SSW-540: Fundamentals of Software Engineering

Software Implementation and Open Source

Dr. Richard Ens
School of Systems and Enterprises

Python Pointers

- Coding Style Guidelines provide consistency which improves code's readability, maintainability.
- Include
  - Consistent use of white space
  - Consistent formatting of program elements
- If your project doesn't have a coding style, adopt one of these:
  - Google: http://google.github.io/styleguide/pyguide.html
  - Python: http://www.python.org/dev/peps/pep-0008/

- And watch the video on Python Coding Guidelines in this module!

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

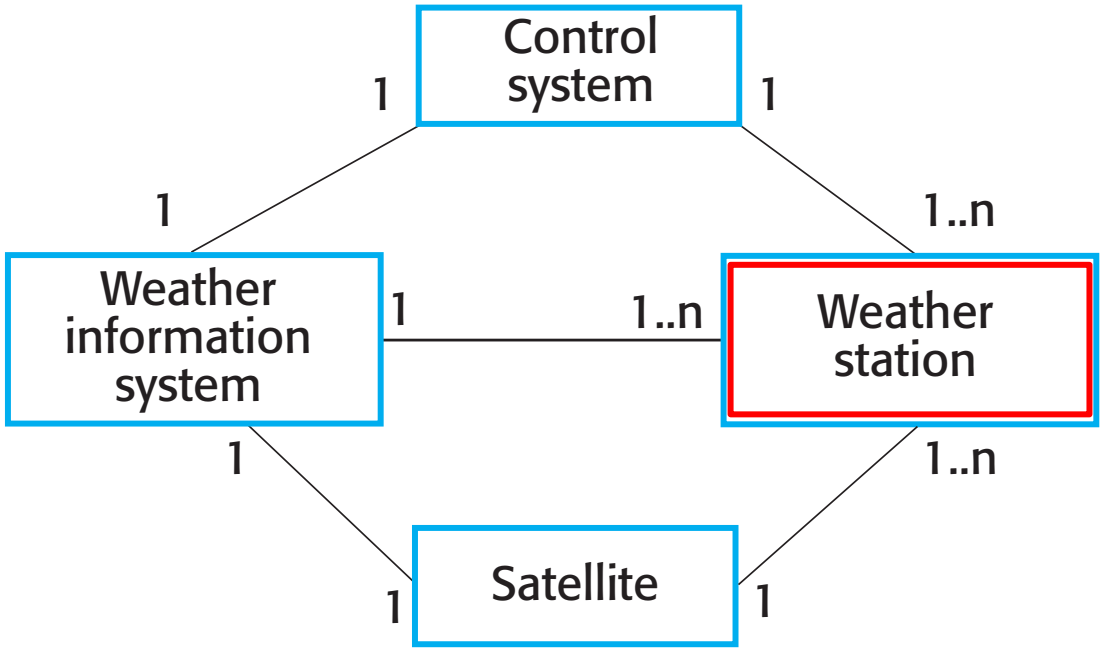
Object-oriented design process

- Object-oriented design is widely used for software systems.
- While there are a variety of different object-oriented design processes found in various organizations, common activities in these processes include:
  - Defining the context and modes of use of the system;
  - Designing the system architecture;
  - Identifying the principal system objects;
  - Developing design models;
  - Specifying object interfaces.
- Sommerville illustrates these process steps using a design for a wilderness weather station.

System context and interactions

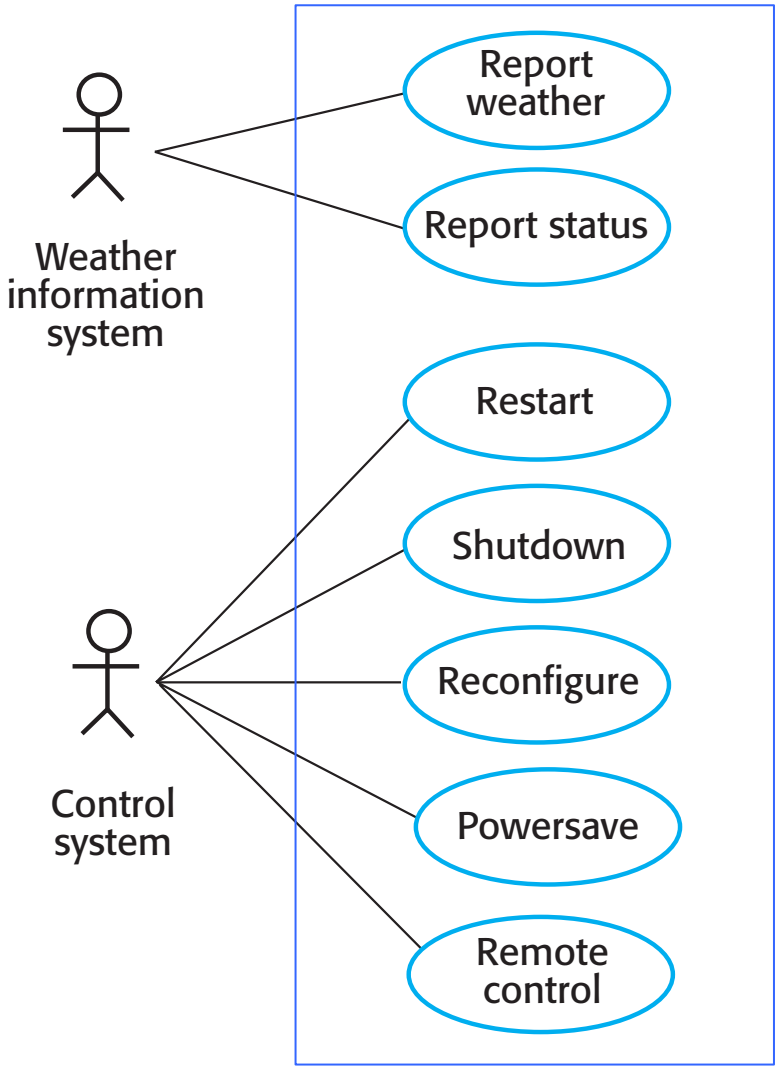
- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- Understanding of the context also lets you establish 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.
- 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 a structural model



Weather station use cases

- The Weather Station is inside the (blue) box!
- Note that the weather station interacts with the Weather Information System and the Control System.
- This diagram shows use cases, but also provides a dynamic context view.



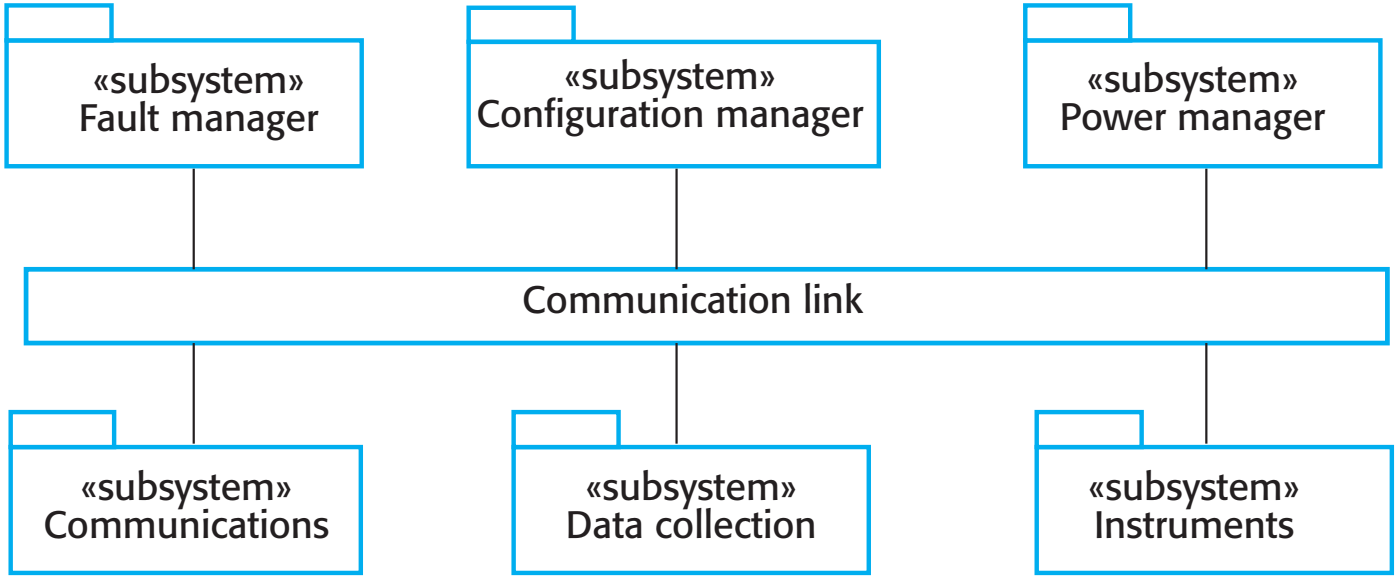
Use case description—Report weather

System	Weather station
Use case	Report weather
Actors	Weather information system
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.

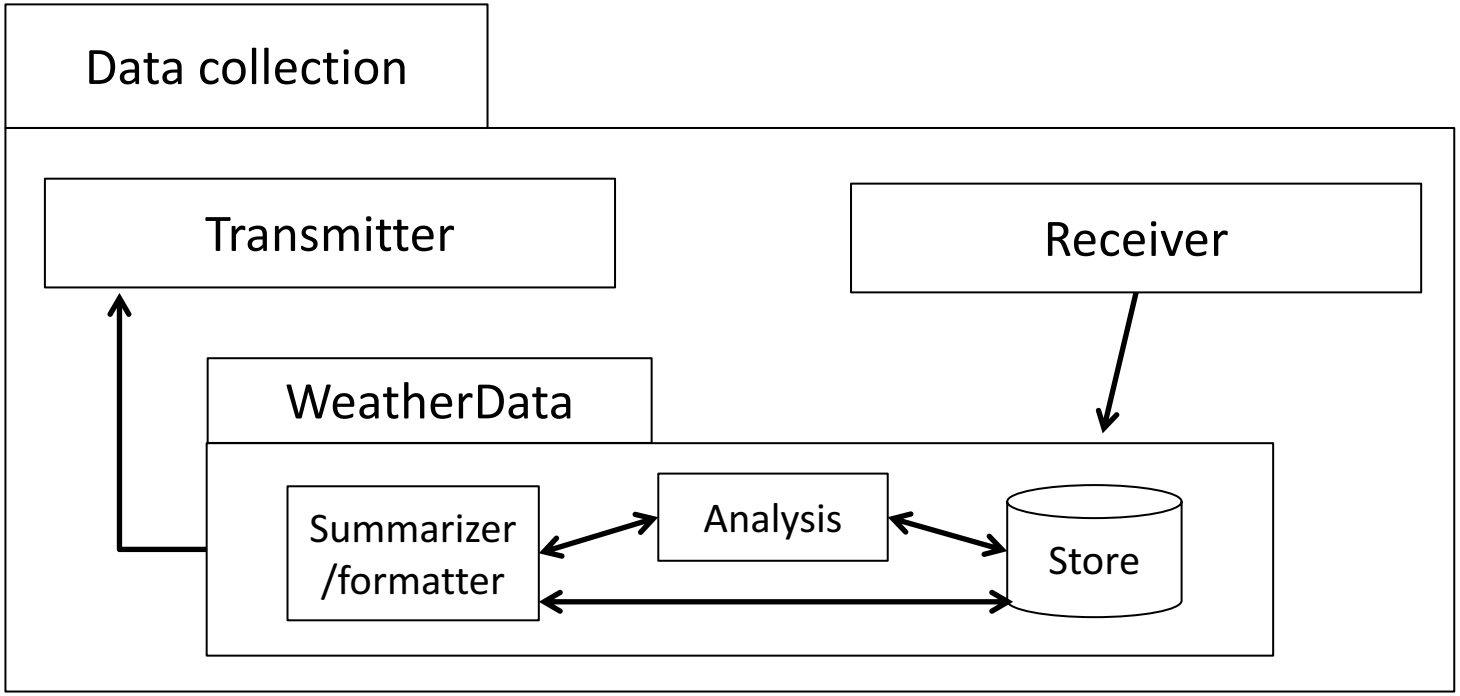
Architectural design

- 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 you may organize the components using an architectural pattern such as a layered or client-server model.

Weather Station: hi-level view



Architecture of data collection sub-system

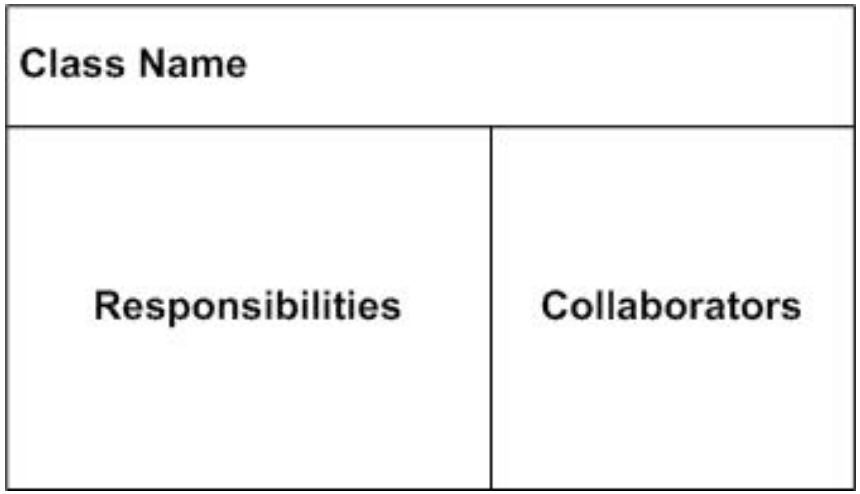


Object class identification

- 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. Try, and iterate!
- Approaches to identifying objects
  - 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.

CRC card for designing objects and classes

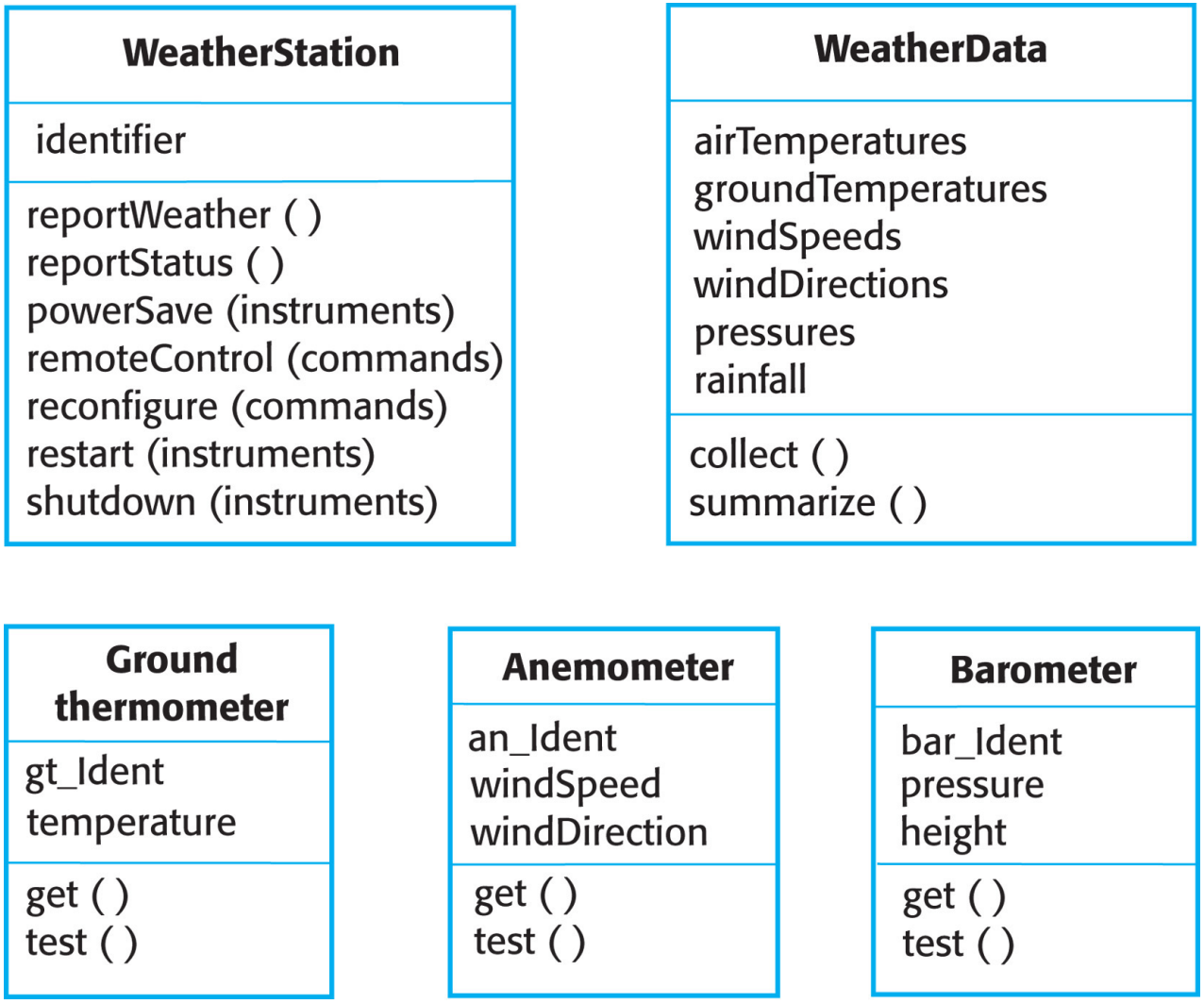
- Class-responsibility-collaboration (**CRC**) **cards** are a brainstorming tool used in the design of object-oriented software.
- The Class – the name of the object
- The Responsibilities – what the class must know and do
- The Collaborations – the other classes that help the class accomplish its responsibilities



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



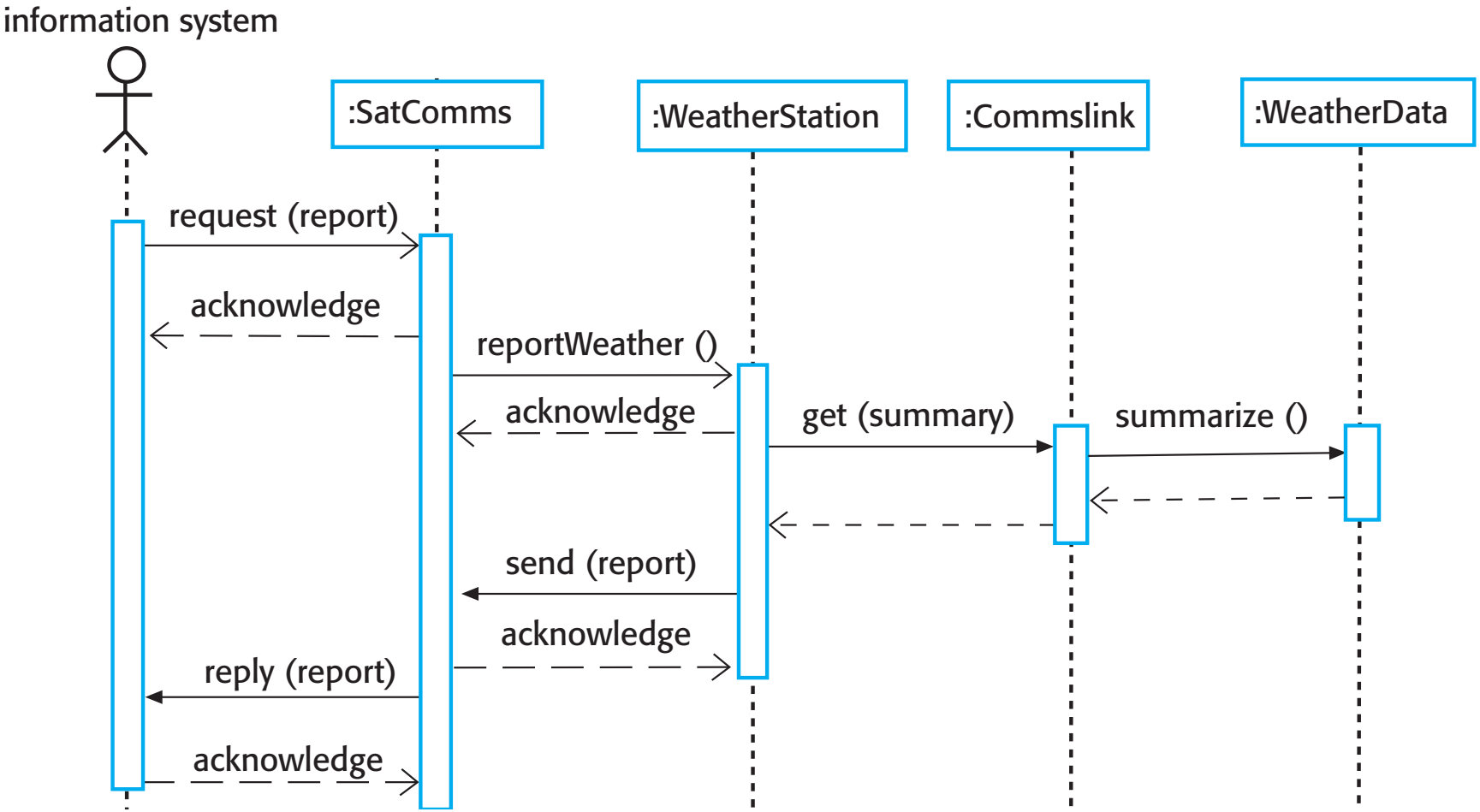
Design models

- Design models show the objects and object classes and relationships between these entities.
- There are two kinds of design model:
  - Structural 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.
  - Show how the design is organized into logically related groups of objects.
  - In the UML, these are shown using packages - an encapsulation construct. This is a logical model, not an actual organization of objects.
- Sequence models that show the sequence of object interactions.
  - 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.
- Also state machine models and other models including use-case models, aggregation models, generalization models, etc.

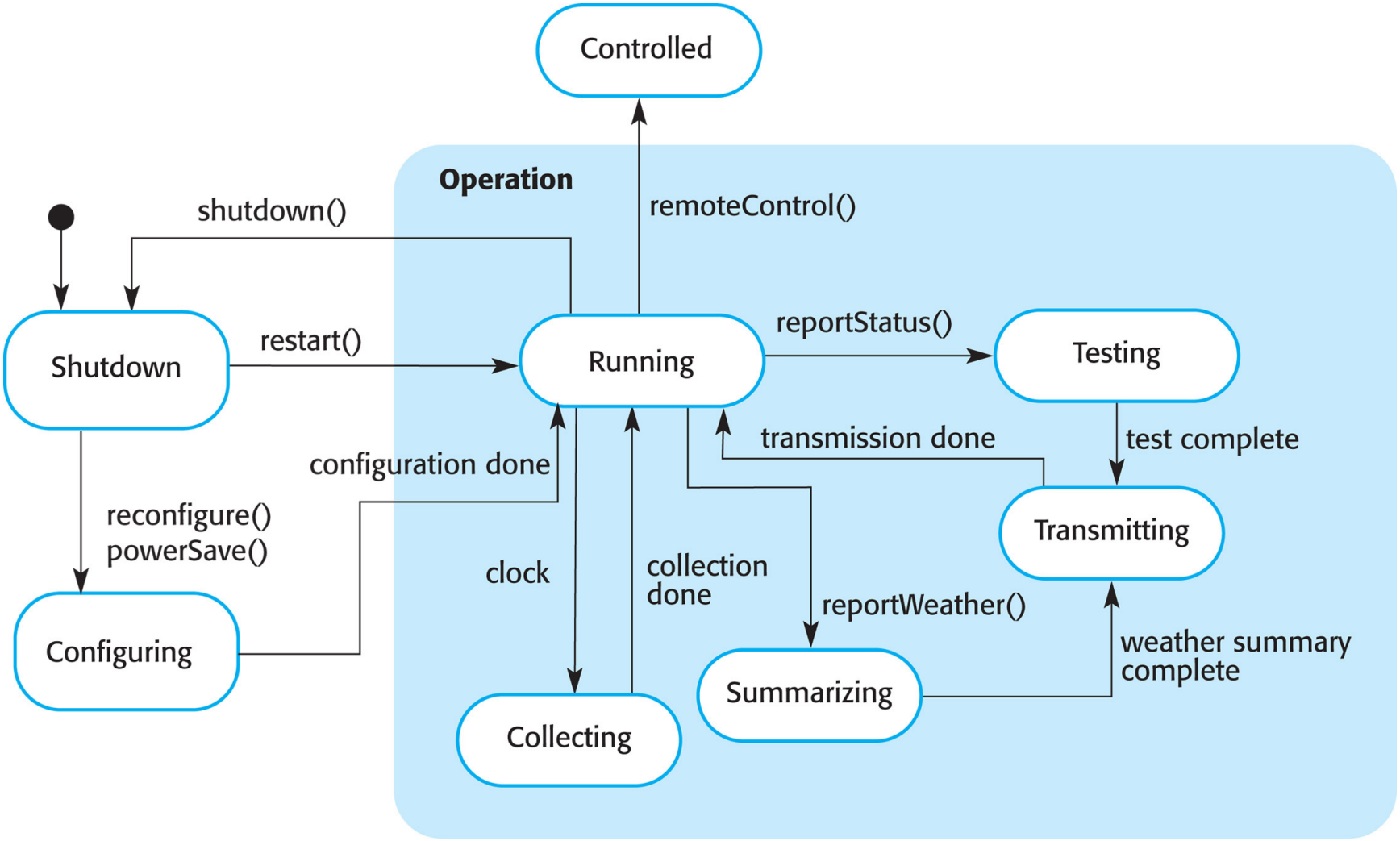
Sequence diagram describing data collection



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 usually don't 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.
- State models are very useful in modeling systems that monitor and respond to events.

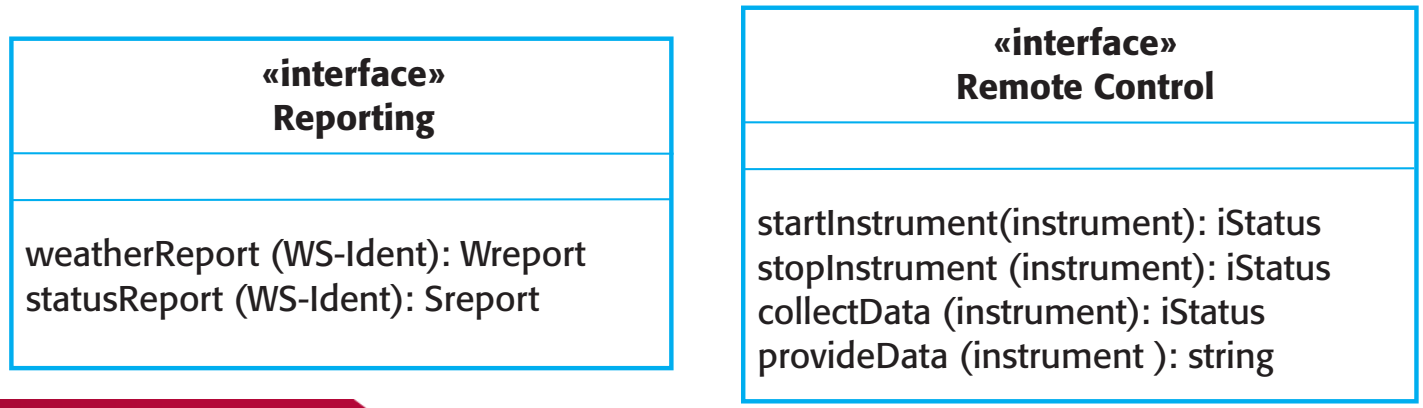
Weather station state diagram



Interface specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Often, 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 class diagrams are used for interface specification (and sequence diagrams are often used to specify interface detail).

Class diagrams



Design patterns

- Analogous to the architecture patterns discussed last week, 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.
  - Inheritance means that any sub-class can inherit the attributes and behaviors of the more general class (or classes). What is good for the parent is good for the child!

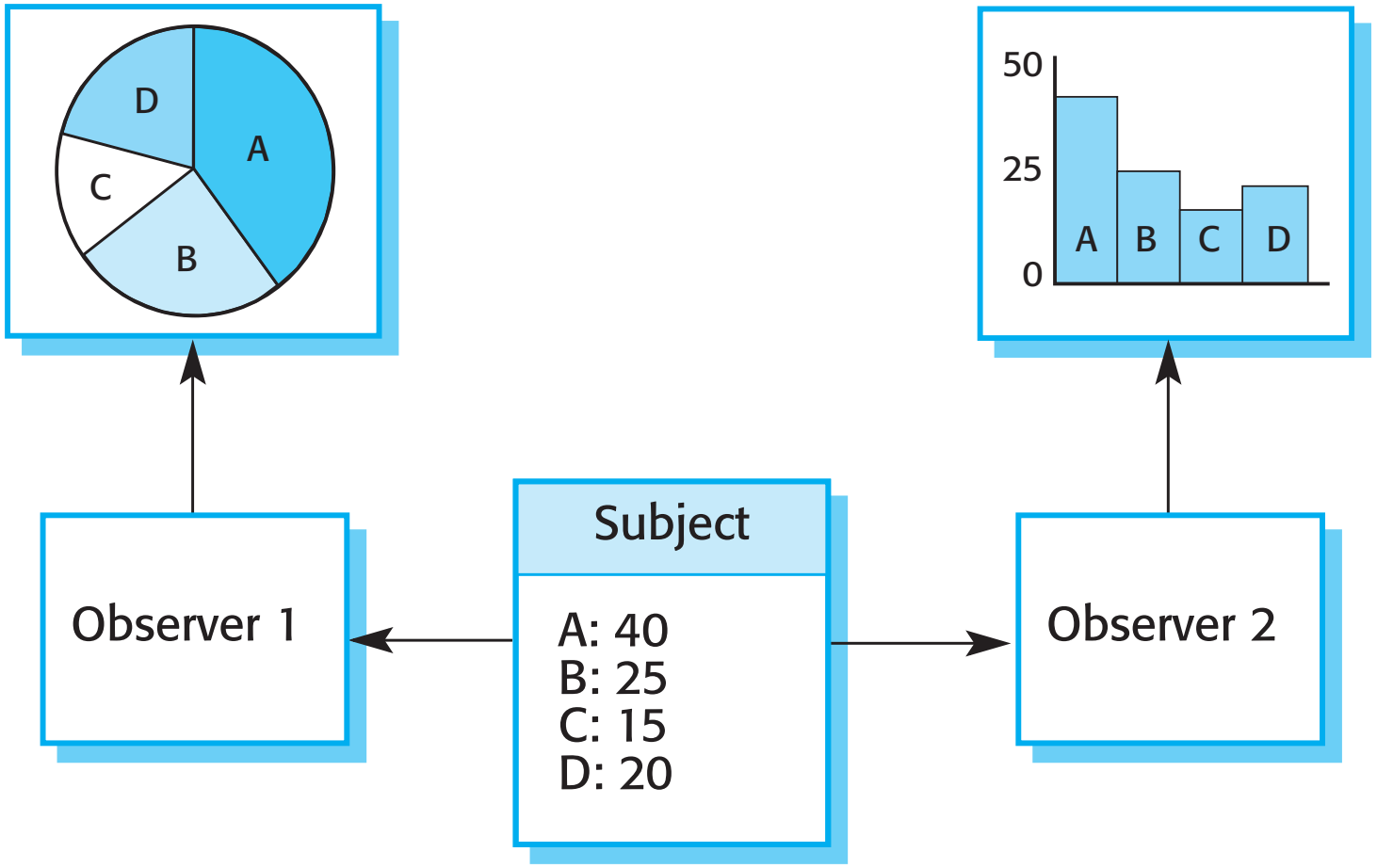
Patterns and pattern elements

- *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.*
- Pattern Elements include:
  - 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.

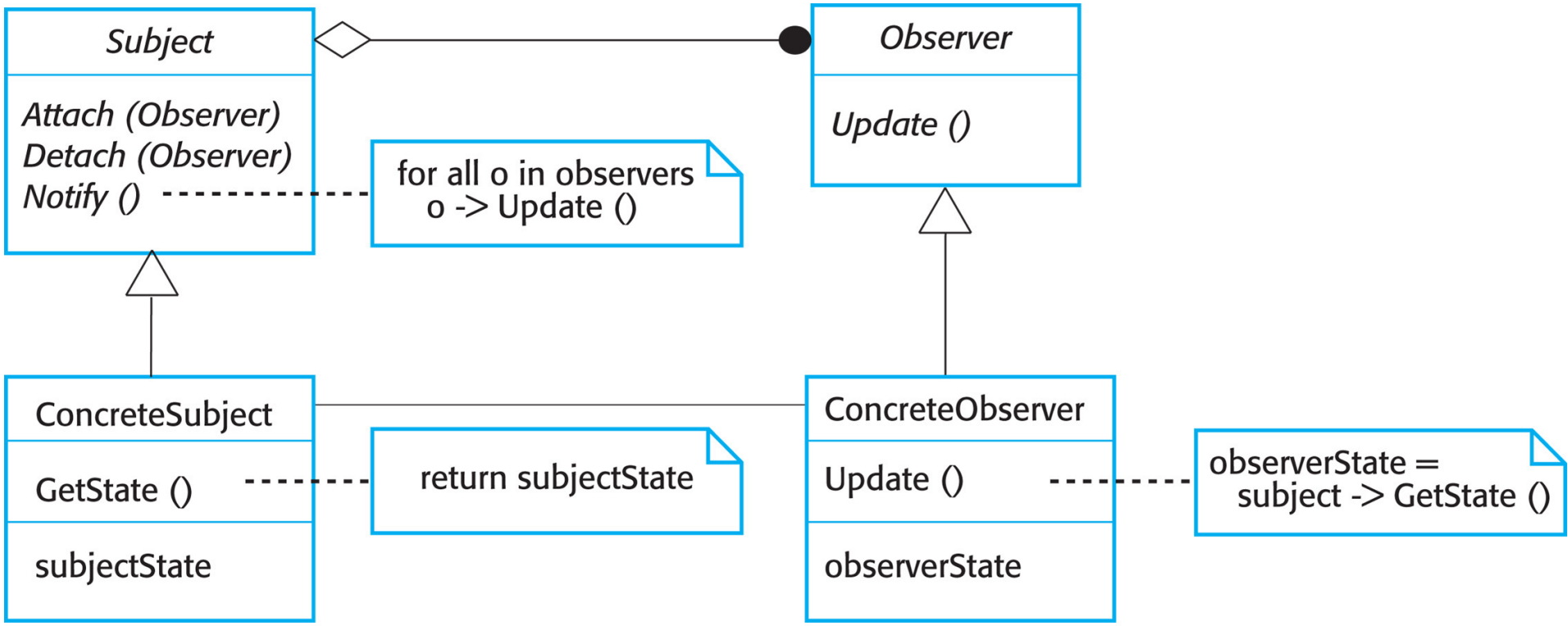
Observer

- Pattern Name - observer, aka dependents, publish-subscribe
- Problem - Problem of maintaining consistency between/among related objects.
- Solution - a subject (object being observed) and one or more observers are implemented and all observers are notified when subject changes state, and then each observer synchronizes with subject. Subject is publisher, observers subscribe to the notifications.
- Consequences - can vary subjects and observers independently, loosely coupled, supports broadcast. As more observers subscribe to a subject, it can be costly changing the subject. Also if it is a simple broadcast (stating simply that a change has occurred but not specifying the change), observers may have significant work finding what has changed.

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).
- Many design patterns exist. See catalog at http://c2.com/cgi/wiki?SoftwareDesignPatternsIndex.

Use of design patterns is form of Reuse

- Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
- 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 or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure made this approach 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.
  - Architectural styles and design patterns are reuse at this level.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself, e.g., language libraries such as those in Python
- The component level
  - Components are collections of objects and object classes that you reuse in application systems, e.g., component frameworks.
- The system level
  - At this level, you reuse entire application systems, e.g., COTS.

Reuse costs

Reuse is not “free”. Costs include:

- The time spent looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the cost of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The expense 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.
- The added costs of maintaining code you didn’t write

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.
- 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.
- Configuration management is a critically important part of such implementations, especially
  - Version management
  - System integration (builds)
  - Change management

Using open source software

- 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
- Increasingly, we see software product companies and large corporations using an open source approach to development.
  - Sometimes their business model is not reliant on selling a software product but on selling support for that product.
- In considering an implementation strategy, we ask:
  - 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 development

- Multiple models exist.
  - Some current open source software was proprietarily developed and then released to the open source communities for continued development and maintenance.
  - Other open source software is the the result of explicitly open development from the start.
- Commercial use of open source arises from the belief that involving the open source community…
  - Allows software to be developed more cheaply,
  - Allows software to be developed more quickly,
  - Creates a community of users for the software.
- Competitive advantage arises not from the software but from the way in which the software is used.

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 one or more individuals) 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.
- Different licenses include differing levels of restriction – know the license BEFORE you invest effort in using the solution!!!

Some of the 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 as long as you include the University of California copyright.

Popularity of open source

- The use of open source in the internet (web servers, networking technologies) has promoted its use in many companies
- Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Anything as a Service (XaaS) offerings (the latter being, in essence, cloud computing) has given impetus to the movement away from proprietary software solutions
- Business models exist to fit most needs
  - Proprietary software on an open-source platform
  - Open-source software embedded within your proprietary software
  - Open-source software acquired from and supported by a third party
  - Open-source software used and supported in house
- Involvement in open source communities can be independent of the models

Classroom Activity – within your EMSS project teams

Activity:

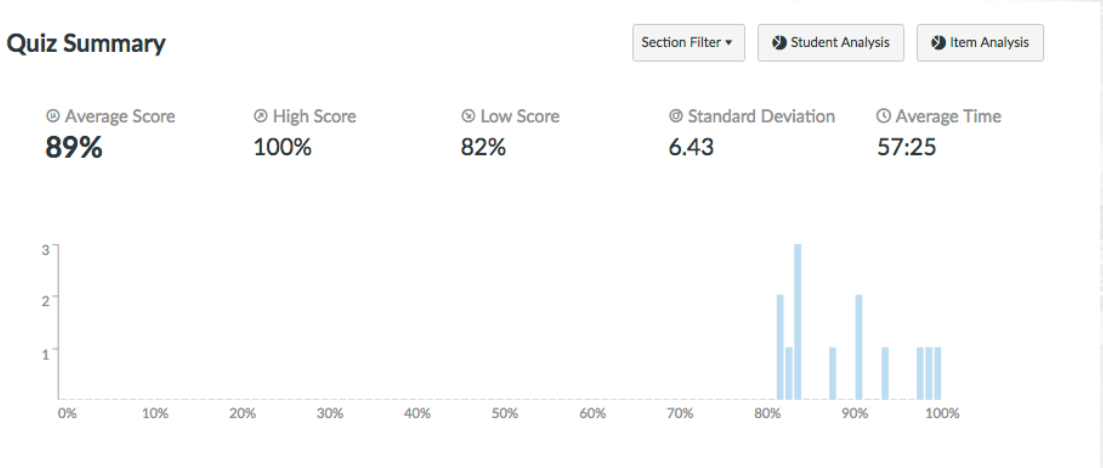
Based on the (use case) diagrams submitted in Week 4 and the architectural patterns of week 5, identify the design structure of your system by defining software object classes and their relationships to one another.

Submission to complete the Canvas Assignment EMSS Week 6:

Submit a UML Class diagram that shows your defined object classes and their relationships to one another.

STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Quiz 2 Results



One question worth reviewing:

What is the most appropriate architectural style or pattern for each of these types of software systems?

Monitoring systems

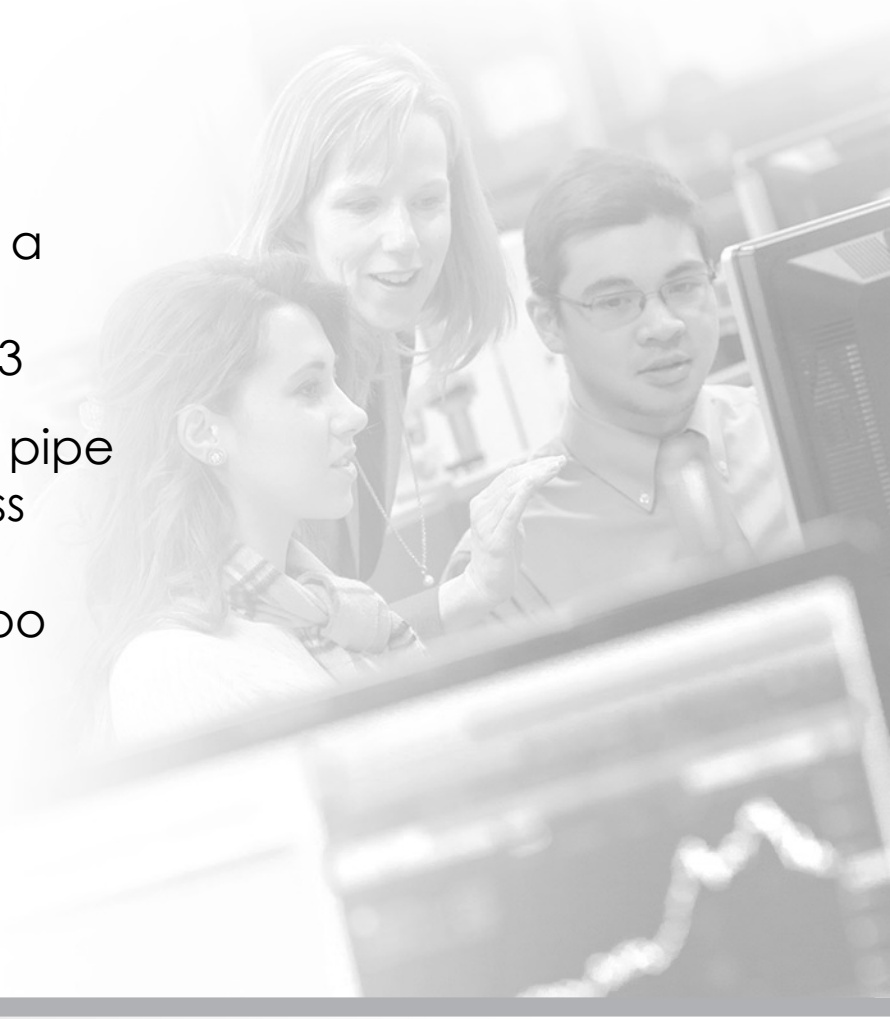
Observe and React Pattern

- Monitoring is an event driven activity
- Environmental Control pattern could also be correct but that was not an option

High Speed data acquisitions system

Process pipeline pattern

- "An example of a system that may use a process pipeline is a high-speed data acquisition system." Sommerville, p. 553
- Several respondents confused this with pipe and filter style; difference is that process pipeline includes buffers for the data approaching their consumer process too quickly to be consumed on arrival.



Transaction Processing System

Pipe and filter style

- Section 6.4.1 of the text describes transaction processing systems
- "Transaction processing systems may be organized as a 'pipe and filter' architecture …Sommerville, p. 167.

Integrated programming support system

Repository style

- Figure 6.9 of the text is a repository architecture for an IDE (integrated development environment).

Resource allocation system

Layered architecture

- A resource allocation system is an information system that typically uses the 4 layer generic structure described in chapter 6.
- Example: a theater ticket management system where a fixed amount of a resource (tickets) is allocated to requesting users, or a library system managing the lending of books (see figure 6.7 in the text)