Reuse-based software engineering is a software engineering strategy where the development process is geared to reusing existing software. The software units that are reused may be of radically different sizes. For example:

1. *Application system reuse* The whole of an application system may be reused by incorporating it without changing into other systems

2. *Component reuse* Components of an application, ranging in size from subsystems to single objects, may be reused.

3. *Object and function reuse* Software components that implement a single function, such as a mathematical function, or an object class may be reused.

**Reuse advantages:**

* *Increased dependability* -- Reused software, which has been tried and tested in working systems, should be more dependable than new software.
* *Reduced process risk* -- The cost of existing software is already known, whereas the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation.
* *Effective use of specialists* -- Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
* *Standards compliance* -- Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.
* *Accelerated development* -- Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system productionbecause both development and validation time may be reduced.

**Reuse problems:**

* Increased maintenance costs -- if the source code of a reused software system or component is not available, then maintenance costs may be higher because the reused elements of the system may become increasingly incompatible with system changes.
* Lack of tool support -- some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system.
* Not-invented-here syndrome -- some software engineers prefer to rewrite components because they believe they can improve on them.
* Creating, maintaining, and using a component library -- populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
* Finding, understanding, and adapting reusable components -- software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include

**Key factors that you should know before start planning reuse:**

*The development schedule for the software* If the software has to be developed quickly, you should try to reuse off-the-shelf systems rather than individual components.

*The expected software lifetime* If you are developing a long-lifetime system, you should focus on the maintainability of the system. You should not just think about the immediate benefits of reuse but also of the long-term implications.

*The criticality of the software and its non-functional requirements* For a critical system that has to be certified by an external regulator

*The platform on which the system will run* Some components models, such as .NET, are specific to Microsoft platforms.

**Application frameworks**

Application frameworks -- Collections of abstract and concrete classes are adapted and extended to create application systems. Frameworks provide support for generic features that are likely to be used in all applications of a similar type. Frameworks support design reuse in that they provide a skeleton architecture for the application as well as the reuse of specific classes in the system. The architecture is defined by the object classes and their interactions. Classes are reused directly and may be extended using features such as inheritance.

Frameworks are implemented as a collection of concrete and abstract object classes in an object-oriented programming language. Therefore, frameworks are language-specific. There are frameworks available in all of the commonly used object-oriented programming languages (e.g., Java, C#, C++, as well as dynamic languages such as Ruby and Python).

Three classes of frameworks:

*System infrastructure frameworks*

*Middleware integration frameworks*

*Enterprise application frameworks*

Web application frameworks (WAFs) are a more recent and very important type of framework. WAFs that support the construction of dynamic websites are now widely available.

Web application frameworks usually incorporate one or more specialized frameworks that support specific application features. Although each framework includes slightly different functionality, most web application frameworks support the following features:

1. *Security* WAFs may include classes to help implement user authentication (login) and access control to ensure that users can only access permitted functionality in the system.

2. *Dynamic web pages* Classes are provided to help you define web page templates and to populate these dynamically with specific data from the system database.

3. *Database support* Frameworks don’t usually include a database but rather assume that a separate database, such as MySQL, will be used. The framework may provide classes that provide an abstract interface to different databases.

4. *Session management* Classes to create and manage sessions (a number of interactions with the system by a user) are usually part of a WAF.

5. *User interaction* Most web frameworks now provide AJAX support, which allows more interactive web pages to be created.

**Software product lines**

One of the most effective approaches to reuse is to create software product lines or application families. A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect different requirements. The core system is designed to be configured and adapted to suit the needs of different system customers. This may involve the configuration of some components, implementing additional components, and modifying some of the components to reflect new requirements. Software product lines usually emerge from existing applications.

Application frameworks and software product lines obviously have much in common. They both support a common architecture and components, and require new development to create a specific version of a system. The main differences between these approaches are as follows:

1. Application frameworks rely on object-oriented features such as inheritance and polymorphism to implement extensions to the framework. Generally, the framework code is not modified and the possible modifications are limited to whatever is allowed by the framework. Software product lines are not necessarily created using an object-oriented approach. Application components are changed, deleted, or rewritten. There are no limits, in principle at least, to the changes that can be made.

2. Application frameworks are primarily focused on providing technical rather than domain-specific support. For example, there are application frameworks to create web-based applications. A software product line usually embeds detailed domain and platform information. For example, there could be a software product line concerned with web-based applications for health record management.

3. Software product lines are often control applications for equipment. For example, there may be a software product line for a family of printers. This means that the product line has to provide support for hardware interfacing. Application frameworks are usually software-oriented and they rarely provide support for hardware interfacing.

4. Software product lines are made up of a family of related applications, owned by the same organization. When you create a new application, your starting point is often the closest member of the application family, not the generic core application.

**COTS product reuse**

A commercial-off-the-shelf (COTS) product is a software system that can be adapted to the needs of different customers without changing the source code of the system. Virtually all desktop software and a wide variety of server products are COTS software. Because this software is designed for general use, it usually includes many features and functions. It therefore has the potential to be reused in different environments and as part of different applications.

**COTS Benefits:**

1. As with other types of reuse, more rapid deployment of a reliable system may be

possible.

2. It is possible to see what functionality is provided by the applications and so it is

easier to judge whether or not they are likely to be suitable. Other companies

may already use the applications so experience of the systems is available.

3. Some development risks are avoided by using existing software. However, this

approach has its own risks, as I discuss below.

4. Businesses can focus on their core activity without having to devote a lot of

resources to IT systems development.

5. As operating platforms evolve, technology updates may be simplified as these

are the responsibility of the COTS product vendor rather than the customer.

**COTS Problem:**

1. Requirements usually have to be adapted to reflect the functionality and mode

of operation of the COTS product. This can lead to disruptive changes to existing

business processes.

2. The COTS product may be based on assumptions that are practically impossible

to change. The customer must therefore adapt their business to reflect these

assumptions.

3. Choosing the right COTS system for an enterprise can be a difficult process,

especially as many COTS products are not well documented. Making the wrong

choice could be disastrous as it may be impossible to make the new system work

as required.

4. There may be a lack of local expertise to support systems development.

Consequently, the customer has to rely on the vendor and external consultants

for development advice. This advice may be biased and geared to selling products

and services, rather than meeting the real needs of the customer.

5. The COTS product vendor controls system support and evolution. They may go

out of business, be taken over, or may make changes that cause difficulties for

customers.

There are two types of COTS product reuse, namely COTS-solution systems and

COTS-integrated systems. COTS-solution systems consist of a generic application

from a single vendor that is configured to customer requirements. COTS-integrated

systems involve integrating two or more COTS systems (perhaps from different vendors) to create an application system.