



SOFTWARE ENGINEERING

Chapter 7 – Detail Design

TUẦN 10, 11, 12

Topics covered

- Object-oriented design using the UML
- Design patterns
- Open source development
- More



Design and implementation

- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- Software design and implementation activities are invariably inter-leaved.
 - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
 - Implementation is the process of realizing the design as a program.

Build or buy

- In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
 - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.



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 and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups design models are an important communication mechanism.

Process stages

- 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.
- Process illustrated here using a design for a wilderness weather station.



System context and interactions

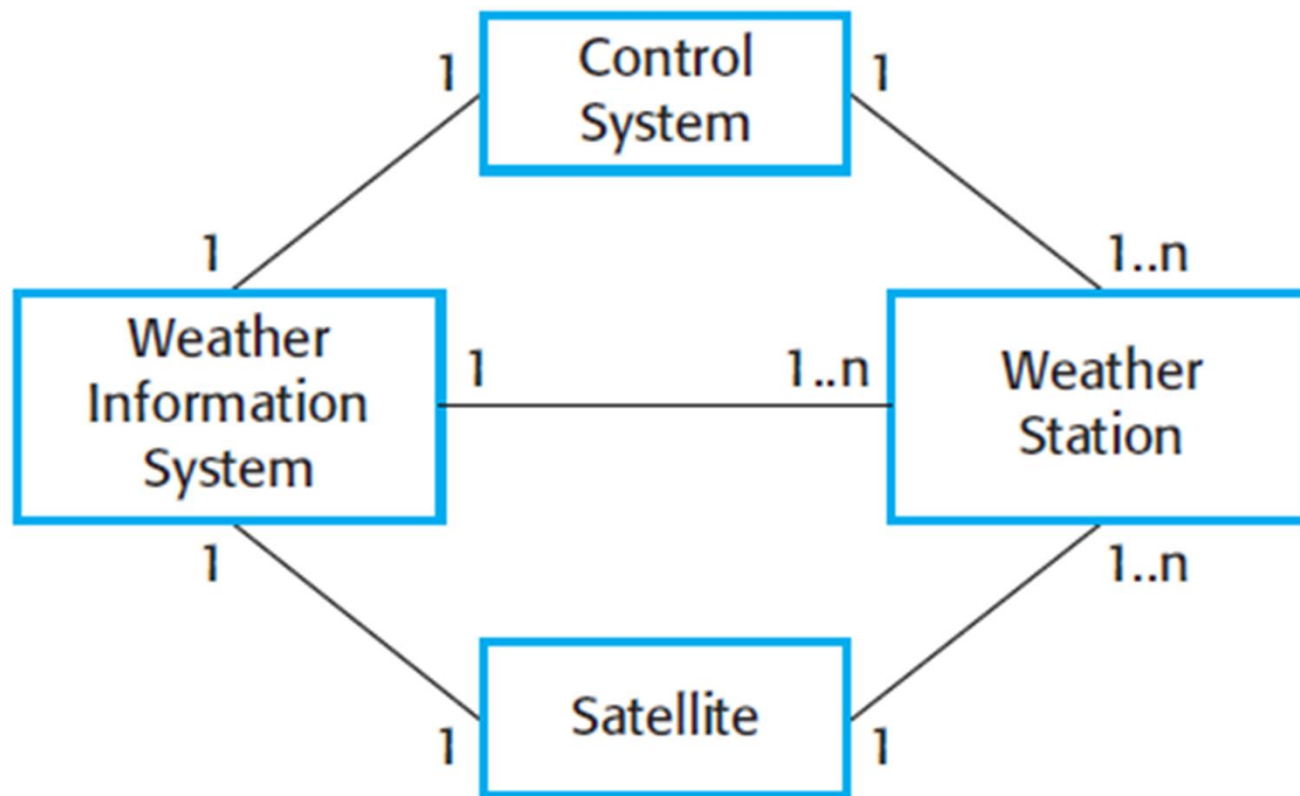
- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.



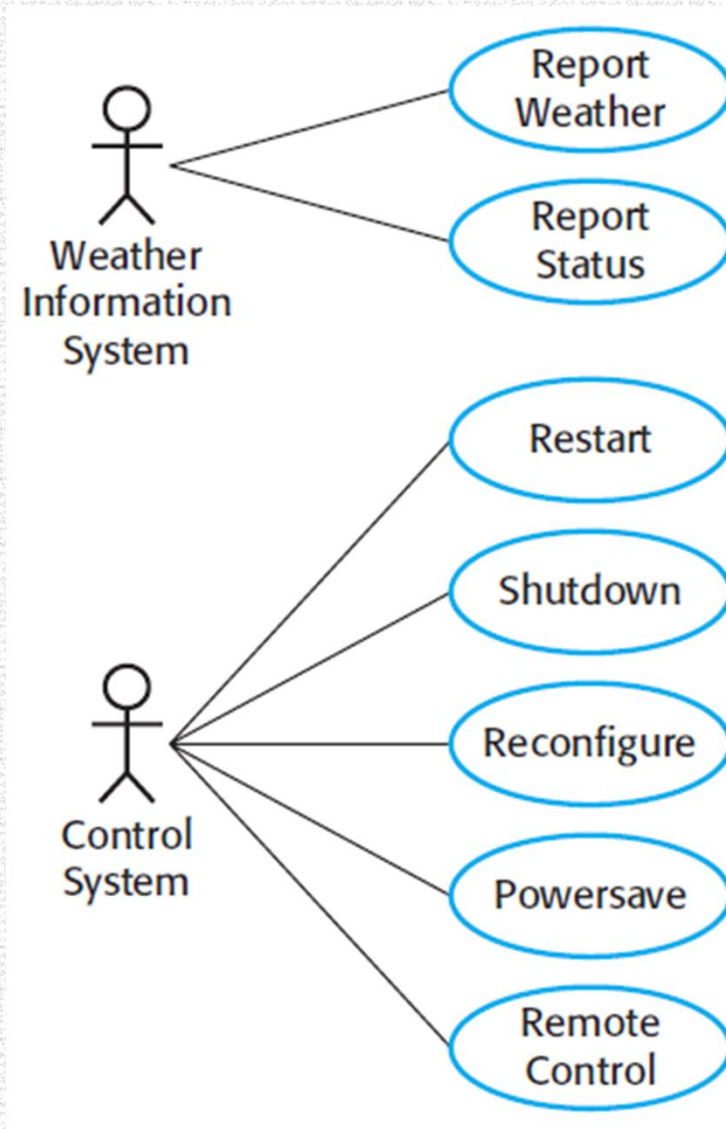
Context and interaction models

- 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



Weather station use cases



Use case description—Report weather

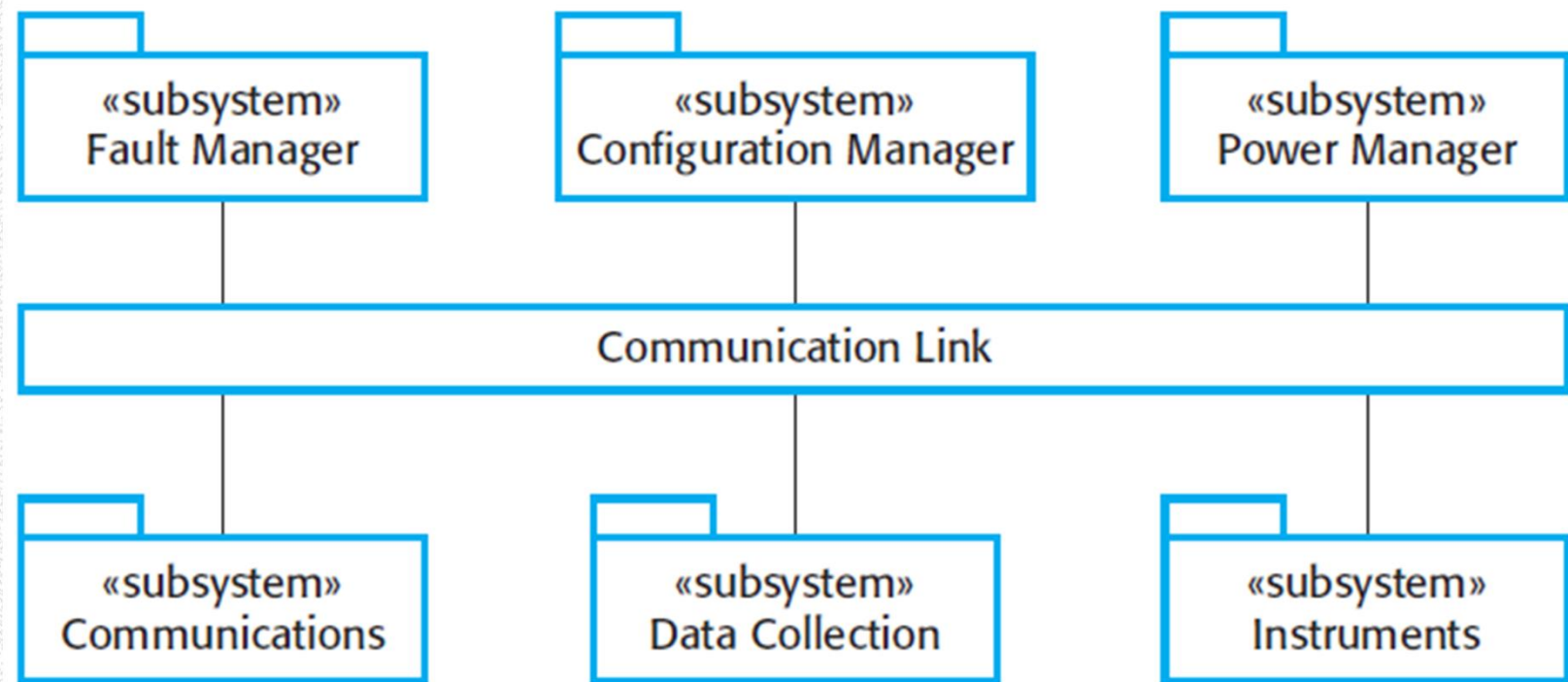
System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, 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

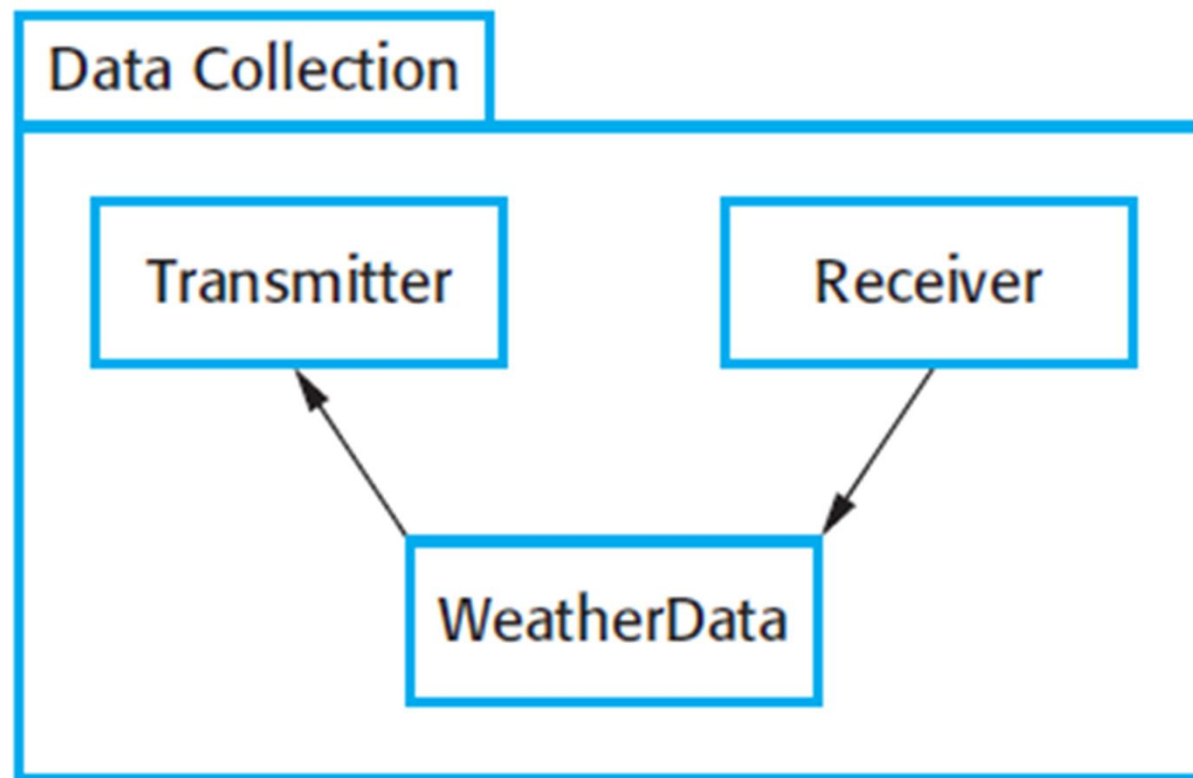
- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.



High-level architecture of the weather station



Architecture of data collection system



Object class identification

- Identifying object classes is often a difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification

- Use a grammatical approach based on a natural language description of the system (used in Hood OOD method).
- 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.

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

WeatherStation

identifier

reportWeather ()
reportStatus ()
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

WeatherData

airTemperatures
groundTemperatures
windSpeeds
windDirections
pressures
rainfall

collect ()
summarize ()

Ground Thermometer

gt_Ident
temperature

get ()
test ()

Anemometer

an_Ident
windSpeed
windDirection

get ()
test ()

Barometer

bar_Ident
pressure
height

get ()
test ()

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.

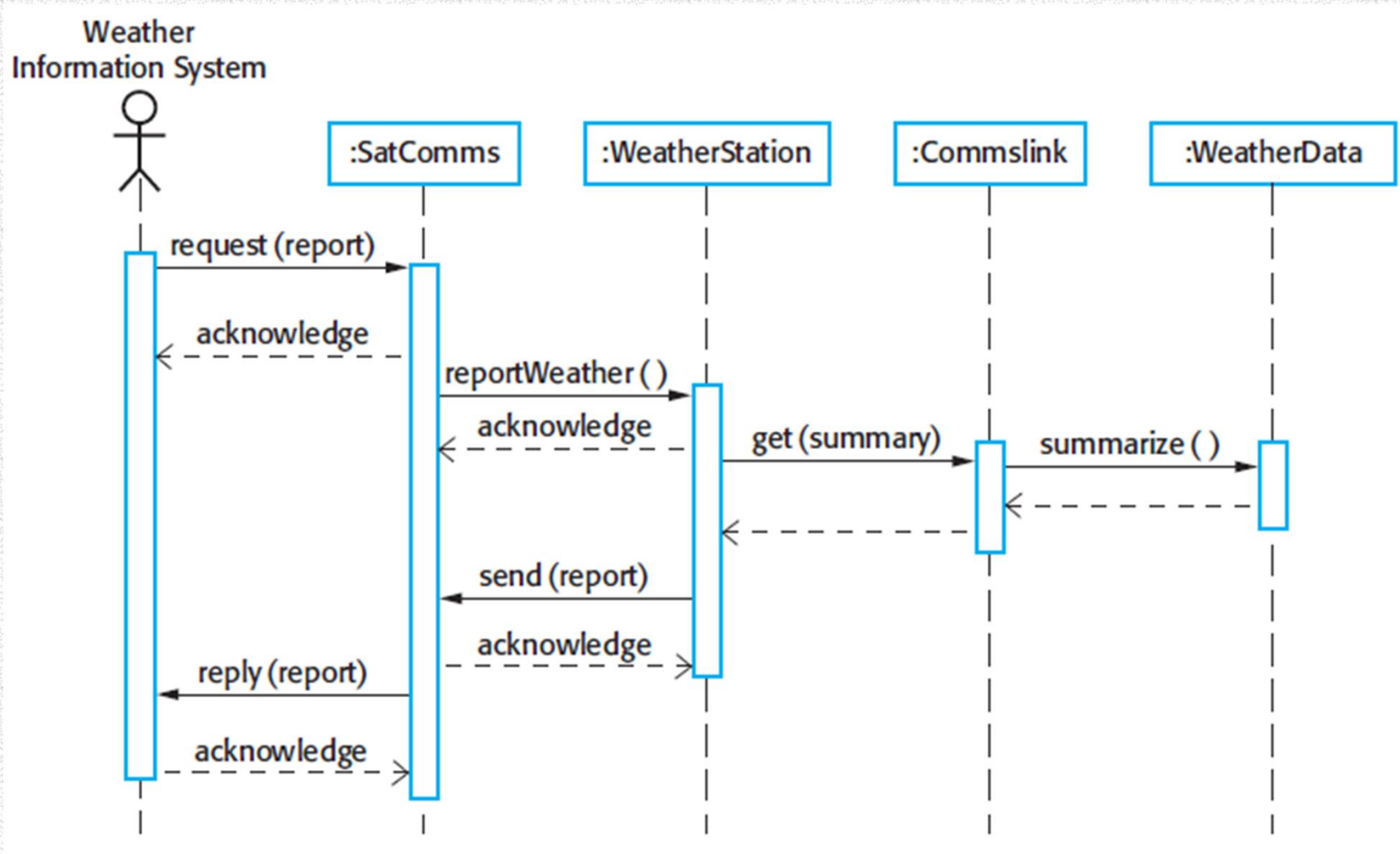
Subsystem models

- 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.

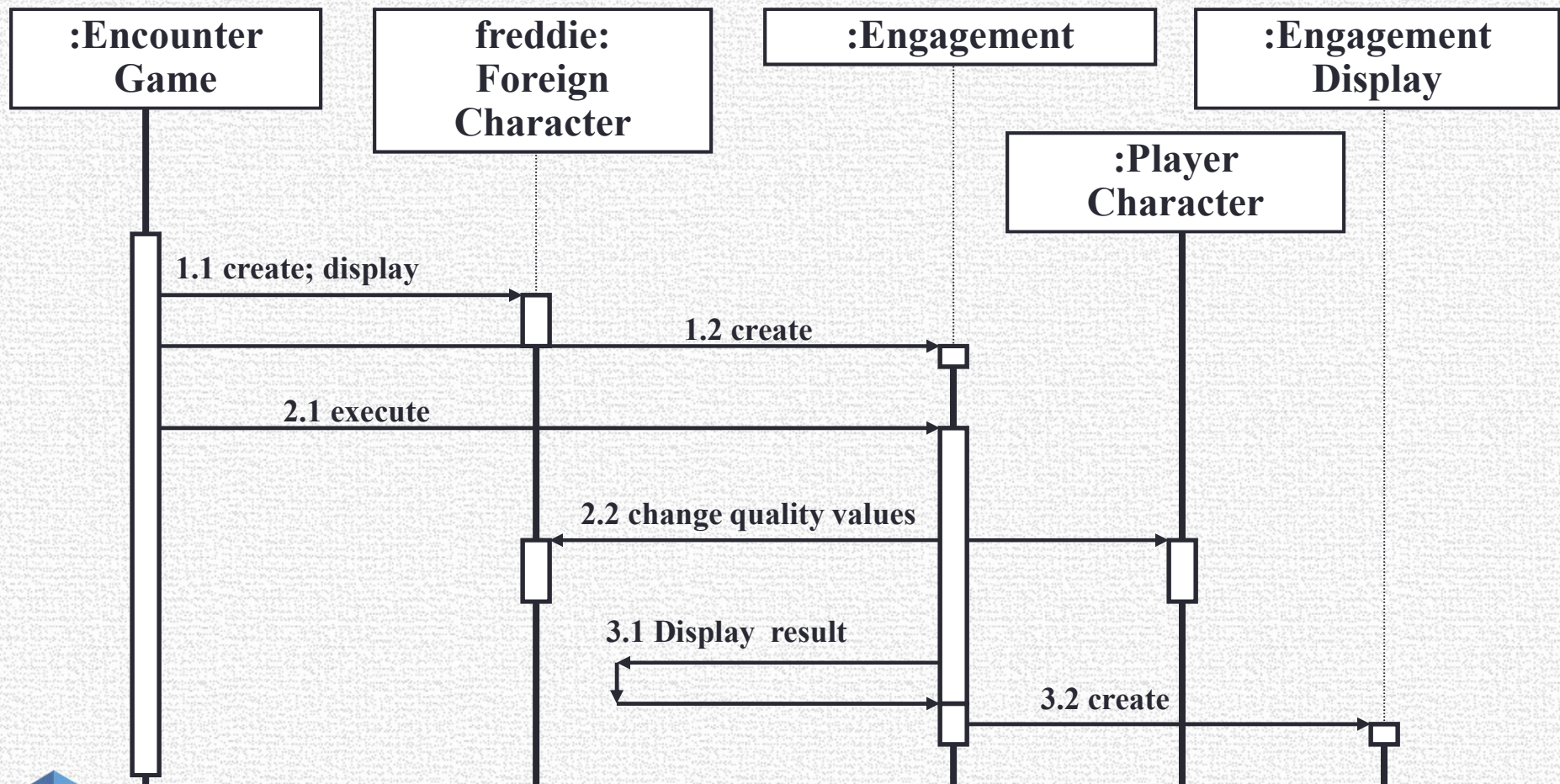
Sequence models

- 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.

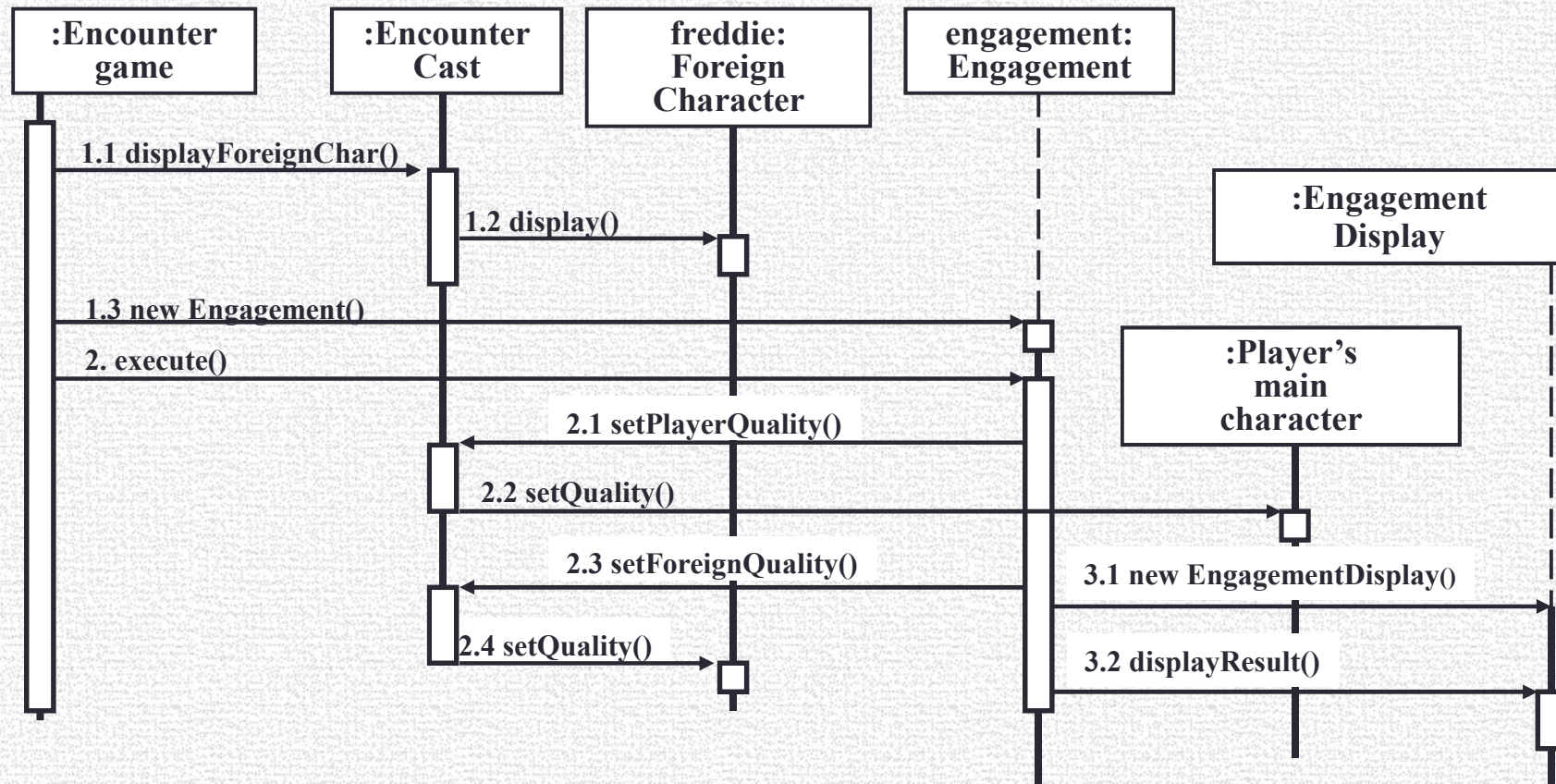
Sequence diagram describing data collection



Sequence Diagram for Engage Foreign Character Use Case



Sequence Diagram for Encounter Foreign Character Use Case



Question: What are the differences to that of SRS?

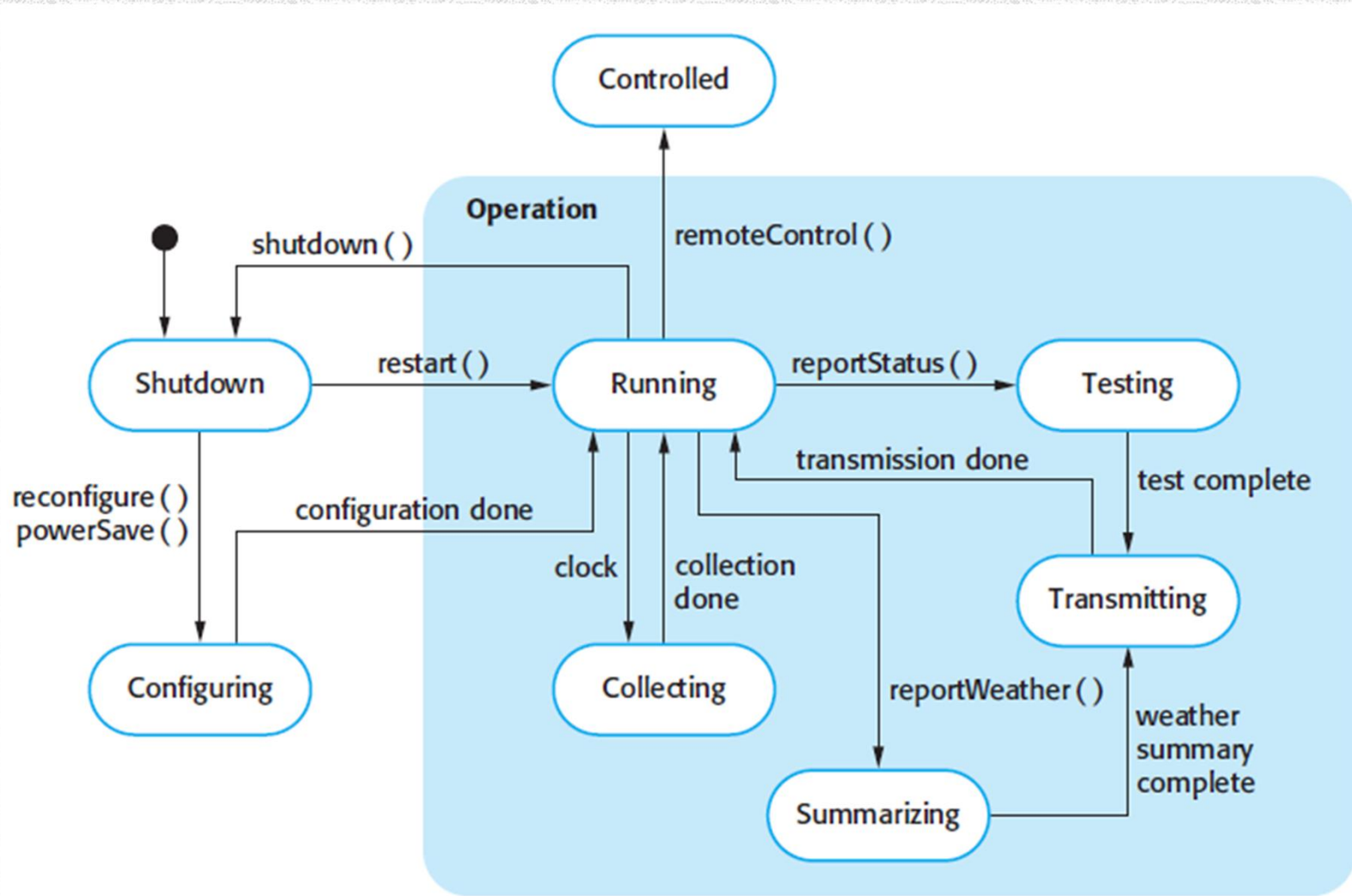
Sequence diagram: srs vs. sdd

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

- 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.

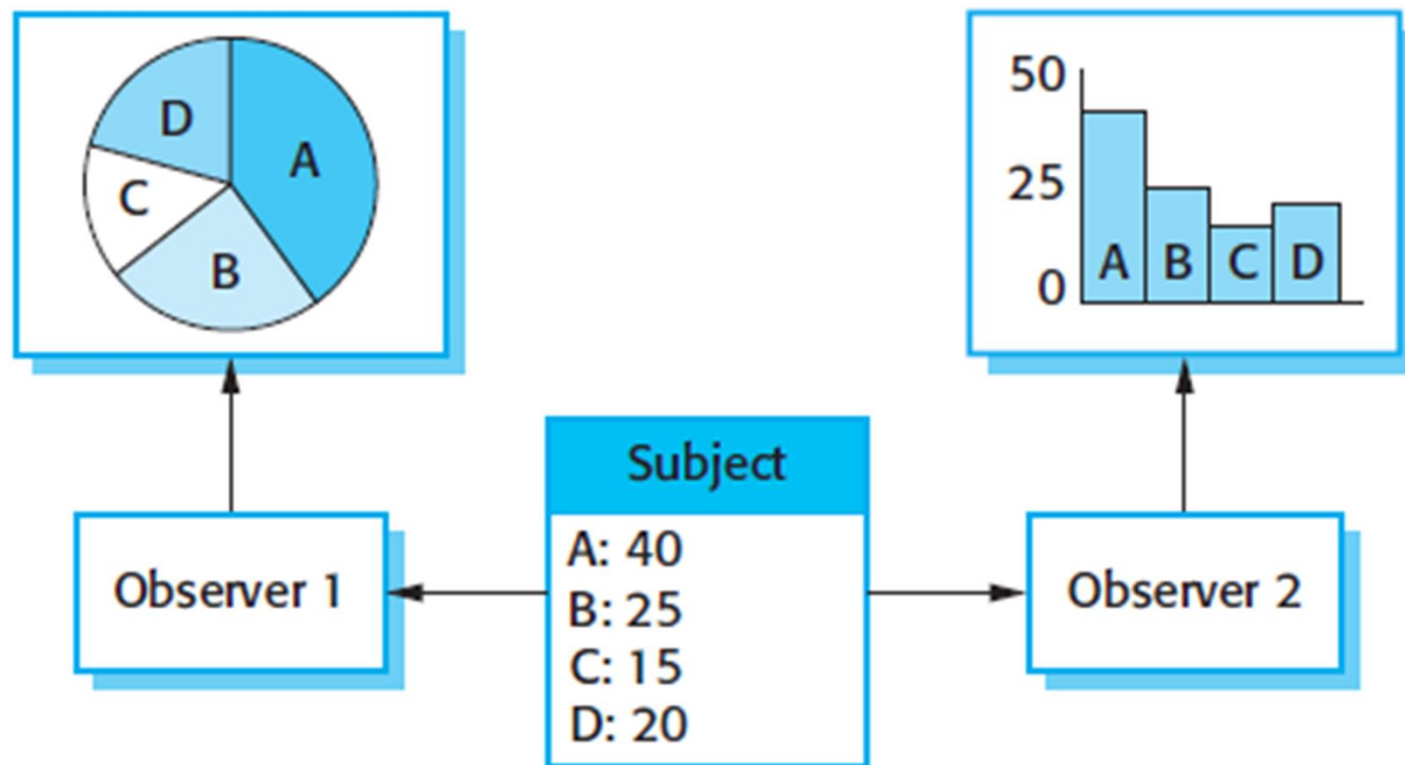
The Observer pattern (1)

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	<p>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.</p> <p>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.</p>

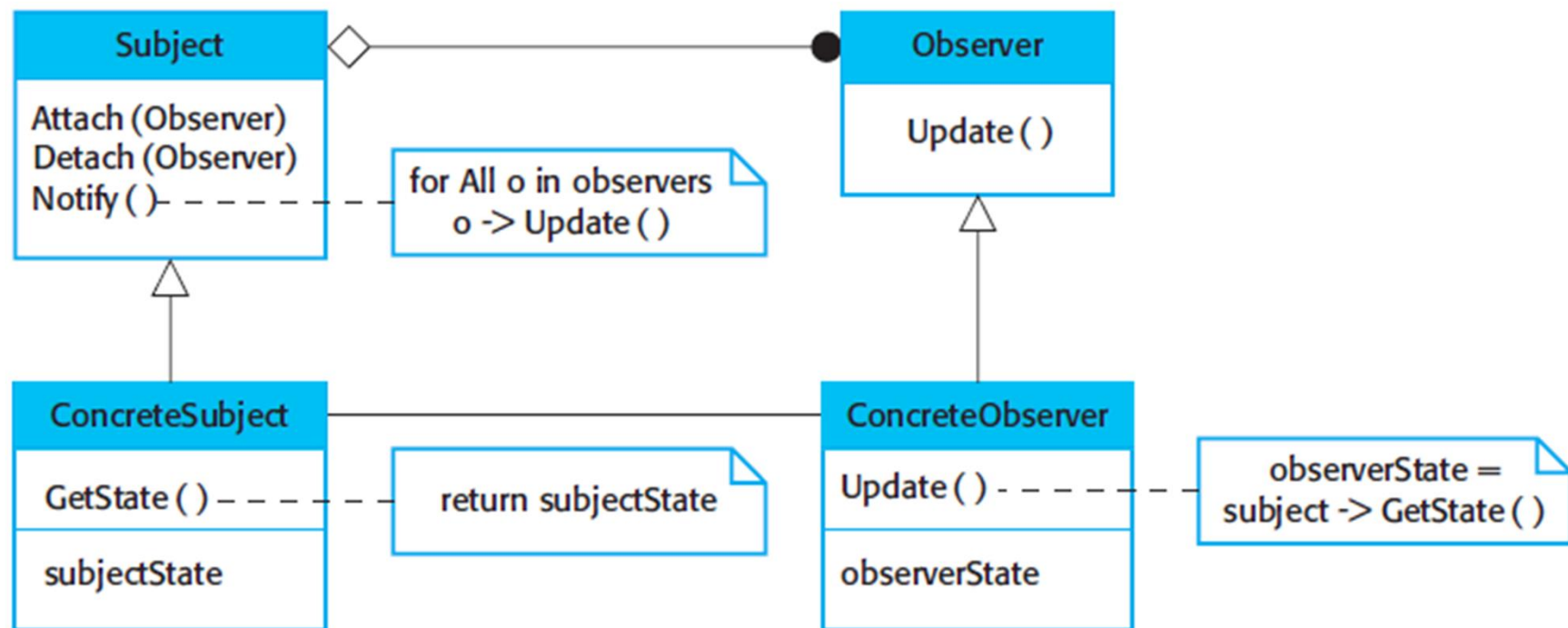
The Observer pattern (2)

Pattern name	Observer
Solution description	<p>This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, 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.</p> <p>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.</p>
Consequences	<p>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.</p>

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).



Reuse

- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
 - The only significant reuse of 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.
- An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.



Reuse levels

- The abstraction level
 - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
 - At this level, you 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
 - At this level, you reuse entire application systems.



Reuse costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.



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)

- Software development tools are often grouped to create an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.



Component/system deployment factors

- If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.



Midterm Exam Revisited (Again)

Xét một hệ thống phần mềm hỗ trợ hoạt động của một chuỗi các cửa hàng bán lẻ thuộc tập đoàn Z. chuyên cung cấp các mặt hàng phục vụ khách tại các nhà ga xe lửa như nhật báo, kẹo bánh, snack, cà phê pha sẵn, khăn giấy... Mỗi cửa hàng được miêu tả qua tên, địa chỉ và một ký hiệu cho biết quy mô của nó (z, zz và zzz tương ứng với cửa hàng loại nhỏ, vừa và lớn). Mỗi cửa hàng được điều hành bởi cửa hàng trưởng, một vài nhân viên tại quầy tính tiền và các nhân viên hậu cần. Chính sách giá cả và chế độ khuyến mãi của tất cả các chuỗi bán lẻ thuộc Z. này được thực hiện đồng nhất trong cả nước (ví dụ mì gói trong ly được khuyến mãi nửa giá trong cả tuần, giá bán snack giống nhau trong toàn quốc). Tuy nhiên mỗi cửa hàng cần quản lý nhập xuất số lượng hàng của riêng mình (ví dụ cà phê đang được khuyến mãi nhưng có thể hết hàng tại một cửa hàng nào đó).

Hệ thống phần mềm này giúp nhân viên quầy thu tiền quét mã vạch và in hóa đơn cho khách. Trong hóa đơn có ghi ngày giờ giao dịch, tên nhân viên thu tiền và địa chỉ cửa hàng và tất nhiên danh sách các mặt hàng đã mua cùng với tổng số tiền. Phần mềm này cũng giúp trưởng cửa hàng tạo báo cáo thống kê theo ngày, tuần, tháng hay quý cũng như kiểm tra các mặt hàng sắp bán hết. Phần mềm có chức năng xác thực bằng thẻ (đối với nhân viên tại quầy tính tiền) cũng như thông qua mặt khẩu (trưởng cửa hàng).



Z chain - Object/Class Identification

- Refers to Slide #16
- Count how many nouns you get from the requirements.
 - Could they be all objects/classes?
 - Some of them really are objects. Others could simply be attributes of the identified objects.
 - Populate your Classes or Objects
- Identify relationship between your classes
- Make a class diagram

Z chain - Interface Design

- Define a few interfaces for your classes

Open source development

- Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.



Open source systems

- The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- Other important open source products are 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?

Open source business

- More and more product companies are using an open source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.



Open source licensing

- A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
 - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
 - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
 - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.



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.



License management

- Establish a system for maintaining information about open-source components that are downloaded and used.
- Be aware of the different types of licenses and understand how a component is licensed before it is used.
- Be aware of evolution pathways for components.
- Educate people about open source.
- Have auditing systems in place.
- Participate in the open source community.

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. These include 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. A UML interface stereotype may be used to define interfaces.
- When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.

Summary (cont.)

- 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. This means that many people can propose changes and improvements to the software.



More ?

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



Specify A Class

One way to ...

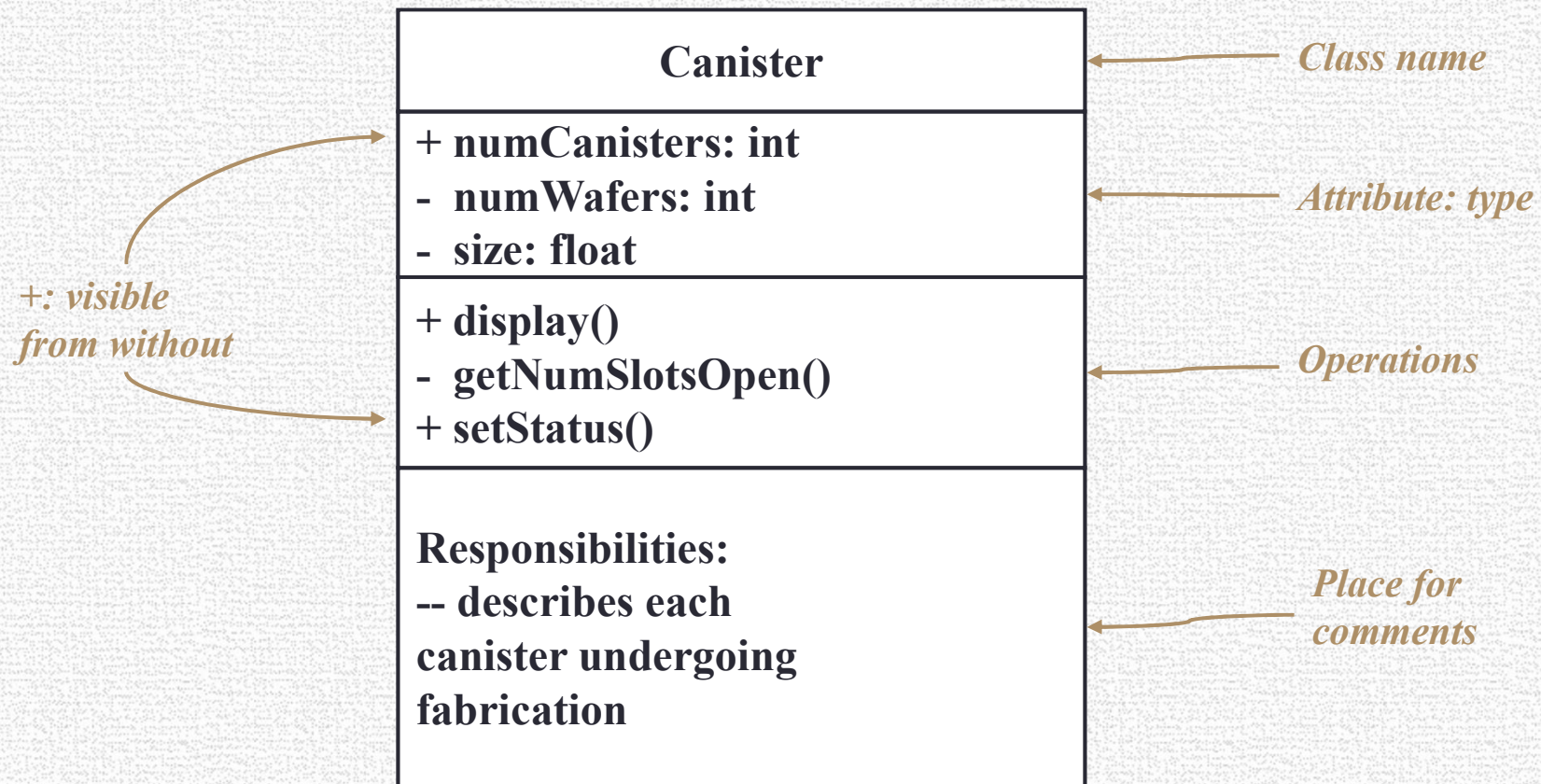
- 1. Gather the attributes listed in the SRS.
 - if the SRS is organized by class
- 2. Add additional attributes required for the design.
- 3. Name a method corresponding to each of the requirements for this class.
 - easy if the SRS is organized by class
- 4. Name additional methods required for the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

Specify a Function

One way to ...

- 1. Note the section(s) of the SRS or SDD which this function (method) satisfies.
- 2. State what expressions the function must leave invariant.
- 3. State the method's pre-conditions (what it assumes).
- 4. State the method's post-conditions (its effects).
- 5. 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: $\text{liquidAssetsI} \leq \text{checkBalanceI} + \text{savingBalanceI}$
- 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

Invariant of *withdraw()*:

$availableFundsI = \max(0, balanceI)$

Precondition*:

$withdrawalAmountP \geq 0 \text{ AND}$

$balanceI - withdrawalAmountP$

$\geq OVERDRAFT_MAX$

Postcondition*:

$balanceI' = balanceI - withdrawalAmountP$

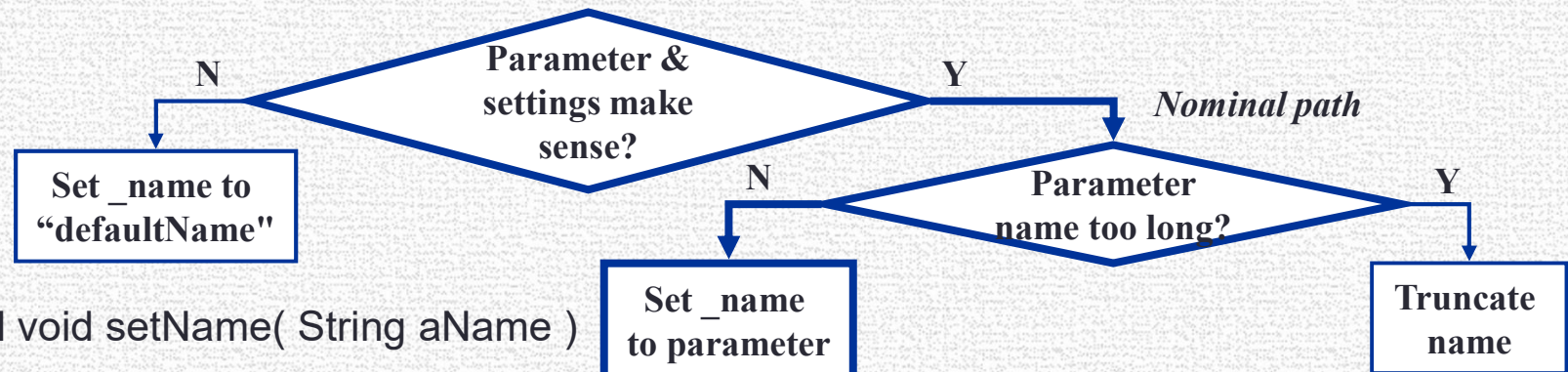
xI denotes an attribute;
 xP denotes a function parameter;
 x' is the value of x after execution;
 X denotes a class constant

*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 decrease 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



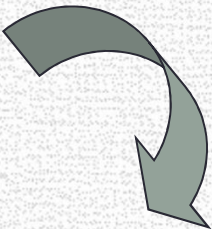
Principles of Good Screen Design

- 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 like elements
 - 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



Type checking ☐ saving ☐ mmf ☐ CD ☐

Branch Main St. ☐ Elm St. ☐ High St. ☐

Privileges newsletter ☐ discounts ☐ quick loans ☐

First name

Middle name

Last name

Street

City

State/country

OK **Apply** **Cancel** **Help**

New Customers

Name

First

Middle

Last

Address

Street

City

State/country

Branch

☐ Main St.

☐ Elm St.

☐ High St.

Account type

☐ checking

☐ saving

☐ mmf

☐ CD

Privileges

☐ newsletter

☐ discounts

☐ quick loans

OK **Apply** **Cancel** **Help**

How Principles of Good Screen Design Were Applied

New Customers

Name

First

Middle

Last

Address

Street

City

State/country

Branch

☐ Main St.

☐ Elm St.

☐ High St.

Account type

☐ checking

☐ saving

☐ CD

Privileges

☐ newsletter

☐ discount

☐ quick loans

Buttons: OK, Apply, Cancel, Help

Annotations:

- Anticipate start**: Points to the first input field.
- Ensure consistency**: Points to the form's border.
- Align like elements**: Points to the alignment of input fields.
- Border around like elements**: Points to the border around the Address section.
- Use Captions**: Points to the section headers.
- Symmetry**: Points to the radio button options.
- Group like elements**: Points to the bottom buttons.
- Balance**: Points to the 'discount' option.

Develop System Menu

- 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?