

CS2103/T Software Engineering

AY2021/22 Semester 2

Software Engineering

1. Introduction

- **Software engineering** is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

Programming Paradigms

1. Object-Oriented Programming

- **Object-Oriented Programming (OOP)** is a programming paradigm. A programming paradigm guides programmers to analyse programming problems, and structure programming solutions, in a specific way.
- OOP does not dispute the view of world as data and operations, but restructure it to a higher level. It groups operations and data into modular units called objects and combines objects into structured networks. Objects and object interactions are the basic elements of design.
- OOP is mainly an **abstraction** mechanism, which is a higher level mechanism than procedural paradigm.

2. Other Programming Paradigms

Procedural Programming Paradigm ¹	C
Functional Programming Paradigm ²	F#, Haskell, Scala
Logic Programming Paradigm ³	Prolog
OOP + (<i>limited</i>) 1 + 2	C++, Java
1 + 2 + 3	JavaScript, Python

3. Objects

- OOP views the world as a network of interacting objects.
- Every object has **state** (data) and **behaviour**.
- OOP solutions try to create a similar object network inside the computer's memory so that a similar result can be achieved programmatically. However, it does not demand that the virtual world object network follows the real work exactly.
- Every object has an **interface** (through which other objects can interact with it) and an **implementation** (that supports the interface but may not be accessible to other objects).
- Objects interact by sending messages.
- **Abstraction** allows us to abstract away the lower level details and work with bigger granularly entities.
- **Encapsulation** protects an implementation from unintended actions and from inadvertent access. An object is an encapsulation of some data and related behaviour in terms of two aspects, packaging (an object packages data and related behaviours together into one self-contained unit) and information hiding (the data inside an object is hidden from the outside world and are only accessible using the object's interface).
- Encapsulation strengthens abstraction.

4. Classes

- A **class** contains instructions for creating a specific kind of objects.
- Class-level members = class-level attributes + class-level methods.

5. Enumerations

- An **enumeration** is a fixed set of values that can be considered as a data type.

6. Associations

Produced by Tian Xiao

- **Associations** are connections between objects to form a network so that they can interact with each other.
- Associations among objects change over time.
- Associations among objects can be generalised as associations between the corresponding classes.
- **Navigability** is the concept of which object in the association knows about the other object. It can be **unidirectional** or **bidirectional**.
- **Multiplicity** is the aspect of an OOP solution that dictates how many objects take part in each association.
- A **dependency** is a need for one class to depend on another without having a direct association in the same direction.
- A **composition** is an association that represents a strong whole-part relationship. When the whole is destroyed, parts are destroyed too (i.e. the part should not exist without being attached to a whole). Composition also implies that there cannot be cyclical links.
- An **aggregation** represents a container-contained relationship. It is a weaker relationship than composition.
- An **association class** represents additional information about an association.

7. Inheritance

- **Inheritance** allows one to define a new class based on an existing class. A superclass is said to be more general than the subclass.
- **Method overriding** is when a subclass changes the behaviour inherited from the parent class by reimplementing the method.
- **Method overloading** is when there are multiple methods with the same name but different type signatures.

Cheatsheet

- An **interface** is a behaviour specification. A class implementing an interface results in an is-a relationship.
- An **abstract class** is a representation of commonalities among its subclasses. It cannot be instantiated, but can be subclassed. An **abstract method** is a method signature without a method implementation.
- **Substitutability:** Every instance of a subclass is an instance of the superclass, but not vice versa.
- **Dynamic VS Static Binding:** Overridden methods are resolved using dynamic binding during runtime, whereas overloaded methods are resolved using static binding during compile time.

8. Polymorphism

- **Polymorphism** is the ability of different objects to respond, each in its own way, to identical messages. It allows you to write code targeting superclass objects, use that code on subclass objects, and achieve possibly different results based on the actual class of the object. It is achieved by **substitutability**, **operation overriding** and **dynamic binding** altogether.

4 Main OOP Principles	Abstraction
	Encapsulation
	Inheritance
	Polymorphism

Requirements

1. Requirements

- A software **requirement** specifies a need to be fulfilled by the software product.
- A software project may be a **brown-field** project (replace/update an existing product) or a **green-field**

project (develop a totally new system with no precedent).

- Requirements come from **stakeholders**. A stakeholder is a party that is potentially affected by the software project (e.g. users, sponsors, developers, interest groups, government agencies, etc.).
- Requirements can be prioritised based on the importance and urgency (e.g. High, Medium, Low). Some requirements can be discarded if they are considered “out of scope”.

2. Non-Functional Requirements

- **Functional requirements** specify what the system should do.
- **Non-functional requirements (NFR)** specify the constraints under which the system is developed and operated. For example,
 1. Data requirements (size, volatility, persistency, etc.).
 2. Environment requirements (technical environment in which the system would operate in or needs to be compatible with.).
 3. Accessibility, capacity, compliance with regulations, documentation, disaster recovery, efficiency, extensibility, fault tolerance, interoperability, maintainability, privacy, portability, quality, reliability, response time, robustness, scalability, security, stability, testability, etc.
- NFRs are easier to miss yet sometimes critical to the success of the software.

3. Gathering Requirements

- **Brainstorming:** A group activity designed to generate a large number of diverse and creative ideas for the solution of a problem. The aim is to generate ideas but not to validate them.

Produced by Tian Xiao

- **User surveys** can be used to solicit responses and opinions from a large number of stakeholders regarding a current product or a new product.

- **Observing** users in their natural work environment can uncover product requirements.

- **Interviewing** stakeholders and domain experts can produce useful information about project requirements.

- **Focus groups** are a kind of informal interview within an interactive group setting. A group of people are asked about their understanding of a specific issue, process, product, advertisement, etc.

- A **prototype** is a mock up, a scaled down version, or a partial system constructed to get users' feedback, to validate a technical concept, to give a preview of what is to come, or to compare multiple alternatives on a small scale before committing fully to one alternative, or to be used for early field testing under controlled conditions. Prototyping can uncover requirements, in particular those related to how users interact with the system.

- **Product surveys** about existing products are used to unearth shortcomings of existing solutions that can be addressed by a new product.

4. Prose

- **Prose:** A textual description used to describe requirements. Prose is useful when describing abstract ideas such as the vision of a product.

5. Feature List

- **Feature List:** A list of features of a product grouped according to some criteria such as aspect, priority, order of delivery, etc.

Example: Minesweeper Game feature list

1. Basic play – Single player play.

- 2. Difficulty levels
 - Medium levels
 - Advanced levels
- 3. Versus play – Two players can play against each other.
- 4. Timer – Additional fixed time restriction on the player.
- 5. ...

6. User Story

- **User story:** Short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.
- User stories capture user requirements in a way that is convenient for scoping, estimation and scheduling.
- User stories should only provide enough details to make a reasonably low risk estimate of how long the user story will take to implement.
- User stories can capture **NFRs** too.
- Given their lightweight nature, user stories are quite handy for recording requirements during early stages of requirements gathering.

Format: As a {user type/role} I can {function} so that {benefit}.

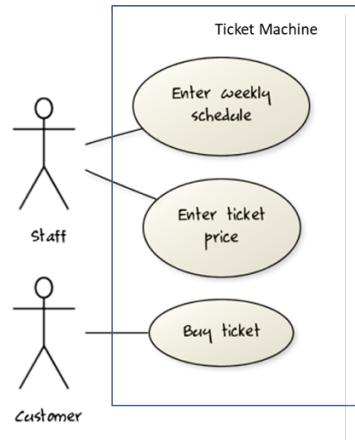
Example: Learning Management System

1. As a student, I can download files uploaded by lecturers so that I can get my own copy of the files.
2. As a lecturer, I can create discussion forums so that students can discuss things online.
3. As a tutor, I can print attendance sheets so that I can take attendance during the class.
4. As a forgetful user, I can view a password hint, so that I can recall my password.

7. Use Case

- **Use case:** A description of a set of sequences of actions, including variants, that a system performs to yield a observable result of value to an actor (a role played by a user). It describes an interaction between the user and the system for a specific functionality of the system.

Example: Use case diagram



Example: Transfer Money use case for a online banking system

System: Online Banking System (OBS)

User case: UC23 - Transfer Money

Actor: User

MSS:

1. User chooses to transfer money.
 2. OBS requests for details of the transfer.
 3. User enters the requested details.
 4. OBS requests for confirmation.
 5. User confirms.
 6. OBS transfers the money and displays the new account balance.
- Use case ends.

Extensions:

3a. OBS detects an error in the entered data.

3a1. OBS requests for the correct data.

3a2. User enters new data.

Steps 3a1-3a2 are repeated until the data entered are correct.

Use case resumes from step 4.

3b. User requests to effect the transfer in a future date.

3b1. OBS requests for confirmation.

3b2. User confirms future transfer.

Use case ends.

- Use cases capture the **functional requirements** of a system.

- To identify a use case, one needs to identify:

1. **Actors:** A use case can involve multiple actors. An actor can be involved in many use cases. A single person/system can play many roles. Many persons/systems can play a single role. Use cases can be specified at various levels of detail.

2. **Details:** The main body of the use case is a sequence of steps that describes the interaction between the system and the actors. A user case describes only the **externally visible behaviours**, not internal details of a system. A step gives the intention of the actor, not the mechanics (i.e. UI details should be omitted). The **Main Success Scenario (MSS)** describes the most straightforward interaction for a given use case, which assumes that nothing goes wrong. **Extensions** are add-ons to MSS that describe exceptional/alternative flow of events.

Format: Extensions in use cases

1. **Numbering:** An extension marked 3a can happen just after step 3 of MSS. An extension marked *a can happen at any step.

2. A use case can **include** another use case.

Underlined text is used to show an inclusion of a use case. For example:

MSS:

1. Staff creates the survey (UC44).

3. **Preconditions** specify the specific state you expect the system to be in before the use case starts. For example:

Actor: User

Preconditions: User is logged in.

MSS:

...

4. **Guarantees** specify what the use case promises to give us at the end of its operation.

Actor: User

Preconditions: User is logged in.

Guarantees:

- Money will be deducted from the source account only if the transfer to the destination account is successful.
- The transfer will not result in the account balance going below the minimum balance required.

MSS:

...

- Advantages:

1. Easy to understand and give feedback because of simple notation and plain English descriptions.

2. Decouples user intention from mechanism, allowing the system designers more freedom to optimise how a functionality is provided to a user.

3. Identifies all possible extensions encourages us to consider all situations that a software product might face during its operation.

4. Separates typical scenarios from special cases encourages us to optimise the typical scenarios.

- **Disadvantage:** Not good for capturing requirements that do not involve a user interacting with the

system. Hence, use cases should not be used as the sole means to specify requirements.

8. Glossary

- A **glossary** serves to ensure that all stakeholders have a common understanding of the noteworthy terms, abbreviations, acronyms, etc.

9. Supplementary Requirements

- A **supplementary requirements** section can be used to capture requirements that do not fit elsewhere. Typically, this is where most **NFRs** will be listed.

Design

1. Software Design

- Design is the creative process of transforming the problem into a solution; the solution is also called design.
 - Software design has two main aspects, **product/external design** (designing the external behaviour of the product to meet the users' requirements) and **implementation/internal design** (designing how the product will be implemented to meet the requires external behaviour).

2. Abstraction

- **Abstraction** is a technique for dealing with complexity. It works by establishing a level of complexity we are interested in, and suppressing the more complex details below that level. The guiding principle of abstraction is that only details that are relevant to the current perspective or the task at hand need to be considered.

- **Data abstraction:** Abstracting away the lower level data items and thinking in terms of bigger entities.

- **Control abstraction:** Abstracting away details of the actual control flow to focus on tasks at a higher level.

- Abstraction can be applied repeatedly to obtain progressively higher levels of abstraction.

- Abstraction is a general concept that is not limited to just data or control abstractions.

3. Coupling

- **Coupling** is a measure of the degree of dependence between components, classes, methods, etc. X is coupled to Y is a change to Y can potentially require a change in X.

Examples: A is coupled to B if

- A has access to the internal structure of B.
- A and B depend on the same global variable.
- A calls B.
- A received an object of B as a parameter or a return value.
- A inherits from B.
- A and B are required to follow the same data format or communication protocol.

- **High coupling** (aka. tight coupling, strong coupling) is discourages due to the following disadvantages:

1. **Maintenance** is harder, because a change in one module would cause changes in other modules coupled to it.

2. **Integration** is harder, because multiple components coupled with each other have to be integrated at the same time.

3. **Testing and reuse of the module** is harder, because of its dependence on other modules.

- Types of Coupling

Content Coupling	One module modifies or relies on the internal workings of another
-------------------------	---

	module (e.g. accessing local data of another module).
Common/Global Coupling	Two modules share the same global data.
Control Coupling	One module controlling the flow of another by passing it information on what to do (e.g. passing a flag).
Data Coupling	One module sharing data with another module (e.g. via passing parameters).
External Coupling	Two modules share an externally imposed convention (e.g. data formats, communication protocols, device interfaces).
Subclass Coupling	A class inherits from another class. A child class is coupled to the parent class.
Temporal Coupling	Two actions are bundled together just because they happen to occur at the same time (extracting a contiguous block of code as a method although the code block contains statements unrelated to each other).

4. Cohesion

- **Cohesion** is a measure of how strongly-related and focused the various responsibilities of a component are.

- Higher cohesion is better. Lower cohesion (aka. weak cohesion) is worse because:

1. Lowers the understandability of modules as it is difficult to express module functionalities at a higher level.

2. Lowers maintainability because a module can be modified due to unrelated causes or many

modules may need to be modified to achieve a small change in behaviour.

- 3. Lowers reusability of modules because they do not represent logical units of functionality.
- Cohesion can be present in many forms. For example, code related to a single concept, or invoked close together in time, or manipulates the same data structure, is kept together.

5. Modelling

- A **model** is a representation of something else. For example, a class diagram is a model that represents a software design. Models are abstractions.
- A model provides a simpler view of a complex entity because a model captures only a selected aspect. For example, a class diagram captures the structure of the software design but not the behaviour.
- Multiple models of the same entity may be needed to capture it fully.
- In software development, models are useful in the following ways:
 1. To analyse a complex entity related to software development.
 2. To communicate information among stakeholders, hence generating models from code is useful.
 3. As a blueprint for creating software.

- **UML class diagrams** are used to model structures ([Appendix I](#)).

- **UML object diagrams** are used to complement class diagrams ([Appendix II](#)).

- **UML activity diagrams** are used to model workflows, which define the flow in which a process or a set of tasks is executed ([Appendix III](#)).

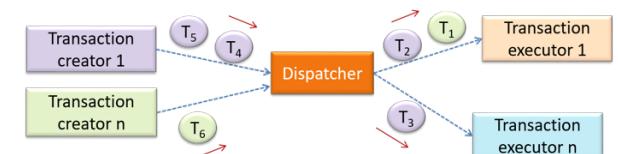
- **UML sequence diagrams** are used to model the interactions between various entities in a system, in a specific scenario ([Appendix IV](#)).

6. Software Architecture

- **Software architecture** shows the overall organisation of the system and can be viewed as a very high-level design. It usually consists of a set of interacting components that fit together to achieve the required functionality. It should be a simple and technically viable structure that is well-understood and agreed-upon by everyone in the development team, and it forms the basis for the implementation.

- Architecture is concerned with the **public** side of interfaces, not the private details of elements.
- The architecture is typically designed by the **software architect**.
- Architecture diagrams are free-form diagrams.
- Software architectures follow various high-level styles (aka. architectural patterns). Most applications use a mix of these architectural styles. For example,
 1. **N-tier** style (aka. multi-layered, layered): Higher layers make use of services provided by lower layers.
 2. **Client-server** style: At least one component playing the role of a server and at least one client component accessing the services of the server.

3. **Transaction processing** style: This style divides the workload of the system down to a number of transactions which are then given to a dispatcher that controls the execution of each transaction.



- 4. **Service-oriented** style: This style builds applications by combining functionalities packaged as programmatically accessible services.

5. **Event-driven** style: This style controls the flow of the application by detecting events from event emitters and communicating those events to interested event consumers. Usually used in GUIs.

7. Software Design Patterns

- A **design pattern** is an elegant reusable solution to a commonly recurring problem within a given context in software design. The common format to describe a pattern consists of the following components:

1. **Context**: The situation or scenario where the design problem is encountered.
2. **Problem**: The main difficulty to be resolved.
3. **Solution**: The core of the solution.
4. **Anti-patterns** (optional): Commonly used solutions which are usually incorrect and/or inferior to the design pattern.
5. **Consequences** (optional): Identifying the pros and cons of applying the pattern.
6. **Other useful information** (optional): Code examples, known users, other related patterns, etc.

- Singleton Pattern

Context	Certain classes should have no more than one instance (e.g. main controller of a system). These single instances are commonly known as singletons.
Problem	A normal class can be instantiated multiple times by invoking the constructor.
Solution	Make the constructor of the singleton class private, because a public constructor will allow others to instantiate the class at will. Provide a public class-level method to access the single instance.

Consequences	<p>Pros:</p> <ul style="list-style-type: none"> • Easy to apply. • Effective in achieving its goal with minimal extra work. • Provides an easy way to access the singleton object from anywhere in the code base. <p>Cons:</p> <ul style="list-style-type: none"> • The singleton object acts like a global variable that increases coupling across the code base. • In testing, it is difficult to replace singleton objects with stubs. • In testing, singleton objects carry data from one test to another even when one wants each test to be independent.
---------------------	--

- Abstraction Occurrence Pattern

Context	There is a group of similar entities that appear to be “occurrences” or “copies” of the same thing, sharing lots of common information, but also differing in significant ways.
Problem	Representing such objects as a single class would be problematic because it results in duplication of data which can lead to inconsistencies in data.
Solution	Let a copy of an entity be represented by two objects instead of one, separating the common and unique information into two classes to avoid duplication.

$\langle\langle\text{Abstraction}\rangle\rangle \xrightarrow{1} \cdot \xleftarrow{\cdot} \langle\langle\text{Occurrence}\rangle\rangle$

- Façade Pattern

Context	Components need to access functionality deep inside other components.
Problem	Access to the component should be allowed without exposing its internal details.
Solution	Include a Façade class that sits between the component internals and users of the component such that all access to the component happens through the Façade class.

```

graph LR
    TextUI[TextUI] -->|dashed line| LibraryLogic[LibraryLogic]
    GUI[GUI] -->|dashed line| LibraryLogic
    LibraryLogic -->|dashed line| Book[Book]
    LibraryLogic -->|dashed line| Book
    LibraryLogic -->|dashed line| Book
    LibraryLogic -->|dashed line| Book
    Book -->|dashed line| Book
    Book -->|dashed line| Book
    Book -->|dashed line| Book
    Book -->|dashed line| Book
  
```

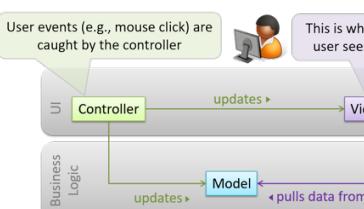
- Command Pattern

Context	A system is required to execute a number of commands, each doing a different task.
Problem	It is preferable that some part of the code executes these commands without having to know each command type.
Solution	The essential element of this pattern is to have a general Command object that can be passed around, stored, executed, etc. without knowing the type of command via polymorphism.

- Model View Controller (MVC) Pattern

Context	Most applications support storage/retrieval of information,
----------------	---

	displaying of information to the user, and changing stored information based on external inputs.
Problem	The high coupling that can result from the interlinked nature of the features described above.
Solution	Decouple data, presentation, and control logic of an application by separating them into three different components: Model , View , and Controller .



- Observer Pattern

Context	An object is interested in being notified when a change happens to another object.
Problem	The observed object does not want to be coupled to objects that are observing it.
Solution	Force the communication through an interface known to both parties.



- Design patterns are usually embedded in a larger design and sometimes applied in combination with other design patterns.
- The most famous source of design patterns is the **Gang of Four** book which contains 23 design patterns divided into three categories – creational, structural and behavioural.

- **Design principles** have varying degrees of formality – rules, opinions, rules of thumb, observations and axioms. They are more general, have wider applicability with correspondingly greater overlap among them than design patterns.

8. Design Approaches

- **Multi-level design**: The design of bigger systems needs to be done/shown at multiple levels.
- **Top-down and bottom-up design**: Multi-level design can be done in a top-down manner (when designing big and novel systems where the high-level design needs to be stable before lower levels can be designed), bottom-up manner (when designing a variation of an existing system or repurposing existing components to build a new system), or as a mix.
- **Agile design**: Agile designs are emergent that are not defined upfront.

Implementation

1. IDEs

- Professional software engineers often write code using **Integrated Development Environments (IDEs)**. IDEs support most development-related work within the same tool.
- An IDE generally consists of
 1. Source code editor: Includes features such as syntax colouring, auto-completion, easy code navigation, error highlighting and code-snippet generation.
 2. Compiler/Interpreter: Facilitates the compilation/linking/running/deployment of a program.
 3. Debugger: Allows the developer to execute the program one step at a time to observe the runtime behaviour in order to locate bugs.

4. Other tools that aid various aspects of coding (e.g. support for automated testing, drag-and-drop construction of UI components, version management support, simulation of the target runtime platform, and modelling support).

- Popular IDEs:

Java	Eclipse, IntelliJ IDEA, NetBeans
C#, C++	Visual Studio
Swift	XCode
Python	PyCharm

2. Debugging

- **Debugging** is the process of discovering defects in the program.

✓	✗
<ul style="list-style-type: none"> • Using a debugger 	<ul style="list-style-type: none"> • Inserting temporary print statements • Manually tracing through the code

3. Code Quality

- Production code needs to be of high quality.
- Among various dimensions of code quality, one of the most important is understandability/readability.

✓	✗
<ul style="list-style-type: none"> • Making the code obvious • Structuring code logically • KISSing (“keep it simple, stupid”) • SLAP (Single Level of Abstraction Principle) 	<ul style="list-style-type: none"> • Long methods • Deep nesting • Complicated expressions • Magic numbers • Unused parameters in method signature • Similar things that look different • Different things that look similar

- Making the happy path prominent

```

1 if (isUnusualCase) { //Guard Clause
2     handleUnusualCase();
3     return;
4 }
5
6 if (isErrorCase) { //Guard Clause
7     handleError();
8     return;
9 }
10
11 start();
12 process();
13 cleanup();
14 exit();

```

- Multiple statements in the same line
- Data flow anomalies such as preassigning values to variables and modifying it without any use of the preassigned value
- Premature optimisations

- Guidelines:

- Follow a consistent style ([Appendix V](#)).
- Name well:** Use nouns for things and verbs for actions, use standard words, use name to **explain**, not too long & not too short, avoid misleading names.
- Avoid unsafe shortcuts:** Use the default branch, do not recycle variables or parameters, avoid empty catch blocks, delete dead code, minimise scope of variables, minimised code duplication.
- Comment minimally but sufficiently:** do not repeat the obvious, write into the reader, explain what and why, not how.

4. Refactoring

- Refactoring** is the process of improving a program's internal structure in small steps without modifying its external behaviour. Refactoring is neither rewriting nor bug fixing.
- Two common refactorings: Consolidate duplicate conditional fragments + extract method.
- Code smells** are one way to identify refactoring opportunities. Periodic refactoring is a good way to pay off the technical debt a code base has accumulated.

5. Documentation

- Developer-to-developer documentation can be in one of two forms:

- Documentation for developer-as-user.
- Documentation for developer-as-maintainer.

- Software documentation is best kept in a text format for ease of version tracking. A writer-friendly source format (e.g. Markdown, AsciiDoc, PlantUML) is also desirable as non-programmers may need to author/edit such documents.

- Guidelines:

- Go top-down, not bottom-up:** A top-down breadth-first explanation is easier to understand.
- Aim for comprehensibility:** Use plenty of diagrams, examples, simple and direct explanations, get rid of statements that do not add value, do not have separate sections for each type of artifact.
- Document minimally, but sufficiently.**

- JavaDoc:** A tool for generating API documentation in HTML format from comments in the source code.

Example: Minimal JavaDoc comments for methods and classes

```

1 /**
2  * Returns lateral location of the specified position.
3  * If the position is unset, NaN is returned.
4 *
5  * @param x X coordinate of position.
6  * @param y Y coordinate of position.
7  * @param zone Zone of position.
8  * @return Lateral location.
9  * @throws IllegalArgumentException If zone is <= 0.
10 */
11 public double computeLocation(double x, double y, int zone)
12     throws IllegalArgumentException {
13     ...
14 }

1 package ...
2
3 import ...
4
5 /**
6  * Represents a location in a 2D space. A <code>Point</code> object corresponds to
7  * a coordinate represented by two integers e.g., <code>3,6</code>
8  */
9 public class Point {
10     ...
11 }

```

6. Error Handling

- Exceptions** are used to deal with unusual but not entirely unexpected situations that the program might encounter at runtime. When an error occurs at some point in the execution, the code being executed creates an exception object and hands it off to the runtime system. The runtime system then attempts to find an **exception handler** to handle it in the call stack. The exception handler chosen is said to catch the exception.

- Assertions** are used to define assumptions about the program state so that the runtime can verify them. An assertion failure indicates a possible bug in the code. If the runtime detects an assertion failure, it typically takes some drastic action (e.g. terminating the execution with an error message). Assertions can be disabled without modifying the code (e.g. `java -disableassertions HelloWorld`).

Example:

```

x = getX();
assert x == 0 : "x should be 0";
This assertion will fail with the message x should be 0 if x is not 0 at this point.

```

- Java assert VS JUnit assertions:** They are similar in purpose but Junit assertions are more powerful and customised for testing. Moreover, Junit assertions are not disabled by default.

- Exceptions VS Assertions:** The raising of an exception indicates an unusual condition created by **the user or the environment** whereas an assertion failure indicates **the programmer** made a mistake in the code.

- Logging** is the deliberate recording of certain information during a program execution for future reference. It can be useful for troubleshooting problems. A log file is like the black box of an airplane. Most programming environments come with

logging systems that allow sophisticated forms of logging.

Example: Logging uses Java

First, import the relevant Java package:

```
1 import java.util.logging.*;
```

Next, create a `Logger`:

```
1 private static Logger logger = Logger.getLogger("Foo");
```

Now, you can use the `Logger` object to log information. Note the use of a logging level for each message. When running the code, the logging level can be set to `WARNING` so that log messages specified as having `INFO` level (which is a lower level than `WARNING`) will not be written to the log file at all.

```
1 // log a message at INFO level
2 logger.log(Level.INFO, "going to start processing");
3 // ...
4 processInput();
5 if (error) {
6     // log a message at WARNING level
7     logger.log(Level.WARNING, "processing error", ex);
8 }
9 // ...
10 logger.log(Level.INFO, "end of processing");
```

- **Defensive programming:** A defensive programmer codes under the assumption “if you leave room for things to go wrong, they will go wrong”.

1. Enforcing compulsory associations.
2. Enforcing 1-to-1 associations.
3. Enforcing referential integrity.
4. Making copies.

- **Design by contract (DbC)** is an approach for designing software that requires defining formal, precise and verifiable interface specifications for software components. It assumes the caller of a method is responsible for ensuring all preconditions are met. It is not natively supported by Java and C++.

7. Integration

- **Integration** is to combine parts of a software product to form a whole.

- Approaches

“Late and One Time”:
Wait till all components

“Early and Frequent”:

are completed and integrate all finished components near the end of the project.	Integrate early and evolve each part in parallel, in small steps, re-integrating frequently.
Big-Bang Integration: Integrate all components at the same time.	Incremental Integration: Integrate a few components at a time.
Top-Down Integration: Higher-level components are integrated before bringing in the lower-level components.	Bottom-up Integration: The reverse of top-down integration. Sandwich Integration: A mix of top-down and bottom-up approaches.

- **Build automation tools** automate the steps of the build process, usually by means of build scripts (e.g. Gradle, Maven, Apache Ant, GNU Make). Some build tools also serve as dependency management tools.

- **Continuous Integration (CI):** Integration, building, and testing happens automatically after each code change (e.g. Travis, Jenkins, Appveyor, CircleCI, GitHub Actions).

- **Continuous Deployment (CD):** The changes are not only integrated continuously, but also deployed to end-users at the same time.

8. Reuse

- By reusing tried-and-tested components, the robustness of a new software system can be enhanced while reducing the manpower and time requirement. Reusable components can be a piece of code, a subsystem or a whole software.

- Costs associated with reuse:

1. The reused code may be an overkill, which increases the size or degrades the performance of the software.

2. The reused software may not be mature/stable enough to be used in an important product.

3. Non-mature software has the risk of dying off as fast as they emerged, leaving one with a dependency that is no longer maintained.

4. The license of the reused software restrict how one can use/develop his software.

5. The reused software might have bugs, missing features, or security vulnerabilities.

6. Malicious code can sneak into one's own product via compromised dependencies.

- An **Application Programming Interface (API)** specifies the interface through which other programs can interact with a software component. It is a contract between the component and its clients.

- A **library** is a collection of modular code that is general and can be used by other programs.

- A **framework** is a reusable implementation of a software providing generic functionality that can be selectively customised to produce a specific application (e.g. Eclipse). Some frameworks provide a complete implementation of a default behaviour which makes them immediately usable. A framework facilitates the adaptation and customisation of some desired functionality.

- **Libraries VS Frameworks:** Libraries are meant to be used “as is” while frameworks are meant to be customised/extended. One’s code calls the library code while the framework code calls one’s code. Frameworks use a technique called inversion of control (aka. Hollywood principle).

- A **platform** provides a runtime environment for applications (e.g. JavaEE, .NET). It is often bundled with various libraries, tools, frameworks, and technologies in addition to a runtime environment but the defining characteristic of a software platform is the presence of a **runtime environment**.

- **Cloud computing** is the delivery of computing as a service over the network, rather than a product running on a local machine. It can deliver computing services at three levels:

1. **Infrastructure as a service (IaaS)**: Delivers computer infrastructure as a service (e.g. virtual servers).

2. **Platform as a service (PaaS)**: Provides a platform on which developers can build applications (e.g. Google App Engine).

3. **Software as a service (SaaS)**: Allows applications to be accessed over the network (e.g. Google Docs).

Quality Assurance

1. Quality Assurance

- Software **Quality Assurance (QA)** is the process of ensuring that the software being built has the required levels of quality.

- **QA = Validation + Verification**

1. **Validation**: Are you building the right system? Are the requirements correct?

2. **Verification**: Are you building the system right? Are the requirements implemented correctly?

- It is not important to distinguish whether something belongs under validation or verification. It is important that both are done.

2. Code Reviews

- **Code review** is the systematic examination of code with the intention of finding where the code can be improved. It can be done in various forms:

1. Pull request reviews.

2. In pair programming: Two programmers working on the same code at the same time.

3. Formal inspections: A group of people systematically examine project artifacts to discover defects, including the author, moderator, secretary and inspector/reviewer.

- **Advantages** of code review over testing:

1. It can detect functionality defects as well as other problems such as coding standard violations.

2. It can verify non-code artifacts and incomplete code.

3. It does not require test drivers or stubs.

- **Disadvantage**: Code review is a manual process and therefore error prone.

3. Static Analysis

- **Static analysis** is the analysis of code without actually executing the code. It can find useful information such as unused variables. In contrast, dynamic analysis requires the code to be executed to gather additional information about the code.

- Higher-end static analysis tools (static analysers, e.g. CheckStyle, PMD, FindBugs) can perform more complex analysis such as locating potential bugs, memory leaks, inefficient code structures, etc.

4. Formal Verifications

- **Formal verification** uses mathematical techniques to prove the correctness of a program.

- **Advantage**: Formal verification can be used to prove the absence of errors. In contrast, testing can only prove the presence of errors.

- **Disadvantages**: Formal verification only proves the compliance with the specification, but not the actual utility of the software. Also, it requires highly specialised notations and knowledge which makes it an expensive technique to administer. Therefore, formal verifications are more commonly used in safety-critical software such as flight control systems.

5. Testing

- **Testing** is the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

- When testing, one executes a set of **test cases**. A test case specifies how to perform a test. At a minimum, it specifies the input to the software under test (SUT) and the expected behaviour.

1. Feed the input to the SUT.

2. Observe the actual output.

3. Compare actual output with expected output.

- Test cases can be determined based on the specification, reviewing similar existing systems, or comparing to the past behaviour of the SUT.

- A **test case failure** is a mismatch between the expected behaviour and the actual behaviour. A failure indicates a potential defect or a bug, unless the error is in the test case itself.

- **Testability**: An indication of how easy it is to test an SUT.

- **Regression testing**: The re-testing of the software to detect regressions (when one modifies a system, the modification may result in some unintended and undesirable effects on the system, called regressions). Regression testing is more effective when it is done frequently, hence more practical when it is automated.

- **Developer testing**: The testing done by the developers themselves as opposed to professional testers or end-users, since it is better to do early testing because if testing is delayed until the full product is complete:

1. Locating the cause of a test case failure is difficult due to a large search space.

2. Fixing a bug found during such testing could result in major rework.

- 3. One bug might hide other bugs.
- 4. The delivery may have to be delayed.
- **Unit testing:** Testing individual units to ensure each piece works correctly. A proper unit test requires the unit to be tested in isolation, and **stubs** can isolate the SUT from its dependencies.
- **Integration testing:** Testing whether different parts of the software work together as expected. It is not simply a case of repeating the unit test cases using the actual dependencies, but additional test cases that focus on the interactions between the parts. In practice, developers use a hybrid of unit and integration tests to minimise the need for stubs.
- **System testing:** Taking the whole system and testing it against the system specification. System testing is typically done by a testing team (aka. QA team). System test cases are based on the specified external behaviour of the system, and it includes testing against NFRs too (e.g. performance testing, load testing, security testing, compatibility testing, interoperability testing, usability testing, portability testing).
- **Alpha and beta testing:** Alpha testing is performed by the users, under controlled conditions set by the software development team; beta testing is performed by a selected subset of target users of the system in their natural work setting.
- **Dogfooding:** Eating your own dog food (aka. dogfooding), is when creators use their own product so as to test the product.
- **Exploratory testing** (aka. reactive testing, error guessing technique, attack-based testing, bug hunting): Devise test cases on-the-fly, creating new test cases based on the results of the past test cases. It depends on the tester's prior experience and intuition.
- **Scripted testing:** First write a set of test cases based on the expected behaviour of the SUT, and

then perform testing based on that set of test cases. It is more systematic and hence likely to discover more bugs given sufficient time.

- **Acceptance testing:** Testing the system to ensure it meets the user requirements. It comes after system testing.

- System testing VS Acceptance testing:

System Testing	Acceptance Testing
Done against system specification	Done against requirement specification
Done by testers of the project team	Done by a team that represents the customer
Done on the development environment or a test bed	Done on the deployment site or on a close simulation of the deployment site
Both negative and positive test cases	More focus on positive test cases

- System specification VS Requirement specification:

System Specification	Requirement Specification
Also includes details on how the system will fail gracefully when pushed beyond limits, how to recover, etc.	Limited to how the system behaves in normal working conditions
Written in terms of how the system solves those problems	Written in terms of problems that need to be solved
Could contain additional APIs not available for end-users (for the use of developers/testers)	Specifies the interface available for intended end-users

- Passing system tests does not necessarily mean passing acceptance testing.

6. Test Automation

- An **automated test case** can be run programmatically and the result of the test case is determined programmatically.
- A simple way to semi-automate testing of a command line interface (CLI) app is by using input/output redirection.
- A **test driver** is the code that drives the SUT for the purpose of testing.
- **Junit** is a tool for automated testing of Java programs.

Example: A JUnit test for a Payroll class

```

1  @Test
2  public void testTotalSalary() {
3      Payroll p = new Payroll();
4
5      // test case 1
6      p.setEmployees(new String[]{"E001", "E002"});
7      assertEquals(6400, p.totalSalary());
8
9      // test case 2
10     p.setEmployees(new String[]{"E001"});
11     assertEquals(2300, p.totalSalary());
12
13     // more tests...
14 }
```

- GUI automated testing tools:

TestFX	JavaFX GUIs
Visual Studio	“Record replay” type of GUI test automation
Selenium	Web application GUIs

7. Test Coverage

- **Test coverage** is a metric used to measure the extent to which testing exercises the code, including the following criteria:

1. **Functional/method coverage:** Based on functions executed.
2. **Statement coverage:** Based on the number of lines of code executed.

3. **Decision/branch coverage:** Based on the decision points exercised.

4. **Condition coverage:** Based on the boolean sub-expressions, each evaluated to both true and false with different test case.

5. **Path coverage:** Based on possible paths through a given part of the code executed. 100% path coverage implies all possible execution paths hence the highest intensity of testing.

6. **Entry/exit coverage:** Based on possible calls to and exists from the operations in the SUT.

- Measuring coverage is often done using **coverage analysis tools**.

- Coverage analysis can be useful in improving the quality of testing.

8. Dependency Injection

- **Dependency injection** is the process of injecting objects to replace current dependencies with a different object (e.g. inject stubs to isolate the SUT).
- Polymorphism can be used to implement dependency injection.

9. Test-Driven Development

- **Test-Driven Development (TDD)** advocates writing the tests before writing the SUT, while evolving functionality and tests in small increments. It guarantees that the code is testable.

- Steps:

1. Decide what behaviour to implement.
2. Write test cases to test that behaviour.
3. Run those test cases and watch them fail.
4. Implement the behaviour.
5. Run the test cases.
6. Keep modifying the code and rerunning test cases until they all pass.
7. Refactor code to improve quality.
8. Repeat for each small unit of behaviour.

10. Test Case Design

- Except for trivial SUTs, exhaustive testing is not practical because such testing often requires a massive/infinite number of test cases. Hence, test cases need to be designed to make the best use of testing resources. In particular,

1. Testing should be **effective** (i.e. finds a high percentage of existing bugs).

2. Testing should be **efficient** (i.e. has a high rate of success).

- Positive VS Negative test cases: A **positive** test case is when the test is designed to produce an expected/valid behaviour, whereas a **negative** test case is designed to produce a behaviour that indicates an invalid/unexpected situation, such as an error message.

- Black-box VS Glass-box: Test case design can be of three types, based on how much of the SUT's internal details are considered when designing test cases:

1. **Black-box** (aka. specification-based, responsibility-based): Test cases are designed exclusively based on the SUT's specified external behaviour.

2. **White-box** (aka. glass-box, structured, implementation-based): Test cases are designed based on what is known about the SUT's implementation (i.e. the code).

3. **Gray-box**: Test cases are designed based on some important information about the implementation.

- **Equivalence partitions (EP)**: A test case design technique that used the observation - **most SUTs do not treat each input in a unique way. Instead, they process all possible inputs in a small number of distinct ways** - to improve the E&E of testing. One should avoid testing too many inputs from one partition, and ensure all partitions are tested.

- **Boundary value analysis (BVA)**: A test case design heuristic that is based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions. It suggests that when picking test inputs from an equivalence partition, values near boundaries are more likely to find bugs. Typically, one should choose one value from the boundary, one value just below the boundary and one value just above the boundary. For example, consider the partition [1-12], one should choose 0, 1, 2, 11, 12, 13.

- An SUT can take multiple inputs. Testing all possible combinations is effective but not efficient, hence one needs strategies to **combine test inputs** that are both effective and efficient:

1. **All combinations** strategy: Generates test cases for each unique combination.

2. **At least once** strategy: Includes each test input at least once.

3. **All pairs** strategy: For any given pair of inputs, all combinations between them are tested.

4. **Random** strategy: Generates test cases using one of the other strategies and them picks a subset randomly.

- Combining test inputs heuristics:

1. **Each valid input must appear at least once in a positive test case**.

2. **No more than one invalid input appears in a test case**.

- Use cases can be used for system testing and acceptance testing.

Project Management

1. Revision Control

- **Revision control** is the process of managing multiple versions of a piece of information. A revision is a

Cheatsheet

state of a piece of information at a specific time that is a result of some changes to it.

- **Revision control software (RCS)** are the software tools that automate the process of revision control. It will track the history and evolution of one's project. It has the following advantages:

1. Easier to collaborate.
 2. Easier to recover from mistakes.
 3. Easier to work simultaneously on, and manage the drift between multiple versions of the project.
- **Repository**: The database of the history of a directory being tracked by an RCS. It has the following functions:
1. **Track**: Specifies which file to track.
 2. **Ignore**: Specifies which file to ignore.
 3. **Commit**: Saves a snapshot (commit) of the current state.
 4. **Stage**: Chooses which changes to commit.
 5. **Hash**: Identifies each unique commit.
 6. **Tag**: Gives a commit a more easily identifiable name.

7. **Diff**: Shows the changes between two points of the history.

8. **Checkout**: Restores the state of the working directory at a point in the past.

9. **Clone**: Creates a copy of a remote repo in another location on computer. The original repo is called upstream repo.

10. **Pull**: Receives new commit from the upstream repo.

11. **Push**: Copies the new commits onto the destination repo.

12. **Fork**: Creates a remote copy of a remote repo.

13. **Pull request (PR)**: Requests to contribute code to a remote repo.

14. **Branch**: Evolves multiple versions of the software in parallel.

15. **Merge**: Merges two branches. Merge conflicts need to be resolved manually.

- CRCS VS DRCS: **Centralised RCS (CRCS)** uses a central remote repo that is shared by the team, while **Distributed RCS (DRCS)** allows multiple remote repos.

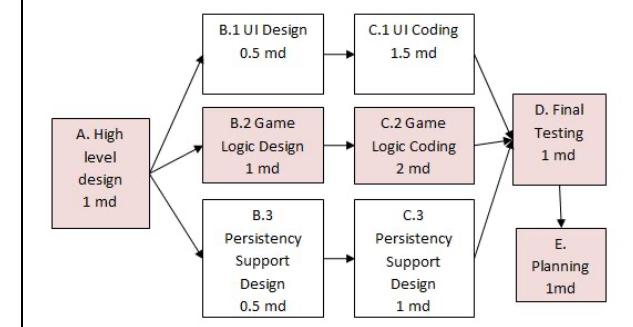
2. Project Planning

- A **Work Breakdown Structure (WBS)** depicts information about tasks and their details in terms of subtasks. The effort is traditionally measured in man hour/day/month. All tasks should be well-defined.
- A **milestone** is the end of a stage which indicates significant progress.
- A **buffer** is a time set aside to absorb any unforeseen delays.
- **Issue trackers** are commonly used to track task assignment and progress.
- A **Gantt chart** is a 2D bar chart, drawn as time VS tasks. A solid bar represents the main task, which is composed of a number of grey bars (subtasks). The diamond shape indicates an important deadline/deliverable/milestone.

Example: PERT chart

- **A Program Evaluation Review Technique (PERT)** chart uses a graphical technique to show the order/sequence of tasks. It can help determine the order of tasks, which tasks can be done concurrently, the shortest possible completion time and the critical path (the path in which any delay can directly affect the project duration).

Example: PERT chart



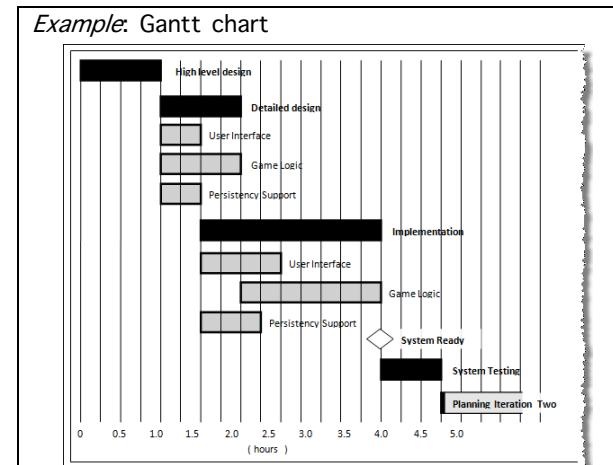
3. Teamwork

- **Egoless team (aka. democratic)**: Every team member is equal in terms of responsibility and accountability. Good for smaller projects.
- **Chief programmer team**: There is a single authoritative figure, the chief programmer.
- **Strict hierarchy team**: A strictly defined organisation among the team members. Good in a large, resource-intensive, complex project.

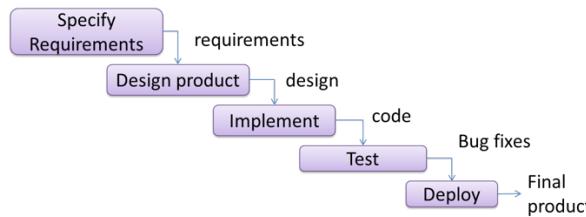
4. SDLC Process Models

- **Software development life cycle (SDLC)**: Requirements, analysis, design, implementation and testing.

- **Sequential model** (aka. waterfall model) models software development as a linear process. When one stage of the process is completed, it should produce some artifacts to be used in the next stage. This



could be a useful model when the problem statement is well-understood and stable, which is yet rare and keeps changing in real world.



- **Iterative model** (aka. iterative and incremental model) advocates having several iterations of SDLC. Each iteration produces a new version of the product. It can take a breadth-first (involves all major components in parallel) or a depth-first approach (fleshing out only some components) to iteration planning. Most projects use a mixture of breadth-first and depth-first iterations.

- **Agile model:** Individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, responding to change over following a plan (e.g. eXtreme Programming (XP), Scrum, Unified Process by the Three Amigos).

Principles

1. Single Responsibility Principle (SRP)

- A class should have one, and only one, reason to change.
- Gather together the things that change for the same reasons. Separate those things that change for different reasons.

2. Open-Closed Principle (OCP)

- A module should be open for extension but closed for modification. That is, modules should be written so that they can be extended, without requiring them to be modified.
- OCP aims to make a code entity easy to adapt and reuse without needing to modify the code entity itself.

3. Liskov Substitution Principle (LSP)

- A subclass should not be more restrictive than the behaviour specified by the superclass. Code that works with the superclass should be able to work with its subclasses.

4. Interface Segregation Principle (ISP)

- No client should be forced to depend on methods it does not use.

5. Dependency Inversion Principle (DIP)

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

6. Separation of Concerns Principle (SoC)

- To achieve better modularity, separate the code into distinct sections, such that each section addresses a separate concern.
- SoC reduces functional overlaps among code sections and limits the ripple effect when changes are introduced to a specific part of the system.
- SoC can be applied at the class level, as well as higher levels (e.g. N-Tier architecture).
- SoC should lead to higher cohesion and lower coupling.

7. Law of Demeter (LoD)

- An object should have knowledge of another object.

- An object should only interact with objects that are closely related to it.
- LoD is also known as *Don't talk to strangers* and *Principle of Least Knowledge*.

8. YAGNI Principle

- “You aren’t gonna need it!”: Do not add code simply because you might need it in the future.

9. DRY Principle

- “Don’t repeat yourself!”: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

10. Brook’s Law

- Adding people to a late project will make it later.
- The additional communication overhead will outweigh the benefit of adding extra manpower, especially if done near a deadline.

Tools

1. UML

See Appendix I, II, III and IV.

2. IntelliJ IDEA

3. Git and GitHub

References

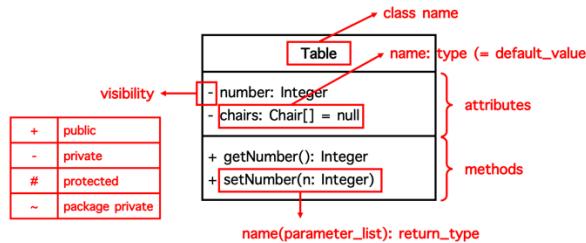
This cheatsheet (aka. reference sheet) is adapted from *Software Engineering for Self-Directed Learners (CS2103/T Edition – 2022 Jan-Apr)* following the link (<https://nus-cs2103-ay2122s2.github.io/website/book-adapted/index.html>). Consequently, it also reuses material from references in the book.

Appendix I: UML Class Diagrams

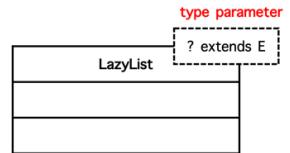
UML class diagrams describe the **structure** (but not the behaviour) of an OOP solution.

1. Classes

- Normal classes

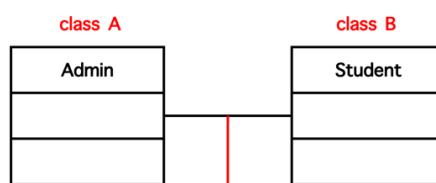


- Generic classes



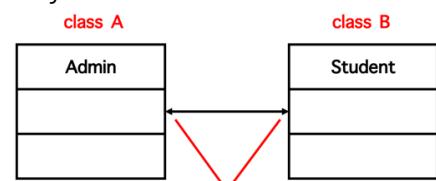
2. Associations

- Normal association



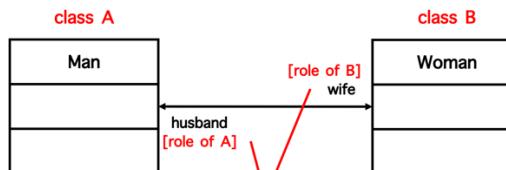
Association between class A and B is represented by a solid line.

- Navigability



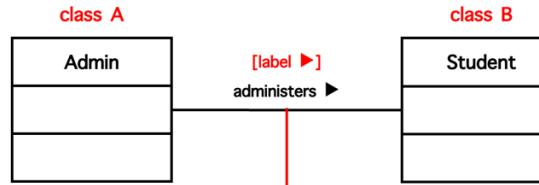
Use arrowheads to indicate the navigability of an association.

- Roles in association



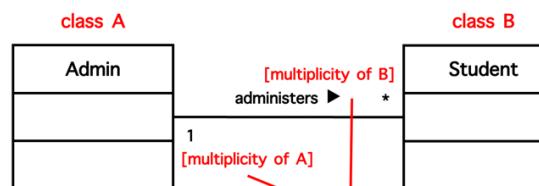
Use labels to indicate the role played by the classes in the association.

- Meanings of association



Admin class is associated with Student class because an Admin object administers a Student object.

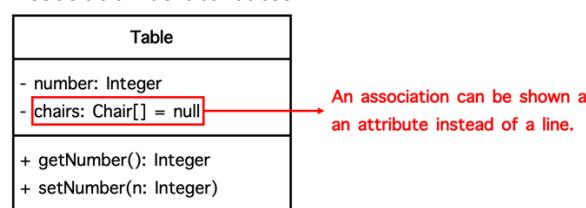
- Multiplicity



multiplicity

0..1	optional, can be linked to 0 or 1 object
1	compulsory, must be linked to 1 object at all times
*	can be linked to 0 or more objects
n..m	must be linked to n to m (inclusive) objects

- Association as attributes



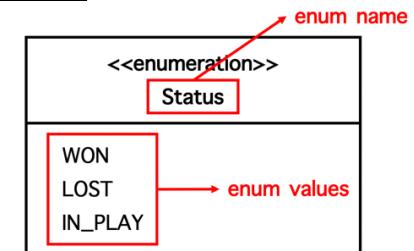
An association can be shown as an attribute instead of a line.

3. Dependencies

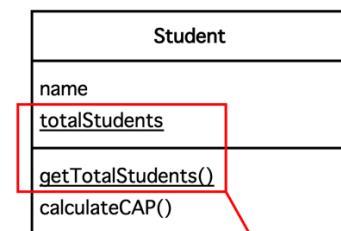


Use dashed arrows to show a dependency.

4. Enumerations

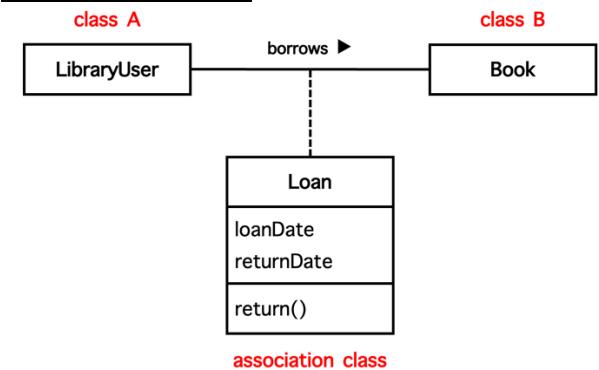


5. Class-Level Members



Use underline to denote class-level members.

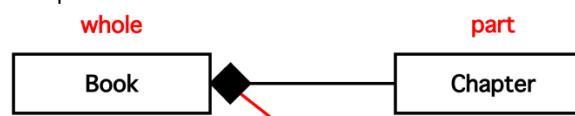
6. Association Classes



association class

7. Composition and Aggregation

- Composition



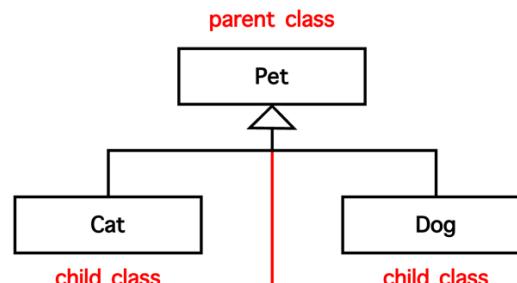
Use solid diamond to denote composition.

- Aggregation



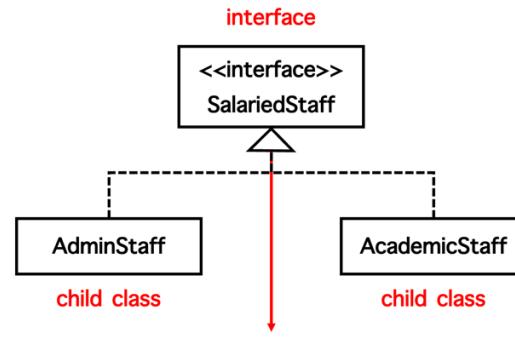
Use hollow diamond to denote aggregation.

8. Class Inheritance



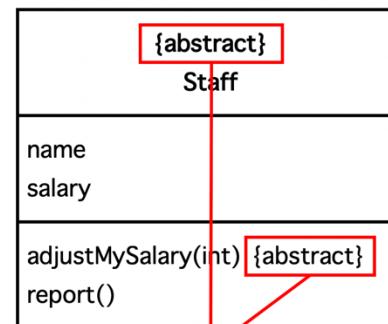
Use triangle and line to denote inheritance.

9. Interfaces



Use triangle and dashed line to denote inheritance from interface.

10. Abstract Classes

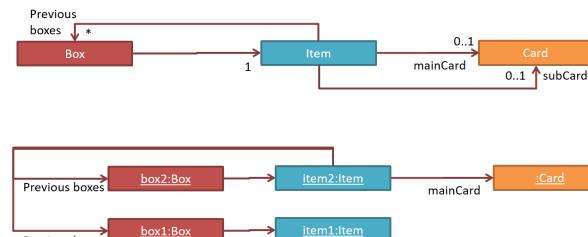


Use {abstract} to denote abstract class/methods.

11. Combine

Example: The following class and object diagrams corresponds to the following code.

Class and object diagrams:



Code:

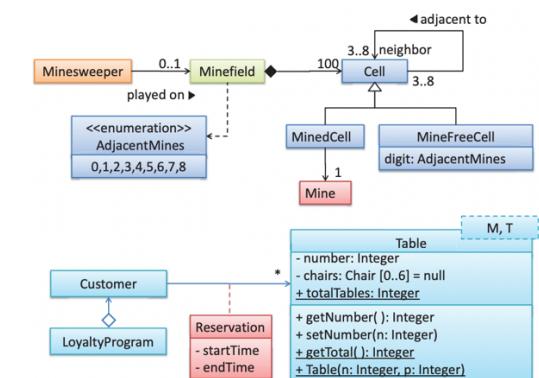
```

1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Main {
5     public static void main(String[] args) {
6         Item item1 = new Item();
7         Item item2 = new Item();
8         Box box1 = new Box(item1);
9         Box box2 = new Box(item2);
10        item2.setMainCard(new Card());
11        item2.addPreviousBox(box1);
12        item2.addPreviousBox(box2);
13    }
14
15    class Box {
16        private Item item;
17        Box(Item item) {
18            this.item = item;
19        }
20    }
21
22
    
```

```

23 class Item {
24     private List<Box> previousBoxes = new ArrayList<>();
25     private Card mainCard = null;
26     private Card subCard = null;
27
28     void setMainCard(Card card) {
29         this.mainCard = card;
30     }
31
32     void setSubCard(Card card) {
33         this.subCard = card;
34     }
35
36     void addPreviousBox(Box previousBox) {
37         previousBoxes.add(previousBox);
38     }
39 }
40
41 class Card {
42
43 }
    
```

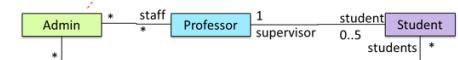
Example: Minefield



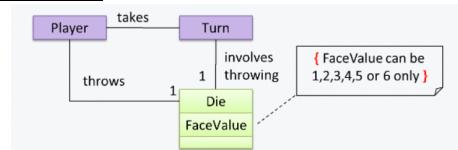
12. Notes

This may be redundant.
To be verified later.

This diagram is only a
work in progress.



13. Constraints

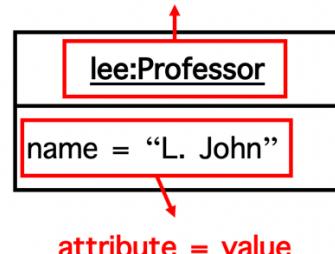


Appendix II: UML Object Diagrams

UML object diagrams show an object structure **at a given point of time**.

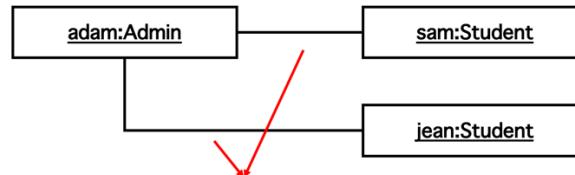
1. Objects

(object name) : class name



No compartment for methods!

2. Associations

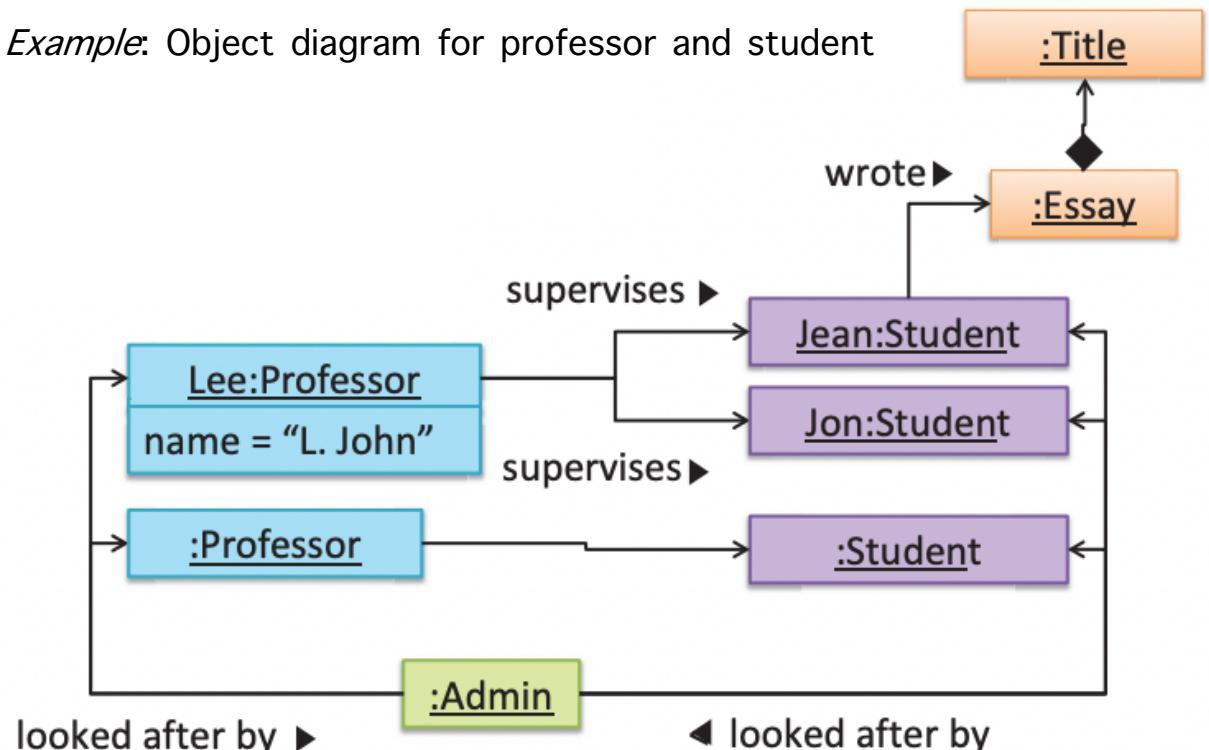


Use solid line to denote associations.

3. Object Diagrams VS Class Diagrams

- Object diagrams show objects instead of classes.
- Method compartments are omitted.
- Multiplicities are omitted.
- Multiple object diagrams can correspond to one class diagram.

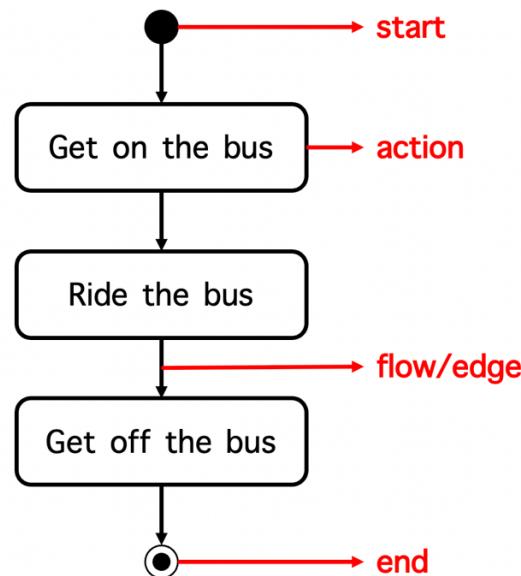
Example: Object diagram for professor and student



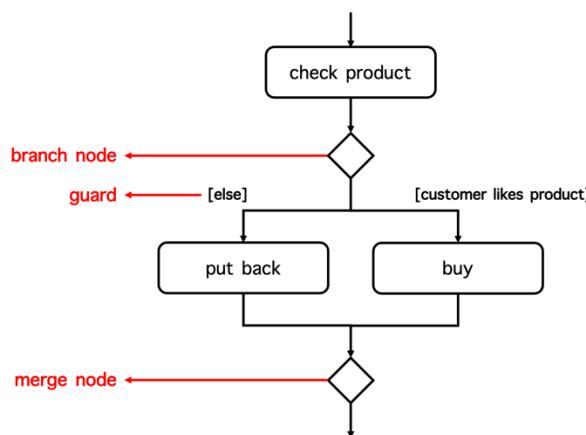
Appendix III: UML Activity Diagrams

UML activity diagrams are used to **model workflows**.

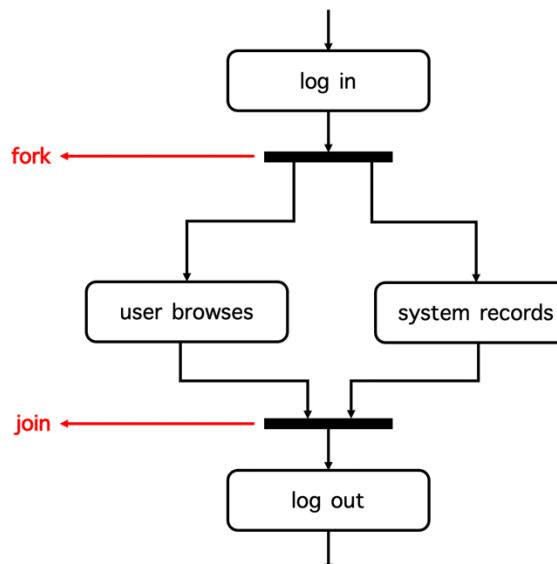
1. Linear Paths



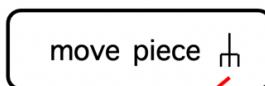
2. Alternate Paths



3. Parallel Paths

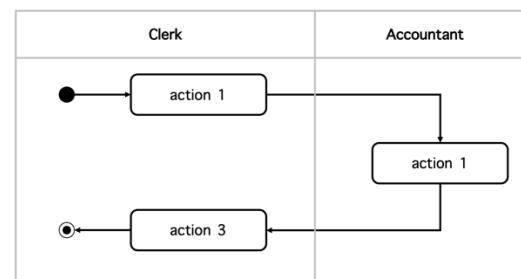


4. Rakes



Use rake symbols to indicate that this action is shown in a separate diagram.

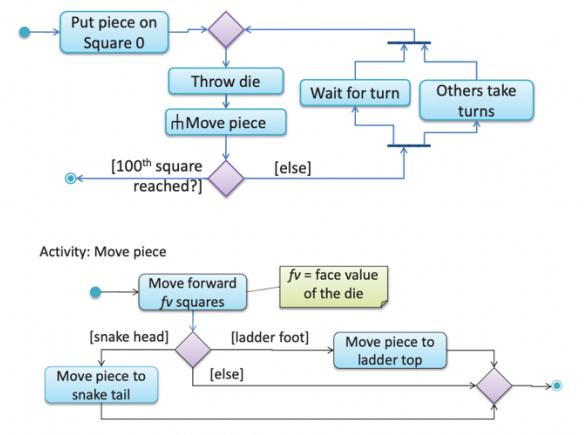
5. Swimlanes



Use swimlanes to partition an activity diagram to show who is doing which action.

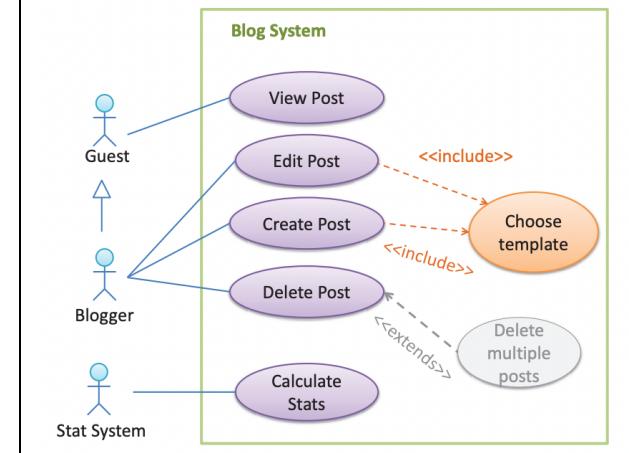
6. Combine

Example:



Appendix III*: UML Use Case Diagrams

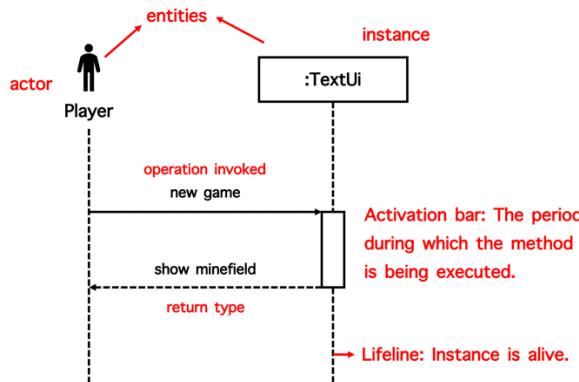
Example:



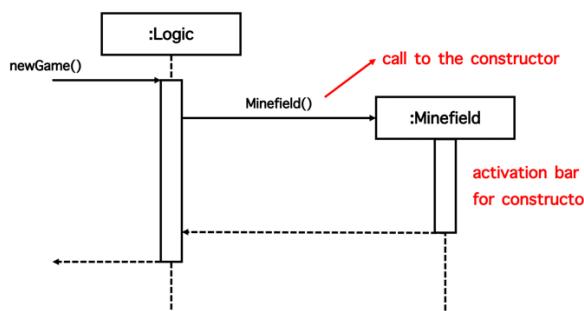
Appendix IV: UML Sequential Diagrams

UML sequence diagrams are used to **capture the interactions between multiple objects** for a given scenario.

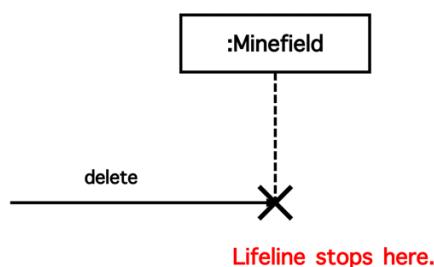
1. Basic Sequential Diagrams



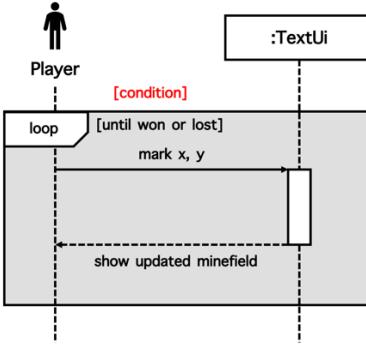
2. Object Construction



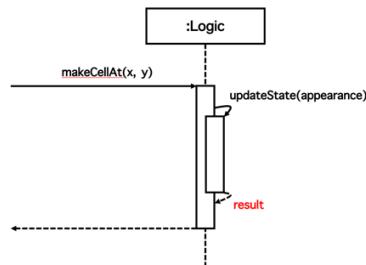
3. Object Deletion



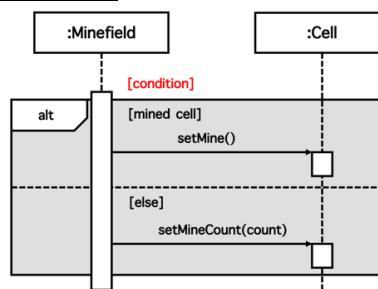
4. Loops



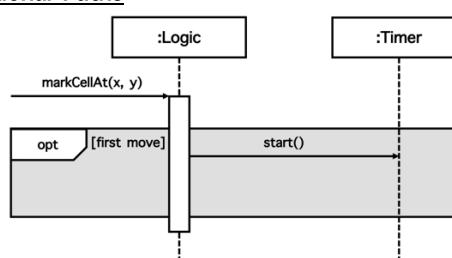
5. Self Invocation



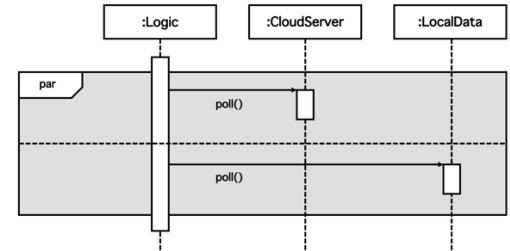
6. Alternative Paths



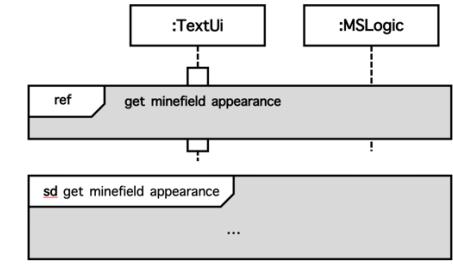
7. Optional Paths



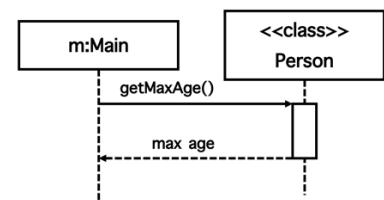
8. Parallel Paths



9. References Frames

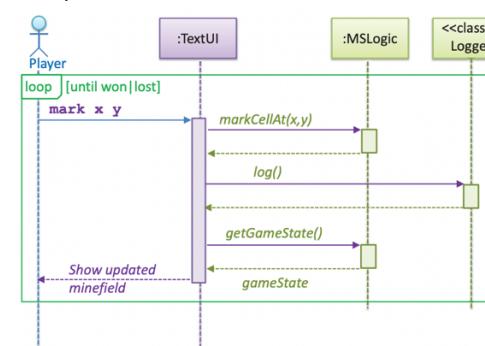


10. Calls to Static Methods



11. Combine

Example:



Appendix V: Java Coding Standard

1. Naming

- Names representing packages should be in all lower case.
- Class/enum names must be nouns and written in PascalCase.
- Variable name must be in camelCase.
- Constant name must be all uppercase using underscore to separate words.
- Names representing methods must be verbs and written in camelCase.
- All names should be written in English.
- Boolean variables/methods should be named to sound like booleans.
- Plural forms should be used on names representing a collection of objects.
- Iterator variables can be called i, j, k, etc.

2. Layout

- Basic indentation should be 4 spaces (not tabs).
- Line length should be no more than 120 characters.
- Indentation for wrapped lines should be 8 spaces.
- Use K&R style (aka. Egyptian style) brackets.
- Method definitions should have the following form:

```
1 public void someMethod() throws SomeException {
2     ...
3 }
```

- The if-else statements should have the following form:

```
1 if (condition) {
2     statements;
3 }
4 else {
5     statements;
6 }
```

```
1 if (condition) {
2     statements;
3 } else if (condition) {
4     statements;
5 } else {
6     statements;
7 }
```

- The for statements should have the following form:

```
1 for (initialization; condition; update) {
2     statements;
3 }
```

- The while statements should have the following form:

```
1 while (condition) {
2     statements;
3 }
```

- The do-while statements should have the following form:

```
1 do {
2     statements;
3 } while (condition);
```

- The switch statements should have the following form:

```
1 switch (condition) {
2     case ABC:
3         statements;
4         // Fallthrough
5     case DEF:
6         statements;
7         break;
8     case XYZ:
9         statements;
10    break;
11    default:
12        statements;
13        break;
14 }
```

- The try-catch statements should have the following form:

```
1 try {
2     statements;
3 } catch (Exception exception) {
4     statements;
5 } finally {
6     statements;
7 }
```

3. Statements

- Put every class in a package.
- Import classes should always be listed explicitly.
- Array specifiers must be attached to the type not the variable.
- The loop body should be wrapped by curly brackets irrespective of how many lines there are in the body.
- The conditional should be put on a separate line.
- Single statement conditionals should still be wrapped by curly brackets.

4. Comments

- All comments should be written in English.
- JavaDoc comments should have the following form:

```
1 /**
2 * Returns lateral location of the specified position.
3 * If the position is unset, NaN is returned.
4 *
5 * @param x X coordinate of position.
6 * @param y Y coordinate of position.
7 * @param zone Zone of position.
8 * @return Lateral location.
9 * @throws IllegalArgumentException If zone is <= 0.
10 */
11 public double computeLocation(double x, double y, int zone)
12     throws IllegalArgumentException {
13     //...
14 }
```

- Comments should be indented relative to their position in the code.

Appendix VI: Module Overview

