CS2103 Helpsheet

Requirements

Specifies a need to be fulfilled by software product/project.
Comes from stakeholders (party affected by project)
Software project may be:

- **Brown-field**
  Develop a product to replace/update an existing software product.
- **Green-field**
  Develop a totally new system with no precedent.

Two different types:

- **Functional Requirements**
  Specify what system should do.
- **Non-functional Requirements**
  Specify constraints under which system is developed and operated.
  Categories include data, environment, accessibility, compliance requirements.
  Example - Performance requirements:
  The system should respond within 2 seconds.

Prioritizing Requirements

Requirements can be prioritized based on importance and urgency.
Example: Essential - Typical - Novel,
High - Medium - Low

Quality of Requirements

Requirements should be, as a whole, consistent, non-redundant and complete.

Gathering Requirements

- **Brainstorming**
  Group activity to generate a large number of diverse and creative ideas for solution of a problem.
- **User Surveys**
  Used to solicit responses and opinions from a large number of stake holders regarding product
- **Observation**
  Uncover product requirements b y observing users in their natural work environment and obtaining usage data.
- **Focus Groups**
  Informal interview within an interactive group setting where the group is asked about their understanding of a certain topic/product.
- **Prototyping**
  Uncover requirements related to how users interact with system through UI mockups/prototypes.
- **Product Surveys**
  Study existing products to unearth shortcomings of existing solutions that can be addressed by new product.

Specifying Requirements

Prose

Normal textual description used to describe requirements such as vision of a product.

Feature List

A list of features grouped according to some criteria such as aspect, priority, order of delivery.

User Stories

Short, simple descriptions of a feature told from the perspective of the person who desires the new capability.

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

Examples
As a student, I can download files uploaded by lecturers, so that I can get my own copy of the files. As a forgetful user, I can view a password hint, so that I can recall my password.

User stories can be written at various levels.
Epic (themes) cover bigger functionality.
[Epic] As a lecturer, I can monitor student participation levels

- As a lecturer, I can view the forum post count of each student so that I can identify the activity level of students in the forum
- As a lecturer, I can view webcast view records of each student so that I can identify the students who did not view webcasts

Conditions of satisfaction to a user story can be added
Specify things that need to be true for user story implementation to be considered 'done'.

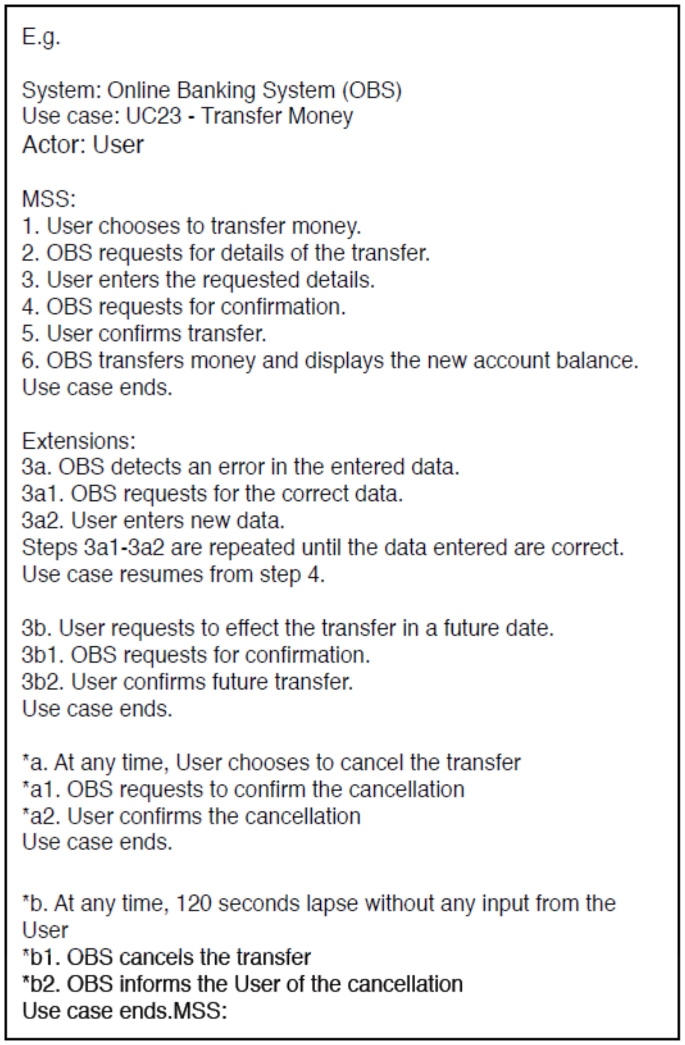
Additional info such as priority, size and urgency can be added to a user story.

Usage

- **User stories capture user requirements in a way that is convenient for scoping, estimation i.e. how much effort each feature will take and scheduling.**
- **Differs from traditional requirements specifications mainly in level of detail.**
- **Can capture non-functional requirements**
- **Handy for recording requirements during early requirements gathering**
  - Define the target user
  - Define the problem scope
  - Don't discard 'unusual' user stories hastily
  - Don't go into too much details
  - Don't be biased by preconceived product ideas
  - Don't discuss about how or who will do implementation

Use Cases

Describes an interaction between the user and the system for a specific functionality.



E.g.

System: Online Banking System (OBS)
Use 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 transfer.
6. OBS transfers 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.

*a. At any time, User chooses to cancel the transfer
*a1. OBS requests to confirm the cancellation
*a2. User confirms the cancellation
Use case ends.

*b. At any time, 120 seconds lapse without any input from the User
*b1. OBS cancels the transfer
*b2. OBS informs the User of the cancellation
Use case ends.MSS:

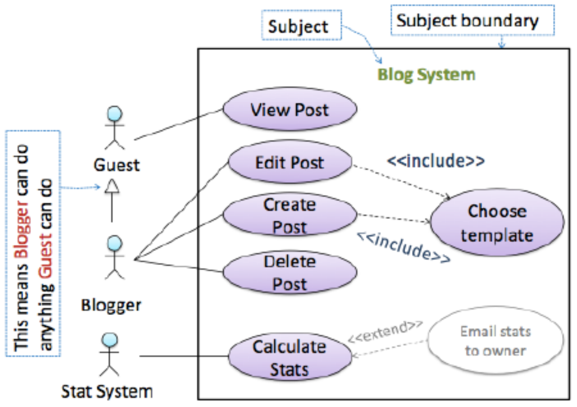
- **MSS** Main Success Scenario, describes the straightforward interaction for a given case.
- **Extensions** Alternative flow of events not expected by MSS
- **Inclusion** Inclusion of another use case: underline text and refer to use case number.
  E.g. 1. Staff creates survey (UC44)
  Include **precondition** to specify the state that the system is supposed to be in before the use case. Include **guarantees** to specify the state that the system promises at the end of operation.
- **Possible extensions**
  - Internet connection goes down
  - Session timeout
  - Invalid data entered
  - Data is being edited simultaneously

Actors in Use Cases

- 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 specif ied at various levels of detail.

Use Case Diagram

Provides an overview of a set of use cases.



⟨ ⟨ **extend** ⟩ ⟩ captures extensions of use cases and ⟨ ⟨ **include** ⟩ ⟩ captures inclusions of other use cases. Try not to set system as an actor.

Glossary

Serves to ensure that all stakeholders have a common understanding of the noteworthy terms, abbreviation, acronyms etc.

Supplementary Requirements

Used to capture requirements that do not fit elsewhere. Typically, this is where most Non Functional Requirements will be listed.

Design

Two main aspects:

- Product/external design:
  Designing the external behavior of the product to meet the users' requirements.
- Implementation/internal design
  This is usually done by software architects and software engineers.

Multi-level Design

Smaller systems: One level (AB-L3)
Bigger systems: Multiple levels (AB-L4)

Top-Down and Bottom-Up Design

Multi-level design can be done in a top-down manner, bottom-up manner, or as a mix.

- Top-down
  Design the high-level design first and flesh out the lower levels later. This is especially useful when designing big and novel systems where the high-level design needs to be stable before lower levels can be designed.
- Bottom-up
  Design lower level components first and put them together to create the higher-level systems later. This is not usually scalable for bigger systems. One instance where this approach might work is when designing a variations of an existing system or re-purposing existing components to build a new system.
- Mix
  Design the top levels using the top-down approach but switch to a bottom-up approach when designing the bottom later.

Design Principles

Abstraction

Technique for dealing with complexity. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level.

- Data abstraction
  Ignoring lower level data items and thinking in terms of bigger entities.
- Control abstraction
  Abstracts away details of the actual control flow to focus on tasks at a simplified level.

Coupling

Measure of the degree of dependence between components, classes, methods, etc.
X is coupled to Y if a change to Y can potentially require a change in X.
Example:

- X has access to the internal structure of Y
- X and Y depend on the same global variable
- X inherits from Y

Disadvantages of high coupling:

- Maintenance is harder because a change in one module could cause changes in other modules coupled to it (i.e. a ripple effect).
- Integration is harder because multiple components coupled with each other have to be integrated at the same time.
- Testing and reuse of the module is harder due to 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. Note that a child class is coupled to the parent class but not the other way around.
- Temporal coupling
  Two actions are bundled together just because they happen to occur at the same time e.g. extracting a contiguous block of code as a method although the code block contains statements unrelated to each other.

Cohesion

Measure of how strongly-related and focused the various responsibilities of a component are.
A highly-cohesive component keeps related functionalities together while keeping out all other unrelated things.

Disadvantages of low cohesion:

- Impedes the understandability of modules as it is difficult to express module functionalities at a higher level.
- Lowers maintainability because a module can be modified due to unrelated causes (reason: the module contains code unrelated to each other) or many many modules may need to be modified to achieve a small change in behavior (reason: because the code realated to that change is not localized to a single module).
- Lowers re-usability of modules because they do not represent logical units of functionality.

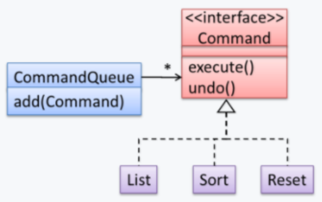
Present in many forms:

- Code related to a single concept is kept together, e.g. the Student component handles everything related to students.
- Code that is invoked close together in time is kept together, e.g. all code related to initializing the system is kept together.
- Code that manipulates the same data structure is kept together, e.g. the GameArchive component handles everything related to the storage and retrieval of game sessions.

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.

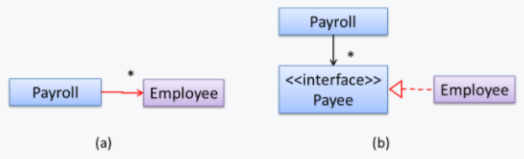
In the design given below, the behavior of the `CommandQueue` class can be altered by adding more concrete `Command` subclasses. For example, by including a `Delete` class alongside `List`, `Sort`, and `Reset`, the `CommandQueue` can now perform delete commands without modifying its code at all. That is, its behavior was extended without having to modify its code. Hence, it was open to extensions, but closed to modification.



Dependency Inversion Principle (DIP)

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

Example:



In design (a), the higher level class `Payroll` depends on the lower level class `Employee`, a violation of DIP. In design (b), both `Payroll` and `Employee` depends on the Payee interface (note that inheritance is a dependency).

Design (b) is more flexible (and less coupled) because now the `Payroll` class need not change when the `Employee` class changes.

Object Oriented Programming (OOP)

OOP is a programming paradigm. It groups operations and data into modular units called objects and lets you combine objects into structured networks to form a complete program. In an object-oriented programming language, objects and object interactions are the basic elements of design.

Object Oriented Programming (OOP) views the world as a network of interacting objects.

Objects

- Every object has both state (data) and behavior (operations on data).
- Every object has an interface and an implementation.
- Objects interact by sending messages.

Objects as Abstractions

Objects in OOP is an abstraction mechanism because it allows us to abstract away the lower level details and work with bigger granularity entities

Encapsulation

An object is an encapsulation of some data and related behavior in two aspects:

1. The packaging aspect
   An object packages data and related behavior together into one self-contained unit.
2. The information hiding aspect
   The data in an object is hidden from the outside world and are only accessible using the object's interface.

Classes

A class contains instructions for creating a specific kind of objects. Multiple objects have the same behavior because they are of the same kind. Instructions for creating a one kind (or class) of objects can be done in one go and use that same instructions to instantiate (i.e. create) objects of that kind.

Class Level Members

While all objects of a class has the same attributes, each object has its own copy of the attribute value.

- Class level attributes Variables whose value is shared by all instances of a class.
- Class level methods Methods that are called using the class instead of a specific instance.

Class-level attributes and methods are collectively called class-level members.

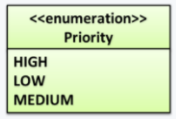
Enumerations

An Enumeration is a fixed set of values that can be considered as a data type.

An enumeration is often useful when using a regular data type such as int or String would allow invalid values to be assigned to a variable.

`Priority` can be considered an enumeration because it has only three values.

`Priority`: `HIGH`, `MEDIUM`, `LOW`



Associations

Connections between objects are called associations. Associations can be reflected among classes too. An association in a class diagram can use association labels and association roles to show additional info. (Refer to UML Section)

Navigability

When two classes are linked by an association, it does not necessarily mean both classes know about each other. The concept of which class in the association knows about the other class is called navigability.

Multiplicity

Multiplicity is the aspect of an OOP solution that dictates how many objects take part in each association.

Dependencies

A dependency is a weaker form of an association where there is an interactions between objects that do not result in a long-term relationship between the said objects.

Composition

A composition is an association that represents a strong whole-part relationship. When the whole is destroyed, parts are destroyed too.
Composition also implies that there cannot be cyclical links.

Aggregation

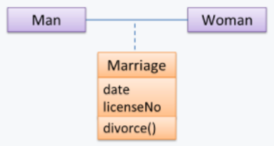
Aggregation represents a container-contained relationship. It is a weaker relationship than composition.

The distinction between composition and aggregation is rather blurred. Omitting the aggregation symbol altogether is encouraged because using it adds more confusion than clarity.

Association Classes

An association class represents additional information about an association. It is a normal class but plays a special role from a design point of view.

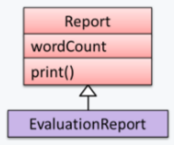
A `Man` class and a `Woman` class is linked with a 'married to' association and there is a need to store the date of marriage. However, that data is related to the association rather than specifically owned by either the `Man` object or the `Woman` object. In such situations, an additional association class can be introduced, e.g. a `Marriage` class, to store such information.



Inheritance

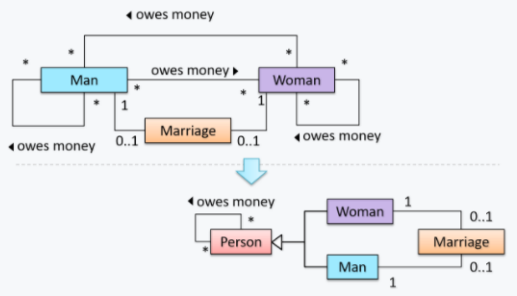
The OOP concept Inheritance allows you to define a new class based on an existing class.

Example: The `EvaluationReport` inherits the `wordCount` attribute and the `print()` method from the *base class* `Report`.



- A super class is said to be more general than the sub class.
  Conversely, a sub class is said to be more specialized than the super class.
- Applying inheritance on a group of similar classes can result in the common parts among classes being extracted into more general classes.

`Man` and `Woman` behaves the same way for the 'owes money' association. However, the two classes cannot be simply replaced with a more general class `Person` because of the need to distinguish between `Man` and `Woman` for the 'marriage' association. A solution is to add the `Person` class as a super class and let `Man` and `Woman` inherit from `Person`.



- Inheritance results in an **is a** relationship.
- Inheritance relationships through a chain of classes can result in inheritance hierarchies (aka inheritance trees).
- Multiple Inheritance is when a class inherits directly from multiple classes.
  Multiple inheritance is allowed among C++ classes but not among Java classes.

Method Overriding

Sub-class changes the behavior inherited from the parent class by re-implementing the method. Overridden methods have the same name, same type signature, and same return type.

Method Overloading

Multiple methods with the same name but different type signatures. Overloading is used to indicate that multiple operations do similar things but take different parameters.

Interfaces

An interface is a behavior specification i.e. a collection of method specifications. If a class implements the interface, it means the class is able to support the behaviors specified by the said interface.

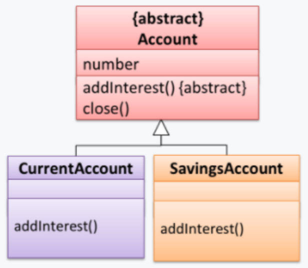
Class implementing an interface results in an **is-a** relationship.

Abstract Classes

An abstract class is a class that is declared abstractit may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is simply the method interface without the implementation.

The `Account` class has an abstract method (`addInterest()`).



Dynamic and Static Binding

- Dynamic Binding
  Overridden operations are resolved at runtime. That is, the runtime decides which version of the operation should be executed based on the actual type of the receiving object.
- Static Binding
  Overloaded methods are resolved at compile time, also called static binding (also called early binding and compile-time binding.)

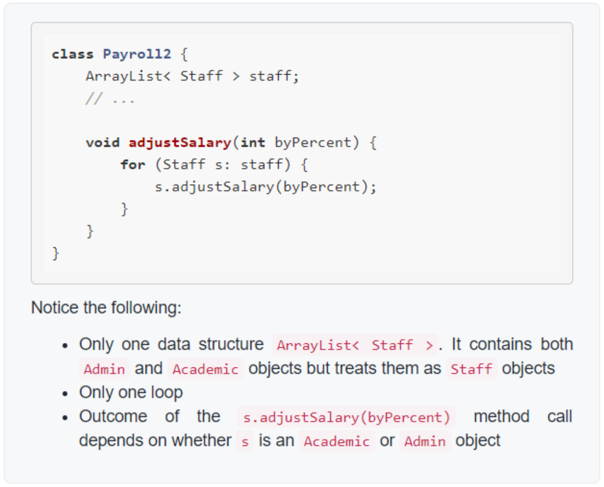
Substitutability

Wherever an object of the superclass is expected, it can be substituted by an object of any of its subclasses.

For example, an Academic is an instance of a Staff, but a Staff is not necessarily an instance of an Academic. However, an Academic can be used in place of a Staff. As a result, inheritance allows substitutability.

Polymorphism

The ability of different objects to respond, each in its own way, to identical messages is called polymorphism.



```
class Payroll2 {
    ArrayList< Staff > staff;
    // ...

    void adjustSalary(int byPercent) {
        for (Staff s: staff) {
            s.adjustSalary(byPercent);
        }
    }
}
```

Notice the following:

- Only one data structure `ArrayList< Staff >`. It contains both `Admin` and `Academic` objects but treats them as `Staff` objects
- Only one loop
- Outcome of the `s.adjustSalary(byPercent)` method call depends on whether `s` is an `Academic` or `Admin` object

Mechanism

There are three issues that are at the center of how polymorphism is achieved: substitutability, operation overriding, and dynamic binding.

- Substitutability
  Because of substitutability, we can write code that expects object of a parent class and yet use that code with objects of child classes. That is how polymorphism is able to treat objects of different types as one type.
- Overriding
  To get polymorphic behavior from an operation, the operation in the superclass needs to be overridden in each of the subclasses. That is how overriding allows objects of different sub classes to display different behaviors in response to the same method call.
- Dynamic binding
  Overridden methods are bound to the method implementation dynamically during the runtime. That is how the code can call the method of the parent class and yet execute the implementation of the child class.

Miscellaneous

Difference between class, abstract class and an interface:

- An interface is a behavior specification with no implementation.
- A class is a behavior specification + implementation.
- An abstract class is a behavior specification + a possibly incomplete implementation.

Difference between overriding and overloading:

- Overloading is used to indicate that multiple operations do similar things but take different parameters. Overloaded methods have the same method name but different method signatures and possibly different return types.
- Overriding is when a sub-class redefines an operation using the same method name and the same type signature. Overridden methods have the same name, same method signature, and same return type.

Modeling

- A model is anything used in any way to represent anything else.
  A class diagram is a model that represents an OOP software design and is drawn using the UML modeling notation.
- A model provides a simpler view of a complex entity because a model captures only a selected aspect.
  A class diagram captures the class structure of the software design but not the runtime behavior.
- Multiple models of the same entity may be needed to capture it fully.
  In addition to the class diagram, a number of sequence diagrams may need to be created to capture various interesting interaction scenarios the software undergoes.

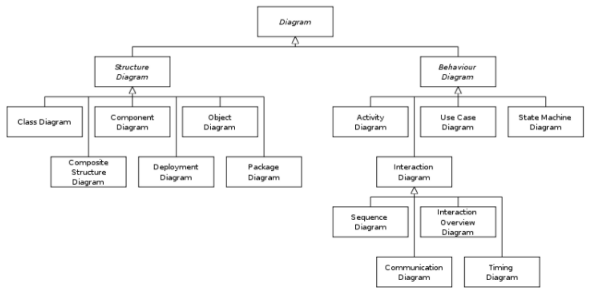
In software development, models are useful in several ways:

- To analyze a complex entity related to software development.
  E.g Architecture diagram, models of problem domain
- To communicate information among stakeholders.
  Models can be used as a visual aid in discussions and documentations.
- As a blueprint for creating software.
  Models can be used as instructions for building software.

Model-driven development (MDD)

An approach to software development that strives to exploits models as blueprints.
MDD uses models as primary engineering artifacts when developing software. That is, the system is first created in the form of models. After that, the models are converted to code using code-generation techniques (usually, automated or semi-automated, but can even involve manual translation from model to code). MDD requires the use of a very expressive modeling notation (graphical or otherwise), often specific to a given problem domain.

UML Models

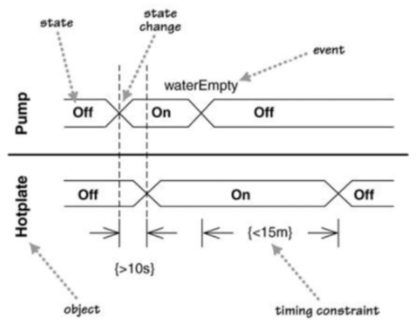


Modeling Structures

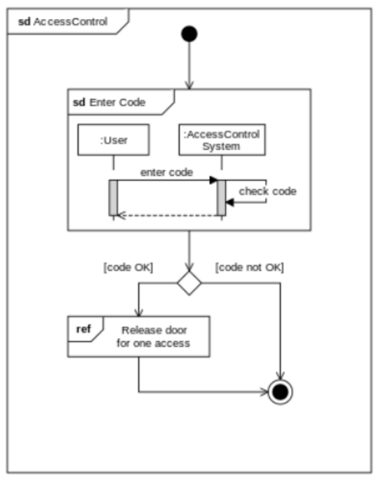
- Class Diagrams
  See UML Section
- Object Diagrams
  See UML Section
- Object Oriented Domain Models (OODM)
  Class diagrams can also be used to model objects in the problem domain (i.e. to model how objects actually interact in the real world, before emulating them in the solution). When used to model the problem domain, such class diagrams are called conceptual class diagrams or OO domain models (OODMs).
  - OODMs do not contain solution-specific classes
    Classes that are used in the solution domain but do not exist in the problem domain
  - OODMs, just like a class diagram, represents the class structure of the problem domain and not their behavior.
  - OODM notation is similar to class diagram notation but typically omit methods and navigability.
- Deployment Diagrams
  Shows a system's physical layout, revealing which pieces of software run on which pieces of hardware.
- Component Diagrams
  A UML component diagram is used to show how a system is divided into components and how they are connected to each other through interfaces.
- Package Diagrams
  Shows packages and their dependencies. A package is a grouping construct for grouping UML elements (classes, use cases, etc.).
- Composite Structure Diagrams
  A composite structure diagram hierarchically decomposes a class into its internal structure.

Modeling Behaviors

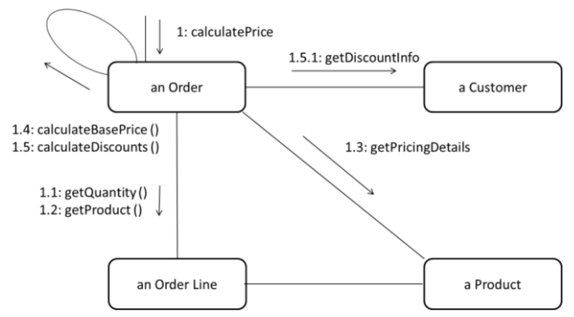
- Activity Diagrams
- Sequence Diagrams
- Use Case Diagrams
- Timing Diagrams
  Timing diagrams focus on timing constraints.



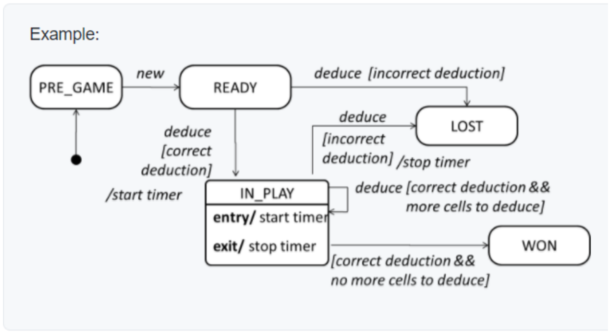
- Interaction Overview Diagrams
  Interaction overview diagrams are a combination of activity diagrams and sequence diagrams.



- Communication Diagrams
  Communication diagrams are like sequence diagrams but emphasize the data links between the various participants in the interaction rather than the sequence of interactions.



- State Machine Diagrams
  State Machine Diagrams model state-dependent behavior.
  An SMD views the life-cycle of an object as consisting of a finite number of states where each state displays a unique behavior pattern. An SMD captures information such as the states an object can be in, during its lifetime, and how the object responds to various events while in each state and how the object transits from one state to another. In contrast to sequence diagrams that capture object behavior one scenario at a time, SMDs capture the objects behavior over its full life cycle.

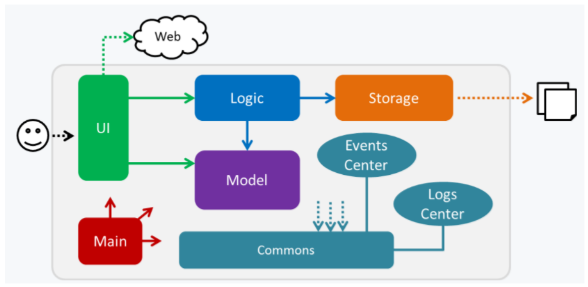

Example:
PRE_GAME
new
READY
deduce [incorrect deduction]
deduce [correct deduction]
/start timer
deduce
[incorrect deduction] /stop timer
LOST
IN_PLAY
entry/ start timer
exit/ stop timer
deduce [correct deduction && more cells to deduce]
WON
[correct deduction && no more cells to deduce]

Software Architecture

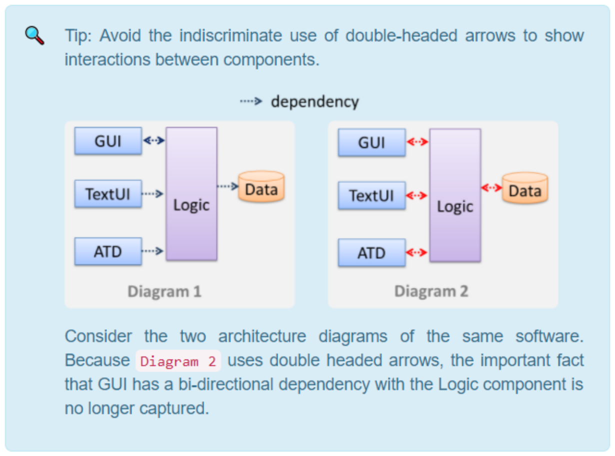
The software architecture shows the overall organization of the system and can be viewed as a very high-level design.

Architecture Diagrams

Architecture diagrams are free-form diagrams.



Drawing



Tip: Avoid the indiscriminate use of double-headed arrows to show interactions between components.

Consider the two architecture diagrams of the same software. Because `Diagram 2` uses double headed arrows, the important fact that GUI has a bi-directional dependency with the Logic component is no longer captured.

Architectural Styles

Software architectures follow various high-level styles (aka architectural patterns).

N-tier Architectural Style

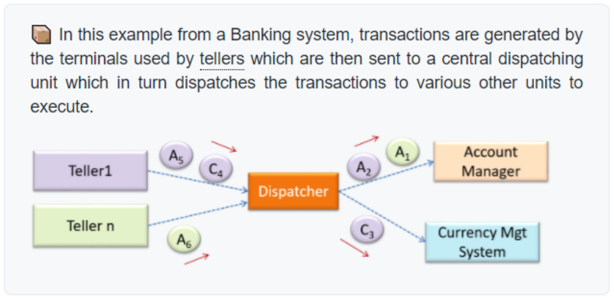
In the n-tier style, higher layers make use of services provided by lower layers. Lower layers are independent of higher layers. Other names: multi-layered, layered.

Client-Server Architectural Style

The client-server style has at least one component playing the role of a server and at least one client component accessing the services of the server. This is an architectural style used often in distributed applications.

Transaction Processing Architectural Style

The transaction processing 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. Task queuing, ordering, undo etc. are handled by the dispatcher.


In this example from a Banking system, transactions are generated by the terminals used by tellers which are then sent to a central dispatching unit which in turn dispatches the transactions to various other units to execute.

Service-oriented Architectural Style

The service-oriented architecture (SOA) style builds applications by combining functionalities packaged as programmatically accessible services.

SOA aims to achieve interoperability between distributed services, which may not even be implemented using the same programming language. A common way to implement SOA is through the use of XML web services where the web is used as the medium for the services to interact, and XML is used as the language of communication between service providers and service users.

Event-driven Architectural Style

Event-driven style controls the flow of the application by detecting events and communicating those events to other interested parts of the code for possible actions. This architectural style is often used in GUIs.

Others

Other well-known architectural styles include the pipes-and-filters architectures, the broker architectures, the peer-to-peer architectures, and the message-oriented architectures.
Most applications use a mix of these architectural styles.

Software Design Patterns

In software development, there are certain problems that recur in a certain context. After repeated attempts at solving such problems, better solutions are discovered and refined over time and these solutions are known as design patterns.

Format

- Context
  The situation or scenario where the design problem is encountered.
- Problem
  The main difficulty to be resolved. The criteria for a good solution are also identified to evaluate solutions.
- Solution
  The core of the solution. It is important to note that the solution presented only includes the most general constraints, which may need further refinement for a specific context.
- Anti-patterns (optional)
  Commonly used solutions, which are usually incorrect and/or inferior to the Design Pattern.
- Consequences (optional)
  Identifying the pros and cons of applying the pattern.
- Other useful information (optional)
  Code examples, known uses, other related patterns, etc.

Singleton Pattern

Context

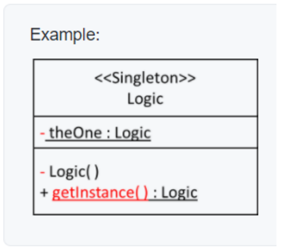
Certain classes should have no more than just one instance (e.g. the main controller class of the system). These single instances are commonly known as singletons.

Problem

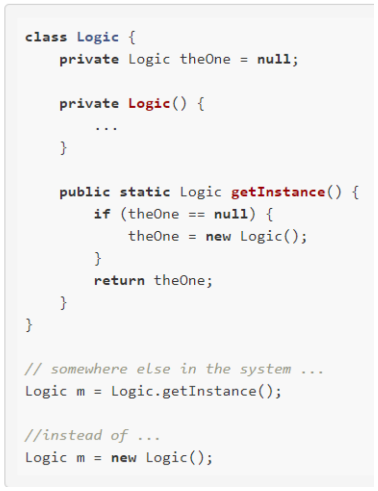
A normal class can be instantiated multiple times by invoking the constructor.

Solution

The key insight of the solution is that the constructor of the singleton class cannot be public. Because a public constructor will allow others to instantiate the class at will, a private constructor should be used instead. In addition, a public class-level method is provided to access the single instance.



Implementation


```
class Logic {
    private Logic theOne = null;

    private Logic() {
        ...
    }

    public static Logic getInstance() {
        if (theOne == null) {
            theOne = new Logic();
        }
        return theOne;
    }
}

// somewhere else in the system ...
Logic m = Logic.getInstance();

//instead of ...
Logic m = new Logic();
```

- The constructor is private, which prevents instantiation from outside the class.
- The single instance of the singleton class is maintained by a private class-level variable.
- Access to this object is provided by a public class-level operation getInstance() which instantiates a single copy of the singleton class when it is executed for the first time. Subsequent calls to this operation return the single instance of the class.

Evaluation

Pros:

- 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.
- 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 (static methods cannot be overridden)
- In testing, singleton objects carry data from one test to another even when we want each test to be independent of the others.

It is recommended that you apply the Singleton pattern when, in addition to requiring only one instance of a class, there is a risk of creating multiple objects by mistake, and creating such multiple objects has real negative consequences.

Abstraction Occurrence Pattern

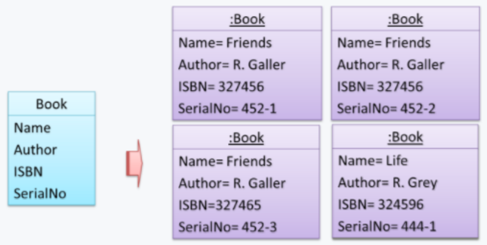
Context

There is a group of similar entities that appears to be occurrences (or copies) of the same thing, sharing lots of common information, but also differing in significant ways.

Problem

Representing the objects mentioned previously as a single class would be problematic because it results in duplication of data which can lead to inconsistencies in data (if some of the duplicates are not updated consistently).

Take for example the problem of representing books in a library. Assume that there could be multiple copies of the same title, bearing the same ISBN number, but different serial numbers.



The above solution requires common information to be duplicated by all instances. This will not only waste storage space, but also creates a consistency problem. Suppose that after creating several copies of the same title, the librarian realized that the author name was wrongly spelt. To correct this mistake, the system needs to go through every copy of the same title to make the correction. Also, if a new copy of the title is added later on, the librarian (or the system) has to make sure that all information entered is the same as the existing copies to avoid inconsistency.

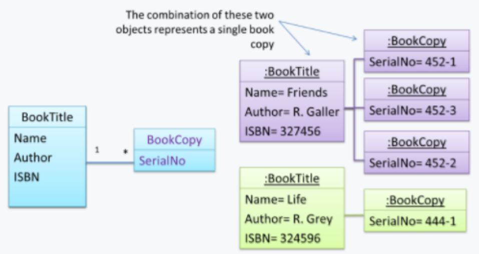
Anti-pattern

Segregate common and unique information into a class hierarchy. Let a copy of an entity (e.g. a copy of a book)be represented by two objects instead of one, separating the common and unique information into two classes to avoid duplication.

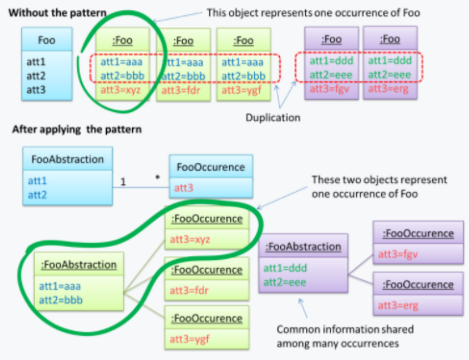
Solution

Let a book copy be represented by two objects instead of one, as given below.

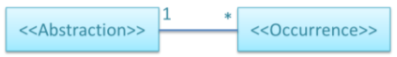
Given below is how the pattern is applied to the Library example:



Here's a more generic example:



The general idea can be found in the following class diagram:

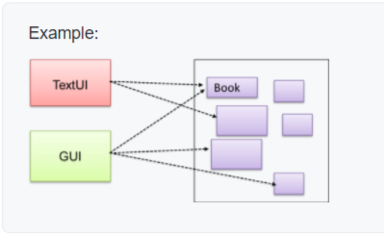


The Abstraction class should hold all common information, and the unique information should be kept by the Occurrence class.

Facade Pattern

Context

Components need to access functionality deep inside other components. For example, the UI component of a Library system might want to access functionality of the Book class contained inside the Logic component.

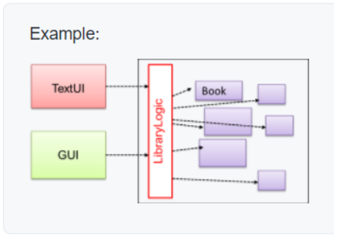


Problem

Access to the component should be allowed without exposing its internal details. For example, the UI component should access the functionality of the Logic component without knowing that it contained a Book class within it.

Solution

Include a Facade class that sits between the component internals and users of the component such that all access to the component happens through the Facade class. The following class diagram shows the application of the Facade pattern to the Library System example. In this example, the LibraryLogic class acts as the Facade class.



Command Pattern

Context

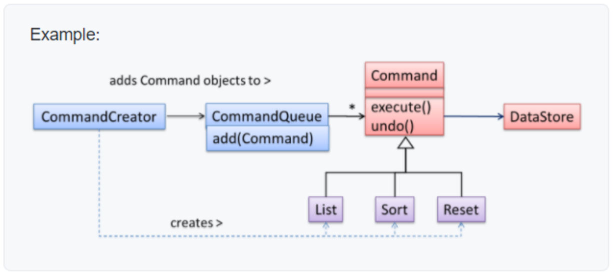
A system is required to execute a number of commands, each doing a different task. For example, a system might have to support Sort, List, Reset commands.

Problem

It is preferable that some part of the code execute these commands without having to know each command type. For example, there can be a CommandQueue object that is responsible for queuing commands and executing them without knowledge of what each command does.

Solution

In the example solution below, the CommandCreator creates List, Sort, and Reset Command objects and adds them to the CommandQueue object. The CommandQueue object treats them all as Command objects and performs the execute/undo operation on each of them without knowledge of the specific Command type. When executed, each Command object will access the DataStore object to carry out its task. The Command class can also be an abstract class or an interface.



Model View Controller (MVC) Pattern

Context

Most applications support storage/retrieval of information, displaying of information to the user (often via multiple UIs having different formats), and changing stored information based on external inputs.

Problem

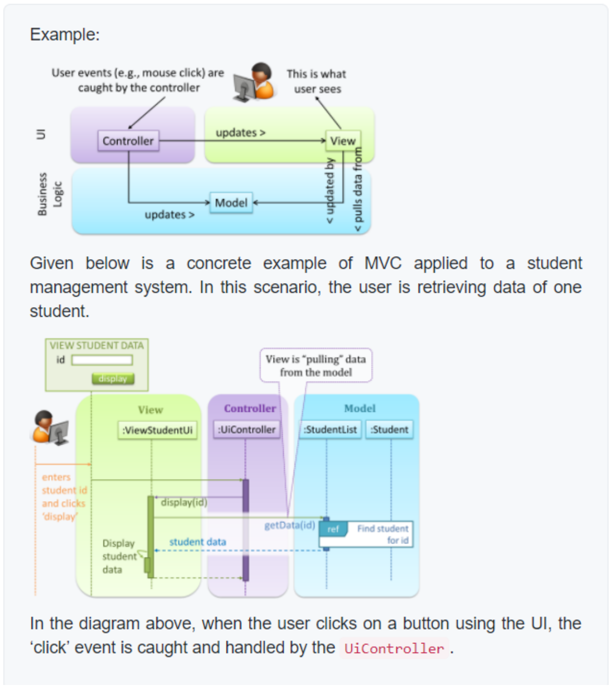
To reduce coupling resulting from the interlinked nature of the features described above.

Solution

To decouple data, presentation, and control logic of an application by separating them into three different components: Model, View and Controller.

- **View:** Displays data, interacts with the user, and pulls data from the model if necessary.
- **Controller:** Detects UI events such as mouse clicks, button pushes and takes follow up action. Updates/changes the model/view when necessary.
- **Model:** Stores and maintains data. Updates views if necessary.

The relationship between the components can be observed in the diagram below. Typically, the UI is the combination of view and controller.

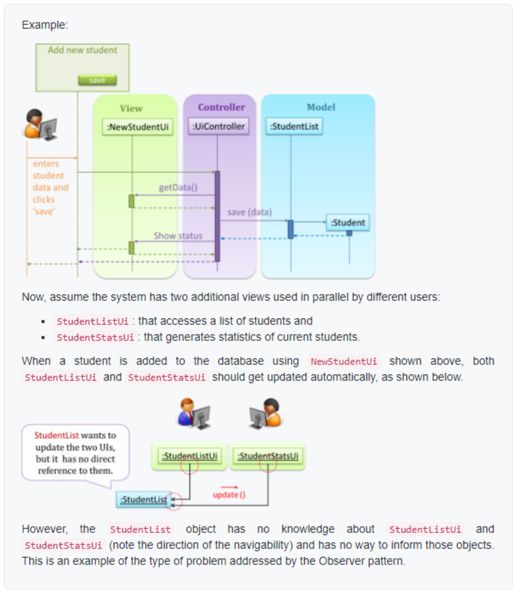


Given below is a concrete example of MVC applied to a student management system. In this scenario, the user is retrieving data of one student.

In the diagram above, when the user clicks on a button using the UI, the 'click' event is caught and handled by the `UiController`.

Observer Pattern

What



Now, assume the system has two additional views used in parallel by different users:

- `StudentListUi` : that accesses a list of students and
- `StudentStatsUi` : that generates statistics of current students.

When a student is added to the database using `NewStudentUi` shown above, both `StudentListUi` and `StudentStatsUi` should get updated automatically, as shown below.

StudentList wants to update the two UIs, but it has no direct reference to them.
:StudentListUi
:StudentStatsUi
:StudentList
update ()

However, the `StudentList` object has no knowledge about `StudentListUi` and `StudentStatsUi` (note the direction of the navigability) and has no way to inform those objects. This is an example of the type of problem addressed by the Observer pattern.

Context

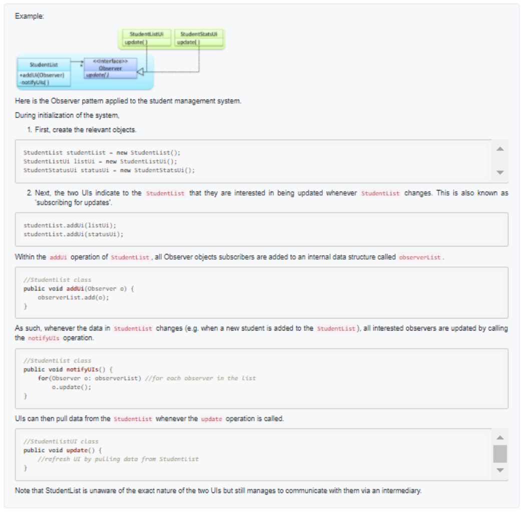
An object (possibly, more than one) is interested to get notified when a change happens to another object. That is, some objects want to observe another object.

Problem

A bidirectional link between the two objects is not desirable. However the two entities need to communicate with each other. That is, the observed object does not want to be coupled to objects that are observing it.

Solution

The Observer pattern shows us how an object can communicate with other objects while avoiding a direct coupling with them. The solution is to force the communication through an interface known to both parties. A concrete example is given below.



Here is the Observer pattern applied to the student management system.

During initialization of the system,

1. First, create the relevant objects.

```
StudentList studentList = new StudentList();
StudentListUi listUi = new StudentListUi();
StudentStatusUi statusUi = new StudentStatsUi();
```

2. Next, the two UIs indicate to the `StudentList` that they are interested in being updated whenever `StudentList` changes. This is also known as 'subscribing for updates'.

```
studentList.addUi(listUi);
studentList.addUi(statusUi);
```

Within the `addUi` operation of `StudentList`, all Observer objects subscribers are added to an internal data structure called `observerList`.

```
//StudentList class
public void addUi(Observer o) {
    observerList.add(o);
}
```

As such, whenever the data in `StudentList` changes (e.g. when a new student is added to the `StudentList`), all interested observers are updated by calling the `notifyUIs` operation.

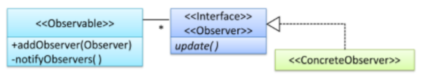
```
//StudentList class
public void notifyUIs() {
    for(Observer o: observerList) //for each observer in the list
        o.update();
}
```

UIs can then pull data from the `StudentList` whenever the `update` operation is called.

```
//StudentListUI class
public void update() {
    //refresh UI by pulling data from StudentList
}
```

Note that StudentList is unaware of the exact nature of the two UIs but still manages to communicate with them via an intermediary.

Here is the generic description of the observer pattern:



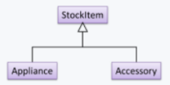
- Observer is an interface: any class that implements it can observe an Observable. Any number of Observer objects can observe (i.e. listen to changes of) the Observable object.
- The Observable¿ maintains a list of Observer objects. addObserver(Observer) operation adds a new Observer to the list of Observers.
- Whenever there is a change in the Observable, the notifyObservers() operation is called that will call the update() operation of all Observers in the list.

Combining Design Patterns

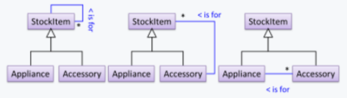
Example case study on the Stock Inventory System (SIS) of a shop.

Example:

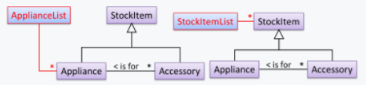
A `StockItem` can be an Appliance or an Accessory.



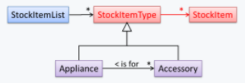
To track that each `Accessory` is associated with the correct `Appliance` , consider the following alternative class structures.



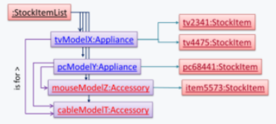
The third one seems more appropriate (the second one is suitable if accessories can have accessories). Next, consider between keeping a list of `Appliances` , and a list of `StockItems` . Which is more appropriate?



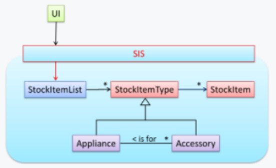
The latter seems more suitable because it can handle both appliances and accessories the same way. Next, an abstraction occurrence pattern is applied to keep track of `StockItems` .



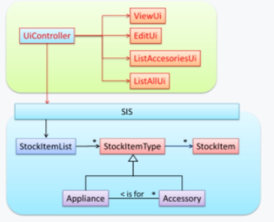
Note the inclusion of navigabilities. Here's a sample object diagram based on the class model created thus far.



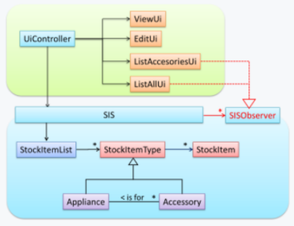
Next, apply the façade pattern to shield the SIS internals from the UI.



As UI consists of multiple views, the MVC pattern is applied here.



Some views need to be updated when the data change; apply the Observer pattern here.



In addition, the Singleton pattern can be applied to the façade class.

Other Design Patterns

Creational

About object creation. They separate the operation of an application from how its objects are created. Abstract Factory, Builder, Factory Method, Prototype, Singleton

Structural

About the composition of objects into larger structures while catering for future extension in structure. Adapter, Bridge, Composite, Decorator, Faade, Flyweight, Proxy

Defining how objects interact and how responsibility is distributed among them. Chain of Responsibility, Command, Interpreter, Template Method, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor

Implementation

IDEs

IDEs support all development-related work within the same tool.

IDEs generally consist of:

- A source code editor
- A compiler and/or an interpreter (together with other build automation support) that facilitates the compilation/linking/running/deployment of a program.
- A debugger that allows the developer to execute the program one step at a time to observe the run-time behavior in order to locate bugs.
- 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 modeling support.

Examples of popular IDEs: Eclipse, Intellij IDEA, NetBeans, Visual Studio, DevC++, DrJava, XCode

Debugging

Debugging is the process of discovering defects in the program. Here are some approaches to debugging:

- By inserting temporary print statements
- By manually tracing through the code
- Using a debugger
  A debugger tool allows you to pause the execution, then step through one statement at a time while examining the internal state if necessary. Most IDEs come with an inbuilt debugger. This is the recommended approach for debugging.

Code Quality

Guideline: Maintain Readability

Among various dimensions of code quality, such as run-time efficiency, security, and robustness, one of the most important is understandability. This is because in any non-trivial software project, code needs to be read, understood, and modified by other developers later on. Even if we do not intend to pass the code to someone else, code quality is still important because we all become 'strangers' to our own code someday.

Basic Guidelines

- Avoid Long Methods
- Avoid Long Methods (¡= 3 levels of indentation)
- Avoid Complicated Expressions
  Avoid complicated expressions, especially those having many negations and nested parentheses. If you must evaluate complicated expressions, have it done in steps (i.e. calculate some intermediate values first and use them to calculate the final value).
- Avoid Magic Numbers
- Make Code Obvious Make the code as explicit as possible, even if the language syntax allows them to be implicit. Here are some examples:
  - Use explicit type conversion instead of implicit type conversion.
  - Use parentheses/braces to show grouping even when they can be skipped.
  - Use enumerations when a certain variable can take only a small number of finite values. For example, instead of declaring the variable 'state' as an integer and using values 0,1,2 to denote the states 'starting', 'enabled', and 'disabled' respectively, declare 'state' as type SystemState and define an enumeration SystemState that has values 'STARTING', 'ENABLED', and 'DISABLED'.
  - When statements should follow a particular order, try to make it obvious (with appropriate naming, or at least comments). For example, if you name two functions 'taskA()' and 'taskB()', it is not

obvious which one should be called first. Contrast this with naming the functions 'phaseOne()' and 'phaseTwo()' instead. This is especially important when statements in one function must be called before the other one.

Intermediate Guidelines

- Structure Code Logically
  Lay out the code so that it adheres to the logical structure. The code should read like a story. Just like we use section breaks, chapters and paragraphs to organize a story, use classes, methods, indentation and line spacing in your code to group related segments of the code.
- Don't 'Trip Up' Reader
  Avoid confusing code such as unused parameters, data flow anomalies, multiple statements in the same line
- Practice KISSing
  As the old adage goes, "keep it simple, stupid (KISS). Do not try to write clever code. For example, do not dismiss the brute-force yet simple solution in favor of a complicated one because of some supposed benefits such as 'better reusability' unless you have a strong justification.
- Avoid Premature Optimizations Optimizing code prematurely has several drawbacks:
  - We may not know which parts are the real performance bottlenecks.
  - Optimizing can complicate the code, affecting correctness and understandability
  - Hand-optimized code can be harder for the compiler to optimize (the simpler the code, the easier for the compiler to optimize it).
- SLAP (Single Level of Abstraction) Hard
  Avoid varying the level of abstraction within a code fragment.

Advanced Guidelines Make the Happy Path Prominent: The happy path (i.e. the execution path taken when everything goes well) should be clear and prominent in your code. Restructure the code to make the happy path unindented as much as possible. It is the unusual cases that should be indented. Someone reading the code should not get distracted by alternative paths taken when error conditions happen. One technique that could help in this regard is the use of guard clauses.

Guideline: Follow a Standard

The aim of a coding standard is to make the entire code base look like it was written by one person.

Guideline: Name Well

Proper naming improves the readability. It also reduces bugs caused by ambiguities regarding the intent of a variable or a method.

Basic

- Use Nouns for Things and Verbs for Actions
- Use Standard Words
  Use correct spelling in names. Avoid 'texting-style' spelling. Avoid foreign language words, slang, and names that are only meaningful within specific contexts/timese.g. terms from private jokes, a TV show currently popular in your country
- Use Name to Explain
  A name is not just for differentiation; it should explain the named entity to the reader accurately and at a sufficient level of detail.
- Not Too Long, Not Too Short
  While it is preferable not to have lengthy names, names that are 'too short' are even worse. If you must abbreviate or use acronyms, do it consistently. Explain their full meaning at an obvious location.
- Avoid Misleading Names
  Related things should be named similarly, while unrelated things should NOT.

Guideline: Avoid Unsafe Shortcuts

It is safer to use language constructs in the way they are meant to be used, even if the language allows shortcuts. Some such coding practices are common sources of bugs. Know them and avoid them.

Basic

- Use the Default Branch
  Always include a default branch in case statements.

  Furthermore, use it for the intended default action and not just to execute the last option. If there is no default action, you can use the 'default' branch to detect errors (i.e. if execution reached the default branch, throw an exception).
- Don't Recycle Variables or Parameters
  Use one variable for one purpose. Do not reuse a variable for a different purpose other than its intended one, just because the data type is the same. Do not reuse formal parameters as local variables inside the method.
- Avoid Empty Catch Blocks
  Never write an empty catch statement. At least give a comment to explain why the catch block is left empty.

Intermediate

- Minimise Scope of Variables
  Minimize global variables. Global variables may be the most convenient way to pass information around, but they do create implicit links between code segments that use the global variable. Avoid them as much as possible. Define variables in the least possible scope.
- Minimise Code Duplication
  Code duplication, especially when you copy-paste-modify code, often indicates a poor quality implementation. While it may not be possible to have zero duplication, always think twice before duplicating code; most often there is a better alternative.

Guideline: Comment Minimally, But Sufficiently

Basic

- Do Not Repeat the Obvious
- Write to the Reader

Intermediate

- Explain WHAT and WHY, not HOW
  Comments should explain what and why aspect of the code, rather than the how aspect.
  - What : The specification of what the code supposed to do. The reader can compare such comments to the implementation to verify if the implementation is correct
  - Why : The rationale for the current implementation.
  - How : The explanation for how the code works. This should already be apparent from the code, if the code is self-explanatory. Adding comments to explain the same thing is redundant.

Refactoring

Process of improving a program's internal structure in small steps without modifying its external behavior.

- Refactoring is not rewriting
  Discarding poorly-written code entirely and re-writing it from scratch is not refactoring because refactoring needs to be done in small steps.
- Refactoring is not bug fixing
  By definition, refactoring is different from bug fixing or any other modifications that alter the external behavior (e.g. adding a feature) of the component in concern.

Improving code structure can have many secondary benefits: e.g.

- Hidden bugs become easier to spot.
- Improve performance (sometimes, simpler code runs faster than complex code because simpler code is easier for the compiler to optimize).

Two common refactorings: Consolidate Duplicate Conditional Fragments and Extract Method

When

It is too much refactoring when the benefits no longer justify the cost. The costs and the benefits depend on the context. That is why some refactorings are opposites of each other (e.g. extract method vs inline method).

Documentation

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

- Documentation for developer-as-user: Software components are written by developers and used by other developers. Therefore, there is a need to document how the components are to be used. API documents form the bulk of this category.
- Documentation for developer-as-maintainer: There is a need to document how a system or a component is designed, implemented and tested so that other developers can maintain and evolve the code.

Guidelines

- Guideline: Go Top-down, Not Bottom-up
  When writing project documents, a top-down breadth-first explanation is easier to understand than a bottom-up one.
  The reader can travel down a path she is interested in until she reaches the component she has to work in, without having to read the entire document or understand the whole system.
- Guideline: Aim for Comprehensibility
- Guideline: Document Minimally, but Sufficiently

Tools

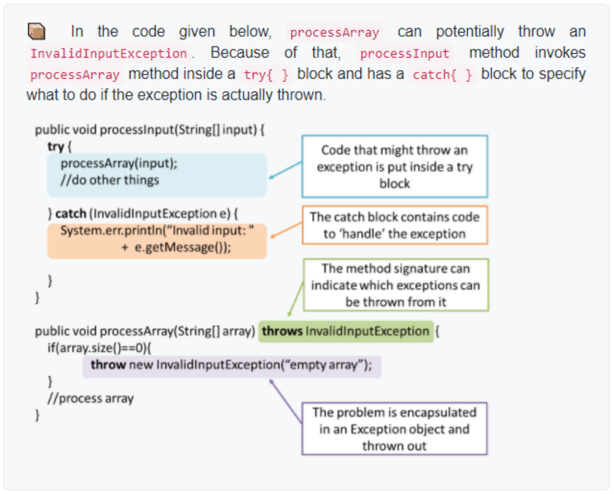
- JavaDoc
  Javadoc is a tool for generating API documentation in HTML format from doc comments in source. In addition, modern IDEs use JavaDoc comments to generate explanatory tool tips.
- Markdown
  Markdown is a lightweight markup language with plain text formatting syntax.
- AsciiDoc
  AsciiDoc is similar to Markdown but has more powerful (but also more complex) syntax.

Error Handling

Exceptions

Exceptions are used to deal with 'unusual' but not entirely unexpected situations that the program might encounter at run time.

Most languages allow a method to encapsulate the unusual situation in an Exception object and 'throw' that object so that another piece of code can 'catch' it and deal with it. Exception objects can propagate up the method call hierarchy until it is dealt with.



In the code given below, `processArray` can potentially throw an `InvalidInputException`. Because of that, `processInput` method invokes `processArray` method inside a `try{ }` block and has a `catch{ }` block to specify what to do if the exception is actually thrown.

```
public void processInput(String[] input) {
    try {
        processArray(input);
        //do other things

    } catch (InvalidInputException e) {
        System.err.println("Invalid input: "
                + e.getMessage());

    }
}

public void processArray(String[] array) throws InvalidInputException {
    if(array.size()==0){
            throw new InvalidInputException("empty array");
    }
    //process array
}
```

Code that might throw an exception is put inside a try block
The catch block contains code to 'handle' the exception
The method signature can indicate which exceptions can be thrown from it
The problem is encapsulated in an Exception object and thrown out

Advantages of exception handling in this way:

- The ability to propagate error information through the call stack.
- The separation of code that deals with 'unusual' situations from the code that does the 'usual' work.

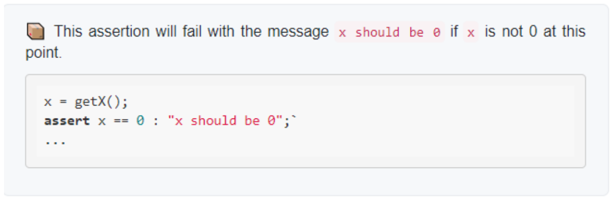
Assertions

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 because the code has resulted in a program state that violates an assumption about how the code should behave.

If the runtime detects an assertion failure, it typically take some drastic action such as terminating the execution with an error message. This is because an assertion failure indicates a possible bug and the sooner the execution stops, the safer it is.

How

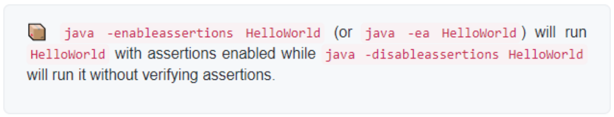
Use the assert keyword to define assertions.



This assertion will fail with the message `x should be 0` if `x` is not 0 at this point.

```
x = getX();
assert x == 0 : "x should be 0";`
...
```

Assertions can be disabled without modifying the code.



`java -enableassertions HelloWorld` (or `java -ea HelloWorld`) will run `HelloWorld` with assertions enabled while `java -disableassertions HelloWorld` will run it without verifying assertions.

Java disables assertions by default. This could create a situation where you think all assertions are being verified as true while in fact they are not being verified at all. Therefore, remember to enable assertions when you run the program if you want them to be in effect.

When

It is recommended that assertions be used liberally in the code. Their impact on performance is considered low and worth the additional safety they provide.

Do not use assertions to do work because assertions can be disabled. If not, your program will stop working when assertions are not enabled. Assertions are suitable for verifying assumptions about Internal Invariants, Control-Flow Invariants, Preconditions, Postconditions, and Class Invariants. Exceptions and assertions are two complementary ways of handling errors in software but they serve different purposes. Therefore, both assertions and exceptions should be used in code.

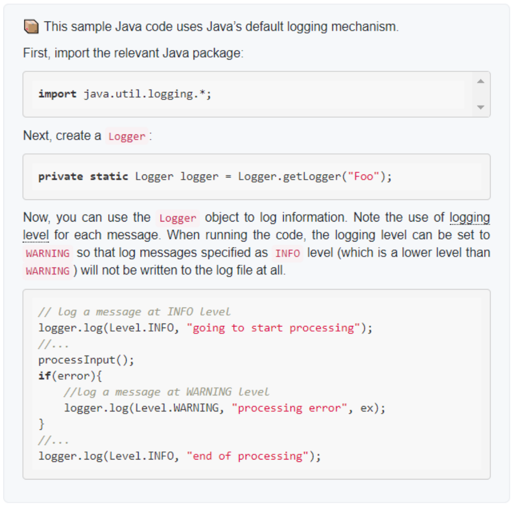
- The raising of an exception indicates an unusual condition created by the user (e.g. user inputs an unacceptable input) or the environment (e.g., a file needed for the program is missing).
- An assertion failure indicates the programmer made a mistake in the code (e.g., a null value is returned from a method that is not supposed to return null under any circumstances).

Logging

Logging is the deliberate recording of certain information during a program execution for future reference. Logging can be useful for troubleshooting problems.

How

Most programming environments come with logging systems that allow sophisticated forms of logging. They have features such as the ability to enable and disable logging easily or to change the logging intensity.



This sample Java code uses Java's default logging mechanism.

First, import the relevant Java package:

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

Next, create a `Logger`:

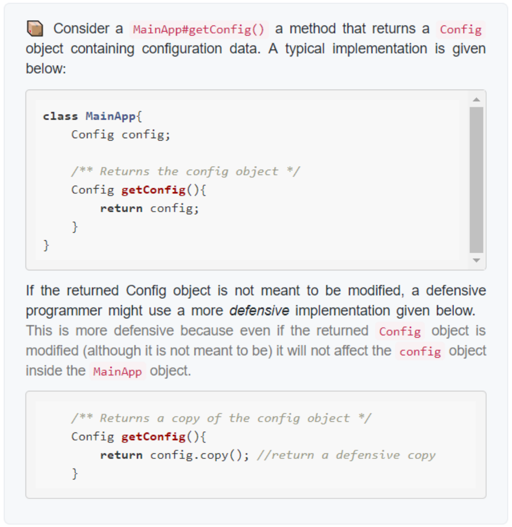
```
private static Logger logger = Logger.getLogger("Foo");
```

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

```
// log a message at INFO level
logger.log(Level.INFO, "going to start processing");
//...
processInput();
if(error){
    //log a message at WARNING level
    logger.log(Level.WARNING, "processing error", ex);
}
//...
logger.log(Level.INFO, "end of processing");
```

Defensive Programming

A defensive programmer codes under the assumption if we leave room for things to go wrong, they will go wrong. Therefore, a defensive programmer proactively tries to eliminate any room for things to go wrong.



Consider a `MainApp#getConfig()` a method that returns a `Config` object containing configuration data. A typical implementation is given below:

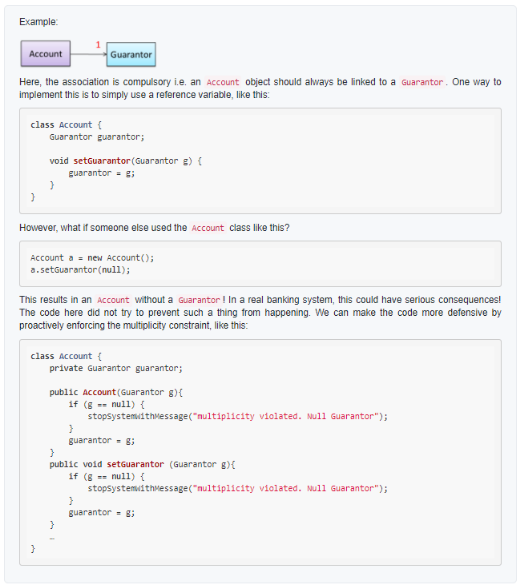
```
class MainApp{
    Config config;

    /** Returns the config object */
    Config getConfig(){
        return config;
    }
}
```

If the returned Config object is not meant to be modified, a defensive programmer might use a more *defensive* implementation given below. This is more defensive because even if the returned `Config` object is modified (although it is not meant to be) it will not affect the `config` object inside the `MainApp` object.

```
    /** Returns a copy of the config object */
    Config getConfig(){
        return config.copy(); //return a defensive copy
    }
```

Enforcing Compulsory Associations

Consider two classes, Account and Guarantor, with an association as shown in the following diagram:



Example:

Here, the association is compulsory i.e. an `Account` object should always be linked to a `Guarantor`. One way to implement this is to simply use a reference variable, like this:

```
class Account {
    Guarantor guarantor;

    void setGuarantor(Guarantor g) {
        guarantor = g;
    }
}
```

However, what if someone else used the `Account` class like this?

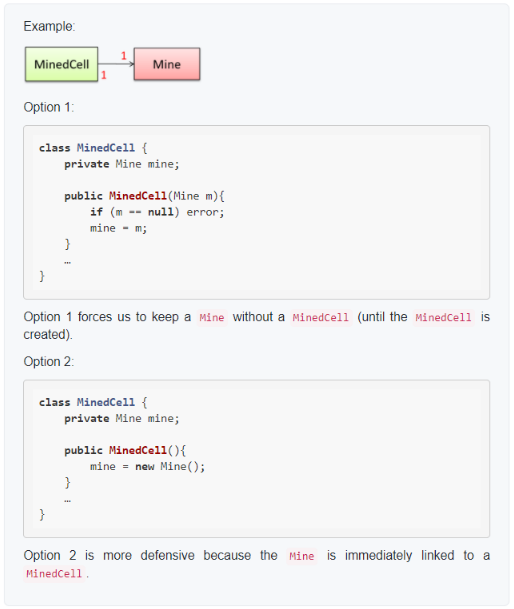
```
Account a = new Account();
a.setGuarantor(null);
```

This results in an `Account` without a `Guarantor`! In a real banking system, this could have serious consequences! The code here did not try to prevent such a thing from happening. We can make the code more defensive by proactively enforcing the multiplicity constraint, like this:

```
class Account {
    private Guarantor guarantor;

    public Account(Guarantor g){
        if (g == null) {
            stopSystemWithMessage("multiplicity violated. Null Guarantor");
        }
        guarantor = g;
    }
    public void setGuarantor (Guarantor g){
        if (g == null) {
            stopSystemWithMessage("multiplicity violated. Null Guarantor");
        }
        guarantor = g;
    }
    …
}
```

Enforcing 1-1 Associations

Consider the association given below. Here, a MinedCell cannot exist without a Mine and vice versa. The only way to enforce this is by simultaneous object creation. However, in Java and C++, only one object can be created at a time. Given below are two alternatives. Both options violate the multiplicity for a short period of time.



Example:

Option 1:

```
class MinedCell {
    private Mine mine;

    public MinedCell(Mine m){
        if (m == null) error;
        mine = m;
    }
    …
}
```

Option 1 forces us to keep a `Mine` without a `MinedCell` (until the `MinedCell` is created).

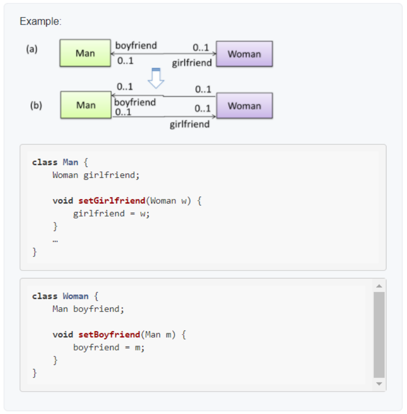
Option 2:

```
class MinedCell {
    private Mine mine;

    public MinedCell(){
        mine = new Mine();
    }
    …
}
```

Option 2 is more defensive because the `Mine` is immediately linked to a `MinedCell`.

Enforcing Referential Integrity

A bidirectional association in the design (shown in (a)) is usually emulated at code level using two variables (as shown in (b)).

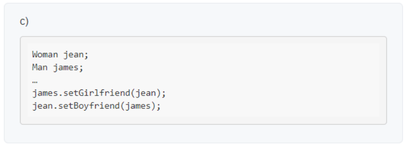


Example:

```
class Man {
    Woman girlfriend;

    void setGirlfriend(Woman w) {
        girlfriend = w;
    }
    …
}
```

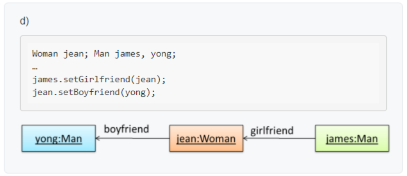
```
class Woman {
    Man boyfriend;

    void setBoyfriend(Man m) {
        boyfriend = m;
    }
}
```

The two classes are meant to be used as shown in (c) below. Now see what happens if the two classes were used as in (d) below. Now James' girlfriend is Jean, while Jean's boyfriend is not James. This situation results as the code was not defensive enough to stop this love triangle. In such a situation, we say that the referential integrity has been violated. It simply means there is an inconsistency in object references.



c)

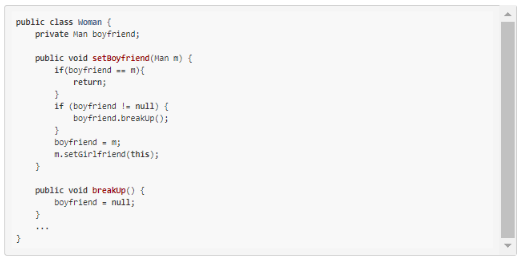
```
Woman jean;
Man james;
…
james.setGirlfriend(jean);
jean.setBoyfriend(james);
```



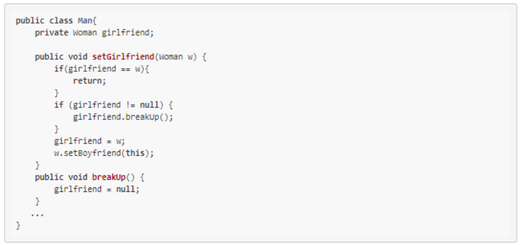
d)

```
Woman jean; Man james, yong;
…
james.setGirlfriend(jean);
jean.setBoyfriend(yong);
```

One way to prevent this situation is to implement the two classes as shown below. Note how the referential integrity is maintained.



```java
public class Woman {
    private Man boyfriend;

    public void setBoyfriend(Man m) {
        if(boyfriend == m){
            return;
        }
        if (boyfriend != null) {
            boyfriend.breakUp();
        }
        boyfriend = m;
        m.setGirlfriend(this);
    }

    public void breakUp() {
        boyfriend = null;
    }
    ...
}
```



```java
public class Man{
    private Woman girlfriend;

    public void setGirlfriend(Woman w) {
        if(girlfriend == w){
            return;
        }
        if (girlfriend != null) {
            girlfriend.breakUp();
        }
        girlfriend = w;
        w.setBoyfriend(this);
    }
    public void breakUp() {
        girlfriend = null;
    }
   ...
}
```

When the code james.setGirlfriend(jean) is executed, the code ensures that james break up with any current girlfriend before he accepts jean as the girlfriend. Furthermore, the code ensures that jean breaks up with any existing boyfriends and accepts james as the boyfriend.

When

It is not necessary to be 100% defensive all the time. While defensive code may be less prone to be misused or abused, such code can also be more complicated and slower to run. The degree of defensiveness depends on many factors such as:

- How critical is the reliability of the system?
- Will the code be used by programmers other than the author?
- The level of programming language support for defensive programming
- The overhead of being defensive

Design-by-Contract Approach

Suppose an operation is implemented with the behavior specified precisely in the API (preconditions, post conditions, exceptions etc.). When following the defensive approach, the code should first check if the preconditions have been met. Typically, exceptions are thrown if preconditions are violated. In contrast, the Design-by-Contract (DbC) approach to coding assumes that it is the responsibility of the caller to ensure all preconditions are met. The operation will honor the contract only if the preconditions have been met. If any of them have not been met, the behavior of the operation is unspecified.

Languages such as Eiffel have native support for DbC. For example, preconditions of an operation can be specified in Eiffel and the language runtime will check precondition violations without the need to do it explicitly in the code. To follow the DbC approach in languages such as Java and C++ where there is no built-in DbC support, assertions can be used to confirm pre-conditions.

Integration

What

Combining parts of a software product to form a whole is called integration. It is also one of the most troublesome tasks and it rarely goes smoothly.

Approaches

- Late and one-time
  Wait till all components are completed and integrate all finished components near the end of the project.
- Early and frequent
  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 few components at a time. This approach is better than the big-bang integration because it surfaces integration problems in a more manageable way.
- Top-down integration
  Higher-level components are integrated before bringing in the lower-level components. One advantage of this approach is that higher-level problems can be discovered early. One disadvantage is that this requires the use of stubs in place of lower level components until the real lower-level components are integrated to the system. Otherwise, higher-level components cannot function as they depend on lower level ones.
- Bottom-up integration: the reverse of top-down integration
  Note that when integrating lower level components, drivers may be needed to test the integrated components because the UI may not be integrated yet, just like top-down integration needs stubs.
- Sandwich integration: a mix of the top-down and the bottom-up approaches
  The idea is to do both top-down and bottom-up so as to 'meet' in the middle.

Build Automation

Build automation tools automate the steps of the build process, usually by means of build scripts.
n a non-trivial project, building a product from source code can be a complex multi-step process. For example, it can include steps such as to pull code from the revision control system, compile, link, run automated tests, automatically update release documents (e.g. build number), package into a distributable, push to repo, deploy to a server, delete temporary files created during building/testing, email developers of the new build, and so on. Furthermore, this build process can be done on demand, it can be scheduled (e.g. every day at midnight) or it can be triggered by various events (e.g. triggered by a code push to the revision control system).

Some of these build steps such as to compile, link and package are already automated in most modern IDEs. For example, several steps happen automatically when the build button of the IDE is clicked. Some IDEs even allow customization to this build process to some extent.

However, most big projects use specialized build tools to automate complex build processes. Popular build tools: Gradle, Maven, Apache ANT, GNU Make

Some build tools also serve as dependency management tools. Modern software projects often depend on third party libraries that evolve constantly. That means developers need to download the correct version of the required libraries and update them regularly. Therefore, dependency management is an important part of build automation. Dependency Management tools can automate that aspect of a project. (Maven, Gradle)

Continuous Integration and Continuous Deployment

An extreme application of build automation is called continuous integration (CI) in which integration, building, and testing happens automatically after each code change.
A natural extension of CI is Continuous Deployment (CD) where the changes are not only integrated continuously, but also deployed to end-users at the same time.
Tools: Travis, Jenkins, Appveyor, CircleCI

Reuse

Reusability is a major theme in software engineering practices. 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 come in many forms; it can be reusing a piece of code, a subsystem, or a whole software.

When

While you may be tempted to use many libraries/frameworks/platform that seem to crop up on a regular basis and promise to bring great benefits, note that there are costs associated with reuse. Here are some:

- The reused code may be an overkill (think using a sledgehammer to crack a nut) increasing the size of, or/and degrading the performance of, your software.
  The reused software may not be mature/stable enough to be used in an important product. That means the software can change drastically and rapidly, possibly in ways that break your software.

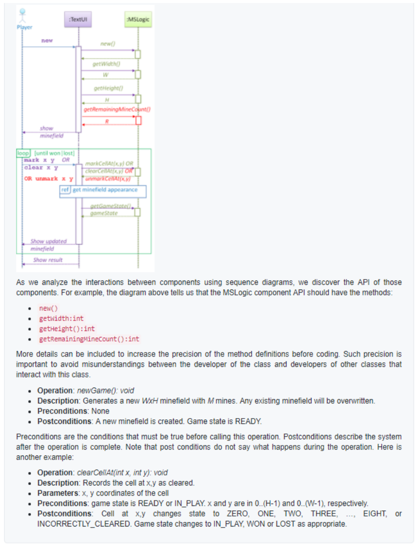
- Non-mature software has the risk of dying off as fast as they emerged, leaving you with a dependency that is no longer maintained.
- The license of the reused software (or its dependencies) restrict how you can use/develop your software.
- The reused software might have bugs, missing features, or security vulnerabilities that are important to your product but not so important to the maintainers of that software, which means those flaws will not get fixed as fast as you need them to. Malicious code can sneak into your product via compromised dependencies.

APIs

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. When developing large systems, if you define the API of each components early, the development team can develop the components in parallel because the future behavior of the other components are now more predictable.

Designing APIs

An API should be well-designed (i.e. should cater for the needs of its users) and well-documented.
When we write software consisting of multiple components, we need to define the API of each component.
One approach is to let the API emerge and evolve over time as we write code.
Another approach is to define the API up-front. Doing so allows us to develop the components in parallel.
We can use UML sequence diagrams to analyze the required interactions between components in order to discover the required API. Given below is an example.



As we analyze the interactions between components using sequence diagrams, we discover the API of those components. For example, the diagram above tells us that the MSLogic component API should have the methods:

- `new()`
- `getWidth:int`
- `getHeight():int`
- `getRemainingMineCount():int`

More details can be included to increase the precision of the method definitions before coding. Such precision is important to avoid misunderstandings between the developer of the class and developers of other classes that interact with this class.

- **Operation**: *newGame(): void*
- **Description**: Generates a new *WxH* minefield with *M* mines. Any existing minefield will be overwritten.
- **Preconditions**: None
- **Postconditions**: A new minefield is created. Game state is READY.

Preconditions are the conditions that must be true before calling this operation. Postconditions describe the system after the operation is complete. Note that post conditions do not say what happens during the operation. Here is another example:

- **Operation**: *clearCellAt(int x, int y): void*
- **Description**: Records the cell at x,y as cleared.
- **Parameters**: x, y coordinates of the cell
- **Preconditions**: game state is READY or IN_PLAY. x and y are in 0..(H-1) and 0..(W-1), respectively.
- **Postconditions**: Cell at x,y changes state to ZERO, ONE, TWO, THREE, …, EIGHT, or INCORRECTLY_CLEARED. Game state changes to IN_PLAY, WON or LOST as appropriate.

Libraries

A library is a collection of modular code that is general and can be used by other programs. E.g String, ArrayList, HashMap, Natty (Java Library for parsing date strings)

How

These are the typical steps required to use a library.

1. Read the documentation to confirm that its functionality fits your needs.
2. Check the license to confirm that it allows reuse in the way you plan to reuse it. For example, some libraries might allow non-commercial use only.
3. Download the library and make it accessible to your project. Alternatively, you can configure your dependency management tool to do it for you.
4. Call the library API from your code where you need to use the library functionality.

Frameworks

The overall structure and execution flow of a specific category of software systems can be very similar. The similarity is an opportunity to reuse at a high scale.
A software framework is a reusable implementation of a software (or part thereof) providing generic functionality that can be selectively customized to produce a specific application. Some frameworks provide a complete implementation of a default behavior which makes them immediately usable. A framework facilitates the adaptation and customization of some desired functionality.
Examples: JavaFx, Drupal, Ruby On Rails, JUnit

Frameworks vs Libraries

Although both frameworks and libraries are reuse mechanisms, there are notable differences:

- Libraries are meant to be used as is while frameworks are meant to be customized/extended. e.g., writing plugins for Eclipse so that it can be used as an IDE for different languages (C++, PHP, etc.), adding modules and themes to Drupal, and adding test cases to JUnit.
- Your code calls the library code while the framework code calls your code.Frameworks use a technique called inversion of control, aka the Hollywood principle (i.e. dont call us, well call you!). That is, you write code that will be called by the framework, e.g. writing test methods that will be called by the JUnit framework. In the case of libraries, your code calls libraries.

Platforms

A platform provides a runtime environment for applications. A platform 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.
Two well-known examples of platforms are JavaEE and .NET, both of which sit above Operating systems layer, and are used to develop enterprise applications. Infrastructure services such as connection pooling, load balancing, remote code execution, transaction management, authentication, security, messaging etc. are done similarly in most enterprise applications. Both JavaEE and .NET provide these services to applications in a customizable way without developers having to implement them from scratch every time.

Cloud Computing

Cloud computing is the delivery of computing as a service over the network, rather than a product running on a local machine. This means the actual hardware and software is located at a remote location, typically, at a large server farm, while users access them over the network. Maintenance of the hardware and software is managed by the cloud provider while users typically pay for only the amount of services they use. This model is similar to the consumption of electricity; the power company manages the power plant, while the consumers pay them only for the electricity used. The cloud computing model optimizes hardware and software utilization and reduces the cost to consumers. Furthermore, users can scale up/down their utilization at will without having to upgrade their hardware and software. The traditional non-cloud model of computing is similar to everyone buying their own generators to create electricity for their own use.

Iaas, Paas and SaaS

Cloud computing can deliver computing services at three levels:

- Infrastructure as a service (IaaS) delivers computer infrastructure as a service. For example, a user can deploy virtual servers on the cloud instead of buying physical hardware and installing server software on them. Another example would be a customer using storage space on the cloud for off-site storage of data. Rackspace is an example of an IaaS cloud provider. Amazon Elastic Compute Cloud (Amazon EC2) is another one.
- Platform as a service (PaaS) provides a platform on which developers can build applications. Developers do not have to worry about infrastructure issues such as deploying servers or load balancing as is required when using IaaS. Those aspects are automatically taken care of by the platform. The price to pay is reduced flexibility; applications written on PaaS are limited to facilities provided by the platform. A PaaS example is the Google App Engine where developers can build applications using Java, Python, PHP, or Go whereas Amazon EC2 allows users to deploy application written in any language on their virtual servers.
- Software as a service (SaaS) allows applications to be accessed over the network instead of installing them on a local machine. For example, Google Docs is an SaaS word processing software, while Microsoft Word is a traditional word processing software.