CHAPTER 6

# Modularized System Integration

In the previous build and source code control chapters the resource emphasis was the individual development engineer. The interaction between development engineers was largely limited to their co-contribution to a shared development branch in the source code control system. For this system to work, each individual engineer will not only need to rebuild and test her own changes outside of the shared branch, but inside it as well. As the number of development engineers, the time required to build, and the complexity of the code increase this individual system starts to have difficulties. Modularized system integration provides some solutions for these difficulties. As regards product development, system integration is the act of putting together all the individual aspects of the product. In our previous discussions, system integration took place as the collective effort of all the individual development engineers and the build tools. In more complicated products, we find the need for a system integrator and a further subdivision of development labor into modules. The system integrator is responsible for integrating these modules into a system.

## When You Do Not Need Modularized System Integration

Modularized system integration is a specific approach to a complex problem in the development environment. This problem is usually precipitated by having too many developers trying to change too much code at the same time. In a conventional development group, modularized system integration is not used. Instead, most developers use an integration process termed "latest is greatest."

"Latest is greatest" refers to the individual files in the source base, or source branch. This process assumes that whatever file has been last checked into the source base is technically correct and causes no functionality regression in the product. A corollary assumption is that this check-in does not adversely affect the working of any other files that make up our product's source. These are very big assumptions with a nontrivial likelihood of being incorrect. People are not perfect, and developers are no exception to this rule. We expect there to be errors in the process of checking files in. The development organization must have the capacity to deal with

errors that are introduced in a latest-is-greatest methodology. Problems arise when this capacity does not exist or is overwhelmed.

Latest-is-greatest is far more economical than modularized system integration. If you can handle the capacity needed for latest-is-greatest, by all means do so. However, if you cannot, modularized system integration can keep you from digging yourself into a hole that you will never be able to climb out of. The problems with capacity are very similar to those discussed in Chapter 3, where we identified problems that precipitate the need for branching. If we consider an extreme case of a single file in a source base, we can see where the issue of capacity comes into play.

Assume a small group of engineers is attempting to fix defects and add features to a product. Bob fixes a defect, but the fix had to change some of the fundamental functionality that is used by all the features of the product. Bob only tested the single feature that he saw the defect in, and it appeared to work fine. However, Bob inadvertently introduced a subtle defect in 42 other features of the product. Along comes Tara. Tara goes to add a new feature, and because "latest is greatest," she is using a code base that has Bob's stealth defect. She implements her feature successfully, but unbeknownst to Tara, her feature has been implemented using the erroneous code introduced by Bob, which makes her feature addition dependent on code that will ultimately have to change. And the beat goes on. Eventually, we will find out about these problems, but you can imagine that the problems can be severely compounded by a myriad of checkins and can be very difficult to ultimately track down and resolve.

The definition of capacity here is the speed with which we can find out we have a problem and get ourselves out of it. If we are only digging ourselves small holes that we quickly find and get ourselves out of, we are okay. If we are regularly digging ourselves into large holes that take a significant amount of time to get out of, it can very quickly seem like we are taking two steps back for every one step forward in our development process. In the latter case, modularized system integration assists us. However, as we will see in this chapter, it does so at no trivial cost.

## Product Modularization

The individual development engineers or groups of engineers are responsible for specific modules that make up the product. The nature of a module will vary from product to product but generally its concept remains the same. A module is a part of the final product that can, to a large degree, be worked on separately from the product as a whole. Modularization of the product code becomes important to the development organization because it allows larger products to be split into smaller, more manageable parts that can be more easily worked on by individuals or small groups. These modules collectively make up the product as a whole. This kind of modularization is similar to the way modules constitute a modern personal computer. Conventionally, there is a motherboard that has connectors in it that accept devices providing specific services. Popular devices like video cards, memory chips, audio cards, and storage device controllers can be attached to the motherboard as modules. The modules are created by separate companies that specialize in the functionality that their modules provide. If there is a better or improved module available, you can take out the old module and replace it.

## System Integration Overview

To continue with the PC example, a system integrator would be the individual responsible for putting each module in the system and verifying that it functions correctly. In software development the process is similar. The system integrator manages the shared branch where multiple development efforts are brought together. Generally, modules are not so much replaced as they are updated. Since modules are developed in parallel, multiple module changes can come into the shared branch at the same time. The role of the system integrator is to verify that the module changes have integrated with the product successfully enough to consider the changes part of the product.

### Assumptions of Modularization

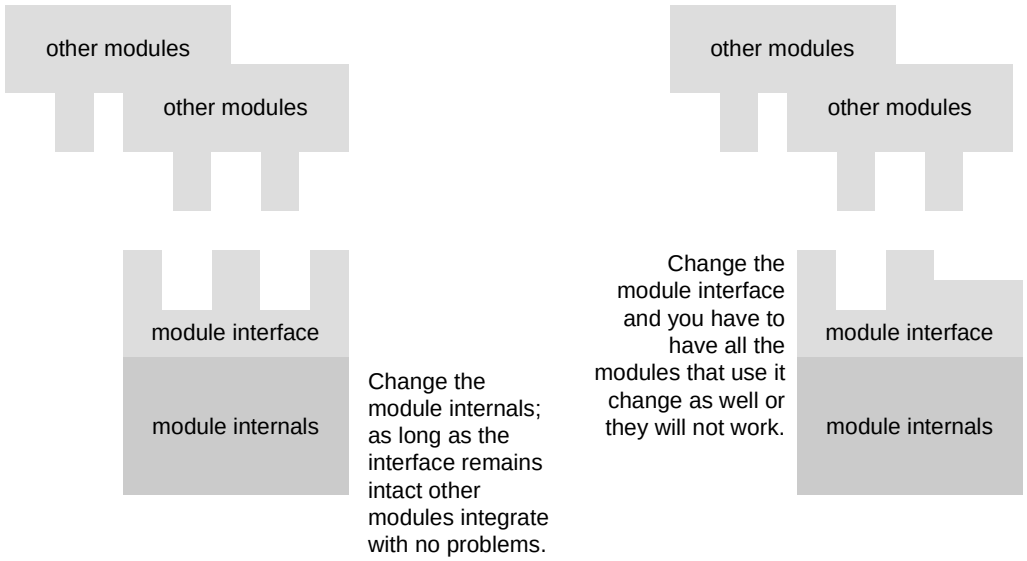
Proper software development modularization is important for the system integration activity to be successful. It is assumed that any module changes are verified as functionally accurate within the scope of the module itself before they are offered up for system integration. A sound card to be integrated into a PC system should, for example, show that it can adjust its audio output volume with its own volume control. The company providing the sound card should at least have verified that self-contained functionality before allowing it to be integrated into a PC system. Similarly, we want our software module to have passed some level of verification prior to going into system integration, not so much because we think module functionality verification will solve all of our problems—far from it. In the system integration activity we would like to focus on integration problems, not self-contained module problems.

An integration problem occurs in a modularized system when one module fails to work correctly with a different module for any one of a number of reasons. Modules can be subdivided into two parts. The part that we are most concerned about is the module interface. The module interface is the what other modules constituting the product interact with when the product is used. The second part is the module internals, which include everything but the interface. This subdivision of the module allows us to quantify more easily whether a change is likely to have integration problems or not. Changes in the module interface happen in different ways. A change in a part of the module involved with the module interface is more likely to cause an integration problem than a change in any of the other module internals. But, a change to the module interface itself is even more likely to cause a problem because it can require all other modules that use that interface to change (see Fig. 6–1).

### The Role of the System Integrator

The system integrator is responsible for managing the branch where the module changes come together. The system integrator maintains a source control tag that identifies the current set of files on the branch as constituting a product with some known state of functionality. The elements in this tag are not changed unless it is known that the changes will not adversely affect the product's functionality.

**Fig. 6–1.** Internal and interface changes to modules




The system integrator perceives module changes as a list of one or more files and their changed-file version values. Using the known functionality tag for the product, a source tree is created that represents all its files. The module changes that have been submitted for system integration are then laid on top of these files. The first step in system integration is to ensure that the product *can* be built successfully with these changes. As we have seen in Chapter 4, the build process can do a very good job of checking dependencies between modules for us. It is important for an organization to identify the expectations of the system integrator when a module fails to build in system integration.

Minimally, the system integrator is responsible for determining that all aspects of the build environment in which the development team operates are correct. Additionally, the files that are shared and used to control the build (e.g., makefiles) may need to be modified by the system integrator to accommodate module changes. Modifications to these files would not be added to the tag, but would instead be considered part of the module change. Depending on the sophistication of the product and the technical capability of the system integrator, additional tasks may be considered. Where simple programming mistakes were made due to build environment considerations or simple typing errors, the system integration may be authorized to make changes to a module's code. Common examples: Reference is made to another product's files that uses a hard-coded path to a developer's workstation instead of to the official location in the product distribution tree; or someone mistyped an include file that is declared in the source file. These are called intent errors and are the easiest to correct. More consideration should be given to more serious errors that involve implementation or even design. The goal of the system integrator is

not, however, to replace development expertise but rather to provide a level of specialization that helps in offloading pure development activities. Where the system integrator is not advised even to attempt resolution of module integration errors, these errors are brought back to the development team that initially submitted them.

After the module changes can be successfully built by the system integrator, the next area of concern is functional verification. Criteria must be set up that identifies the desired functionality before a change gets committed into our known-functionality tag. Setting these criteria can be very tricky. If the criteria are set too high it becomes too difficult to get changes into the product, but if they are set too low the product source base can quickly become nonfunctional. After the criteria are set, the system integrator will either evaluate the product's ability to meet the criteria or have some other individual perform the verification. If verification fails, we immediately know by definition what change caused it, since only the module that we are attempting to integrate has changed the product. The problem is then bounced back to the group that submitted the module change.

Eventually, we should have a module change that gets past the verification testing. If so, we are ready to update our known-functionality tag to include the module changes. The updated tag is then available for all developers to use as the latest known functional state of the product. The system integrator should be the only individual to change the known-functionality tag. The system integrator continuously loops in these tasks (see Fig. 6–2).

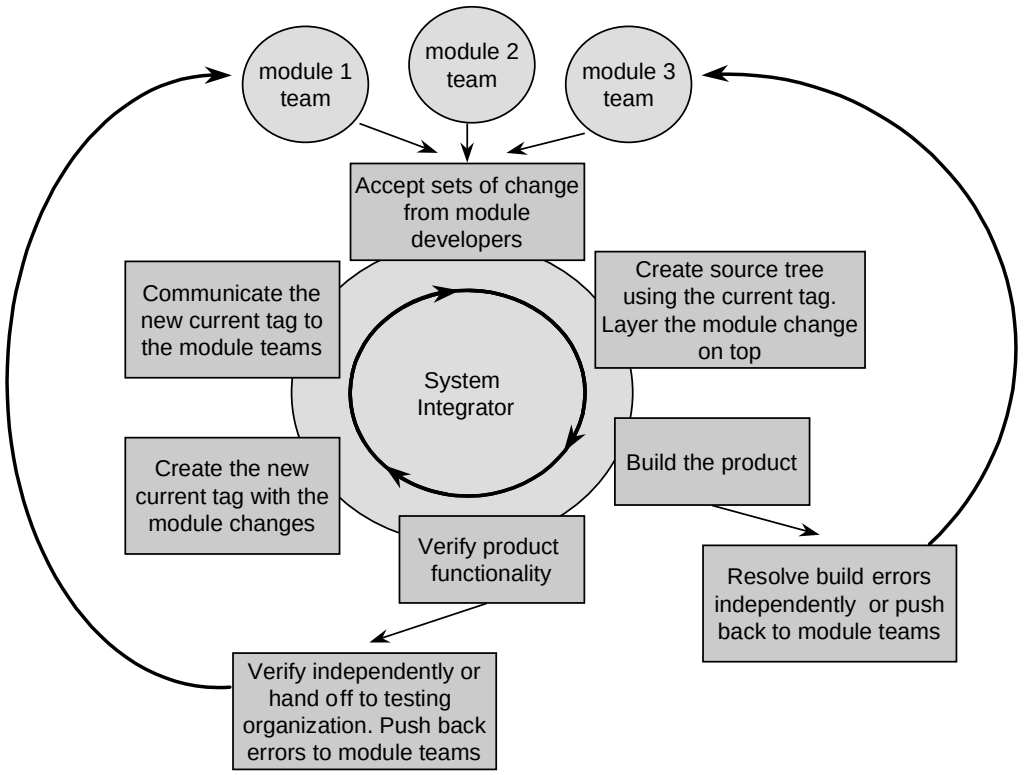
### Developing Around The Tag

All module changes must be buildable before they can proceed with the system integration process. Our tag of known functionality is the set of source files that represents our latest integrated development work. As work continues, it is expected that the development organization should be basing all of its changes on top of this tag. Unfortunately, this tag will change with each new module change that gets through our system integration process. The dilemma: We keep having to change the tag so that it will be current, but our developers keep having to change the source base they develop from so that their changes will get through. This forces us to choose carefully the frequency of such changes so that development organizations can plan appropriately.

We want a tag change frequency that allows the groups developing the modules to have enough time to get a set of changes together for submission. Setting the tag change rate at one day would probably be to often. The development group would not have enough time to successfully develop or test any reasonable amount of change. Setting it at once a year would be too long. By the time the module change came in for system integration, it would be so different from what it was before that it would take a very long time to work out all the conflicts it had with the other modules that interacted with it. A good frequency of change, one or two weeks long, will usually work well for at least one of our two classes of development activity.

As we have discussed in previous chapters, there is usually a significant amount of activity involved with fixing product defects present in the source base. This class of activity will be termed product maintenance. Our tag revision frequency should be set up to allow an efficient amount of product maintenance during the tag period. Ideally, there should also be enough time

**Fig. 6–2.** The system integrator's role




to do some of the more minor aspects of our second class of activity, feature development. Feature development involves adding functionality to the product. We cannot expect a major feature to be developed and tested in the time span between two tags. To do so would require too long a tagging frequency. Major feature development must occur in a separate branch and will be required to sync to the current tag before submitting for system integration.

### Scheduled Integration vs. Event-Driven Integration

Regular updating of the current source tag imposes a schedule for the system integration cycle. The schedule identifies the windows of opportunity for getting change into the system and also when new current tags will be available for development groups to use. A regular schedule for these events is preferable to a purely event-driven system. In an event-driven tagging process, the tag would be generated when deemed appropriate. Although having tags generated as frequently or infrequently as needed may seem advantageous, it is not. Regular system integration scheduling allows us to divide and conquer the enormous amount of change that we are trying to get into the product source base in an organized fashion.

Quantitatively, the amount change that occurs within the maintenance class of development is usually much larger than the amount of change associated with feature addition. This is because the product can be viewed as an assemblage of features, where each feature is added to the product only once. Once the feature is in the source base, any activities that involve fixing the feature are considered maintenance. This maintenance can occur many times over. We will want to have an efficient mechanism for getting maintenance into the current source tag. In an event-driven process, the developer wishing to apply changes to the current source will never know for certain when the next tag will be made available. This is a dilemma because most developers don't have just one change to the source base that they are working on. The developer will have to decide whether to continue making additional changes to her source or to stop all work for some unknown period of time and wait for the next window to get changes into the tag. The latter choice is rarely made because it is not practical or profitable for a development organization to do business this way. If instead our developer continues with additional changes, she could have a very large amount of change to put into the system once a window opens up. Too many changes then defeats the purpose of the tag updates.

The larger the amount of change submitted to the system integrator, the longer it will take to resolve the build and verification errors that result. Eventually we can reach a point where we are slowing down the system integration process dramatically. As each successive module goes through the process, it takes longer to resolve all of the system integration errors. A module will, through no direct fault of its own, encounter difficulties compounded by all the changes to the source base made to accommodate the previous modules. The module will then require additional unplanned changes, and so on successively with each submitted module. To relieve this problem we must resolve the initial scheduling problem that caused our development engineers to make their module changes larger.

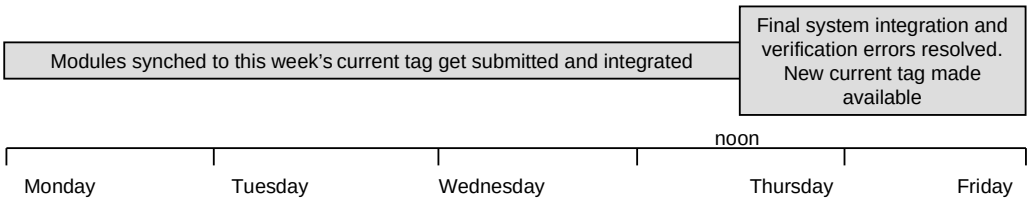
The development group is provided with a regular system integration schedule so that the developers know that they have a regular window of opportunity and can expect a new current tag to be made available on a regular basis. This allows the developers to plan their efforts around a predictable system integration process.

### System Integration Schedule

The system integration schedule we elaborate will have a frequency of one week with occasional variations. The week will consist of five working days and will be divided into preferred times of system integration (see Fig. 6–3). The schedule will indicate when system integration can accept module changes and when it cannot.

Starting on Monday, developers wanting to get their changes into the tag for that week will have a new current tag made available for them. They will have until noon on Thursday to submit module changes to the system integrator. Thursday afternoon will be spent integrating the last of the module changes for the week. Friday will be spent resolving any remaining build or functional verification errors. As soon as these are resolved, the next current tag will be made available.

**Fig. 6–3.** System integration schedule




### Branching Strategy for Module Development

Module development encompasses both classes of development activity, maintenance and feature development. The mechanism that we use with our source code control system to manage this will be defined as our branching strategy for module development. We use a defined strategy so that we will have an infrastructure of assumptions that can be relied on throughout the system integration process.

Prior to the submission of a change to system integration, the module development team must have completed actual changes to the files that are being submitted. In order to record these changes properly we require them to be checked into some branch or trunk of the source code control system, whichever is more appropriate. The files that have been changed will be identified by their own tags. The module developers will view their changes as a layering, where the current product tag is used to instantiate a source tree and their change tag is used to create a set of files that are layered on top them.

## Parallelism in System Integration

Obviously the system integration cycle will not work efficiently enough taking just one change a week through the system. We will need to introduce parallelism into this process to support multiple changes. The amount of parallelism will depend on several important factors.

### Limits on Parallelism

- The level of modularity of the product source base. If the source base is very modular, we should be able to make changes in several modules simultaneously. Consider our word processing example where one module controls the spell checking of text and the other module controls the user interface dialogue box that comes up when we want to save a file to disk. In a well-modularized system we should expect that, even if both these changes were developed separately, they should be able to be integrated together without any problems.
- The length of time required to complete a build. The build process is used to test a portion of a change's ability to integrate. If this process is unduly long it can affect our ability to use parallelism.

- The amount of resources available to process changes. The amount of system integration resources, both human and mechanical, available to process multiple changes.
- The length of time required to perform functional verification.
- The reasonable turnaround time to get build and functional verification errors resolved, both independently and outside of the system integrator.

These limitations will determine the level of parallelism in the system integration process and, consequently, that process's bandwidth. A goal of the system integration process and the development process as a whole is to provide system integration at a level that keeps up with the demand of change generated by the development organization. This goal must also be supported with adequate control on the functional verification process. This requirement provides a check on the amount of change generated by the development organization. If a development organization generates a large amount of change but each change is unable to meet the functional verification criteria we would have a strong indication that the number or organization of these changes may not what we would like them to be.

### Parallelism Strategies

Returning to our system integration schedule in Fig. 6–3, it is obvious that we cannot wait until Thursday noon to start integrating all the changes we have received since Monday. The system integrator will begin working on any submitted change as soon as possible because they will be maintaining a rolling tag that will eventually be our next current product tag. The system integrator starts the process by creating what we will call the working tag. This working tag is a private tag defined by the system integration and is initially a copy of the current product tag. Eventually the working tag will become the next current tag. There are two ways to do this and a good system integration will use whichever way appears to be most applicable to the tasks at hand.

Single-change parallelism takes each individual change separately and starts the system integration process in parallel, so if five changes are submitted, we would start five individual system integration processes. We do this because we don't yet know if any of these submitted changes are going to be successful and we want to assume that, if they fail, we have not wasted any time. As each change makes its way through the process and hits an error, it can be dealt with separately without affecting the progress of the other changes. Eventually one change will make it through the process. That first change will be added to our working tag. Now our remaining four changes must be dealt with.

The objective is to conserve work where practical and to cut losses as soon as possible. Each remaining change must be evaluated for its potential to integrate cleanly with the new working tag. At one extreme, if we believe a change will integrate cleanly and is near successful completion of its functional verification, we will wait for it and integrate the change on top of our new working tag. Assuming we can get through the build aspect of this integration with none or only trivial errors, we will add it to the tag. Likewise for each of the remaining three changes. Notice that we did not proceed to the functional verification step again. We could have proceeded with

functional verification and required it for all four changes, but experience shows that we generally do not have the time to do this. The change has already passed functional verification, so we will make an explicit assumption that the code will support a level of modularity that will allow the change to work with an acceptable amount of risk. There is no question that this process does involve risk. To minimize the risk, some testing that requires little time to complete and analyze can be used. Of course we could go through full functional verification, but we would have to consider its time cost.

At the other extreme, if a change is still having difficulty getting through the build portion of the process, it may be time to cut our losses. We should consider the option to pull the change out of the current system integration cycle altogether. Any change may be aborted during the system integration process. If a change has serious problems in building or functional verification, the system integrator and development team may opt to pull it out of system integration. This would be done in situations where the change was too far out of sync with the current source tag or where previously unknown dependencies surfaced and needed additional work to resolve. A change is pulled from system integration when it is more efficient to resolve its issues back in the module development state or when working on the change jeopardizes the next scheduled system integration cycle (see Fig. 6–4).

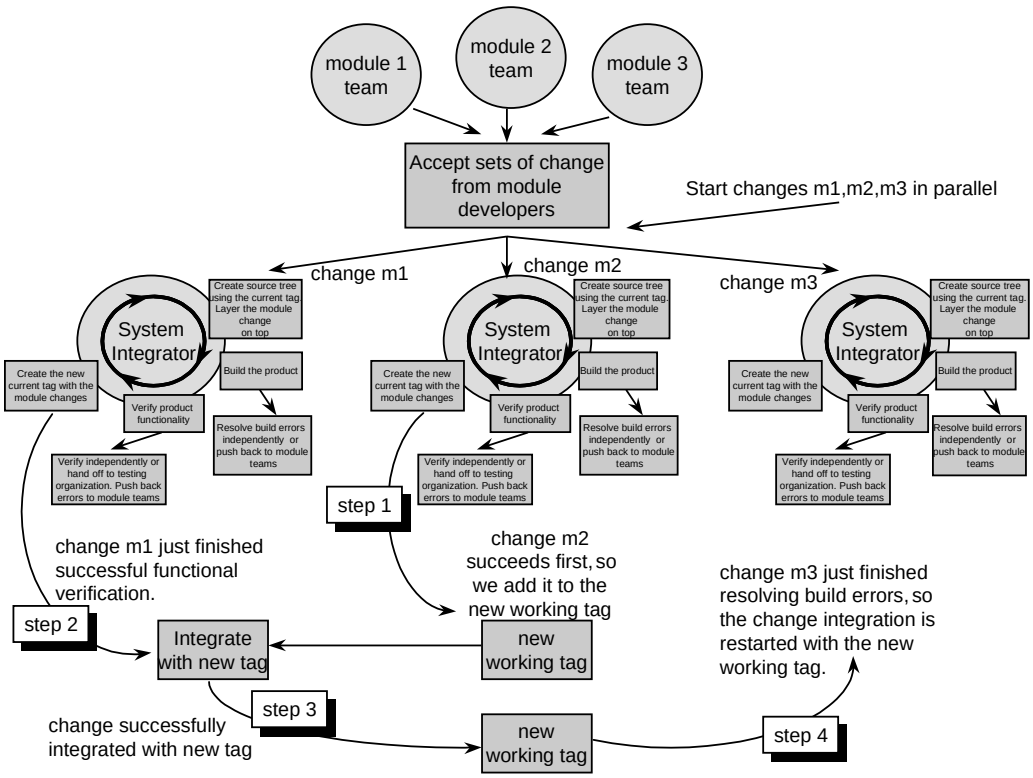
As each change completes, we start with the next change. Each new change will use the working tag, not the current tag. In this process we slowly start to diverge from what our development engineers have been using as the source tag that they developed their change on. There is no way to avoid this without serializing the entire change process. In a completely serialized change process, no change would be able to start until the previous change completed. Assume three development engineers each wanted to do a different change. While Moe was completing his, Larry would wait on Moe and Curly would wait for Larry's change. The vast majority of development time would be spent waiting. This would not be practical, so we customarily attempt to get away with as much managed parallelism in development as possible.

Clustered-change parallelism is a variation on single-change parallelism. Instead of taking each change individually, we can opt to combine changes from the outset. This is particularly useful if we know that the change may have dependencies. It is best to start work on dependency error resolution as soon as possible in system integration. By clustering changes we can increase the number of changes that are going through the parallel integration process. But with this increase in raw bandwidth we introduce additional risk. If we cluster two changes and one of the changes has an error, the error will hold up both changes from proceeding through integration. Since clustering uses the same process as parallel single changes, it can help us better utilize our allocated resources.

## Major Aspects of the System Integration Cycle

Major aspects of the system integration cycle remain consistent and each development organization will tailor the details to best fit their developers. We will now review each major aspect of the system integration cycle and discuss various implementation strategies.

**Fig. 6–4.** Change parallelization




### Accepting Sets of Changes

The system integration cycle always starts with the system integrator accepting a proposed change to the current source tag. If there are no changes, the cycle does not start. Change must be uniquely quantified to avoid misinterpretation. Our source-tagging functionality provides an excellent mechanism for accomplishing this. Each developer will create a source tag that represents all the files that she has changed. The tag name must be unique. This symbolic name of this tag and its owner are communicated to the system integrator. This is all that is needed for the system integrator to proceed.

As we have seen in system integration parallelism, it is important to make the system integration process as smooth as possible so it can run as quickly as possible. It helps to have an admission standard for the goal changes submitted to system integration. Ideally we would want the problems that we find in system integration to be unique to that activity. We should not expect a change to be submitted that would not have compiled even in the developer's own environment using the current product source tag so, at the very least, each change must

successfully build and execute using the current product source tag before it can be submitted to system integration.

### Source Tree Creation and Layering

The system integrator is in the business of evaluating change to the current product source tag. The evaluation begins by establishing a known source base, a tree represented by the current product source tag, and then layering change on top of it. The evaluation ends with a decision whether that change should be added to the next iteration of the current product source tag.

### Building the Product

The product build serves two purposes. First, the actual build will tell us much about the change's ability to integrate successfully into the source base. A change that does not accurately anticipate the dependencies that will need to be resolved during the build will alert us to potential integration problems with the change. Second, the build is needed to ascertain whether the changes have adversely affected the product's ability to meet functional verification. Changes needed for the files that control the build, like makefiles, commonly need to be fixed by the system integrator. Although we could expect the first change made against the current source tag not to need modification, any additional changes that involve the same control files could not have anticipated the first's alterations. In resolving build errors other than these, careful consideration is required.

When resolving a build error, it is important to understand the implications of making a change. Obvious errors like typos, spelling mistakes, or improper filename paths can be easily rectified. With simple changes like these, the developer should at least be notified that a trivial error was found and given details as to how it was resolved. Where change is needed that can affect product functionality, however, the developer must be involved. Whenever a change needs to be effected, its corresponding tag should be changed as well. Notification of this event should be given to all individuals involved with this change's integration.

If there are substantial problems found by the developers during the build process, consideration should be given to whether the submitted change should continue. Consideration must give to the amount of time required to resolve the problem during the integration process and to the validity of the changes being made. In this centralized system integration model, the system integrator is a shared resource. If one change takes undue repair time, the entire system integration process with all the other modules that want change are being slowed as well. The functional ramifications of the level of change being performed should also be considered. Remember, a successful build is only the first step. Generally, if there are major problems in the build, it is safe to assume that at least some of the changes performed will introduce some problems in functional verification. Part of the build portion of this process must carefully consider when it is time to cut losses on a change by aborting it.

It is important to understand that aborting changes will be a normal part of this process. It is part of the model. The model assumes that for reasonable periods of time the majority of

change development and integration activities can happen independently of each other. But, as we have seen, this cannot always be the case. We will have changes that occurred in parallel but break during integration. This is not the fault of the developer or the system integration. It is simply a side effect of the process that allows us to use a parallel model instead of a serial one.

### Product Functional Verification

The successful building of an integrated change is the first hurdle for a change. The second and last is functional verification. Functional verification gets into an area that is purposely avoided in this book, namely product testing. Software testing is a complete field in its own right and we will not try to cover it. Suffice to say that we will want to complete some activity after a build that gives us some estimation of the product's functional capabilities. For a massively parallel system integration process this activity has to be relatively brief.

The product development process must have some activity that tests the product. This activity can be automated, manual, distributed, centralized, or—more than likely—a combination of all of the above. It will not be practical to run each built change through this process, but we will want to know something about the product's functionality before proceeding to the next change. We will focus on using a highly automated subset of the product's testing. Tests should be chosen for their ability not only to run automatically but to have their results interpreted automatically as well. A test that requires a human to read and interpret thousands of output lines is not a good candidate for functional verification. The duration that it takes to run these tests must be directly related to the ability of system integration to keep pace with the rate of submitted development changes while maintaining the level of quality needed for the next change integration. A test set that takes too long to run will result in an ever-growing backlog of changes that need to be integrated. A test set that is too short and does not cover enough functionality will result in a layering of broken changes that will be more difficult to resolve. Finally, the set of tests must be chosen on the basis of a solid set of functionality that the product should always meet. During system integration, we do not have the time to debate whether it's acceptable not to pass a specific test. We will want the set of tests performed in functional verification to be nonnegotiable. They must be passed before we proceed with change integration. When a test fails it should clearly indicate what functional aspect of the product is not working.

When a functional part of the product is not working, we start a process similar to that used to resolve build errors. The system integrator brings in the appropriate module modification most likely to introduce a change that would fail the test. Remember, as part of our system integration process, the most recent change must pass these tests. So this change should either directly or indirectly cause a test failure. The change that has a direct effect is obviously more easily resolved. The indirect effect will be more troublesome and may require bringing in developers that had nothing to do with the initial change. This is to be expected. In software development, seemingly unrelated events can be related through long chains of loose dependencies. In the extreme, subtle program errors can surface through the mere rearrangement of a code in the

executable file. As with build errors, we have to consider how much time it takes to resolve these functional errors. In addition, how much change is needed to correct them? System integration allows us to answer these questions using the same power tool that we can use in the build portion. We can easily abort a single change and proceed with the other changes.

### New Current Source Tag

After the change has been successfully integrated and the functionality has been verified, we are ready to update the current source tag. An example of the name used as the current source tag would be "product.current." In a parallel system integration model, we add some minor complexity to this activity in order to preserve the modularity of our change process (see Table 6–1). As we have already seen, it can become important to be able to back out previously integrated changes.

The system integrator will support the creation of two types of tags. First, the current product tag is a single tag that represents the current set of source files that constitute the prod-

**Table 6–1.** Parallel system integration source tag types

| Tag type | Tag syntax | Tag examples | Description of tag use |
|---|---|---|---|
| Current product tag | <product>.current | ynix.current | Developers are required to use this tag as the base for their changes. All changes layered on this tag should build a functional product. |
| Weekly current product tag | <product>.current.<month-day-year> | ynix.current.11-16-97 | A historical reference tag. To be used to recreate a product source tree from a particular point in time. |
| Work in process tag | <product>.inprocess | ynix.inprocess | The work inprocess tag is used by the system integrator as he prepares for updating the current product tag. It is a combination of the current product tag and all successfully integrated changes this week. |
| Change-by-change in-process tags | <product>.inprocess.<month-day-year>.rev[number] | ynix.inprocess.11-16-97.rev3 | A historical tag for the system integrator. This tag is actually a copy of the inprocess tag as each new change is added to it. The date tells us the week, and the rev tells how many changes have been introduced during the week. |

uct. On our tag revision schedule (Fig. 6–3) we can expect this tag to change every Friday of the week. Before this tag is changed, we will copy the tag to a name that identifies the previous week's tag. The current tag changes, but each previous week's tag stays consistent because it is a historical marker of what file was current for the week. The previous week's tag will reflect this by capturing the week it represents in its name. An example of a historical weekly tag name would be "product.current.11-16-97," which identifies the tag as being current for the week of November 16, 1997.

The second tag is the system integrator's working tag. An example of a name used for a working tag would be "product.inprogress." As each change is successfully integrated during the week, it is added to the working tag. We will also want to have a historical tag that represents each individual change as it went in. To do this correctly, the system integrator will need to have a unique identifier associated with each change scheduled for integration. The system integrator will keep a rolling list of what changes were introduced with each new addition to the working tag.

The change-by-change inprocess tags always allow the system integrator to go back to any previously successful source tree. Since we have a date stamp on the tag label for convenience, the revision counting can always start at one as soon as each new week's integration begins. The system integrator will associate whatever unique changes went into each tag revision. An easy way to represent each change is by citing the product the change is for, the submitter's login id, and the submission date. In Table 6–2, we can view the amount of change as seen by the system integrator for an example week which starts on Monday, November 10, 1997, and ends on Friday, November 14, 1997.

### Communicating the New Current Tag to the Developers

For ease of use we use a rolling tag for the current product source tree. As we have seen, it is important for developers to know when this tag changes so they can begin or prepare work for submission into system integration. When the current tag is changed, it should be communicated reliably by the system integrator. The majority of developers will only need exposure to the current product tag most of the time. However, we can also let developers take advantage of the system integration inprocess tag. The developers most likely to use this tag are those attempting to resolve change difficulties encountered during their attempted change integration. If a change was aborted, the development team may endeavor to correct this by getting the change to work correctly before this week's system integration window closes. In this case they will want to use the inprocess tag because their changes likely had problems originally because they collided with another change that went in that same week. This mechanism gives developers an opportunity to get a change back in without having to wait for next week's new tag.

## Build Qualification

As we might expect, the level of functional verification we do during each change's integration will not always be enough. We have specifically allocated a substantial amount of time in our

**Table 6–2.** System integrator view via tags of a week's changes

| Day of the week | Example tag | Tag contents |
|---|---|---|
| Mon. | ynix.current.11-7-97<br>ynix.current<br>ynix.inprocess | All identical starting on Monday. They all contain the current product source. |
| Tues. | ynix.inprocess<br>ynix.inprocess.11-14-97.rev1 | We have successfully integrated our first change of the week. The inprocess tag is updated and copied to the rev1 tag. Rev1 is the combination of "current" plus a change from Mary that was submitted 11-10-97. |
| Tues. | ynix.inprocess<br>ynix.inprocess.11-14-97.rev2 | Successful integration of second item. Rev2 is the combination of rev1 and a change from Joe that was submitted 11-10-97. |
| Wed. | ynix.inprocess<br>ynix .inprocess.11-14-97.rev3<br>. . . up to rev8 | On Wed. we integrated in 5 separate changes submitted by a number of different engineers during the week. |
| Thurs. | ynix.inprocess<br>ynix.inprocess.11-14-97.rev9 | No changes outstanding for today, so we submit the build for some additional testing. A problem is found and fixed in rev9 of the inprocess tag. |
| Fri. | ynix.current.11-14-97<br>ynix.current<br>ynix.inprocess<br>ynix.inprocess.11-14-97.rev9 | We are finished with the integration for this week, so on Friday all of these tags are identical. Note: we have a slight inefficiency with two identical historical tags. This is required because we never know which inprocess rev will be final until it is. |

system integration cycle to allow additional testing. Additional testing must occur after the final week's changes have been integrated on Thursday afternoon and must be completed with enough time left to address problems that arise from the testing.

This type of testing is called build qualification. The focus of this testing is to qualify the build for additional work by the developers. Developers require a build with requisite levels of stability and functionality in order to work on it. When a product's functionality is not stable enough, it becomes difficult to determine whether changes are making the stability problem worse. In addition, a product that continually crashes will slow down the development process dramatically. A crash is a general term to describe product that goes into a nonfunctional state.

Qualifying a build involves the execution of tests and the interpretation of results. Although this testing should be a portion of the larger testing effort that determines whether a product is improving toward shippable quality, it is also important to evaluate the product for

our ongoing system integration cycle. In order to accomplish this, an assessment must be made during the build qualification process. Is the build stability and functionality at a high enough level to proceed with development? And, is there any information regarding specific functionality or aspects of the product that should be brought to the attention of the developers?

A good deal of the changes going into the system integration process are expected to be defects detected during some level of product testing. The goal of each system integration cycle is to provide a new internal iteration of the product that gets closer to the functionality target of the next planned customer release. The build qualification process provides the lowest level where we can measure our progress toward this goal. Conceivably, we may need to take exceptions to our regular system integration cycle to address problems with the product.

### Identifying Product Problems

During build qualification and other testing, we may identify two types of product problems with regard to the system integration cycle. The first involves product defects that prevent any developers from using the product. A good example would be a defect that causes the product to crash randomly after ten minutes of use. It is relatively obvious that we will not want to proceed with product development until this problem is resolved. We make this judgment based on the amount of time potentially wasted by developers attempting to use the product in such a state and on the amount of additional damage that could be introduced since the developers would be extremely limited in how thoroughly changes could be tested in such a product. Generally, this type of problem should be easily accommodated by the regular system integration cycle.

The second type of problem cannot be accommodated by the system integration cycle. As we have seen with defect metrics, we will always want to have a reasonably small margin between our reported defects and closed defects. If the margin rapidly becomes larger and larger, we may have difficulty. One of the worst scenarios in product development is to have so many defects that you cannot fix any defects. This is analogous to getting knots in your string when you are flying a kite. The first knot is easy enough to untangle, so easy that you may put it off while you are having so much fun flying the kite. The second knot, unfortunately right on top of the first, can still be resolved in a fairly straightforward manner. You realize that untangling the two merged knots will take longer than it would had you untangled each individual knot as soon as it came along. If each additional knot increased the rate of getting new knots, your kite string would approximate our reported and closed defect margins. And, just as with the knots, if we have too many overlapping defects, we can quickly paralyze our development efforts by spending all our resources untangling our product from its defects.

### Resolving Product Problems

The avoidance of defect tangles are one of the reasons it is important to monitor this margin. If caught quickly enough, defects need not sink our product. We will observe this metric for trends. We may not need to take action when we see one dramatic weekly increase in the margin, but we may consider action if we see the margin width accelerating over eight weeks. The

action we take is simple. The defects are identified, and we know what margin we want. We simply restrict use of system integration to resolve only the defects that will reduce our margin to a manageable scale.

As we have seen, the build qualification process is an important part, not only of the system integration process, but of the product development process as a whole. The ongoing quality assessment of the product will affect all members of the development team. The qualification results should be communicated when the new current tag is announced to the developers. Other concerns and actions involving deviations from the regular system integration process should be communicated in the same manner.

## Coordination for Centralized System Integration

The organization and coordination of centralized system integration involves many aspects of the development organization. System integration occurs for all software development activities that contribute to a product. System integration is viewed as a centralized effort that services the modularization of software development. The organization and coordination of centralized system integration can be broken down into personnel allocation and integration protocols.

### Different Models of Personnel Allocation

As we have seen, actual system integration is no trivial activity—no more so than product development. Successful system integration can yield a valuable parallelism across the development process. Ultimately though, someone has to actually do the system integration. In this section we will discuss the benefits and costs of different approaches to personnel allocation for this activity.

System integration occurs whether we put a system in place to handle it or not. Eventually all code must be able to interoperate where needed or the product will not function. By default, system integration occurs as an uncoordinated activity of each and every developer that is contributing software to the product. When the developer needs to get her code into the product she takes the role of the system integrator and resolves all her product conflicts. Unfortunately, there is no guarantee that other developers are taking the identical role at the same time, so we can and do see circular conflicts. For example, Mary and Joe are trying to system integrate their changes during the same time frame. Mary sees conflicts with Joe's code, so she has to make changes to it and does. Joe brings in his own changes and sees that someone has already changed his code; he reworks his code and then sees conflicts with Mary's code. Joe then makes changes to Mary's code. Mary and Joe are locked in a cycle where they could be undoing each others changes forever. Additionally, without centralized system integration, it is unlikely that there are scheduled testing requirements that will allow Mary and Joe to know that they can reasonably expect that they have completed work on their changes. In this system, Mary and Joe could have to revisit the functional implications of their integrated changes on an ongoing basis.

This uncoordinated system integration approach works relatively well for a small number of developers, say, five. But when we start to deal with larger number of developers, weaknesses

appear. The first reasonable approach we should consider is a coordinated sharing of the system integration responsibility across the development organization. In coordinated sharing, the primary difference is that we only allow one engineer at a time to act as system integrator. Additionally, we have specific requirements that the system integrator take his change all the way through the system integration cycle.

Coordinated sharing helps alleviate some of the problems of uncoordinated system integration, such as circular change conflicts and functional verification, but it does not address other difficulties. Coordinated sharing is restricted to nonparallel system integration because the developer is only concerned with her changes and we cannot have multiple uncoordinated system integrators. This introduces a serialization to our process that can slow us down, as previously discussed. Although coordinated sharing will eventually expose many of the engineering developers to much of the system, it also places a much higher general overhead on all development resources. Suddenly, we don't need a developer that only has expertise in video driver writing; he will also need to have expertise in code merging and conflict resolution for all the technical areas of the product affected by video driver writing. There is also an overriding tendency for developers to dislike having to switch from developer mode to system integrator mode. The system integration role is often viewed as overly administrative.

This leads us to the most common mechanism for addressing system integration personnel resource allocation—the designated central integration engineer. This individual is specifically to perform the system integration activity. Our previous discussions regarding the system integration process, schedule, and parallelism all assume that we have at least this level of resource. For those who still like an organization where development engineers participate in this role, it can be structured as a responsibility rotated throughout the pool of development engineers. But it is not a part-time responsibility, it will have to be a full-time job with all the requisite level of training, expectations, and support. The rotated responsibility provides the benefit of exposing development engineers to the technical intricacies of managing the product as a whole, but the temporary nature of the rotated responsibility, the amount of overhead knowledge required, and the inconvenience of rotated roles usually result in a designated system integrator.

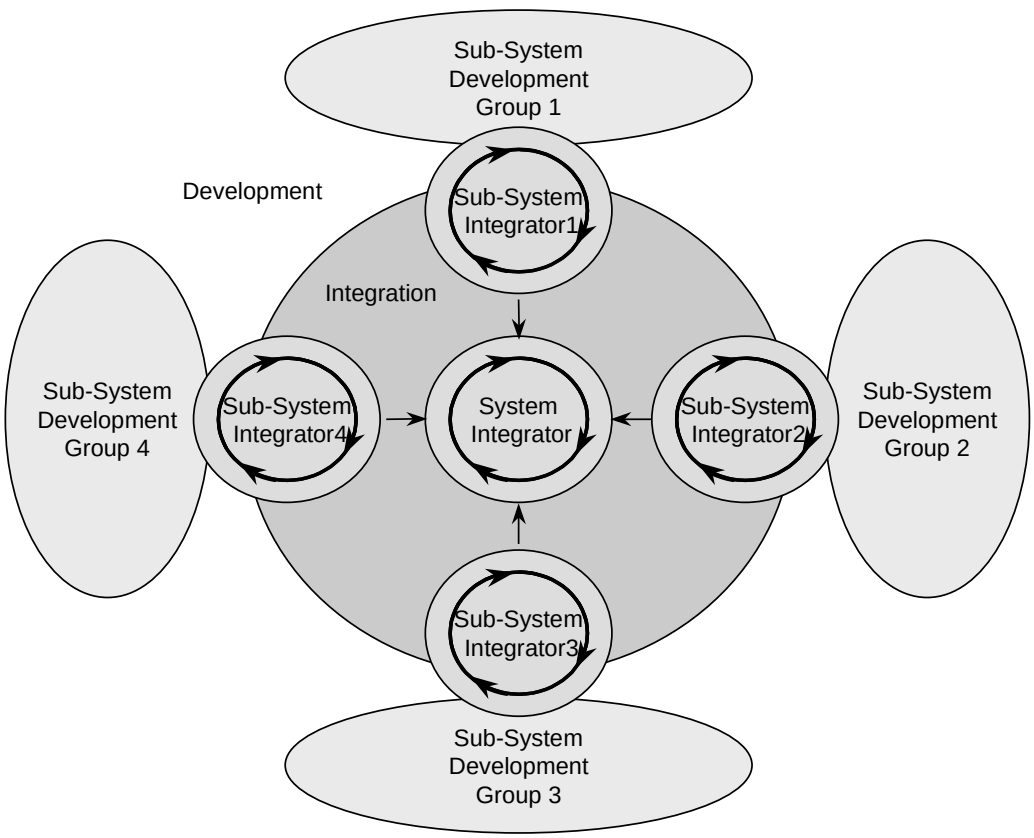
Depending on the capacity needed for system integration, more than a single resource may be necessary. Multiple resources will, however, need to have their activities well coordinated, but as we have seen with parallelism of the system integration process, multiple system integrators can double the module change capacity. There are other advantages to having more than one system integrator. The system integration resource becomes critical in the day-to-day operation of development engineering. Beyond considerations of additional system integration capacity, we must give careful consideration to what happens to our development process when a single system integrator is out sick, on vacation, or quits. At a minimum, when the development organization makes the decision to use the single designated integration model, it has a corresponding responsibility to plan how to handle situations when that resource is absent. Generally, halting the development process will not be sufficient.

### Further Subdivisions of System Integration

For even larger and/or more complex integration endeavors, we have a further level of abstraction. Subsystem integration provides a shell around system integration. In conventional system integration, our integration resource deals directly with individual changes coming from development engineers. In subsystem integration, we break the product into functional subsystems, each with a designated subsystem integration resource. The subsystem integrator is responsible for performing tasks identical to the system integrator but she only manages the changes coming in from her subsystem. Multiple subsystem integrators hand their groupings of subsystem changes to the system integrator (see Fig. 6–5).

The subsystem integrators work with their subsystem development groups in the role previously described for the conventional system integrators. They provide parallelism for subsystem change integration and some level of functional verification. Additionally, they can become

**Fig. 6–5.** Subsystem integration



more focused on the subsystem they are integrating for. In theory, a subsystem integrator can be more efficient when he is focused on a single subsystem, just as a developer can be more efficient in the area where she has expertise.

After the subsystem integrator has completed integration of the scheduled changes, they hand off her set of successful changes to the system integrator. The subsystem integrator can also provide an integration service that is more customized to the development group they are serving. The primary goal of subsystem integration must be to meet the schedule requirements of the system integrator, but the secondary goal can be to enhance the productivity of subsystem integration. For example, some subsystem development groups may not have a subsystem development cycle that is in sync with the system cycle. They may have a capacity that requires getting subsystem changes back to their subsystem development engineers sooner. In this case, the subsystem integrator can provide the source tag that represents their sets of successfully integrated changes back to the subsystem development more frequently than the schedule for the overall system integration. A centralized system integrator would never be able to handle the coordination complexity of offering a service like this across the entire development organization, but a specialized subsystem integration can.

Introducing subsystem integration can also increase the capacity of the system integrator. The subsystem integrators provide a sort of preintegration qualification before they see any changes. The subsystem integrators also provide a valuable service where they resolve any conflicts and errors that occur specifically in sets of changes in a subsystem. This allows the system integrator to focus specifically on the class of problems that arise in intrasystem changes. This type of subdivision is very similar to those in the field of software testing. In testing, it is often helpful to differentiate between subsystem and system testing.

Subsystem integration has some very attractive benefits but also very substantial costs. Not only does it have the obvious human resource cost but, as you can see from the description, it also makes description-specific requirements on the development organization. If there is no clean division of the development group into subsystems because of organizational or source base limitations, there is little likelihood of subsystem integration working. Subsystem integration should be used to address specific capacity issues within system integration.

### Integration Protocols

Regardless of what kind of system or subsystem integration model the development process uses, the issues of integration protocols must be addressed. Integration protocols define the communication and execution expectations across the development organization during code integration activities. We use the term “integration” to cover system and/or subsystem integration.

At the lowest level, a change occurring to the system can be communicated with a source code tag that represents the changed files that are to be integrated and an email message that identifies the tag and requests that it be integrated. A corresponding email response to indicate that the change has successfully gone into the product can close the loop on a simple protocol for communicating change. The communication protocol should also define how difficulties are handled.

The centralized nature of integration makes it a time-critical activity for the entire development organization. The development organization needs a deterministic process for getting problems that occur in integration resolved as soon as possible, so we will also need the protocol to define our expectations on the timeliness of problem action. There is no way to set a time requirement on how fast a build or functional verification error can be fixed. The only action that the development organization can attempt to control is how quickly and at what quantity resources are applied to resolve the problem. A problem that stops this cycle can eventually stop the entire development process, so its priority is high. Each development organization will need to address its ability to communicate and mobilize resources to address a high-priority problem effectively.

Prioritization depends on the corporate cultural aspects of the development organization. Corporate cultural aspects involve the level of urgency the organization can handle and how efficiently the urgency can be communicated. A company without reliable means to communicate an urgent problem to the development organization can expect to have a great deal of difficulty addressing high-priority problems quickly. Or, if a company's corporate cultural attitude does not allow the concept of urgency, we will have similar problems.

Assuming that an organization recognizes the importance of getting system integration problems resolved before they start to impact the development process as a whole, two extremes represent the range of approaches to the problem.

In an unstructured approach, the system integrator uses a sophisticated process to resolve problems. First, she attempts to analyze and resolve the problem independently. If this is not possible, the integrator will know what other personnel in the company would be most likely to assist with the problem's resolution. Because development organizations rarely have individuals that are expert enough to deal with sophisticated integration problems and do nothing but that, the integrator will also know which individuals would be the most appropriate to interrupt for assistance. She will also know of a variety of appropriate ways to contact these individuals.

The unstructured approach is often the most efficient and least offensive to development organizations. The organization as a whole understands the need to get integration problems resolved and is more tolerant of an approach that minimizes the number of distractions it causes. Unfortunately, this approach requires our integrator to be completely familiar with the vast majority of the development organization staff, what projects each engineer is engaged with at any time, and the cultural particulars of how to contact individuals for assistance. Although this approach has many advantages, it will always be difficult to both train and retain individuals that have the requisite skills.

The fully structured approach introduces a documented-problem escalation protocol that can be used to contact and acquire resources for integration problem resolution. Although this is much less personal than the unstructured approach, it has the advantage of not requiring the extensive general knowledge we expect from our integration engineer in the unstructured approach. It also allows the organization to specify exactly how it wants integration problems escalated within the organization. The protocol has two parts: The first is some sort of representation of what technical expertise lies in each of the organization's engineering groups and

**Table 6–3.** Sample contact table

| Integration problem | Functional area | Contact individual(s) and their groups |
|---|---|---|
| Compiler, linker, or other development tools | Tools | Tools manager: Tom |
| Tests and testing tools | Test | Test manager: Bill |
| Functional product | User interface<br>File control<br>Text control engine | UI manager: Xioping<br>FS manager: Jill<br>Text manager: Stacey<br>Editing: John<br>Formatting: Kim<br>Language: Kern |

which individuals should be tapped for it. The integrator will be directed first to contact the engineer or engineers responsible for the change that caused the problem to occur. After the submitting engineer has confirmed that the problem is not in his change but elsewhere in the system, a contact table (see Table 6–3) can be consulted.

In addition to the information in Table 6–3, a fully functional table would also list the means by which each of the individuals could be contacted, including information such as email address, office phone number, company pager or cell phone number, and possibly even home phone numbers, depending on the specific corporate culture.

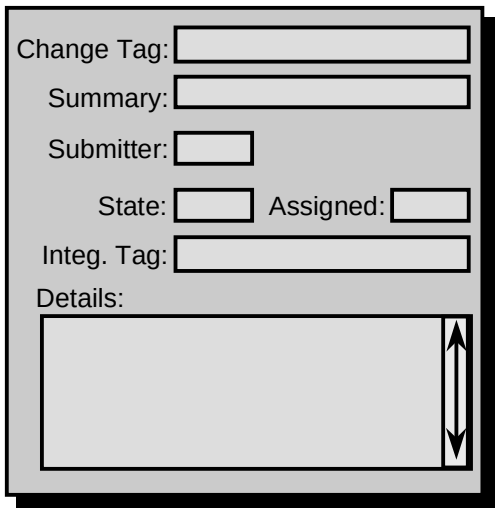
Some organizations will need additional protocols that identify what should be done if the specified contact cannot be made. This is called an escalation protocol and its use is less cut and dried. The course of action for an escalated problem varies according to the nature and impact of the problem. The problem is specifically escalated to get a ruling on whether attempts at contact should continue or whether the problem warrants a different course of action. The simplest escalation protocol would have the integration engineer first escalate the problem to his manager, who would in turn escalate the problem to whomever manages the contact that cannot be reached. Similar escalation would propagate up the reporting chain.

## Using Computer Applications as Change Control Systems

Computer applications can provide assistance in managing the level of complexity inherent in our change control process. Although the integration resource helps offload some of the more time intensive activities associated with a large product integration effort, it also adds several levels of complexities for all involved. Just as we used a computer application, the defect management system, to help assist in tracking and resolving product defects, we can use a change control system to help manage the integration process.

In its simplest form the change control system is used to communicate and track the status of each change that has been submitted for integration (see Fig. 6–6). The change control system is used instead of the previously mentioned email system as the entry mechanism into integration.

**Fig. 6–6.** Simple change control record

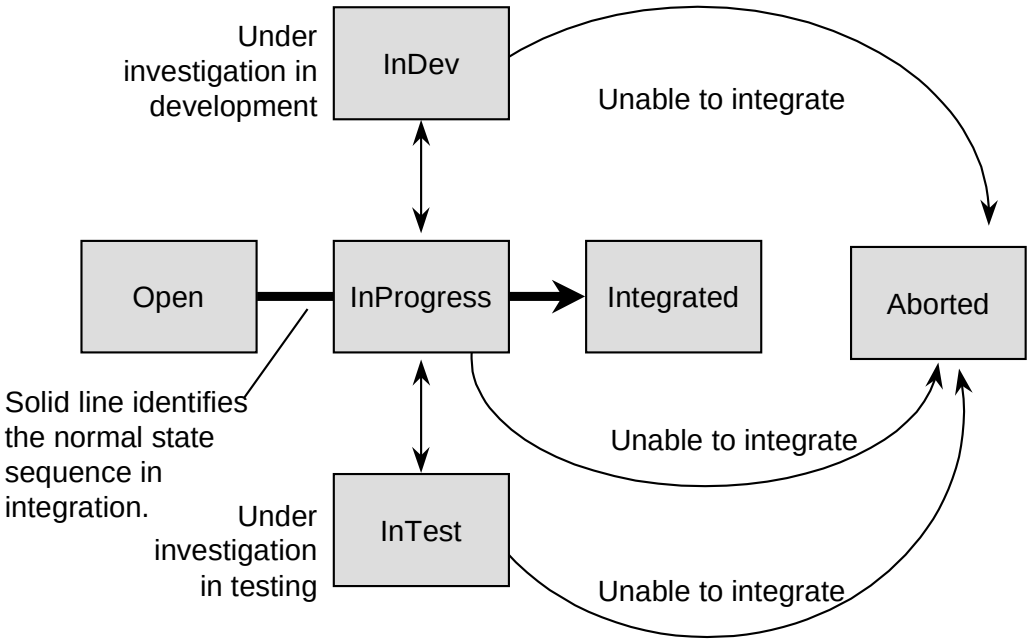



Just as we track the status of a defect in the defect management system, we will have a corresponding status of the change. Each change record in the change control system will key off the source tag that was used to capture the set of files that represents the change. We will impose a requirement on our developers that their tags be unique. The change submitter will also enter into the change record information such as her name, the name of the tag, a summary of what the change is, and, if needed, specific details about the change that may be needed by the integrator.

The change record (see Fig. 6–7) will also support two additional fields to be used to track the change as it makes its way through the integration process. The first of these fields is the state field, which is similar to the state field in a defect-tracking system. We also have a field that identifies who the change is currently assigned to. This field will be filled out automatically when the defect is first submitted for integration, and if all goes well it will never change. However, if problems occur and development or testing takes the change for further investigation and/or defect resolution, the assignment field will be used to identify the temporary owner of the change. This makes it much easier to access individuals who are holding outstanding changes. We also have an integration tag field that is used by the integrator when the change is added to an integration tag. As we have seen in our integration discussions, there can be several levels of integration tags. Each tag the change has been added to will be listed by the integrator in this field.

When the defect is first submitted, it is assigned to an integrator responsible for reviewing open defects on a very regular basis. As the integrator goes through the integration process, the state of the defect is "In Progress." If during the process of resolving difficulties with a change the control of that submission needs to change, the state will reflect this. In our state example, there are only two phases that the change can be diverted to, development or test engineering.

**Fig. 6–7.** State transitions for the change during the integration process




As with the "Inprogress" state, the "Indev" and "Intest" states are temporary states. As we have seen in the integration process, one change will not hold up the system integration process. If a change cannot be integrated in time for the creation of the next current tag for the product, its integration cycle may need to be restarted. And since the source base that the change was originally intended for has changed, it will become more and more unlikely that the change will be possible to integrate.

In addition to giving the integration engineers a useful mechanism for managing change, the change control system also provides the development engineer with a convenient tool to track her change as it is integrated. Additionally, a change control system can be used to gather change control metrics similar to defect metrics. Measurements of how long it takes changes to get through the system, how often a change submission needs to be aborted, and how much change is going into the system are useful metrics to have for the product.