**XML data structure design and XML processing**
- Avoid redundancy in XML instance documents
- Use schema for validation as much as possible
- Exploit code reuse within a schema
- Apply guidelines about time/date representations

**XML programming in Java (with JAXB)**
- Abstract factory pattern – understand it and use it correctly
- Understand JAXB annotations and binding principles and use them correctly
- Understand importance of developing robust Java code (i.e. avoid runtime exceptions and manage all exceptions that may arise)
- Management of generated code: don't modify generated code but use it
- Use XML processors (e.g. JAXB marshaller/unmarshalller) for parsing/serialization of XML documents rather than direct reading/writing of XML
- When reading an XML document, validate it against a schema so that the code can be made robust with a limited effort
- Achieve portability of code (absence of dependencies on locale, location)
- Understand the importance of build automation and use it correctly

**Consuming web services in Java (with JAX-RS)**
- Robustness: catch (and manage) possible runtime exceptions in your client
- Know the standard meaning of HTTP methods and status codes and use them accordingly
- limit the number of interactions with the service (for performance)
- Follow links instead of building them when made possible by the service

**REST API design**
- Understand/apply the principles, best practices, guidelines about interface design:
  - do not expose more than strictly necessary (information hiding)
  - choose adequate granularity level for resources (performance)
  - use resource concept correctly (no resource that represents a verb, no multiple URIs for same resource, etc)
  - minimize the number of interactions necessary to perform operations (performance)
  - limit the size of messages (let clients choose what to get, etc)
  - use HTTP methods consistently with their standard meaning and features
  - foresee all particular cases and decide how errors will be communicated to clients
  - use HTTP status codes consistently with their standard meaning
  - support HTTP headers as much as possible/convenient
  - use HTTP media-type headers to control types
  - use idempotent methods when possible and convenient
  - provide hyperlinks so as to enable HATEOAS
  - exploit HTTP features for efficiency (e.g. use GET whenever possible because it is more efficient (caching), etc)
  - use stateless interactions only (no server status about sessions apart from resources: use only resources to store state in service)
  - use HTTP self-documentation features
  - complement self-documentation features with user documentation (make sure a user has all the necessary information about the service)
REST API implementation in Java (Java EE / JAX-RS)
- Understand JAX-RS annotations and data types and apply them so that the implementation is consistent with the interface design
- Organize implementation code into layers (interface, application logic, storage), in order to make each part simpler and future modifications easier.
- Use singletons when sharing of data in main memory is necessary. Implement singletons correctly, so that having more than one instance is impossible.
- Implement HTTP methods according to their standard definition (e.g. use Location header when new resource is created by means of POST)
- Use validation of request body and parameters as early as possible
- Deal with concurrency/performance issues
  - general aim: create thread safe implementations but allow as much concurrency as possible
  - understand what data are shared and what are not and limit shared data as much as possible and convenient
  - prefer re-entrant code when possible and use synchronization only if necessary
    - use shared data only when necessary
    - when shared data are necessary, use immutable data if applicable
    - analyse the code that uses shared non-immutable data to look for sequences of operations that should be executed atomically
  - if synchronization is necessary, prefer thread-safe Java library classes, especially the ones optimized for concurrency, rather than introducing synchronization yourself
  - make sure all shared non-immutable library objects not protected by synchronization belong to classes that are thread-safe (Javadoc)
- Avoid memory leakage: make sure that objects that are no longer necessary can be garbage collected (i.e. no references remain)
- Avoid information leakage from implementation (e.g. unhandled exceptions)
- Avoid unhandled exceptions also to avoid unforeseen behaviour of your service and to limit 500 errors when the responsibility of the error is not with the service
- Use loggers rather than output to console
- Separate cross-cutting aspects from application logic (e.g. by means of providers, filers and interceptors)

Security Aspects
- Make sure your design and implementation follow the OWASP guidelines that are under the responsibility of the designer/programmer (for our assignment we assume a public web service, not requiring authentication):
  - Input validation
  - Explicitly define content types
  - Avoid leaking information via error handling