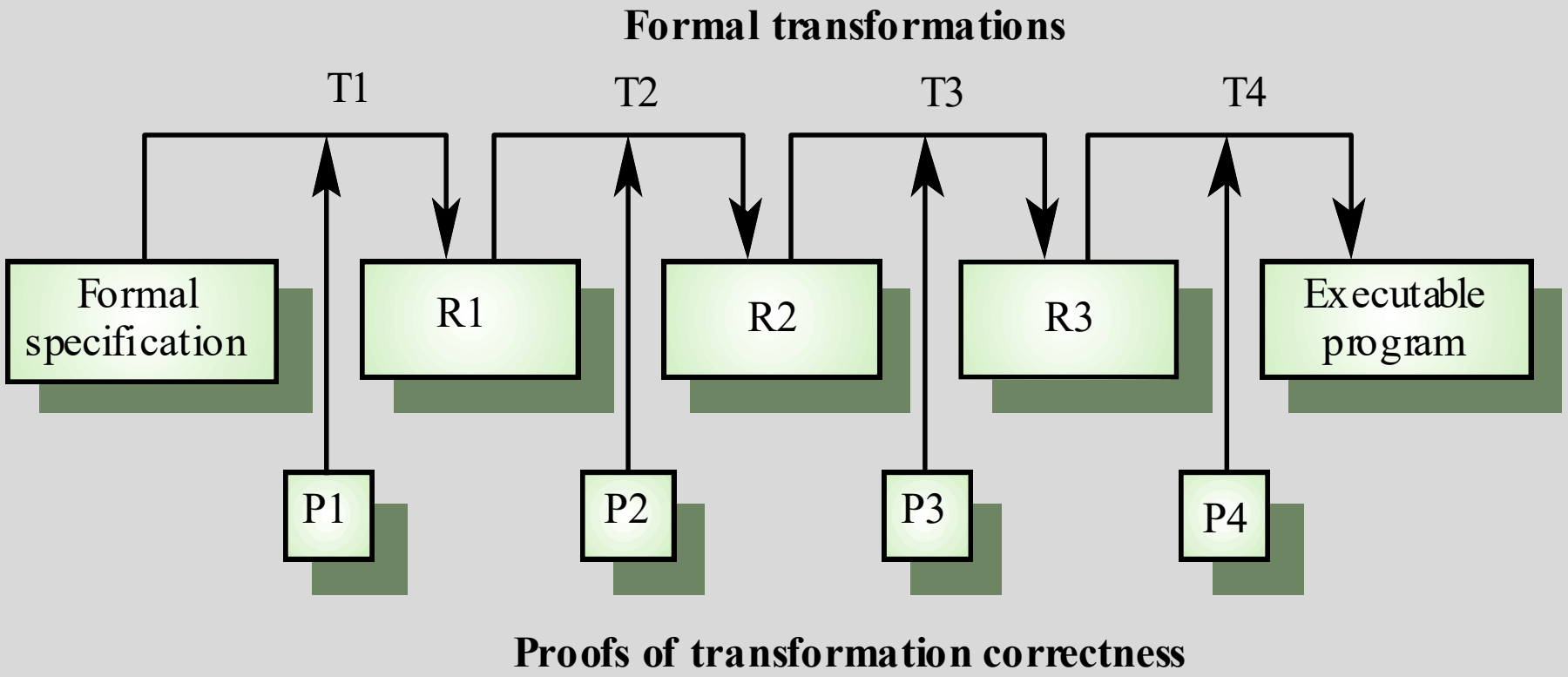
Evolutionary Development



Formal Systems Development



Formal Transformations



Requirements

What Are Requirements?

- It may range from a **high-level** abstract statement of a service or of a system constraint to a **detailed** mathematical functional specification
- This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation
  - May be the basis for the contract itself - therefore must be defined in detail
  - Both of these statements may be called requirements

Functional Requirements

- Statements of services the system should provide
- How the system should react to particular inputs
- How the system should behave in particular situations
- Depend on the type of software, expected users and the type of system where the software is used
- **Functional user requirements** may be high-level statements of what the system should do **BUT functional system requirements** should describe the system services in detail

Non-Functional Requirements

- Constraints on the services or functions offered by the system such as:
  - timing constraints
  - constraints on the development process, standards, etc.
- Usually defined on the system as a whole
- **Process requirements** may also be specified, mandating a particular CASE system, programming language or development method
- **Non-functional requirements** may be more critical than functional requirements. If these are not met, the system is useless
- Eg:
  - reliability, response time and storage requirements
  - key length for encrypting secure email must be >=256 bits

System Models

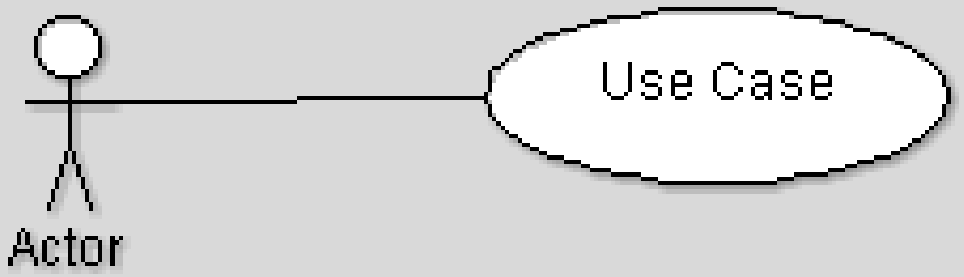
Use Case Diagrams – Scenarios using UML

- Functional modelling of requirements
- Show the external view of the system (we don't look at internal behaviour)
- Show the system relative to different users of the system
- A set of use-cases should describe all possible interactions with the system.
  - Actors involved
  - Details of the interaction
- **Sequence diagrams** may be used to add detail to use-cases by showing the sequence of event processing in the system

Actors and Use Cases

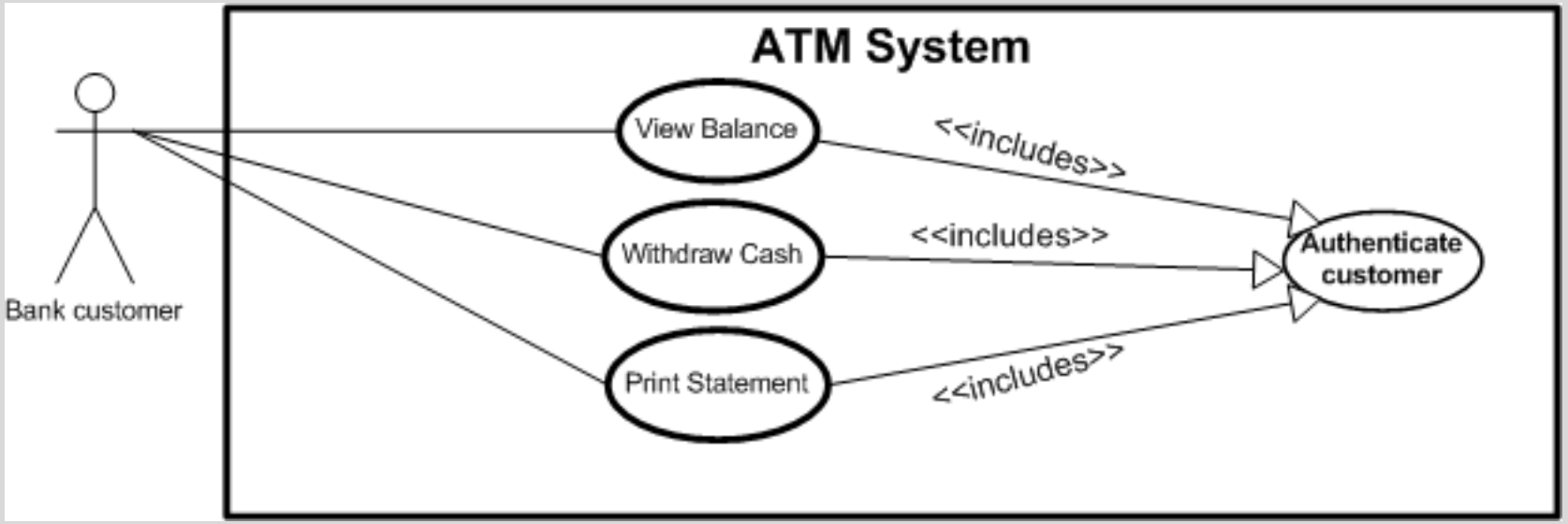
- An **actor** is a user of the system acting in a particular role
  - Something external to the system
  - Can be human (Customer, Cashier, …)
  - Can be non-human (Robotic Arm, Sensor, …)
- A **use-case** is a task which the **actor** needs to perform with the help of the system
  - e.g., find details of a book, print receipt, …
- The details of each use case should also be documented by a use case description
  - **Print receipt** – A customer has paid for an item via a valid payment method. The till should print a receipt indicating the current date and time, the price, the payment type and the member of staff who dealt with the sale.
  - **[Alternate Case]** – No print paper available – Print out “Please enter new till paper” to the cashier’s terminal. Try to print again after 10 seconds.

Example – Actor and Use Case



<<includes>> relation

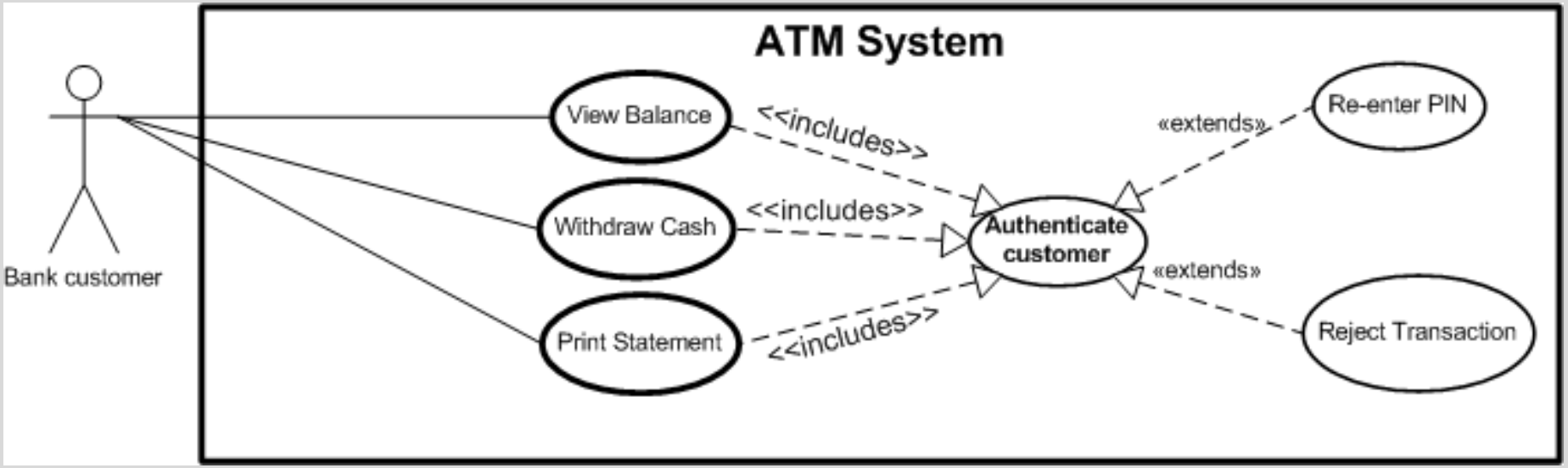
- A use case **always** makes use of another



Note the direction of the arrows!

<<extends>> relation

- A use-case has two or more significantly different outcomes
- **Extends** the use case to a main use case and one or more subsidiary cases.
  - Use cases are **sometimes** needed

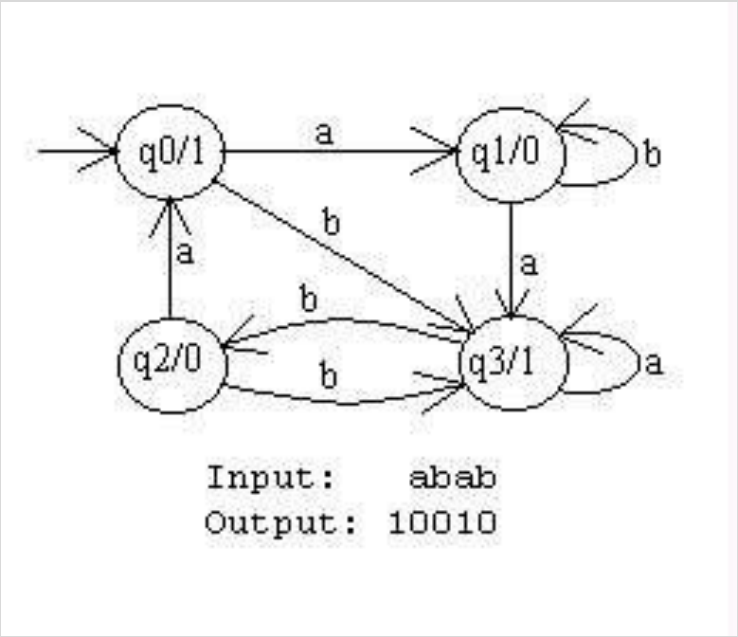


Note the direction of the arrows!

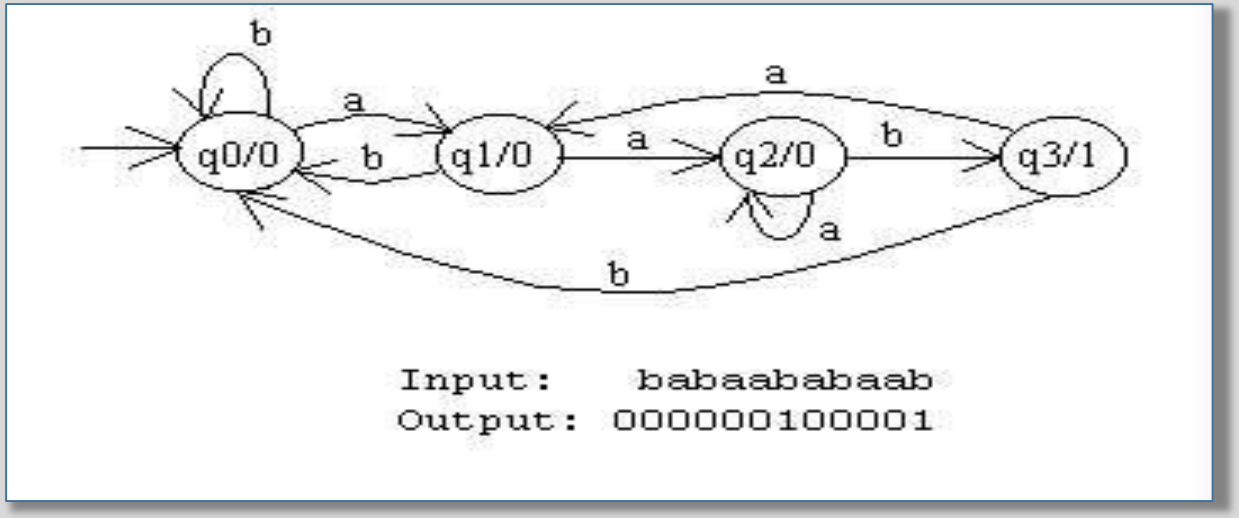
Moore Machine

- A **Moore machine** is a FSA with two extra attributes.
  1. It has TWO alphabets, an input and output alphabet.
  2. It has an output letter associated with each state.

  The machine writes the appropriate output letter as it enters each state.



Example - Moore Machine

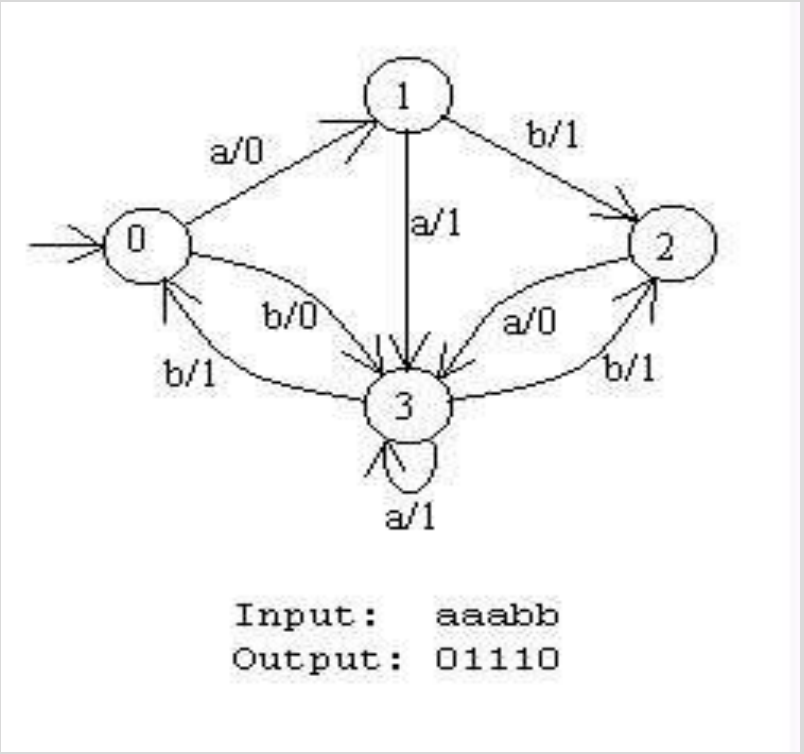


This machine might be considered as a "counting" machine.

The output produced by the machine contains a 1 for each occurrence of the substring **aab** found in the input string.

Mealy Machine

- **Mealy machines** are computationally equivalent to Moore machines
- However, Mealy machines move the output function from the state to the transition.
- This turns out to be easier to deal with in practice, making Mealy machines more practical.



A Mealy Machine Produces Output on a Transition

Transitions are labelled *i/o* where

- *i* is a character in the input alphabet and
- ***o*** is a character in the output alphabet.

The following Mealy machine takes the one's complement of its binary input. In other words, it flips each digit from a 0 to a 1 or from a 1 to a 0.

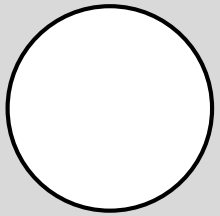
Input: 010110
Output: 101001

What is a Petri Net?

- A Model that uses Places, Tokens, Arcs and Transitions to **model processes**
- Presence of tokens denotes availability of resources (**the state** of the petri net)
- Transitions **Fire** and the state changes – different transitions can then fire

Petri Net Components

- Places
  - **Resources**/Conditions
  - Can hold Tokens
  - Infinite capacity (unless labelled)



- Tokens
  - **Availability** of a Resource
  - Condition is satisfied

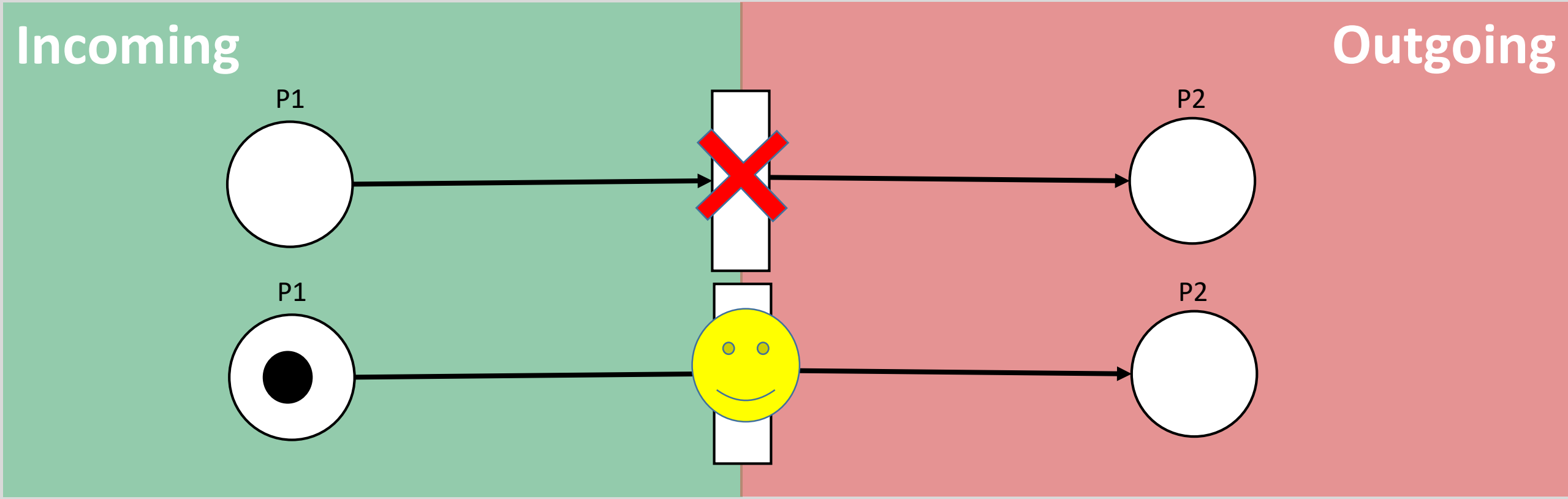
- Transitions
  - **Processes**
  - Consumes Tokens
  - Generates NEW Tokens

- Arcs
  - Shows **dependencies**
  - Connects Places with Transitions
  - Tokens travel along arcs
  - Unlabelled arc has capacity 1
  - Labelled arcs have the labelled capacity

Is The Transition Enabled?

A transition must be **enabled** for it to fire

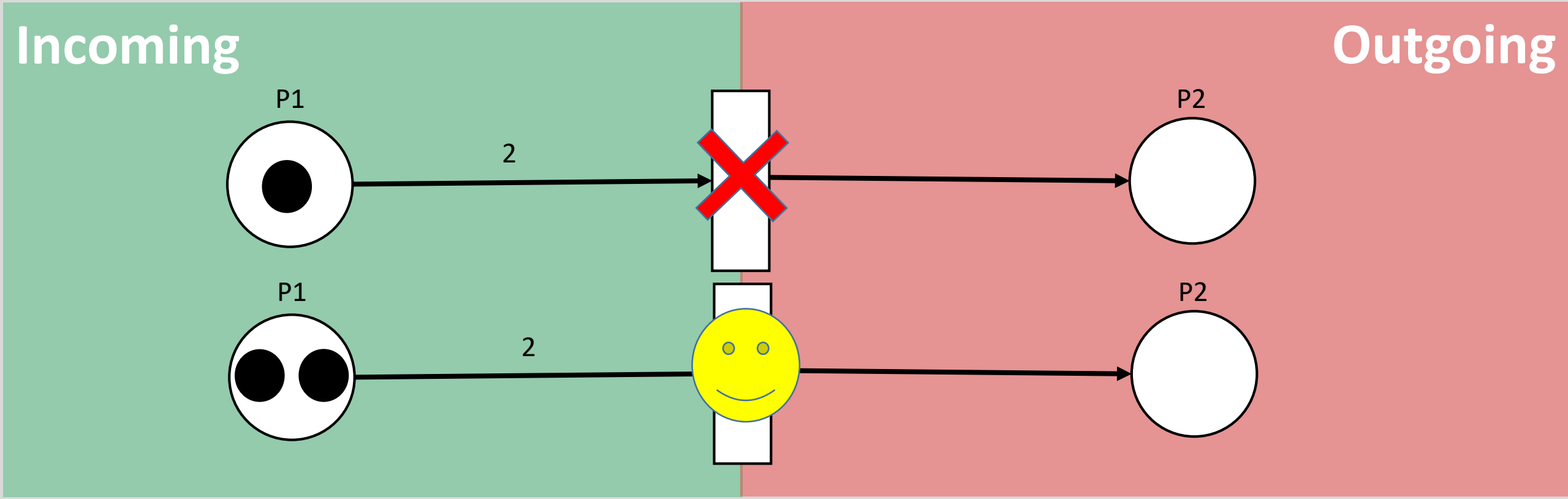
Each incoming place must have tokens **equal to or greater than** arc capacity



Is The Transition Enabled?

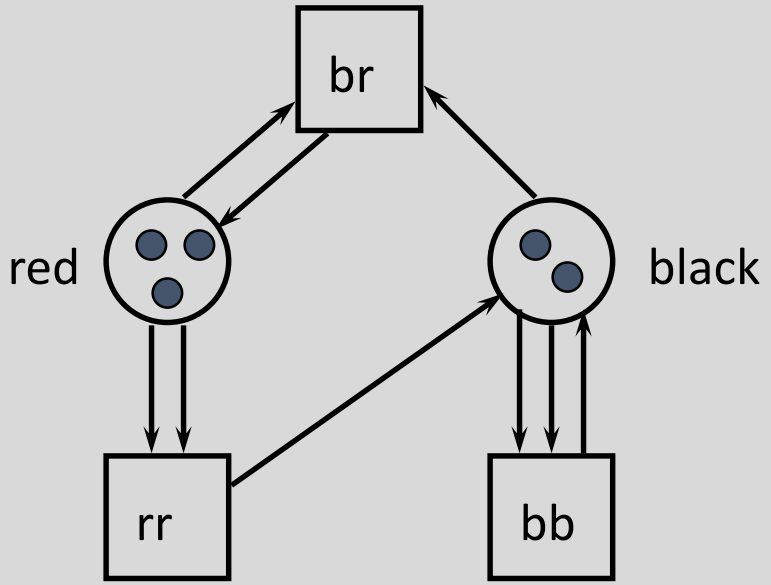
A transition must be **enabled** for it to fire

Each incoming place must have tokens **equal to or greater than** arc capacity

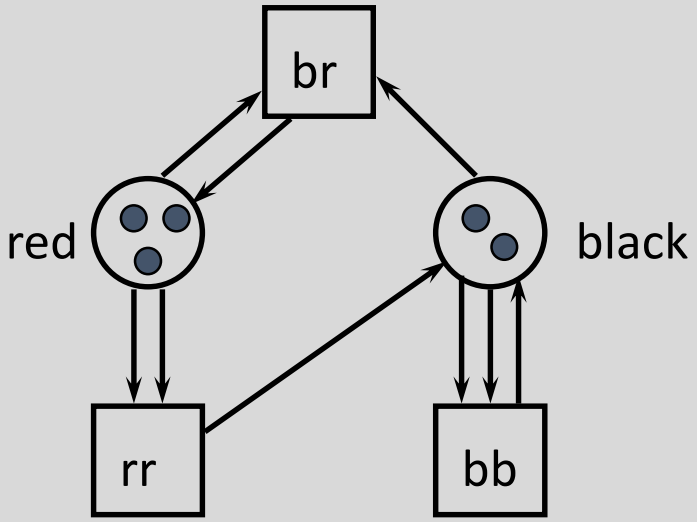


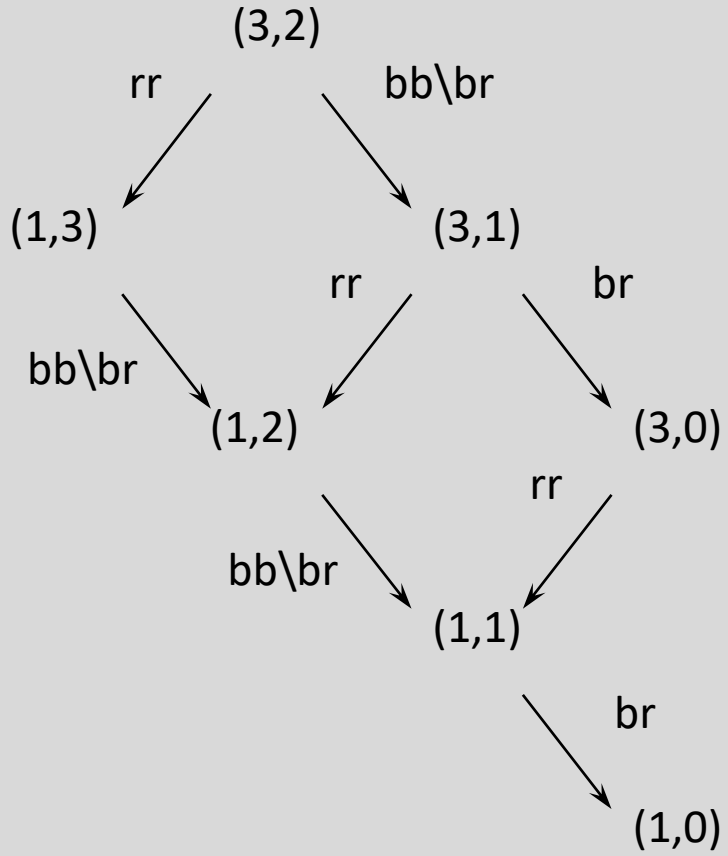
Notation

- If we write the places in some fixed order (red, black say), then we can use a tuple: (n,m) to denote the number of tokens in each corresponding place (n tokens in “red” and m tokens in “black”).
- The example below is thus in state (3,2).
- After firing transition “rr”, it will move to state (1,3) etc..



- 7 reachable states, 1 deadlock state.





Design

Coupling

- **Coupling** is a measure of the strength of the inter-connections between system components.
- **Loose coupling** means component changes are unlikely to affect other components.
  - Shared variables or control information exchange lead to **tight coupling** (usually bad).
  - Loose coupling can be achieved by:
    - state decentralization (as in objects)
    - component communication via parameters or message passing.

Coupling and Inheritance

- Object-oriented systems are loosely coupled because there is no shared state and objects communicate using message passing.
- **However, an object sub-class is coupled to its super-classes.**
- Changes made to the attributes or operations in a super-class propagate to all sub-classes.

Cohesion

- A measure of how well a component “fits together”
- A component should implement a single logical entity or function.
- Cohesion is a desirable design component attribute
  - when a change has to be made, it is localized in a single cohesive component.

Cohesion Levels

- Coincidental cohesion (weak)
  - Parts of a component are simply bundled together.
- Logical association (weak)
  - Components which perform similar functions are grouped.
- Temporal cohesion (weak)
  - Components which are activated at the same time are grouped.

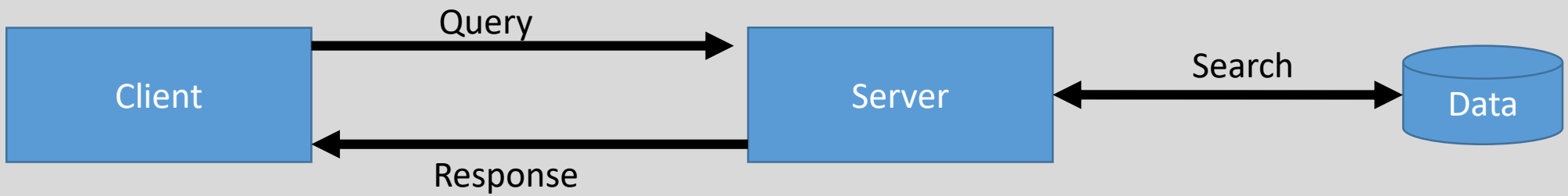
Cohesion Levels

- Communicational cohesion (medium)
  - All the elements of a component operate on the same input or produce the same output.
- Sequential cohesion (medium)
  - The output for one part of a component is the input to another part.
- Functional cohesion (strong)
  - Each part of a component is necessary for the execution of a single function.
- Object cohesion (strong)
  - Each operation provides functionality which allows object attributes to be modified or inspected.

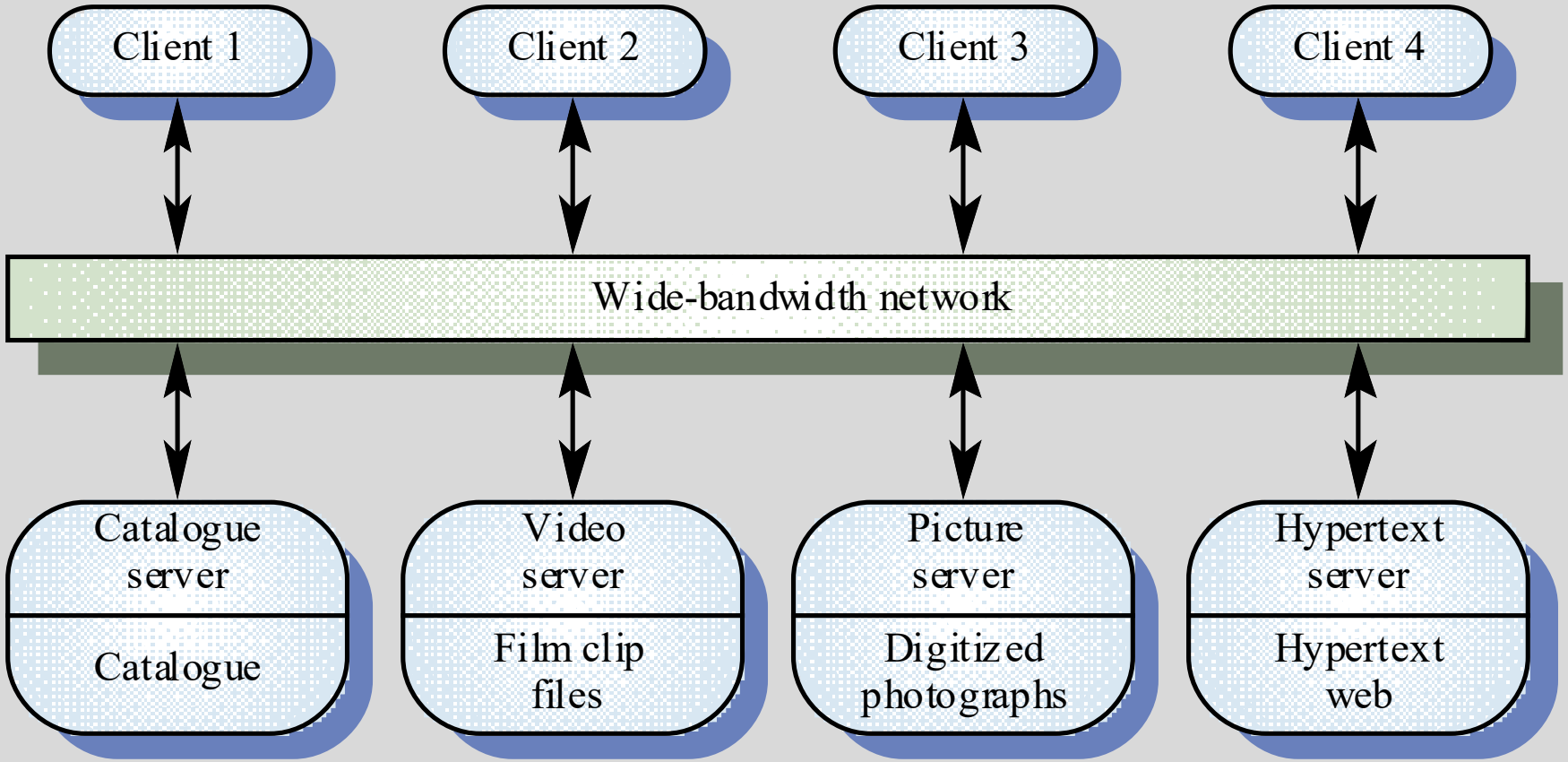
Structure

Client-Server Architecture

- Distributed system model
- Shows how data and processing is distributed across a range of components:
  - **Servers** provide specific services such as printing, data management, etc.
  - **Clients** call on these services
  - **Network** allows clients to access servers



Example: Film and Picture Library



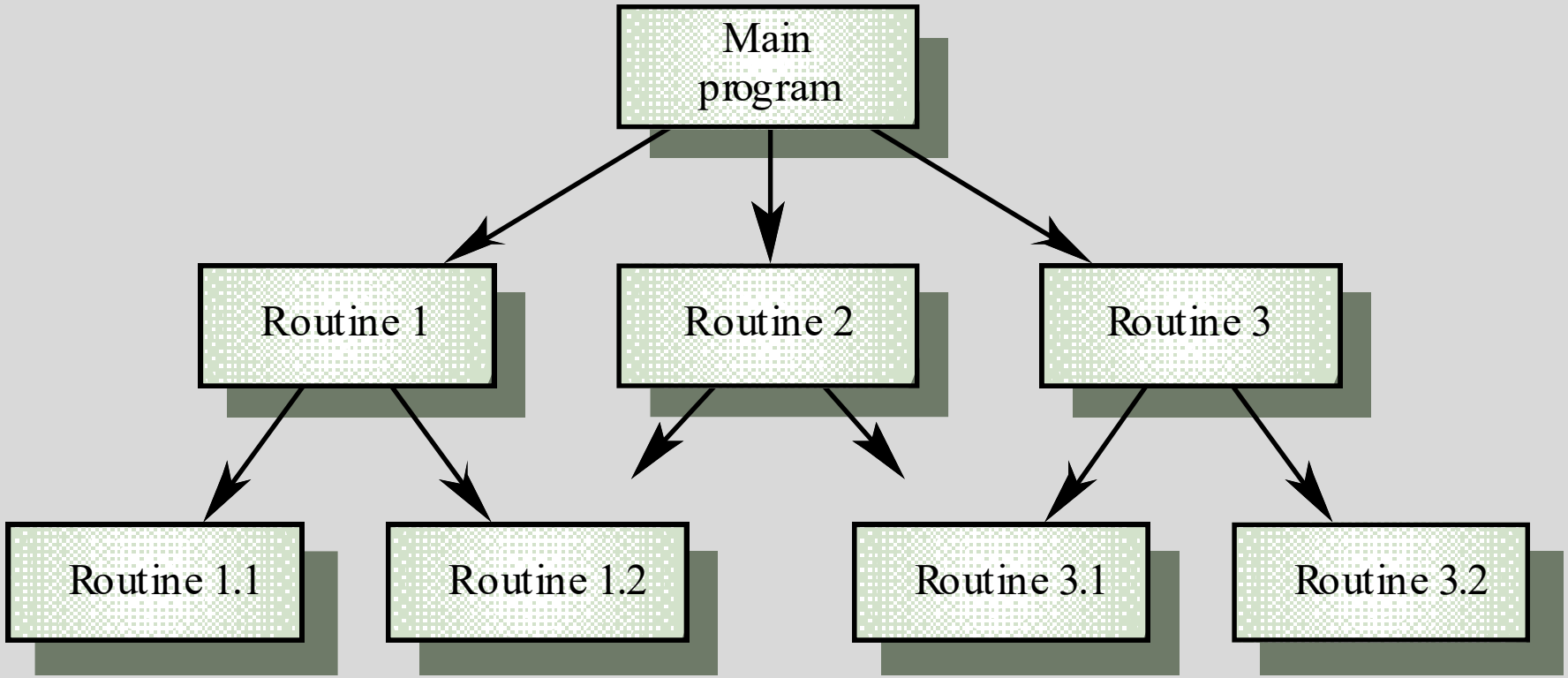
Client-Server Characteristics

- Advantages
  - ✓Distribution of data is straightforward
  - ✓Makes effective use of networked systems.
  - ✓Could get away with using cheaper hardware
  - ✓Easy to scale
- Disadvantages
  - × No shared data model
  - × Redundant management in each server
  - × No central register of names and services

Centralised Control

- A control sub-system takes responsibility for managing the execution of other sub-systems.
- There are two main types of centralised control models (sequential or parallel):
  - **Call-Return model (Top-Down subroutine model)**
    - Control starts at the top of a subroutine hierarchy and moves downwards. Applicable to **sequential systems**
    - Such a model is embedded into familiar programming languages such as C, Java …

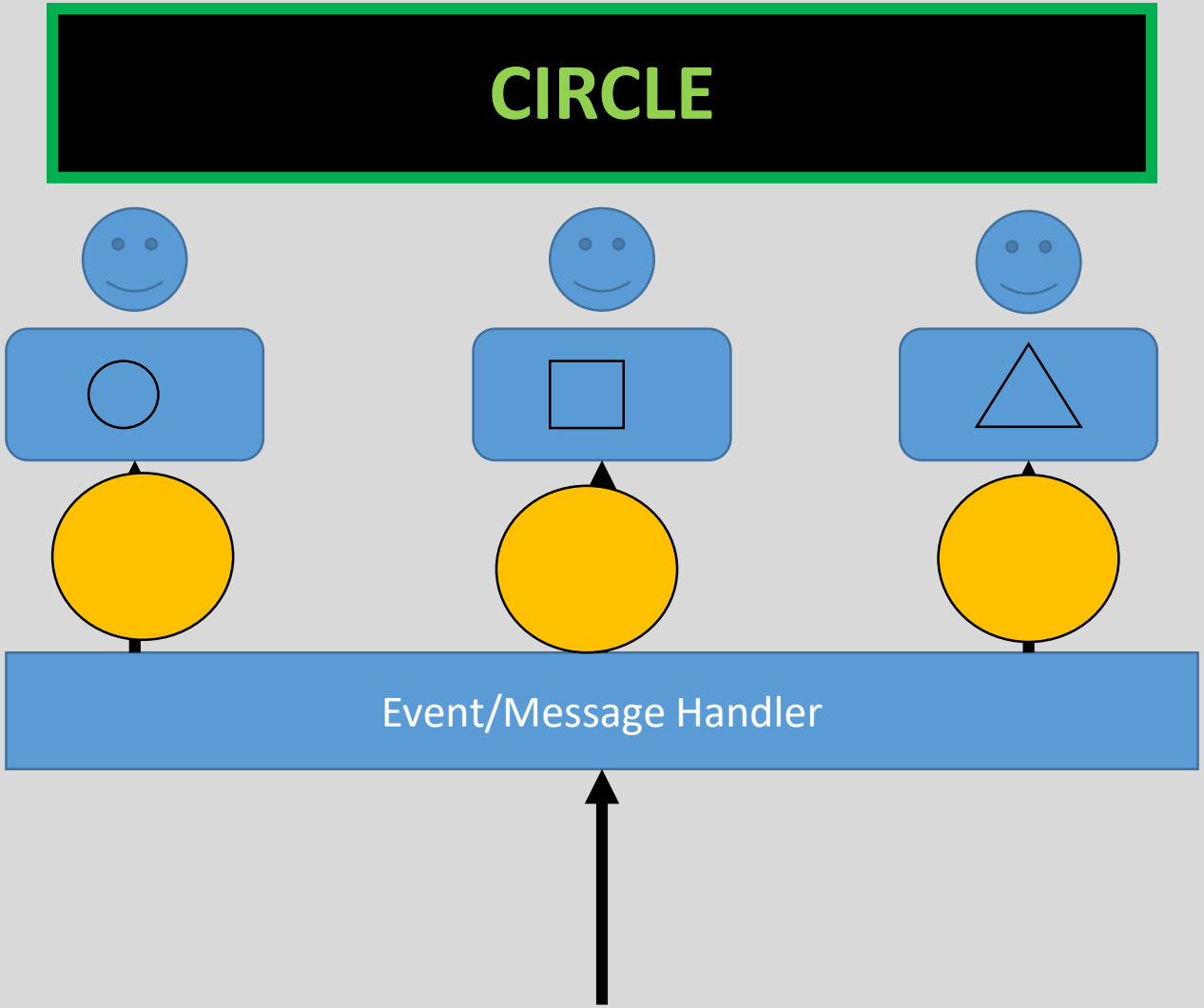
Call-Return Model



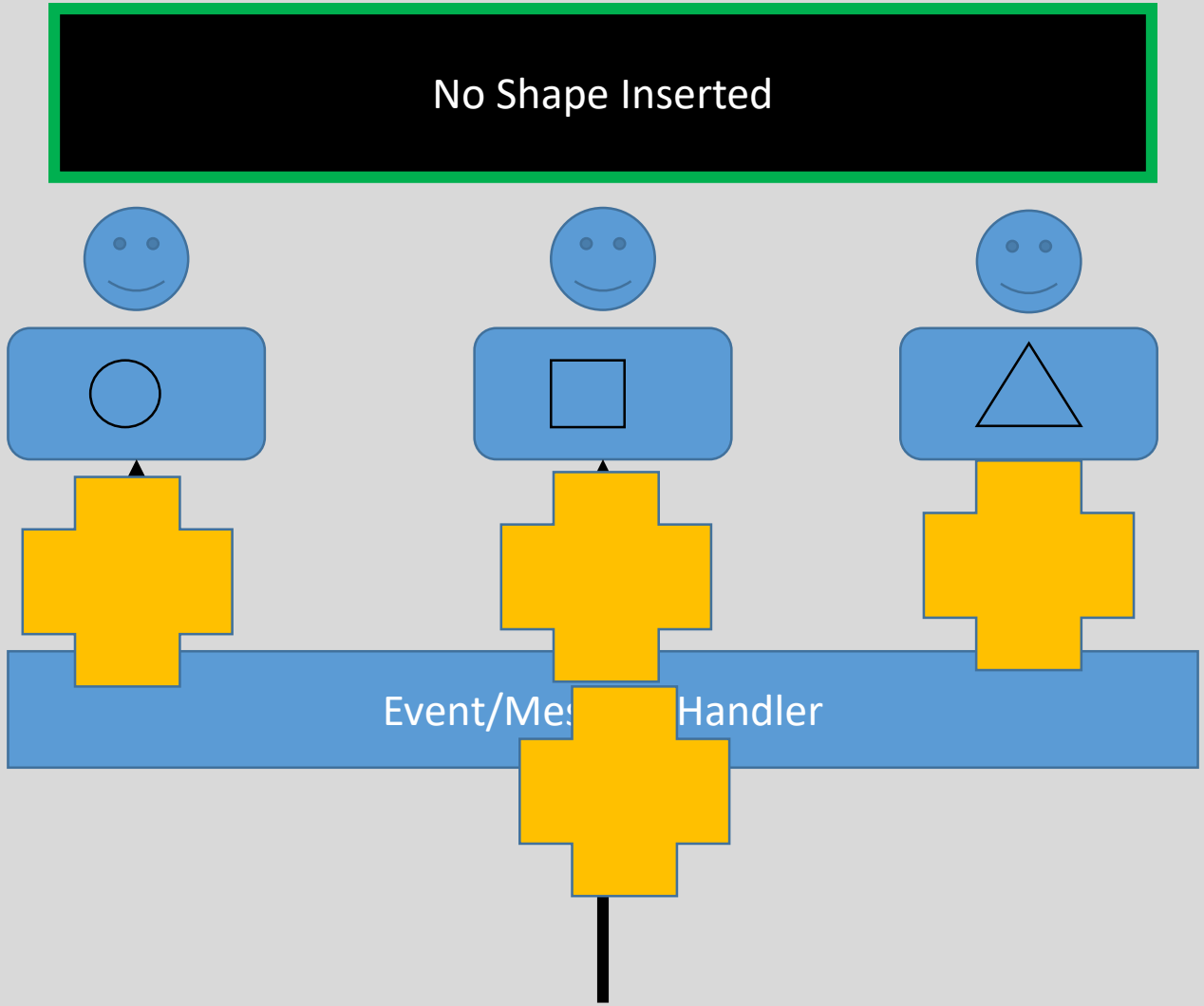
Broadcast Model

- Sub-systems register an interest in specific events.
- When these occur, control is transferred to the sub-system which can handle the event
- **Control policy is not embedded in the event** and message handler.
- Sub-systems decide on events of interest to them
- **However**, sub-systems don't know if or when an event will be handled

Example: Shape Game



Example: Shape Game



Testing

Test Data and Test Cases

- *Test data* Inputs which have been devised to test the system
- *Test cases* Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification

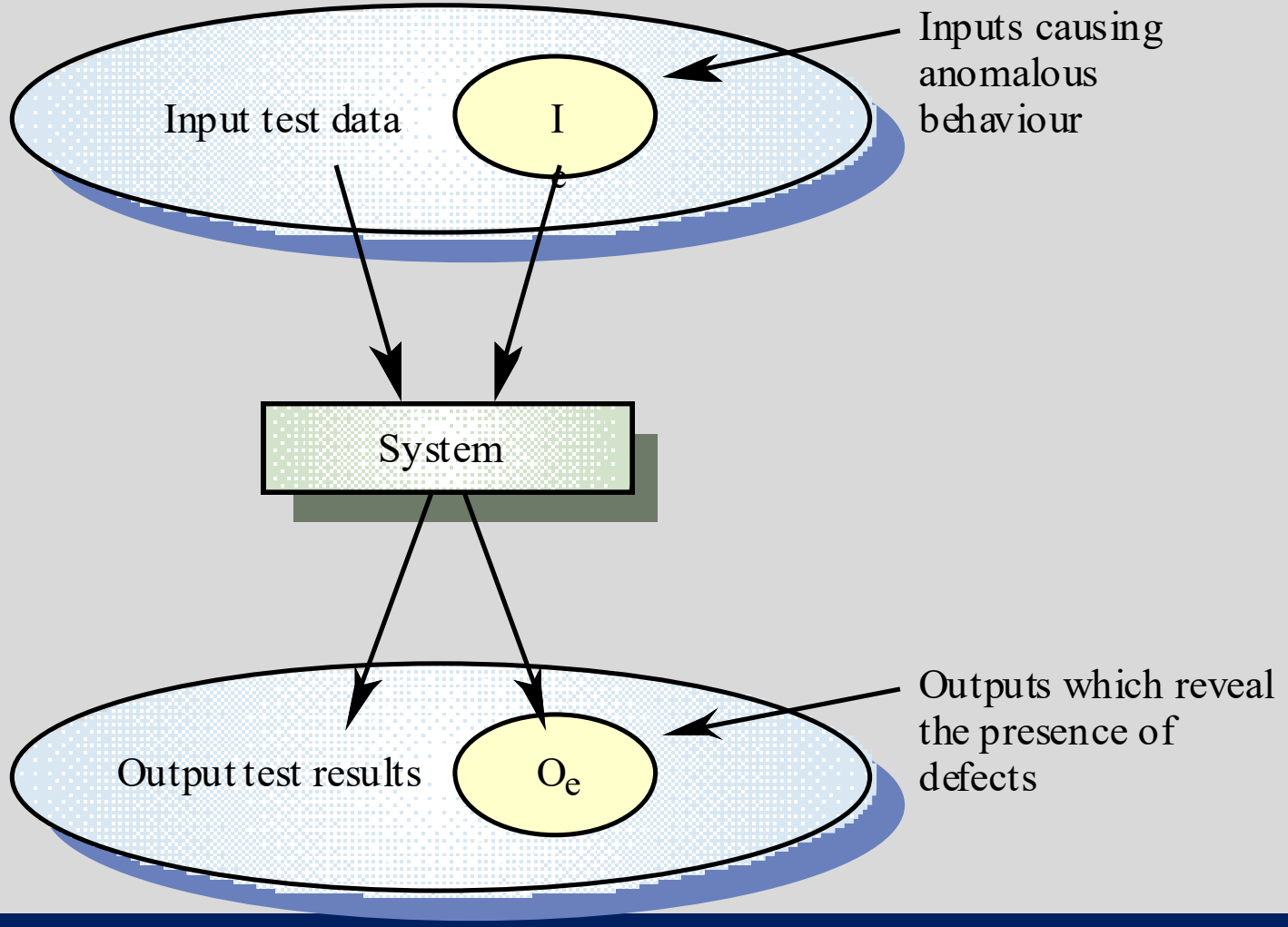
Test plan template

Name of case	Description	Input data	Action	Expected output	Actual output	Success/ Fail
LoginOKPass	Tests login with a good username and password	Username=test 1 Password=pass 1	Click login	OK	Login OK	Success
LoginBadPass	Tests login with good username but wrong password	Username=test 1 Password=pass 2	Click login	Failed to Login	Failed to login	Success
LoginNoPass	Tests login with password field empty	Username=test 1 Password=	Click login	Failed to login	Login OK	Fail

Black-box Testing

- An approach to testing where the program is considered as a '**black-box**'
- The program test cases are based on the system specification
- Test planning can begin early in the software process

Black-box Testing



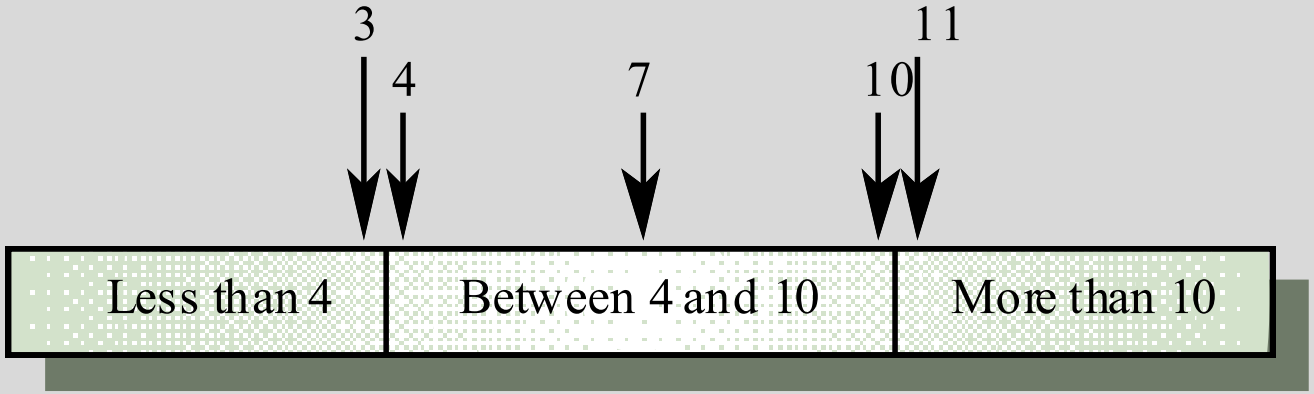
Equivalence Partitioning

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition

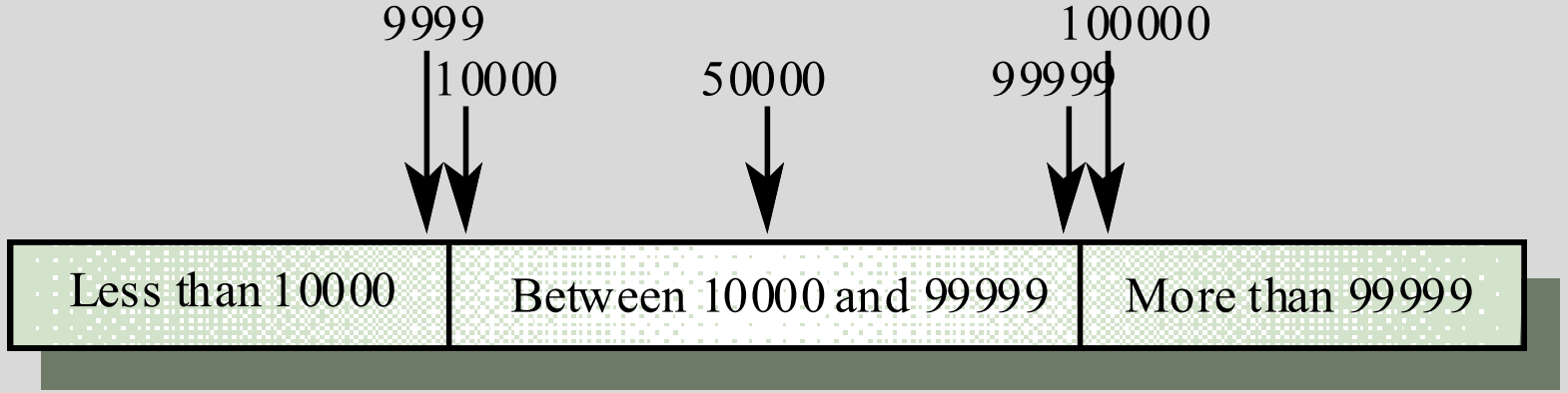
Equivalence Partitioning

- Partition system inputs and outputs into ‘equivalence sets’
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are <10,000, 10,000-99, 999 and > 10, 000
- Choose test cases at the boundary of these sets
  - 00000, 09999, 10000, 99999, 10001
- These are more likely to display erroneous behaviour than choosing random values

Equivalence Partitions



Number of input values



Input values

And Finally…

Finally…

- I have **not** just told you what will be in the exam!
- I have reviewed a little of the more *difficult* material
- You should revise **everything from the lectures**

Drop-In Lab
TODAY 1200-1300
Lab 3

Exam style questions-
Thursday (TOMORROW) 1700-1800
CHAD-ROTBLAT