

# YAM- A Framework for Rapid Software Development

Abhinandan Jain, Jeffrey Biesiadecki  
 Jet Propulsion Laboratory  
 California Institute of Technology  
 4800 Oak Grove Drive, Pasadena, CA 91109

**Abstract**—YAM is a software development framework with tools for facilitating the rapid development and integration of software in a concurrent software development environment. YAM provides solutions for thorny development challenges associated with software reuse, managing multiple software configurations, the development of software product-lines, multiple platform development and build management. YAM uses release-early, release-often development cycles to allow developers to incrementally integrate their changes into the system on a continual basis. YAM facilitates the creation and merging of branches to support the isolated development of immature software to avoid impacting the stability of the development effort. YAM uses **modules and packages** to organize and share software across multiple software products. It uses the concepts of **link and work modules** to reduce sandbox setup times even when the code-base is large. One side-benefit is the enforcement of a strong **module-level encapsulation** of a **module's** functionality and interface. This increases design transparency, system stability as well as software reuse. YAM is in use by several mid-size software development teams including ones developing mission-critical software.

## I. INTRODUCTION

Managing change is a key requirement for successful software development efforts. Changes may be driven by new requirements, design evolution, refactoring needs, bug fixes etc. While change may be the one constant during development, it can also create instabilities in the development process and the software. An acceptable development pace is one where the software development activity remains stable, i.e. the team is productive and the quality of the software remains high. The quality of the software development infrastructure and processes used to provide structure for a development effort also determine the development pace. There is a fine line between having too little structure, leading to development instability, and too much structure, creating excessive overhead and constraints on the development effort.

As illustrated in Figure 1, the scope of software development efforts can range from a single developer developing a single software product to one involving distributed teams developing a software product-line. Key software development challenges include coordination of changes across the team, version control, handling large code-bases, integration and test, multiple target development, software reuse etc. Yard sticks for measuring the success of a software development activity are the quality of the software itself, and the development pace.

Brooks' *The Mythical Man-Month* [1] was one of the early influential essays that analyzed the software development process. Brooks argues that the dynamics of software development

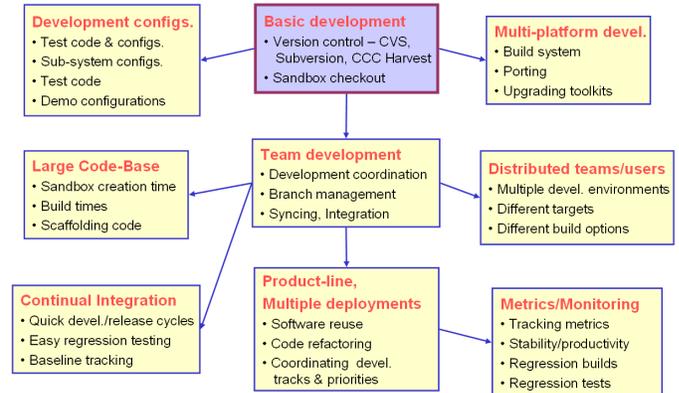


Fig. 1. Levels of software development environments ranging from individual developer/single product to distributed team with product-lines..

is such that contrary to conventional wisdom, adding more developers to a team late in a project can delay rather than speed up the completion of the project. He highlights the need for getting the design right - asserting that the project should be willing to throw away the first implementation. Eric Raymond's *Cathedral and the Bazaar* and other essays [2] studies the open-source software development model. Eric lists several insights to explain the success of the "bazaar" style open-source development in contrast with the "cathedral" style based on the principles of command and discipline. There have also been a series of software development processes referred to as *agile software development* [3] which advocate short iteration cycles. One of the best known examples of agile development is the *Extreme Programming* style of software development [4] which advocates that all development be done by pairs of developers, and asserts the importance of testing to the extent that test cases are written first before software implementation can begin.

This paper describes the YAM software development framework for software development. The YAM development framework YAM addresses several of the challenges facing software development efforts depicted in Figure 1. YAM integrates elements from areas such as configuration management, concurrent development, build management, software organization, and software reuse. YAM provides solutions to key development challenges associated with: continual software integration and test, concurrent development across distributed teams, software development for software product-lines, multi-

ple platform development, handling large code-base etc. **YAM** is a lightweight tool written in Perl [5]. As illustrated in Figure 2, **YAM** builds upon other open source tools such as CVS

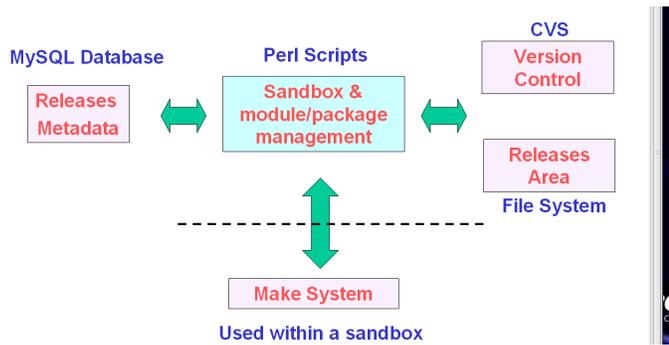


Fig. 2. The key components of **YAM- CVS** for version control, MySQL for a database backend and Perl for scripts.

for version control [6], a MySQL database for releases data [7], and PHP for Web interface [8]. **YAM** provides a set of user commands, in the style of CVS, that are implemented in Perl to support day-to-day development. Additional **YAM** commands are also available for administration support. While **YAM** advocates continual development and integration cycles in the spirit of agile development - it views reusability as an essential ingredient of a good software design as well as the development process itself.

#### A. Software Development Needs

The **YAM** software development paradigm is guided by a “hierarchy of needs” (illustrated in Figure 3) that drive software development efforts. This hierarchy is in the spirit of the hierarchy of needs developed by Maslow [9] in the arena of human psychology<sup>1</sup> to explain what drives human behavior. Maslow posited that human beings are incapable of addressing a higher level need until the lower level ones have been satisfied. This holds true for software development as well. Starting from the lowest level, the needs are:

- **Basic Development Needs:** This is the most basic level representing the minimal needs for a functional development environment. These needs encompass development tools (eg. compilers, debuggers, third party software etc.) to support software development; a software version-control and configuration management system; processes to rollback software; build systems; regression testing setup; support for multi-platform development. These needs apply to even individual software development.
- **Stable Concurrent Development Needs:** This class of needs is for development efforts involving more than one developer. Concurrent development requires processes to manage, coordinate and merge changes from parallel development within a team. During the course of development, developers need access to stable versions of the rest of the system while at the same time shielding the rest of the team from their immature, developmental software.

<sup>1</sup>Maslow’s needs range from physiological needs, survival needs, belongingness needs, esteem needs, to actualization needs.

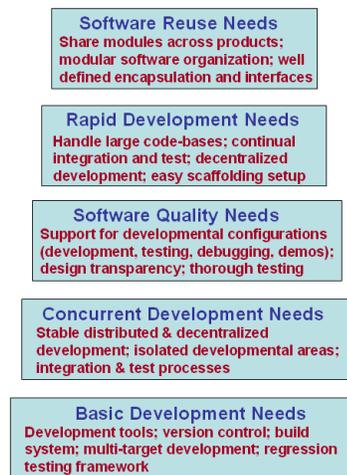


Fig. 3. Hierarchical pyramid of needs that need to be met by software development processes.

Processes for merging and syncing up changes need to be clearly defined. Additional coordination may be needed if the team is distributed across remote sites. Sound processes are needed to avoid instability and confusion during concurrent development.

- **High Quality Software Needs:** Beyond stable development are the needs for developing high quality software - as measured by performance, robustness, maintainability and defect rates. Meeting these needs requires extensive integration and testing of the software at the unit and system levels. Keys to improving the software quality is in depth, multiple-levels of testing of the software and adjusting the design and implementation based on feedback. Testing (as well as development) at the unit, sub-system and system level require the ability to create software configurations tailored for these purposes. The software development process needs to facilitate the easy creation and use of such configurations as a natural part of the development process. It is inevitable that such configurations will be created, and without built-in support, a significant part of the team effort can be wasted in creating brittle, custom environments to meet these needs.
- **Rapid Development Needs:** Having met the needs for a high quality software development, the next level of needs is geared towards increasing the team’s productivity and development pace. Increasing the team development pace requires processes to: reduce integration and test times; reduce build and development times as the system code base gets large; support fast set up of scaffolding software needed for development; decentralize development and reduce coordination overhead; and facilitate prototyping and refinement of design concepts for the software.
- **Reuse Needs:** While the previous needs focused on a “single” software product, it is often the case that the team or the organization is responsible for multiple software products, i.e. a software product-line. For cost-effectiveness, such product-lines can involve significant software sharing and reuse across the different products.

While software reuse is often seen as a software design issue, good design is a necessary, but not sufficient condition for the software to be reusable in practice. Addressing reuse needs requires processes to allow the creation of product specific software configurations from the code base with software shared across different products. Furthermore, the development process needs to be able to accommodate the inter-dependencies among multiple products in so far as they affect the shared software. Changes to such shared software may be driven by internal refactoring needs, availability of new software technology, new platforms, new performance drivers or by requirements that trickle down from other products. The inability to handle such needs can result in incompatible variants of reusable modules ending up in different products. Indeed, it is inevitable that without processes that are reuse-friendly, over time reusable software will cease to be reusable.

The above hierarchy of needs is useful for analyzing the quality of a software development process. A consequence of the needs hierarchy is that attempts to satisfy higher-level needs without having addressed the lower level needs are doomed to failure. For instance, attempting to carry out concurrent software development without adequate regression testing or version control infrastructure is unlikely to get very far. Similarly, it is pointless to attempt to increase development pace without first having a process that produces acceptable, high quality software. Often team development environments struggle to get past the first and second levels. It is not uncommon to see the development pace being throttled back to a slow deliberate pace with extensive coordination overhead to help provide stability to the development process. Software reuse is often little more than an after thought in such an environment.

While providing an analysis framework, the needs hierarchy does not however imply that one should go about designing a software development processes based on these tiers, or that a successful software development model is separable into distinct layers meeting the different tiers of needs. A software development process needs to be viewed in its organic whole - with its internal dynamics determined by the development scope and the team. Indeed, Raymond [2] makes a strong case for how software reuse serves to improve software quality by making the “grass taller, the more it is grazed”. Our experience with the **YAM** development model has been that concepts that might be perceived - and even at times conceived - for higher-level needs, often end up significantly helping meet lower level needs as well. **YAM**'s organization of all software into **YAM modules**, while necessary for reusing software across products, also turns out to be ideal for creating developmental software configurations that are essential during a development effort. **YAM modules** improve software modularity, and help better manage the mix of stable, and new and immature, portions of the code base. **YAM**'s use of link modules for the rapid creation of user sandboxes has the side-effect of enforcing strong module-level encapsulation which enforces software organization with significantly reduced coupling among the software modules. In turn, this helps increase design transparency and software quality. **YAM**'s use of conventions and tools

for the easy creation and merging of module-level branches, simplifies version control as well as concurrent development. The remainder of this paper describes in detail the key ideas and concepts underlying **YAM** and how they address the software development hierarchy of needs.

## II. **YAM** FRAMEWORK CONCEPTS

This section describes some of the key design ideas and concepts that make up the **YAM** software development framework.

- **Low-level version control using CVS**  
The **CVS** open source version control package is used for low-level version control of the software. This includes the use of branches, tagging, handling of directory structures as well as remote client/server use.
- **All software is organized into **YAM** “modules”**  
All software is broken up and organized into software units referred to as **modules**. The size, functionality and number of a **modules** is completely up to the discretion of the development team.
- **Support software is also stored in modules**  
Even support software, e.g. test harnesses and scripts, software stubs, utilities etc. that support the development effort are managed in **modules**. Keeping such support software within version controlled **modules** gives all developers easy access to them, and avoids the risk of “losing” them within developers’ custom setups.
- ****YAM** “packages” are defined as module bundles**  
As illustrated in Figure 4, **YAM packages** are simply

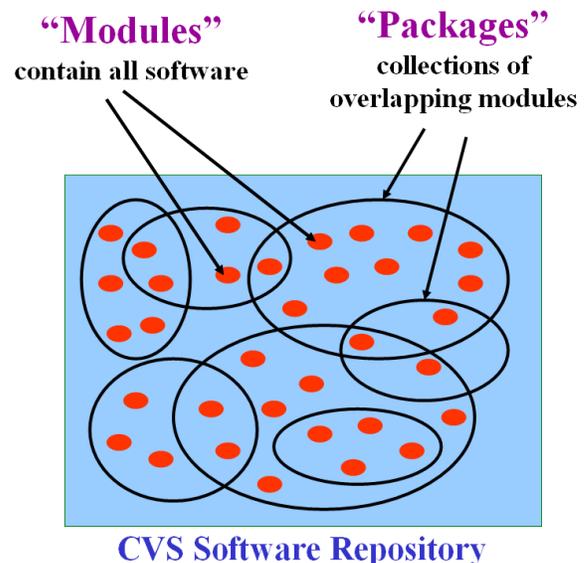


Fig. 4. All software being developed is organized into modules, with packages representing different bundles of modules.

bundles of **YAM modules**. They do not contain any additional software. Thus different packages can share modules between them, and can also contain other packages.

- **All module development takes place on independent CVS branches**  
Developers do all their development on private **CVS**

branches. This isolates them from each others changes during development. The **CVS** main trunk is used as the **release** branch. Thus, on conclusion of a development phase, a developer releases his/her changes by merging the changes on the development onto the main trunk and committing and releasing the relevant modules. **YAM** provides commands to simplify the creation, merging and release of such branches.

- **Development via fast branch/release cycles**

Frequent and regular releases of modules and packages are highly encouraged. Module development branches are expected to be short-lived and to be used to implement specific features or fixes. This is driven by two important goals. Firstly, long life branches can require disproportionately large merge and integration effort on the part of the developer. Secondly, such large integration efforts can also require a large effort from the rest of the team to absorb and handle such large changes after they are released. **YAM**'s typical module branch/release cycles are illustrated in Figure 5.

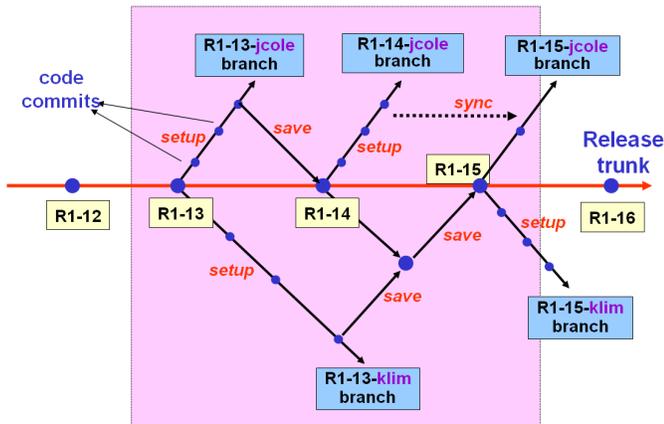


Fig. 5. The software development proceeds through a continual series of branch, develop, merge and integration cycles.

- **Releases are managed at the module and package level**

Developers make releases of individual modules. On release, the source code for the module is tagged with a release number for later use if needed. The built version of the module is stored for later use in a module release area. The typical requirement for a module release is that it pass the test suite. During a module release, **YAM** generates candidate entries for a module's ChangeLog file from CVS log messages from individual file commits. Packages can also be released. Package releases simply consist of a specific grouping of module releases. All release information is recored in a database and a PHP script provides on-line access to the information.

- **Built version of module releases are available as link modules**

On release, the built version of the module is moved from the developer's sandbox into a module release area. The purpose of this module release area is to support a tightly integrated development environment for sub-teams

sharing common file systems. These built versions can be used as **link** modules to quickly create virtual presence for these modules within developer sandboxes to use as scaffolding code.

- **All work is done in independent sandboxes**

All development is carried out within independent user sandboxes containing the modules to be worked on. **YAM** provides commands to simplify the creation of private **YAM** package sandboxes for developers. These commands also create branches for and checkout modules to be worked on within the sandbox. On completion of the development work, **YAM** commands can be used to make releases of the modules. Developers can have as many sandboxes as they need.

- **Developers can tailor a sandbox configurations**

Developers can tailor the modules contained in a sandbox. Moreover, they can specify which of the modules are present as simply **link modules** and which ones are **work modules**, i.e. ones whose source code is checked out into the sandbox. As illustrated in Figure 6, a sandbox configuration can consist of only link modules (eg.

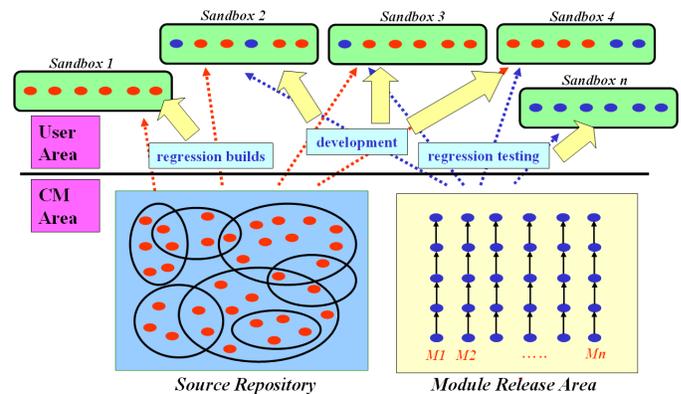


Fig. 6. Sandbox configurations can range from all work modules, to all link modules, to a mix of link and work modules.

for running regression tests) or all work modules (eg. for regression builds) or any mix in between. Usually, a developer will create a sandbox with modules that he/she plans to work on, and the rest of the required modules as link modules to provide scaffolding code to create a complete, working sandbox. It is worth noting that since link modules require just symbolic links they are much faster to create than work modules that need to be checked out and built. The developer can also select the versions of link modules and module release branches to include in the sandbox. **YAM** also provides support for updating a sandbox's configuration to add and remove work and link modules.

- **Explicit exporting of module interfaces**

To meet the requirement that modules be usable as work modules as well as link modules within developer sandboxes, all **YAM** modules export their public interfaces via symbolic links to a common **top-level** area within the sandbox. This allows module A to access files from module B irrespective of whether module B is a link

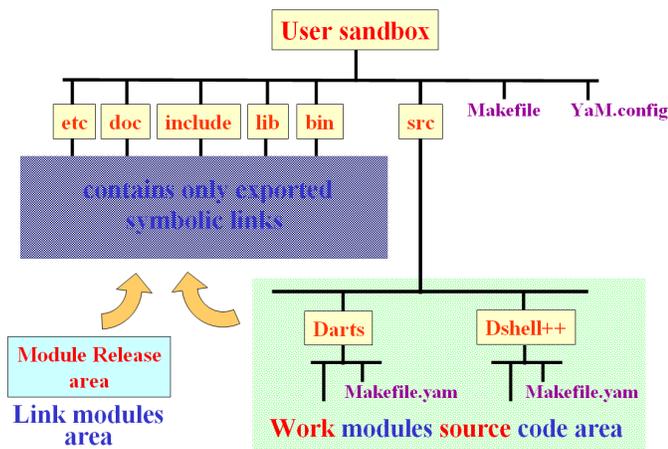


Fig. 7. The structure of a typical YAM sandbox.

or a work module in the sandbox. Thus, if a module builds a library that is needed by other modules, then this module should export links for this module into the top-level area. On the other hand, if the module needs header files from other modules in order to build the library, then it should look for them in the top-level area and not directly in the module directory. The other modules should have exported links for these header files to the top-level area. Thus the top-level area serves as a central nexus for interactions between the different modules. YAM provides make rules to simplify such exporting of links by modules.

- **modules have a standardized build interface**

Modules have a uniform `make` file interface for building the module. These module-level makefiles are required to support a small number of make rules such as for cleaning builds, building the module, exporting links etc. YAM supports the generation of such makefiles from templates, and keeps the module specific customization largely to the definition of variables using other supporting make files provided by a special `SiteDefs` module. Of course, modules can internally have arbitrary levels of additional build infrastructure as long as this minimal interface makefile requirement is satisfied. The module make interface establishes uniform conventions much like `autoconf` has done for open source software.

- **Track third party tool dependency information**

The module makefiles do not contain hard-wired paths for external third party tools but instead derive the information from the `SiteDefs` module. Though not precluded from doing so, YAM modules do not typically use `autoconf` for builds in order to allow explicit control of external software versions (required for using link modules), and for version controlling this external dependencies information. As a result developers can rollback the software - and the external tool configuration - to earlier points in time.

- **Development effort is organized as a YAM Project**

A single YAM installation can be used to support multiple YAM projects. A YAM project denotes a development effort with a CVS repository, its own releases area, its

own module/package definitions and its own MySQL database. YAM allows project-level as well as user-level tailoring of the default behavior of YAM commands to fine-tune the processes for policies defined for the development effort.

- **Support for multi-site development:**

To support distributed teams not sharing a common development environment, and for supporting builds of software for deployment in other environment, YAM supports the definition of `site` specific build information in the `SiteDefs` module. This information is version controlled. Developers are free to specify custom site information to override the information in the `SiteDefs` module. This is convenient for users working in isolated environments such as laptops.

- **Support for multi-target development:**

Often, the development effort needs to support the use and deployment of the developed software on more than one target platform hosts. A target platform here is used to denote variants in computing hardware, different types or flavors of operating systems, or differences in compilers for the target hosts. The key point is that binaries built for different platforms cannot be inter-mixed. YAM provides build conventions and rules to support building modules for multiple platforms within the same directory structure. This is convenient for verifying and testing new capabilities across all the targets of interest in one place. YAM also provides convenient rules for building modules in parallel for the different targets. There is also support for gracefully disabling a module's builds for targets it does not support.

- **Module inter-dependency tracking:**

YAM provides support for automatically extracting information about module inter-dependencies<sup>2</sup>. This information is very useful for generating alerts about modules impacted by API changes within a module, as well as for automatically pulling in modules needed by a module into a sandbox.

- **RPM support:**

A recent YAM capability is the support for creating RPMs [10] for YAM packages and modules for external deployment of the software. An earlier capability for creating customized package tarballs for deployment continues to be supported.

### III. MEETING DEVELOPMENT HIERARCHY NEEDS

In this section we examine how the elements of YAM's software development framework described in Section II address the hierarchy of needs for software development described in Section I-A.

#### A. Meeting Basic Development Needs

YAM provides extensive support for meeting the basic development needs including

- version control support using CVS

<sup>2</sup>Currently implemented for only C/C++ parts of the software.

- module/package level release management, automated ChangeLog entry generation
- tracking of external software dependencies in the SiteDefs module
- build management with uniform make conventions for modules
- support for multi-target development

**YAM** does not directly provide support for test harnesses (for which many good options are available) but instead provides a make rules at the `module` level for invoking regression tests. Also, test software is expected to be version controlled as **YAM** modules.

### B. Meeting Stable Concurrent Development Needs

- **Support for independent development sandboxes:** One of **YAM**'s central tenets - supported by the toolkit - is the use of independent sandboxes by developers for pursuing their development activities without stepping on one another's' toes.

- **Concurrent development on short-lived branches:**

Software changes are in many ways like seismic tremors. A long sequence of mild tremors is easily handled, while large, though occasional shocks (with perhaps less total energy) can be very disruptive. Frequent branch/release cycles allows the team to absorb and adjust to each other's changes with relative ease.

**YAM** provides commands that simplify the creation and merging of `module` branches. The main trunk is used for releases while branches are used for development. The use of private CVS branches allows isolation of developmental, immature code to avoid premature interactions and instability. Furthermore, it allows multiple developers to be work independently on the same `module` with minimal coordination overhead. Developer's merge only mature and tested code into the release branch. **YAM**'s simple interface to the management of branches helps keep them short-lived and avoids integration difficulties and instabilities.

**YAM** provides conventions for auto-generating strings used as release and branch tags. Release tags contain a release number that increments with every release, and branch tags contain the release tag as well as the user id to create private branches.

To allow catching up with released changes for a `module`, **YAM** provides a `sync` command that creates a new branch by merging in development on the existing branch with the released changes on the main trunk.

- **Isolation from others' releases:**

During development, other `modules` continue to evolve and be released by the team. However, individual developers are free to choose the right time to sync up his/her sandbox with the new releases which may effect the scaffolding code. This choice helps avoid unexpected and forced sync up efforts that distract from the development at hand. If a team release happens to contain a relevant bug fix or new feature then the developer may make the effort to sync up, or else the developer may well choose to wait till the

development task is completed before bothering to sync up. Such selective isolation provided by sandboxes enables productive development even in the midst of fast evolving code within the team.

While such short developmental cycles are the norm, there are times when specific developments require longer than usual development on isolated branches. Though not recommended, these are not precluded by **YAM**.

- **Module-Level encapsulation increases stability:**

The requirement that `modules` be usable as link as well as work `modules` requires `modules` to explicitly exporting of their interfaces. A powerful side-benefit is that it enforces a strong **module-level encapsulation** on the `module`'s public interfaces, which in turn helps significantly reduce the coupling across `modules`. While compilers enforce class-level encapsulation by flagging disallowed access to private and protected parts of a class, the `module`-level encapsulation means that `modules` cannot access classes, header files and libraries from other `modules` that have not been exported by the `module`. Hence, this mechanism allows the enforcement of private and public parts of a `module` and the permissible cross-coupling across `modules`. `Modules` can "hide" classes that are meant to be private by simply not exposing them in the exported interface for the `module`. The `module`-level encapsulation also means changes within a `module` that do not effect the exported API are hidden from other `modules`. This implies that significant amount of change can occur internal to a `module` while keeping the rest of the system unaffected - promoting a higher degree of system level stability.

### C. Meeting High Quality Software Needs

- **Explicit module-Level interfaces improves software organization:**

The need to be able to use a `module` as a work as well as a link `module` has the important benefit of requiring an explicit definition of the externally visible API for the `module`. Such explicit communication enforces clear functional and interface boundaries across `modules` eliminating the possibilities of hidden coupling and interactions across the `modules`. It also helps exposes poor `module` definitions and improves the modularity and organization of the software. Poor `module` definitions will be reflected in the large number of links that need to be exported for the `module`. Such poor design choices are caught early leading to better software organization and design transparency.

Due to the `module`-level API encapsulation enforced by **YAM**, the only coupling possible across `modules` is that through the exported API from the `modules`. Thus classes, header files, libraries etc. that are not explicitly exported remain private to the `module`. This encapsulation thus takes the compiler enforced encapsulation of class definitions much further by enforcing privacy of classes themselves! Exposure of classes within the `module` API has to be explicit - avoiding inadvertent inter-module

cross-coupling and complexity in the process. It has been observed [1] that such information hiding is the key to improving software quality.

- **Separation of stable and immature software**

The module-based organization is especially conducive to keeping stable and legacy software in modules different from those with new and immature software, allowing careful monitoring and attention to the new developments.

- **Refactoring of software is not a big deal**

A key requirement to improving software quality is the ability to re-factor whenever the need arises. The module-based organization together with the minimal API exposure facilitates refactoring by localizing it to modules, reducing the risk and impact on the rest of the system, and helping reduce the effort needed to achieve it. One strategy, when significant refactoring is required, is to create a new module that co-exists with the existing one. This facilitates side-by-side testing and on conclusion, the old module is retired and replaced with the re-factored one.

- **Extensive module verification prior to release**

There is a strong onus on developers to thoroughly test their changes before releasing them to the rest of the team. Furthermore, YAM's release commands automatically run several checks to verify a module's readiness for release such as: that the modules are built against the latest scaffolding code; that all binaries are properly built; that a ChangeLog entry has been created etc. before allowing a release to proceed. Such continual testing of incremental changes and monitoring of quality helps improve the overall quality of the software and to identify problems early.

- **Continual Integration and Test**

When there is a miscue, rolling back a change is not a big issue either. The team settles into a "release-early, release-often" process where the software evolves and anneals at a steady and smooth pace. As changes keep merging into the release stream, they continue to be exercised and shaken-out as the team uses them. These continual development cycles painlessly replace the otherwise large and carefully coordinated integration efforts that are otherwise needed to merge changes from long-lasting branches. YAM makes co-developers into the first line of beta testers of new changes. This takes advantage of the observations [2] that "all debugging is parallelizable" and "given enough eyes, all bugs are shallow". YAM embeds continual quality improvement into the development process itself reducing the effort needed for addition quality assurance efforts.

When others on the team make module releases, it takes only a few seconds for a developer to update their sandbox to pick up the new release as a link module in their sandbox since no builds are involved. This enables continual exposure and use of new developments by the team making them effectively beta testers that help shake-out new developments on a continual basis. This also reduces the impact on the system from each of these smaller releases facilitating a smooth annealing of the ongoing software development.

- **Support for sub-system packages**

While YAM packages are used for product configurations, they can also be used to define sub-system configurations. Thus sub-teams responsible for sub-system development can manage their development efforts using the same YAM infrastructure as for the system-level development. Working with sub-system packages allows the teams to develop, test and manage releases at the sub-system level. This is a natural partitioning and decentralization of the development effort, permitting a thorough vetting of the sub-system components before integration into the rest of the system.

- **Support for developmental configurations**

The process of software development often requires the creation of development configurations specific for unit testing, extra instrumentation for debugging and profiling, use of stub software, for demos, system level tests etc. The support software needed for such configurations is valuable for the development effort. The YAM framework insists that **all** software - including such support software - be captured in YAM modules. Furthermore, YAM allows the creation of YAM packages for these important development configurations. Sandboxes for these packages are easy to create, and their evolution can be tracked through package releases. Thus, YAM brings such support software and development configurations out of the hidden shadows and integrates them directly into the main-stream development effort. Once available, it is not surprising to see how valuable a resource they are for the team and how widely such configurations get used. Furthermore, once easily accessible via modules, portions of such support software get reused from one development configuration to another.

- **Module level regression tests**

The module level organization also facilitates the use of regression tests throughout the software by including module level unit tests within the module. The make system supports rules for automating the running of regression tests in a uniform manner across the modules. Developers are expected to run and pass the regression tests before making a module release. This allows version control of the test code and its evolution as to keep up with changes in the modules.

- **Built-in reuse model improves quality**

The very process of making software reusable by organizing them as modules promotes their reuse by other development efforts. Such reuse helps improve the quality of the module software. get reused. It promotes the "grass grows taller as it is grazed" [2] principle for software development.

- **Multi-site and multi-target support**

By building in support for multi-site, multi-platform development YAM avoids the messy issues that can arise when the software needs to be ported to a new target, or a new compiler needs to be used, or the software needs to be deployed in a new environment. The transition effort to accommodate such events are easily handled by adding new targets and sites to the YAM project configuration. This requires no changes to the existing targets and sites and

can thus be developed and tested independently without destabilizing and impacting the development team.

- **Templatized makefiles**

**YAM** provides makefile templates (which can be tailored by projects) as well as a host of helper make rules and conventions that allow the module level makefiles to be very simple. This allows the use of uniform conventions across the software, but also removes the burden of developing and maintaining complex make rules from individual modules.

- **Inter-module dependency tracking:**

**YAM** provides support for the automatic generation of inter-dependencies among the C/C++ modules. With the decentralized organization of the software into modules, such dependency information is valuable for assessing the impact of API changes within a module on other modules prior to making the changes. Furthermore, the release process uses this information to alert the user to any API changes that may have occurred so that the user is able to follow up and make additional releases as needed.

#### D. Meeting Rapid Development Needs

- **Fast development sandbox setup and turnaround**

**YAM's** use of link modules allows new sandboxes (even large ones) to be built up easily and quickly. The scaffolding code for the modules under development, can be generated very quickly using link modules. Beyond the obvious speed benefit from fast sandbox setup, it also helps get developers used to the idea of creating and abandoning sandboxes for specific developments as a matter of course. Such agility in the development process is in sharp contrast to the not uncommon behavior where developers hang onto their functional sandboxes for as long as possible to from the fear of long build times needed for new sandboxes, or the fear of crippling their existing sandboxes through a merge!

Furthermore, the separation of stable and developmental software into different modules allows sandboxes to use link modules for the stable parts of the software and just source for the developmental modules- with much smaller build times compared with building the full software source.

- **Fast system sandbox setup**

The use of link and work modules also allows the set up sandboxes for integration and test, training and demonstration purposes.

- **Handling large code-base**

As the code-base gets large, build and checkout times to get the full source can become unacceptably large. However, growth in the size of the code-base is typically a non-issue during day-to-day development for the **YAM** framework. At any given time, an individual developer is typically working on small number of modules. The build time for a sandbox is directly proportional to the size of these modules alone since the rest of the modules serve as scaffolding link modules.

- **Parallel makes for multi-target builds**

As part of its support for multi-target development, **YAM**

provides build and make rules to carry out builds for all the supported targets in parallel. Thus the build time stays more or less constant as new targets are added as opposed to scaling linearly with the number of targets.

- **Simplified branch and release process**

**YAM** provides scripts that make the creation of development branches effortless for the user. The user does not have to know the intricacies of the branch and merge process for the underlying version control system. **YAM** uses conventions for tags and branch names which once again the user does not have to deal with. This allows users to go through the module development and release cycles with very little effort. Such low cost encourages developers to release modules and terminate the branches whenever development is done and avoids the big integration push-ups that can result from long-lived branches.

- **Decentralized development and release process**

Having organized the software into modules with strong encapsulation allows **YAM** to decentralize the development process and reduce the mundane coordination overhead within the team. The compartmentalization of the software functionality, the isolation from developing on branches, and the incremental development process provide a very stable foundation to add new capabilities with low risk of jeopardizing the development process. Even when changes turn out to be not acceptable, rolling back is not a big effort. The reduced risk allows developers to spend their energy more productively on development rather than on coordination overhead. Extensive team coordination however is necessary when far-reaching module API changes are to be undertaken,

- **Metrics collection from releases meta-data**

The MySQL database tracks extensive information about the module and package releases. For those so inclined, the data is available to analyze and fine tune the development process. Thus it is easy to find the modules that are going through a lot of change, or the ones that are being worked on simultaneously by several developers, or assess the team's productivity in terms of the number of releases etc. from the releases meta-data.

#### E. Meeting Reuse Needs

- **YAM module interface facilitate reuse**

**YAM's** module and package concept is at the heart of module reuse. While there are virtually no restrictions on what software can go into a module, the process of assigning a software to a **YAM** module requires a few critical steps that go a long way to making the module reusable. Firstly, the modules public interface has to be clearly defined so that it can be exported. Similarly, the dependency of the module on other modules, as well as external software, has to be clearly defined as well. These steps very clearly define the functional boundaries and interfaces for the module. The strong encapsulation provides a layer of protection from inadvertent hidden dependencies and cross-coupling creeping back in since all external interactions have to explicitly exported or imported.

- **Packages can share modules**

**YAM** modules can belong to more than one **YAM** package. Packages serve as simple bundles of constituent modules, and there is nothing package specific within modules themselves. **YAM** provides simple mechanisms to add and remove modules from package definitions to facilitate the sharing of modules.

- **Standardized make interface for modules**

Furthermore, **YAM**'s use of uniform conventions and rules for building a module makes it easy for users to bring unfamiliar modules into the mix without having to work through complex build issues for the modules that might otherwise be necessary. Thus it is trivial to reuse a module belonging to a **YAM** package into another **YAM** package.

- **Packages are also reusable**

**YAM** allows packages to contain other packages. Such containment simply adds the modules from the contained package to the list of modules for the parent package. Such hierarchical package definitions are effective for not only managing a product-line, but for also building up more complex products from existing ones.

- **Packages releases manage changes**

**YAM**'s package releases are simply specific set of module releases for the modules that belong to the package. As modules evolve, there is no guarantee that an arbitrary combination of module releases are compatible with one another. Package releases establish baselines where the combination of module releases in a package release have been tested for compatibility. Package releases are very lightweight since they do not contain any software of themselves. Regular package releases help the team to track the evolution of the package as well as to help bracket the source of any bugs that may be discovered. The existence of such baselines also establishes the quality of the constituent modules and facilitates their reuse elsewhere.

#### IV. USAGE

**YAM** started out as a framework for supporting fast-paced simulation software development in a team environment. **YAM** has evolved considerably over the years and has been adopted by several other software development teams for managing their software development. Such development efforts include projects such as: physics-based simulation development at the DARTS Lab [11], CLARATy robotics software development [12] as well as mission-critical software development for missions such as the Mars Exploration Rover flight software [13], Space Interferometry Mission [14], Deep Impact fault protection software [15] etc. The size of development teams has been in the range of 5-30 developers. Most of the development efforts have involved teams sharing common development environments permitting the use of link modules. However, there is significant diversity in the development policies and models used by the teams. Some of the determining factors in defining the development models include the stability of the development team, experience level of the team, mission

criticality and schedule drivers, homogeneity in the software, product-line needs etc. Indeed, even though several of the teams are working towards the development of a single product, they have nevertheless adopted **YAM** to support their development.

We now take a closer look at the DARTS Lab's software development experience. This lab is responsible for product-line consisting of physics-based simulation tools for a range of application domains including spacecraft, planetary rovers, entry/descent/landing systems, airships etc. These distinct simulation products share a number of infrastructure modules for vehicle dynamics models, device and environment models, simulation framework, graphics visualization, user interface etc. In this **YAM** setup, there are around 200 active **YAM** modules and 30 **YAM** packages and the software supports 2 to 5 host targets. The development team is collocated and has ranged in size between 6 to 10 developers and includes engineers and software personnel. The software is a mix of C/C++ software, with significant amounts of Python, Tcl, Perl and some Fortran and is approximately 2 million lines of code. The development pace has been fast with over 2400 module/package releases during 2004, and over 2800 in 2005. The development process is fairly decentralized. Of course, the **YAM** software is itself managed as a **YAM** module!

#### V. CONCLUSIONS

The **YAM** software development framework provides solutions to many thorny challenges involved in team-based concurrent software development, managing software product-lines and developing reusable software. The **YAM** processes integrate concepts spanning software organization, build management, release management, and software reuse that work together to provide a flexible development environment for managing complex software development. Software designed for reuse will not stay that way for long without reuse-friendly software processes such as **YAM**. **YAM** is in use by several mid-sized software development efforts. It has also been successfully used for managing mission critical software for NASA's space missions (eg. MER, Dawn), where traditionally, the cathedral development paradigm has been the norm.

#### ACKNOWLEDGMENTS

We would like to acknowledge Garth Watney for the many constructive suggestions that have helped improve **YAM**'s usability over the years. The research described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration, and has been partly supported by the National Science Foundation Grant ASC 92 19368.

#### REFERENCES

- [1] F. N. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 1995.
- [2] "The Cathedral and the Bazaar website." <http://www.catb.org/~esr/writings/cathedral-bazaar/>.
- [3] "Agile Alliance website." <http://www.agilealliance.org/>.
- [4] "Extreme Programming Resource website." <http://www.xprogramming.com/>.
- [5] "The Perl Directory website." <http://www.perl.org/>.

- [6] "CVS Wiki website." [http://ximbiot.com/cvs/wiki/index.php?title=Main\\_Page/](http://ximbiot.com/cvs/wiki/index.php?title=Main_Page/).
- [7] "MySQL website." <http://www.mysql.com/>.
- [8] "PHP website." <http://www.php.net/>.
- [9] A. Maslow, *The Farther Reaches of Human Nature*. Viking, 1971.
- [10] "RPM Package Manager website." <http://www.rpm.org/>.
- [11] "DARTS Lab website." <http://dartslab.jpl.nasa.gov/>.
- [12] "CLARty website." <http://claraty.jpl.nasa.gov/>.
- [13] "2003 Mars Exploration Rovers website." <http://www.jpl.nasa.gov/missions/current/marsexplorationrovers.html/>.
- [14] "Space Interferometry Mission website." <http://planetquest.jpl.nasa.gov/SIM/sim.index.cfm/>.
- [15] "Deep Impact website." <http://deepimpact.jpl.nasa.gov/home/index.html/>.