## Deployment, Maintenance, and Support

Software products are used (by the user) in what is often called the **production environment**, which is the hardware and software systems that the user interacts with through the software product. The process of making the product available in the production environment is known as **deployment**. After the software product has been deployed, the developer (or an agent of the developer) will need to help the customer use it. This kind of interaction is known as **support**. Finally, since faults may be discovered, the production environment may change, and requirements may change, the software product will need to be modified after it is deployed. This process is known as **maintenance**.

All of these activities are important aspects of software engineering that warrant some discussion. However, to understand them it is first necessary to understand physical architectures.

### Physical Architectures

A **physical architecture** is the realization of a software product as artifacts (such as files) residing on and executing on computational resources. A product's physical architecture is distinct from its **logical architecture** (the software's major components and the relationships between them) which we discussed before.

Physical architectures can vary in many ways. However, software engineers often categorize them based on three characteristics—where the software is stored or installed, where the software is executed, and where the user's data operated on by the software (including configuration information) are stored. Since each of these characteristics can take one of two possible values (user device or shared device), these three characteristics lead to eight different categories of production environments, the following four of which are archetypical.

*Personal*—The software is stored (installed) on a user device (such as a personal computer, phone, or tablet) and executed on that device. User data is also stored on that device. (If the device has multiple users, each user's data is associated with that user.)

*Shared*—The software is stored on a shared device (such as a shared disk), temporarily delivered to the user device, and executed on the user device. User data is stored on the user device.

*Mainframe*—The software product is stored on a shared device and executed on that shared device but is accessible from a user device (known as a terminal). User data is stored on the shared device.

*Cloud*—The software product is stored on a shared device, temporarily delivered to the user device, and executed on the user device. User data is stored on the shared device.

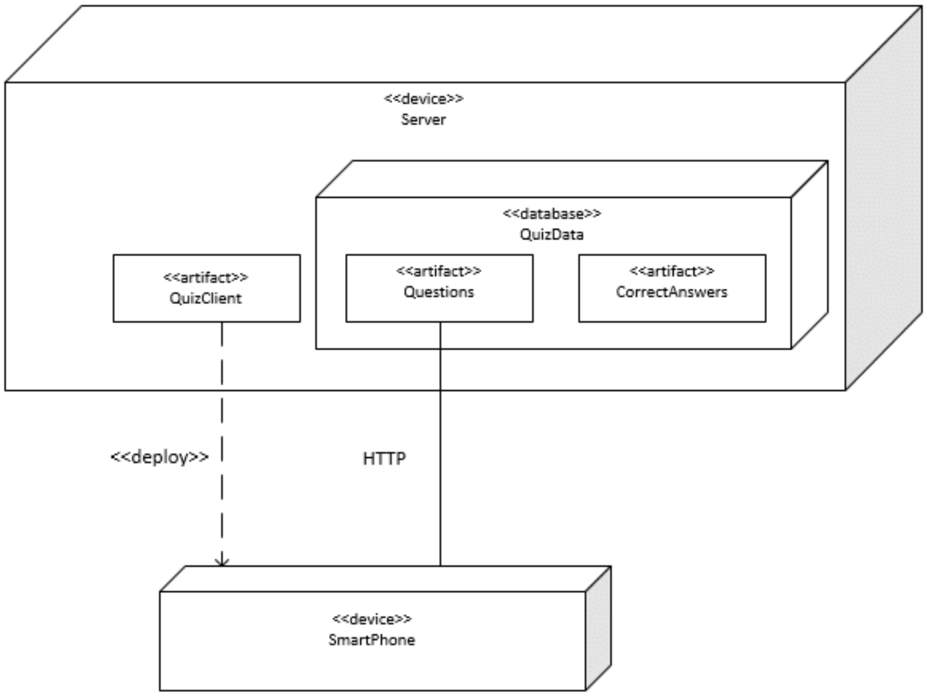
Two of the other four possible categories involve the software being stored on the user device and executed on the shared device. Such software is sometimes called a *mobile agent*. The other two involve the software being stored and executed on one device but the user data stored on the other. While such environments exist, they are rare.

Physical architectures can be modeled using UML deployment diagrams, which contain artifacts and nodes. An *artifact* is a physical manifestation of a component of a software system. In other words, an artifact is the physical manifestation of data that is created or used during software development or operation. Examples include files, documents, source code, diagrams, audio clips, and video clips. A *node* is either a physical device or an execution environment (such as a virtual machine or an operating system), both of which are computational resources.

Artifacts are represented as rectangles (with either the stereotype «artifact» or an iconic representation of the type of the artifact) and nodes are represented as boxes (with either the stereotype «device», «execution environment», or a description of the computational resource). Both artifacts and nodes can either be types or instances; instances are distinguished from types by underlining the identifier. The identifier for a node instance is of the form *name:type*, but either *name* or *:type* may be omitted.

Communication between different nodes is represented using a solid line called a *communication path*. The deployment relationship (between artifacts and nodes) can be represented in three different ways—an artifact can be placed inside of a node, a node can contain a list of artifact names, or the artifact and node can be connected using a dependency arrow with the stereotype «deploy».

The following UML deployment diagram illustrates a mobile quiz system. The QuizClient, written in JavaScript, resides permanently on the Server. When the user wants to take a quiz, it is temporarily deployed to the SmartPhone. The QuizClient then makes an HTTP request to GET the Questions. The CorrectAnswers never leave the Server. Both the Questions and the CorrectAnswers are managed by a QuizData database.



## Deployment

As mentioned above, deployment is the process of making the software product available in the production environment. Deployment, in its idealized form, involves the following steps.

*Release*—The artifacts are assembled into a distributable package.

*Install*—The distributable package is brought to the production environment, the artifacts are disassembled, and the artifacts are moved to the appropriate nodes.

*Activate*—All of the executable artifacts are started.

When the product is no longer needed it is deactivated (all of the executable components are stopped) and uninstalled (the parts, resources, or artifacts are removed from the production environment).

#### *Atomicity and Rollback*

The installation and activation steps, by their very nature, change the state of the production environment. To minimize the disruption that is caused by this change of state, it is very important that the installation and activation steps appear to be **atomic**. That is, though they will likely involve several actions that occur over time, they must appear to be a single activity. Also, it is very important to be able to *roll back* (undo) these activities, and return the production environment to its previous state.

#### *Distributable Packages*

There are a variety of different ways to create distributable packages, and they have changed considerably over time. In the early days of the software industry, the distributable package contained everything needed to build or make the product and the product had to be installed by someone with considerable technical knowledge and skill. Later, distributable packages became *installers*, executable products whose only job was to install another product. More recently, software is distributed through *stores* (e.g., the Apple Store, Play Store or Windows Store) or using a *package manager* (e.g., dpkg, fink, or RPM). There are also efforts underway to use *virtual machines* (e.g., VMWare, Virtual Box) and *containers* (e.g., Docker) to distribute software.

In any case, it is important to be able to guarantee the integrity of distributable packages; in particular, clients have confidence that a product deployed on their hardware or responsible for storing their data is trustworthy. This is often accomplished through the use of digital signatures. In some cases, distributable packages also make use of *copy protection* or *digital rights management* to protect the intellectual property in the distributable package. Obviously, this latter concern does not arise with free and open source software (FOSS).

## Maintenance

**Maintenance** is any change to a software product after it has been deployed to a production environment. ISO/IEC 14764 identifies four different kinds of maintenance.

**Corrective Maintenance**—Modifications that correct faults after they give rise to failures (in the production environment).

**Preventive Maintenance**—Modifications that correct faults (discovered in the development environment after the product has been deployed) before they give rise to failures (in the production environment), or to correct defects of other kinds to improve maintainability, portability, and so forth.

**Adaptive Maintenance**—Modifications that keep the product usable in a changed or changing environment (e.g., migration to another platform).

**Perfective Maintenance**—Modifications that satisfy additional functional requirements (e.g., adding new features) or non-functional requirements (e.g., improving performance).

Regardless of the kind of modification, it takes some time to decide on a modification and carry it out. Hence, it is common to collect data both about the number of modifications requested and completed, and the mean or median amount of time required to complete a modification.

Though, by definition, maintenance only occurs after the product is deployed, the timing and frequency of maintenance activities will vary with the software process used. In general, agile methods deploy more frequently than traditional methods. Hence, maintenance activities tend to occur more frequently. Indeed, some people might argue that, in an agile process, maintenance activities are indistinguishable from design and construction activities. While this might be true for corrective, preventive and perfective maintenance, adaptive maintenance activities seem to be easily identified even in an agile process. Though maintenance activities are less frequent in traditional processes, they are no less significant. In fact, many empirical studies have argued that development accounts for 20% of the total effort devoted to a software product and maintenance accounts for the remaining 80%.[1]

Maintenance is expensive for several reasons. The main reason is that it goes on for so long: a successful product may be in the field for decades with new releases occurring every year or so. Maintenance activities are bound to accrue huge costs over such a long period. But there are other reasons why maintenance is costly, including the following.

- Much software is poorly written to begin with. This makes it expensive to maintain, and the difficulty only grows as the software product gains features over time (as most do). Often products become so impenetrable that they must be completely rewritten.
- As a software system is modified over time, its structure deteriorates. This occurs at all levels of the system, including the architecture, low-level design, and the code itself.
- Maintenance done using traditional processes has the same problems as original development: the new release does not appear for a long time and only then do fundamental problems become obvious. Also, stakeholder needs and desires may have changed during the maintenance activity. Thus efforts to create new releases can fail just as development projects can, and this costs a lot of money.

Solutions to these problems are mostly obvious. Software should be written from the start to be as maintainable as possible, a point we have stressed throughout. This means making and documenting a good design, implementing it with well written and well-documented code, creating thorough tests that are maintained along with the software, and so forth. The problem of code deterioration is addressed by carefully modifying product architectures and designs over time, then implementing these changes in code that is written with the same care as original code, not just by patching here and there. Refactoring to improve code structure is an especially important tool in this regard. Finally, an organization can switch to agile processes for maintenance even though a product was developed using a traditional process, obviating many of the problems of traditional processes.

While maintenance activities are intended to improve a software product, they can interfere with the use of that product. In many cases, maintenance activities may make a software product completely unavailable to users for an extended period of time (for example, the maintenance of an online banking system). Hence, maintenance activities involve significant organizational tradeoffs and compromises.[2] In addition, as with any problem-solving situation, there is always more than one way to make a modification. At the extremes, they are often characterized as the “elegant solution” and the “quick and dirty solution,” the latter usually being the one that will have the smallest short-term impact on users. Hence, technical compromises must often be made.

[1] A key question that often arises when considering maintenance is the replace versus maintain decision. In other words, should an existing system be maintained or should it be retired and replaced with a new system? Such questions are often answered by calculating the life-cycle cost. Such issues are discussed in the chapter on Financial and Economic Planning.

[2] In some cases, both the software architecture and the physical architecture must be designed with these tradeoffs in mind.

Maintenance activities may be performed by the original development team (which, of course, is likely to evolve over time) or by a distinct maintenance team. Obviously, while the development team will have an excellent understanding of the product, they may not understand the operational environment (e.g., business requirements, user needs) and, because of their technical orientation, may not understand or appreciate the tradeoffs. On the other hand, though a maintenance team will understand the tradeoffs, they may have limited understanding of the product and, hence, may face a steep learning curve. Also, maintenance teams may have, or perceive themselves as having, lower status than development teams. Hence, morale problems may arise and maintainer may have a higher turnover rate than developers.

Maintenance processes generally recapitulate development processes. As noted above, in agile processes maintenance requests are simply folded into the queue of pending work (e.g., the product backlog in Scrum), and in traditional processes a project is mounted to create a new release. The maintenance project is planned, runs through the traditional life cycle phases, and delivers a result just like a from-scratch development project. The only difference is one of starting point: all work products are modifications or elaborations of existing work products.

## Support

The goal of the product design phase is to specify software product features, capabilities and interfaces that satisfy the needs and desires of stakeholders. However, no product design is perfect and, as discussed in the section on requirements, the needs and desires of different stakeholders may conflict. Hence, users will almost always require assistance with the product after it is deployed. Interactions between the user and the developer (or an agent of the developer) for the purpose of improving the users' experience with the product are referred to as support activities.

At the two extremes, support activities can be characterized as either professional or community.

**Professional Support**—The person interacting with the user is employed, either directly or indirectly[3], by the developer or provider of the software product.

**Community Support**—The person interacting with the user is another user or an expert who is not employed by the developer or product provider.

Of course, hybrids (a community system with a small number of professionals) are also both possible and fairly common.

Though professional support teams are employed by the developer or provider of the software product, they are almost always completely distinct from both the development and maintenance teams. There are several reasons for this. First, members of the support team tend to have less technical knowledge and fewer technical skills, and as a result, they are paid significantly less than members of the development or maintenance teams. Second, the members of the development team (and to a lesser extent, the members of the maintenance team) may not have the interpersonal skills necessary to deal with frustrated users. Third, the members of the support team are likely to have greater knowledge of the application domain.[4]

In practice, support teams tend to be segmented. Individual members of the team tend to have or develop specific areas of both technical and domain expertise. In some support organizations this

[3] Support activities are sometimes **outsourced**, that is, performed by a company that is employed by the developer/provider of the software product. Such a company is sometimes referred to as a **third party** since it is neither the developer/provider nor the user/client.

[4] In some situations, a member of the development team may be brought in to help with particular technical issues.

segmentation is formalized while in others it is more informal. For example, the initial point of contact may try to resolve the issue using a knowledge base of frequently asked questions (FAQ) and answers. Then, if that isn't possible, the issue may be referred to a member of the support team with more in-depth or specific knowledge.

Support activities obviously involve *communication*, that is, the transmission or exchange of information, knowledge, or ideas, by means of speech, writing, or electronic media. A *communication channel* is a particular means, either physical or logical, of transmitting or exchanging information. Some channels are synchronous and others are asynchronous. In a *synchronous* channel both parties (e.g., the user and the member of the support team) participate in the conversation at the same time and any transmission that is not immediately processed by the recipient is lost. In an *asynchronous* channel, the parties participate in the conversation over time (i.e., each transmission is stored by the recipient and processed at a later time).[5] Support activities can and frequently do use any or all of the following channels, all of which are electronic.

*Telephone*—A synchronous channel used to transmit voice (or voice and video).

*Chat/Messaging*—A synchronous channel used to transmit text (or text and pictures).

*Electronic Mail*—An asynchronous channel used to transmit text (or text and pictures).

Some support systems also use variants of the last two like social networking sites and WWW forms.

The channel used can have a significant impact on both the cost of providing support and the users' assessment of the quality of the support—both cost and quality tend to go down as you move down the list. Because of this, attempts have been made to reduce the cost of telephone support, including telephone trees (i.e., menu systems), voice recognition, and voice FAQs.

Support teams commonly use an **issue tracking system** (sometimes called a **call management system** or **ticketing system**) to accept, prioritize, assign, track and report on requests for support. Such systems are not dissimilar to the bug tracking systems discussed in the chapter on quality assurance in construction.

Support can either be provided for free or there can be a fee. Fee-based support activities tend to fall into one of several categories. Some organizations provide support for free. Other organizations charge per contact or per unit of calendar time (e.g., per year). Still others charge based on the amount of time required to resolve the issue, some charging per issue while others charging for blocks of time (either in advance or as you go). Organizations that charge a fee for support often provide different levels of support to different users. Such systems are said to be *tiered*. Tiers can be defined based on the hours that support is available (e.g., M-F, 9:00-5:00 vs. 24/7), the time to engage and respond (e.g., same-day vs. within five minutes), or the channel used.

Though it seems self-evident, it is worth explicitly noting that support systems should be created and put into place before the software product is deployed. Many software products have failed because of insufficient support. Less obvious, though equally important, it is necessary to estimate the cost of providing support and incorporate it into the pricing of the software product.

## References

1. Lehman, M.M. (1980) "Programs, Life Cycles, and the Laws of Software Evolution", *Proceedings of the IEEE*, 68, pp. 1060-1076.

[5] Note that with both a synchronous and an asynchronous channel, the *protocol* that is used may involve turn-taking.