COMP 3111
SOFTWARE ENGINEERING

SYSTEM ANALYSIS AND DESIGN

LEARNING OBJECTIVES

1. Understand the purpose and importance of system analysis and design in software development.

2. Know the major activities that take place during system analysis and design.

3. Know how to realize design goals and deal with the implementation environment.

4. Know what are architectural patterns and design patterns and when to use them.

SYSTEM ANALYSIS AND DESIGN OUTLINE

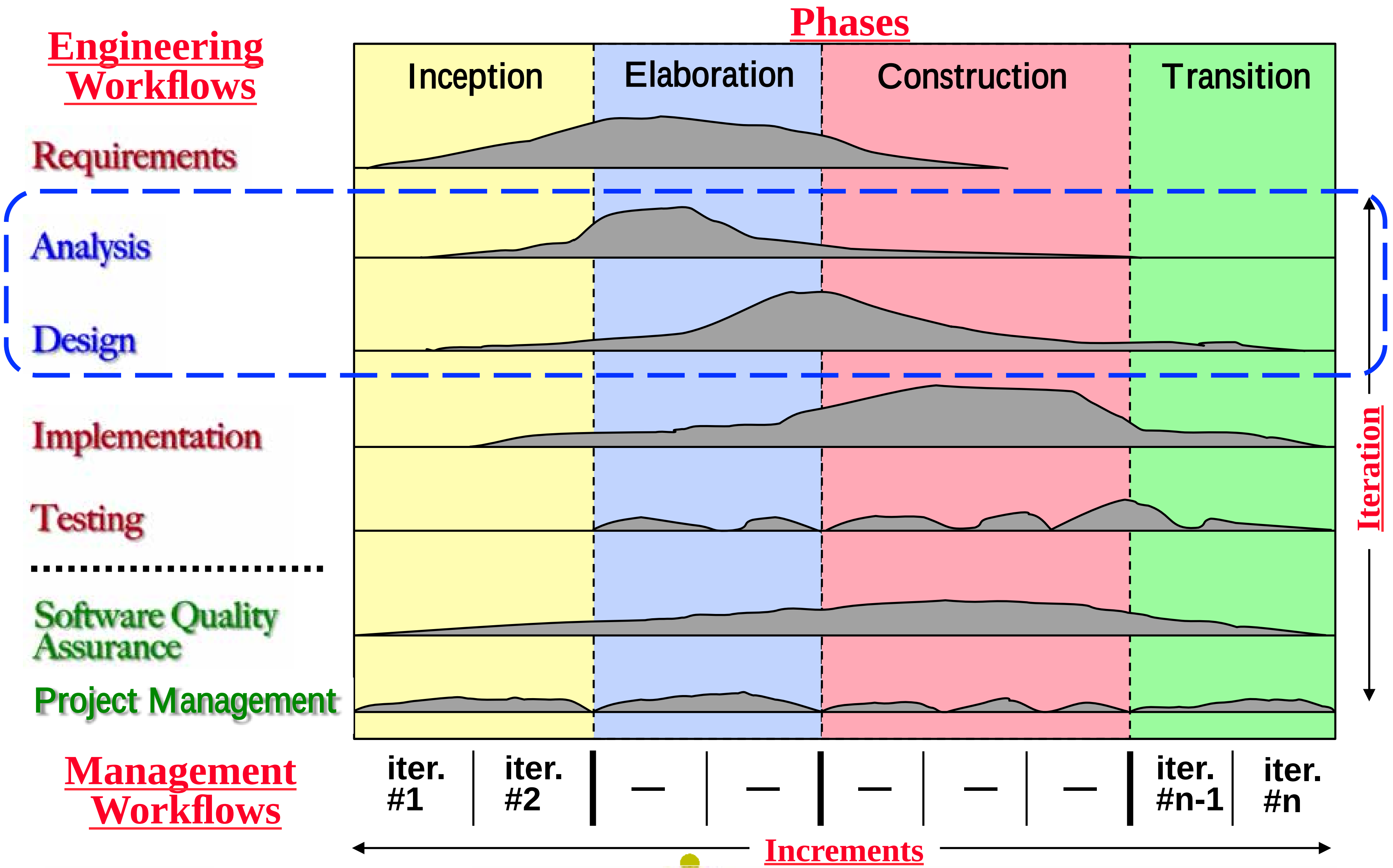
System Analysis and Design Overview

- Life Cycle Role
- The Purpose and Importance of System Analysis and Design
- Realizing Design Goals
- Dealing with the Implementation Environment

System Analysis and Design Activities

- Architectural Analysis and Design
- Analysis Class
- Design Class
- State Machine Diagrams
- Design Patterns
- Anti Patterns

SYSTEM ANALYSIS AND DESIGN LIFE CYCLE ROLE



SYSTEM ANALYSIS AND DESIGN LIFE CYCLE ROLE (CONT'D)

Objective: Achieve a sound and stable architecture.
Create a blueprint for the implementation model.

- Design involves merging in technical solutions from the solution domain.
- Since the design model is very close to the implementation, it is natural to keep and maintain it throughout the software life cycle.
- In round-trip engineering, the design model can be used to visualize the implementation and to support graphical programming techniques.

THE PURPOSE OF SYSTEM ANALYSIS AND DESIGN

***System analysis and design* structures the use-case model into a form that is both robust and maintainable, adapts it to the implementation environment and prepares it for implementation.**

Objective: to achieve a sound and stable architecture.

The system is *gradually structured* into basic “parts” with well-defined relationships among them in two stages:

1. **building the skeleton** is the process by which the parts come together to produce a stable basis.

   ☞ **The *skeleton* is provided by the software architecture.**

2. **growing the system** is the process by which the parts aggregate around the skeleton to produce a finer system structure.

   ☞ **The *growing* is accomplished by the analysis and design workflows.**

THE IMPORTANCE OF SYSTEM ANALYSIS AND DESIGN

Why not go straight to implementation?

- Since the use-case model is not sufficiently formal, we need to:
  - complete specification to remove any ambiguities and redundancies.
  - describe requirements more precisely using developer's language.
  - structure requirements for easier understanding and maintenance.

- We also need to consider issues previously ignored, such as how to deal with:
  - nonfunctional requirements.
  - implementation environment.

It gives us a more precise and detailed understanding of the requirements *before* we start to implement.

USE-CASE MODEL	ANALYSIS & DESIGN MODEL
Uses client's language	Uses developer's language
External view of the system	Internal view of the system
Use cases structure the external view	Classes and packages structure the internal view
Used primarily as a contract between client/developer to agree on system functionality	Used primarily by developers to understand how to implement the system
May contain ambiguities, redundancies, inconsistencies, etc.	Should not contain ambiguities, redundancies, inconsistencies, etc.
Captures system functionality	Outlines system implementation

REALIZING DESIGN GOALS

We need to decide how design goals will be realized.

Design goals come mainly from nonfunctional requirements (i.e., system qualities that we try to “optimize”).

☞ **Usually only a small subset of the design goals can be effectively realized simultaneously.**

THEREFORE we need to prioritize design goals and possibly develop trade-offs against each other as well as against managerial goals (e.g., schedule, budget).

Examples: space vs. speed; delivery time vs. reliability, etc.

REALIZING DESIGN GOALS: IDENTIFYING DESIRABLE SYSTEM QUALITIES

Design goals guide the decisions made by developers, especially when implementation trade-offs are needed.

Dependability
- robustness
- reliability
- availability
- fault tolerance
- security
- safety

Maintenance
- extensibility
- modifiability
- adaptability
- portability
- readability
- traceability

Cost
- development
- deployment
- upgrade
- maintenance
- administration

Performance
- response time
- throughput
- memory

End user
- utility
- usability

DEALING WITH THE IMPLEMENTATION ENVIRONMENT

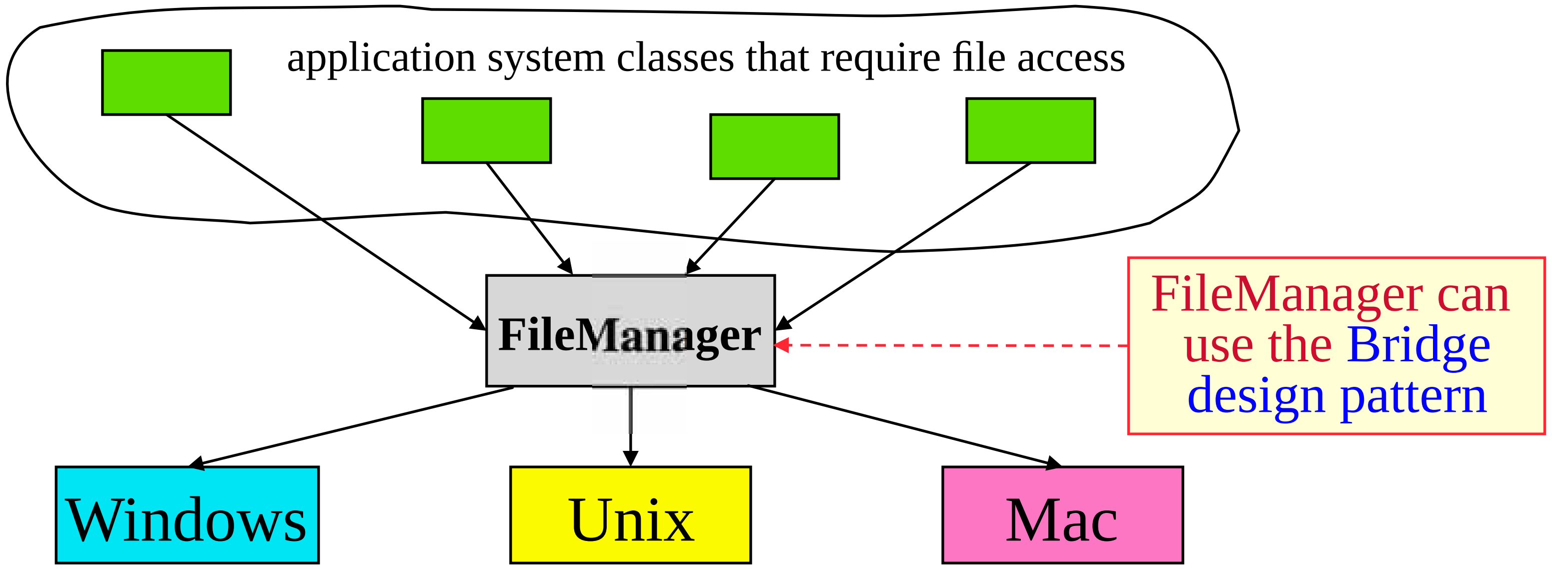
We need to consider the technical and managerial constraints under which the system will be built.

- What hardware/software will the system run on?
  - hardware (limitations)
  - operating system
  - need for distribution
  - network architecture
- What programming language will be used?
  - OO
  - non-OO
- What (existing) software do we need to use?
  - DBMS
  - UI development toolkits
  - network facilities
  - legacy (existing) systems
- What development people/organizations will be involved?
  - team competencies
  - distributed development

DEALING WITH THE IMPLEMENTATION ENVIRONMENT (cont'd)

General design strategy for technical issues:
Encapsulate and isolate the implementation environment.

Create "bridge" classes that represent occurrences of components in the implementation environment.


application system classes that require file access
FileManager
FileManager can use the Bridge design pattern
Windows
Unix
Mac

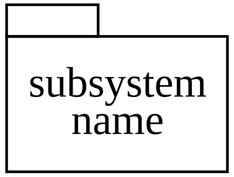
☞ **We often need to define many additional classes to deal with the implementation environment so as to facilitate maintenance.**

ARCHITECTURAL ANALYSIS AND DESIGN

- Architectural analysis and design focuses on
  - understanding how a system should be organized.
  - designing the overall structure of the system.
- It identifies the main structural components and the relationships between them.
- Nonfunctional requirements are highly dependent on the system architecture.

Architectural analysis and design is an essential tool for managing complexity.

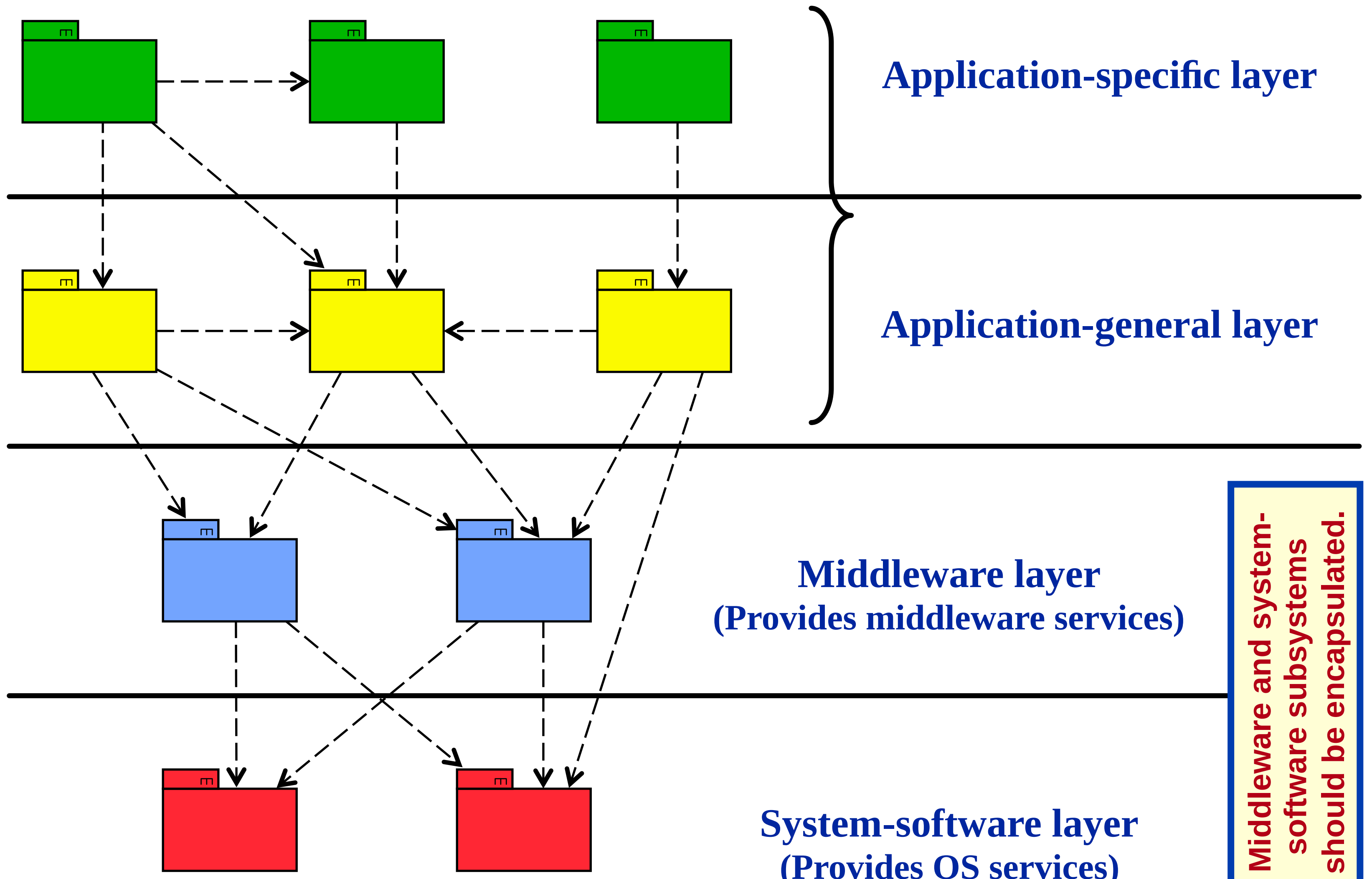
ARCHITECTURAL ANALYSIS AND DESIGN: SUBSYSTEM



A *subsystem* organizes artifacts of the design model into more manageable pieces.

- Subsystems encapsulate their contents and provide *services* (operations) to other subsystems via an interface.
- A subsystem provides information hiding.
- Subsystems should be highly cohesive and loosely coupled.

ARCHITECTURAL ANALYSIS AND DESIGN: LAYERS AND PARTITIONS


Application-specific layer
Application-general layer
Middleware layer
(Provides middleware services)
System-software layer
(Provides OS services)

Middleware and system-software subsystems should be encapsulated.

ARCHITECTURAL ANALYSIS AND DESIGN: LAYERS AND PARTITIONS

- Architecture types:

  1. closed layered: Each layer can only depend on the layer immediately below it → lower coupling (but more overhead).
  2. open layered: Each layer can depend on any layer below it → higher coupling (but less overhead).

- Subsystem partitions:
  - Divide services in one layer into different subsystems.
  - Results in peer to peer services within a layer.

Usually there are 3 to 5 subsystem layers in practice.

ARCHITECTURAL ANALYSIS AND DESIGN: ARCHITECTURE DECISIONS

- Based primarily on the nonfunctional requirements.

Performance

- Use a small number of subsystems to localize critical functions.

Security

- Use a layered architecture with most critical assets in the innermost layer.

Safety

-

Availability

- Include redundant subsystems to allow replacement or update without stopping the system.

Maintainability

-

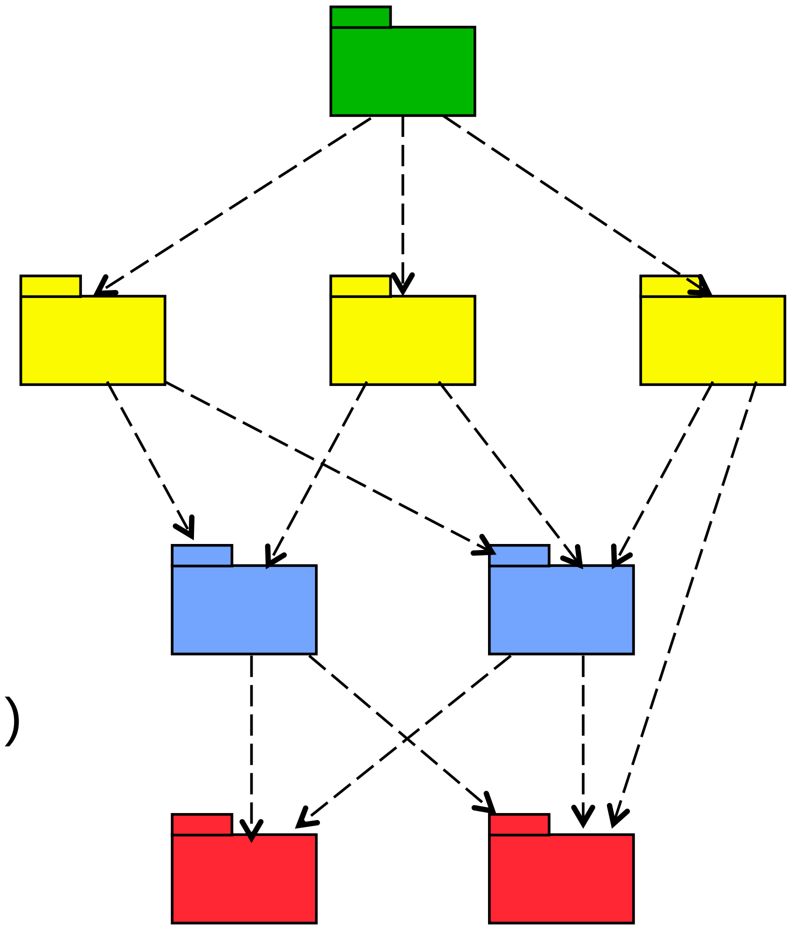
ARCHITECTURAL PATTERNS

An *architectural pattern (style)* specifies how to organize and connect a set of software components.

- It is a blueprint for system organization that defines the software architecture of a system.
- It includes the specification of:
  - system decomposition
  - global control flow
  - error handling policies
  - inter module/component communication protocols
- Most large software systems use several architectural patterns in different parts of the system.

ARCHITECTURAL PATTERN: MULTI-LAYER

Goal: **To build the software in layers so that each layer communicates only with the layers below it.**



Top layer: user interface

Middle layers: application functions

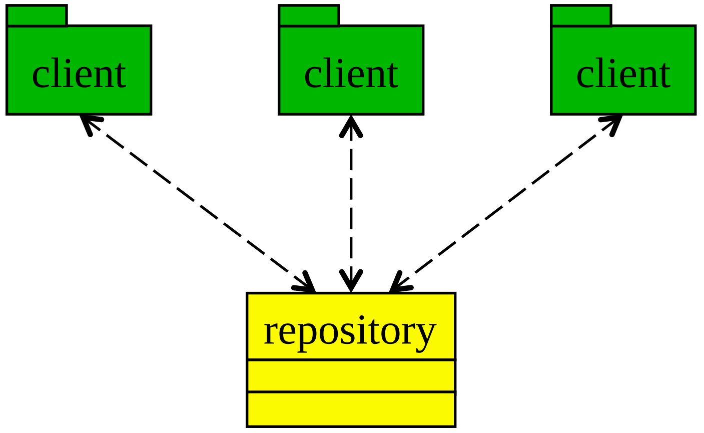
Bottom layers: common services (e.g., data storage, communication, etc.)

☞ Each layer has a well-defined API defining the services it provides.

Often combined with other patterns.

ARCHITECTURAL PATTERN: REPOSITORY

Goal: To centralize the management of data.



- Control flow can be dictated by:
  - the repository via triggers on the data.
  - subsystems using locks to synchronize.
- Possible drawbacks:
  - the repository can be a performance and modifiability bottleneck.
  - there is high coupling between the repository and subsystems.

A typical pattern for applications requiring data management.

ARCHITECTURAL PATTERN: CLIENT-SERVER

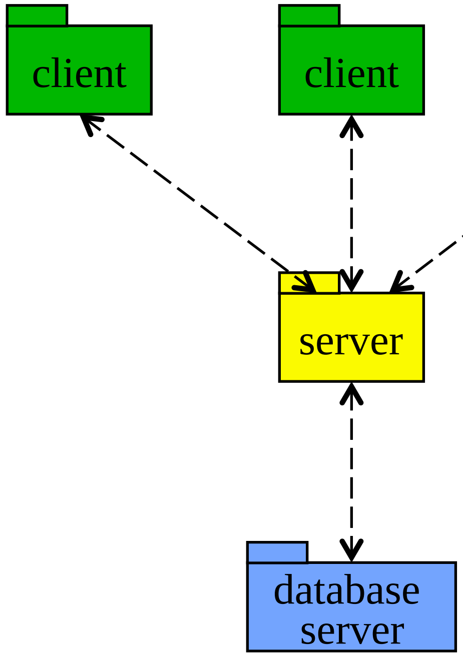
Goal: **To separate and distribute the system's functionality.**

three-tier

The server communicates with both a client (usually via the Internet) and with a database server (usually via an intranet).

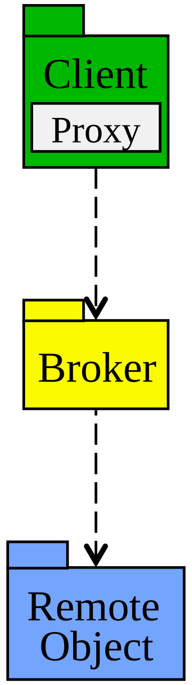
peer-to-peer

Each subsystem can be both a server and a client and can set up a communication channel to exchange information as required.



ARCHITECTURAL PATTERN: BROKER

Goal: **To distribute aspects of the system transparently to different nodes.**



Examples:

CORBA - Common Object Request Broker Architecture

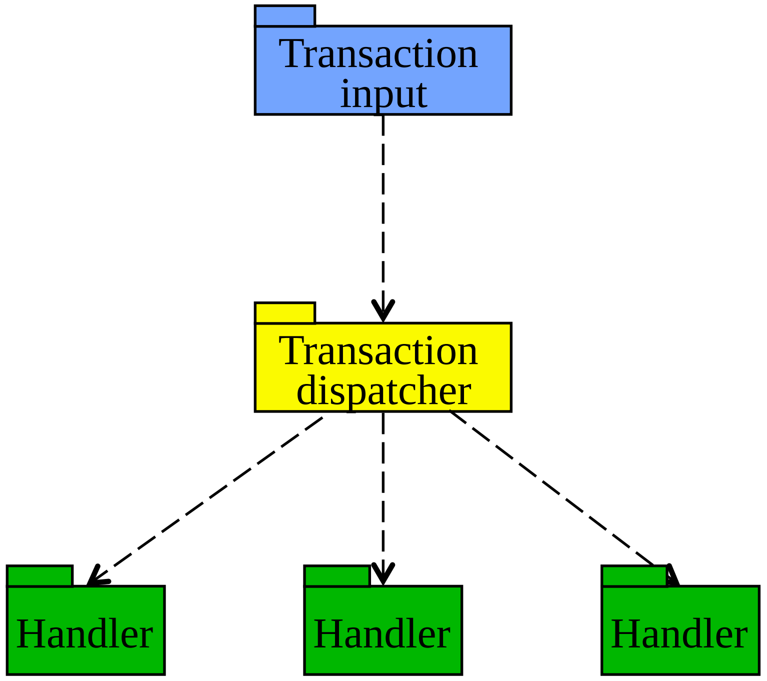
Microsoft COM - Common Object Model

The Proxy design pattern can be used to implement the broker architecture.

ARCHITECTURAL PATTERN: TRANSACTION PROCESSING

Goal: To direct input to specialized subsystems (handlers).

- A transaction dispatcher decides what to do with each input transaction, dispatching a procedure call or message to a handler that will handle the transaction.

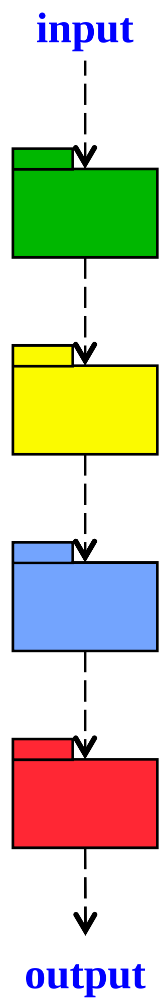


Example: A database engine.

ARCHITECTURAL PATTERN: PIPE-AND-FILTER

Goal: To provide flexibility, modifiability and reusability of subsystems.

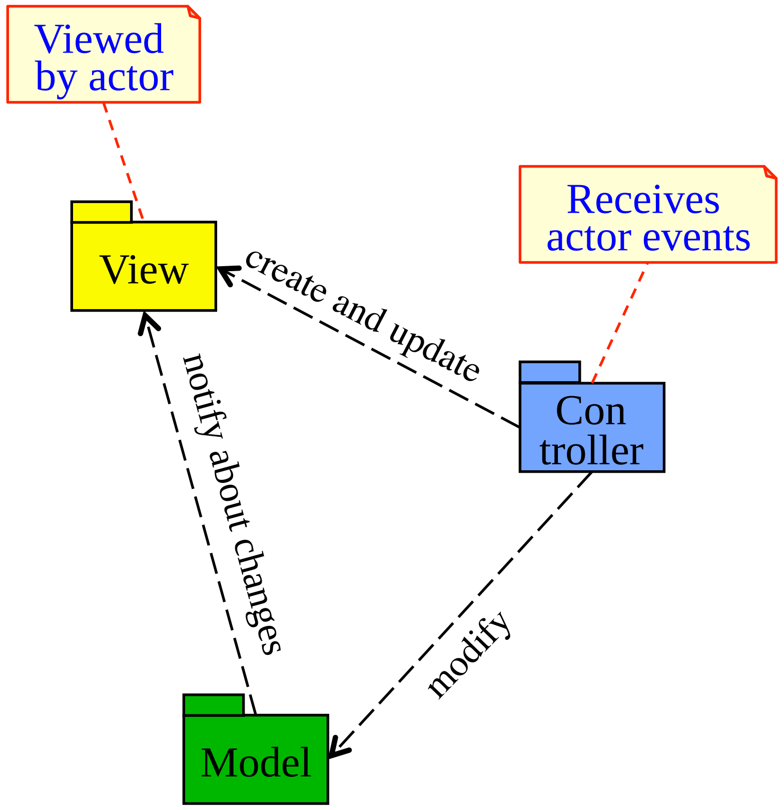
- A stream of data is passed through a series of filters (subsystems), called a pipeline, each of which transforms it in some way.
- Each filter is executed concurrently.



ARCHITECTURAL PATTERN: MODEL-VIEW-CONTROLLER (MVC)

Goal: To separate the user interface layer from other parts of the system.

- The model contains the classes (data) to be viewed and manipulated.
- The view contains objects used to render the data from the model and also the various controls with which the user can interact.
- The controller contains the objects that control and handle the user's interaction with the view and the model.



The Observer design pattern is normally used to separate the model from the view.

ANALYSIS CLASS

An ***analysis class*** is an **abstraction** of one or several classes in the final system implementation.

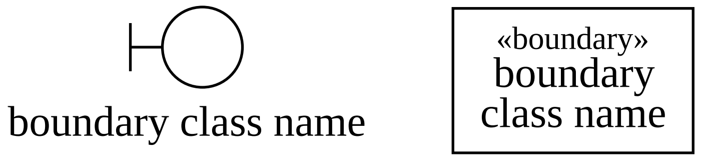
System design usually has many—up to 5 times—more classes!

☞ **We focus on handling *only* functional requirements.**

- Class descriptions are conceptual, not implementation oriented:
  - attribute types are conceptual, *not* programming language types.
  - behavior is defined by textual descriptions of *responsibilities.*
  - relationships are conceptual, *not* implementation oriented.

- Analysis classes are one of the three following stereotypes:
  - boundary
  - entity
  - control

ANALYSIS CLASS: BOUNDARY CLASS



A *boundary class* models the interaction between the system and its actors.

- Represents an *abstraction* of UI elements (windows, forms, panes, etc.) or devices (printer interfaces, sensors, terminals, etc.).
- Interacts with *actors* outside the system as well as with *objects* inside the system.
- The description should be at a fairly high conceptual level.

DESCRIBE WHAT *NOT* HOW!

(Do not describe every button, menu item, etc. of a UI!).

Encapsulates and isolates changes in the system's interface.

IDENTIFYING BOUNDARY CLASSES

Usually found from use case descriptions → start from the actors:

- Identify forms and windows needed to enter data into the system.
- Identify notices and messages the systems uses to respond.
- Do not model the visual aspects of the interface.
- Always use the user's terms for describing interfaces.

Specification strategy

- Initially, identify one boundary class instance (i.e., object) for each actor/use case pair.
  - ☞ For human actors, it represents the *primary UI window* with which the actor interacts with the system.
  - ☞ For external system actors, it represents the *communication interface* to the external system.

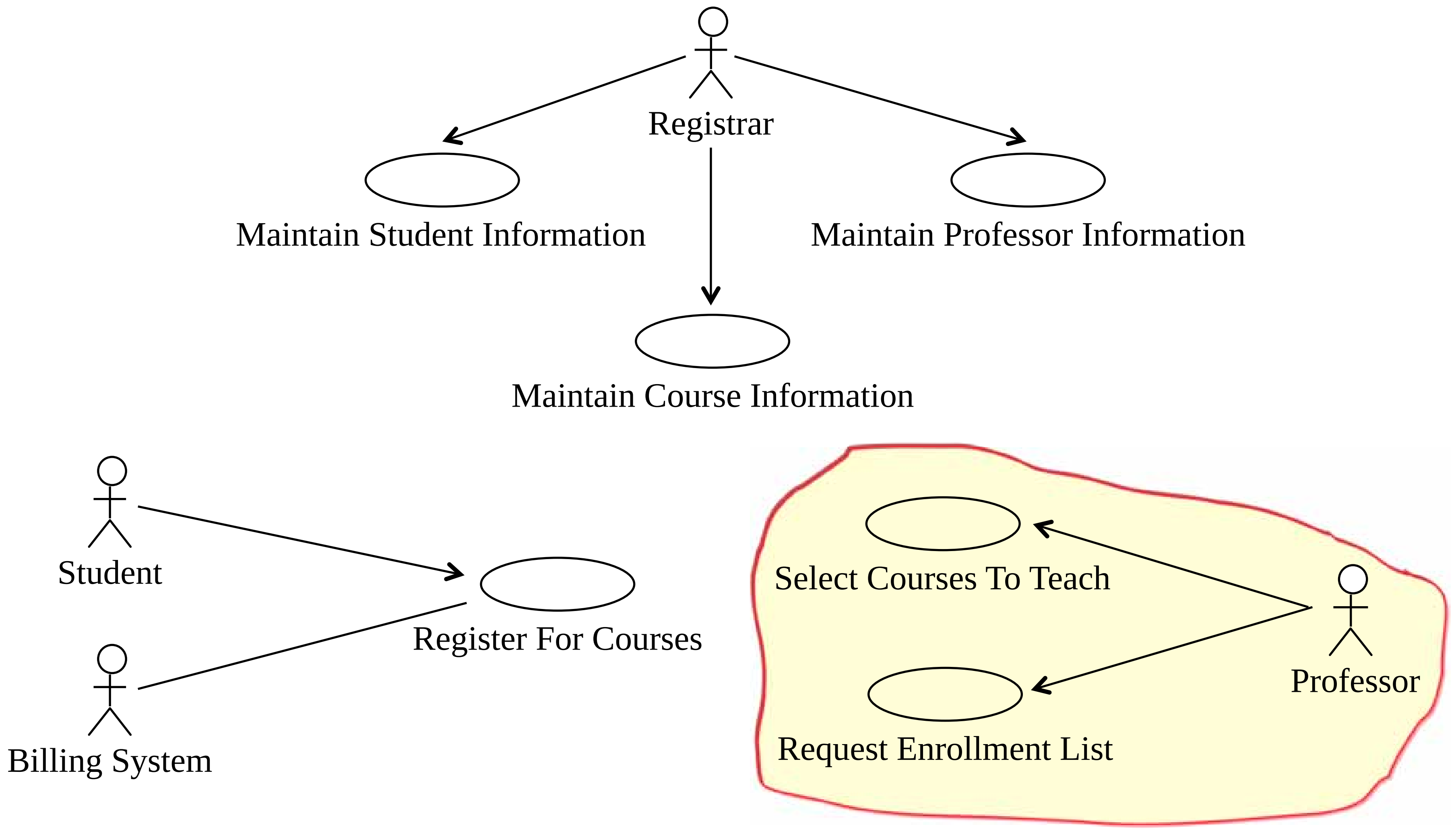
ASU Course Registration System

System Analysis

Boundary Classes

Actor: *Professor*

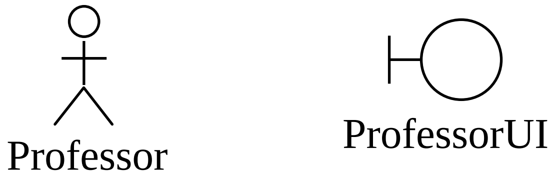
ASU: USE-CASE MODEL



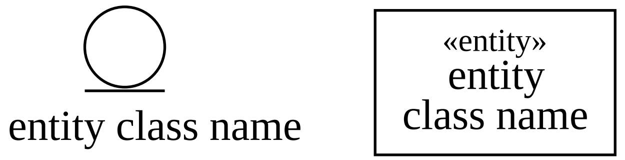
ASU BOUNDARY CLASSES: PROFESSOR ACTOR

Initial specification

Identify one boundary class (object) for each actor/use case pair.
Try to reuse the same boundary object for an actor, if possible.



ANALYSIS CLASS: ENTITY CLASS



An *entity class* models information that is long-lived and often persistent.

- Represents concepts such as an individual, a real-life object or a real-life event.

☞ **An entity object will likely get its data from a database**
(but, it actually represents a dynamic object in the analysis model).

Encapsulates and isolates changes to the information it represents.

Specification strategy

- Examine the scenarios of a use-case to determine what domain model classes are needed to carry out the use case.

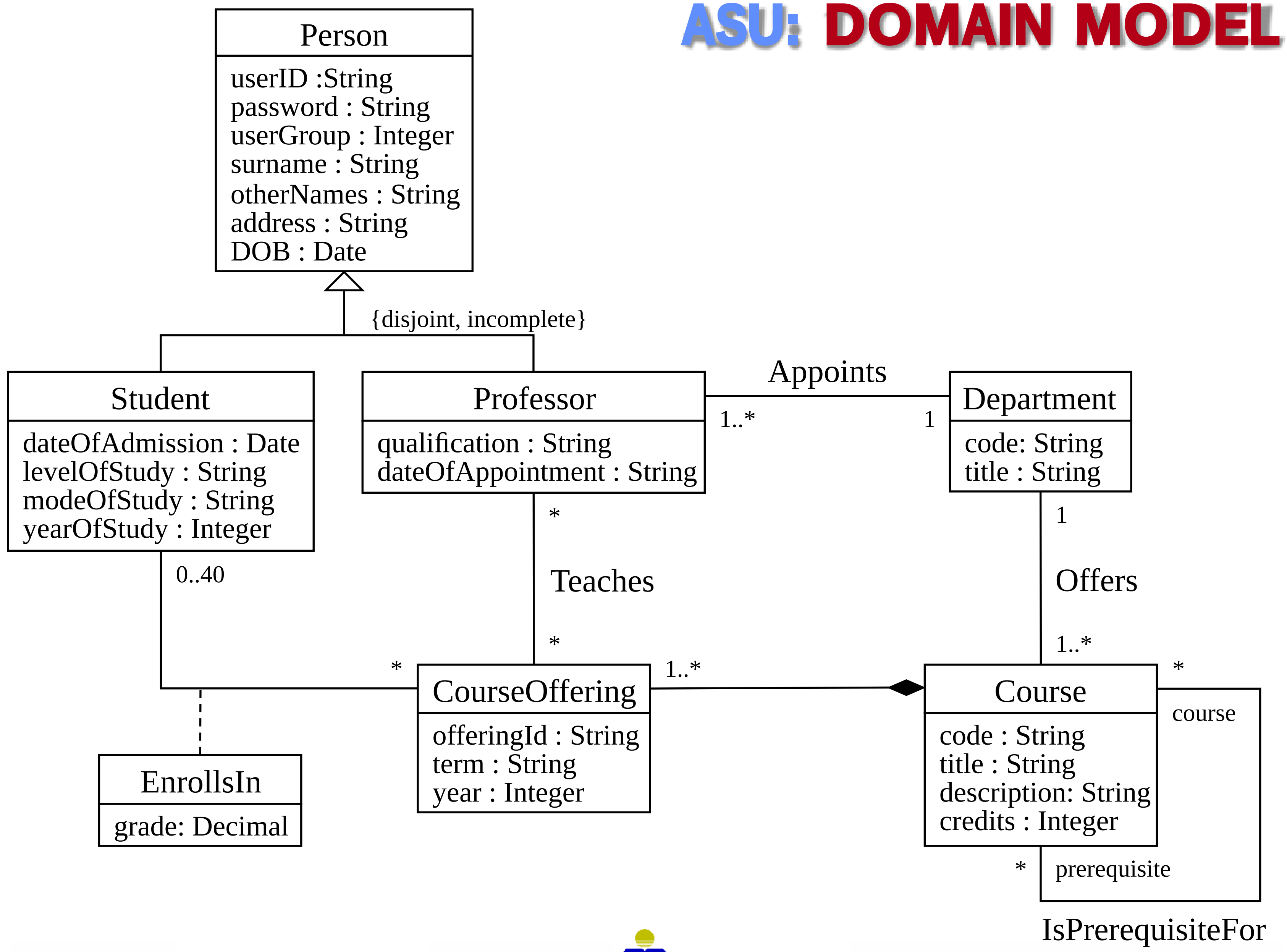
ASU Course Registration System

System Analysis

Entity Classes

Use Case: *Select Courses To Teach*

ASU: DOMAIN MODEL



ASU USE CASE: SELECT COURSES TO TEACH

Flow of events — *Create Schedule* scenario

1. The use case begins when the Professor actor chooses to select the courses he wants to teach.
2. The system displays the interface for selecting the courses to teach.
3. The Professor indicates the term and year in which he would like to teach courses.
4. The system retrieves and displays the available course information for the given term if the final date for changes has not passed.
5. The Professor selects the courses that he would like to teach.
6. The Professor confirms the selection.
7. The system creates the Professor's teaching schedule in the database.
8. The system notifies the Professor that the teaching schedule has been created.
9. The use case ends.

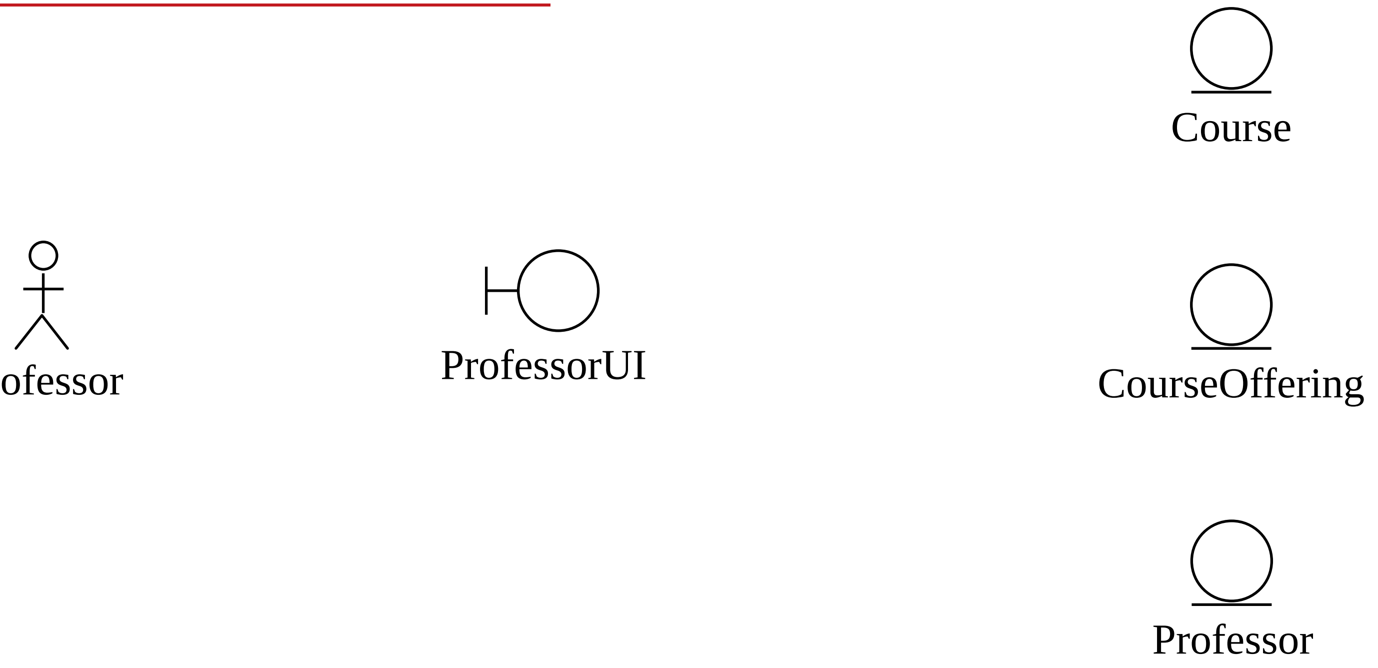
In which class(es) are these attributes?

Which class(es) contain this information?

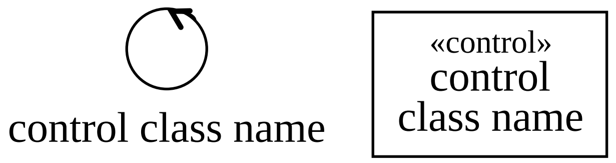
Which class(es) are needed to represent this?

ASU ENTITY CLASSES:
SELECT COURSES TO TEACH

Create Schedule scenario



ANALYSIS CLASS: CONTROL CLASS



A *control class* models coordination, sequencing, transactions and control behaviour (for one or more use cases).

- A control class instance (object) usually does not have a correspondence in the application domain.

  ☞ A control object provides the “glue” that ties other classes together into one use-case realization.

- A control class instance (i.e., object) is used to represent:
  - control related to a specific use case.
  - business logic (e.g., complex derivations and calculations).

Encapsulates and isolates changes to control/business logic.

ANALYSIS CLASS: CONTROL CLASS (CONT'D)

Specification strategy

- Initially, assign one control object for each actor/use case pair; this object is responsible for the flow of events in the use case.
  - Complicated behavior may need several control objects.
  - Can also combine/eliminate some control objects.

Handle later!

☞ **A control object should only be tied to at most one actor!**

WHY?

ASU Course Registration System

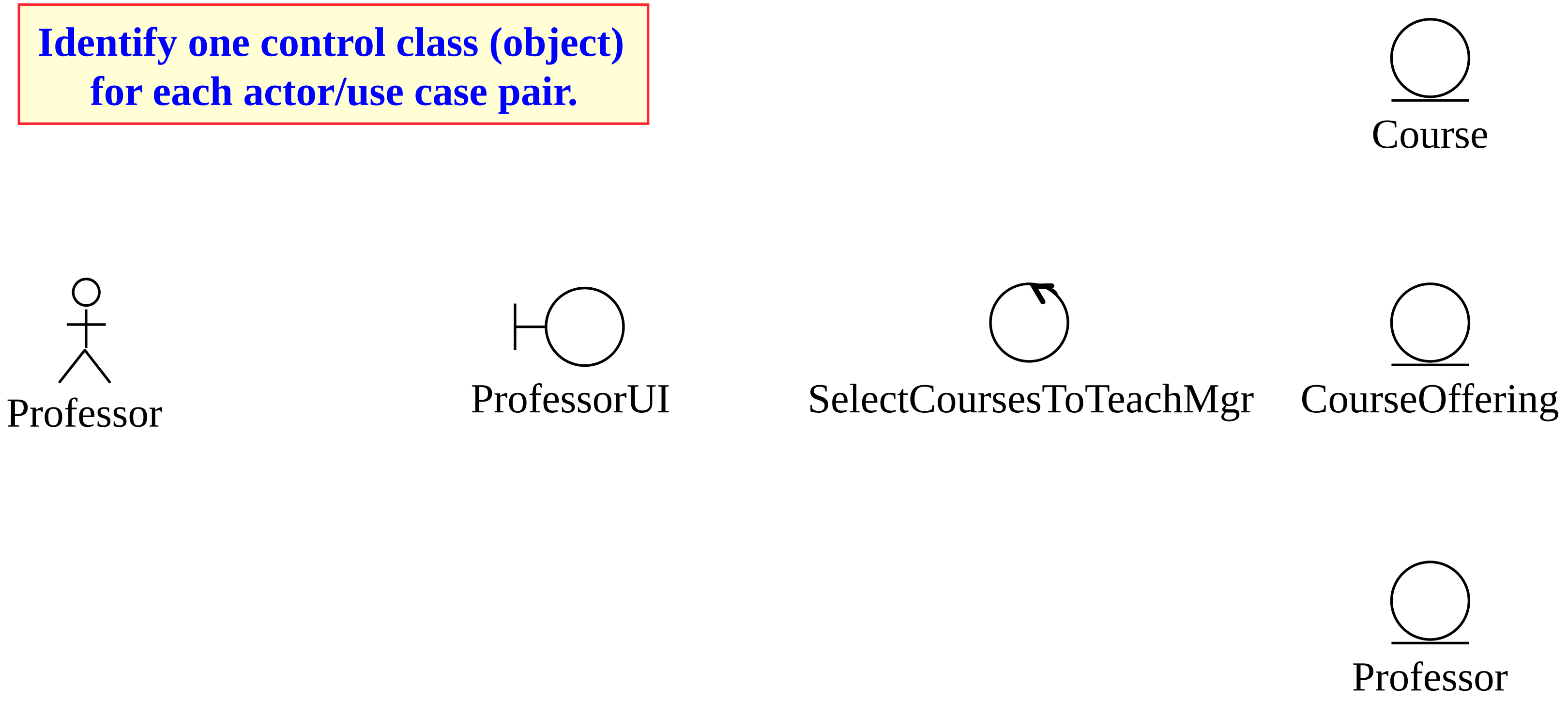
System Analysis

Control Classes

Use Case: ***Select Courses To Teach***

ASU CONTROL CLASSES: SELECT COURSES TO TEACH

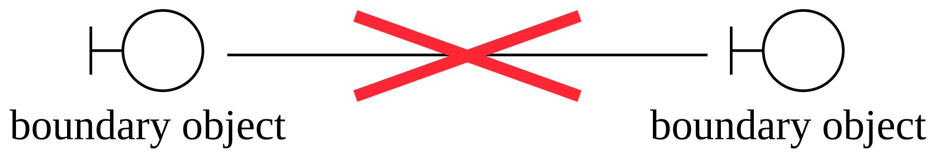
Identify one control class (object) for each actor/use case pair.



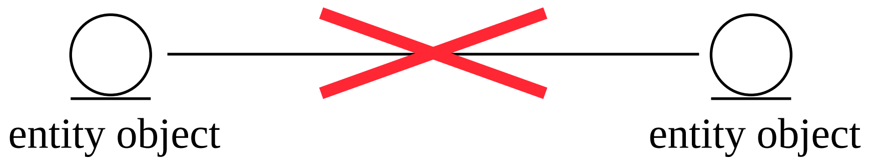
How should we "connect" the boundary, control and entity objects (i.e., which objects should be allowed to interact with each other)?

ANALYSIS OBJECT INTERACTION: BEST PRACTICES

- Actors can interact *only* with boundary objects.
- Boundary objects can interact *only* with control objects and actors (*initially*).

☞ 

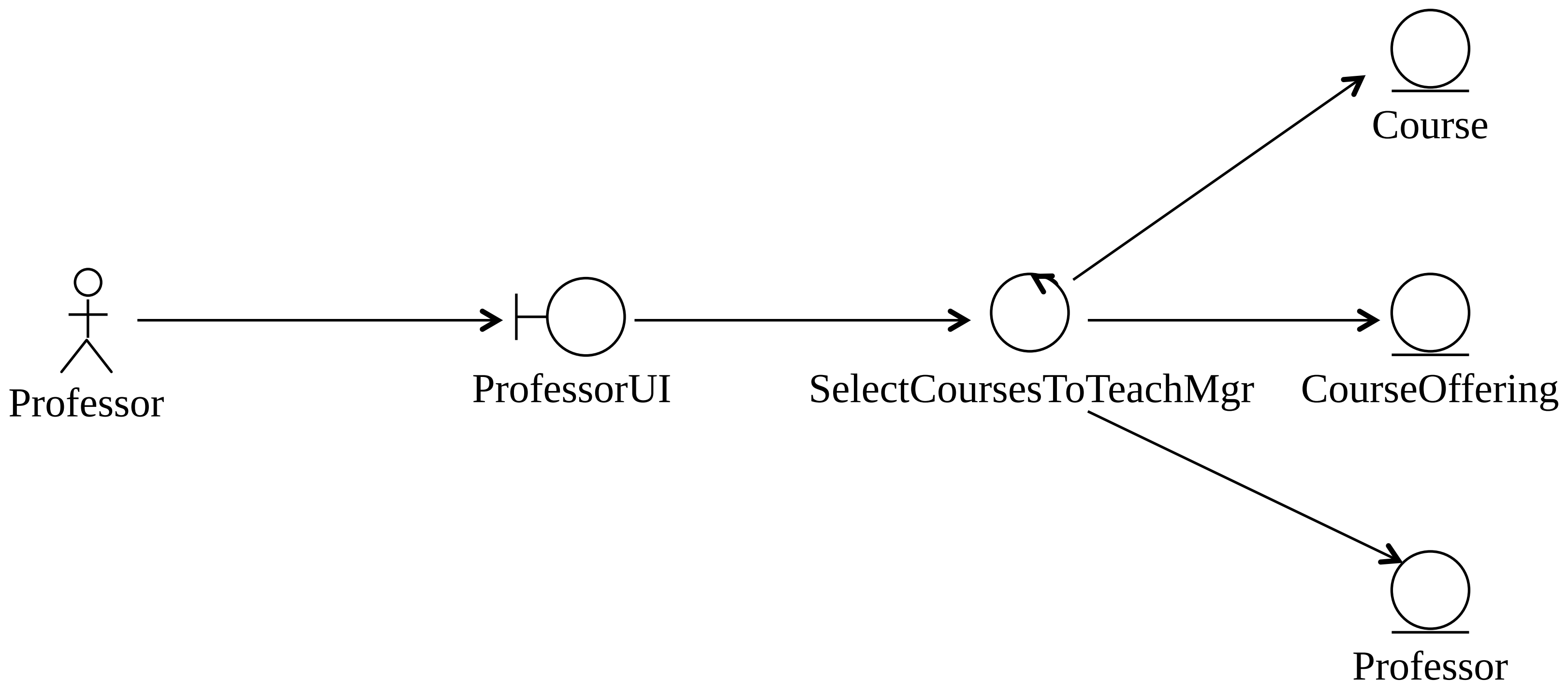
- Entity objects can interact *only* with control objects (*initially*).

☞ 

- Control objects can interact with boundary, entity and *other control objects*.

These best practices result in a well structured and maintainable system.

ASU CONTROL CLASSES: SELECT COURSES TO TEACH



How should we "connect" the boundary, control and entity objects (i.e., which objects should be allowed to interact with each other)?

DESIGN CLASS

A *design class* is a class whose specifications has been completed to such a degree that it can be implemented.

We need to implement problem domain (analysis model) classes using solution domain (implementation technology) classes.

Boundary class → determined by the use of specific UI technologies,

Entity class → determined by the use of data management technologies,

Control class → determined by issues related to:

- distribution → Do we need a separate design class at each node?
- performance → Do we merge with boundary class?
- transactions → Do we need transaction management?

Should preserve traceability to analysis classes!

CLASS DESIGN GOAL: WELL-FORMED CLASSES

High cohesion

☞ A class should model a single abstract concept and should have operations that support the intent of the class.

Low coupling

☞ A class should be associated with *just enough* other classes to allow it to realize its responsibilities.

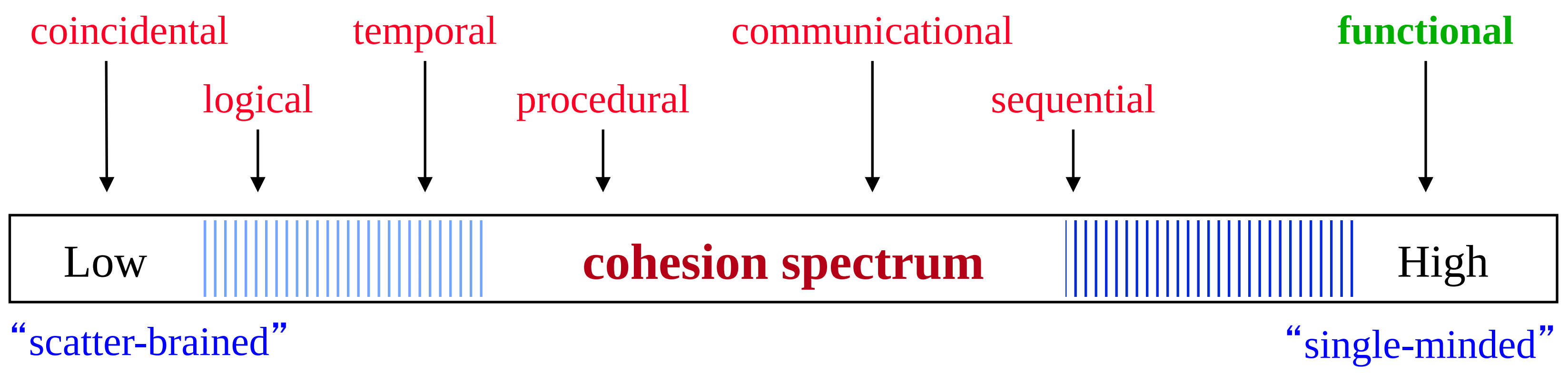
Complete and sufficient

☞ A class should do what users of the class expect – no more and no less.

Primitive

☞ A class should always make available the simplest and smallest possible set of operations.

CLASS DESIGN: COHESION



WORST **coincidental** - The class does not achieve any definable function.

logical - The class does several similar, but slightly different things.

temporal - The class contains various operations that happen to be executed at the same time.

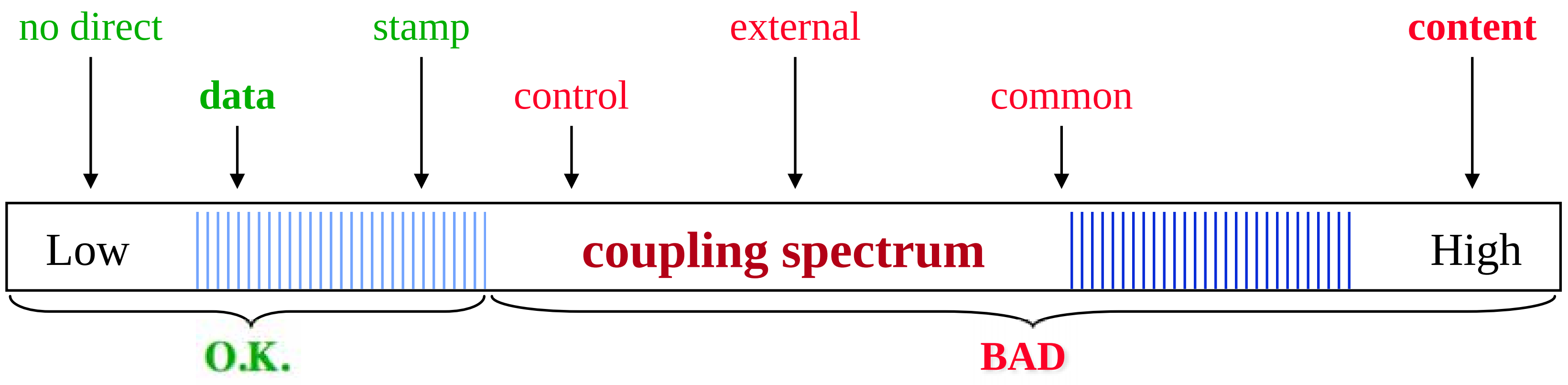
procedural - The class operations must be executed in a pre-specified order.

communicational - All class operations operate on the same data stream or data structure.

sequential - The output from one class is input to the another class.

BEST **functional** - The class has one and only one identifiable function.

CLASS DESIGN: COUPLING



no direct - The classes are not related.

data - The classes take as input only simple data (e.g., a parameter list). ***BEST!***

stamp - The classes pass a portion of a data structure as an argument.

control - The classes pass control information (e.g., via a flag or switch).

external -The classes are tied to an environment external to the system.

common - Several classes reference a global common data area.

content - One class makes use of data or control information maintained within another class. **AVOID!**

DESIGN CLASS: ACTIVITIES

- **Select reusable components**
  - class libraries and design patterns.
- **Complete the specification**
  - **Add:** attributes; associations; operations; types; visibility; constraints.

    ☞ **Identify: active classes**.
- **Restructure the design model**
  - add associations; increase reuse with inheritance.
- **Optimize the design model**
  - revise access paths; collapse classes; cache or delay computations.

Design classes are described using the syntax of the programming language.

DESIGN CLASS: ACTIVE CLASS

- An active class has its own thread of control, is usually a boundary or control class and is *shown with a thicker border* in a class diagram.

- An active class can be identified by considering:

☞ The performance, throughput and availability requirements of different actors as they interact with the system,

e.g., a need for fast (real-time) response might be managed by a dedicated active object for taking input and providing output.

☞ The system's distribution onto nodes — active objects are needed to support distribution onto several nodes,

e.g., one active object per node and separate active objects to handle inter-node communication.

☞ Other requirements:

e.g., system startup and termination, liveness, deadlock avoidance, starvation avoidance, reconfiguration of nodes, connection capacity, etc.
→ An active object is needed to initiate/monitor activity.

STATE MACHINE DIAGRAM

A *state machine diagram* describes the behavior inside an object.

(What an object does when it receives a message.)

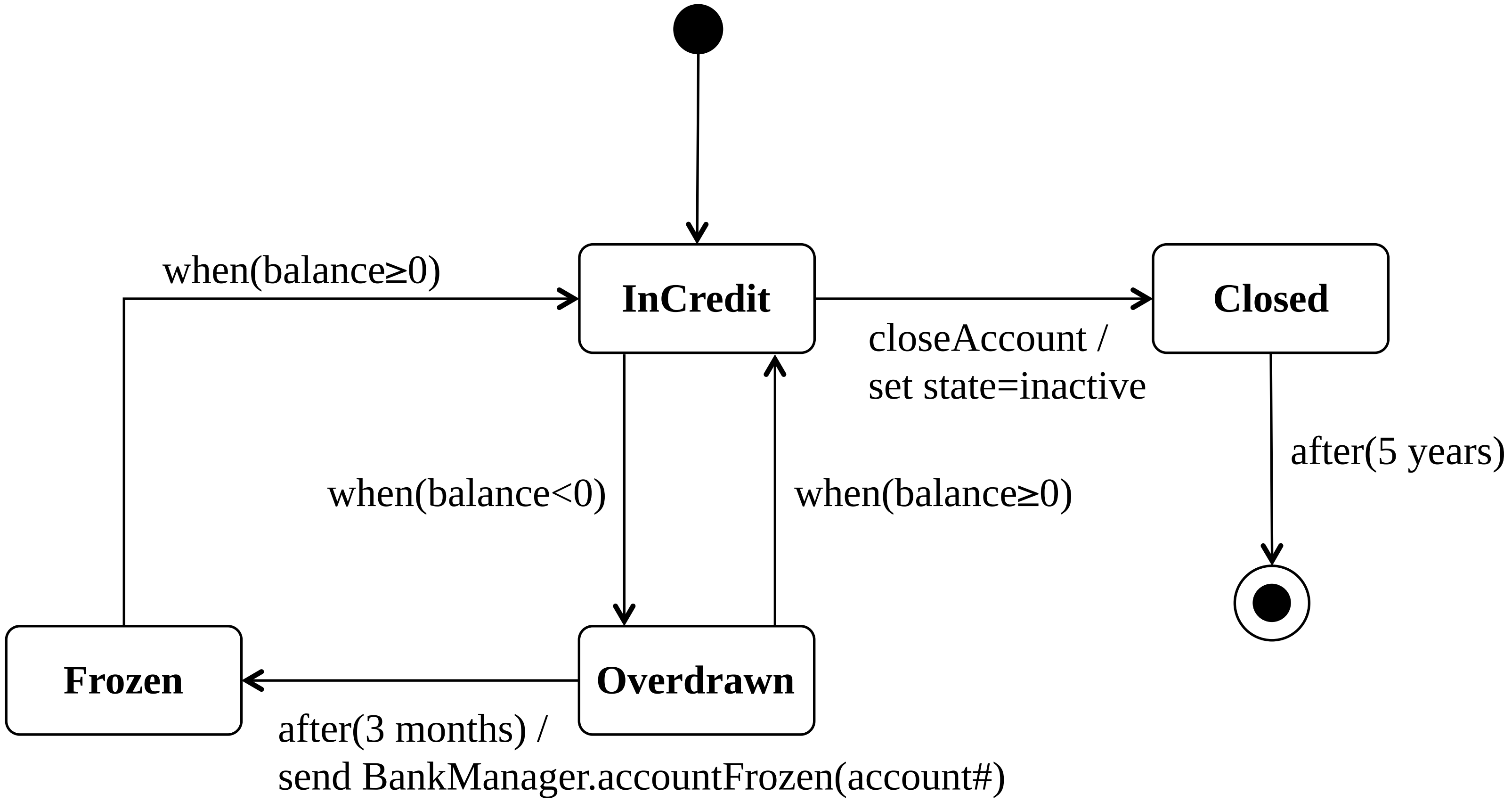
- It is a directed graph that shows:
  - the states of a single object (nodes).
  - the events that cause state changes (arcs).

- It shows all the messages that an object can send and receive.

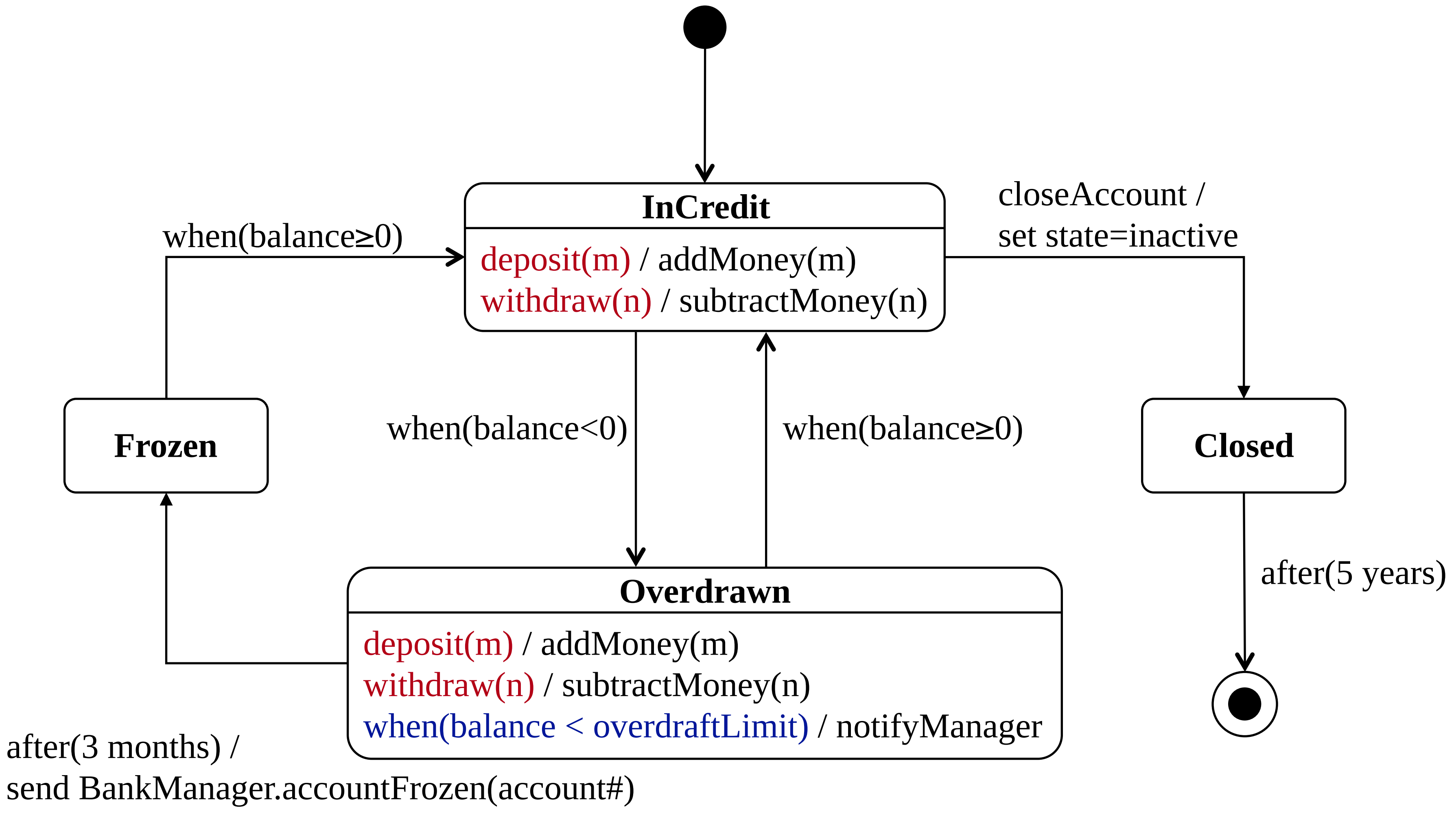
- It describes all the possible states an object can get into during its life time.

- It is drawn for a single class to show the lifetime behavior of a single object.

EXAMPLE STATE MACHINE DIAGRAM: BANKACCOUNT CLASS


when(balance≥0)
InCredit
Closed
closeAccount / set state=inactive
after(5 years)
when(balance<0)
when(balance≥0)
Frozen
Overdrawn
after(3 months) / send BankManager.accountFrozen(account#)

EXAMPLE STATE MACHINE DIAGRAM: BANKACCOUNT CLASS


InCredit
deposit(m) / addMoney(m)
withdraw(n) / subtractMoney(n)
when(balance≥0)
closeAccount / set state=inactive
Frozen
when(balance<0)
when(balance≥0)
Closed
after(5 years)
Overdrawn
deposit(m) / addMoney(m)
withdraw(n) / subtractMoney(n)
when(balance < overdraftLimit) / notifyManager
after(3 months) / send BankManager.accountFrozen(account#)

STATE MACHINE DIAGRAM: STATE

A *state* is a time during the life of an object when it satisfies some condition, performs some action or waits for an event.

☞ **A state has duration.**

- A state may be characterized by the:
  - value of one or more attributes of the class (e.g., a BankAccount object can be overdrawn or in credit based on the value of its *balance* attribute).
  - existence of a link to another object (e.g., a BankAccount object may be an individual or joint account based on the existence of links to one or two Customer objects).

STATE MACHINE DIAGRAM: SPECIAL STATES

● **initial state (start state)**

- ☞ **Each diagram must have at most one initial state.**

◉ **final state (end state)**

- ☞ **Each diagram can have multiple final states.**

☞ **No initial or final state indicates looping behaviour.**

- States may be named.

- States may be unnamed (called anonymous states)

STATE MACHINE DIAGRAM: TRANSITION

→ **A *transition* is a change of state from an originating state (source state) to a successor state (target state).**

☞ **The source and target states may be the same state.**

☞ **A transition takes zero time and cannot be interrupted.**

- Transition adornments (*all are optional*) include:

  event trigger [guard condition] / effect list

  – **event trigger** → an event name plus optional parameters.
    - ➢ An *event trigger* is the (implicit) event that causes a transition to occur.
    - ➢ The event is said to **trigger** the transition; the transition is said to **fire**.

  – **guard condition** → a Boolean expression which must be true for the transition to fire.

  – **effect list** → an atomic procedural expression executed if and when a transition fires.

STATE MACHINE DIAGRAM: TRANSITION (cont'd)

event trigger

– The parameters of an event are available within effects specified on the transition or within activities initiated in the target state.

guard condition

– The condition is written in terms of parameters of the triggering event and attributes and links of the object.

– A guarded transition only fires when its event occurs and if the condition is true.

– A guard condition can specify tests on concurrent states of the state machine diagram (see later slides), e.g., in State1; not in State2.

– State addressing uses the notation State1::State2:: … ::State*n*.

effect list

– An effect list may contain more than one action, which may include call, send, and other kinds of actions.

– An effect list is written in terms of operations, attributes and links of the source object and parameters of the triggering event.

STATE MACHINE DIAGRAM: EVENT

An *event* is something that happens at an instantaneous point in time.

Event types

call event – the receipt of a *synchronous call* from an object (a request that an operation be performed)

change event – a specified Boolean condition becoming true → when(balance < 0)

time event – absolute time → when(date=07/03/2009)
– elapsed time → after(10 seconds)

signal event – the receipt of an *asynchronous communication* from an object

STATE MACHINE DIAGRAM: EVENT TYPES

call event

- The event trigger specifies the operation and the parameters of the event trigger are the parameters of the operation.

change event

- A change event occurs whenever the value of a designated condition (expression) changes from *false* to *true.*
- All values in the Boolean expression must be attributes of the state machine's object; there are no parameters of the event.
- *A change event implies a continuous test for the condition.*
  Note: A change event is not the same as a guard since a guard is passive (evaluated only when it is encountered) while a change event is active (evaluated continuously).

signal event

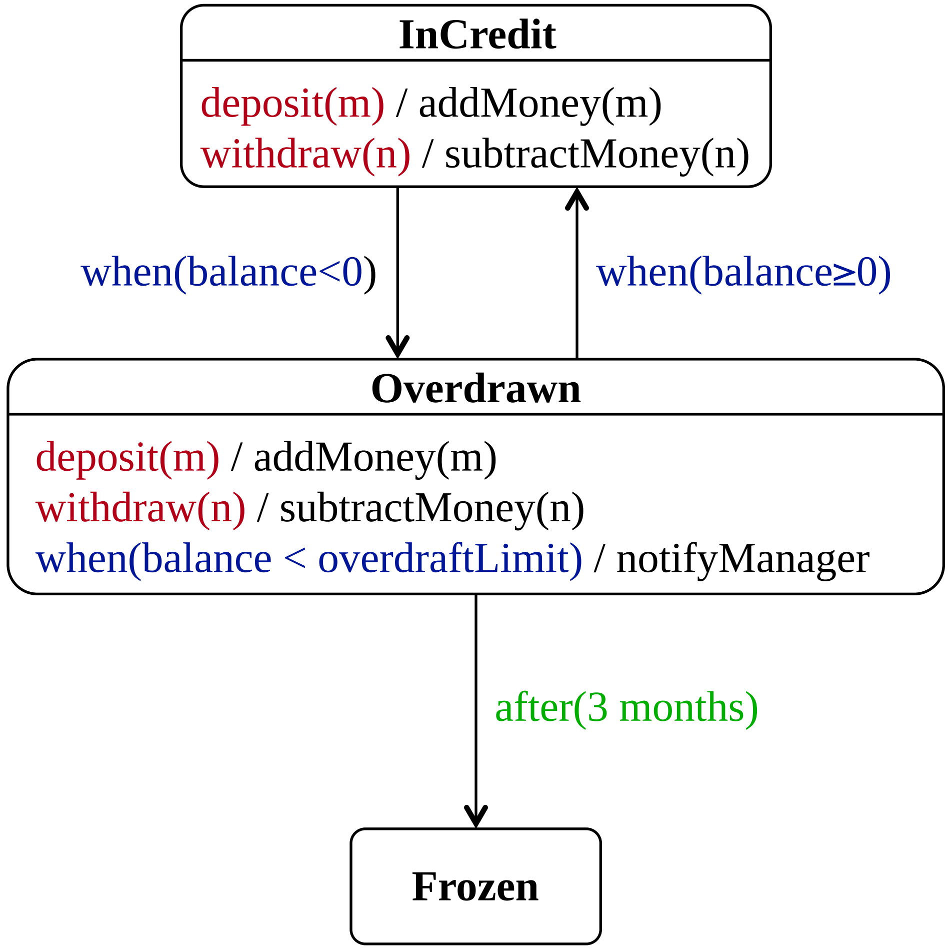
- A signal event is modeled as a stereotyped class («signal») and designates that a package of information is sent asynchronously between objects.
- It is most often used in real-time systems specification.

STATE MACHINE DIAGRAM: EVENT TYPE EXAMPLES

Call event

Change event

Time event



STATE MACHINE DIAGRAM: EVENT / TRANSITION MECHANICS

- Events are processed one at a time.

  ☞ **If the event does not trigger any transition it is ignored.**

- Only one transition within a (sequential) state machine diagram may fire.
  - If two transitions can fire, then the choice may be nondeterministic (i.e., a race condition) → *This is probably a specification error!*

  ☞ **All state transitions must correspond to different events.**

- There are two ways to transition out of a state:
  - **automatic** - when the activity of the state completes.

    ☞ **Transitions without labels fire immediately when the state activity, if any, completes.**
  - **non-automatic** - caused by an event.

STATE MACHINE DIAGRAM: ACTIONS & ACTIVITIES

action - instantaneous and cannot be interrupted

☞ Processing for transitions between states and entry/exit of a state.

activity - takes time to complete and can be interrupted

☞ Processing that occurs while in a state.

- Possible forms of state behavior:

  no behavior → wait until an event occurs that exits the state.

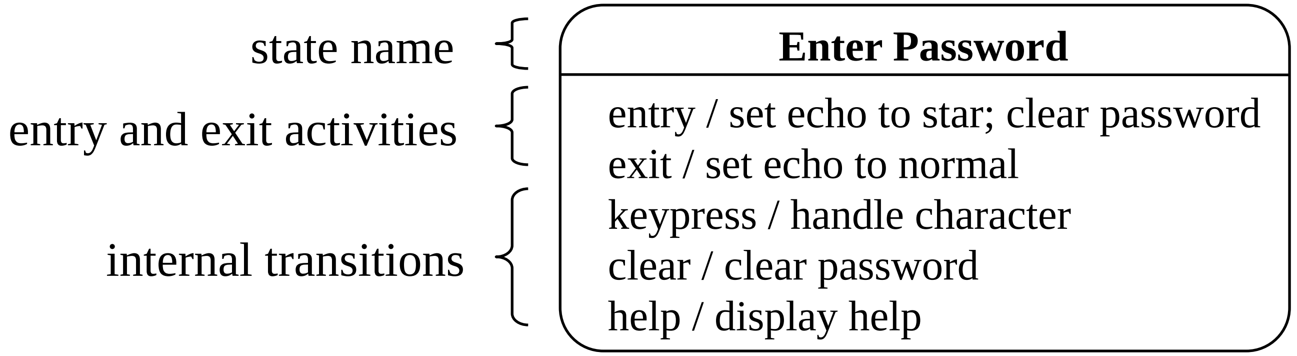
  event trigger [guard condition] / effect list → when *event trigger* occurs and *guard condition* is true, *effect list* is performed.

  do / activity → ongoing execution of behaviour (e.g., operations).

  entry / activity → performed every time the state is entered.

  exit / activity → performed every time the state is exited.

STATE MACHINE DIAGRAM: ACTIONS & ACTIVITIES EXAMPLE


state name
entry and exit activities
internal transitions
Enter Password
entry / set echo to star; clear password
exit / set echo to normal
keypress / handle character
clear / clear password
help / display help

ASU Course Registration System

System Analysis

Class Analysis

State Machine Diagram: *Section Class*

ASU COURSE REGISTRATION REQUIREMENTS

At the beginning of each term, students may request a course catalogue containing a list of course offerings needed for the term. Information about each course, such as instructor, department, and prerequisites are included to help students make informed decisions.

The new system will allow students to select four course sections for the coming term. In addition, each student will indicate two alternative choices in case a course section becomes filled or is canceled. No course section will have more than forty students or fewer than ten students. A course section with fewer than ten students will be canceled. Once the registration process is completed for a student, the registration system sends information to the billing system so the student can be billed for the term.

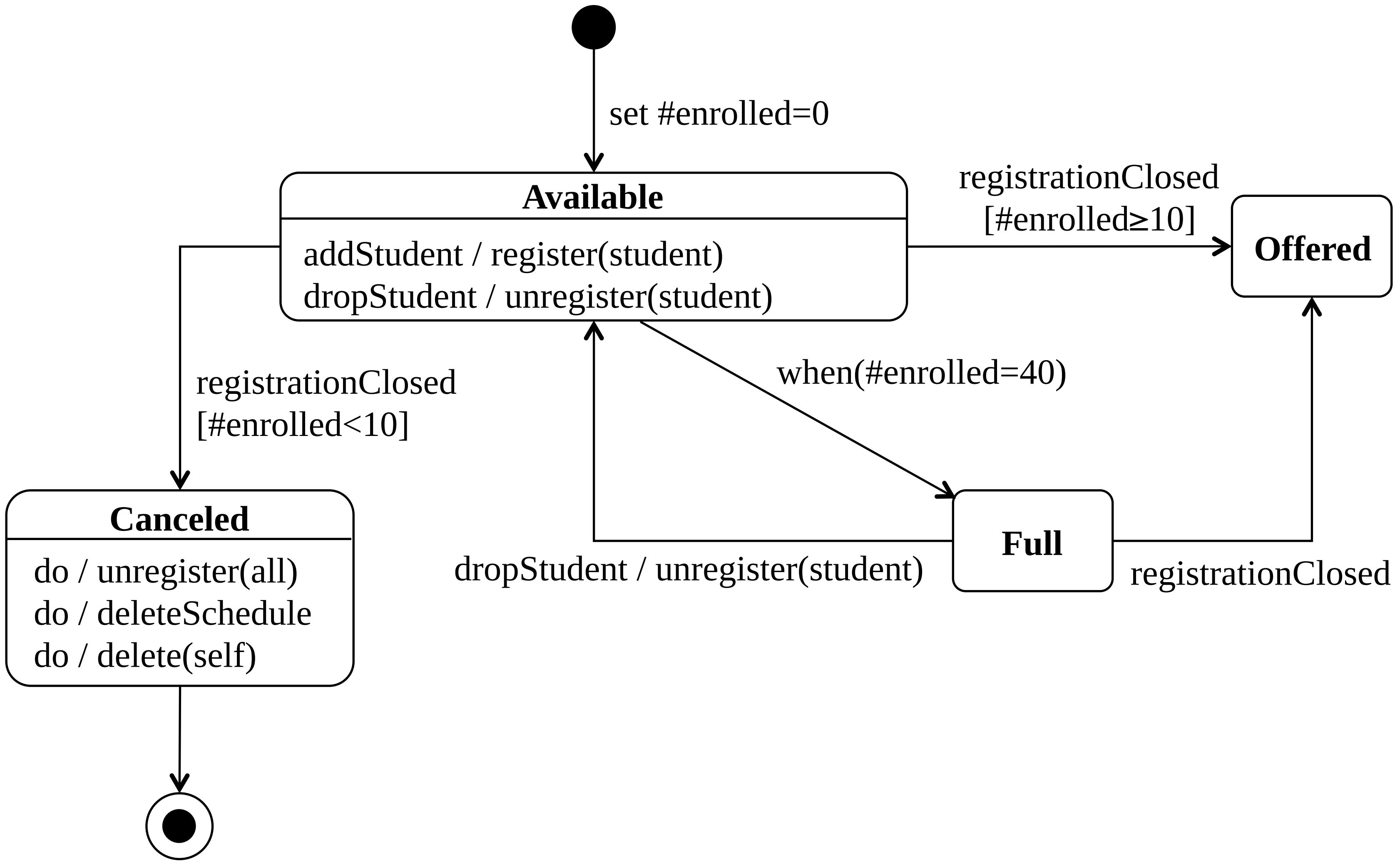
Professors must be able to access the online system to indicate which courses they will be teaching, and to see which students signed up for their course offerings.

For each term, during the registration period that students can change their schedule. Students must be able to access the system during this time to add or drop courses.

ASU STATE MACHINE DIAGRAM: SECTION CLASS

- To construct a state machine diagram we ask the following questions:
  - What states can the class be in?
  - What determines the state that the class is in?
  - To what events (messages) does each state respond and what happens when the event occurs?

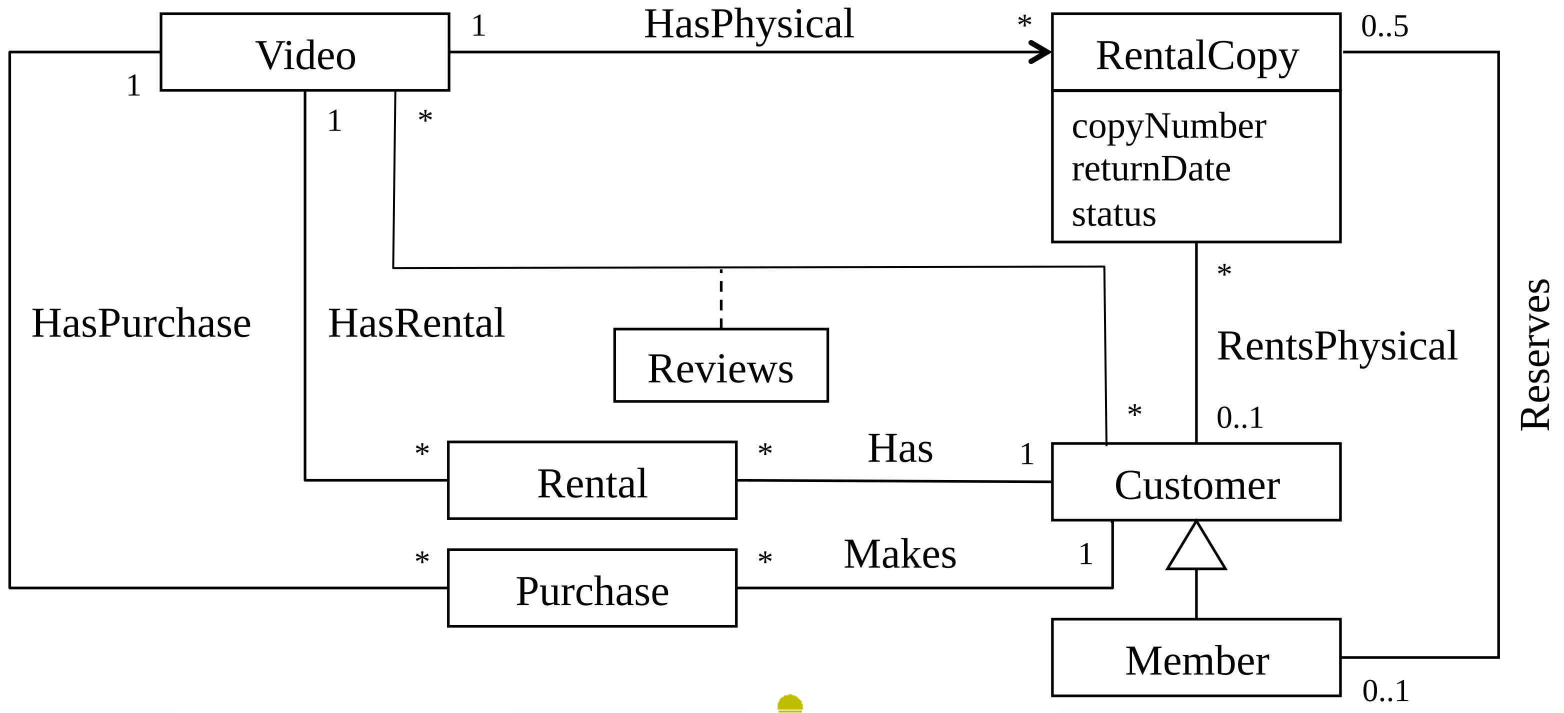
ASU STATE MACHINE DIAGRAM: SECTION CLASS



set #enrolled=0
Available
addStudent / register(student)
dropStudent / unregister(student)
registrationClosed [#enrolled≥10]
Offered
when(#enrolled=40)
registrationClosed [#enrolled<10]
Canceled
do / unregister(all)
do / deleteSchedule
do / delete(self)
dropStudent / unregister(student)
Full
registrationClosed

STATE MACHINE DIAGRAM EXAMPLE

Part of the information kept in the **RentalCopy** class in the domain model shown below is the rental status of a video. Construct a state machine diagram showing the states that an instance of the **RentalCopy** class can be in *with respect to its rental status*. Show only the states, transitions and the events and/or conditions, if any, that cause a transition to be taken. *Do not show the activities that can occur within a state.*



STATE MACHINE DIAGRAM EXAMPLE

The problem statement requirements that could be relevant to determining the states of a RentalCopy object:

- It must be able to record which videos are sold and rented and by whom.
- For sold videos, the quantity sold should be recorded; for physical video rental, which copy is rented and when it is due back should be recorded.
- The system should keep track of overdue rentals of physical videos and send email notices to customers who have videos overdue.
- Members should be able to make reservations for physical video rentals either in person at the shop, by telephone or via the Web.
- A member can reserve at most five physical videos at any one time, but there is no limit on how many physical videos a member or nonmember can rent at any one time.
- A sales clerk should be able to sell and rent physical videos and process the return of rented physical videos.

WHEN TO USE A STATE MACHINE DIAGRAM

- A state machine diagram is good at describing the behavior of an object across several use cases.

☞ **It is not necessary to produce a state machine diagram for every class.**

- A state machine diagram should be used only for classes with **significant** dynamic behaviour.

DESIGN PATTERNS

A *design pattern* is a general reusable solution to a commonly occurring problem in software design.

- Represents a *solution* to a common *problem* that arises within a particular *context* when developing software.

  ☞ design pattern ≡ problem/solution pairs *in a context*

  ☞ **Particularly useful for describing how and why to resolve *nonfunctional requirements*.**

- It is a description or template for how to solve a certain problem that can be used in many different situations.

  ☞ **It is not a finished design that can be transformed directly into code.**

DESIGN PATTERNS

A *design pattern* is a general reusable solution to a commonly occurring problem in software design.

- It typically shows relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

  ☞ **Design patterns make extensive use of inheritance and delegation.**

- Facilitates reuse of successful software designs.

  ☞ Helps novices to learn by example to behave more like experts.

  ☞ A pattern catalog documents design patterns that are useful in a certain context.

HOW TO BECOME A CHESS MASTER

- First learn rules and physical requirements
  - e.g., names of pieces, legal movements, chess board geometry and orientation, etc.

- Then learn principles
  - e.g., relative value of certain pieces, strategic value of center squares, power of a threat, etc.

- However, to become a chess master, one must study the games of other masters.
  - These games contain *patterns* that must be understood, memorized and applied repeatedly and correctly.

There are hundreds of these patterns!

HOW TO BECOME A SOFTWARE DESIGN MASTER

- First learn the rules
  - e.g., the algorithms, data structures and languages of software.
- Then learn the principles
  - e.g., structured programming, modular programming, object-oriented programming, generic programming, etc.
- However, to truly master software design, one must study the designs of other masters.
  - These designs contain *patterns* that must be understood, memorized and applied repeatedly and correctly.

There are hundreds of these patterns!

INTRODUCTION TO DESIGN PATTERNS: SIMUDUCK

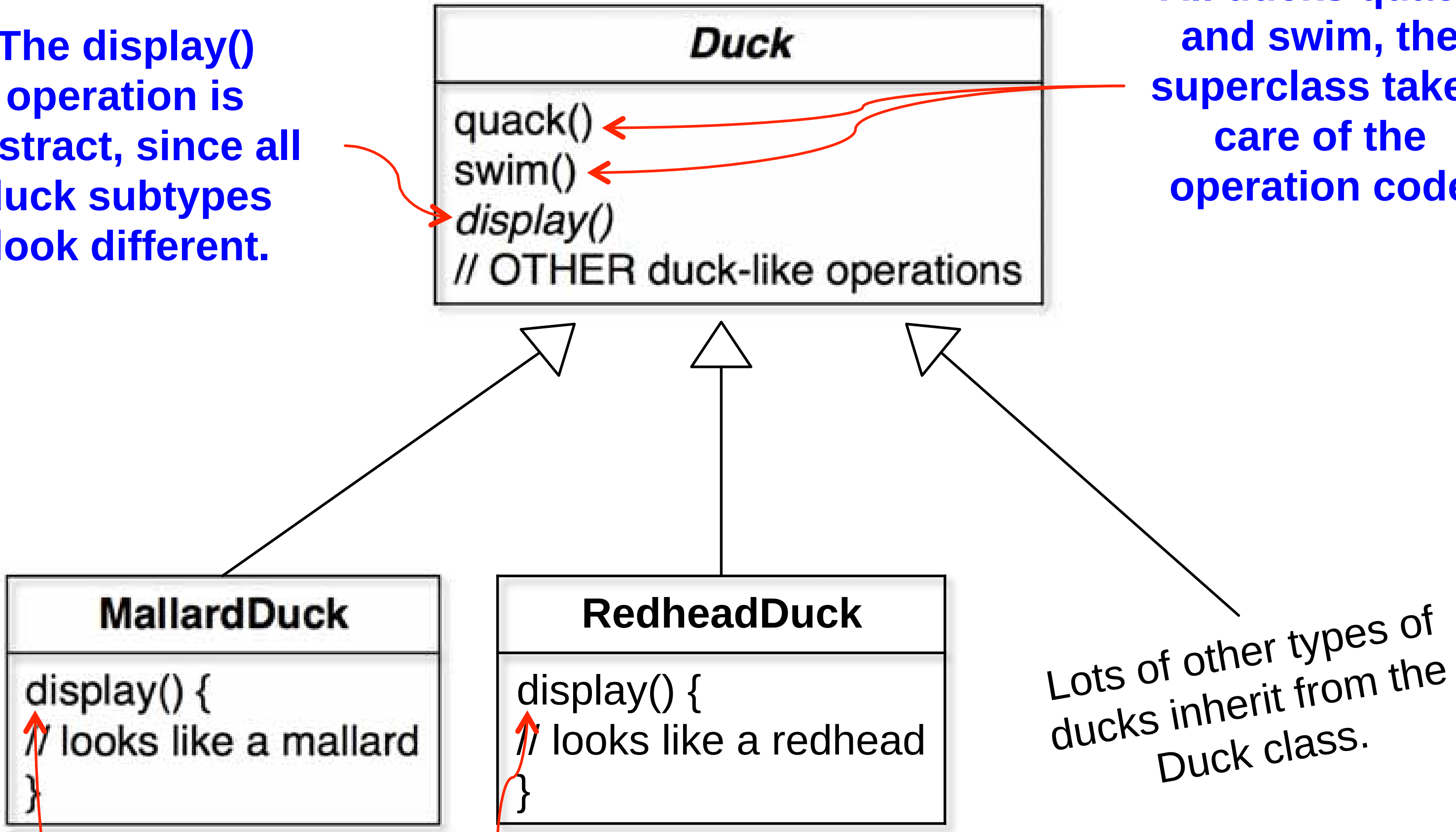
Joe works on SimUDuck.app

Meet Joe

... a successful duck pond simulation game

SimUDuck example from Tom Zimmermann, Microsoft Research

SIMUDUCK IS OO



The display() operation is abstract, since all duck subtypes look different.

All ducks quack and swim, the superclass takes care of the operation code.

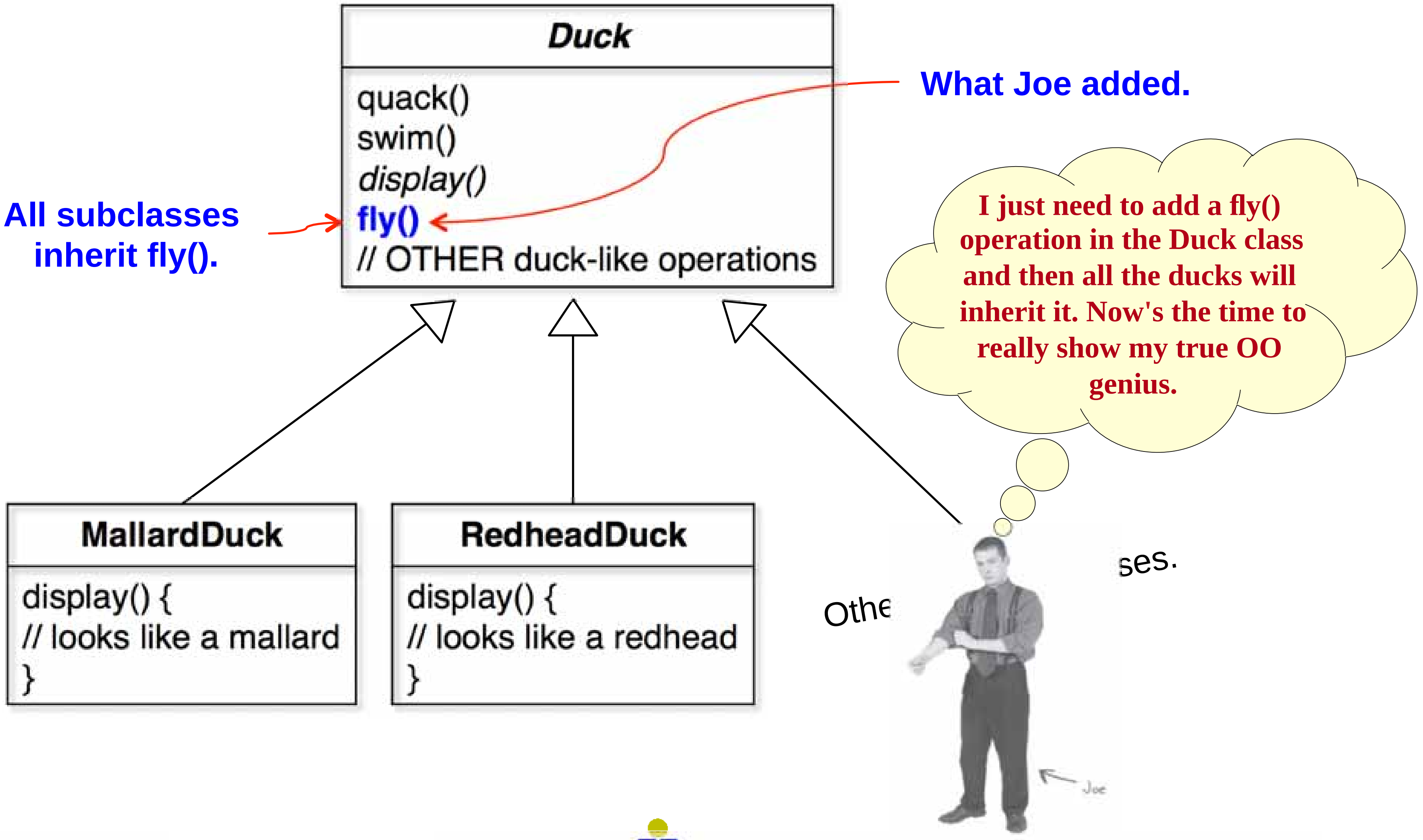
Each duck subtype is responsible for implementing its own display() operation for how it looks on the screen.

SIMUDUCK

Prepare for the big innovation …

now ducks need to fly.

SIMUDUCK:
ADD FLY() OPERATION TO DUCK CLASS


Duck
quack()
swim()
display()
fly()
// OTHER duck-like operations
What Joe added.
All subclasses inherit fly().
I just need to add a fly() operation in the Duck class and then all the ducks will inherit it. Now's the time to really show my true OO genius.
MallardDuck
display() {
// looks like a mallard
}
RedheadDuck
display() {
// looks like a redhead
}
Othe
ses.
Joe

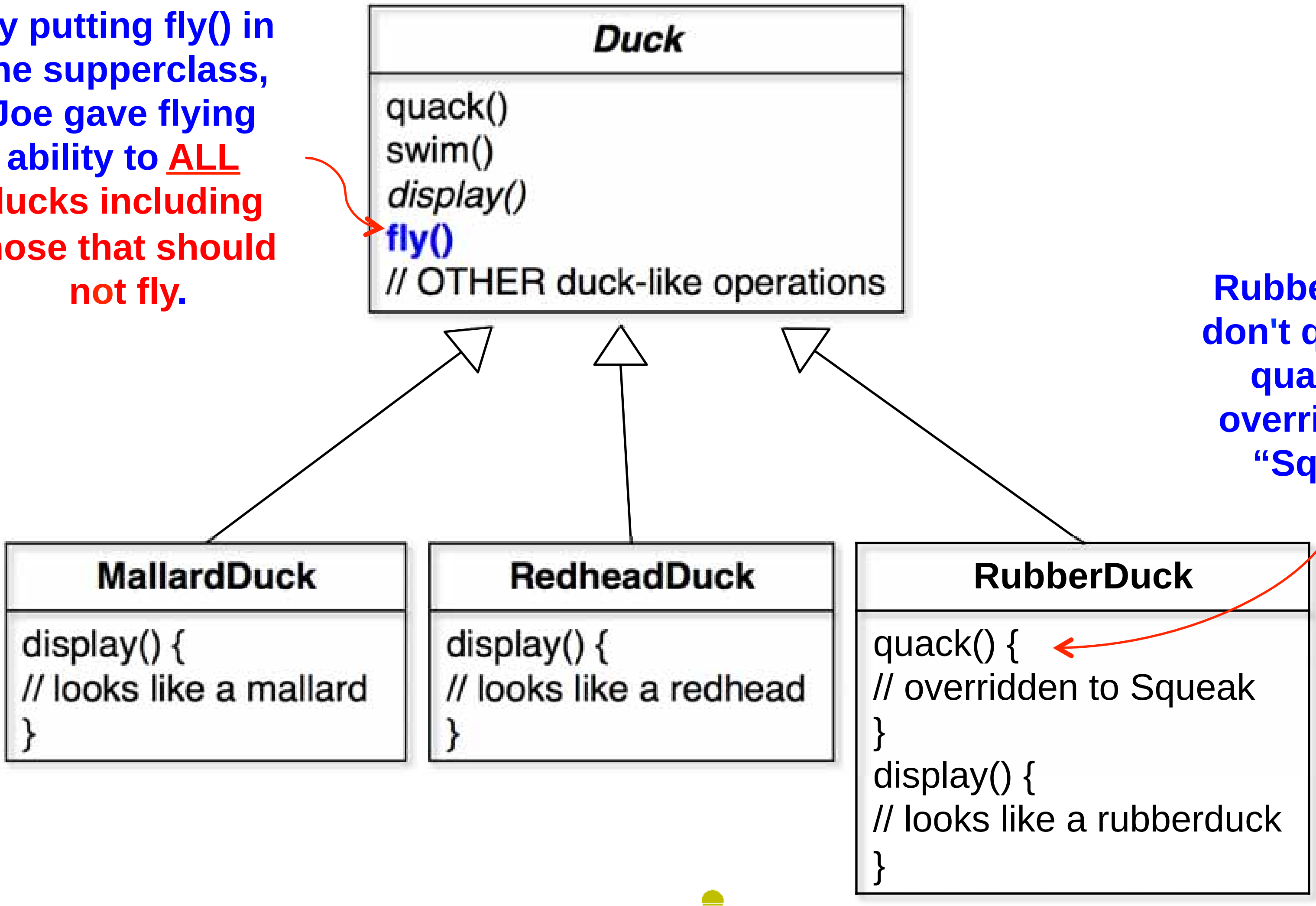
SIMUDUCK: ADD FLY() TO DUCK

But something went wrong …

flying rubber duckies!

SIMUDUCK: WHAT HAPPENED?

By putting fly() in the supperclass, Joe gave flying ability to ALL ducks including those that should not fly.



Rubber ducks don't quack, so quack() is overridden to “Squeak”.

SIMUDUCK: WHAT NOW?

I could always just override the fly() operation in rubber duck, the way I do with the quack() operation …

RubberDuck

quack() {// override to Squeak}
display() {// rubberduck}
fly() {\\ override to do nothing}

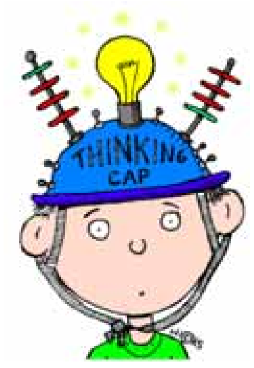
But then what happens when we add wooden decoy ducks to the program? They are not supposed to fly or quack …

Here's another class in the hierarchy. Notice that like RubberDuck, it does not fly, but it also does not quack.

DecoyDuck

quack() {// override to do nothing}
display() {// decoy duck}
fly() {\\ override to do nothing}

SIMUDUCK: PUT ON YOUR THINKING CAP



Which of the following are disadvantages of using *inheritance* to provide Duck behaviour? (Choose all that apply.)

☐ A. Code is duplicated across subclasses.

☐ B. Runtime behaviour changes are difficult.

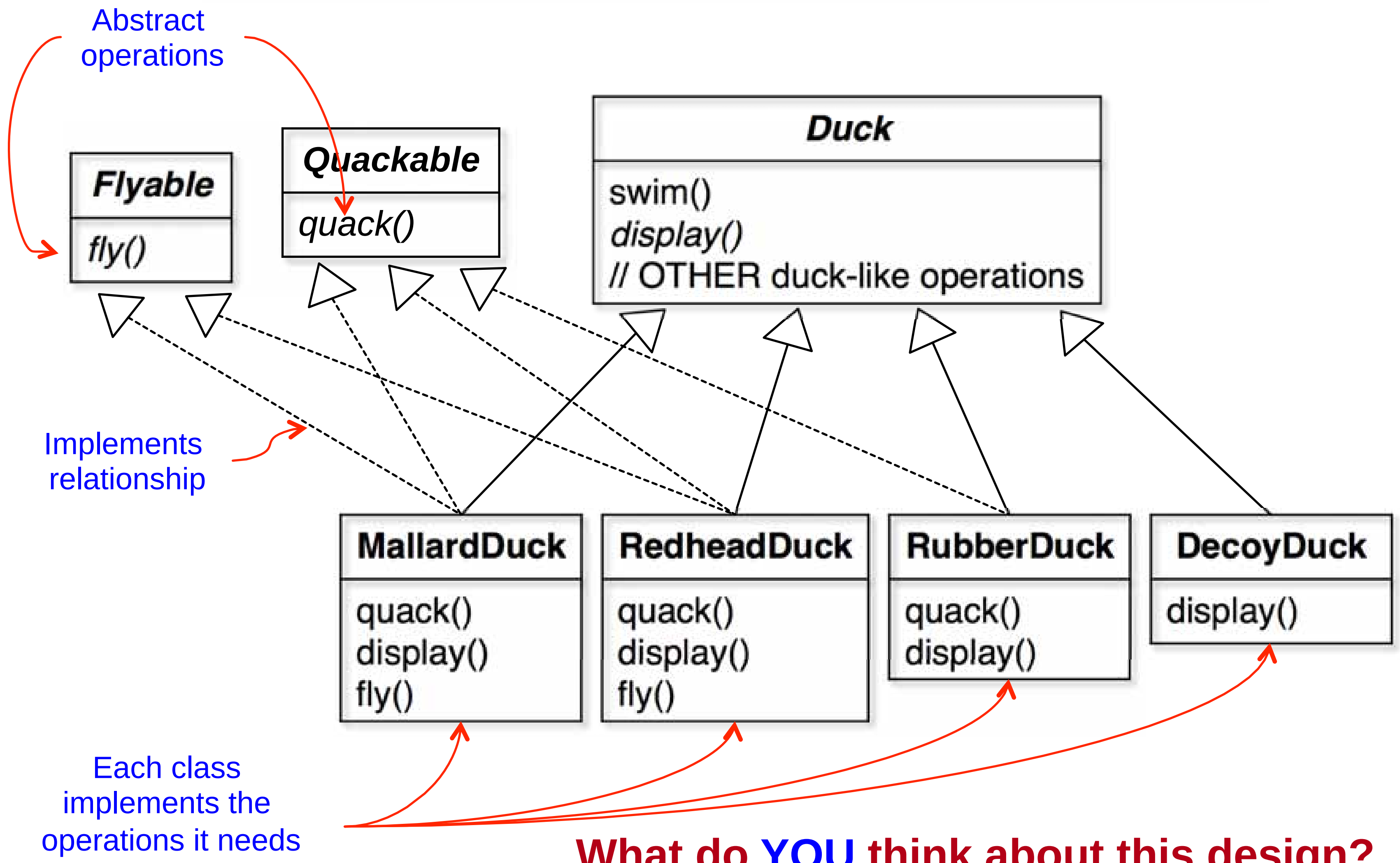
☐ C. We can't make ducks dance.

☐ D. Hard to gain knowledge of all duck behaviours.

☐ E. Ducks can't fly and quack at the same time.

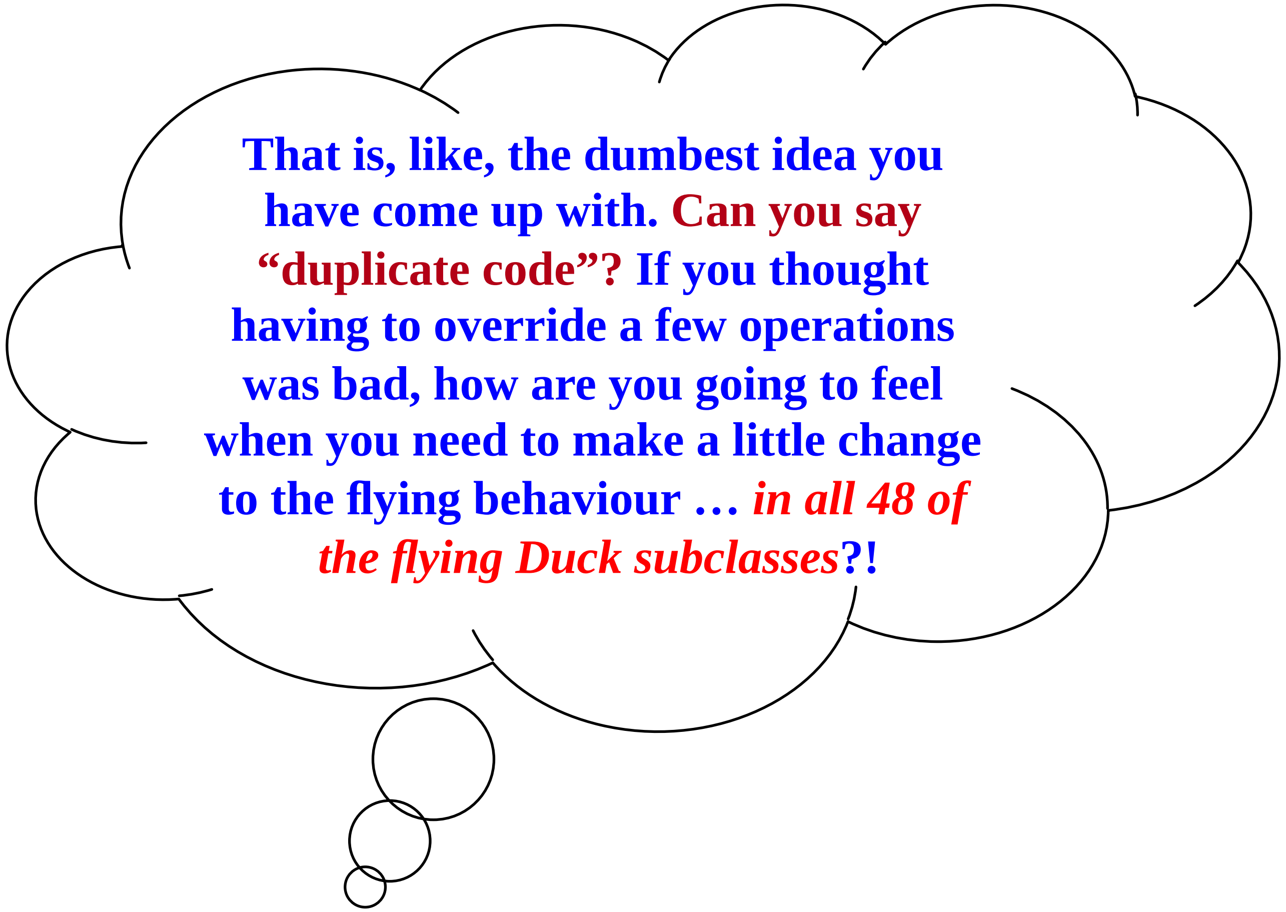
☐ F. Changes can unintentionally affect other ducks.

SIMUDUCK: HOW ABOUT USING AN INTERFACE?



What do YOU think about this design?

SIMUDUCK: DUPLICATED CODE



That is, like, the dumbest idea you have come up with. Can you say “duplicate code”? If you thought having to override a few operations was bad, how are you going to feel when you need to make a little change to the flying behaviour … ***in all 48 of the flying Duck subclasses*****?!**

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

Take what varies and "encapsulate" it so it won't affect the rest of your code.

The result?

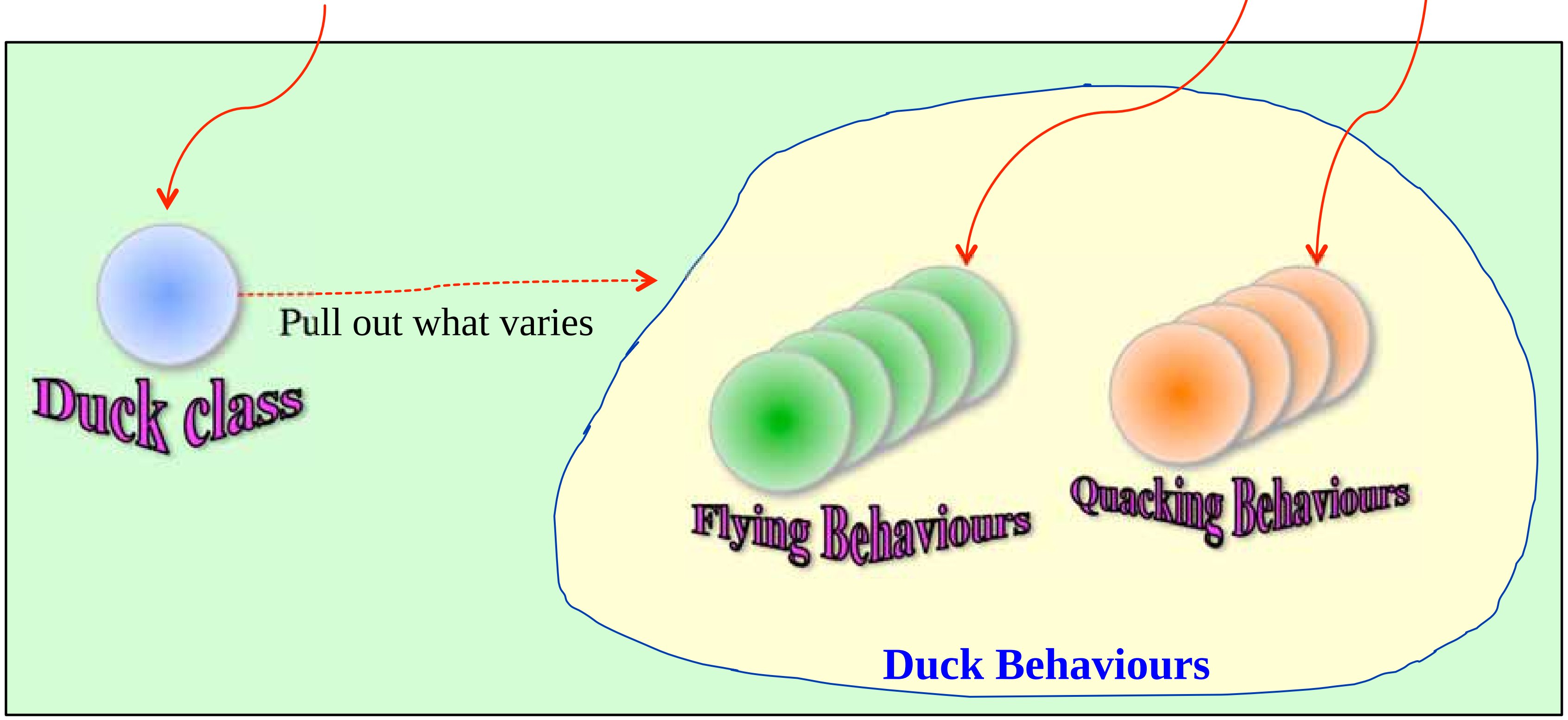
Fewer unintended consequences from code changes and more flexibility in your systems.

SIMUDUCK: SEPARATING THE DUCK BEHAVIOURS

The Duck class is still the superclass of all ducks, but we pull out the fly and quack behaviours and put them into another class structure.

Now flying and quacking each get their own set of classes.

Various behaviour implementations are going to live here.

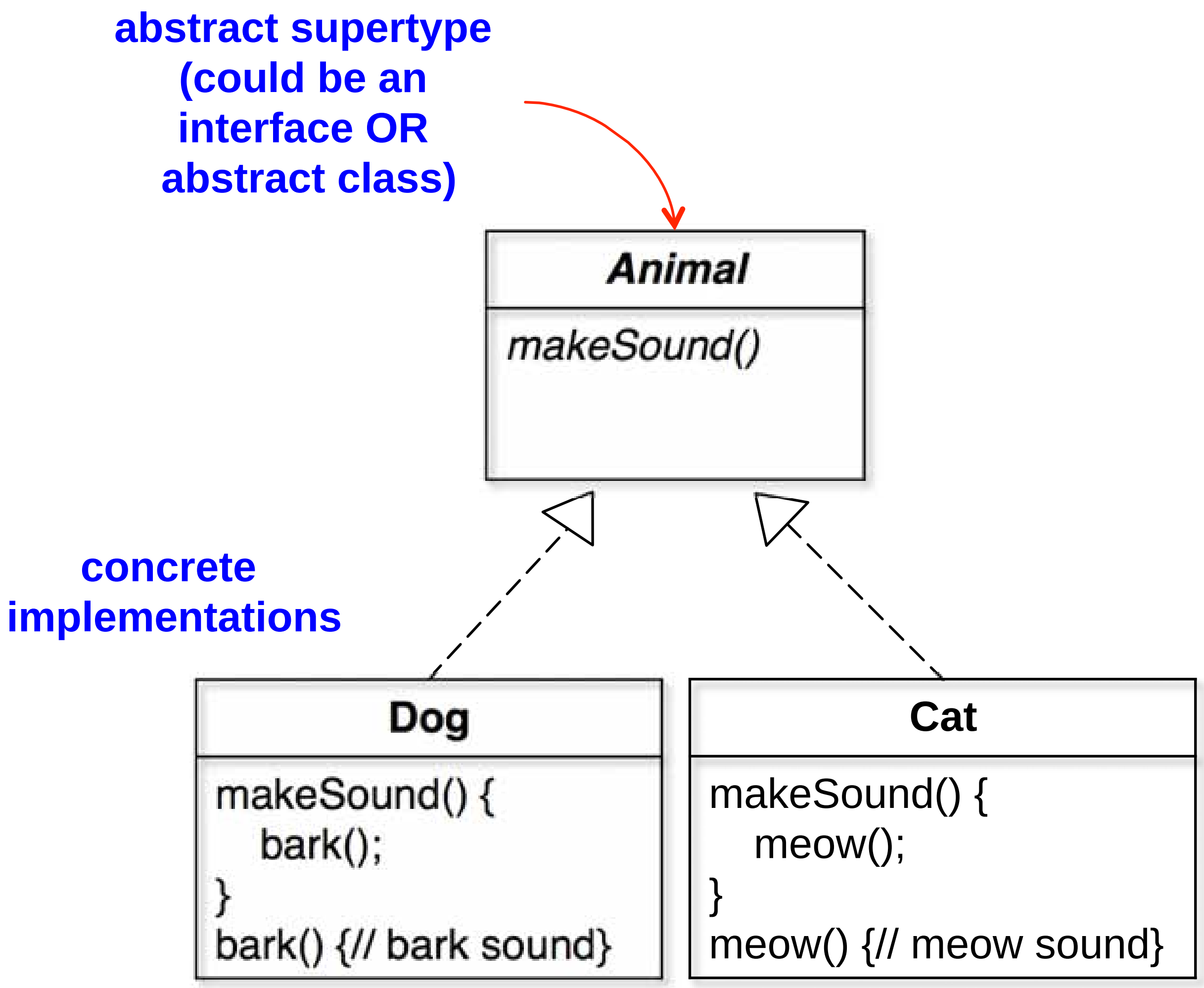


Design Principle

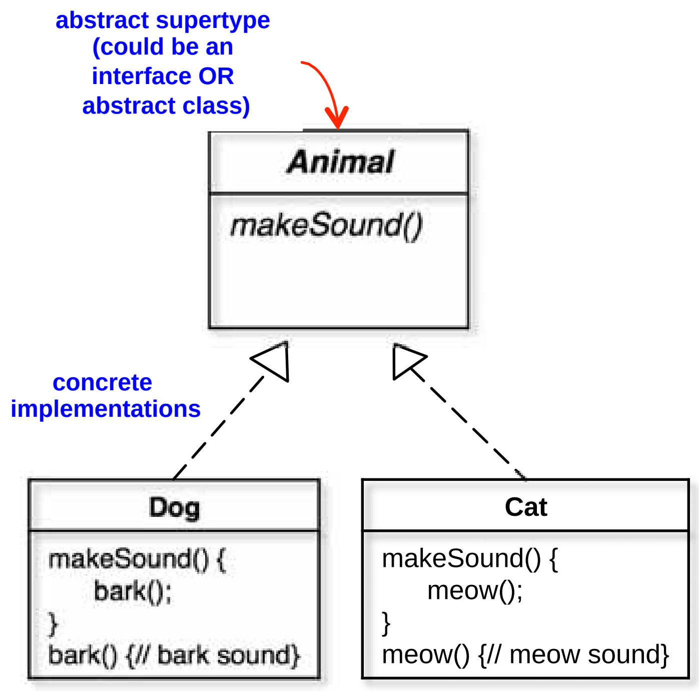
Program to an interface, not an implementation.

That way, the Duck classes won't need to know any of the implementation details of their behaviours.

PROGRAM TO AN INTERFACE/SUPERTYPE



PROGRAM TO AN INTERFACE/SUPERTYPE



Programming to an implementation would be:

```
Dog d = new Dog();
d.bark;
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

But **programming to an interface/supertype** would be:

```
Animal animal = new Dog();
animal.makeSound();
```

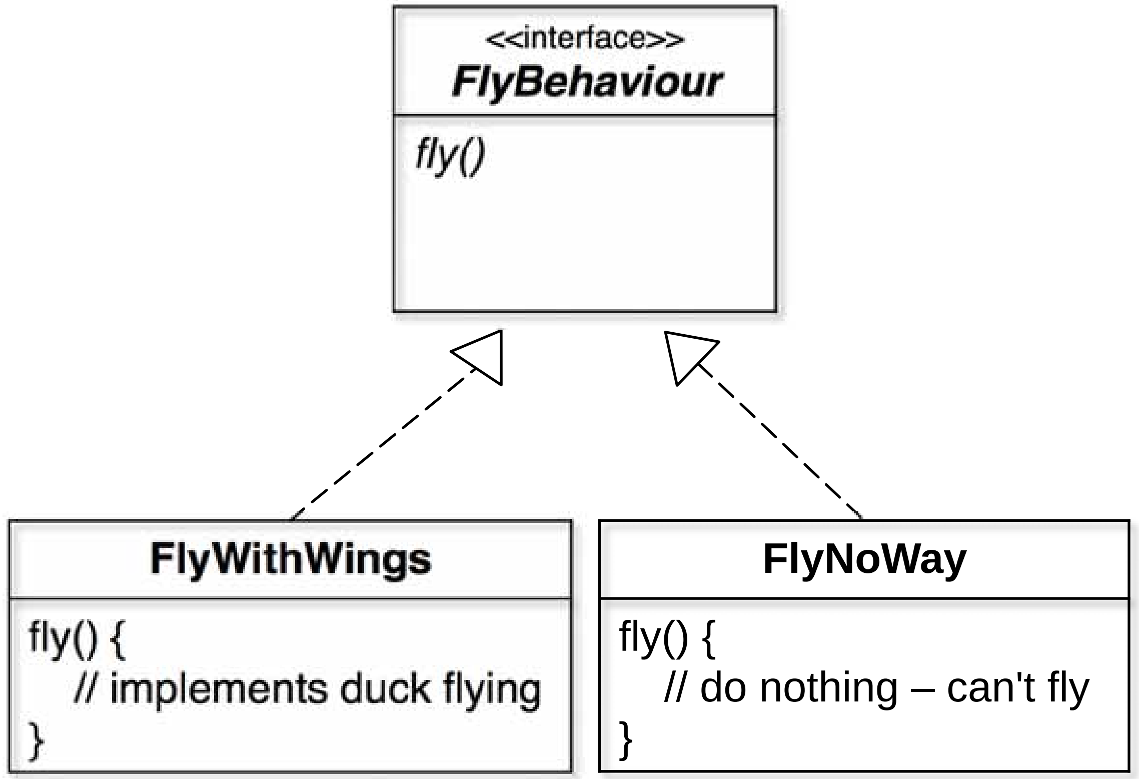
We know it's a Dog, but we can now use the animal reference polymorphically.

Even better, rather than hard-coding the instantiation of the subtype (like `new Dog()`) into the code, **assign the concrete implementation object at runtime**:

```
a = getAnimal();
a.makeSound();
```

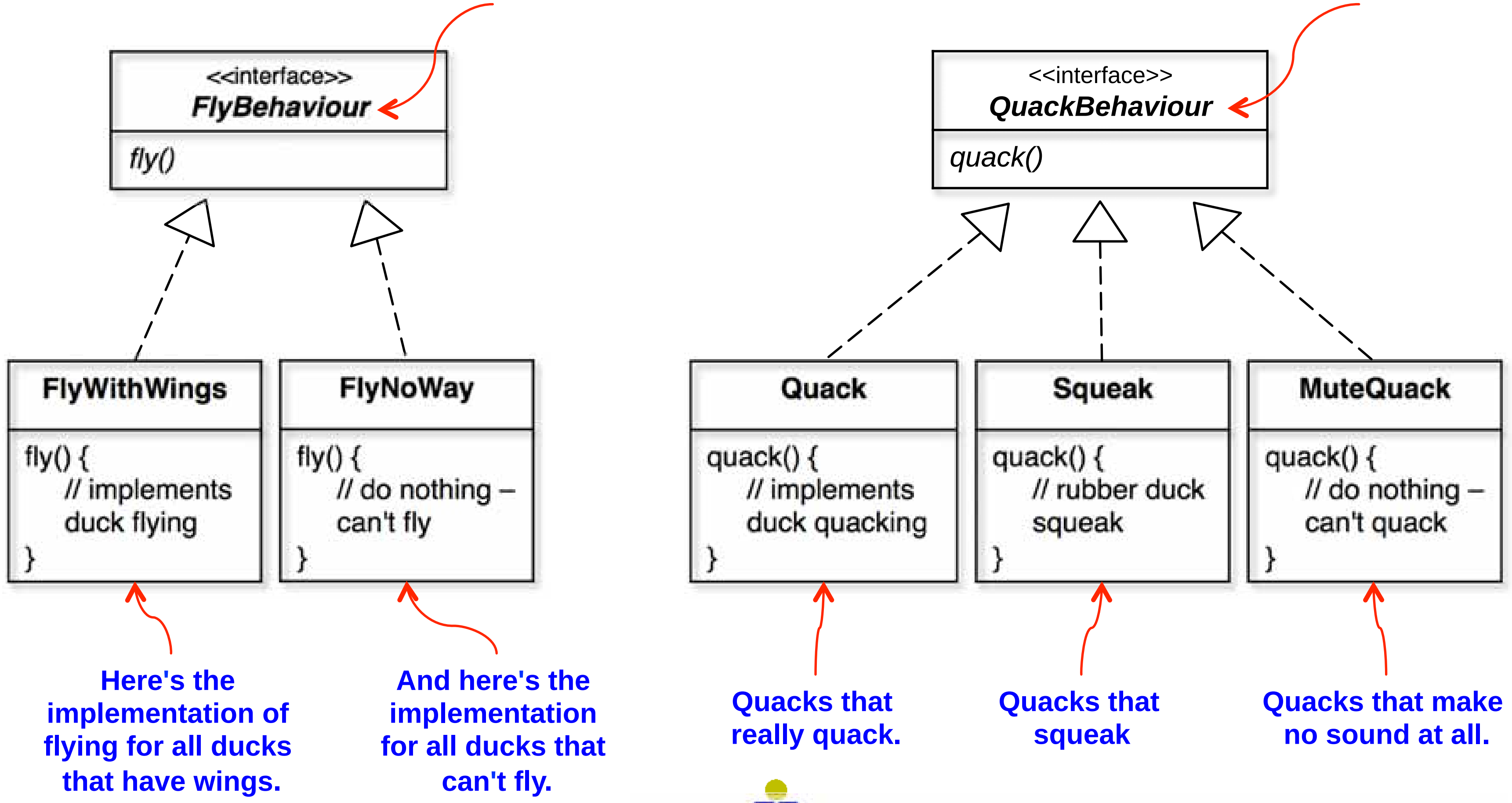
We don't know WHAT the actual animal subtype is … all we care about is that it knows how to respond to makeSound().

USE INTERFACES FOR BEHAVIOURS



similar for QuackBehaviour

SIMUDUCK: DUCK BEHAVIOURS


FlyBehaviour is an interface that all flying classes implement. All new flying classes just need to implement the fly operation.
Same thing here for the quack behaviour; we have an interface that just includes a quack() operation that needs to be implemented.
<<interface>>
FlyBehaviour
fly()
<<interface>>
QuackBehaviour
quack()
FlyWithWings
fly() { // implements duck flying }
FlyNoWay
fly() { // do nothing – can't fly }
Quack
quack() { // implements duck quacking }
Squeak
quack() { // rubber duck squeak }
MuteQuack
quack() { // do nothing – can't quack }
Here's the implementation of flying for all ducks that have wings.
And here's the implementation for all ducks that can't fly.
Quacks that really quack.
Quacks that squeak
Quacks that make no sound at all.

SIMUDUCK: OUR DESIGN ROCKS!

- We can reuse the fly and quack behaviours.
  - They are not hidden inside Duck anymore.

- We can add new behaviours.
  - Without modifying any existing behaviour classes or Duck classes that use behaviours.

- SimUDuck.app now has the **BENEFIT OF REUSE** while still being **MAINTAINABLE**!

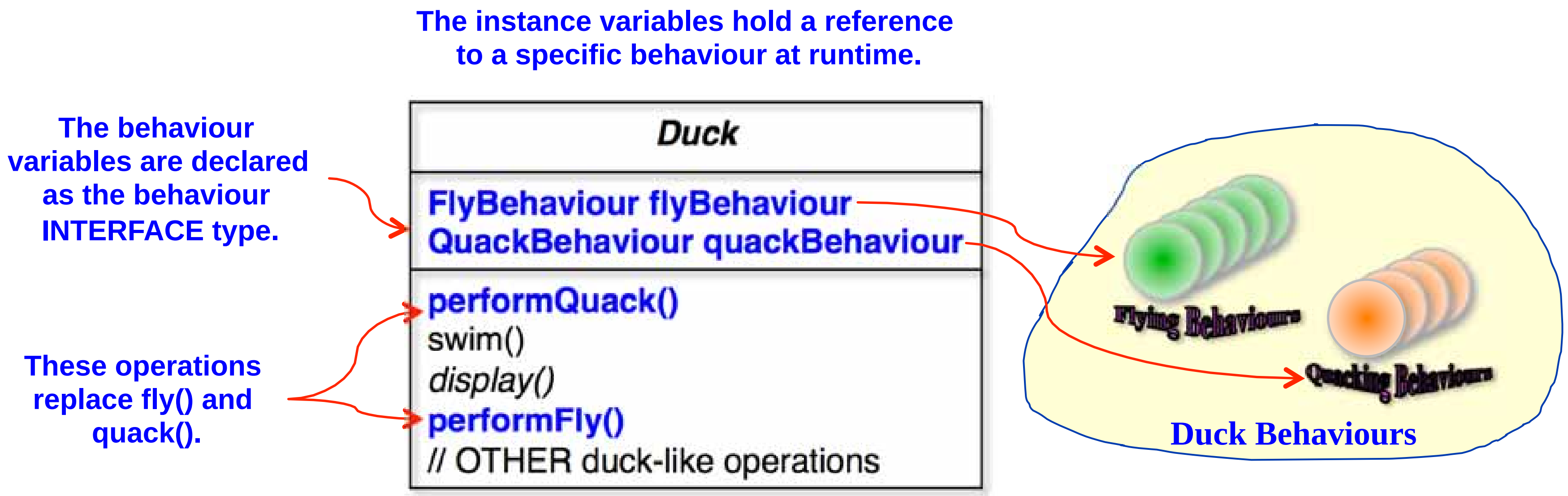
SIMUDUCK: PUT ON YOUR THINKING CAP

1. Using our new design, what would you do if you needed to add rocket-powered flying to the SimUDuck app?

2. Can you think of a class that might want to use the Quack behaviour that is not a duck?

SIMUDUCK: INTEGRATING THE DUCK BEHAVIOUR

① First we will add two instance variables to the Duck class.


The instance variables hold a reference to a specific behaviour at runtime.
The behaviour variables are declared as the behaviour INTERFACE type.
These operations replace fly() and quack().
Duck
FlyBehaviour flyBehaviour
QuackBehaviour quackBehaviour
performQuack()
swim()
display()
performFly()
// OTHER duck-like operations
Flying Behaviours
Quacking Behaviours
Duck Behaviours

SIMUDUCK:
INTEGRATING THE DUCK BEHAVIOUR

② Now we implement performQuack().

```
public class Duck {
   QuackBehaviour quackBehaviour;
   // more

   public void performQuack() {
   quackBehaviour.quack();
   }
}
```

Each Duck has a reference to something that implements the QuackBehaviour interface.

Rather than handling the quack behaviour itself, the Duck object delegates that behaviour to the object referenced by quackBehaviour.

SIMUDUCK: INTEGRATING THE DUCK BEHAVIOUR

```
public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehaviour = new Quack();
        flyBehaviour = new FlyWithWings();
    }
```

A MallardDuck uses the Quack class to handle its quack, so when performQuack is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehaviour type.

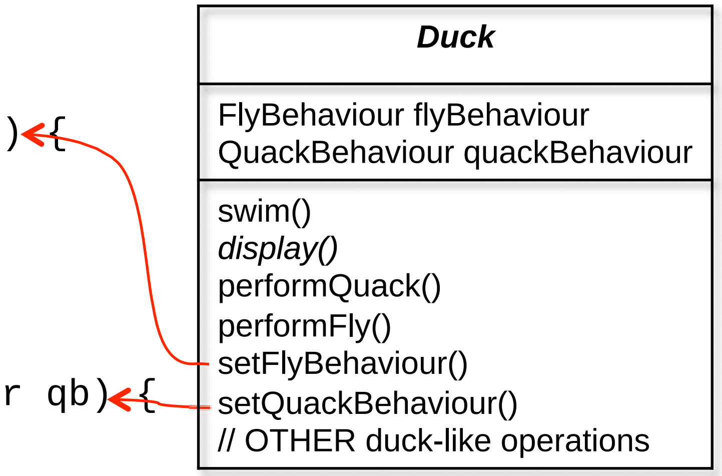
Remember, MallardDuck inherits the quackBehaviour and flyBehaviour instance variables from class Duck.

```
    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

Initialize the behaviours in subclasses.

SIMUDUCK: SETTING BEHAVIOUR DYNAMICALLY

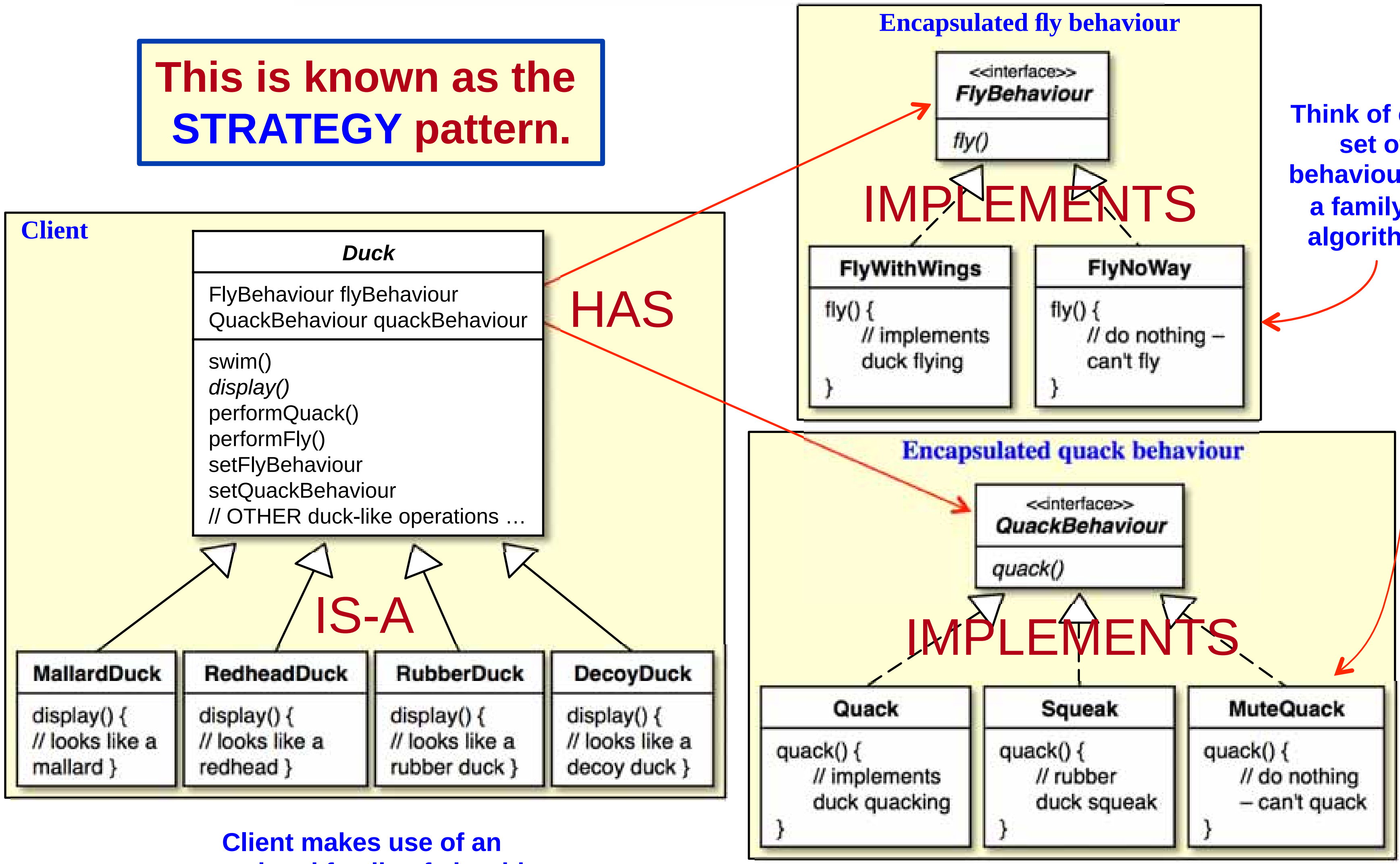
```
public void setFlyBehaviour(FlyBehaviour fb){
    flyBehaviour = fb;
}


public void setQuackBehaviour(quackBehaviour qb){
    quackBehaviour = qb;
}
```



These operations can be called anytime to set the behaviour of a duck on the fly.

SIMUDUCK: THE BIG PICTURE

This is known as the STRATEGY pattern.



Client
Duck
FlyBehaviour flyBehaviour
QuackBehaviour quackBehaviour
swim()
display()
performQuack()
performFly()
setFlyBehaviour
setQuackBehaviour
// OTHER duck-like operations …
HAS
IS-A
MallardDuck
display() { // looks like a mallard }
RedheadDuck
display() { // looks like a redhead }
RubberDuck
display() { // looks like a rubber duck }
DecoyDuck
display() { // looks like a decoy duck }
Encapsulated fly behaviour
<<interface>> FlyBehaviour
fly()
IMPLEMENTS
FlyWithWings
fly() { // implements duck flying }
FlyNoWay
fly() { // do nothing – can't fly }
Encapsulated quack behaviour
<<interface>> QuackBehaviour
quack()
IMPLEMENTS
Quack
quack() { // implements duck quacking }
Squeak
quack() { // rubber duck squeak }
MuteQuack
quack() { // do nothing – can't quack }

Think of each set of behaviours as a family of algorithms.

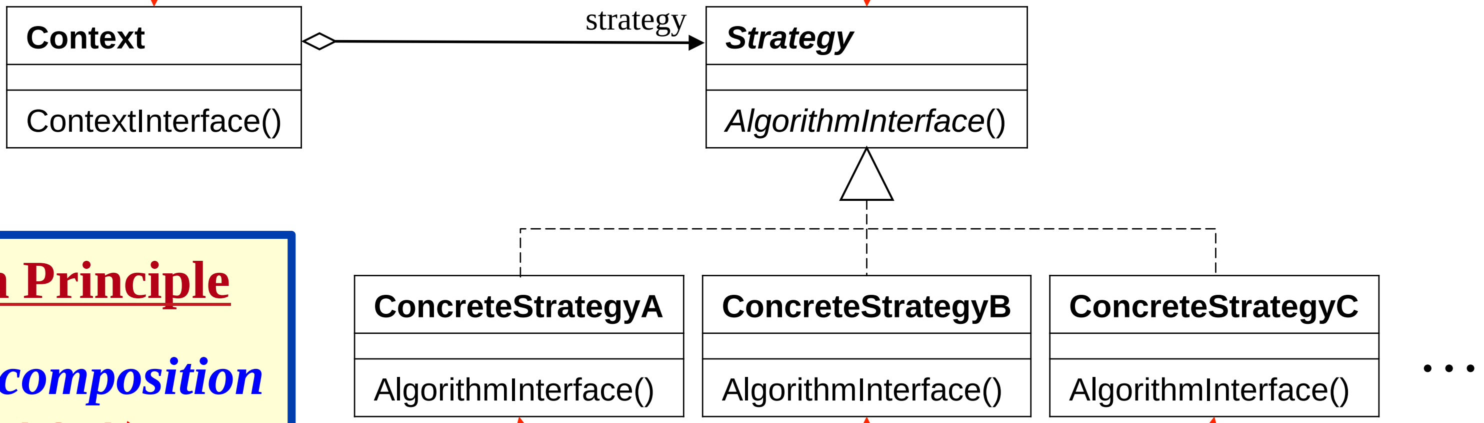
Client makes use of an encapsulated family of algorithms for both flying and quacking.

These ~~behaviours~~ "algorithms" are interchangeable.

STRATEGY PATTERN: CLASS DIAGRAM

The *Context* class is configured with a *ConcreteStrategy* object, maintains a reference to a *Strategy* object and may define an interface that lets *Strategy* access its data.

The *Strategy* class declares an interface common to all supported algorithms. *Context* uses this interface to call the algorithm defined by a *ConcreteStrategy*.



Design Principle

Favour composition (HAS-A) over inheritance (IS-A).

Each *ConcreteStrategy* implements the algorithm using the *Strategy* interface.

STRATEGY PATTERN: DESCRIPTION

Intent

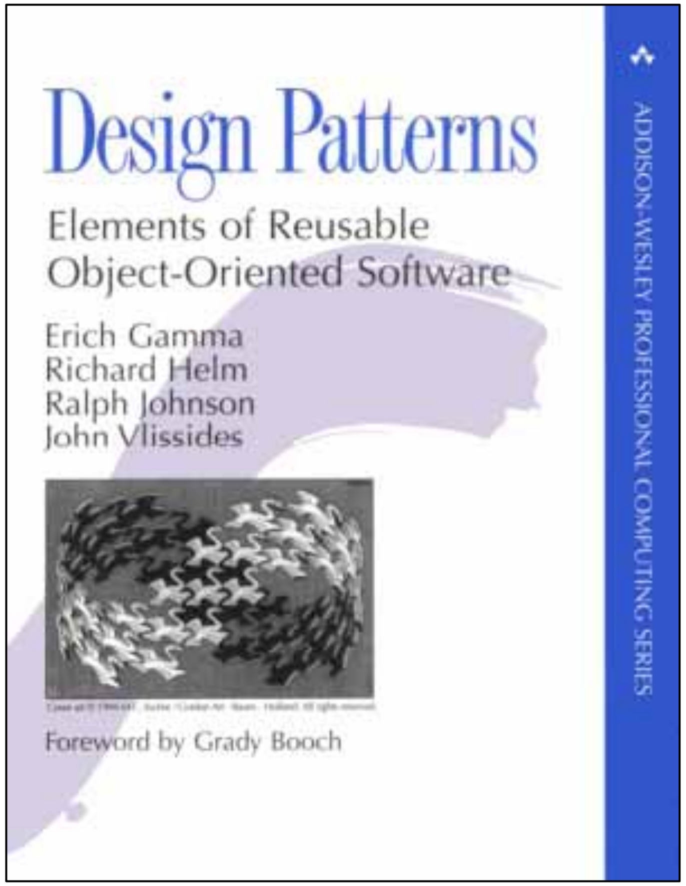
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Applicability (use when …)

- Many related classes differ only in their behaviour. Strategy provides a way to configure a class with one of many behaviours.
- You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs.
- An algorithm uses data that clients should not know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviours and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

DESIGN PATTERNS SPACE

- **Creational patterns**
  - Deal with initializing and configuring classes and objects
- **Structural patterns**
  - Deal with decoupling interface and implementation of classes and objects
- **Behavioural patterns**
  - Deal with dynamic interactions among societies of classes and objects
- **Concurrency patterns**
  - Deal with multi-threaded programming


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

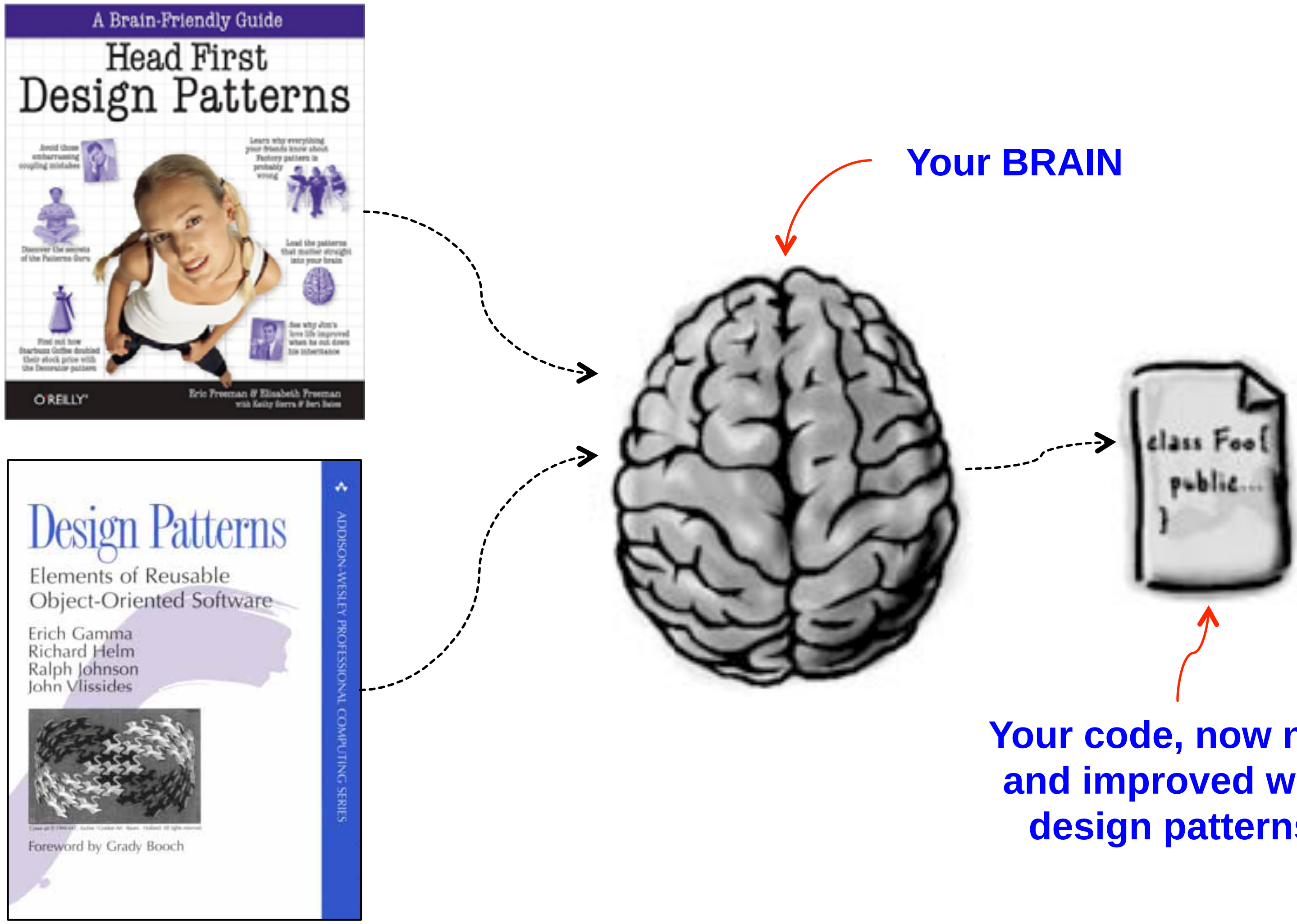
DESIGN PATTERNS: DESCRIPTION

Design patterns are described using the following main parts:

1. Name and intent (what does the pattern do).
2. Problem (the difficulty being addressed).
3. Context (the general situation in which the pattern applies).
4. Concern(s) (issues that need to be considered when solving the problem).
5. Solution (an abstract description of the structure and collaborations in the solution).
6. Positive and negative consequence(s) of use.
7. Implementation guidelines and sample code.
8. Known uses and related patterns.

Design pattern descriptions are usually independent of programming language or implementation details.

DESIGN PATTERNS: HOW TO USE


A Brain-Friendly Guide
Head First Design Patterns
O'REILLY
Eric Freeman & Elisabeth Freeman
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
class Foo{
public...
}

Your BRAIN

Your code, now new and improved with design patterns!

DESIGN PATTERNS SPACE

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Creational

Singleton
Builder
Prototype
Abstract Factory
Factory

Concurrency

Active object
Balking
Read write lock
Monitor object
Reactor
Thread pool
Scheduler

Concurrency patterns manage concurrent interactions among multiple objects.

Behavioural patterns are concerned with how classes and objects interact and distribute responsibility.

Behavioural

Mediator
Template Method
Command
Memento
Observer
Visitor
Iterator
Interpreter
Chain of responsibility
State
Strategy

Structural

Proxy
Decorator
Façade
Bridge
Composite
Flyweight
Adapter

Structural patterns compose classes or objects into larger structures.

OBSERVER PATTERN: EXAMPLE

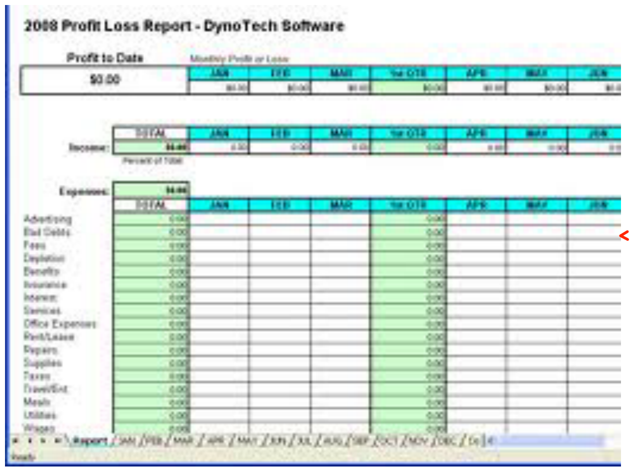
Consider a spreadsheet:

- You enter data into a cell.
- As as you do so, any related formulas and charts change.
- New formulas/charts can be created and are updated as the data changes.

SUBJECT | OBSERVER

~~Data~~ + ~~Charts~~

= Observer Pattern



2008 Profit Loss Report - DynoTech Software

Profit to Date: $0.00

Monthly Profit or Loss

JAN	FEB	MAR	1st QTR	APR	MAY	JUN
$0.00	$0.00	$0.00	$0.00	$0.00	$0.00	$0.0

	TOTAL	JAN	FEB	MAR	1st QTR	APR	MAY	JUN
Income:	$0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0

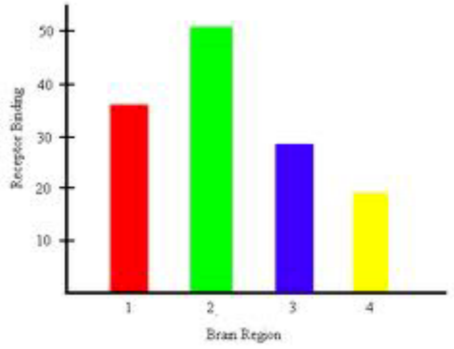
Percent of Total

Expenses: $0.00

	TOTAL	JAN	FEB	MAR	1st QTR	APR	MAY	JUN
Advertising	0.00				0.00			
Bad Debts	0.00				0.00			
Fees	0.00				0.00			
Depletion	0.00				0.00			
Benefits	0.00				0.00			
Insurance	0.00				0.00			
Interest	0.00				0.00			
Services	0.00				0.00			
Office Expenses	0.00				0.00			
Rent/Lease	0.00				0.00			
Repairs	0.00				0.00			
Supplies	0.00				0.00			
Taxes	0.00				0.00			
Travel/Ent	0.00				0.00			
Meals	0.00				0.00			
Utilities	0.00				0.00			
Wages	0.00				0.00			

Report / JAN / FEB / MAR / APR / MAY / JUN / JUL / AUG / SEP / OCT / NOV / DEC / Ex

Ready



Product Y
$12,000
$30,000
$23,000
17,000
$21,000
$30,000

OBSERVER PATTERN

Problem: A large monolithic design does not scale well as new requirements arise.

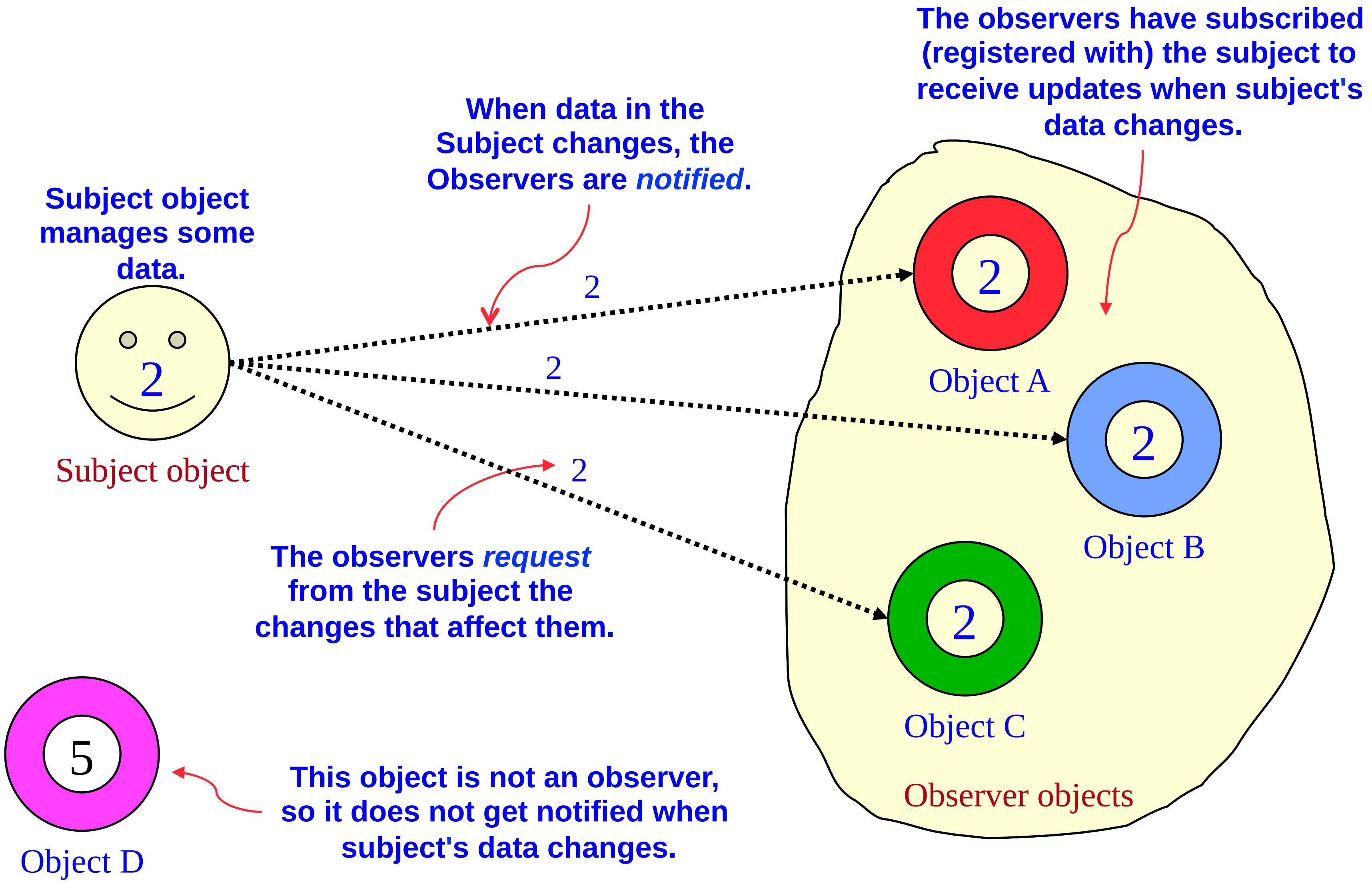
Solution: Define an object that is the "keeper" of the data model and/or business logic (the Subject).

Delegate all "view" functionality to decoupled and distinct Observer objects.

Observers register themselves with the Subject as they are created.

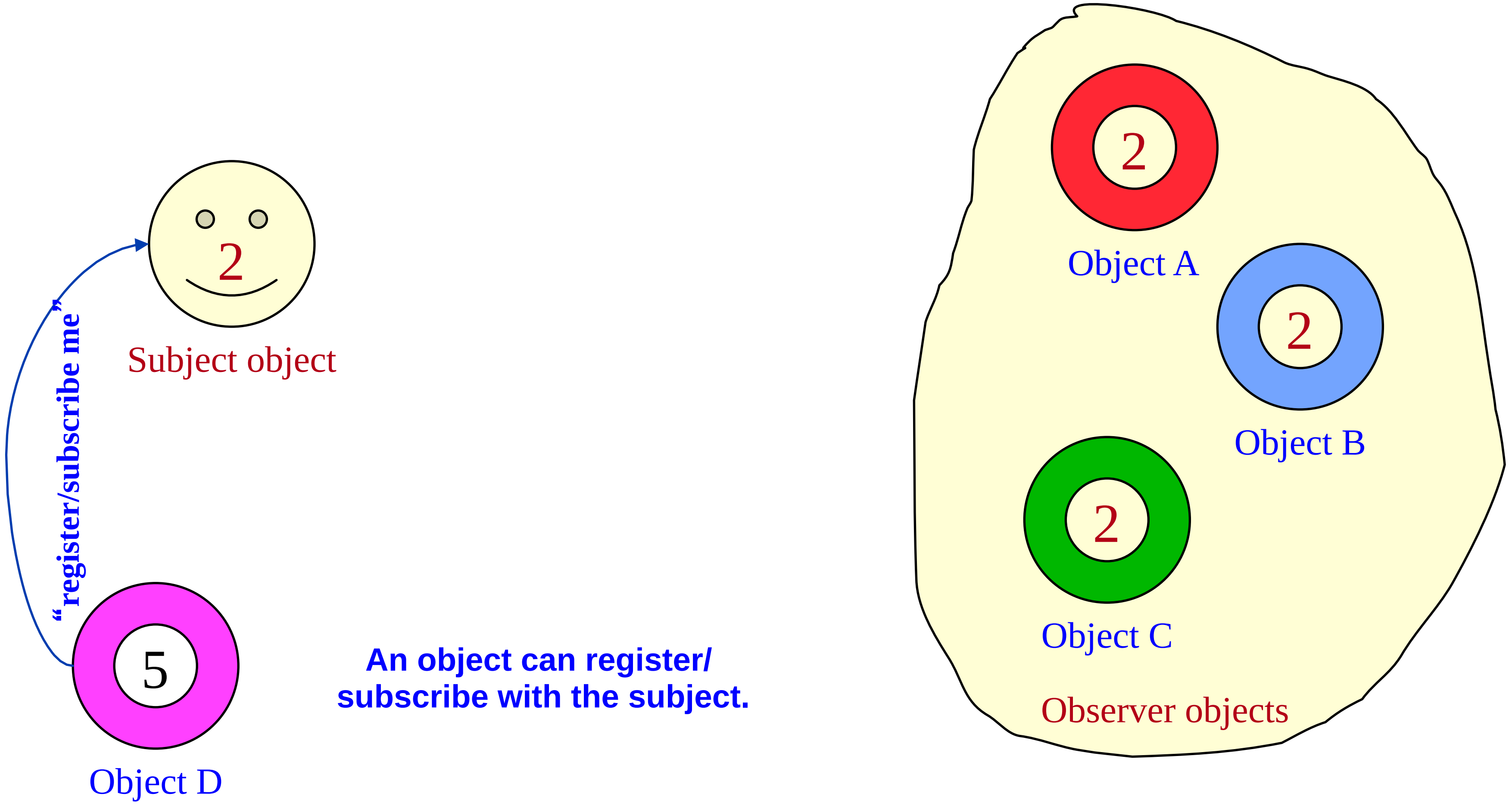
Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

OBSERVER PATTERN: HOW IT WORKS



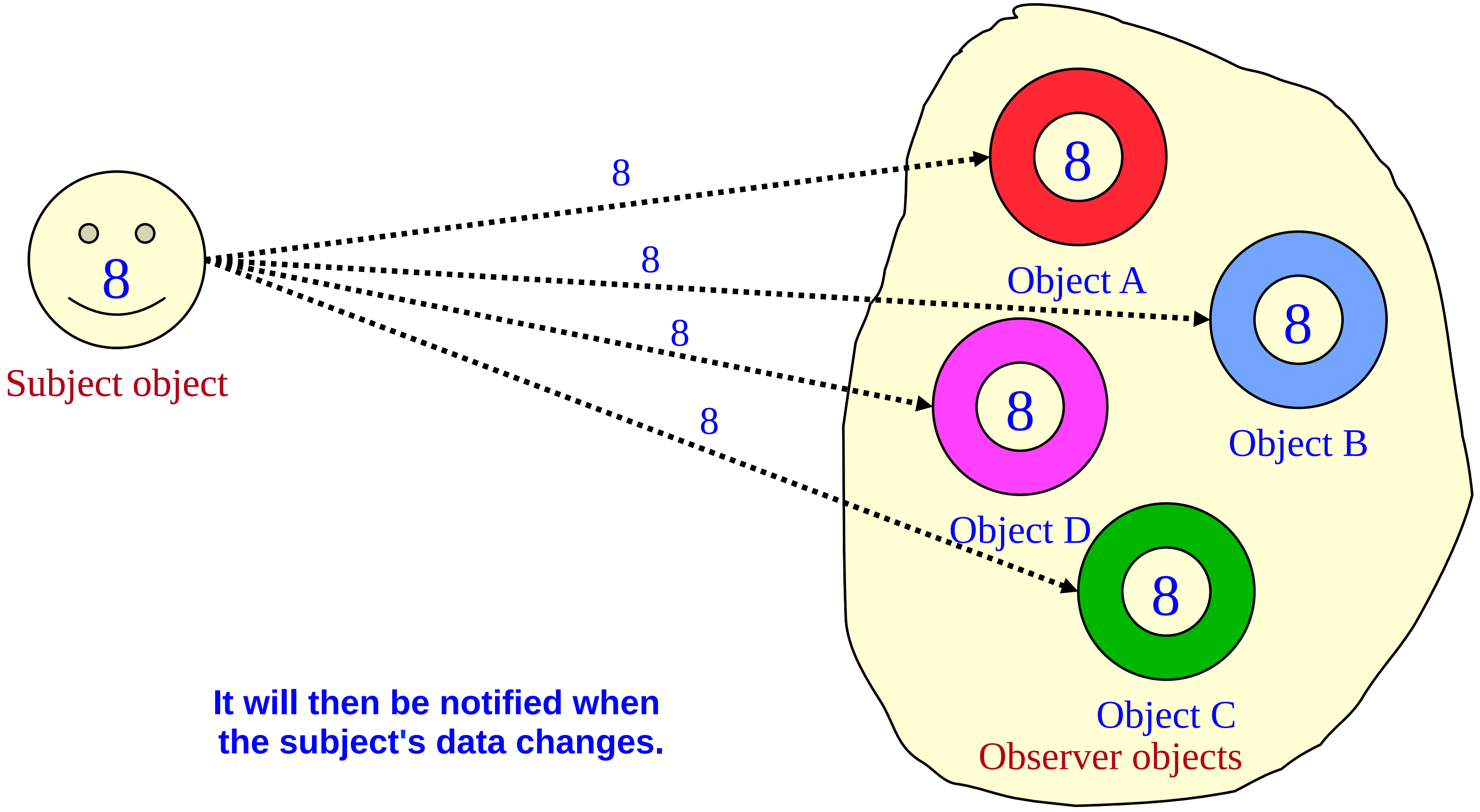
The observers have subscribed (registered with) the subject to receive updates when subject's data changes.
When data in the Subject changes, the Observers are *notified*.
Subject object manages some data.
2
2
2
2
Subject object
2
Object A
2
Object B
The observers *request* from the subject the changes that affect them.
2
Object C
5
Object D
This object is not an observer, so it does not get notified when subject's data changes.
Observer objects

OBSERVER PATTERN: HOW IT WORKS



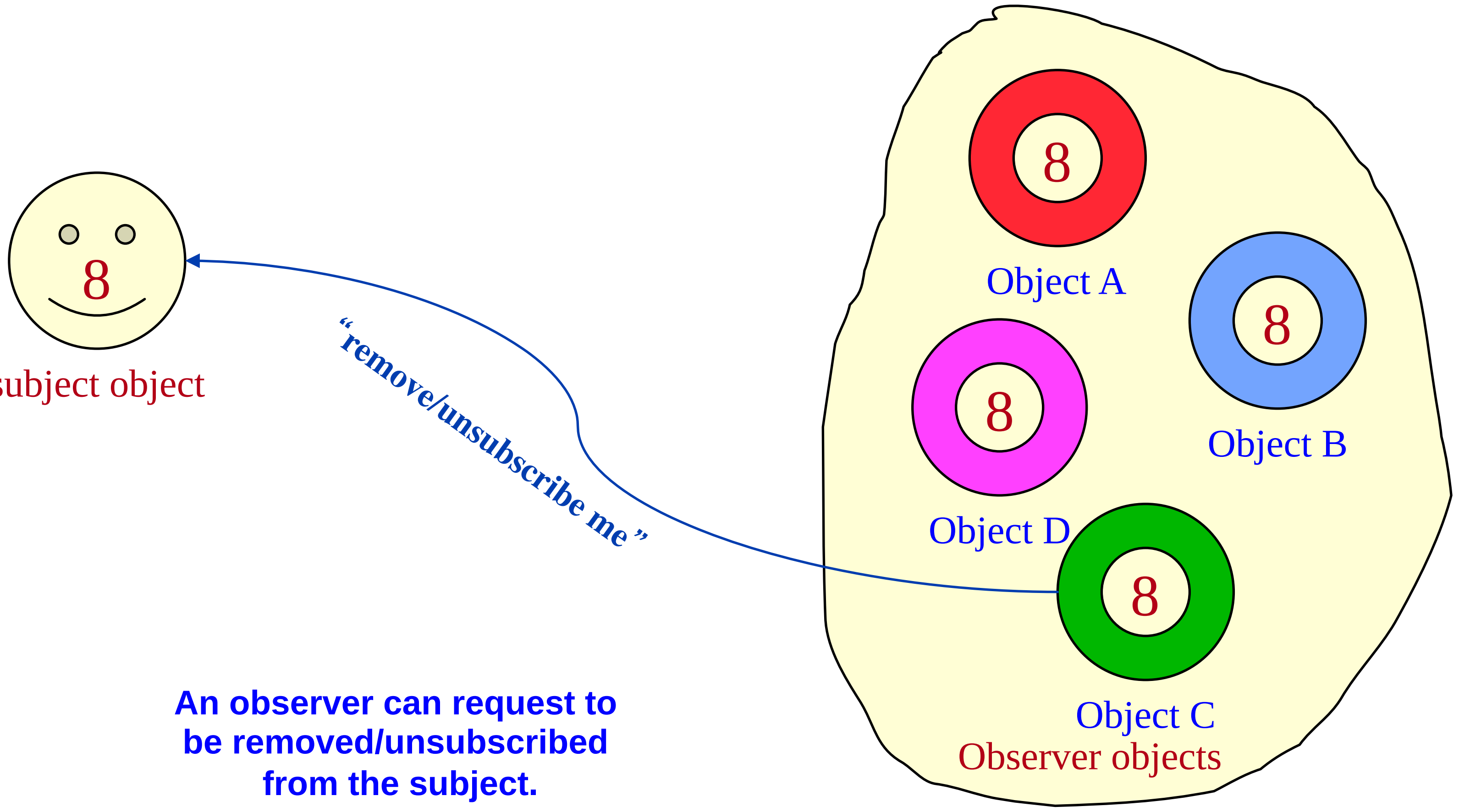
An object can register/ subscribe with the subject.

OBSERVER PATTERN: HOW IT WORKS



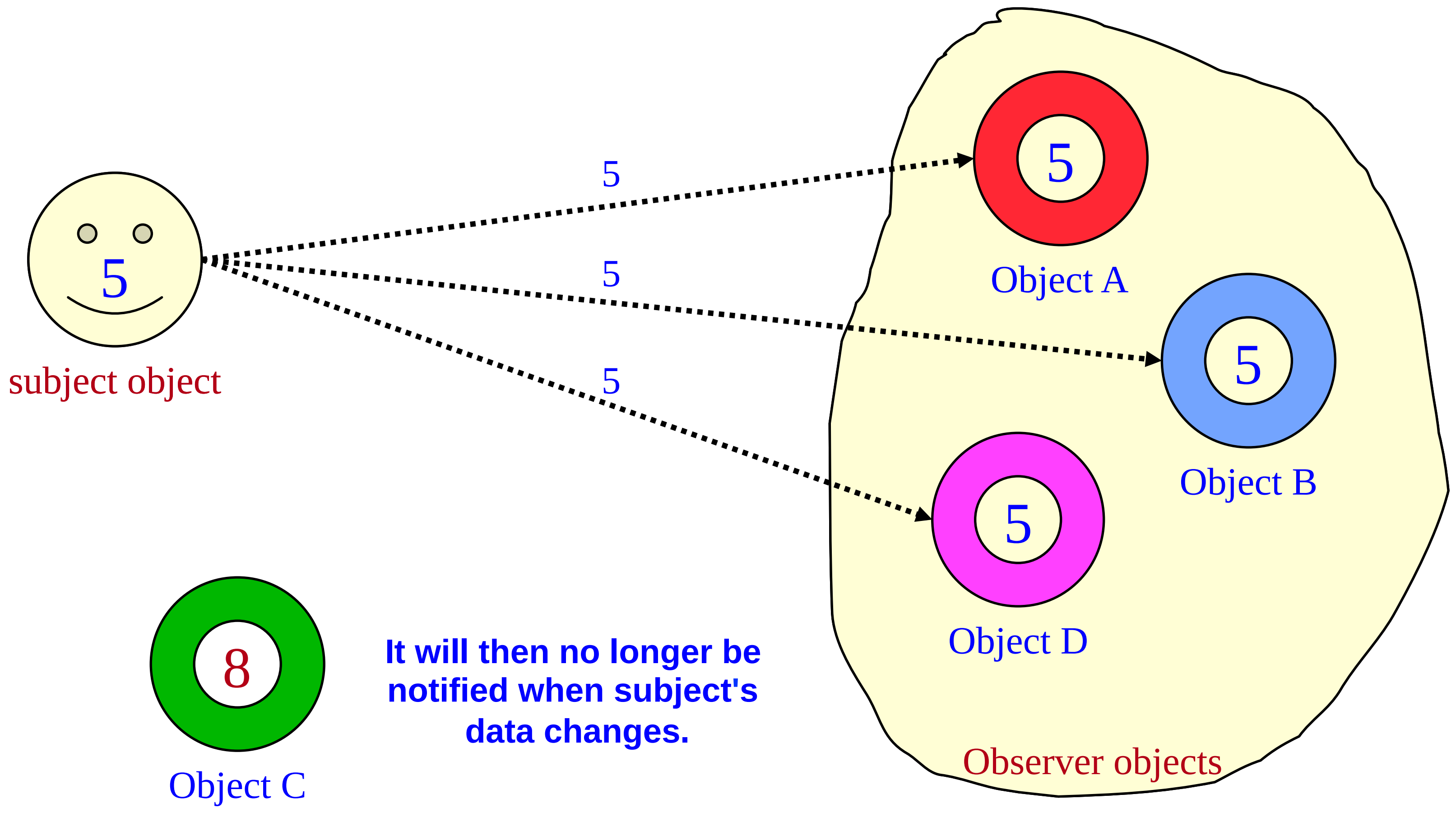
It will then be notified when the subject's data changes.

OBSERVER PATTERN: HOW IT WORKS



An observer can request to be removed/unsubscribed from the subject.

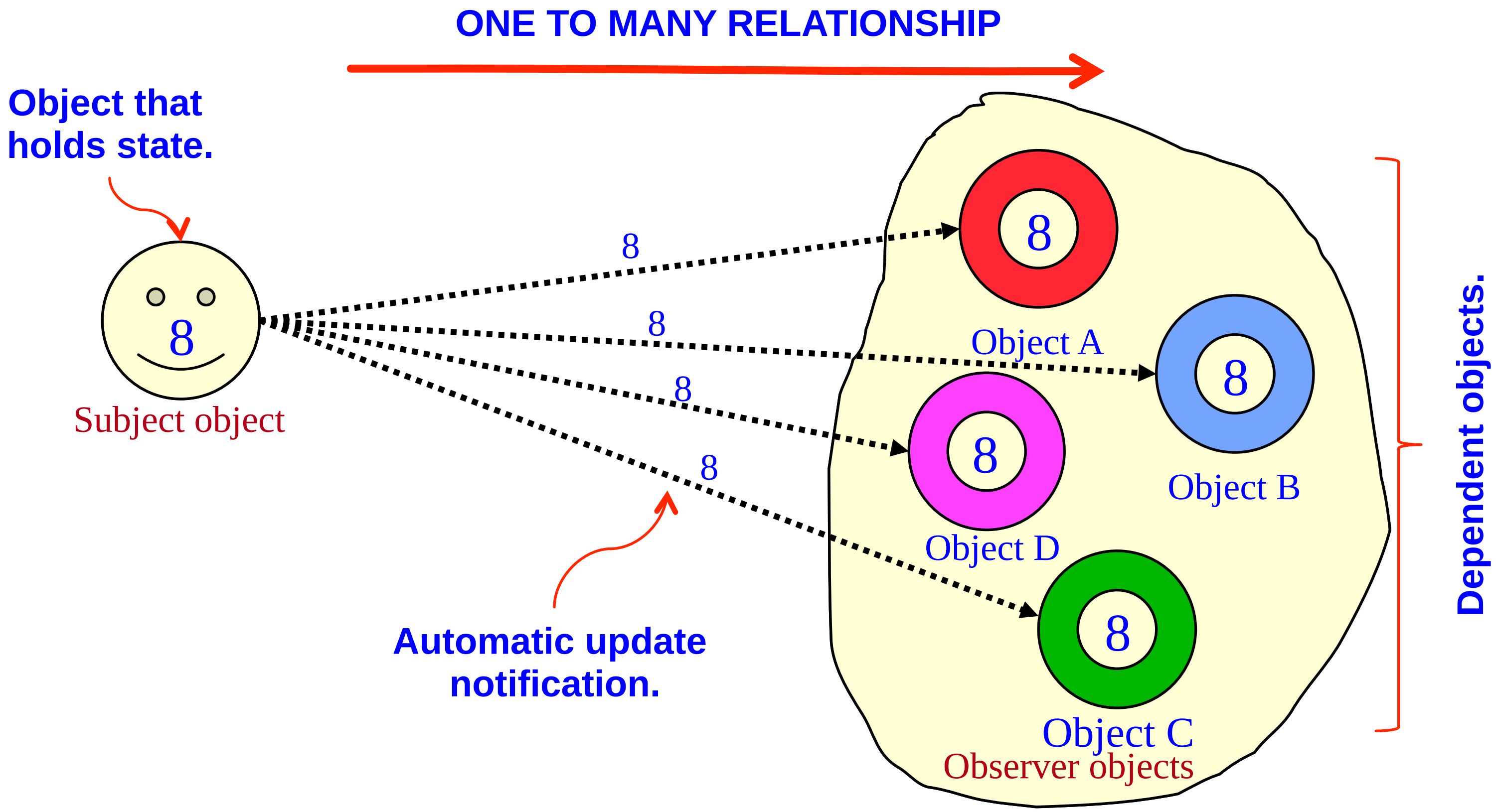
OBSERVER PATTERN: HOW IT WORKS



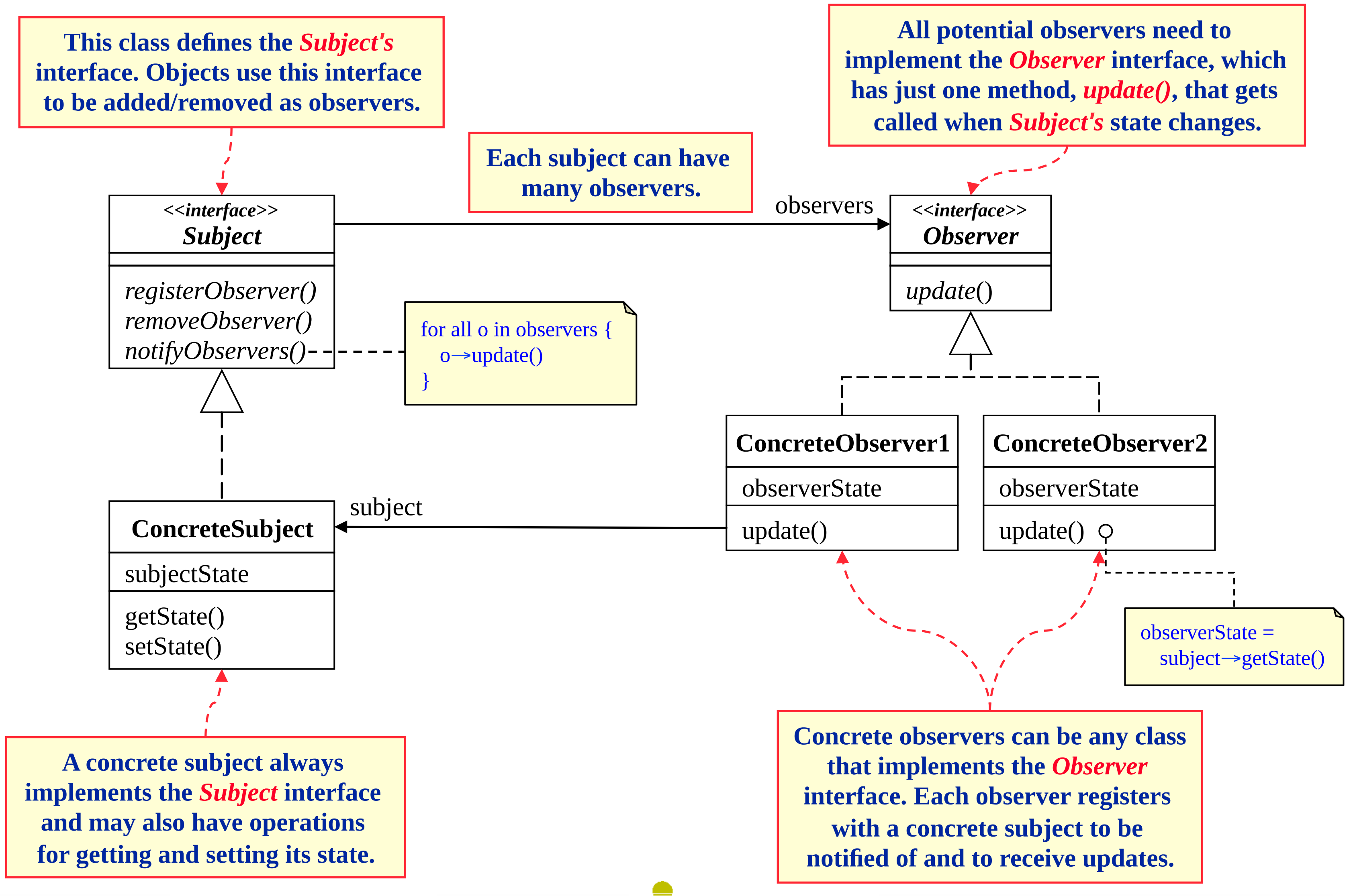
It will then no longer be notified when subject's data changes.

OBSERVER PATTERN: DEFINITION

The **OBSERVER** pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



OBSERVER PATTERN: CLASS DIAGRAM


This class defines the *Subject's* interface. Objects use this interface to be added/removed as observers.
Each subject can have many observers.
All potential observers need to implement the *Observer* interface, which has just one method, *update()*, that gets called when *Subject's* state changes.
<<interface>> Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>> Observer
update()
for all o in observers { o→update() }
ConcreteObserver1
observerState
update()
ConcreteObserver2
observerState
update()
ConcreteSubject
subjectState
getState()
setState()
subject
observerState = subject→getState()
A concrete subject always implements the *Subject* interface and may also have operations for getting and setting its state.
Concrete observers can be any class that implements the *Observer* interface. Each observer registers with a concrete subject to be notified of and to receive updates.

OBSERVER PATTERN: DESCRIPTION

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Applicability (use when …)

- An abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- A change to one object requires changing others and you do not know how many objects need to be changed.
- An object should be able to notify other objects without making assumptions about who those objects are.

OBSERVER PATTERN: THE POWER OF LOOSE COUPLING

- When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

- The **Observer Pattern** provides an object design where subjects and observers are loosely coupled.

WHY?

DESIGN PRINCIPLE

Strive for loosely coupled designs between objects that interact.

Loosely coupled designs minimize the interdependency between objects.

OBSERVER PATTERN: OBSERVATIONS AND ISSUES

- **Support for broadcast communication is needed**
  - No need for ConcreteSubject to worry about the number of observing objects.
  - Observing objects can change at runtime.

- **Event-notification protocol is needed**
  - What events should the subject announce?
  - Should every event be announced to every observer?
  - Should the subject define different kinds of events and allow the observers to subscribe selectively?

DESIGN PRINCIPLE CHALLENGE

How does the observer pattern use these principles?

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

Design Principle

Program to an interface, not an implementation.

Design Principle

Favour composition over inheritance.

BOB'S HOUSE OF THE FUTURE

- Bob has a Java-enabled auto-house, thanks to the good folks at **HouseOfTheFuture**.
- All of his appliances are designed to make his life easier.
- When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing.
- Even though life is good for Bob, he and other clients are always asking for lots of new features:
  - No coffee on the weekends ...
  - Turn off the sprinkler 15 minutes before a shower is scheduled ...
  - Set the alarm early on trash days ...

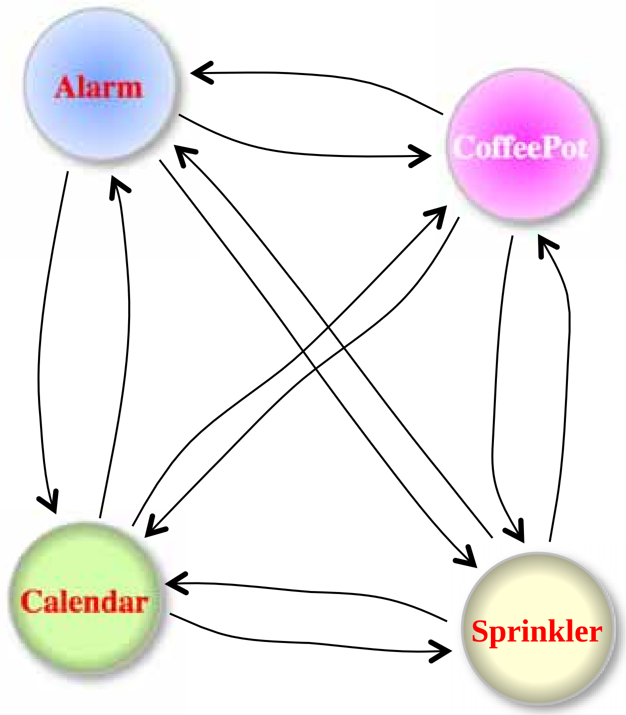
BOB'S HOUSE OF THE FUTURE

Alarm

```
onEvent() {
  checkCalendar()
  checkSprinkler()
  startCoffeePot()
  // do more stuff
}
```

CoffeePot

```
onEvent() {
  checkCalendar()
  checkAlarm()
  // do more stuff
}
```



Calendar

```
onEvent() {
  checkDayOfWeek()
  doSprinkler()
  doCoffeePot()
  doAlarm()
  // do more stuff
}
```

Sprinkler

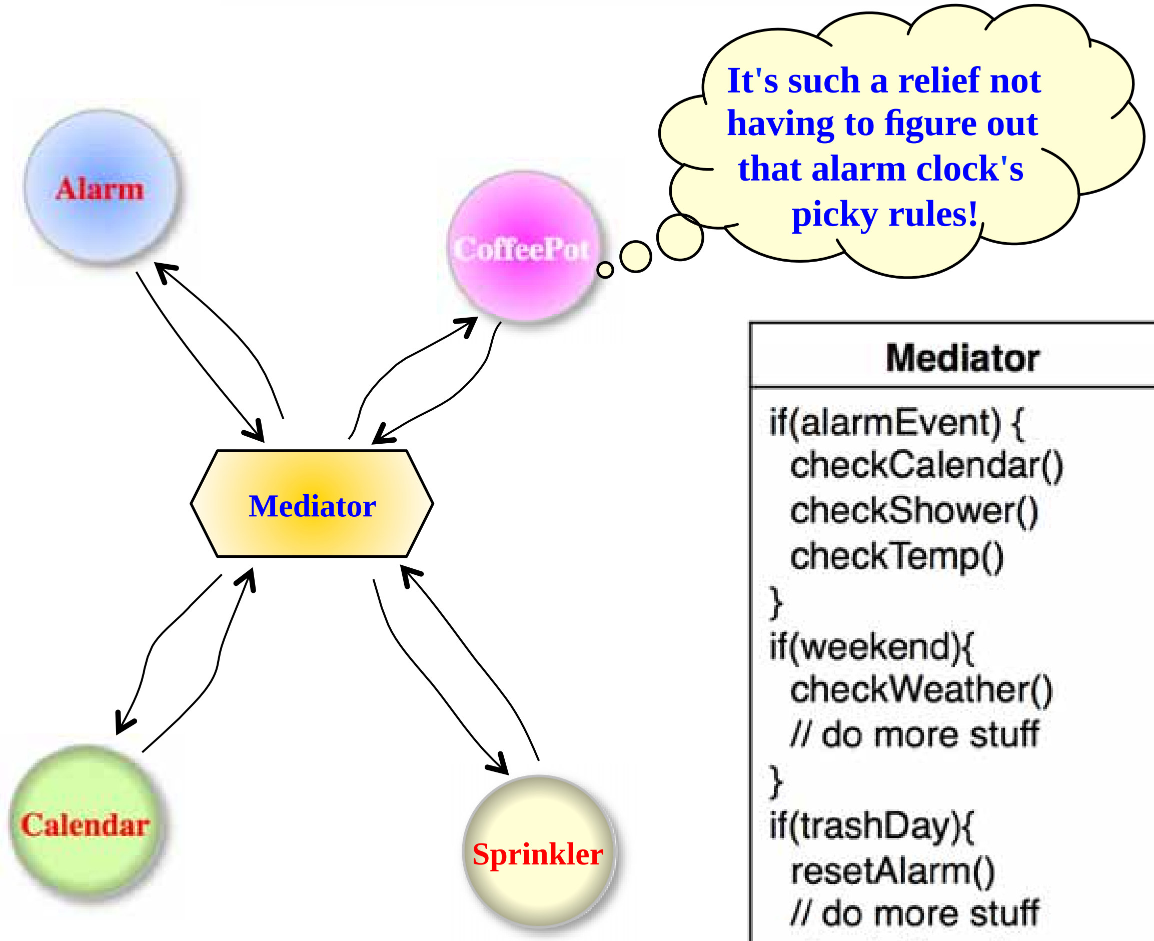
```
onEvent() {
  checkCalendar()
  checkShower()
  checkTemp()
  checkWeather()
  // do more stuff
}
```

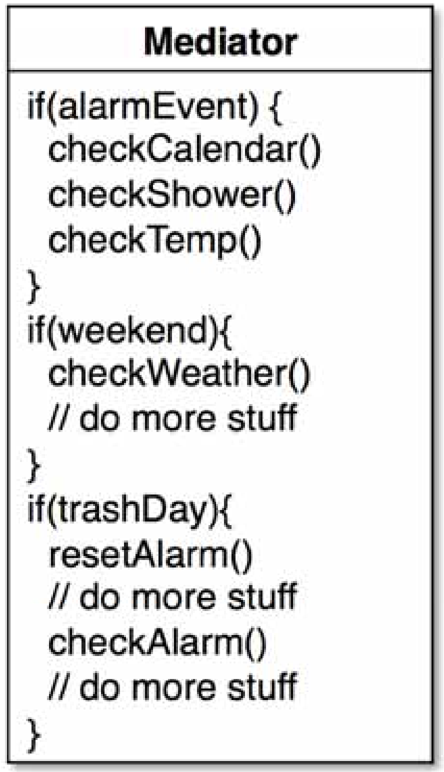
BOB'S HOUSE OF THE FUTURE

HouseOfTheFuture's dilemma

- It's getting really hard to keep track of which rules reside in which objects.
- It's hard to keep track of how the various objects should relate to each other.
- Adding new objects is challenging.

MEDIATOR PATTERN


Alarm
CoffeePot
It's such a relief not having to figure out that alarm clock's picky rules!
Mediator
Calendar
Sprinkler



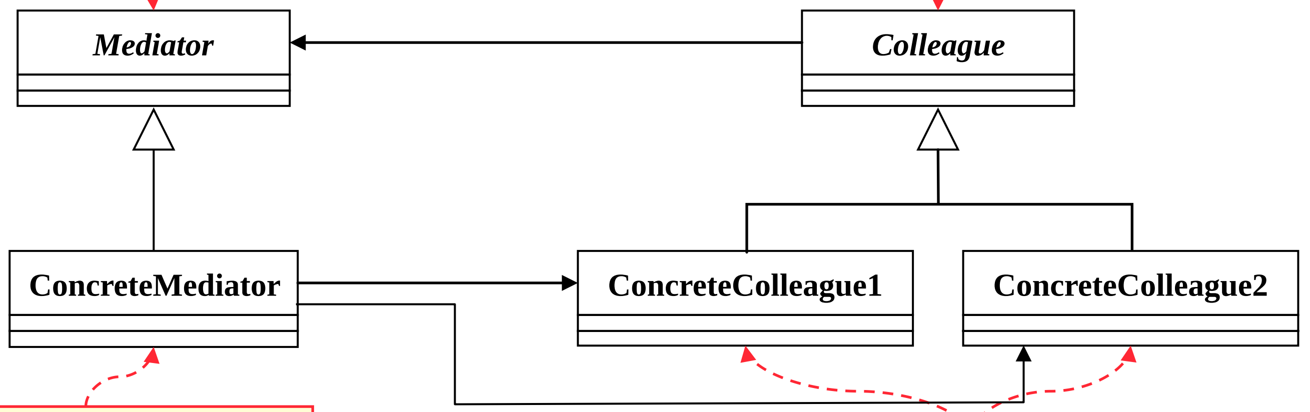
Mediator

```
if(alarmEvent) {
  checkCalendar()
  checkShower()
  checkTemp()
}
if(weekend){
  checkWeather()
  // do more stuff
}
if(trashDay){
  resetAlarm()
  // do more stuff
  checkAlarm()
  // do more stuff
}
```

MEDIATOR PATTERN: CLASS DIAGRAM

The *Mediator* class defines an interface for communicating with *Colleague* objects.

The *Colleague* class defines the interface for *ConcreteColleague* classes.



ConcreteMediator knows the *ConcreteColleague* classes and keeps a reference to each *ConcreteColleague* object.

Each *ConcreteColleague* knows its *ConcreteMediator* object and communicates with the *ConcreteMediator* whenever it would have communicated with another *ConcreteColleague*.

The MEDIATOR pattern is commonly used to coordinate related GUI components.

MEDIATOR PATTERN

Intent

To define an object that encapsulates how a set of objects interact.

To avoid having those objects explicitly know about each other.

Applicability (use when …)

- A set of objects communicate in well-defined, but complex ways.
- Reuse of an object is difficult because it refers to and communicates with many other objects.
- Behavior is distributed between several classes and should be customizable without subclassing.

MEDIATOR PATTERN

Advantages

- **Decoupled Colleagues:** The colleague classes are totally decoupled. Adding a new colleague class is very easy.
- **Comprehension:** Simplifies understanding and maintenance since the mediator centralizes and encapsulates the control logic.
- **Simplified Object Protocols:** The colleague objects need to communicate only with the mediator object (reduces communication from many-to-many to one-to-many).
- **Limits Subclassing:** Only need to extend the mediator class when the logic needs to be extended.

Disadvantages

- **Complexity:** The Mediator object can become overly complex.

PROXY PATTERN

Problem

Sometimes we want to defer the full cost of creating and initializing an object until we actually need to use it.

Example

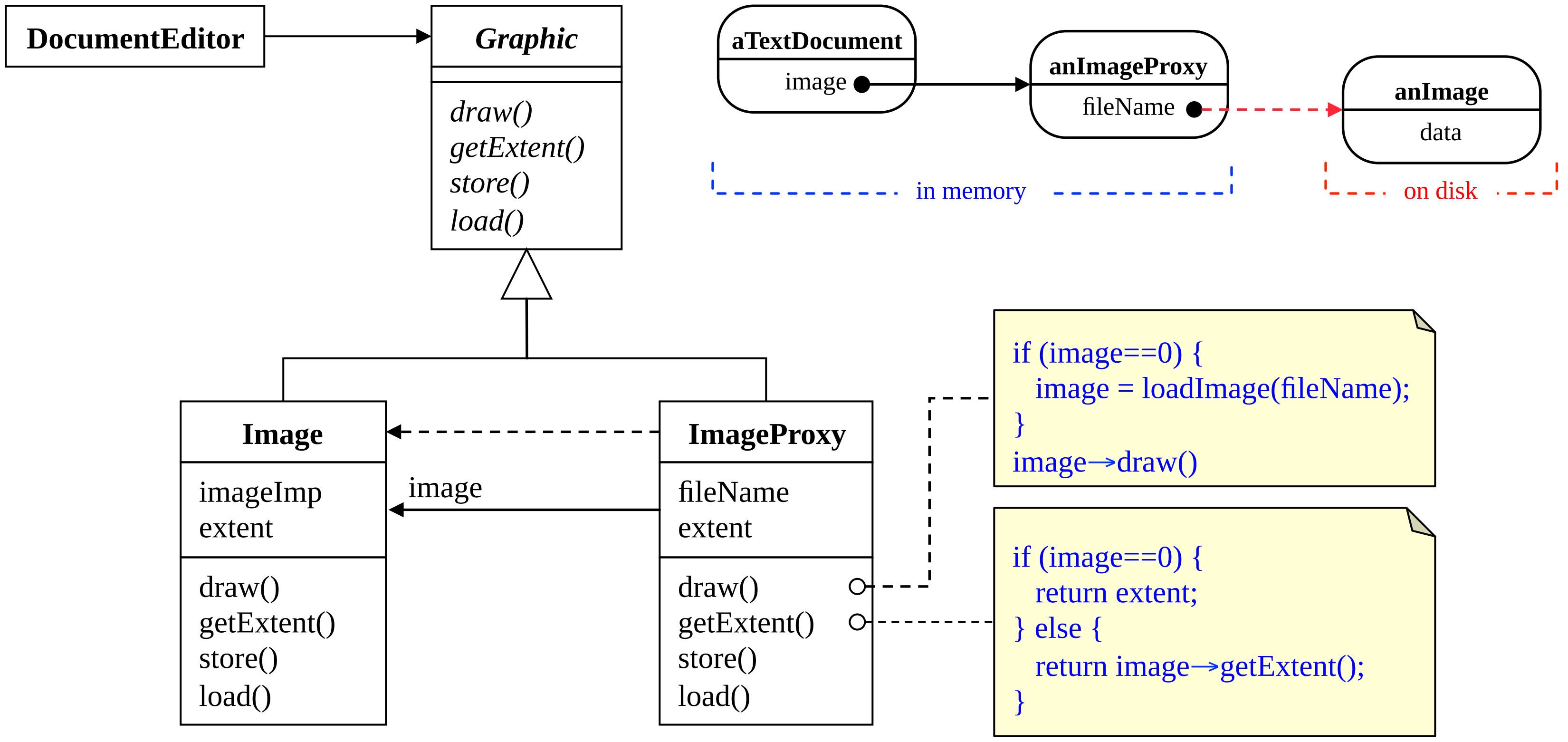
- In a document editor that can embed graphical objects in a document, objects like large raster images can be expensive to create.
- However, opening a document should be fast. Thus, we should avoid creating expensive objects all at once when the document is opened. Also, usually not all objects are visible at the same time anyway.

☞ **Expensive objects should be *created on demand*.**

PROXY PATTERN: SOLUTION

Use an *image proxy* that acts as a stand-in for the real image.

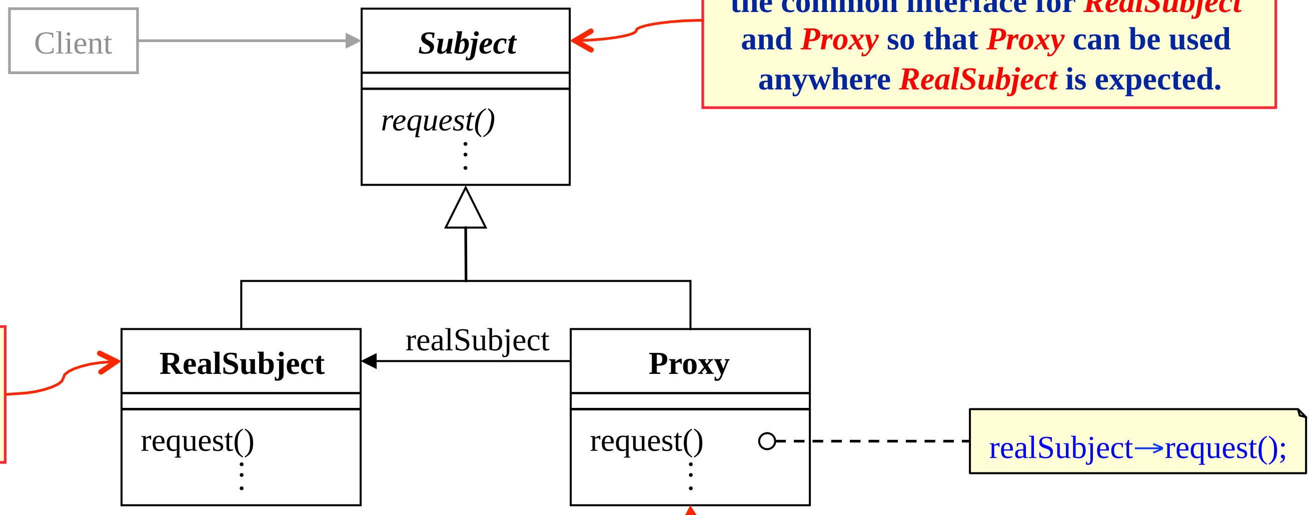
The proxy acts just like the image, instantiates it when needed and forwards request to it after creating it.



```
if (image==0) {
  image = loadImage(fileName);
}
image→draw()
```

```
if (image==0) {
  return extent;
} else {
  return image→getExtent();
}
```

PROXY PATTERN: CLASS DIAGRAM



Subject is an abstract class that defines the common interface for *RealSubject* and *Proxy* so that *Proxy* can be used anywhere *RealSubject* is expected.

RealSubject defines the real object that *Proxy* represents.

The *Proxy* class maintains a reference that lets it access *RealSubject*. *Proxy* may refer to *Subject* if the *RealSubject* and *Subject* interfaces are the same. *Proxy* provides an interface identical to *Subject*'s so that *Proxy* can be substituted for *RealSubject*. *Proxy* controls access to *RealSubject* and may be responsible for creating and deleting it.

PROXY PATTERN

Intent

Provide a surrogate or placeholder for another object to control access to it.

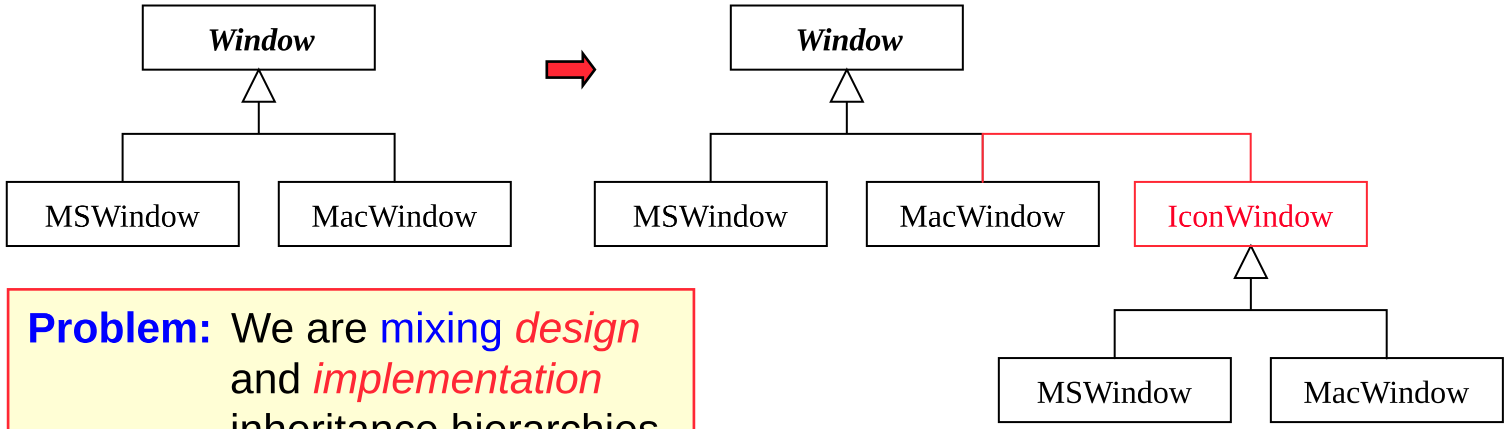
Applicability (use when …)

- There is a need for a more versatile or sophisticated reference to an object than a simple pointer.
- You want to:
  - distribute objects to different object spaces or across networks (remote proxy).
  - create expensive objects on demand (virtual proxy).
  - control access to the original object (protection proxy).
  - allow additional housekeeping tasks to be done when an object is accessed (smart reference).

BRIDGE PATTERN

Problem: When an abstraction can have several possible implementations, inheritance is usually used to handle this. However, this binds an implementation to an abstraction permanently making it difficult to modify, extend and reuse abstractions and implementations *independently*.

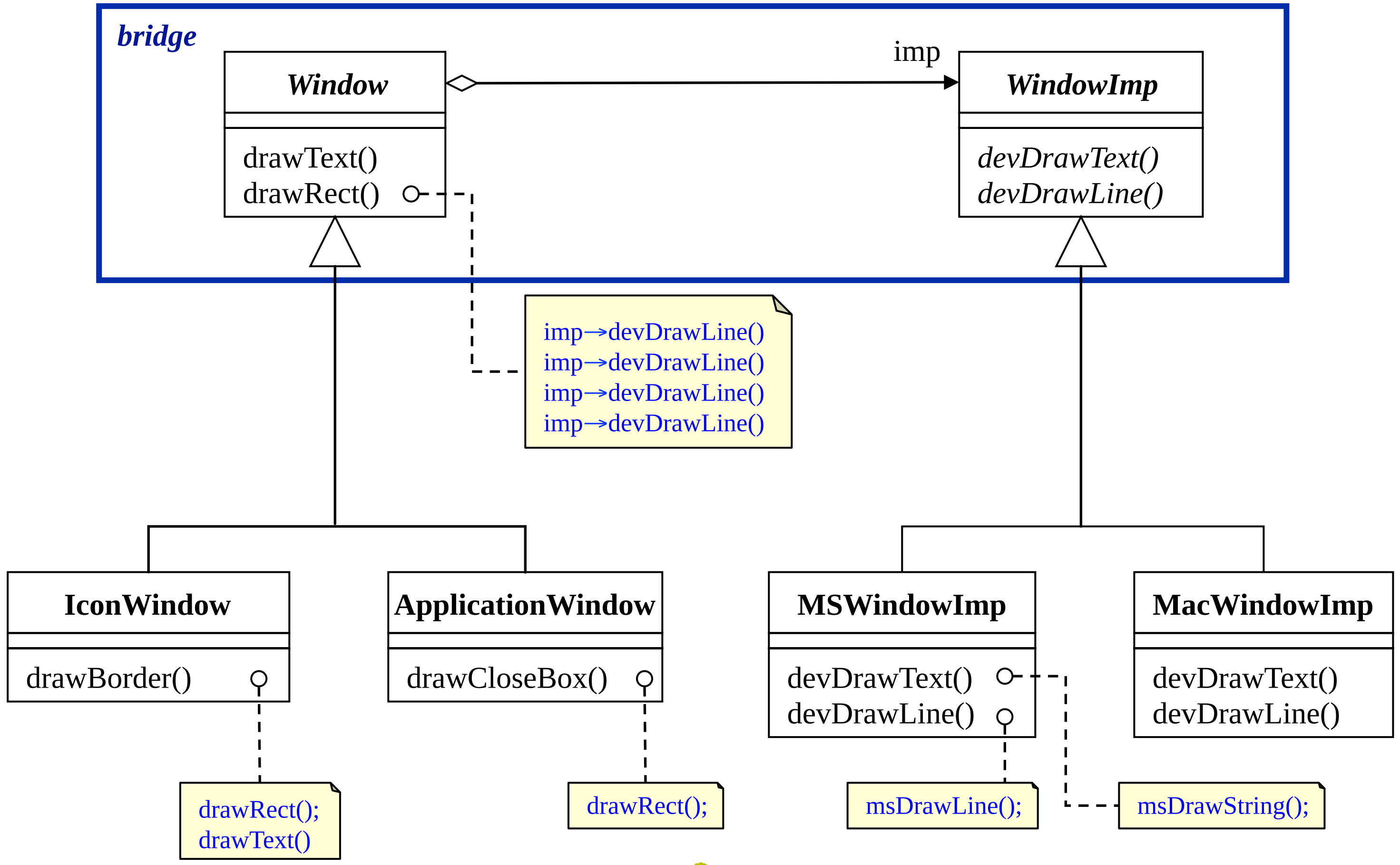
E.g., implementation of a portable Window abstraction in a user interface toolkit.



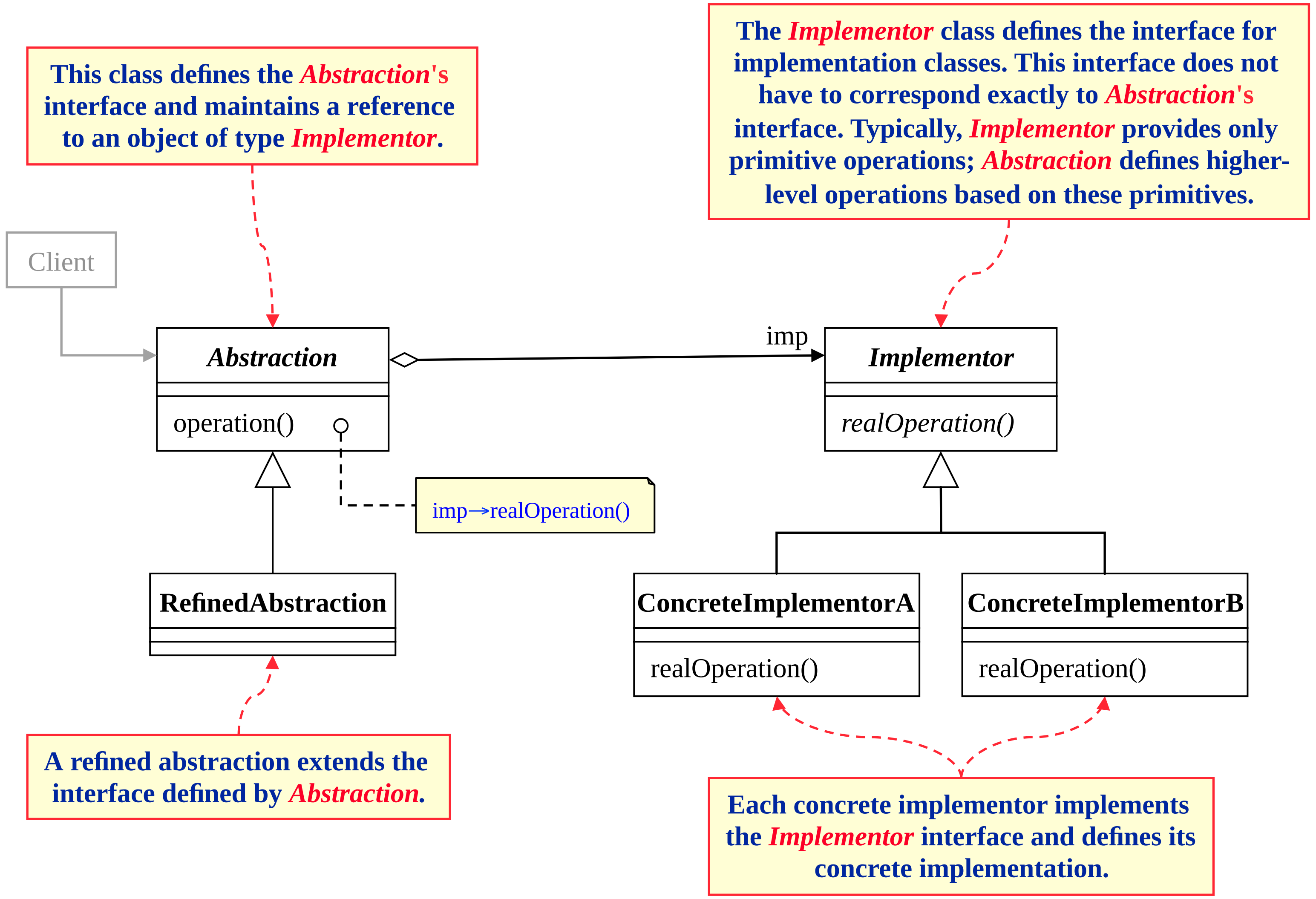
Problem: We are mixing *design* and *implementation* inheritance hierarchies.

BRIDGE PATTERN: SOLUTION

Separate the window abstraction and implementation class hierarchies and connect them using a *bridge*.



BRIDGE PATTERN: CLASS DIAGRAM



This class defines the *Abstraction*'s interface and maintains a reference to an object of type *Implementor*.
The *Implementor* class defines the interface for implementation classes. This interface does not have to correspond exactly to *Abstraction*'s interface. Typically, *Implementor* provides only primitive operations; *Abstraction* defines higher-level operations based on these primitives.
Client
Abstraction
operation()
imp
Implementor
realOperation()
imp→realOperation()
RefinedAbstraction
ConcreteImplementorA
realOperation()
ConcreteImplementorB
realOperation()
A refined abstraction extends the interface defined by *Abstraction*.
Each concrete implementor implements the *Implementor* interface and defines its concrete implementation.

BRIDGE PATTERN

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Applicability (use when …)

- You want to avoid a permanent binding between an abstraction and its implementation.
- Both the abstractions and their implementations should be extensible by subclassing.
- Changes in the implementation of an abstraction should have no impact on clients.
- You want to hide the implementation of an abstraction completely from clients.
- You have a proliferation of classes (as in the interface toolkit example).
- You want to share an implementation among multiple objects and hide this fact from clients.

SINGLETON PATTERN

Problem: It is important for some classes to have ***exactly one instance***. How to ensure that a class has only one instance and that the instance is easily accessible?

A global variable can make an object accessible, but how to prohibit instantiation of multiple instances?

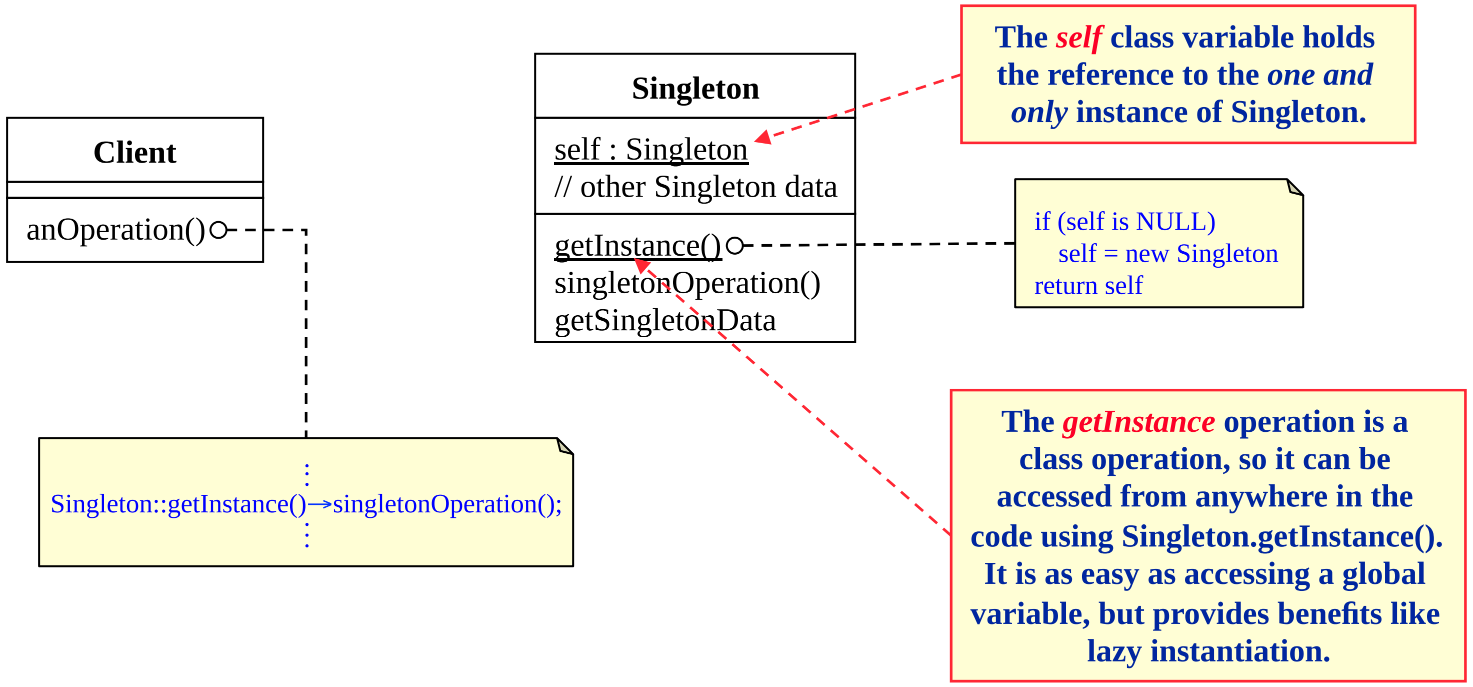
Why would we need only one instance of an object?

- Thread pools
- Caches
- Dialog boxes
- Objects used for logging
- Objects that handle graphics cards?
- System.out?

SINGLETON PATTERN: CLASS DIAGRAM

Make the singleton class itself responsible for

- keeping track of its sole instance,
- ensuring that no other instance can be created, and
- providing a way to access the instance.


Client
anOperation()
Singleton
self : Singleton
// other Singleton data
getInstance()
singletonOperation()
getSingletonData
The *self* class variable holds the reference to the *one and only* instance of Singleton.
if (self is NULL)
self = new Singleton
return self
Singleton::getInstance()→singletonOperation();
The *getInstance* operation is a class operation, so it can be accessed from anywhere in the code using Singleton.getInstance(). It is as easy as accessing a global variable, but provides benefits like lazy instantiation.

SINGLETON PATTERN

Intent

Ensure a class only has one instance and provide a global point of access to it.

Applicability (use when …)

- There must be exactly one instance of a class and it must be accessible to clients from a well-known point.
- The sole instance should be extensible by subclassing and clients should be able to use an extended instance without modifying their code.

FACTORY PATTERN: MAKING PIZZA

```
Pizza orderPizza() {
      Pizza pizza = new Pizza();

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

We want to be able to make different types of pizza.

So, for flexibility, we really want this to be an abstract class or interface, but we cannot directly instantiate either of those.

FACTORY PATTERN: MAKING PIZZA

```
Pizza orderPizza(string type) {
      Pizza pizza;

      if (type.equals("cheese")) {
        pizza = new CheesePizza();
      } else if (type.equals("greek") {
        pizza = new GreekPizza();
      } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
      }

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

We are now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce and add the toppings & cheese), then we bake it, cut it and box it!

Each Pizza subtype (CheesePizza, GreekPizza, etc. knows how to prepare itself.

THE OPEN-CLOSED PRINCIPLE

<u>Design Principle</u>

Classes should be open for extension, but closed for modification.

We should allow a design to incorporate new behaviour without modifying existing code.

How best to achieve it?

FACTORY PATTERN: ADDING MORE PIZZA TYPES

This code is NOT closed for modification. If the Pizza Shop always changes its pizza offering, we have to get into this code and modify it.

```
Pizza orderPizza(string type) {
      Pizza pizza;

      if (type.equals(“cheese”)) {
        pizza = new CheesePizza();
      } ~~else if (type.equals(“greek”)~~ {
        ~~pizza = new GreekPizza();~~
      } else if (type.equals(“pepperoni”) {
        pizza = new PepperoniPizza();
      } else if (type.equals(“clam”) {
        pizza = new ClamPizza();
      } else if (type.equals(“veggie”) {
        pizza = new VeggiePizza();
      }

      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

This is what varies. As the pizza selection changes over time, you will have to modify this code over and over.

This is what we expect to stay the same. For the most part, preparing, baking and packaging a pizza has remained the same for years and years. So, we do not expect this code to change, just the pizza it operates on.

FACTORY PATTERN: ENCAPSULATING OBJECT CREATION

```
if (type.equals(“cheese”)) {
      pizza = new CheesePizza();
   } else if (type.equals(“pepperoni”) {
      pizza = new PepperoniPizza();
   } else if (type.equals(“clam”) {
      pizza = new ClamPizza();
   } else if (type.equals(“veggie”) {
      pizza = new VeggiePizza();
}
```

```
Pizza orderPizza(string type) {
      Pizza pizza;



      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
}
```

First we pull the object creation code out of the orderPizza operation.

What’s going to go here?

Then we place that code in an object that is only going to create pizzas. If any other object needs a pizza created, this is the object to come to.

FACTORY PATTERN: A SIMPLE PIZZA FACTORY

Here is our new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

First we define a createPizza() operation in the factory. This is the operation all clients will use to instantiate new objects.

```
public class SimplePizzaFactory {
    public Pizza createPizza(string type) {
        Pizza pizza = null;

        if (type.equals(“cheese”)) {
          pizza = new CheesePizza();
        } else if (type.equals(“pepperoni”) {
          pizza = new PepperoniPizza();
        } else if (type.equals(“clam”) {
          pizza = new ClamPizza();
        } else if (type.equals(“veggie”) {
          pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

Here is the code we pulled out of the orderPizza() operation.

This code is still parameterized by the type of the pizza, just like our original orderPizza() operation was.

FACTORY PATTERN: REWORKING THE PIZZA STORE

Now we give PizzaStore a reference to a SimplePizzaFactory.

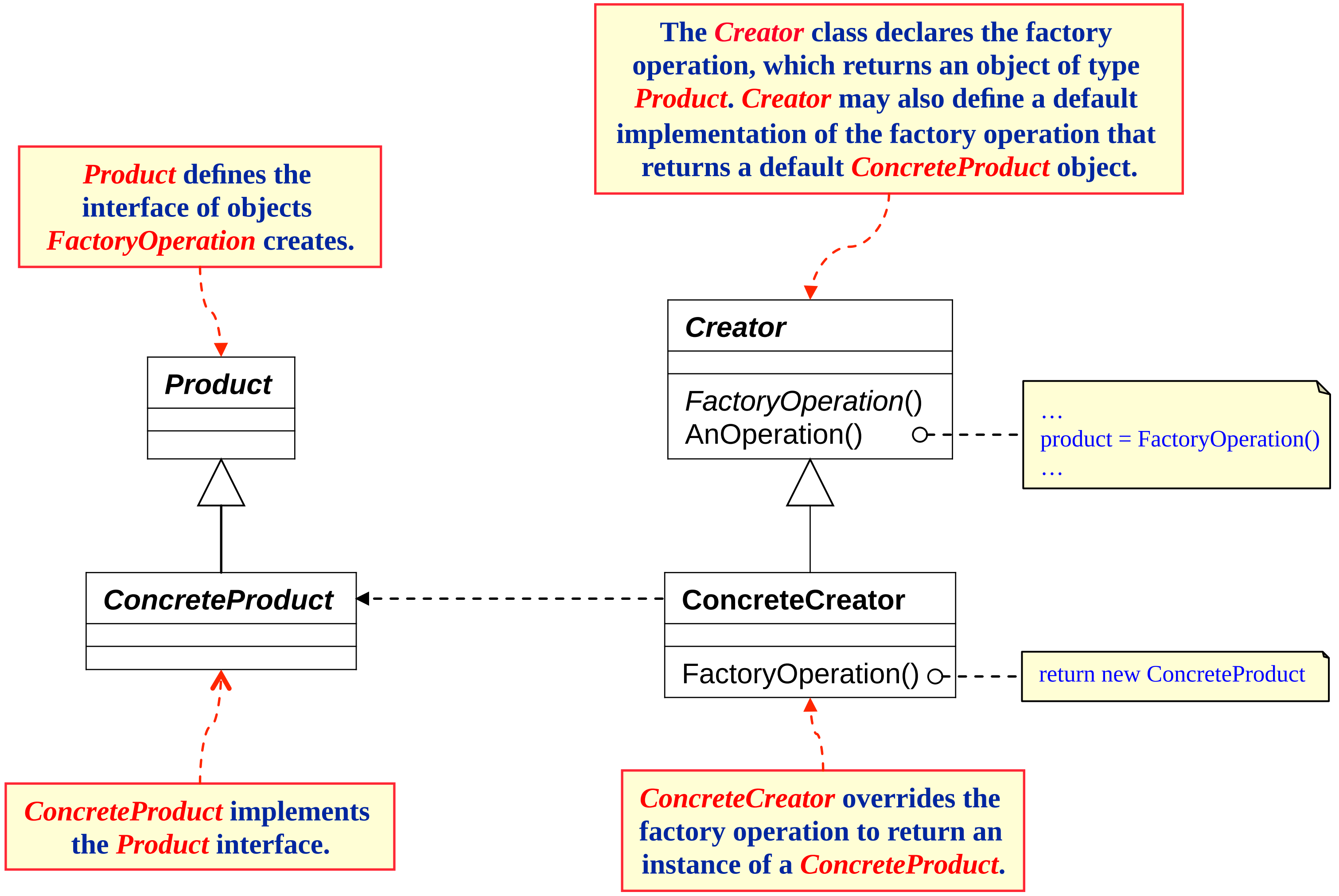
```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory
    }

    public Pizza orderPizza(string type) {
        Pizza pizza

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    // other operations here
}
```

PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() operation uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we have replaced the new operator with a create operation on the factory object. No more concrete instantiation here!

FACTORY PATTERN: CLASS DIAGRAM


Product defines the interface of objects *FactoryOperation* creates.
The *Creator* class declares the factory operation, which returns an object of type *Product*. *Creator* may also define a default implementation of the factory operation that returns a default *ConcreteProduct* object.
Product
Creator
FactoryOperation()
AnOperation()
…
product = FactoryOperation()
…
ConcreteProduct
ConcreteCreator
FactoryOperation()
return new ConcreteProduct
ConcreteProduct implements the *Product* interface.
ConcreteCreator overrides the factory operation to return an instance of a *ConcreteProduct*.

FACTORY PATTERN

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory lets a class defer instantiation to subclasses.

Applicability (use when …)

- A class cannot anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses and you want to localize the knowledge of which helper subclass is the delegate.

WHEN TO USE DESIGN PATTERNS

1. **Solutions to *problems that recur* with variations**
   - No need for reuse if the problem only arises in one context!

2. **Solutions that *require several steps***
   - Not all problems need all steps.
   - Design patterns can be overkill if the solution is a simple linear set of instructions.

3. **Solutions where the solver is *more interested in the existence of the solution* than its complete derivation**
   - Design patterns leave out too much to be useful to someone who really wants to understand.
   - **BUT**, they can be a temporary bridge …

BENEFITS OF DESIGN PATTERNS

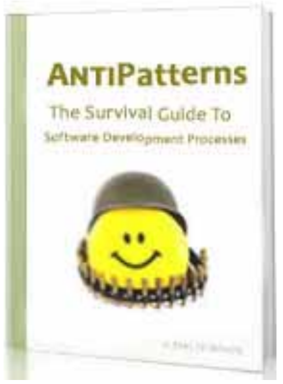
- Design patterns *enable large-scale reuse* of software designs.
  - They also help document systems to enhance understanding.
- Design patterns *explicitly capture expert knowledge and design tradeoffs* and make this expertise more widely available.
- Design patterns *help improve developer communication*.
  - Design pattern names form a vocabulary among the developers.
- Design patterns *help ease the transition to object-oriented technology*.

DRAWBACKS OF DESIGN PATTERNS

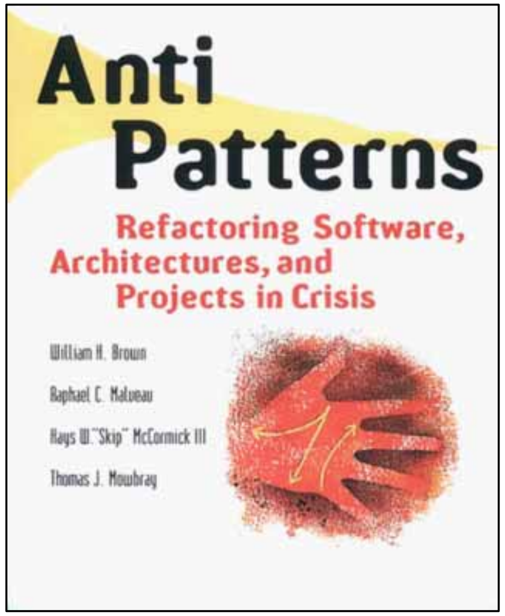
- Design patterns *do not lead to direct code reuse*.
- Design patterns are *deceptively simple*.
- Development teams *may suffer from design pattern overload*.
- Design patterns are *validated by experience and discussion* rather than by automated testing.
- Integrating design patterns into a software development process is a *human-intensive activity*.

ANTI PATTERNS

- An anti pattern tells you what is a **bad solution**.
  - Why a bad solution is attractive.
  - Why the solution is bad (in the long term).
  - What alternative (good solutions) you have.



- There are many types of anti patterns
  - development
  - object-oriented
  - organizational
  - domain specific


Anti Patterns
Refactoring Software, Architectures, and Projects in Crisis
William H. Brown
Raphael C. Malveau
Hays W. "Skip" McCormick III
Thomas J. Mowbray

WELL-KNOWN ANTI PATTERNS

- Anti patterns are all around us. They are often used as tools for social control.

Social Anti Patterns

- Criminal
- Terrorist
- Pervert
- Drug Addict/Pusher
- Heretic/Witch
- Dilbert's Pointy-haired Manager

Software Anti Patterns

- Spaghetti Code
- Stovepipe System
- Analysis Paralysis
- Design By Committee
- God Class
- Mythical Man Month
- Death March Project

DEVELOPMENT ANTI PATTERN: SPAGHETTI CODE

✧ **spa•ghet•ti code** [Slang] *an undocumented piece of software code that cannot be extended or modified without extreme difficulty due to its convoluted structure.*

Un-structured code is a liability

Well structured code is an investment

DEVELOPMENT ANTI PATTERN: SPAGHETTI CODE

Symptoms

- Quick demonstration code (prototype) that becomes operational.
- "Lone Ranger" programmer (*who was that masked man*)?
- Obsolete or scanty documentation.
- **50%** of maintenance is spent on system rediscovery.
- Hesitant Programmer Syndrome
  - More likely to break it than extend it.
  - Easier to just rewrite it.
- Cannot be reused
  - System software and COTS packages can't be upgraded.
  - Performance cannot be optimized.
- User work arounds.

DEVELOPMENT ANTI PATTERN: SPAGHETTI CODE

Object-Oriented Spaghetti Code

- Many object methods with no parameters.
- Suspicious class or global variables.
- Intertwined and unforeseen relationships between objects.
- Procedure-oriented methods, objects with procedure-oriented names.
- OO advantage lost — inheritance cannot be used to extend the system, polymorphism is not effective either.

Bottom Line

The software has reached a point of diminishing returns where the effort involved to maintain existing code exceeds the cost of developing a new “ground up” solution.

DEVELOPMENT ANTI PATTERN: THE BLOB / GOD CLASS

Symptoms

- A single class with many attributes and/or operations (i.e., a class that knows too much or does too much).
- Violates the cohesion (single responsibility) property of OO design.
- A nightmare to maintain (changes or bug fixes introduce new bugs).

DEVELOPMENT ANTI PATTERN: THE BLOB / GOD CLASS

Typical Causes

- Inappropriate requirements specification
- Lack of an object-oriented architecture
- Lack of (any) architecture
- Lack of architecture enforcement
- Too limited intervention

DEVELOPMENT ANTI PATTERN: THE BLOB / GOD CLASS

Consequences

- Lost OO advantage.
- Too complex to reuse or test.
- Unnecessary code
- Expensive to load.

Refactor the code!

SYSTEM ANALYSIS AND DESIGN: RETROSPECTIVE

Architectural design

- Outlines the overall software structure of the system in terms of cohesive, loosely-coupled subsystems.
- Makes use of architectural patterns to organize subsystems.

Analysis Class

- Structures the system by partitioning use cases into boundary, control and entity classes.
- Assigns responsibilities to classes.

Design Class

- Completes the specification of each class (attributes, operations).
- Restructures and optimizes classes.
- Makes use of design patterns to handle commonly occurring design requirements, particularly nonfunctional requirements.

SYSTEM DESIGN: SUMMARY

The Analysis and Design Model contains:

Subsystems, their dependencies and their interfaces.

- ☞ **Implemented by implementation subsystems containing source code, scripts, binaries, executables, etc.**

Analysis classes and their responsibilities.

- ☞ **Localizes changes to boundary, entity or control classes.**

Design classes, their operations, attributes, relationships and implementation requirements.

- ☞ **Implemented by files containing source code.**