Reuse

Introduction

What

★★☆☆ Can explain software reuse

Reuse 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

★★★☆ Can explain the costs and benefits of reuse

While you may be tempted to use many libraries/frameworks/platforms 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, and/or 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- Non-mature software has the **risk of dying off** as fast as they emerged, leaving you with a dependency that is no longer maintained.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.

Exercises

APIs

What

★★☆☆ Can explain 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.

A class has an API (e.g., API of the Java `String` class, API of the Python `str` class) which is a collection of public methods that you can invoke to make use of the class.

The GitHub API is a collection of web request formats that the GitHub server accepts and their corresponding responses. You can write a program that interacts with GitHub through that API.

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

Exercises

Designing APIs

★★★★ **Can design reasonable quality APIs**

An API should be well-designed (i.e. should cater for the needs of its users) and well-documented.

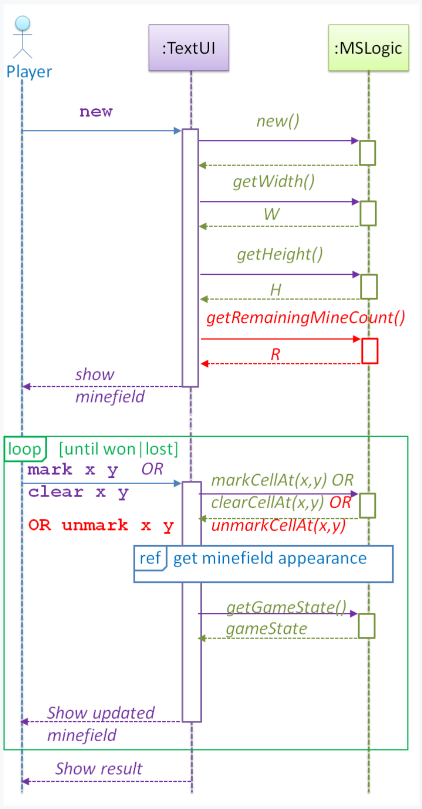
When you write software consisting of multiple components, you need to define the API of each component.

One approach is to let the API emerge and evolve over time as you write code.

Another approach is to define the API up-front. Doing so allows us to develop the components in parallel.

You can use UML *sequence diagrams* to analyze the required interactions between components in order to discover the required API. Given below is an example.

Example:



As you analyze the interactions between components using sequence diagrams, you 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 the 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 postconditions 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

What

★☆☆☆ Can explain libraries

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

Java classes you get with the JDK (such as `String`, `ArrayList`, `HashMap`, etc.) are library classes that are provided in the default Java distribution.

Natty is a Java library that can be used for parsing strings that represent dates e.g. `The 31st of April in the year 2008`

How

★★☆☆ Can make use of a library

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's functionality.

Frameworks

What

★★☆☆ 🏆 Can explain 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.

> Running example:
>
> IDEs for different programming languages are similar in how they support editing code, organizing project files, debugging, etc.

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.

> Running example:
>
> Eclipse is an IDE framework that can be used to create IDEs for different programming languages.

Some frameworks provide a complete implementation of a *default* behavior which makes them immediately usable.

> Running example:
>
> Eclipse is a fully functional Java IDE out-of-the-box.

A framework facilitates the adaptation and customization of some desired functionality.

> Running example:
>
> The Eclipse plugin system can be used to create an IDE for different programming languages while reusing most of the existing IDE features of Eclipse.
>
> E.g. https://marketplace.eclipse.org/content/pydev-python-ide-eclipse

Some frameworks cover only a specific component or an aspect.

> JavaFX is a framework for creating Java GUIs. Tkinter is a GUI framework for Python.

> More examples of frameworks
>
> - Frameworks for web-based applications: Drupal (PHP), Django (Python), Ruby on Rails (Ruby), Spring (Java)
> - Frameworks for testing: JUnit (Java), unittest (Python), Jest (JavaScript)

Frameworks vs Libraries

★★★☆ 🏆 Can differentiate between frameworks and 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. don't call us, we'll 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.

Exercises

Platforms

What

★★★☆ Can explain 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.

Technically, an operating system can be called a platform. For example, Windows PC is a platform for desktop applications while iOS is a platform for mobile applications.

Two well-known examples of platforms are JavaEE and .NET, both of which sit above the 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.

- JavaEE (Java Enterprise Edition) is both a framework and a platform for writing enterprise applications. The runtime used by JavaEE applications is the JVM (Java Virtual Machine) that can run on different Operating Systems.
- .NET is a similar platform and framework. Its runtime is called CLR (Common Language Runtime) and it is usually used on Windows machines.

Cloud computing

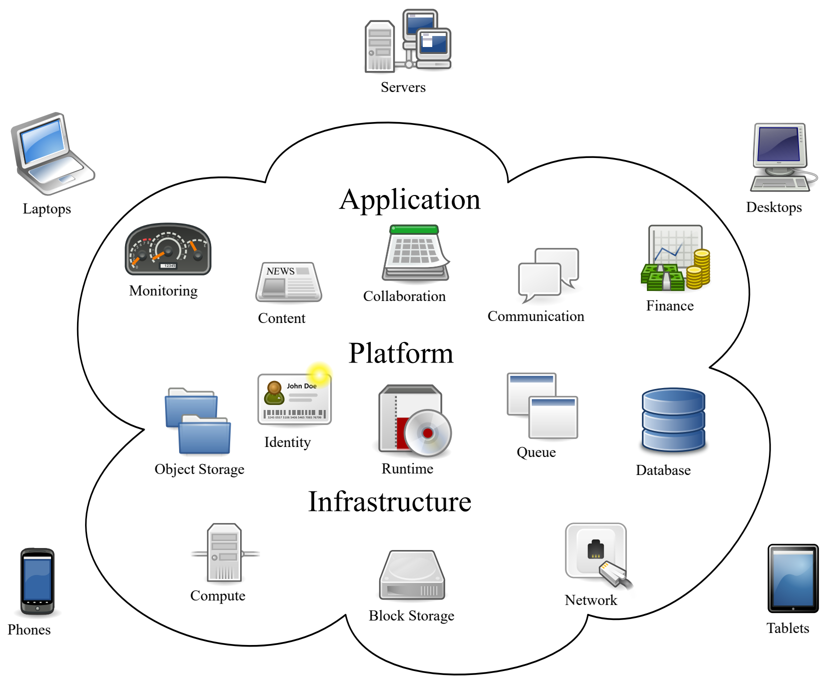
What

★★★★ Can explain 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

★★★★ Can distinguish between IaaS, PaaS, and SaaS



source: https://commons.wikimedia.org

Cloud computing can deliver computing services at three levels:

1. **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.

2. **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 applications written in any language on their virtual servers.

3. **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 a SaaS word processing software, while Microsoft Word is a traditional word processing software.

Exercises