Joomla! Development Strategy

Introduction

From its very core, Joomla is designed and built to help bring people together. It is centered on simplicity and ease of use. For these reasons we have a certain way of thinking as we approach Joomla development.

Objectives

- To continue to offer a stable and reliable platform for our current and future user base.
- To make innovation available to users and developers on a more timely basis.
- To make it easy for developers to contribute code to the project at any time.

There are five major principles to the Joomla development strategy aimed at achieving those objectives: maintain a stable trunk; predictable, incremental software releases; strong backward compatibility support; a sound security policy; and an open development process.

Maintain a Stable Trunk

It is critical to our mission that the trunk must always contain a stable and tested version of the code base that can be made ready for a release within a short time. To ensure that the trunk is always stable write access is only granted to a limited number of disciplined and trusted people.

Trunk represents the current development build, or the most current functioning version of the software. When features or issues are being “worked on” that should happen in a branch or in a separate system altogether until completion. Changes should only be committed to the trunk when they are complete, working, and all automated tests pass.

Requirements for inclusion into trunk.

There are a few requirements that must be met before changes are considered for inclusion into the trunk. These measures help to ensure that the trunk is always in a stable state.

Satisfy the Joomla! Coding Standards.

Coding standards exist to keep code consistent, easily readable, and maintainable. Before any code contributions are included into the trunk they should meet the Joomla Coding Standards which are described in detail at the documentation wiki page: http://docs.joomla.org/Coding_style_and_standards.

Pass all automated tests.

Automated tests are used to isolate specific parts or features of the software and ensure that they behave correctly. The Joomla project maintains Unit Tests for API classes and methods as well as System Tests for various application functionality. Before code contributions are included into the trunk all the automated tests should pass so that we can be reasonably assured that the trunk remains stable and reliable.

Provide automated tests for all new API classes and methods.

Any new methods or classes added to the Joomla platform API must be accompanied by working Unit Tests to verify their correct behavior. This serves to help guarantee that not only the current code works, but that future changes do not cause an unstable trunk. Where appropriate system tests should be created to verify expected behavior.

Provide basic documentation for all new additions to the code base.

To maintain features and systems added to the code base, as well as to help ensure an ability to uphold our principle of backward compatibility some basic documentation is required for new API classes and methods or new application logic when it is added to the trunk. If you are writing a new class or method, describe what it is supposed to be used for and why it exists and a couple of brief examples of its use. For application features, describe the addition or describe each layout and their intended purpose.

Predictable, Incremental Software Releases

A software release is the distribution of software code, documentation, and support materials. The Joomla project strategy for software releases borrows heavily from the Ubuntu project's time-based release process, which likewise borrowed heavily from the GNOME project.

Version Strategy

The version identifiers for Joomla follow a three level numerical convention where the levels are defined by change significance. Depending upon the significance of the change between a release and the previous release it will be designated as one of those three levels: major, minor, or maintenance.

```
Major.Minor[.Maintenance]
```

If the release is designated as major the first number in the version identifier will be incremented and the remaining numbers set to zero such as 1.0.0 or 2.0.0. If the release is a minor release the second number will be incremented and the last one set to zero such as 1.5.0 and 1.6.0. For maintenance releases the last number is incremented as in 1.6.1 and 1.6.2.

Major Release

For a release to be classified as major it will include a high degree of change compared to the previous release. Generally this will involve massive architectural and/or user interface changes. Substantial changes to the underlying data model can also cause a release to be identified as major.

Minor Release

To be categorized as a minor release a high degree of continuity should be present both architecturally as well as in data model between the release and its predecessor while still providing new or improved functionality.

Maintenance Release

A maintenance release will include fixes to bugs, security vulnerabilities, and usability issues only. New functionality is not introduced unless specifically addressing a problem with the previous release that must be handled before the next minor release.

Version Assignment

Once a release is feature complete and on the road to stabilisation, an assessment will take place based on the changes from the previous release to determine its classification as major, minor, or maintenance.

Release Life Cycle

The software release life cycle is composed of distinct phases and milestones that express the software's maturity as it advances from planning to development to release and support. Not every release will go through every possible phase. As an example, most maintenance releases would omit all phases up to the release candidate milestone because the changes from the previous release would be minimal.

Software Milestones

There are four major software milestones in the Joomla Release Life Cycle: alpha, beta, release candidate, and general availability.

Alpha

The alpha milestone signifies that there is new technology in the software that is ready for testing. Software packages marked as alpha are not feature complete and are not suitable for production environments.

Beta

Once a release has reached the beta milestone it is considered feature complete. Like software packages marked alpha, those which are marked as beta are not considered suitable for production environments. They are intended to be tested thoroughly for backward compatibility issues as well as security and stability problems.

Note: New features are generally not introduced after the beta milestone for a given release unless the features correct faulty implementations or missing behavior found in backward compatibility testing.

Release Candidate

When a release is complete and has been thoroughly tested can be declared as having reached the release candidate (RC) milestone. Packages marked as release candidates are considered complete and suitable for production environments. They have potential to be a product, ready for general availability release unless critical problems emerge.

General Availability

The general availability (GA) milestone indicates that the release is very stable and appropriate for mass distribution and use by end users.

Release Phases

There are three distinct phases in the Joomla software life cycle. For the purposes of looking at them, we begin at the point where a a release has just hit the general availability milestone as described above.

Maintenance

For six weeks after a release reaches the general availability milestone, it is still the primary focus of the production working group. During this maintenance phase the Joomla Bug Squad and other teams' primary responsibility is triaging and fixing any emerging issues with the current stable release. Only stability changes after thorough testing for the new release are accepted for inclusion into the trunk. Maintenance releases may be issued during this phase as necessary. At the end of the maintenance phase the current release will only be updated in the case of found security vulnerabilities until its stated end of life date.

Feature Merge

The transition from the maintenance phase to the feature merge phase marks a shift in the focus of development efforts from the current stable release to the next release. For twelve weeks after the completion of the maintenance phase new features and enhancements can be merged into the trunk from branches and patches. At this time the trunk no longer represents the current stable release but the next release.

During this time the release will reach the alpha milestone and one or more alpha packages may be made available to preview new features or technology that has been merged into the trunk.

Note: Development work in branches can be (and is encouraged to be) ongoing at any time in the development life cycle and is not confined to the feature merge phase. This is just the phase of the life cycle where such changes can be merged into the trunk for the next release.

Release Testing

The beginning of the release testing phase is marked by the reaching of the beta milestone. During this phase, which should last around eight weeks, the production working group teams' focus shifts to comprehensive testing of the release. New packages are made available every two weeks for testing by end users throughout the release testing phase. Also during this phase online help documentation and translation strings are finalised. It is important that all extension developers test their software during this phase for backward compatibility issues so they can be resolved before the general availability milestone.

The general availability milestone marks the end of the release testing phase. Final packages for the release are built and distributed as long with any announcements and/or press releases. After this phase the cycle restarts with the maintenance phase.

Support Lifetime

Every Joomla software release has a lifetime in which it is supported by the Joomla project. There are two different classifications of release concerning the support lifetime: standard and long term. Regardless of the support lifetime only security updates and fixes for critical breakages will be issued for a release once it has left the maintenance phase of the life cycle.

Standard Support

Most releases are standard releases which are only officially supported for approximately six months which coincides with the release cycle as described above. A standard support release reaches its end of life one month after the general availability milestone of the next major or minor release.

Long Term Support

Every third major or minor release will be classified as a long term support release (LTS). These releases are officially supported until three months after the general availability milestone of the next long term support release. The longer support schedule is aimed at making transitions for users who seek a longer, more stable release cycle where eighteen months before a major software update makes considerably more sense than six months.

End of Life

Once a release has reached the end of its support lifetime (its "end of life" or EOL) it will no longer be maintained by the Joomla project teams for either stability or security issues.

Example Support Scenario

Date	Current Version	Event Description
September 2010	1.6	1.6 GA released with long term support.
March 2011	1.7	1.7 GA released with standard support.
September 2011	1.8	1.8 GA released with standard support
October 2011	1.8	1.7 reaches end of life.
March 2012	1.9	1.9 GA released with long term support.
April 2012	1.9	1.8 reaches end of life.
June 2012	1.9	1.6 reaches end of life.

In the above example version 1.6 is available as a long term support release, thus meaning that users can choose to upgrade it with 1.7 when it is available six months later or with 1.9 (the next LTS release) when it is available eighteen months later. A direct upgrade path will be provided for consecutive standard support releases and for consecutive long term support releases. Any other combination will not be officially supported.

Open Development Process

Our development process is designed to be open and accessible for anyone who wishes to participate. We strive to create an environment where people work together to solve problems and bring innovative, fresh ideas to life in the software we produce.

Aligning with a Vision

Every release of Joomla is different and needs to balance sometimes convergent, sometimes divergent needs and wants from a variety of stakeholders. By far the most difficult task is to marry people who want things done with volunteers who want to do it. This is further complicated by people who want to do things that not many people want or need. So how do we solve this indeterminate problem?

Our solution is to plan a theme or a vision for each minor release as a catalyst to align those people with good ideas and those with the ability to implement them. The vision is discussed at a developer summit held before or very soon after a new minor release reaches the GA (General Availability) milestone. This summit looks at ideas from a variety of sources (Joomla Leadership Team, individual contributors, the Joomla Idea Pool to name a few) and attempts to distill an achievable vision for the next release based on the interest and effort available within the community at the time.

Just because a vision is set, does not prevent people from making contributions outside of that vision. The vision, rather, is designed to help guide the general direction of the software for the next release and set a theme for the contributor community to rally around.

Communication

The most critical aspect of collaboration is communication. To that end, there are several different mechanisms to aid in providing an open, collaborative development process such as mailing lists, IRC channels, issue trackers, and forums.

Most often the best work is done in small teams, and we encourage people to create ad-hoc teams to work on features and fixes for Joomla. It is important, however, for these small teams to regularly communicate with the larger development community regarding progress and direction to build support for their work.

Bug Squad

The most important standing team within the Joomla development effort is the bug squad. The Joomla Bug Squad is tasked with managing the release testing and maintenance phases of the development cycle. Because of this the bug squad is largely responsible for quality assurance for the Joomla project. Maintaining a low barrier to entry for new contributors is a key principle of the bug squad, and it is a great place for less experienced contributors to get familiar with the code base.

Idea Pool

A Request for Comments (RFC) with respect to the Joomla development process is a document submitted for peer review which describes functionality, ideas, or other information regarding a change in the Joomla software platform. The Joomla Idea Pool is where developers and users can work together to draft RFC documents and turn ideas into reality.

Good ideas are everywhere.

Innovation starts with a good idea. The idea pool is a tool for people to be able share their ideas for improving Joomla, be them simple or complex. They could involve highly technical changes to the code, or they could describe a way in which a user interfaces with the software, or anything in between.

Collaboration with peers turns good ideas into great ideas.

Good ideas need the right input to produce great ideas, and that input comes from discussing the idea with your peers. It is the responsibility of everyone who favors the idea to help mould it into a complete thought. For an RFC to be seriously considered, it must answer at least one of the following two questions:

1. How can the idea be implemented on a technical level?
2. How does the idea make the Joomla platform behave on a functional level?

An RFC does not necessarily have to be a fully coded solution, though, working proofs-of-concept are most certainly welcome. One way an RFC can reach maturity is for it to describe how an idea can be implemented, usually in "code" speak. Not everyone is a developer and so another way ideas can reach maturity is to describe how a change will behave such that a team of developers could take a functional specification and build it.

Turning great ideas into reality.

Once an RFC has sufficient specification on the change to be made, whether how it is to be coded, or how it is intended to behave, a decision will made by the Joomla Production Leadership Team as to whether it is marked as a priority for inclusion in the current development release. The priority for the change will take into account its popularity and the synergy with the current development vision.

Submitting Code

For the most part, code submitted to the Joomla project will be written against the current trunk at the time of submission. There are two paths by which a code submission can be approved and added to the trunk:

1. Patches containing tested fixes from the Joomla Bug Squad.
2. Approved merges from feature branches.

Anyone who signs the Joomla Contributor Agreement (JCA) is free to create a branch, write code, and propose that their code be included in the Joomla core. However, the Production Leadership Team (PLT) makes the final decisions as to which code gets included into the core code base.

Will my code get included into the core?

It is important to understand that there is absolutely no guarantee to anyone that a given enhancement will be included in Joomla. Given this, what can you do to improve the chance that your work will be included into the Joomla code base?

The following three items are required (but not necessarily sufficient) to have your code accepted:

1. The code meets the Joomla Coding Standards, as discussed above.
2. It includes appropriate automated tests. If there are framework changes, complete unit tests should be included to test any new or changed functionality. If there are new or changed user interface elements, system tests should be provided to cover the new functionality.
3. Basic documentation is provided for all new functionality.

In addition, the following characteristics will increase the chance of code being accepted:

1. The proposed change aligns with the vision for the release or with a feature requested by an RFC. This is not always necessary, but will increase the chance of a feature being accepted. Some improvements could be important but are not part of the vision or have not been suggested by others. In this case, they can still be considered.
2. The idea for the change was discussed on the mailing list, and there is general support for it from other list members and from the PLT.
3. Others in the community have not already coded a similar enhancement. If two or more groups work on overlapping functionality, that is fine. In the end, the PLT will decide what to include. It could be one or the other, a combination, or neither. However, in many cases, it would be more productive for people to work together in a branch rather than code a different solutions -- unless you believe the different solution will be much better and you can't persuade the group to change approaches.
4. The idea needs to be in core and cannot (or should not) be done as an extension. This is very important to understand. If a feature can be implemented as an extension, then the presumption is that it should be done as an extension. If there are important reasons why it should be in the core code base, then these need to be discussed and agreed upon.
    1. If a feature cannot be coded as an extension because the framework fails to provide the correct event or other API call, then it might make sense to add the required framework changes. Then the new functionality could be coded as an extension.
    2. Writing a great extension is also a good way to eventually get your code into core. For example, if you have an extension that is superior to the same functionality in core, then that could be a great candidate for inclusion in the core.

How can I avoid duplication of effort?

As discussed above, anyone who signs the JCA can write and propose code for Joomla. There is no topdown management of this process. Given this, how can you minimize the chances for duplication of effort?

The first thing is to be able to know what people are currently working on. We envision using the Wiki to maintain a list of the active branches and who is working on which projects. This will provide a convenient way for a new developer to quickly find out what is underway and perhaps to help out on an existing development project. In addition, the development lists can be used to discuss ongoing projects or new proposals.

Remember that this is a decentralized ad-hoc organization. There are no "official" groups charged with writing the "authorized" version of a particular enhancement. Ultimately, the PLT will decide to the best of it's ability based on the merits of each proposed code set.

Also, it is not necessarily a bad thing to have two or more groups working on overlapping projects. In most cases, there is more than one approach to a particular problem, and it is often not clear what the tradeoffs are until part or the whole solution is coded. So having two or more approaches to the problem can lead to a better solution in the end.

Example Scenario of Project Best Practices

Here is an example scenario of how you might proceed in a way to maximize the chance of having your code get included. First, do some checking to see if others have discussed the idea already. Is it something that is widely considered a valuable feature? Is it consistent with the vision for the release? Has it been requested in the RC? If the answer to these question is yes, then perhaps propose a specific solution on list and see what people think. If this is promising, set up a branch and invite other interested people to help out. Post an article in the Wiki section so that everyone can see that you are starting to work on this project.

Next, you might code a proof-of-concept in the branch and ask people in the community to try it. At that point, it should be possible for the PLT to give a preliminary indication of how the changes fit into the release and whether it is likely to gain PLT approval. If so, then code the remainder of the changes and any related automated tests and again let the community try it. At this point, the branch would be ready for merging into the next release.

Please note that this is only one scenario. A second scenario would be to code a really great extension that people think should be in core and offer to convert the extension so that it can be included in core. A third scenario -- and a much more risky one -- would be to just code a really great change in a branch and then show people the branch when you are finished.

It should also be stated that most people who contribute code to Joomla or other open-source projects will have some of their proposed code not be included. This is normal and to be expected. It is simply not realistic to expect that 100% of any developer's code will get included. Most FOSS developers enjoy the process of writing code and what they learn in the process can be used in future work regardless of whether a particular code set gets included or not.

Strong Backward Compatibility Support

Backward compatibility for any software platform is a high priority. Clean, maintainable code is also a high priority for any software platform. Sadly, in practice, these two ideals are often at odds. Providing backward compatibility support makes software more complex and less maintainable. The Joomla project's policy for addressing these conflicting ideals is as follows:

Backward compatibility support is important.

During development, alpha testing, and beta testing of new releases, backward-incompatible changes will be considered issues to be resolved. Every effort will be made to make sure that each release of Joomla is backward compatible with the previous minor release.

Backward compatibility support will be limited.

When a documented API method is removed or changed, the previously documented method will be retained and deprecated. Deprecated methods will have added code to generate deprecation notices when they are used. The deprecation notices will state specifically when the backward-compatibility support for the method will expire. This expiration will be expressed as a release number which could be as early as the next minor release. In this way old API methods will not have to be maintained longer than necessary, but plenty of warning will be given before their removal.

When data changes are involved, we will provide data conversion tools that will be available before the GA release.

It is often very hard to determine if any applications are relying on or extending a subtle feature. Because of this it is difficult to assess whether backward-compatibility support needs to be provided for that feature and judgement calls will be made. In these cases if the judgement is wrong it will need to be caught during testing and reported.

Backward compatibility support is the responsibility of everyone.

It is very important to catch backward compatibility problems during development of new releases. In particular, it is important to test new Joomla releases before they are released in final form. If a backward-compatibility problem is found before the final release, it will be considered an issue and will generally be fixed (by adding backward compatibility support where it is reasonable to do so) if possible before the final release.

After the final release, we may elect not to fix outstanding or new backward-compatibility problems. Consumers of Joomla have a right to backward compatibility, but they also have a responsibility to test Joomla against their applications during the beta release cycle (or sooner) to make sure their applications work.

Read commentary and answers to frequently asked questions about the backward compatibility policy in Appendix A »

Sound Security Policy

The team of developers and security experts tasked with implementing the Joomla Project Security Policy is the Joomla Security Strike Team (JSST). The JSST is tasked with investigating and responding to reported core vulnerabilities, executing core code reviews before software releases, providing a public presence regarding security issues, and helping the public to better understand Joomla security issues.

Security Announcement Policy

Verified vulnerabilities will only be publicly announced **after** a release is issued which fixes the vulnerability. All announcements will contain as much information as possible, but will **not** contain step-by-step instructions for the vulnerability. As a member of oCERT, the Joomla project follows the policy. We strive to release a new version of Joomla within 14 days of a vulnerability in a currently maintained version being reported to us.

Public Response Policy

Articles are written about Joomla all the time. In many circumstances, these articles (even from reputable sources) contain a significant amount of misinformation. The JSST will assess and address articles written about security issues. If the article contains valid information about a vulnerability not yet fixed, they will ask the publisher to suspend the article until the issue can be resolved. If the article contains invalid information, the JSST will note what is invalid and ask the publisher to either correct or remove the article.

The JSST will be available to answer questions and/or validate any Joomla security related articles at the publisher's request.

Security Release Policy

- Critical and high-level vulnerabilities trigger an immediate release cycle.
- Other vulnerabilities will trigger a release within the 14 day embargo period allowed for by oCERT.
- All security releases will be accompanied by one (or more) appropriate security announcements.
- A release containing a fix for a critical or high-level security issue will be called a *Critical Security Release*.
- A release containing a fix for a security issue will be called a *Security Release*.

Vulnerability Threat Levels

There are two main details that contribute to a vulnerability's priority or "threat level": impact and severity. The following tables provide generic guidelines for how vulnerabilities may be assessed. In practice each vulnerability will be assessed for damage potential and ranked accordingly.

Impact

Critical	“0-day" attacks, and attacks where site control is compromised (allows attacker to take control over site).
High	SQL injection attacks, remote file include attacks, and other attack vectors where site data is compromised.
Moderate	XSS attacks, write ACL violations (editing or creating of content where not allowed).
Low	Read ACL violations (reading of content where not allowed).

Severity

Critical	VERY easy to perform. Relies on no outside information (TRUE 0-day attack).
High	Moderately easy to perform. May rely on readily available outside information.
Moderate	Not easy to perform. May rely on sensitive information.
Low	Difficult to perform. Relies on sensitive information or requires special circumstances to perform.

Appendix A

Backward Compatibility Commentary

Developer API

The developer API is the code that a developer uses to write extensions for Joomla. Where reasonable, new minor releases will be backward compatibility with at least the previous minor release. For example, API not marked as deprecated in version 1.5 will be backward compatible with version 1.6. Unavoidable exceptions can arise (hopefully infrequently). For example, the permission codes in 1.6 are not backward compatible with 1.5 but the goal is for such circumstances to be rare.

Compatibility between major versions, for example 1.x and 2.x, is desirable but not mandatory nor expected.

Data Model

Changes to the data model are inevitable in every release. When data model changes occur, they will be documented and an SQL patch will be provided. Where complex transformations are required then code based solutions (for example, upgrade code in the installation or a separate component) will be provided.

User Interface and Site Behaviour

Changes in the UI between minor releases are to be expected. However, proposed changes should attempt to improve the usage of Joomla, not just change for change's sake. Judgment needs to be used to ensure the "jump" to learn enhancements balances with the net benefit that such enhancements bring. In other words, if a big change is made, make it worthwhile to the end user.

Site Migration

Our goal is to provide a mechanism for SQL upgrades to automatically happen and not subject the end-user to running SQL upgrade queries manually.

Core Output

Core output should not be subjected to dramatic change without significant warning. For example, massive changes will have occurred in the core output of Joomla 1.6 compared with Joomla 1.5 and this has been communicated constantly throughout the development of this release.

Extension Migration

While Joomla core extensions are obviously handled by the Joomla project, third-party migration is the responsibility of the author concerned. Many tools are available since Joomla 1.6 which the developer can take advantage of to assist with this process. We encourage the normal process is that a user will upgrade their site, then install upgrades for all the extensions which will make appropriate adjustments to extension specific code and data.

Backward Compatibility Frequently Asked Questions

I have a site running on version x.y. How difficult will it be for me to migrate to the next Joomla version?

For maintenance releases (for example, from 1.6.3 to 1.6.4), the upgrade should be very easy, fully automatic, and 100% backward compatible.

For minor releases (for example, from 1.6 to 1.7), there will be normally be some conversion required. An automatic data conversion will be provided for the Joomla core features. Third-party extensions may or may not require a data conversion, depending on whether the core changes required changes to the specific extension.

Major releases (for example, from 1.9 to 2.0) are not conceptually different from minor releases. However, they are expected to have a greater number of significant changes. Accordingly, they will be more likely to require data conversions for both the core database and for third-party extensions. It is likely that extensions will need to be modified to work with a new major release. This means that you will need to make sure all the extensions in your site have versions that are compatible with the new major release.

I have an extension that runs with version x.y. How difficult will it be for me to release a version of that extension for the next Joomla version?

For maintenance releases, there should be no changes required. For dot releases, it will depend on how the specific changes in the dot release interact with your extension. It is expected that most dot releases will not require changes to most extensions.

I have an application that uses the framework for Joomla version x.y. How difficult will it be for me to release a new version of that application that uses the next Joomla version?

As mentioned in the Developer API section above, the API will change over time. Developers will get warning when an API is deprecated. When this happens, it will be the application developer's responsibility to change their code to use the supported API before the next release, when the deprecated API will be removed.

When changes are made to the API, functionality will be preserved and documentation will be provided to show how to use the new API to cover the same functionality as the deprecated API.

I have a template that works with Joomla version x.y. How difficult will it be for me to release a version of that template that works with the next Joomla version?

As discussed above under Core Output, we will try to keep the output stable over time and to provide warning when something is going to change. This means that templates should not require changes just because a new version of Joomla is released.

I am in a large organization that wants to use Joomla for an enterprise site. How can I be sure that the product will be stable and that version upgrades will not cause problems?

In a large site, you will be doing some combination of the activities discussed above: namely, site administration, using third-party extensions (or internally developed extensions), using the framework, and using or building templates. Because large, complex sites take longer to test and deploy, frequent version upgrades can be a disadvantage.

For this reason, we will designate some releases to receive long-term security upgrades. For example, let us assume that version 1.6 will be a long-term security release. This will mean that version 1.6 will continue to receive all relevant security fixes for an extended time. So large enterprise sites that want to run the same Joomla version for an extended time should build their sites using one of these special versions. In this way, the frequency of upgrades can be reduced.