# SOFTWARE ENGINEERING

CO3001

## CHAPTER 7 – DESIGN AND IMPLEMENTATION

**WEEK 8, 9**

Adapted from https://iansommerville.com/software-engineering-book/slides/

# TOPICS COVERED

- ✓ Object-oriented design using the UML
- ✓ Design patterns
- ✓ Implementation issues
- ✓ Open source development

# DEVELOPMENT = DESIGN + IMPLEMENTATION

- Software design:
  - identify software components and their relationships, based on a customer's requirements.
- Implementation:
  - realizing the design as a program.

- Software design and implementation activities are invariably inter-leaved.

# OBJECT-ORIENTED DESIGN USING THE UML

# AN OBJECT-ORIENTED DESIGN PROCESS

- ✓ Structured object-oriented design processes involve developing a number of different system models.

- ✓ They require a lot of effort for development and maintenance of these models
  - for small systems, this may not be cost-effective.
  - for large systems developed by different groups, design models are an important communication mechanism.

# OBJECT-ORIENTED DESIGN PROCESS

- ✓ There are a variety of different object-oriented design processes that depend on the organization using the process.

- ✓ Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.

# SYSTEM CONTEXT AND INTERACTIONS

- ✓ Understanding the relationships between the developing software and its external environment
  - how to provide the required system functionality and how to structure the system to communicate with its environment.

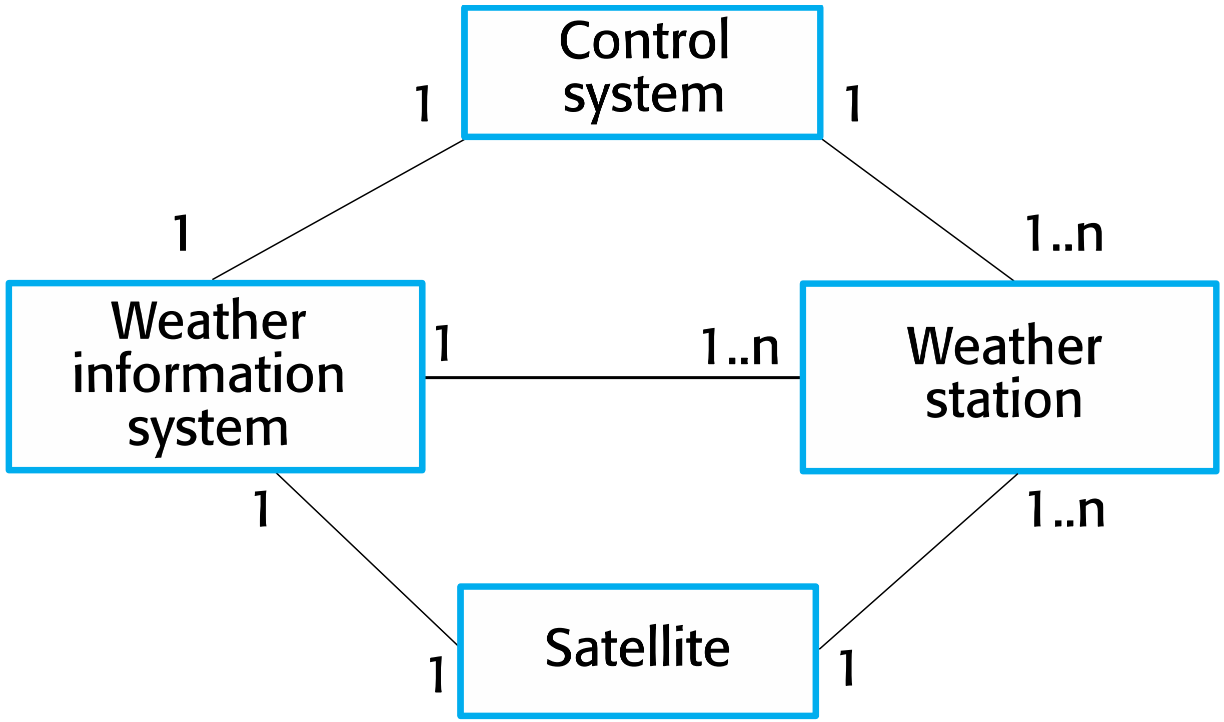
- ✓ Understanding of the context also lets you establish the boundaries of the system.
  - System boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

# CONTEXT AND INTERACTION MODELS

- ✓ A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.

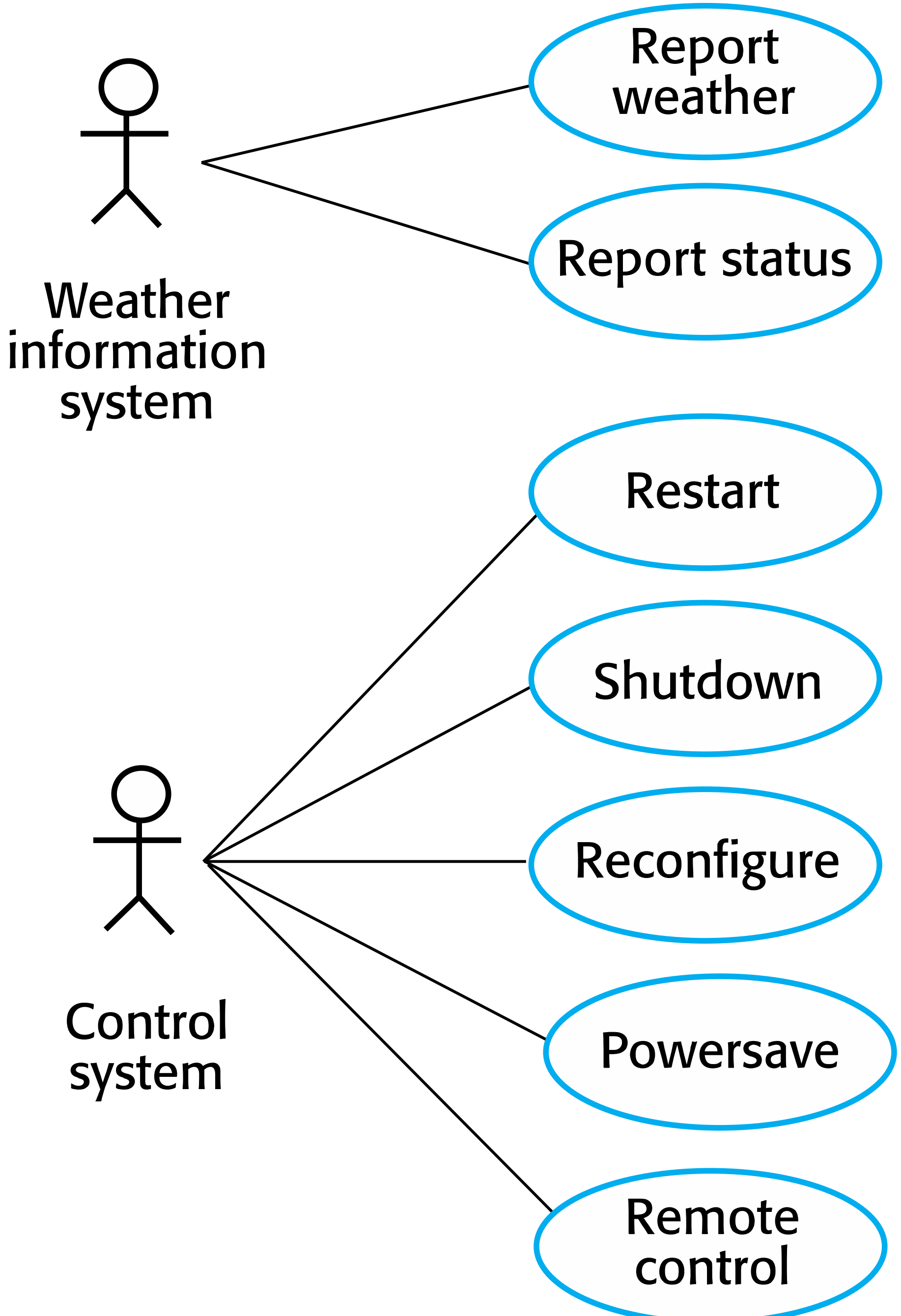
- ✓ An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

# SYSTEM CONTEXT FOR THE WEATHER STATION




In this example, the system about the weather station is a system of weather stations as a weather information system, an onboard satellite system, and a control system. The cardinality information on the link shows that there is one control system but several weather stations, one satellite, and one general weather information system.

# WEATHER STATION USE CASES



# USE CASE DESCRIPTION - REPORT WEATHER

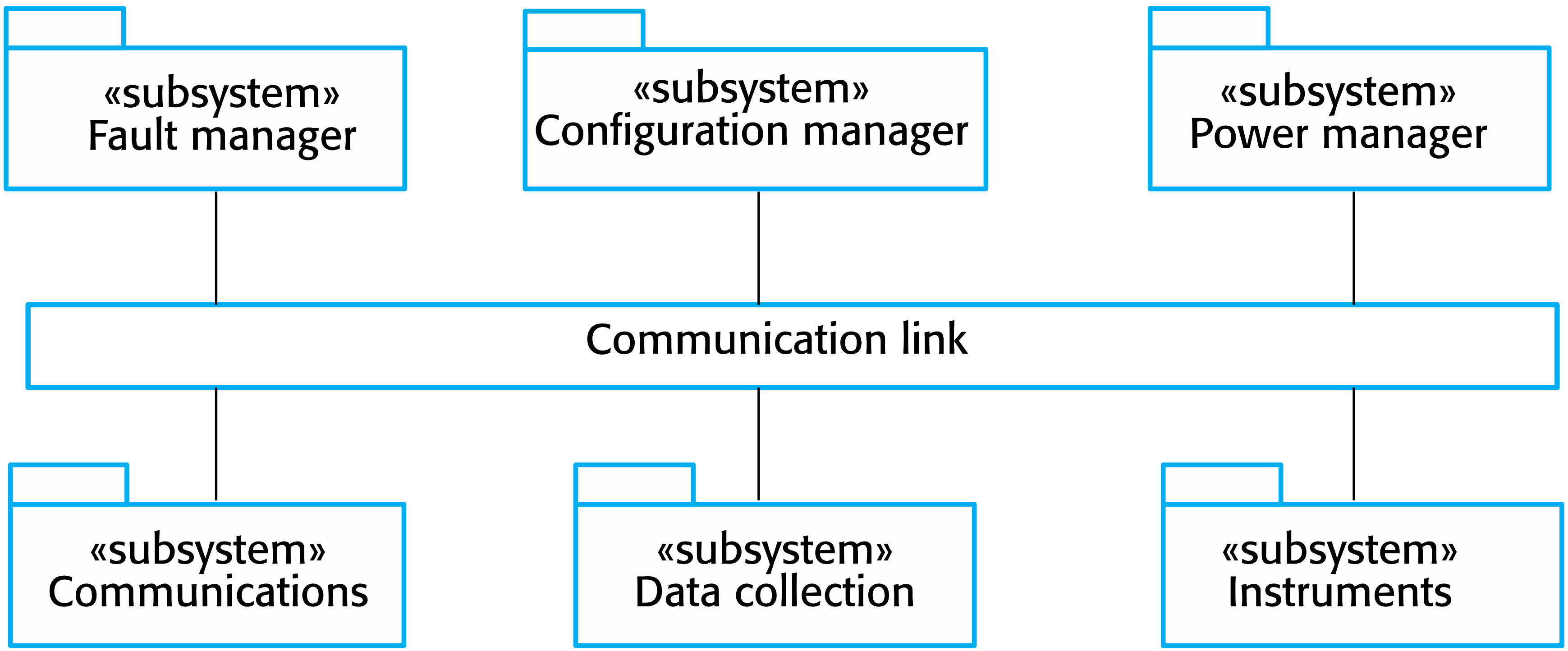
| System | Weather station |
| --- | --- |
| Use case | Report weather |
| Actors | Weather information system, Weather station |
| Description | The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals. |
| Stimulus | The weather information system establishes a satellite communication link with the weather station and requests transmission of the data. |
| Response | The summarized data is sent to the weather information system. |
| Comments | Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future. |

# ARCHITECTURAL DESIGN

- ✓ Once interactions between the system and its environment have been understood, we use this information for designing the system architecture.

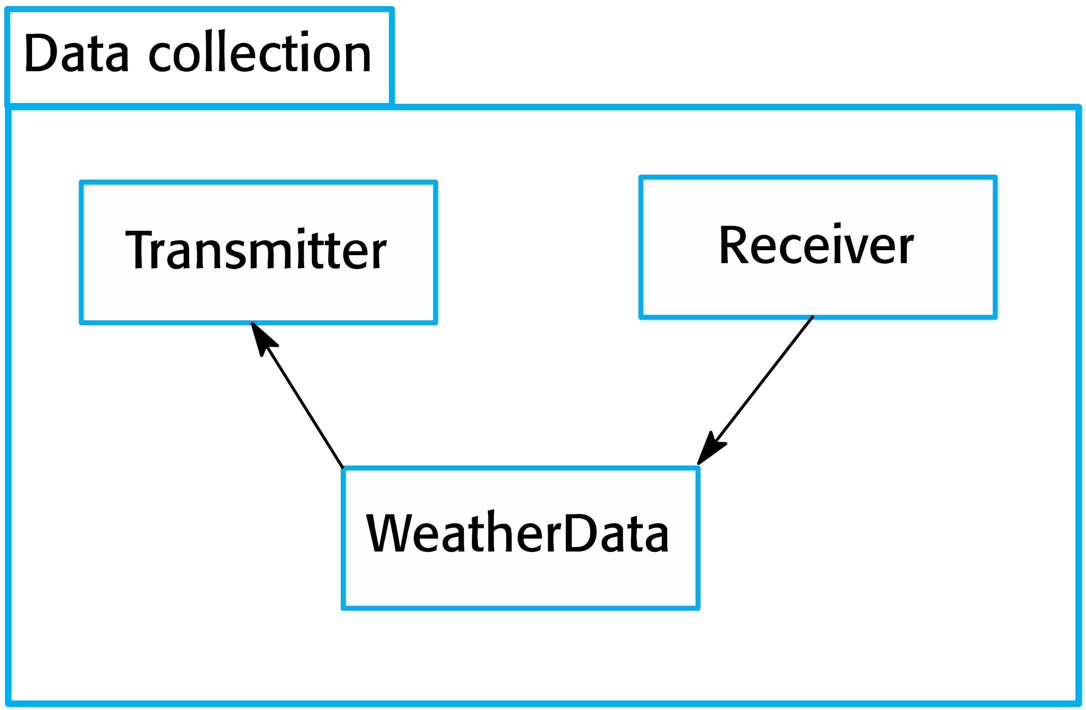
- ✓ Identify the major components that make up the system and their interactions,
- ✓ and then may organize the components using an architectural pattern such as a layered or client-server model.

# HIGH-LEVEL ARCHITECTURE OF THE WEATHER STATION



# ARCHITECTURE OF DATA COLLECTION SYSTEM

- Each component can be decomposed into sub-components.



# OBJECT CLASS IDENTIFICATION

- Any 'magic formula' for object identification?
  - NO

# APPROACHES TO IDENTIFICATION

- ✓ Use a grammatical approach based on a natural language description of the system.
- ✓ Base the identification on tangible things in the application domain.
- ✓ Use a behavioural approach and identify objects based on what participates in what behaviour.
- ✓ Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

# EXAMPLE: WEATHER STATION DESCRIPTION

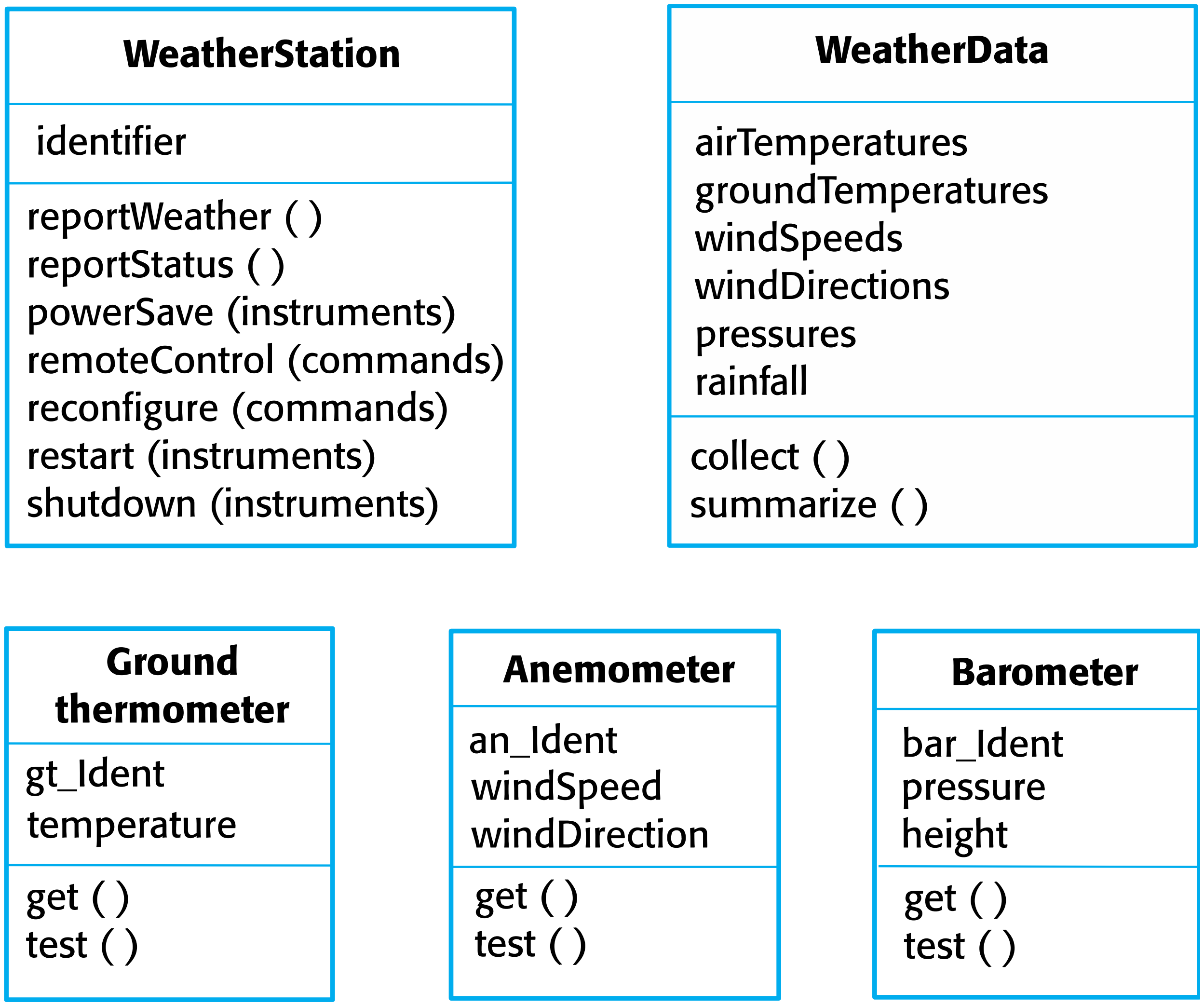
A weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected periodically.

When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.

# WEATHER STATION OBJECT CLASSES

- Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - Ground thermometer, Anemometer, Barometer
    - Application domain objects that are 'hardware' objects related to the instruments in the system.
  - Weather station
    - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
  - Weather data
    - Encapsulates the summarized data from the instruments.

# WEATHER STATION OBJECT CLASSES



# DESIGN MODELS

- ✓ Design models show the objects and object classes and relationships between these entities.

- ✓ Static models describe the static structure of the system in terms of object classes and relationships.
- ✓ Dynamic models describe the dynamic interactions between objects.

# EXAMPLES OF DESIGN MODELS

- Subsystem models that show logical groupings of objects into coherent subsystems.
- Sequence models that show the sequence of object interactions.
- State machine models that show how individual objects change their state in response to events.
- Other models include use-case models, aggregation models, generalisation models, etc.

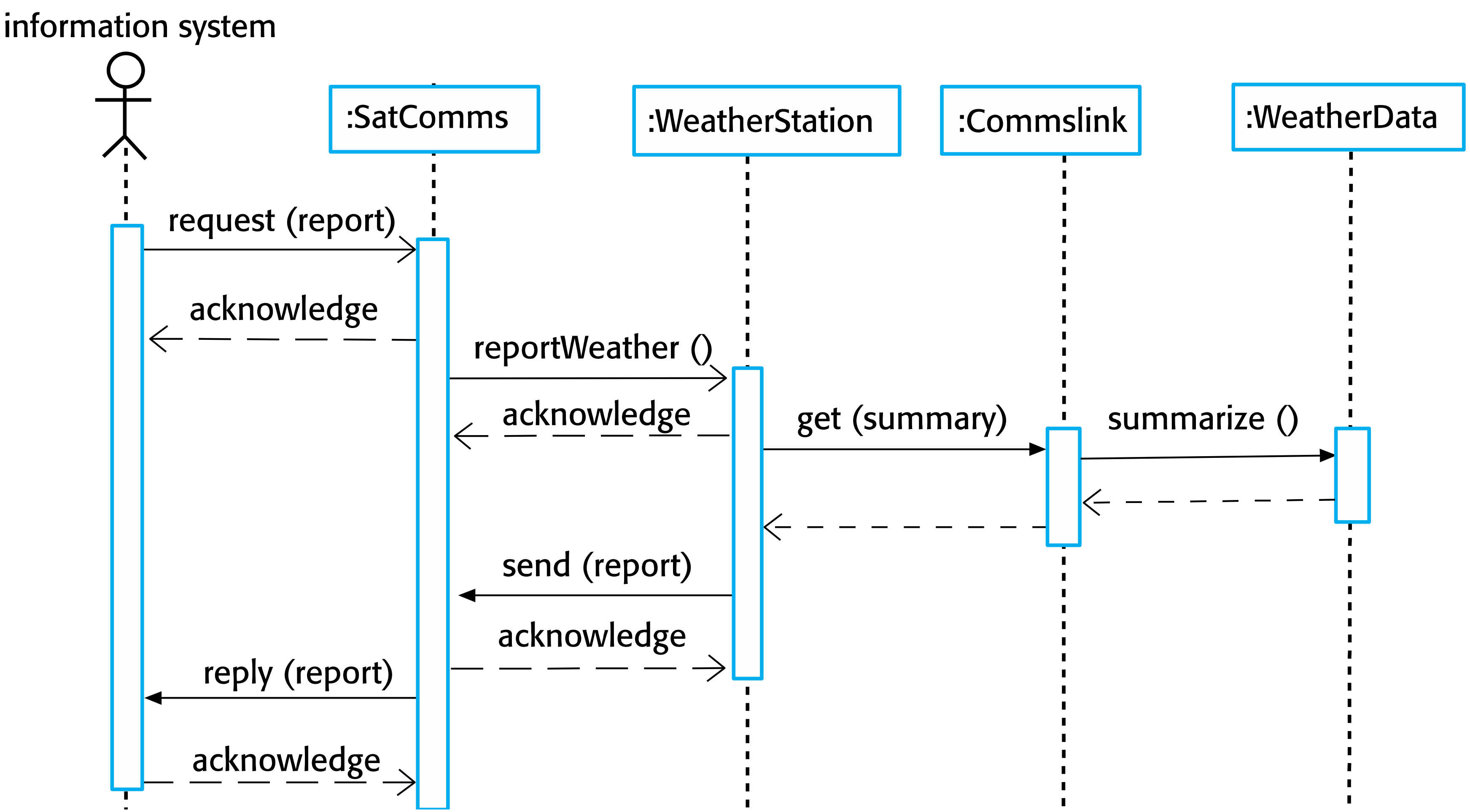
# SUBSYSTEM MODELS

- Shows how the design is organised into logically related groups of objects.
- In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

# SEQUENCE MODELS

- Sequence models show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

# SEQUENCE DIAGRAM DESCRIBING DATA COLLECTION

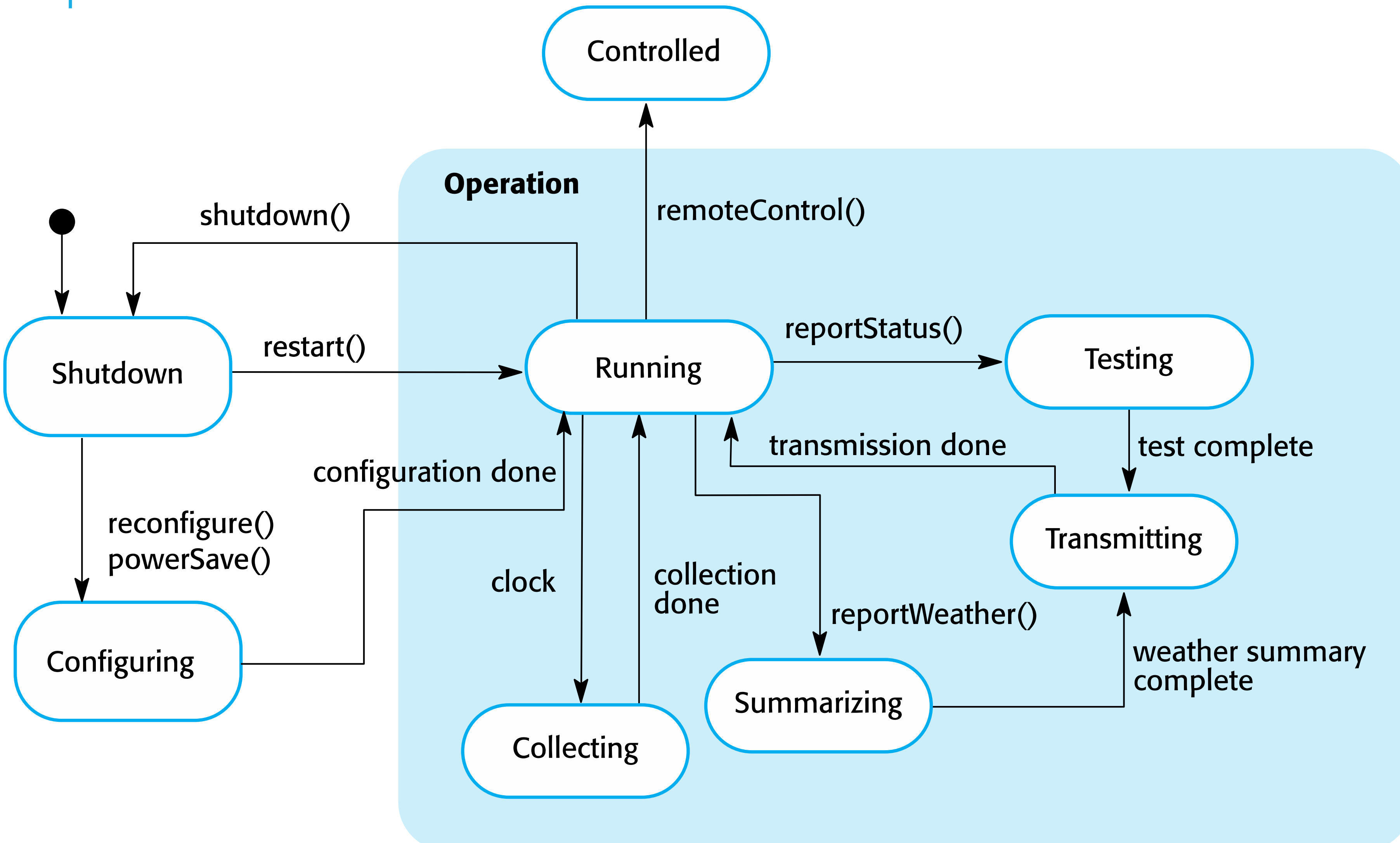


# STATE DIAGRAMS

- ✓ State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- ✓ State diagrams are useful high-level models of a system or an object's run-time behavior.

- ✓ You don't usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

# WEATHER STATION STATE DIAGRAM



# INTERFACE SPECIFICATION

- ✓ Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- ✓ Designers should avoid designing the interface representation but should hide this in the object itself.
- ✓ Objects may have several interfaces which are viewpoints on the methods provided.

- ✓ The UML uses class diagrams  for interface specification but Java may also be used.

# WEATHER STATION INTERFACES

**«interface»**
**Reporting**

weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport

**«interface»**
**Remote Control**

startInstrument(instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument ): string

# DESIGN PATTERNS

# DESIGN PATTERNS

- ✓ A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- ✓ A pattern is a description of the problem and the essence of its solution.
- ✓ It should be sufficiently abstract to be reused in different settings.
- ✓ Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

# PATTERN ELEMENTS

- ✓ Name
  - A meaningful pattern identifier.
- ✓ Problem description.
- ✓ Solution description.
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- ✓ Consequences
  - The results and trade-offs of applying the pattern.

# THE OBSERVER PATTERN

- Name
  - Observer.
- Description
  - Separates the display of object state from the object itself.
- Problem description
  - Used when multiple displays of state are needed.
- Solution description
  - See slide with UML description.
- Consequences
  - Optimisations to enhance display performance are impractical.

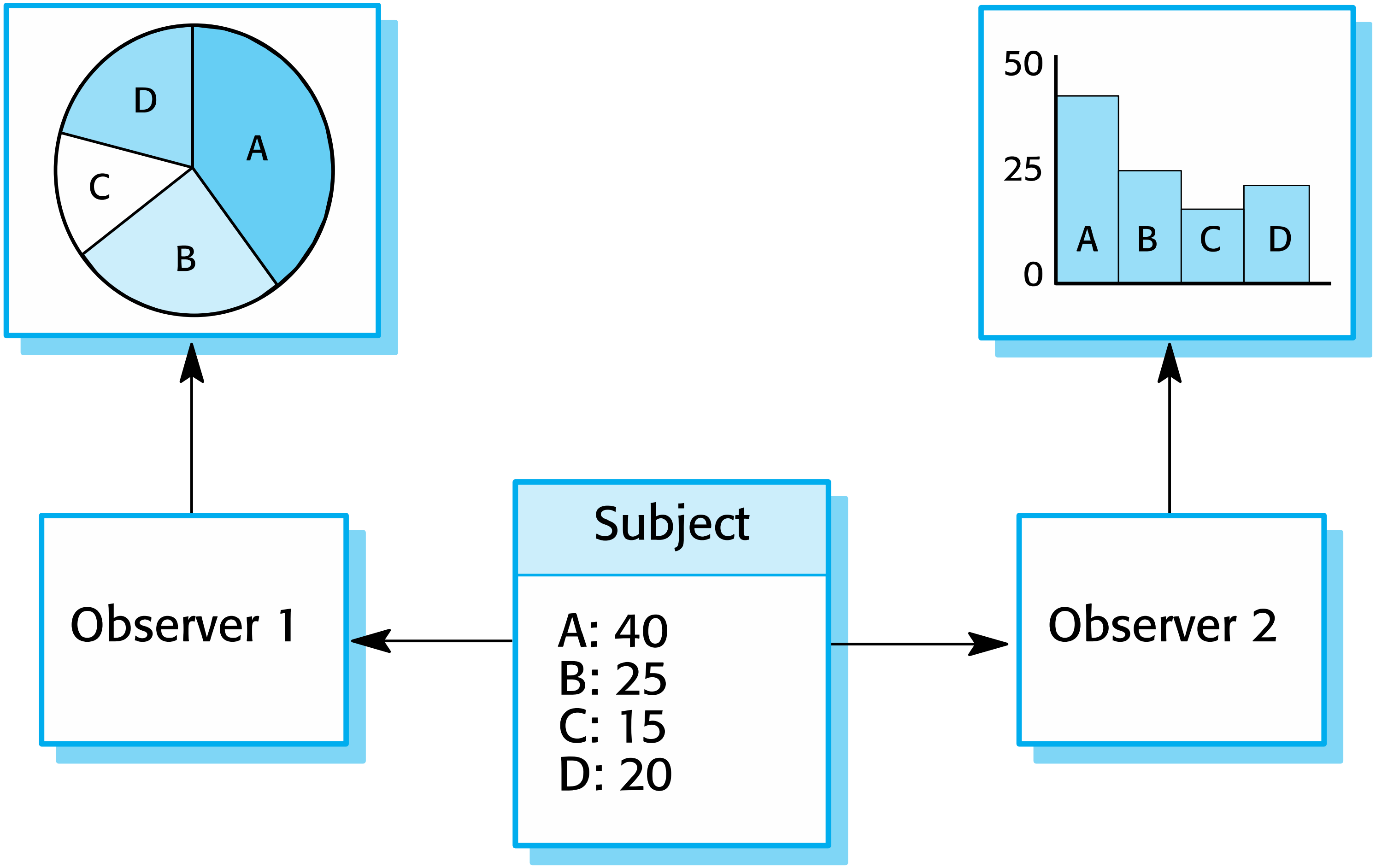
# THE OBSERVER PATTERN (1)

| Pattern name | Observer |
|---|---|
| Description | Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change. |
| Problem description | In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.<br>This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used. |

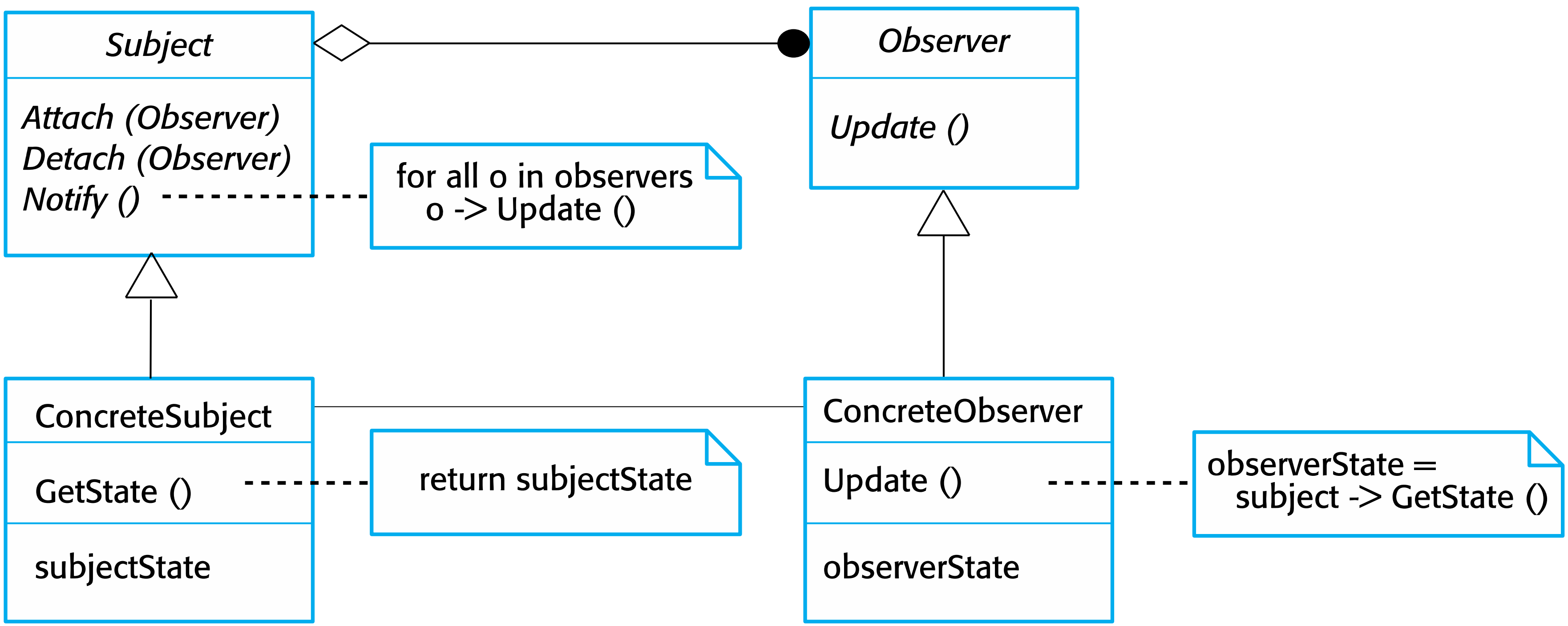
# THE OBSERVER PATTERN (2)

| Pattern name | Observer |
|---|---|
| Solution description | This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObserver, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.<br>The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated. |
| Consequences | The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary. |

# MULTIPLE DISPLAYS USING THE OBSERVER PATTERN



# A UML MODEL OF THE OBSERVER PATTERN



# DESIGN PROBLEMS

- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
  - Tell several objects that the state of some other object has changed (Observer pattern).
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
  - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

# IMPLEMENTATION ISSUES

# REUSE

- Up to the 1990s, most new software was developed from scratch
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.

- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.

# REUSE LEVELS

- ✓ The abstraction level
  - No software is reused directly but knowledge of successful abstractions in the design of your software.
- ✓ The object level
  - Directly reuse objects from a library rather than writing the code yourself.
- ✓ The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- ✓ The system level
  - Reuse entire application systems.

# REUSE COSTS

- Costs of searching for software to reuse
- Costs of buying the reusable software
- Costs of adapting and configuring the reusable software components or systems
- Costs of integrating reusable software elements

# DEVELOPMENT PLATFORM TOOLS

- ✓ An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- ✓ A language debugging system.
- ✓ Graphical editing tools, such as tools to edit UML models.
- ✓ Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- ✓ Project support tools that help you organize the code for different development projects.

# INTEGRATED DEVELOPMENT ENVIRONMENTS (IDES)

- ✓ Which IDE you use and why?

# COMPONENT/SYSTEM DEPLOYMENT FACTORS

- ✓ The hardware and software requirements of a component
  - For a specific hardware/ software system: a platform that provides the required hardware and software support.
- ✓ The availability requirements of the system
  - High availability systems may require more than one platform
- ✓ Component communications
  - High level of communications traffic between components may require the same platform or on platforms that are physically close to one other

# OPEN SOURCE DEVELOPMENT

# OPEN SOURCE DEVELOPMENT

- Open source development
  - the source code of a software system is published
  - and volunteers are invited to participate in the development process
- Free Software Foundation (www.fsf.org)
- Many volunteer developers are also users of the code.

# OPEN SOURCE SYSTEMS

- ✓ The best-known open source product:
  - Linux operating system

- ✓ Other important open source products:
  - Java,
  - the Apache web server
  - and the mySQL database management system.

# OPEN SOURCE ISSUES

- ✓ Should the product that is being developed make use of open source components?
- ✓ Should an open source approach be used for the software's development?
- ✓ Can do business?

# OPEN SOURCE LICENSING

- ✓ Open-source = source code should be freely available
- ✓ Not mean that anyone can do as they wish with that code.
- ✓ => License models

# LICENSE MODELS

- The GNU General Public License (GPL).
  - This is a so-called ‘reciprocal’ license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- The GNU Lesser General Public License (LGPL)
  - is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- The Berkley Standard Distribution (BSD) License.
  - This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

# SUMMARY

- ✓ Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- ✓ The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- ✓ A range of different models may be produced during an object-oriented design process: static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).

# SUMMARY (CONT.)

- ✓ Component interfaces must be defined precisely so that other objects can use them.
- ✓ When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- ✓ Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- ✓ Open source development involves making the source code of a system publicly available.

# MORE?

# MORE ?

- ✓ UI (User Interface) design
  - Graphic (GUI) ?
- ✓ Package/Class design (?)
- ✓ Database design (?)
- ✓ Story board (?)

# SPECIFY A CLASS

One way to ...

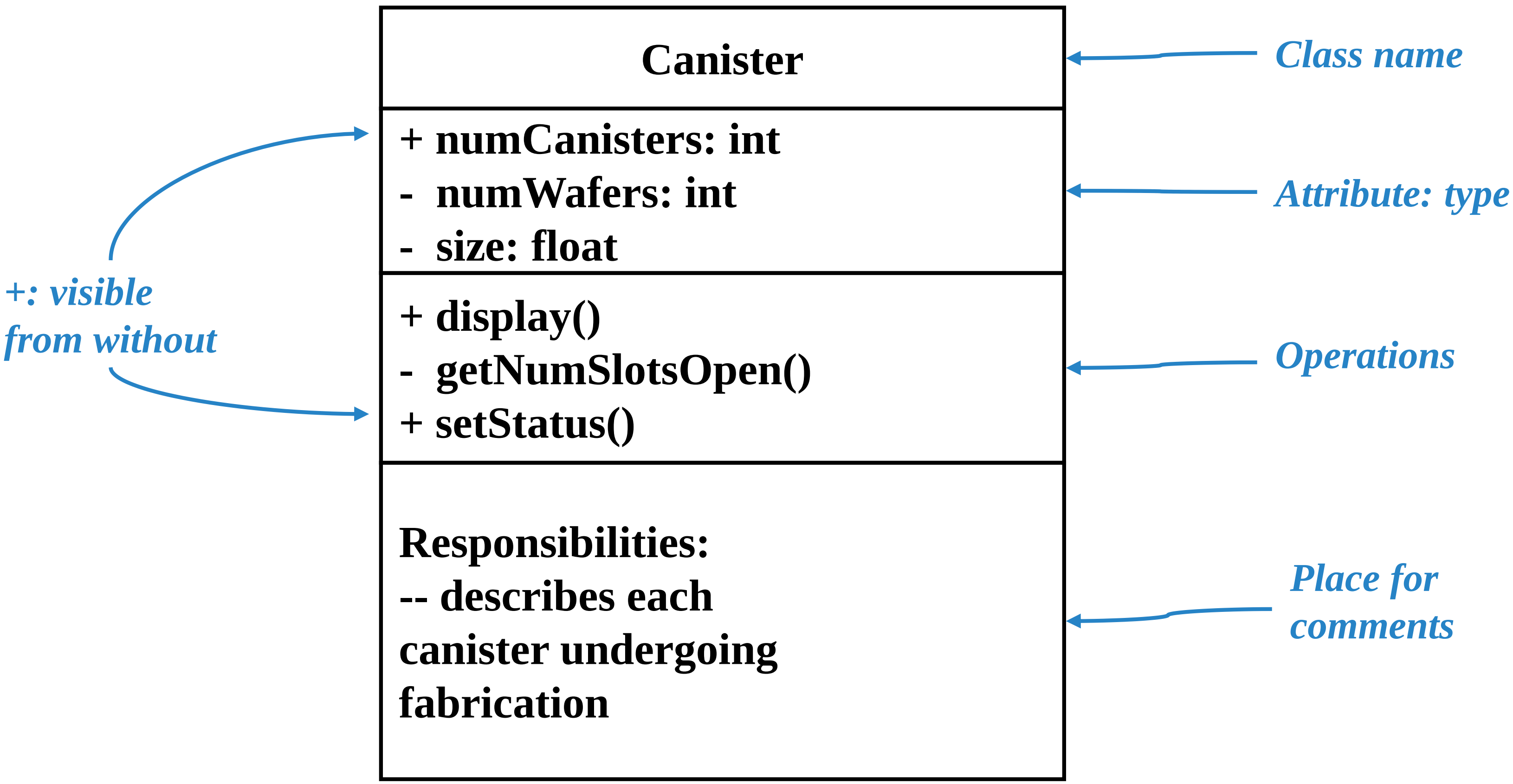
- ✓ Gather the attributes listed in the SRS.
  - if the SRS is organized by class
- ✓ Add additional attributes required for the design.
- ✓ Name a method corresponding to each of the requirements for this class.
  - easy if the SRS is organized by class
- ✓ Name additional methods required for the design.
- ✓ Show the attributes & methods on the object model.
- ✓ State class invariants.

# SPECIFY A FUNCTION

One way to ...

- ✓ Note the section(s) of the SRS or SDD which this function (method) satisfies.
- ✓ State what expressions the function must leave invariant.
- ✓ State the method's pre-conditions (what it assumes).
- ✓ State the method's post-conditions (its effects).
- ✓ Provide pseudocode and/or a flowchart to specify the algorithm to be used.
  - unless very straightforward

# CLASSES AT DETAILED DESIGN



# CLASS/FUNCTION INVARIANTS, PRE- AND POST-CONDITIONS

- Class invariant:
  - Remain true throughout a designated computation
  - Ex: Account class: liquidAssets <= checkBalance + savingBalance
- Function invariant:
  - Assertions about relationships among variables that functions are guaranteed to obey.
  - Ex: adjustQuality() : the sum of the values of the qualities is unchanged.

# SPECIFYING FUNCTIONS: WITHDRAW() IN ACCOUNT

**<u>Invariant</u> of *withdraw()*:**

***availableFundsI* = max( 0, *balanceI* )**

**<u>Precondition*</u>:**

***withdrawalAmountP* >= 0 AND**

***balanceI* - *withdrawalAmountP***

**>= *OVERDRAFT_MAX***

**<u>Postcondition*</u>:**

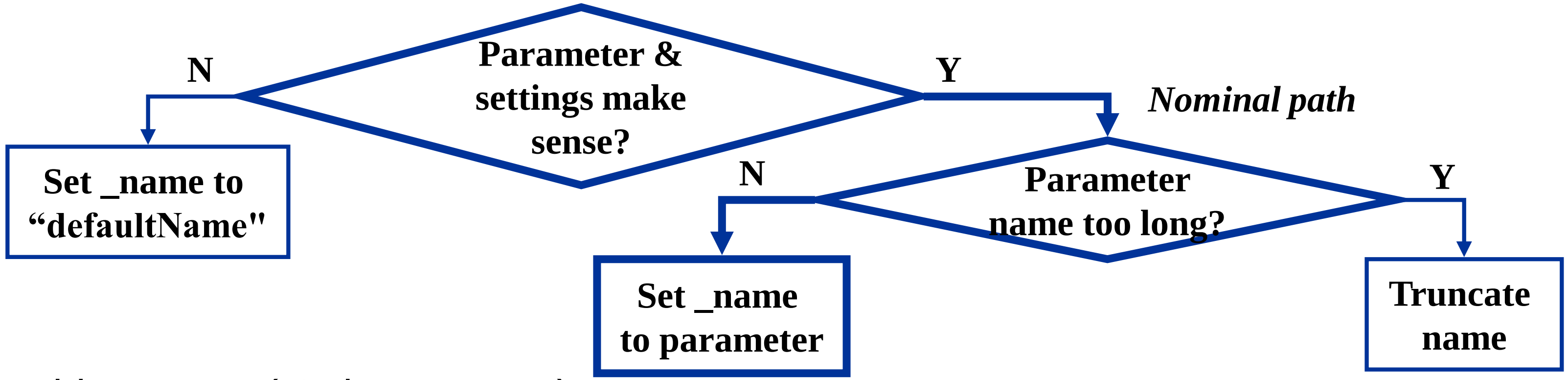
***balanceI'* = *balanceI* - *withdrawalAmountP***

*xI* denotes an attribute;
xP denotes a function parameter;
*x'* is the value of *x* after execution;
*X* denotes a class constant

*The function invariant is an additional pre- and post-condition

# SPECIFYING ALGORITHMS: FLOWCHARTS AND PSEUDOCODE

# FLOWCHART EXAMPLE




```
protected final void setName( String aName )
{
	// Check legitimacy of parameter and settings
	if( ( aName == null ) || ( maxNumCharsInName() <= 0 ) ||
	 ( maxNumCharsInName() > alltimeLimitOfNameLength() ) )
	{   _name = new String( "defaultName" );
	    System.out.println ( "defaultName selected by GameCharacter.setName()");
	}  else
	    // Truncate if aName too long
	    if( aName.length() > maxNumCharsInName() )
	        _name = new String ( aName.getBytes(), 0, maxNumCharsInName() );
	    else // assign the parameter name
	        _name = new String( aName );
}
```

# PSEUDOCODE EXAMPLE

```
FOR number of microseconds supplied by operator
    IF number of microseconds exceeds critical value
            Try to get supervisor's approval
            IF no supervisor's approval
                abort with "no supervisor approval for unusual duration"
            message
            ENDIF
        ENDIF
        IF power level exceeds critical value
            abort with "power level exceeded" message
        ENDIF
        IF ( patient properly aligned & shield properly placed
            & machine self-test passed )
            Apply X-ray at power level p
        ENDIF
ENDFOR
```

# ADVANTAGES OF PSEUDOCODE & FLOWCHARTS

- ✓ Clarify algorithms in many cases
- ✓ Impose increased discipline on the process of documenting detailed design
- ✓ Provide additional level at which inspection can be performed
  - Help to trap defects before they become code
  - Increases product reliability
- ✓ May decreases overall costs

# DISADVANTAGES OF PSEUDOCODE & FLOWCHARTS

- ✓ Creates an additional level of documentation to maintain
- ✓ Introduces error possibilities in translating to code
- ✓ May require tool to extract pseudocode and facilitate drawing flowcharts

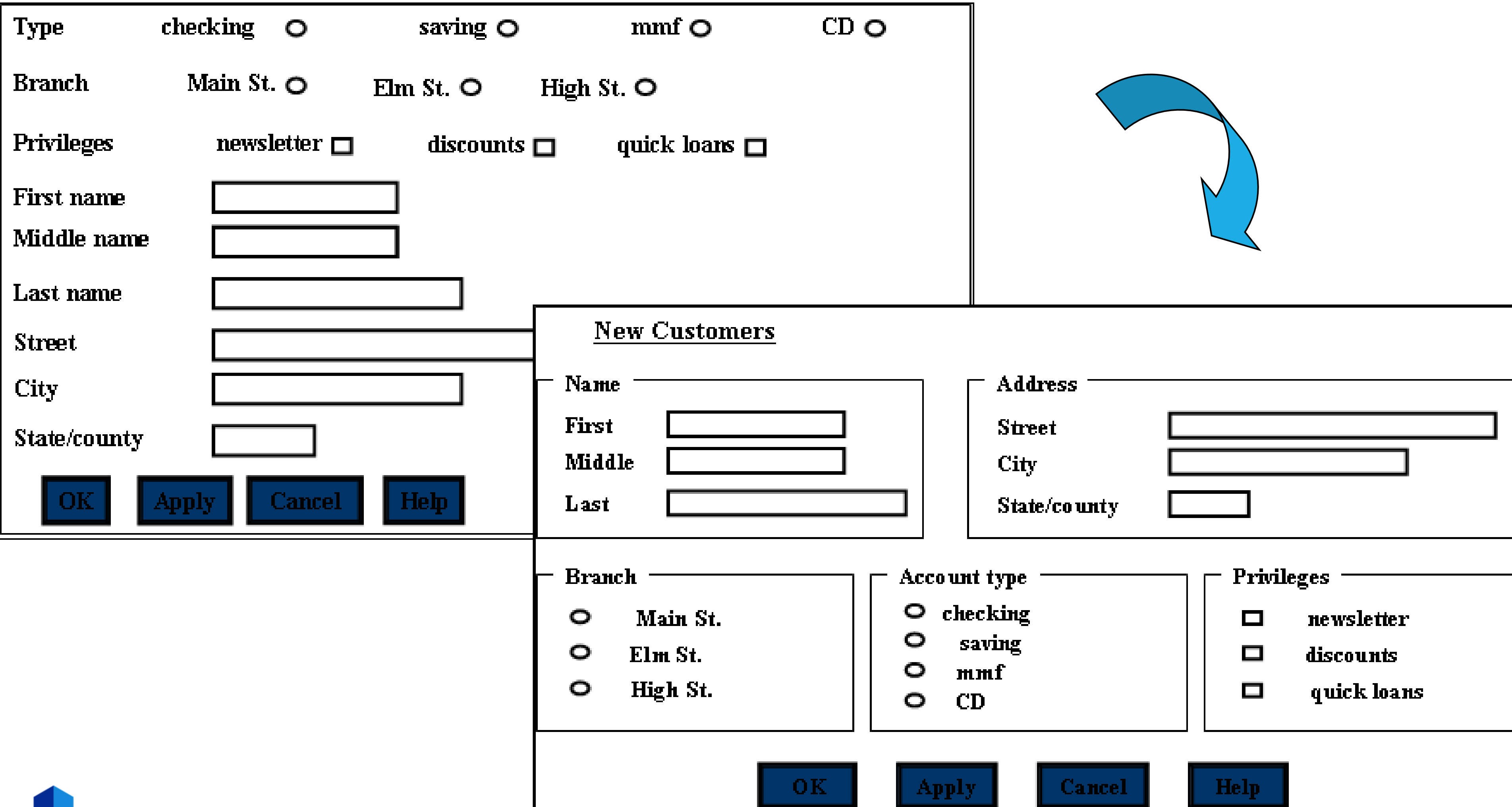
# STEPS FOR CONSTRUCTING USER INTERFACES

- ✓ Step 1: Know your user
- ✓ Step 2: Understand the business function in question
- ✓ Step 3: Apply principles of good screen design
- ✓ Step 4: Select the appropriate kind of windows
- ✓ Step 5: Develop system menus
- ✓ Step 6: Select the appropriate device-based controls
- ✓ Step 7: Choose the appropriate screen-based controls
- ✓ Step 8: Organize and lay out windows
- ✓ Step 9: Choose appropriate colors
- ✓ Step 10: Create meaningful icons
- ✓ Step 11: Provide effective message, feedback, & guidance

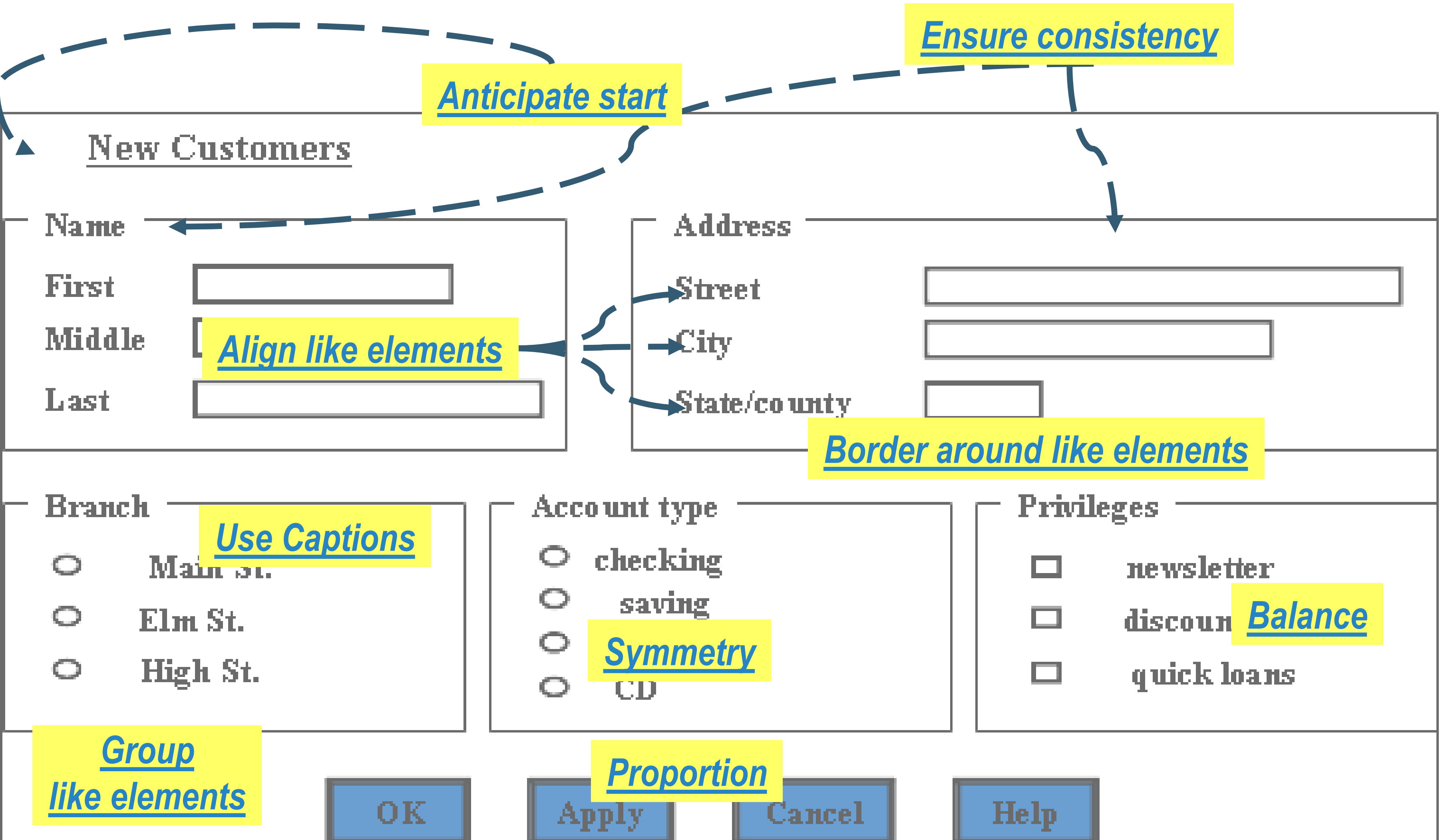
# PRINCIPLES OF GOOD SCREEN DESIGN

- ✓ Ensure consistency among the screens of designated applications, and among screens within each
  - conventions; procedures; look-and-feel; locations
- ✓ Anticipate where the user will usually start
  - frequently upper left -- place “first” element there
- ✓ Make navigation as simple as possible
  - Align/group like elements
  - consider borders around like elements
- ✓ Apply a hierarchy to emphasize order of importance
- ✓ Apply principles of pleasing visuals -- usually:
  - balance; symmetry; regularity; predictability
  - simplicity; unity; proportion; economy
- ✓ Provide captions

# APPLYING PRINCIPLES OF GOOD SCREEN DESIGN



# HOW PRINCIPLES OF GOOD SCREEN DESIGN WERE APPLIED



# DEVELOP SYSTEM MENUS

- ✓ Provide a main menu
- ✓ Display all relevant alternatives (only)
- ✓ Match the menu structure to the structure of the application's task
- ✓ Minimize the number of menu levels
  - Four maximum?