Software

Design and Implementation

ดัดแปลงจาก slides ของหนังสือ Software Engineering [1]

หัวข้อที่จะศึกษา

- Object-oriented design using the UML
- Design patterns
- Implementation issues
- Opensource development

Design and implementation

- การออกแบบและสร้างซอฟต์แวร์เป็นขั้นตอนหนึ่งในกระบวนการวิศวกรรมซอฟต์แวร์
  - เป็นกระบวนการที่ทำให้เกิดซอฟต์แวร์ที่ใช้งานได้จริง (executable software) ขึ้นมา
- การออกแบบและการสร้างซอฟต์แวร์เป็นกระบวนการที่ต้องดำเนินการควบคู่กันเสมอ
- การออกแบบซอฟต์แวร์ (software design) เป็นกิจกรรมที่สร้างสรรค์ หมายความว่าต้องสร้างสิ่งที่ยังไม่มีอยู่
  - เราต้องสร้างส่วนประกอบซอฟต์แวร์และจัดการความสัมพันธ์ของส่วนประกอบเหล่านั้น ตามความต้องการของลูกค้า
- การดำเนินการสร้างซอฟต์แวร์ (software implementation) เป็นกระบวนการทำให้ แบบกลายเป็นโปรแกรม
  - ต้องใช้ความสามารถด้านการ programming

Build or buy

- โดยส่วนใหญ่แล้ว ในหลาย ๆ โดเมนเราสามารถซื้อระบบที่พัฒนาเสร็จแล้วที่สามารถปรับแต่งตามความต้องการของผู้ใช้
  - ตัวอย่างเช่น ถ้าลูกค้าต้องการใช้ระบบเวชระเบียนในโรงพยาบาล เราสามารถซื้อแพคเกจที่ใช้ในโรงพยาบาลอื่น ๆ มาดัดแปลง
  - อาจจะถูกกว่าและเร็วกว่าที่จะพัฒนาระบบขึ้นมาใหม่ทั้งหมด
- ถ้าเราเลือกวิธีการพัฒนาในลักษณะนี้ งาน implementation จะเทไปทางด้าน configuration management แทนที่จะเป็น programming
  - การเก็บ requirement ก็อาจจะต่างออกไป

Object-oriented design using the UML

An object-oriented design process

- กระบวนการออกแบบเชิงวัตถุ มีโมเดลต่าง ๆ ให้ใช้งานจำนวนมาก
- ในการพัฒนาและบำรุงรักษาโมเดลเหล่านั้นจะต้องใช้ความพยายามและทรัพยากรเป็นจำนวนมาก
  - อาจไม่คุ้มค่าสำหรับระบบขนาดเล็ก ๆ
- สำหรับระบบขนาดใหญ่ที่พัฒนาขึ้นโดยกลุ่มคนจำนวนมาก การออกแบบโมเดลถือเป็นกลไกการสื่อสารที่สำคัญ ที่จะนำไปสู่ความสำเร็จ

Process stages

- ในการออกแบบระบบ มีกระบวนการที่แตกต่างกันหลายรูปแบบ ขึ้นอยู่กับวัตถุประสงค์ของงาน
- โดยทั่วไป กิจกรรมในกระบวนการจะประกอบด้วย
  - การกำหนดบริบทและรูปแบบการใช้งานระบบ
  - การออกแบบสถาปัตยกรรมระบบ
  - การระบุวัตถุหลักในระบบ
  - การออกแบบและพัฒนาแบบจำลอง
  - การกำหนดการเชื่อมโยงความสัมพันธ์ระหว่างวัตถุในระบบ

System context and interactions

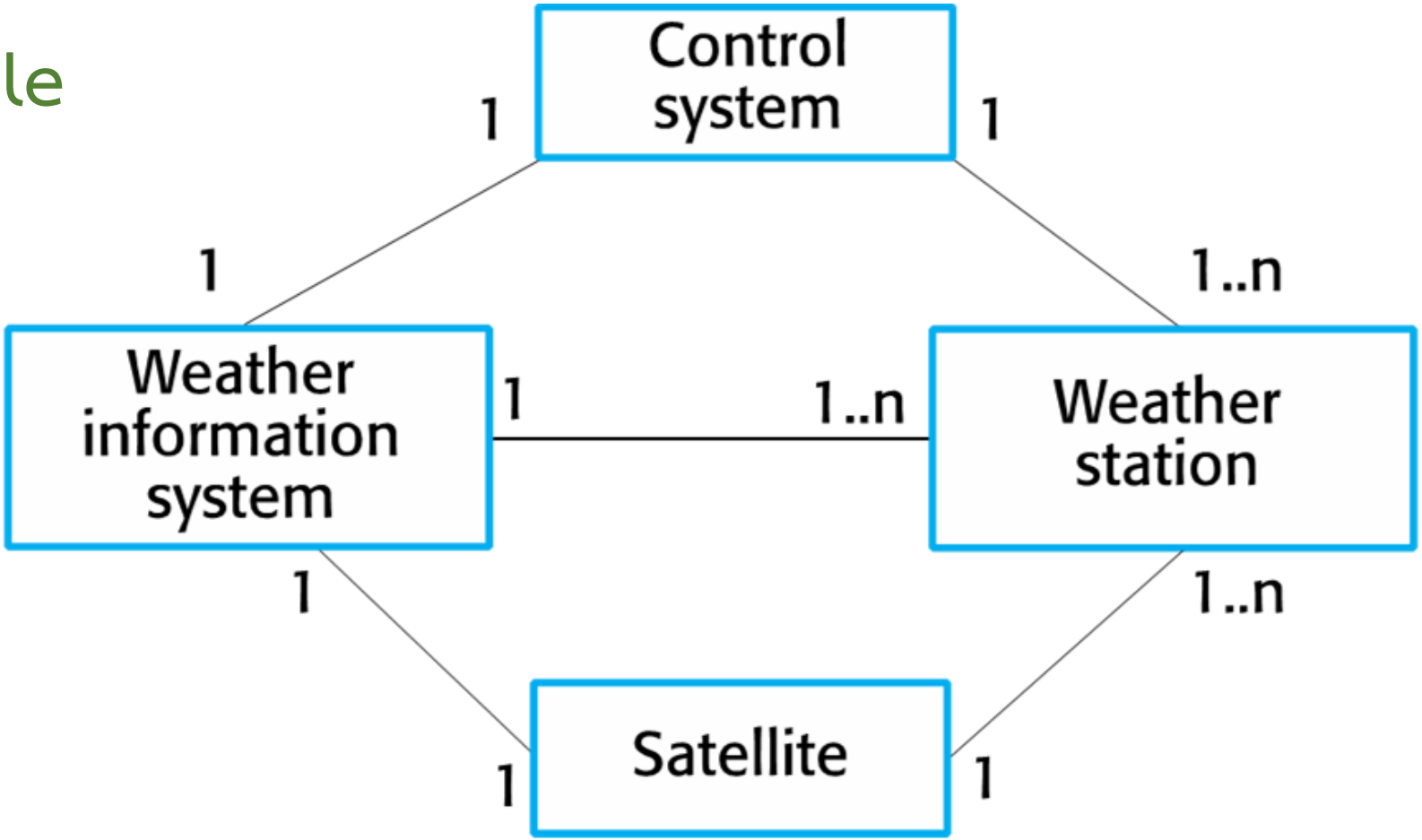
- ทำความเข้าใจความสัมพันธ์ระหว่างซอฟต์แวร์ที่ออกแบบและสภาพแวดล้อมภายนอก
  - เป็นสิ่งสำคัญสำหรับการตัดสินใจว่าจะสร้างระบบที่มีการทำงานอย่างไร
  - สามารถกำหนดวิธีจัดโครงสร้างระบบเพื่อสื่อสารกับสภาพแวดล้อม
- ทำความเข้าใจบริบทของระบบ
  - ช่วยให้สามารถกำหนดขอบเขตของระบบ
  - ช่วยให้ตัดสินใจได้ว่า จะบรรจุคุณลักษณะใดในระบบที่ออกแบบและใช้งานคุณลักษณะใดจากระบบอื่นที่เกี่ยวข้อง

Context and interaction models

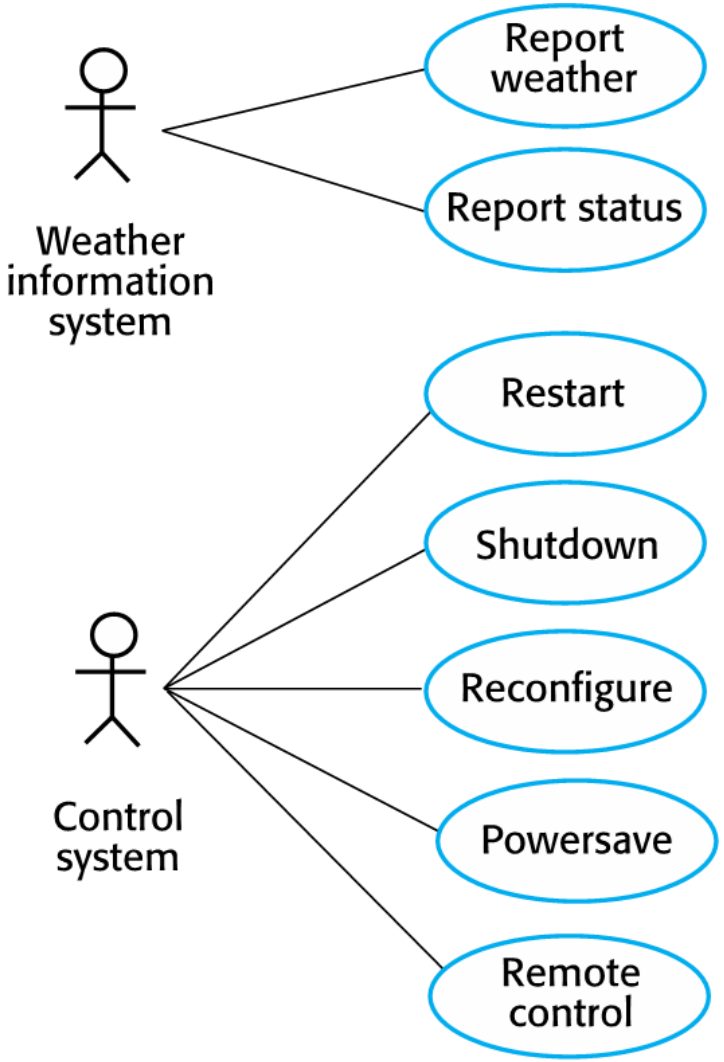
- แบบจำลองบริบทของระบบ (context model)
  - เป็นแบบจำลองโครงสร้างที่แสดงให้เห็นถึงระบบอื่น ๆ ที่อยู่ในสภาพแวดล้อมของระบบที่พัฒนา
- แบบจำลองการโต้ตอบ (interaction model)
  - เป็นแบบจำลองไดนามิก ที่แสดงให้เห็นว่าระบบมีปฏิสัมพันธ์กับสภาพแวดล้อมในขณะต่าง ๆ อย่างไร

System context for the weather station

Example



Weather station use cases



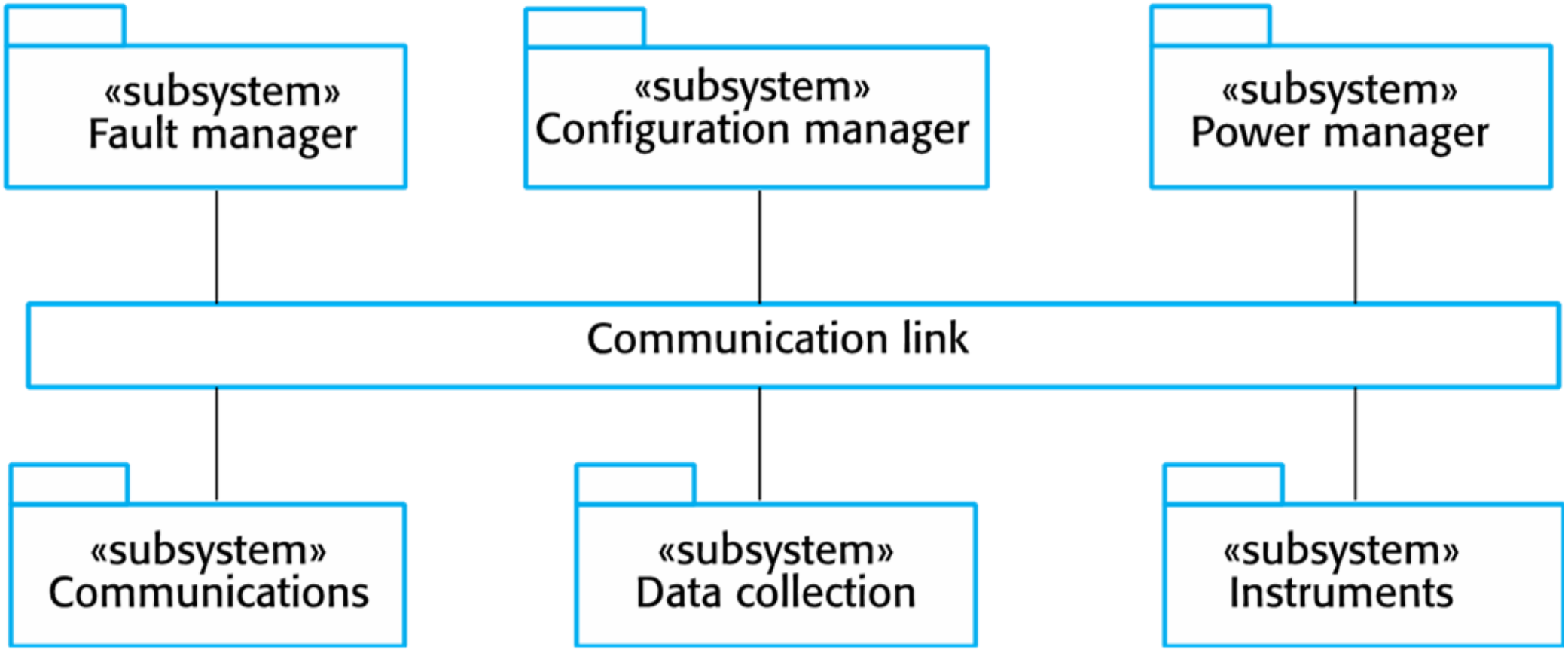
Use case description—Report weather

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.

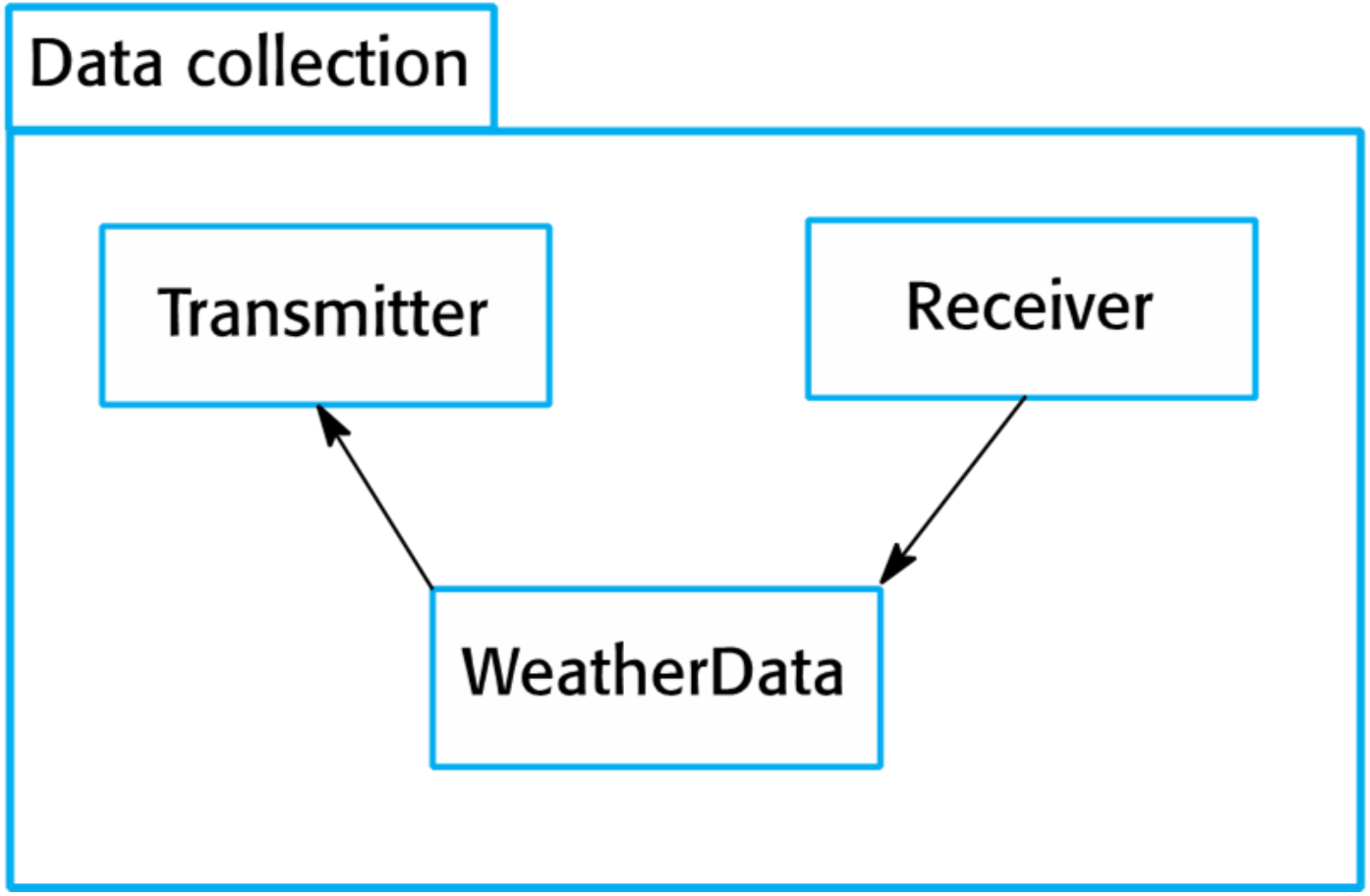
Architectural design

- เมื่อเข้าใจปฏิสัมพันธ์ระหว่างระบบกับสิ่งแวดล้อมแล้ว เราสามารถใช้ข้อมูลนี้ในการออกแบบสถาปัตยกรรมระบบ
  - เริ่มจากการระบุส่วนประกอบสำคัญ ๆ ที่ประกอบกันเป็นส่วนหนึ่งของระบบและปฏิสัมพันธ์ระหว่างองค์ประกอบเหล่านั้น
  - จากนั้นอาจจัดองค์ประกอบต่าง ๆ โดยใช้รูปแบบสถาปัตยกรรมที่เป็นมาตรฐาน เช่น แบบเลเยอร์ (layer) หรือ แบบไคลเอ็นต์เซิร์ฟเวอร์ (client-server)
- สถานีอากาศประกอบด้วยระบบย่อยอิสระ ที่สื่อสารโดยการส่งข้อความ (message broadcasting)

High-level architecture of the weather station



Architecture of data collection system



Object class identification

- การระบุ object class มักเป็นส่วนที่ยากในการออกแบบเชิงวัตถุ
- ไม่มี 'สูตรสำเร็จ' สำหรับการออกแบบวัตถุ มันขึ้นอยู่กับทักษะประสบการณ์และความรู้เกี่ยวกับโดเมนของนักออกแบบระบบ
- การระบุวัตถุเป็นกระบวนการซ้ำซ้อน ที่ต้องทำซ้ำจนกว่าจะเป็นที่น่าพอใจ
  - ไม่มีใครที่จะทำได้สำเร็จอย่างงดงามได้ในครั้งแรก หรือเพียงรอบเดียว

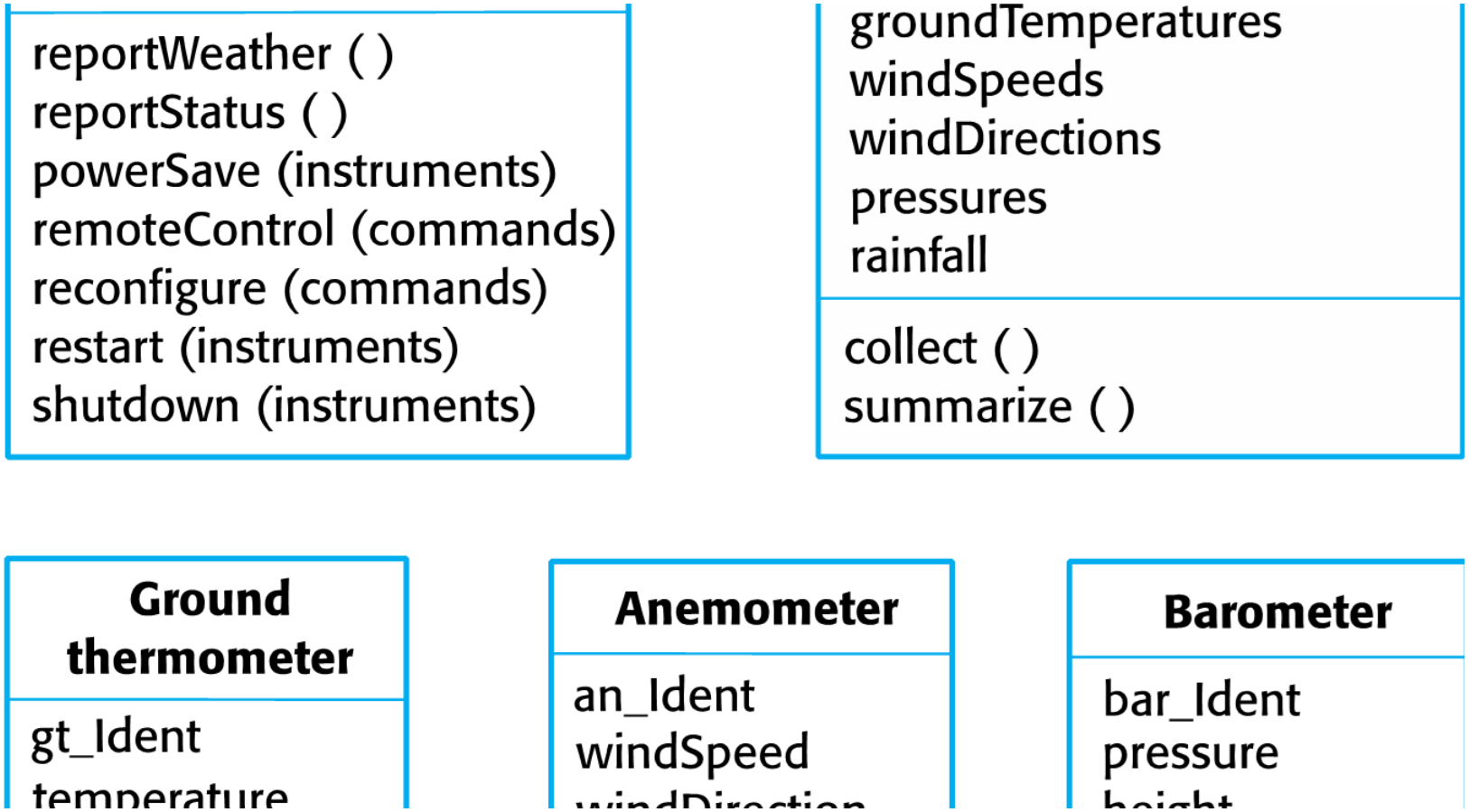
Approaches to identification

- อธิบายระบบ โดยใช้วิธีการทางไวยากรณ์ตามภาษาธรรมชาติ
- อธิบายวัตถุ โดยใช้สิ่งที่จับต้องได้ในโดเมนเป็นจุดอ้างอิง
- อธิบายพฤติกรรม ระบุวัตถุตามการมีส่วนร่วมในพฤติกรรมนั้น
- วิเคราะห์สถานการณ์ แล้วระบุ object, attribute และ method ตามแต่ละสถานการณ์

Weather station object classes

- การระบุ object class ในระบบสถานีวัดอากาศอาจพิจารณาจากฮาร์ดแวร์ที่จับต้องได้รวมทั้งข้อมูลในระบบ เช่น
  - เครื่องวัดอุณหภูมิภาคพื้นดิน, เครื่องวัดความเร็วลม, บารอมิเตอร์
    - เป็นวัตถุของโดเมนแอ็พพลิเคชันซึ่งเป็น "ฮาร์ดแวร์" ที่เกี่ยวข้องกับเครื่องมือในระบบ
  - สถานีอากาศ
    - ประกอบด้วยส่วนติดต่อพื้นฐานของสถานีอากาศกับสภาพแวดล้อม ดังการโต้ตอบที่ระบุไว้ในแบบจำลอง use case
  - ข้อมูลสภาพอากาศ
    - ประกอบด้วยข้อมูลจากเครื่องมือวัดต่าง ๆ

Weather station object classes



Design models

- design model แสดง object และ object class รวมทั้งความสัมพันธ์ระหว่างสิ่งเหล่านั้น
- design model มีสองรูปแบบ ได้แก่
  - โมเดลแบบโครงสร้าง (structural) อธิบายถึงโครงสร้างแบบคงที่ของระบบในแง่ของคลาสวัตถุและความสัมพันธ์
  - โมเดลแบบไดนามิก (Dynamic) อธิบายปฏิสัมพันธ์แบบไดนามิกระหว่างวัตถุ

Examples of design models

- แบบจำลองระบบย่อยที่แสดงการจัดกลุ่มของ object ในระบบ (subsystem model)
- แบบจำลองแสดงลำดับของการโต้ตอบของวัตถุ (sequence model)
- แบบจำลองที่แสดงให้เห็นว่า object เปลี่ยนสถานะอย่างไรเพื่อตอบสนองต่อเหตุการณ์ (State machine models)
- แบบจำลองอื่น ๆ เช่น use-case models, aggregation models, generalisation models, เป็นต้น

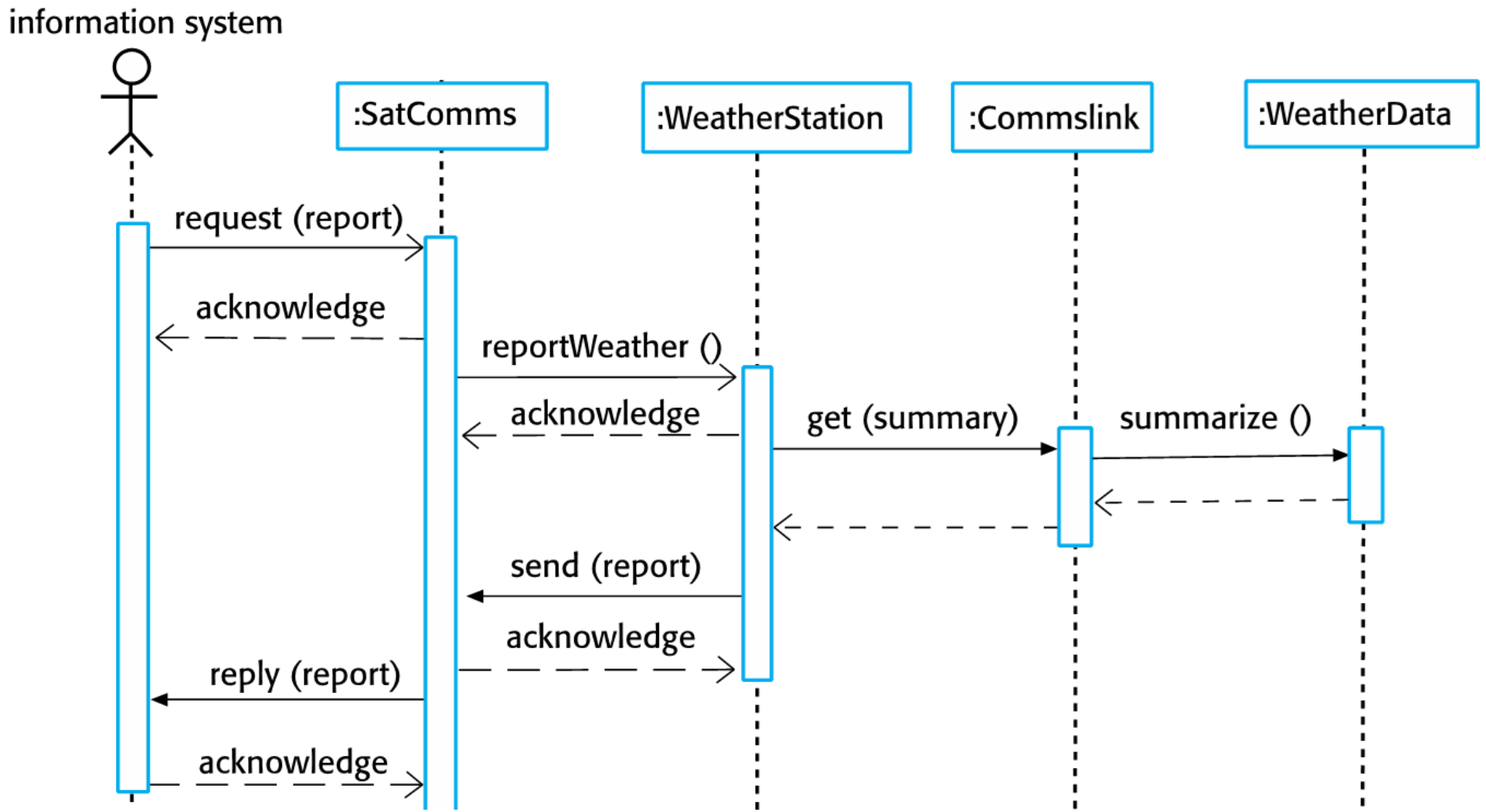
Subsystem models

- แสดงการจัดวางระบบโดยให้ object ที่เกี่ยวข้องกันอยู่รวมกันอย่างเป็นหมวดหมู่
- ใน UML วัตถุเหล่านี้จะแสดงรวมกันเป็นแพคเกจ (package) โดยมีแนวคิด encapsulation เป็นสำคัญ
  - แบบจำลองนี้จะเป็นเชิงตรรกะ
  - ในการจัดองค์ประกอบที่แท้จริงของวัตถุในแต่ละระบบอาจแตกต่างกัน

Sequence models

- แสดงลำดับการโต้ตอบของอ็อบเจ็กต์ที่เกิดขึ้นตามเวลา
  - วัตถุเรียงตามแนวนอนอยู่ด้านบน
  - เวลาจะแสดงในแนวตั้ง แสดงลำดับเหตุการณ์จากบนลงล่าง
  - ปฏิสัมพันธ์แสดงด้วยลูกศรที่มีป้ายกำกับ
    - ลูกศรที่มีรูปแบบแตกต่างกัน แสดงถึงปฏิสัมพันธ์ที่แตกต่างกัน
  - รูปสี่เหลี่ยมผืนผ้าบาง ๆ ในเส้นชีวิตของวัตถุ แสดงถึงระยะเวลาที่วัตถุเป็นตัวควบคุมในระบบ

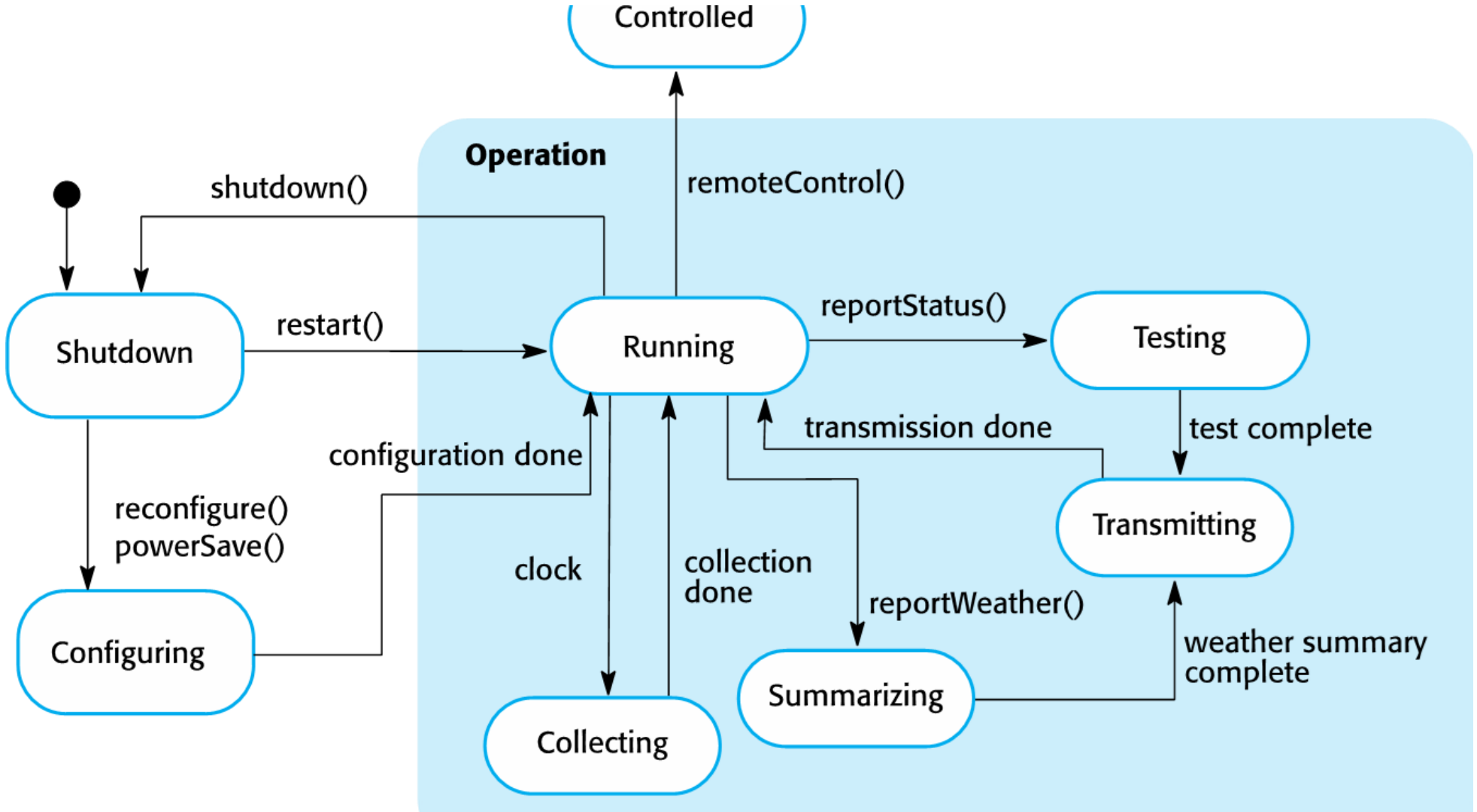
Sequence diagram describing data collection



State diagrams

- แผนผังสถานะ (state diagram) ใช้เพื่อ
  - แสดงวิธีที่วัตถุตอบสนองต่อคำขอบริการต่าง ๆ
  - การเปลี่ยนสถานะที่เรียกใช้โดยคำขอเหล่านั้น
- แผนผังสถานะเป็นโมเดลระดับสูง
  - แสดงพฤติกรรมในขณะทำงานของวัตถุ
- ไม่จำเป็นต้องมีแผนภาพ state diagram สำหรับ object ทั้งหมดในระบบ
  - Object จำนวนมากในระบบมีความเรียบง่าย
  - แบบจำลอง state diagram จะเพิ่มรายละเอียดที่ไม่จำเป็นให้กับการออกแบบ

Weather station state diagram



Design patterns

Design patterns

- Design pattern เป็นวิธีการ reuse ความรู้เชิงนามธรรมเกี่ยวกับปัญหาและแนวทางแก้ไข
- Pattern คือคำอธิบายของปัญหาและสาระสำคัญของการแก้ปัญหา
- Pattern เป็นนามธรรมเพียงพอที่จะนำมาใช้ซ้ำ โดยปรับแต่งเล็ก ๆ น้อย ๆ
- Pattern descriptions มักใช้ประโยชน์จากลักษณะเชิงวัตถุ เช่น inheritance และ polymorphism

Pattern elements

- Name
  - Identifier ของ pattern ที่สื่อความหมายชัดเจน
- Problem description.
- Solution description.
  - ไม่ใช่การออกแบบที่เป็นรูปธรรม
  - เป็นเพียงเทมเพลตสำหรับการออกแบบที่สามารถปรับใช้งานได้ในรูปแบบต่าง ๆ
- Consequences
  - ผลกระทบและเงื่อนไขในการตัดสินใจใช้ pattern นั้น ๆ

The Observer pattern

- Name
  - Observer.
- Description
  - แยกการแสดงสถานะของ object ออกจาก ตัว object
- Problem description
  - ใช้เมื่อจำเป็นต้องแสดงสถานะของ state ได้หลายสถานะ
- Solution description
  - ดูสไลด์หน้าถัดไป
- Consequences
  - การ Optimisations เพื่อเพิ่ม performance ในการแดงผลอาจทำได้ยาก

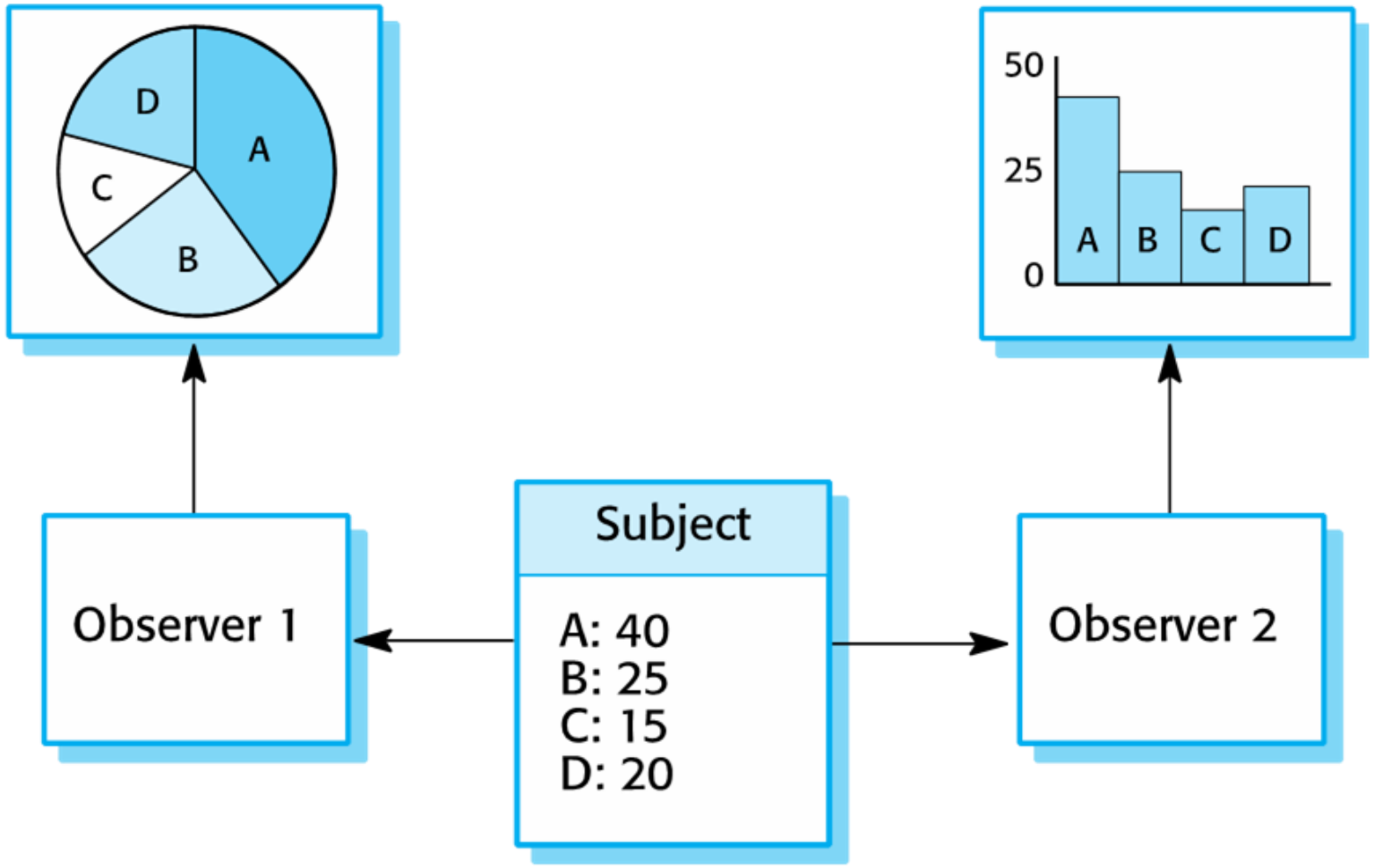
The Observer pattern (1)

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. 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.

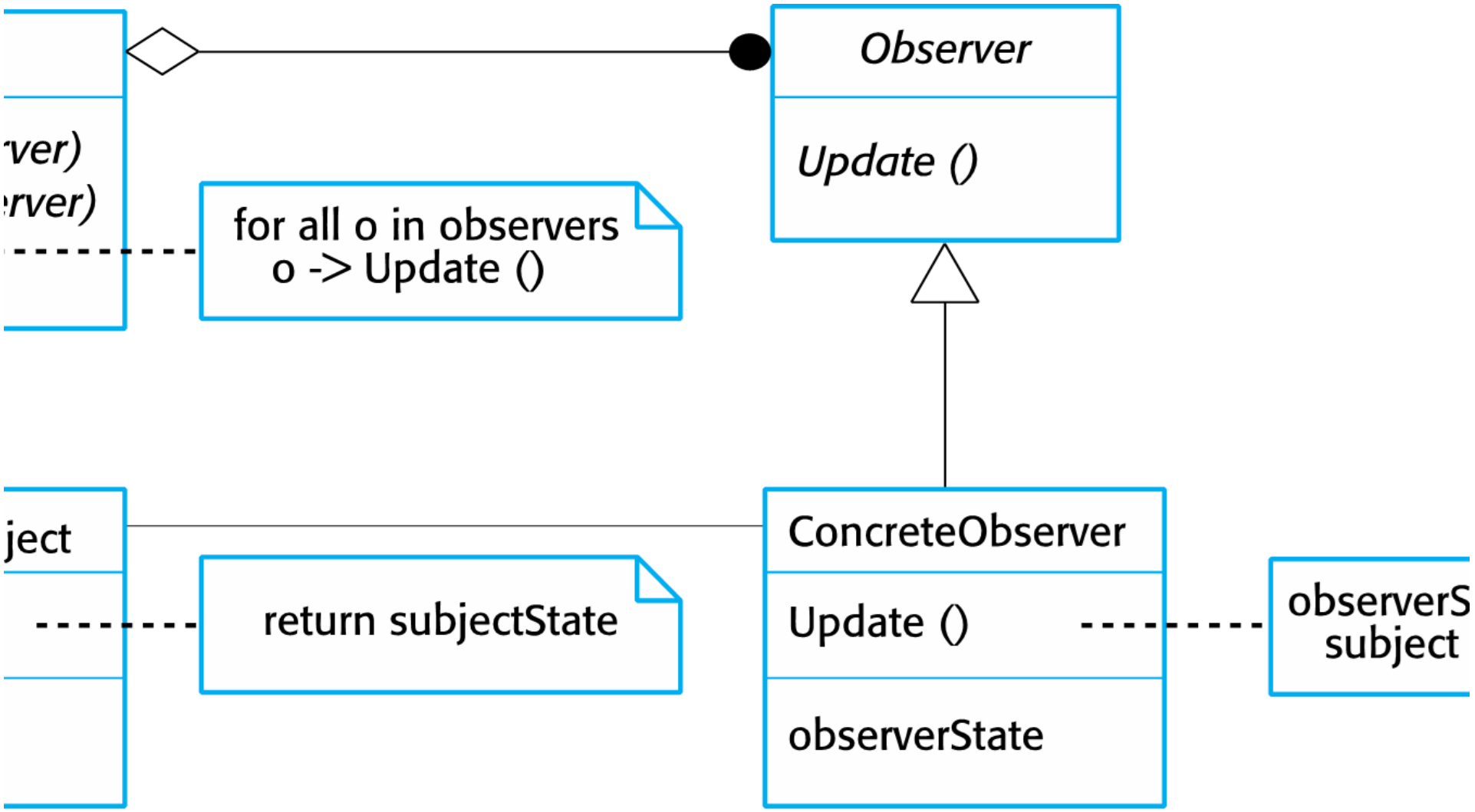
The Observer pattern (2)

Pattern name	Observer
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and 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. 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

- หากต้องการใช้ design pattern ต้องแน่ใจว่าปัญหาด้านการออกแบบที่เรากำลังเจอนั้นมี pattern ที่เกี่ยวข้องซึ่งสามารถนำมาใช้ได้ เช่น
  - บอก object ต่าง ๆ ว่า object หนึ่งมีการเปลี่ยนแปลงสถานะ (Observer pattern)
  - จัดระเบียบส่วนติดต่อกับ object ที่ได้รับการพัฒนาแบบ incremental (Façade pattern)
  - จัดเตรียมวิธีการมาตรฐานในการเข้าถึง element ในคอลเล็กชัน โดยไม่คำนึงถึงว่าคอลเล็กชันดังกล่าวมีการสร้างอย่างไร (Iterator pattern)
  - อนุญาตให้มีการขยายฟังก์ชันการทำงานของคลาสที่มีอยู่ในขณะทำงาน (run-time) (Decorator pattern)

Implementation issues

Implementation issues

- การ implement ไม่ได้หมายถึงการเขียนโปรแกรม (coding) เพียงอย่างเดียว (แม้ว่าการเขียนโปรแกรมจะเป็นสิ่งสำคัญที่ทำให้เกิดซอฟต์แวร์) ยังมีประเด็นการ implement แบบอื่น ๆ ที่มักไม่ครอบคลุมในตำราการเขียนโปรแกรม เช่น
  - Reuse ซอฟต์แวร์ที่ทันสมัยที่สุดในปัจจุบัน ถูกสร้างขึ้นโดยนำส่วนประกอบหรือระบบที่มีอยู่เดิมมาใช้ใหม่
    - เมื่อต้องการพัฒนาซอฟต์แวร์ ควรใช้ code ที่มีอยู่ให้มากที่สุดเท่าที่จะเป็นไปได้
  - Configuration management ในระหว่างขั้นตอนการพัฒนา จะมีหลาย ๆ เวอร์ชันของส่วนประกอบซอฟต์แวร์แต่ละตัวในระบบที่ต้องคอยจัดการการกำหนดค่า
  - Host-target development เป้าหมายในการผลิตซอฟต์แวร์มักไม่ใช้คอมพิวเตอร์เครื่องเดียวกับสภาพแวดล้อมในการพัฒนาซอฟต์แวร์ โดยส่วนใหญ่มักจะมีการพัฒนาบนคอมพิวเตอร์เครื่องหนึ่ง (ระบบโฮสต์) และรันบนคอมพิวเตอร์เครื่องหนึ่ง (ระบบเป้าหมาย)

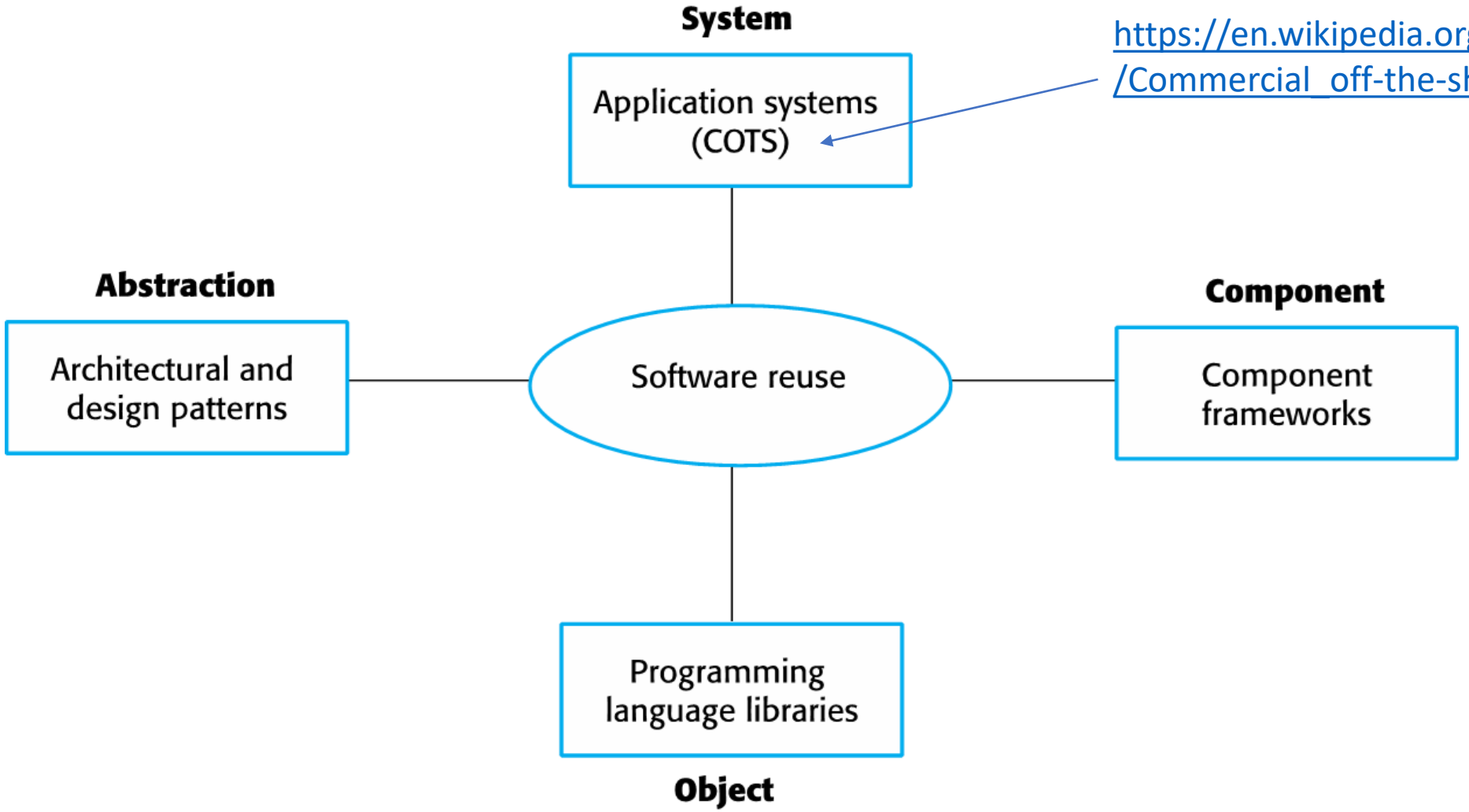
Reuse

- จากทศวรรษที่ 1960 ถึง 1990 ซอฟท์แวร์ใหม่ ๆ ได้รับการพัฒนาตั้งแต่เริ่มต้น (กระดาษเปล่า) โดยการเขียนโค้ดทั้งหมดในภาษาโปรแกรมระดับสูง
  - มีการ reuse ที่สำคัญเพียงอย่างเดียวคือการใช้ฟังก์ชันและไลบรารี
- วิธีนี้ดูเหมือนจะสามารถใช้งานได้น้อยลงเรื่อย ๆ
  - โดยเฉพาะอย่างยิ่งสำหรับระบบเชิงพาณิชย์และอินเทอร์เน็ต
  - แรงกดดันที่สำคัญคือ ค่าใช้จ่ายและเวลาที่จะวางตลาด
- การพัฒนาซอฟต์แวร์โดยอาศัยการ reuse นั้นมีอยู่และเกิดขึ้นมานานแล้วสำหรับซอฟต์แวร์ทางธุรกิจและวิทยาศาสตร์

Reuse levels

- The abstraction level
  - ในระดับนี้เราไม่ได้ reuse ซอฟต์แวร์โดยตรง แต่ใช้ความรู้เกี่ยวกับการออกแบบซอฟต์แวร์ที่ประสบความสำเร็จ
- The object level
  - ในระดับนี้ เรานำวัตถุมาใช้ใหม่จากไลบรารีโดยตรงแทนที่จะเขียนโค้ดด้วยตัวเอง
- The component level
  - component คือชุดของ objects และ classes ที่นำมาใช้ใหม่ใน application systems
- The system level
  - ในระดับนี้เราจะ reuse ระบบแอ็พพลิเคชันทั้งหมด

Software reuse



Reuse costs

- ต้นทุนจากเวลาที่ใช้ในการมองหาซอฟต์แวร์เพื่อ reuse และประเมินว่าเป็นไปตามความต้องการหรือไม่
- ค่าใช้จ่ายในการซื้อซอฟต์แวร์ที่สามารถ reuse ได้
  - สำหรับระบบ COTS (commercial of the shelf) ขนาดใหญ่ อาจมีค่าใช้จ่ายนี้ที่สูงมาก
- ค่าใช้จ่ายในการปรับ (adapt) และกำหนดค่า (configuring) ส่วนประกอบหรือระบบซอฟต์แวร์ที่นำมา reuse ให้ตรงกับความต้องการของระบบที่กำลังพัฒนา
- ค่าใช้จ่ายในการรวม (integrating) ส่วนประกอบซอฟต์แวร์ที่นำมา reuse
  - ถ้าใช้ซอฟต์แวร์จากแหล่งต่าง ๆ ร่วมกับรหัสใหม่ที่เราพัฒนาขึ้น

Configuration management

- Configuration management คือชื่อที่กำหนดให้กับกระบวนการทั่วไปในการจัดการการเปลี่ยนแปลงซอฟต์แวร์
- จุดมุ่งหมายของ Configuration management คือการสนับสนุนกระบวนการรวมระบบ
  - เพื่อให้นักพัฒนาซอฟต์แวร์ทุกคนสามารถเข้าถึงรหัสโครงการและเอกสาร
  - เพื่อให้นักพัฒนาซอฟต์แวร์ตรวจสอบว่ามีการเปลี่ยนแปลงอะไรบ้าง
  - เพื่อให้นักพัฒนาซอฟต์แวร์คอมไพล์และเชื่อมโยงองค์ประกอบเพื่อสร้างระบบ

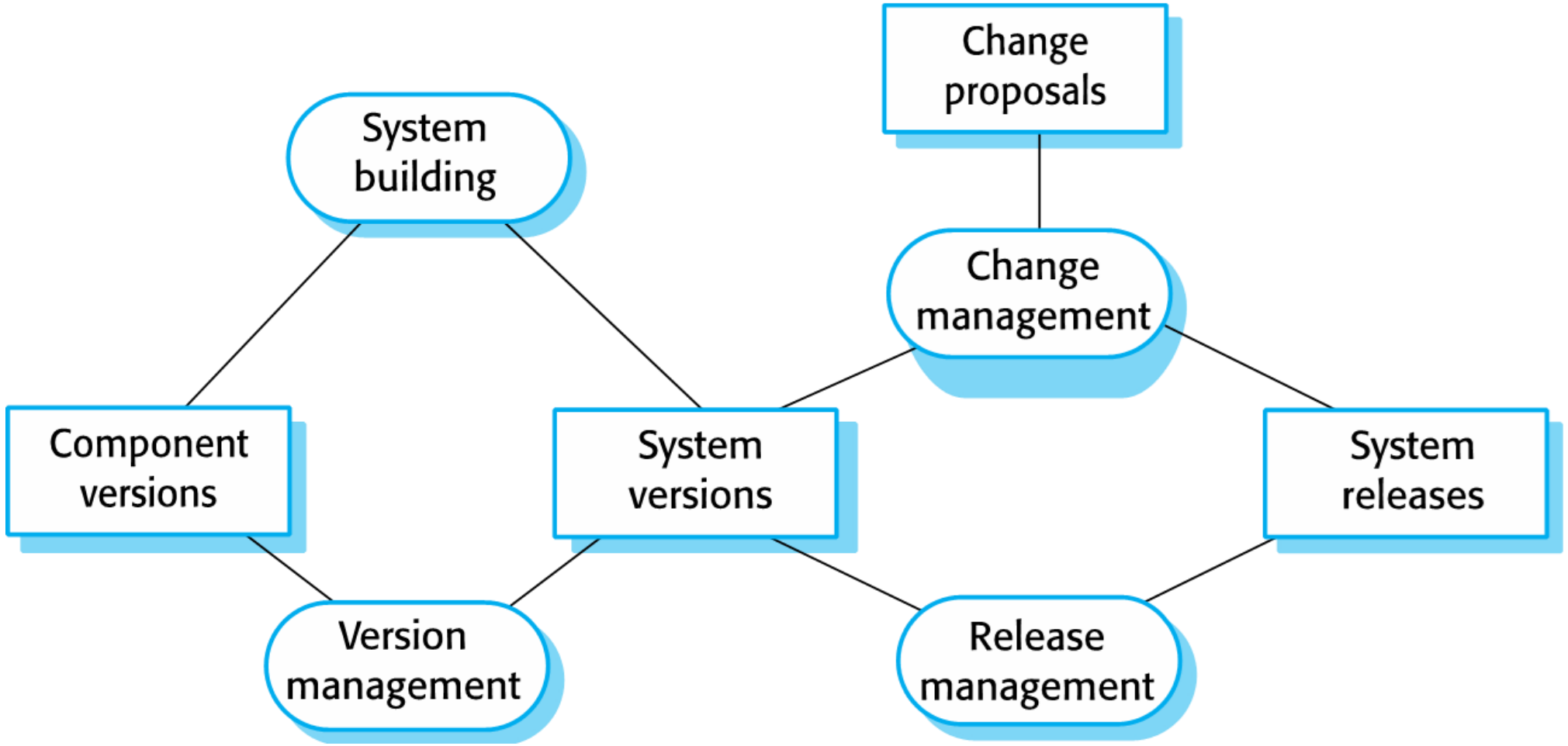
Configuration management activities

- Version management
  - สามารถติดตามส่วนประกอบของซอฟต์แวร์เวอร์ชันต่างๆ
  - ระบบการจัดการเวอร์ชัน ประกอบด้วยสิ่งอำนวยความสะดวกในการร่วมพัฒนาโดยโปรแกรมเมอร์หลาย ๆ คน
- System integration
  - ช่วยให้นักพัฒนาสามารถกำหนดรุ่นของส่วนประกอบที่จะใช้ในการสร้างแต่ละรุ่นของระบบ
  - ใช้เพื่อสร้างระบบโดยอัตโนมัติ โดยการรวบรวมและเชื่อมโยงส่วนประกอบที่จำเป็น

Configuration management activities (ต่อ)

- Problem tracking
  - เพื่อให้ผู้ใช้รายงานข้อผิดพลาดและปัญหาอื่น ๆ
  - เพื่อให้นักพัฒนาซอฟต์แวร์ทั้งหมดสามารถดูว่าใครกำลังทำงานเกี่ยวกับปัญหาเหล่านี้และแก้ไขปัญหาเมื่อใด

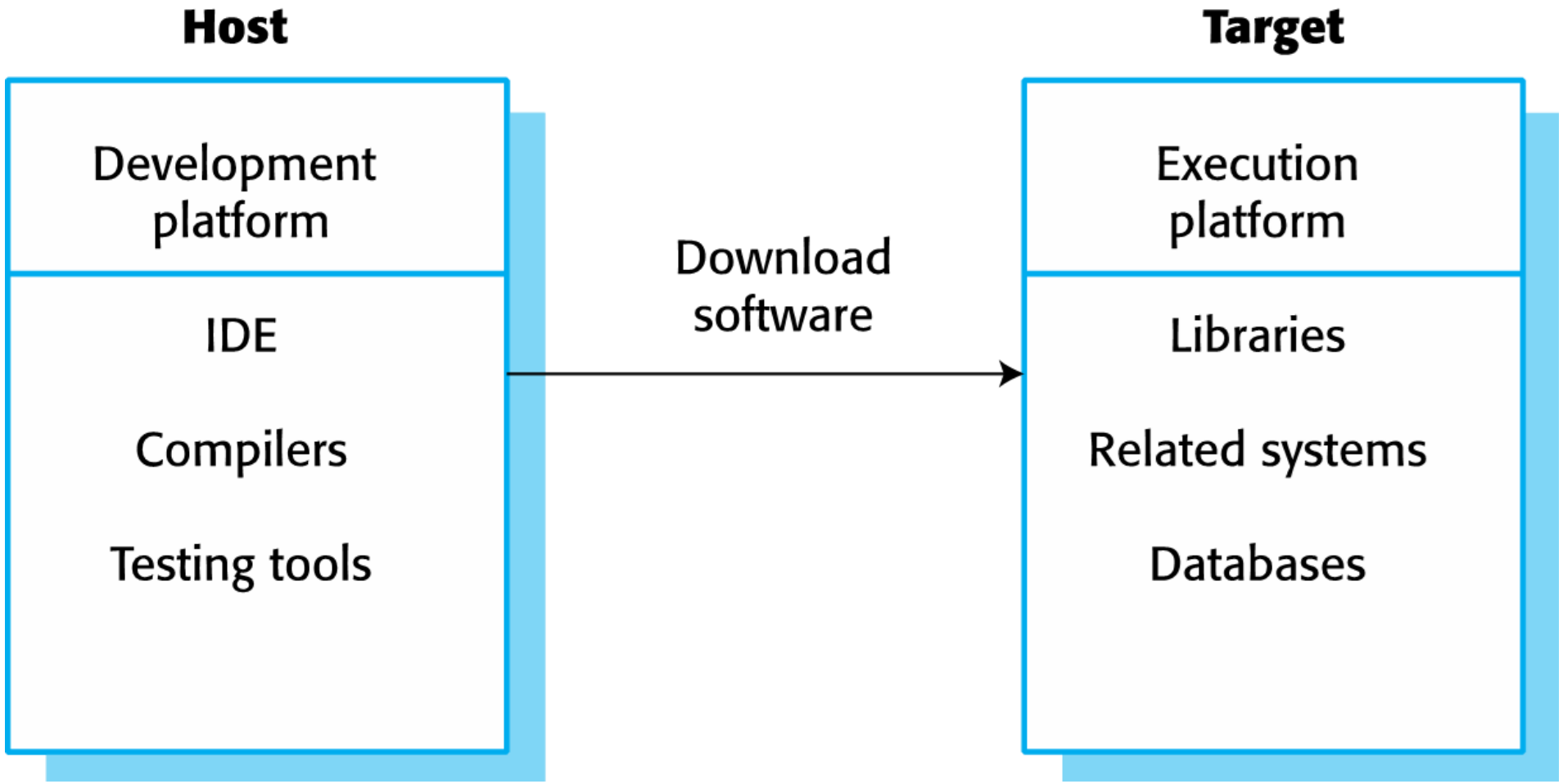
Configuration management tool interaction



Host-target development

- ซอฟต์แวร์ส่วนใหญ่ (แทบทั้งหมด) ได้รับการพัฒนาบนคอมพิวเตอร์หนึ่งเครื่อง (host) แต่ทำงานบนเครื่องอื่น ๆ (target)
- โดยทั่วไปเรามักจะเรียกว่า development platform และ execution platform
  - platform ไม่ได้หมายถึงแค่ ฮาร์ดแวร์
  - platform ประกอบด้วยระบบปฏิบัติการที่ติดตั้งไว้พร้อมทั้งซอฟต์แวร์สนับสนุนอื่น ๆ เช่น database management system, development platforms, interactive development environment เป็นต้น
- development platform มักจะมีซอฟท์แวร์ติดตั้งมากกว่า execution platform
  - platform เหล่านี้อาจมีสถาปัตยกรรมที่แตกต่างกัน

Host-target development



Development platform tools

- ระบบ Integrated compiler และการแก้ไขตามไวยากรณ์ (syntax-directed editing)
  - ช่วยให้สามารถสร้าง แก้ไข และคอมไพล์โค้ดได้
- ระบบดีบักภาษา
- เครื่องมือแก้ไขกราฟิก เช่น เครื่องมือเพื่อแก้ไข UML
- เครื่องมือทดสอบเช่น Junit ที่สามารถรันชุดทดสอบในโปรแกรม version ใหม่ได้โดยอัตโนมัติ
- เครื่องมือสนับสนุนโครงการที่ช่วยจัดระเบียบ code สำหรับ project ต่างๆ

Integrated development environments (IDEs)

- เครื่องมือการพัฒนาซอฟต์แวร์มักถูกจัดกลุ่มเพื่อสร้างสภาพแวดล้อมการพัฒนาแบบรวม (IDE)
- IDE คือชุดเครื่องมือซอฟต์แวร์ที่สนับสนุนด้านต่าง ๆ ของการพัฒนาซอฟต์แวร์
  - มักจะประกอบด้วย common framework และ user interface
- IDEs ถูกสร้างขึ้นเพื่อสนับสนุนการพัฒนาในภาษาการเขียนโปรแกรมเฉพาะ
  - เช่น Java IDE อาจได้รับการพัฒนาขึ้นเป็นพิเศษ
  - หรืออาจเป็น IDE ทั่ว ๆ ไป แล้วมีเครื่องมือสนับสนุนภาษาเฉพาะให้เลือกใช้ (เช่น eclipse, vscode)

Open source development

Opensource development

- การพัฒนา opensource เป็นแนวทางในการพัฒนาซอฟต์แวร์ซึ่งมีการเผยแพร่ซอร์สโค้ดของระบบซอฟต์แวร์และเปิดโอกาสให้อาสาสมัครได้เข้าร่วมในกระบวนการพัฒนา
- จุดเริ่มต้นเกิดจาก Free Software Foundation (www.fsf.org) ซึ่งระบุว่าซอร์สโค้ดไม่ควรเป็นกรรมสิทธิ์ แต่ควรมีให้ผู้ใช้สามารถตรวจสอบและแก้ไขได้ตามต้องการ
- ซอฟต์แวร์ open source ขยายแนวคิดนี้โดยใช้อินเทอร์เน็ตเพื่อรับสมัครนักพัฒนาอาสาสมัครจำนวนมากขึ้น

Opensource systems

- ผลิตภัณฑ์ open source ที่รู้จักกันดีคือระบบปฏิบัติการ Linux ซึ่งใช้กันอย่างแพร่หลายในฐานะระบบเซิร์ฟเวอร์และเป็นสภาพแวดล้อมเดสก์ท็อป
- ผลิตภัณฑ์ open source ที่สำคัญอื่น ๆ ได้แก่ Java, เว็บเซิร์ฟเวอร์ Apache และระบบจัดการฐานข้อมูล mySQL

Opensource issues

- ผลิตภัณฑ์ที่มีการพัฒนาควรใช้ส่วนประกอบ open source หรือไม่?
- ควรใช้วิธี open source เพื่อการพัฒนาซอฟต์แวร์หรือไม่?

Opensource business

- มีบริษัทจำนวนมากขึ้นเรื่อย ๆ ที่กำลังใช้แนวทาง open source ในการพัฒนา
- รูปแบบธุรกิจของพวกเขาไม่ได้ขึ้นอยู่กับการขายผลิตภัณฑ์ซอฟต์แวร์
  - แต่ขายการ support ผลิตภัณฑ์นั้น
- พวกเขาเชื่อว่าการมีส่วนร่วมของชุมชน open source จะช่วยให้ซอฟต์แวร์สามารถพัฒนาได้อย่างรวดเร็ว
  - จะสร้างชุมชนของผู้ใช้ซอฟต์แวร์ได้เร็วยิ่งขึ้น
  - จะขายการ support ได้มากขึ้น

Opensource licensing

- หลักการพื้นฐานของการพัฒนา open source คือซอร์สโค้ดควรมีอิสระในการใช้งาน
  - แต่ไม่ได้หมายความว่าทุกคนสามารถทำทุกย่างที่ต้องการกับซอร์สโค้ดนั้น
- ตามกฎหมาย ผู้พัฒนา code (ทั้งบริษัทหรือบุคคล) ยังคงเป็นเจ้าของโค้ดอยู่
  - พวกเขาสามารถวางข้อจำกัดเกี่ยวกับวิธีการใช้ โดยการรวมเงื่อนไขที่มีผลผูกพันตามกฎหมายในใบอนุญาตซอฟต์แวร์ open source
- นักพัฒนาซอฟต์แวร์ open source บางคนเชื่อว่าหากใช้ส่วนประกอบ open source ในการพัฒนาระบบใหม่ ระบบดังกล่าวควรเป็น open source ด้วย
- นักพัฒนาซอฟต์แวร์บางคนยินดีที่จะอนุญาตให้ใช้ code ของตนโดยไม่มีข้อจำกัด
  - ซึ่งบางครั้งระบบที่พัฒนาแล้วอาจเป็นกรรมสิทธิ์และจำหน่ายในรูปแบบระบบปิด

License models

- GNU General Public License (GPL)
  - หรือใบอนุญาตที่เรียกว่า 'reciprocal' ซึ่งหมายความว่าหากเราใช้ซอฟต์แวร์ open source ที่ได้รับอนุญาตภายใต้ใบอนุญาต GPL เราต้องทำซอฟต์แวร์นั้นให้เป็น open source ด้วย
- GNU Lesser General Public License (LGPL)
  - เป็นรูปแบบใบอนุญาต GPL ที่เราสามารถเขียน code ที่เชื่อมโยงไปยัง open source โดยไม่ต้องเผยแพร่แหล่งที่มาของ component เหล่านั้น

License models (ต่อ)

- ใบอนุญาตจัดจำหน่ายมาตรฐาน Berkley (BSD)
  - เป็นใบอนุญาตแบบ non-reciprocal ซึ่งหมายความว่า ไม่จำเป็นต้องเผยแพร่การเปลี่ยนแปลงหรือแก้ไขใด ๆ ที่ทำกับ open source
  - สามารถใช้ code ในระบบที่เป็นกรรมสิทธิ์ที่ขายได้

License management

- สร้างระบบ เพื่อดูแลรักษาข้อมูลเกี่ยวกับส่วนประกอบ open source ที่ดาวน์โหลดและใช้
- ศึกษาถึงใบอนุญาตประเภทต่าง ๆ และทำความเข้าใจว่า component ที่ได้รับอนุญาตมีการใช้งานอย่างไร
- เฝ้าระวัง-ดูแล เส้นทางการพัฒนาของส่วนประกอบต่างๆ
- ให้ความรู้บุคคลต่าง ๆ เกี่ยวกับ open source
- ให้มีการตรวจสอบระบบได้ ว่ามี open source อยู่หรือไม่
- มีส่วนร่วมในชุมชน open source

Key points

- การออกแบบและการใช้งานซอฟต์แวร์เป็นกิจกรรมที่เกี่ยวเนื่องกัน
  - ระดับของรายละเอียดในการออกแบบขึ้นอยู่กับประเภทของระบบ ไม่ว่าจะใช้วิธีการใด เช่น plan-driven หรือ agile
- กระบวนการของการออกแบบเชิงวัตถุ ประกอบด้วยกิจกรรมในการออกแบบสถาปัตยกรรมระบบ, การระบุออบเจ็กต์ในระบบ, การอธิบายการออกแบบโดยใช้โมเดลวัตถุที่แตกต่างกัน และการจัดทำเอกสารอธิบายการรวมระบบ

Key points

- กระบวนการออกแบบเชิงวัตถุ ทำให้เกิดแบบจำลองที่หลากหลาย เช่น
  - โมเดลแบบ static (ได้แก่ class model, generalization models, association models)
  - โมเดลแบบ dynamic (ได้แก่ sequence models, state machine models)
- ต้องมีการกำหนดวิธีการใช้งานหรือการเชื่อมต่อของ object ต่างๆ อย่างแม่นยำเพื่อให้ object อื่น ๆ สามารถใช้งานมันได้
  - อาจนำ UML มาใช้เพื่ออธิบาย

Key points

- เมื่อพัฒนาซอฟต์แวร์ ควรพิจารณาความเป็นไปได้ที่จะนำซอฟต์แวร์ที่มีอยู่มาใช้ใหม่
  - ไม่ว่าจะเป็น component, service, หรือระบบที่สมบูรณ์
- Configuration management คือกระบวนการของการจัดการการเปลี่ยนแปลงในระบบซอฟต์แวร์ที่พัฒนาขึ้น
  - เป็นสิ่งสำคัญเมื่อมีทีมงานร่วมมือกันพัฒนาซอฟต์แวร์
- การพัฒนาซอฟต์แวร์ส่วนใหญ่เป็นการพัฒนาแบบ host-target development
  - ใช้ IDE บนเครื่องโฮสต์เพื่อพัฒนาซอฟต์แวร์ จากนั้นจะถูกโอนย้ายไปยังเครื่องเป้าหมายเพื่อการใช้งาน

Key points

- การพัฒนา open source เกี่ยวข้องกับการทำให้ source code ของระบบเป็นแบบสาธารณะ
  - ซึ่งหมายความว่าคนทั่วไปสามารถเสนอการเปลี่ยนแปลงและปรับปรุงซอฟต์แวร์

คำถาม???