Lecture 8

Spring Security (2)
Design & Implementation Process

Topics covered

- ✧ Hibernate Validation Basics
- ✧ Spring Security
  - JPA-based authentication
  - User registration
- ✧ Design & Implementation Process
  - Object-oriented design using the UML
  - Implementation issues
  - Open source development

Hibernate Validation

What is Hibernate Validation?

- A an implementation of the Bean Validation specification.
  - There are different versions of Bean Validation: 1.0 (JSR 303), 1.1 (JSR 349), 2.0 (JSR 380), Jakarta Bean Validation 2.0, Jakarta Bean Validation 3.0, Jakarta Bean Validation 3.1.
  - Different Hibernate versions implement different versions of the specification. But the core & basic parts are similar.
- It is used to define validation constraints on Entity objects and handle validation errors.
- To use Hibernate Validation, add this dependency:

```xml
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
```

Validation Constraints

- ✧ Constraints are defined using annotations on your Java bean (Entity/DTO) attributes.
- ✧ Some common constraints:
  - `@NotNull`: Ensures the field is not null.
  - `@Size`: Ensures the field's size is within the specified range.
  - `@Min` and `@Max`: Ensures the field's value is within the specified range.
  - `@Pattern`: Ensures the field matches the specified regular expression.

Hibernate Validation Annotation Example

```java
import jakarta.validation.constraints.*;

public class User {
    @NotNull
    private String name;

    @Size(min = 2, max = 14)
    private String username;

    @Min(18) @Max(100)
    private int age;

    @Pattern(regexp = ".+@.+\\..+")
    private String email;

    // Getters and setters
}
```

Validating Constraints

- The `@Valid` annotation is one of the ways (a convenient one) to validate a Java bean whose attributes are annotated with validation constraints.
  - Validation results are stored in the `BindingResult` object.

```java
@RequestMapping(value = "/save")
public String saveUpdate(@Valid Employee employee,
                         BindingResult result) {
    if (result.hasErrors()) {
        return "employeeUpdate";
    }
    employeeRepository.save(employee);
    return "redirect:/update/" + employee.getId();

}
```

Adding validation error messages

```java
public class User {
    @NotNull(message = "Name cannot be null")
    private String name;

    @Size(min = 2, max = 15,
        message = "Username must be between 2 and 15 chars")
    private String username;

    @Min(value = 18, message = "Age must be at least 18")
    @Max(value = 100, message = "Age must be at most 100")
    private int age;

    @Pattern(regexp = ".+@.+\\..+", message = "Invalid email address")
    private String email;

    // Getters and setters
}
```

Getting error messages

✧ **Step 1:** validate bean & receive the `BindingResult` object in controller method.

```
@PostMapping("/add-user")
public String addUser(@Valid User user,
                      BindingResult result, Model model) {
    if (result.hasErrors()) {
        return "addUserForm";
    }
    model.addAttribute("user", user);
    return "addUserSuccess";
}
```

Retrieving error messages from Thymeleaf

✧ **Step 2:** display error messages in the Thymeleaf view.

```
<form action="#" th:action="@{/add-user}"
      th:object="${user}" method="post">
    <div>
        <label for="name">Name:</label>
        <input type="text" id="name" th:field="*{name}" />
        <div th:if="${#fields.hasErrors('name')}"
             th:errors="*{name}">
            Name Error
        </div>
    </div>
    <!-- code omitted -->
</form>
```

Spring Security
JPA-based Authentication

Spring Security Configuration

- ✧ Instead of providing a `UserDetailsManager`, we can supply a `UserDetailsService` as a source of users.

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http)
        throws Exception {
    return http
            .userDetailsService(jpaUserDetailsService)
            .authorizeHttpRequests(req -> req
                    .requestMatchers("/register").permitAll()
                    .anyRequest().authenticated()
            )
            .formLogin(Customizer.withDefaults())
            .build();
}
```

The “User” concept in Spring Secutity

- Spring Security uses a `UserDetails` interface for performing authentication.

```
public interface UserDetails extends Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();
    String getPassword();
    String getUsername();
    boolean isAccountNonExpired();
    boolean isAccountNonLocked();
    boolean isCredentialsNonExpired();
    boolean isEnabled();
}
```

Creating a `User` entity

- ✧ This entity stores all related user information
  - **Required:** username, password, roles (authorities)
  - Additional: phone number, address, avatar…
- ✧ It is possible to have relationships between this entity and other entities
  - E.g. one-to-many relationship with `Post` entity

`User` entity example

```java
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private String roles; // "ADMIN,USER,MOD"
    private String address;
    @OneToMany(mappedBy = "author") // relationship
    private List<Post> posts;
```

Implementing `UserDetails` for Spring Security

- ✧ **Approach #1:** make `User` entity class implement `UserDetails` interface
  - **Not recommended:** entity-related stuffs (attributes, constructors, getters, setters) will be mixed up with `UserDetails` methods (`getAuthorities()`, `isEnabled()`, `isAccountNonExpired()`…)
  - ➢ Messy code
- ✧ **Approach #2:** create another class named `MyUserDetails` to implement `UserDetails` interface
  - Create a `User` attribute in `MyUserDetails`
  - Supply `MyUserDetails` objects to Spring Security instead of providing `User` objects
  - **Recommended:** `User` class is untouched, separation of concerns is achieved

User implementing UserDetails (messy code)

```java
@Entity
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private String roles;
    private String address;
    @OneToMany(mappedBy = "author")
    private List<Post> posts;
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {...}
    @Override
    public boolean isAccountNonExpired() { return true; }
    @Override
    public boolean isAccountNonLocked() { return true; }
    @Override
    public boolean isCredentialsNonExpired() { return true; }
    @Override
    public boolean isEnabled() { return true; }
    // constructors, getters & setters
```

MyUserDetails **class** (cleaner code)

```java
public class MyUserDetails implements UserDetails {
    private User user;
    public MyUserDetails(User user) { this.user = user; }
    @Override
    public String getUsername() { return user.getUsername(); }
    @Override
    public String getPassword() { return user.getPassword(); }
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        // create & return a List<GrantedAuthority> from roles
    }
    @Override
    public boolean isAccountNonExpired() { return true; }
    @Override
    public boolean isAccountNonLocked() { return true; }
    @Override
    public boolean isCredentialsNonExpired() { return true; }
    @Override
    public boolean isEnabled() { return true; }
}
```

UserTemplate class for Validation

- The problem
  - User password will be encoded
  - Validation is only needed on raw password
  - ➢ Therefore, should not put validation annotation on the `password` field of `User` class
- Solution
  - Create another class for validation user information (the class can be named `UserTemplate`)
  - Create a constructor in `User` to accept a (valid) `UserTemplate` and encode the password, as well as initialize other fields (such as `roles`)

`UserTemplate` class for Validation

- Validation annotations are placed in this class

```java
public class UserTemplate {
    @Length(min = 6, max = 60)
    private String username;
    @Pattern(regexp = "^(?=.*\\d)(?=.*[A-Z]).{6,60}$",
            message = "6 chars min (at least 1" +
                    "digit & 1 uppercase letter)")
    private String password;
    private String address;
    // getter & setter methods
}
```

Showing a registration form

- Use `UserTemplate` instead of `User` when working with a form.

```
@GetMapping("/register")
public String register(Model model) {
    model.addAttribute("user", new UserTemplate());
    return "register";
}
```

Handling registration form submission

```
@PostMapping("/register")
public String registerHandle(Model model, PasswordEncoder encoder,
          @Valid UserTemplate ut, BindingResult result) {
    if (result.hasErrors()) {
        model.addAttribute("user", ut);
        return "register";
    } else {
        userRepository.save(new User(ut, encoder));
        model.addAttribute("user", new UserTemplate());
        model.addAttribute("success", true);
        return "register";
    }
}
```

Creating a UserDetailsService

```
@Service
public class JpaUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepo;

    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        Optional<User> user = userRepo.findByUsername(username);
        if (user.isPresent()) {
            return new SecurityUser(user.get());
        } else {
            throw new UsernameNotFoundException(
                    "User not found: " + username
            );
        }
    }

}
```

Auto-wiring is not recommended

- Auto-wiring uses Java Reflection, which introduces performance overheads (and possibly security issues)
- Dependencies are not exposed explicitly (compared to, for instance, constructor dependency injection)
- What’s recommended?
  - Dependency Injection through constructor parameters is recommended.

Back to the Security Configuration

- Permit all users to access `/register` and create a `PasswordEncoder` bean.

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http)
        throws Exception {
    return http
            .userDetailsService(jpaUserDetailsService)
            .authorizeHttpRequests(req -> req
                    .requestMatchers("/register").permitAll()
                    .anyRequest().authenticated()
            ).formLogin(Customizer.withDefaults())
            .build();
}

@Bean
PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

The generated tables in DB

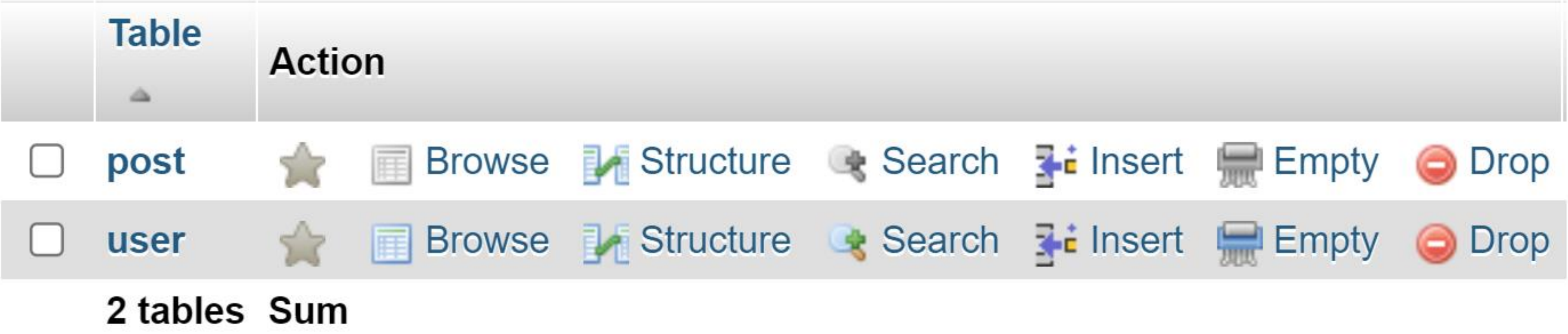


	Table ▲	Action						
□	post	☆	Browse	Structure	Search	Insert	Empty	Drop
□	user	☆	Browse	Structure	Search	Insert	Empty	Drop
	2 tables	Sum						

Design & Implementation Process

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 **c**ommercial **o**ff-**t**he-**s**helf (COTS) systems 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.

Object-oriented design using 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 and, for small systems, this may not be cost-effective.
- ✧ However, for <u>large</u> systems developed by <u>different groups</u>, 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.

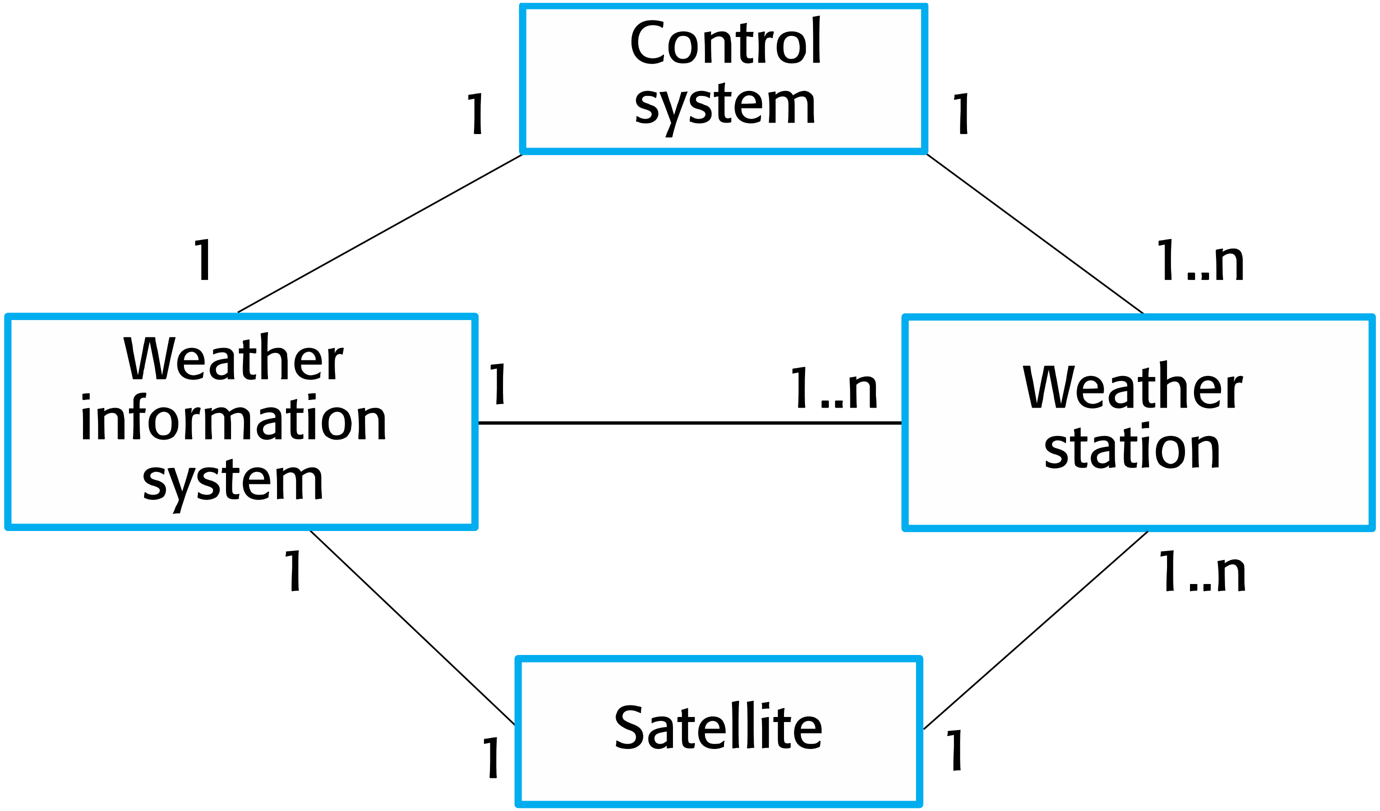
System context and interactions

- The relationships between the software that is being designed and its external environment.
- 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.

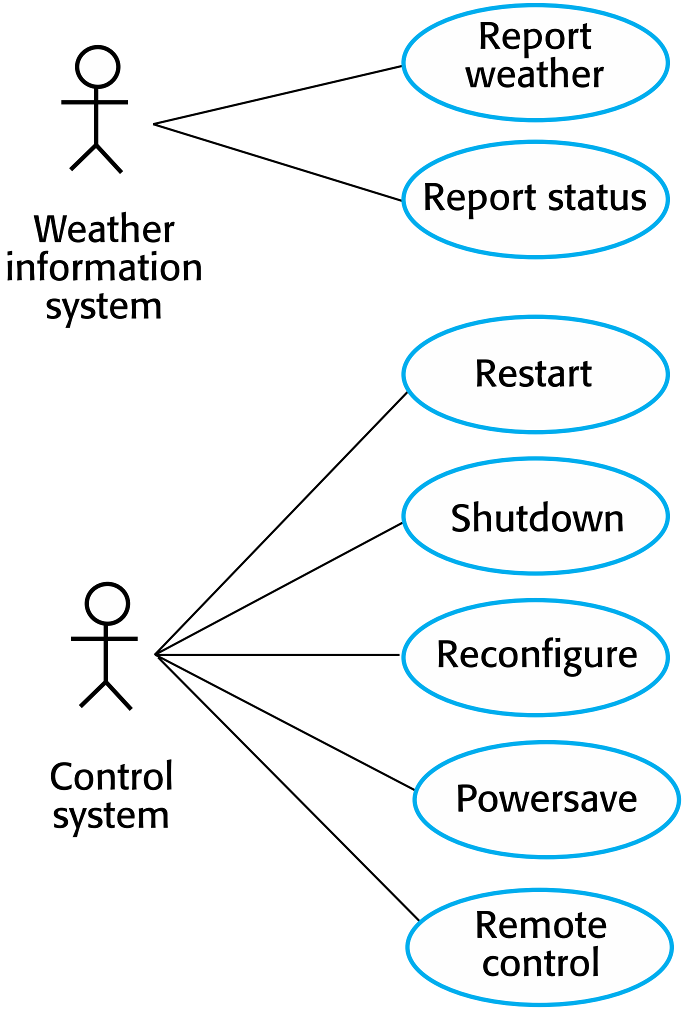
Context and interaction models

- 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



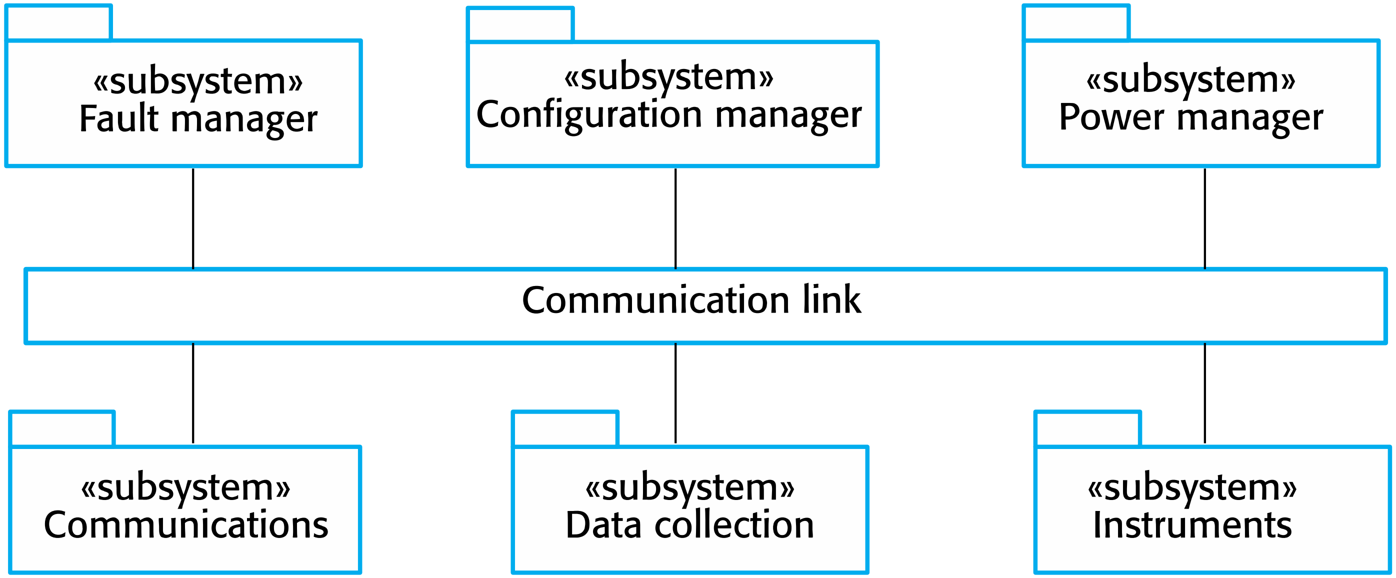
Use case description—Report weather

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.

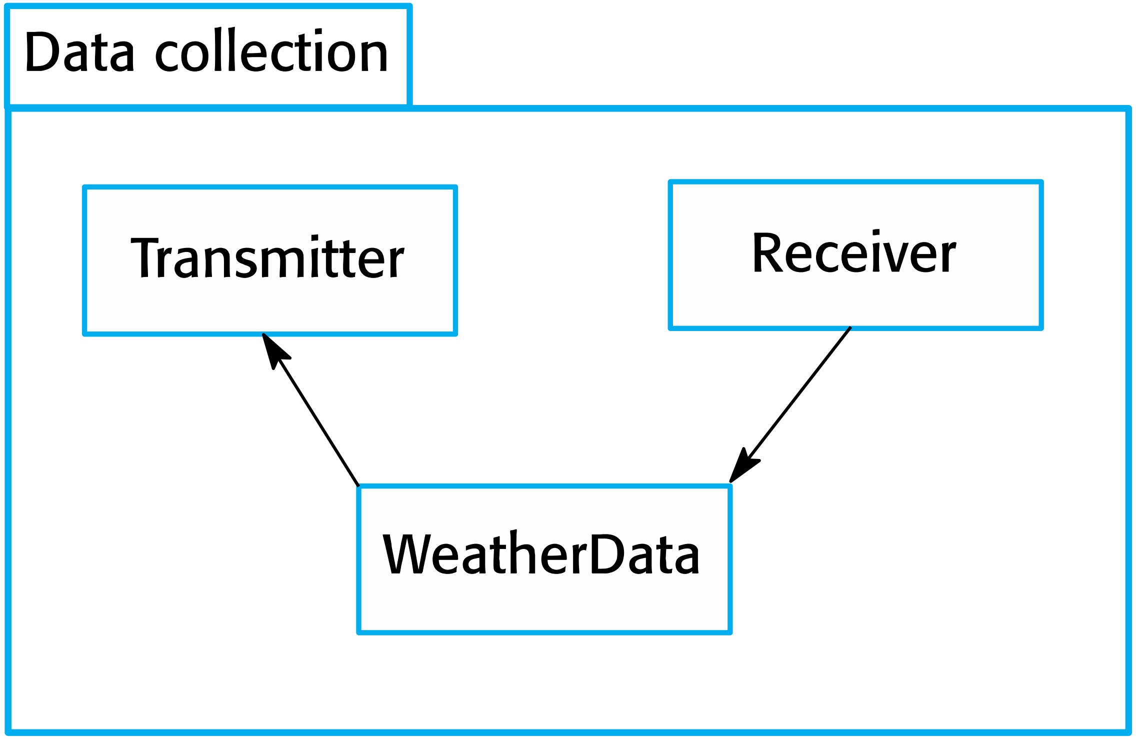
Architectural design

- ✧ 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.

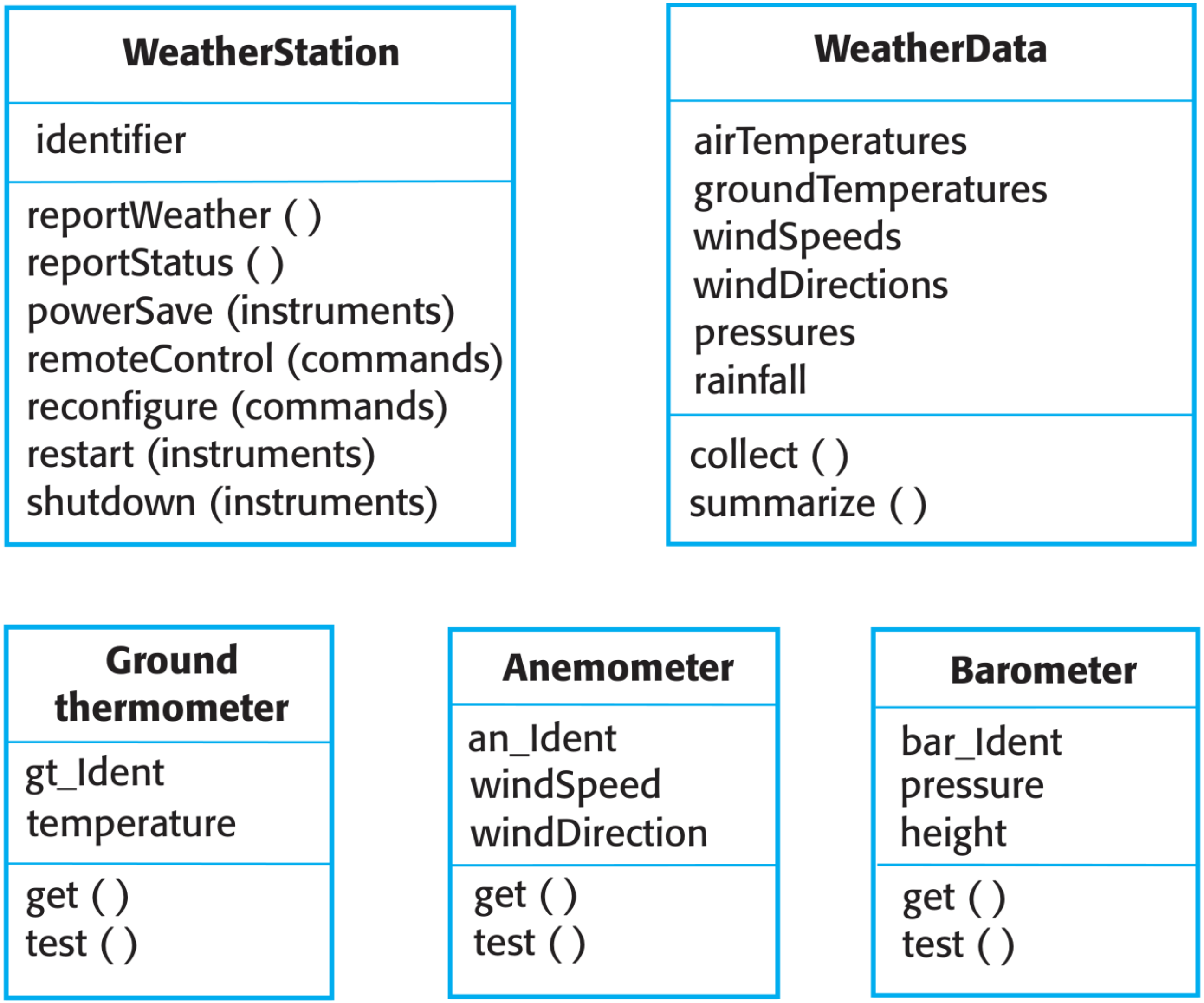
Approaches to identification

- Use a grammatical approach based on a natural language description of the system.
- 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 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.
- ✧ 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.
- 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.

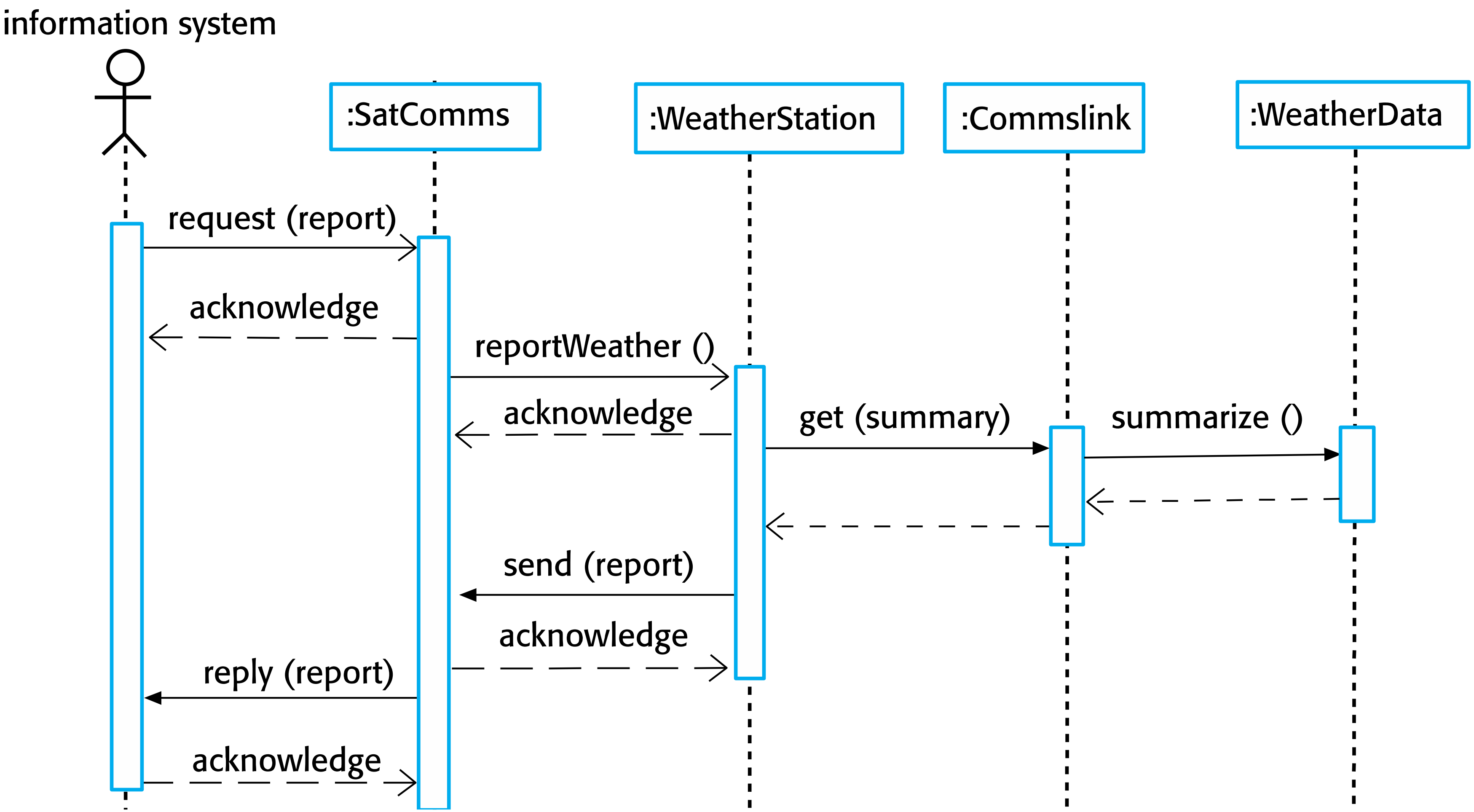
Subsystem models

- ✧ 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.

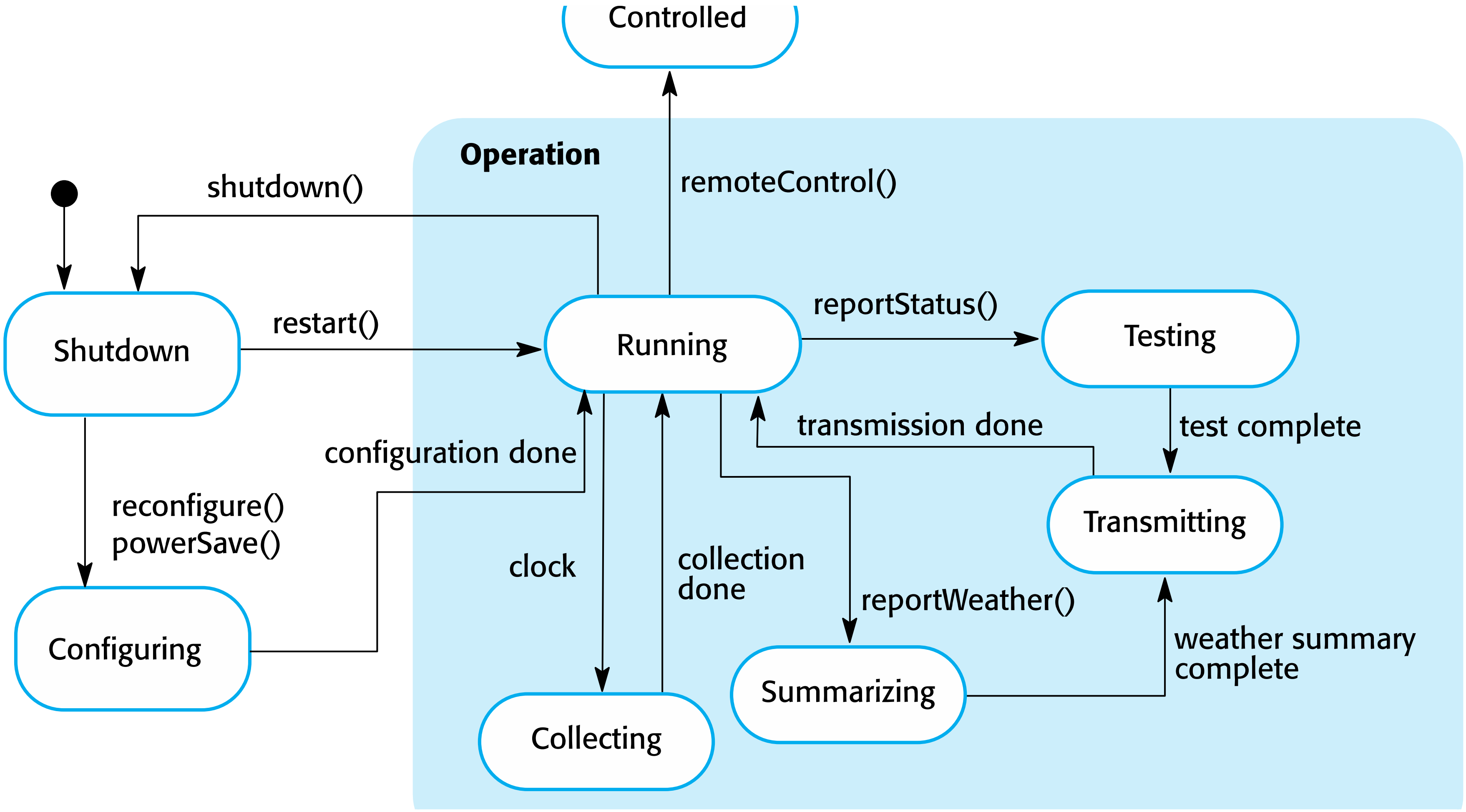
Sequence diagram describing data collection



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.

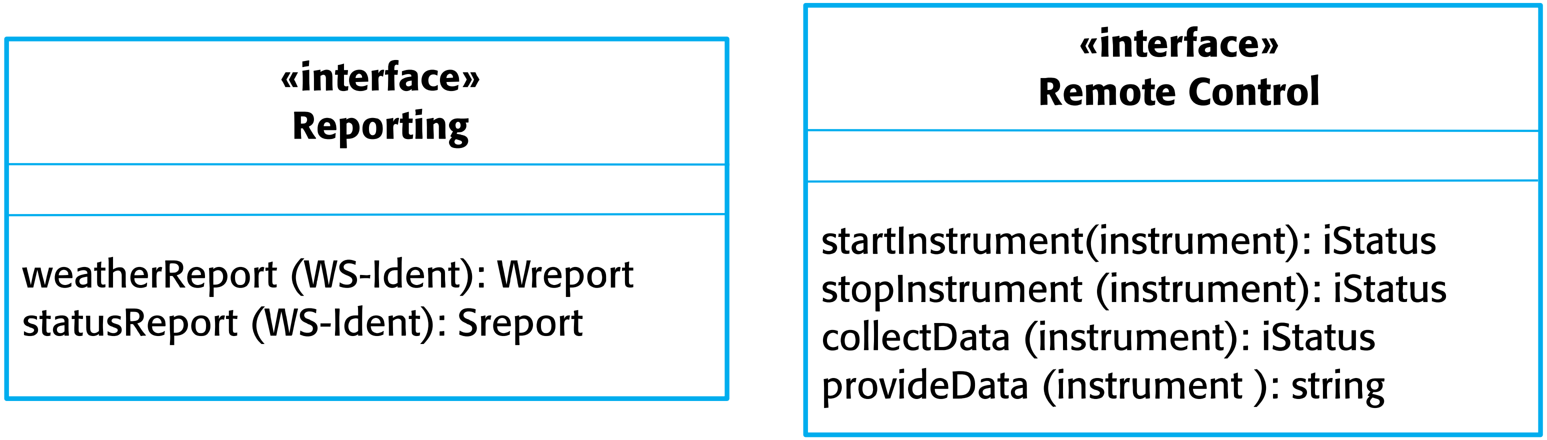
Weather station state diagram



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

Implementation issues

Implementation issues

- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - 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.
  - Configuration management During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - Host-target development Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

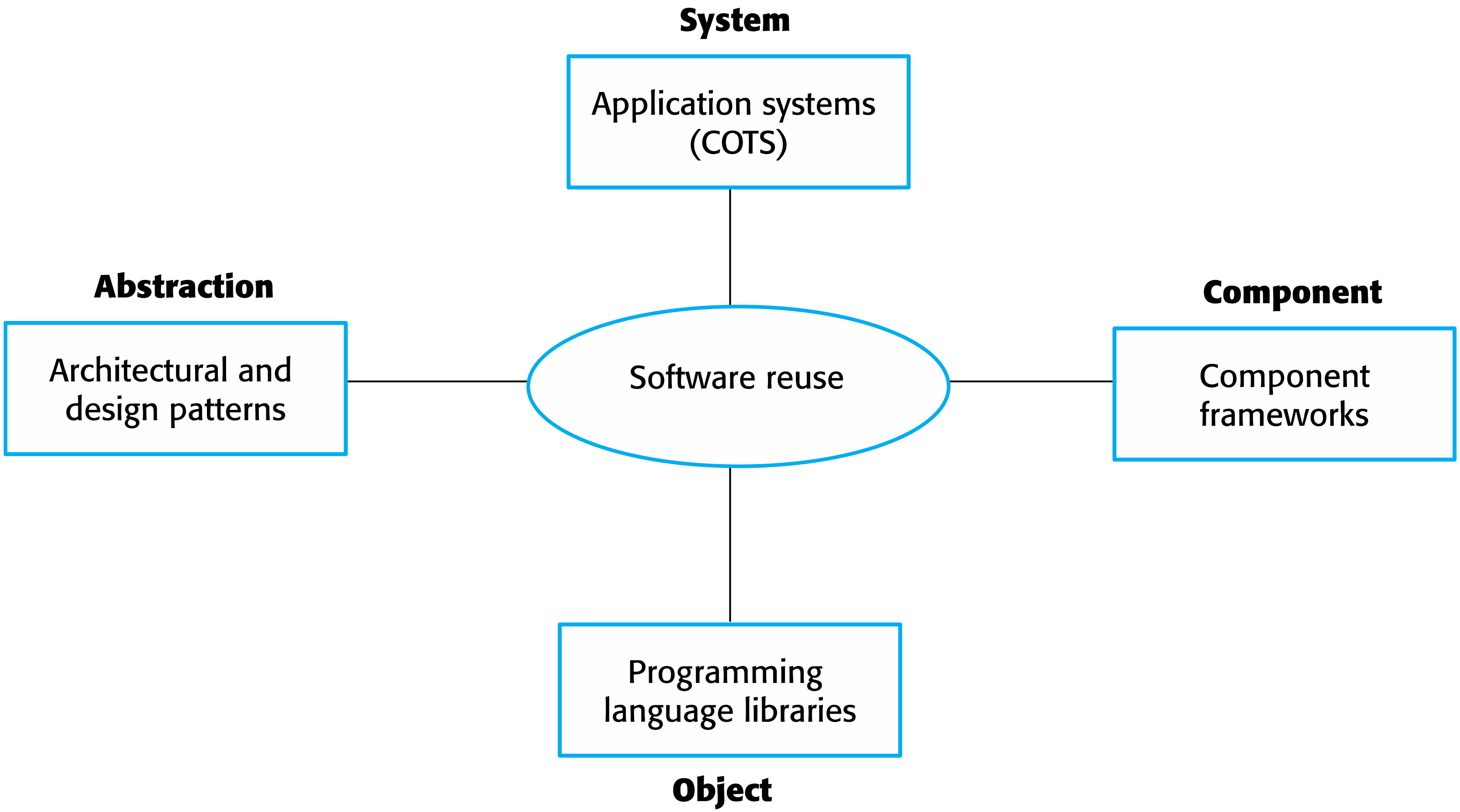
Reuse

- 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 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.
- 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.

Software reuse


System
Application systems (COTS)
Abstraction
Architectural and design patterns
Software reuse
Component
Component frameworks
Programming language libraries
Object

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.

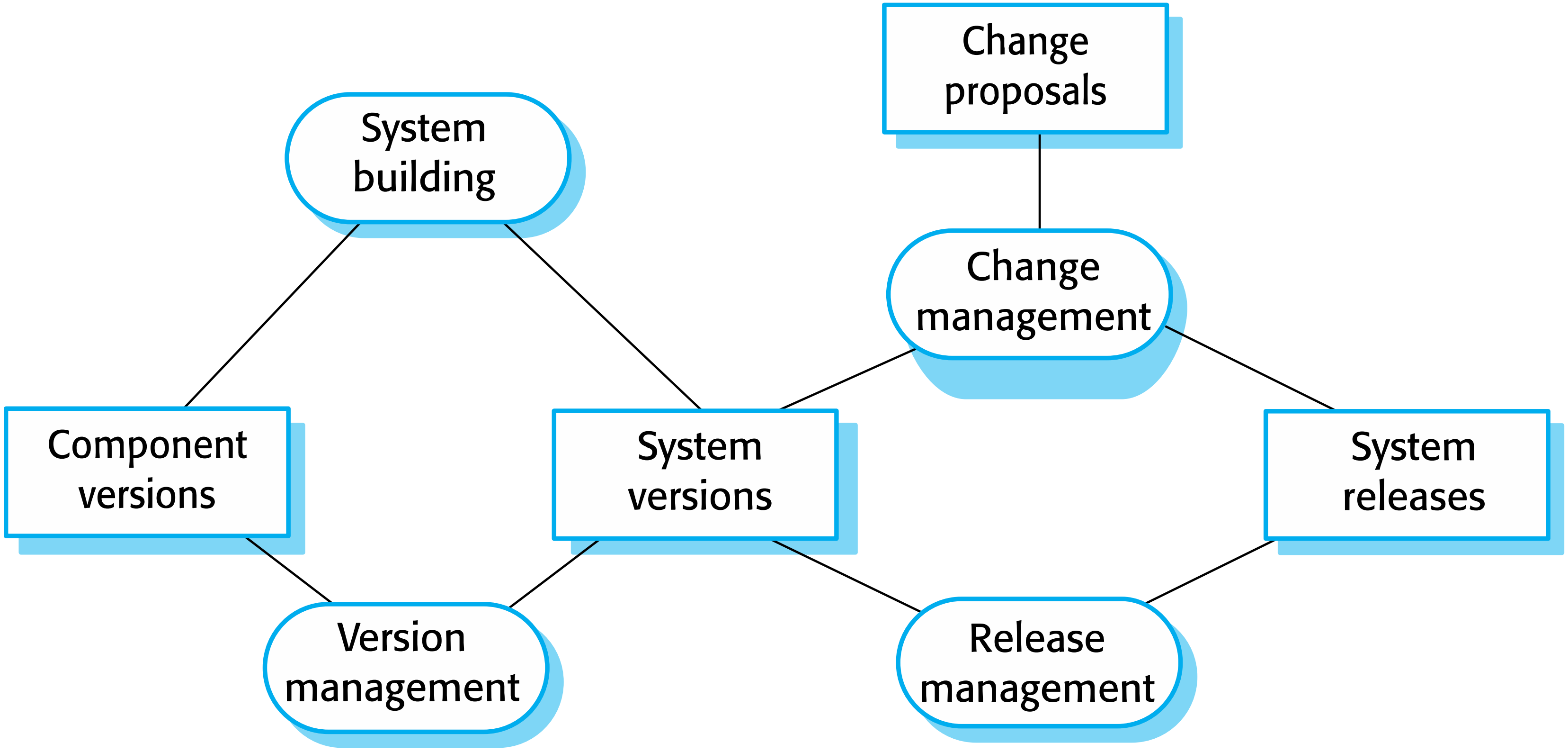
Configuration management

- ✧ Configuration management is the name given to the general process of managing a changing software system.
- ✧ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

Configuration management activities

- ✧ Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- ✧ System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- ✧ Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

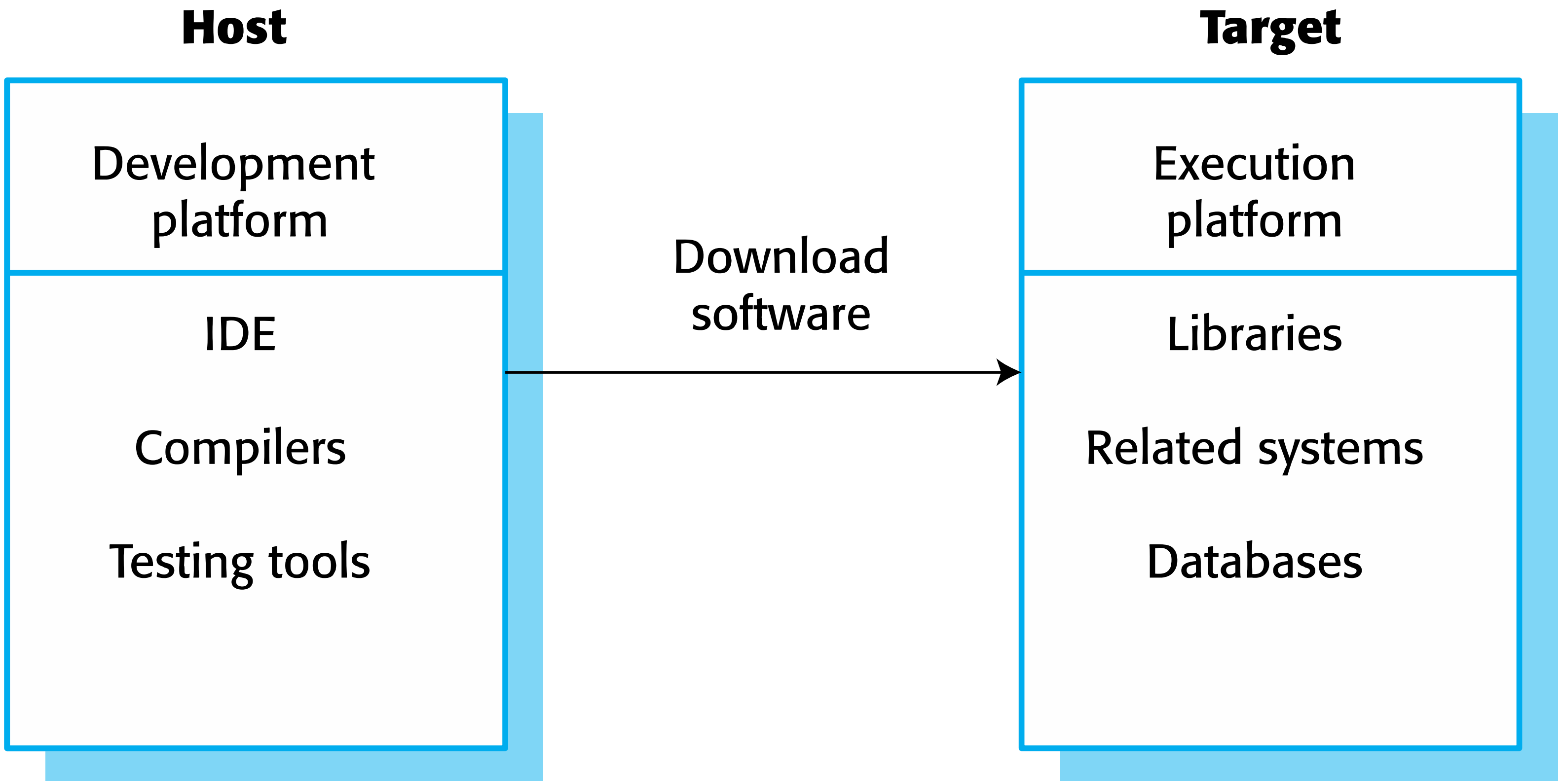
Configuration management tool interaction



Host-target development

- ✧ Most software is developed on one computer (the host), but runs on a separate machine (the target).
- ✧ More generally, we can talk about a development platform and an execution platform.
  - ▪ A platform is more than just hardware.
  - ▪ It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- ✧ Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Host-target development



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.

Open source development

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.