Daily Build
– the Best of Both Worlds: Rapid Development and Control

Daily Build - Rapid Development and Control.
Swedish Engineering Industries 1999
Sveriges Verkstadsindustrier

Preface

The study of *Daily Build* was undertaken at the request of Swedish Engineering Industries. The content of the study has been formulated by Swedish Engineering Industries' "Teknikråd 12" and the steering commitee for the study.

The study was performed from April 1999 until December 1999 by Kent Olsson and Even-André Karlsson. The study is based on interviews with persons in the Swedish software industry, literature, and the experience of the authors.

The steering commitee has defined the content of the report, reviewed the report and also approved the final report.

The authors would like to thank the persons in the steering commitee for the support when writing this report. A special thanks is also extended to all the companies that contributed to this report.

Members of the steering commitee:

Lars-Göran Andersson	Ericsson Radio System	(chairman)
Mikael Rudin	ABB Automation Products	
Mats Ljungberg	Microsoft	
Olle Landström	IAR Systems	
Jan Christian Herlitz	Excosoft	
Mats Ekman	SAAB Military Aircraft	
Kent Boortz	Ericsson Utvecklings AB	
Bengt Asker	Konsult	
Göran Östlund	Swedish Engineering Industries	

Authors:
Kent Olsson, Main author
Even-André Karlsson, co-authour
Q-Labs AB
IDEON Research Park
S-223 70 Lund
Sweden
http://www.q-labs.com

1 Introduction 1
1.1. The Purpose of the Study 1
1.2. Intended Readership 2
1.3. Report Structure 2
2 Conclusions and Recommendations 4
2.1. Conclusions 4
2.2. Recommendations 4
3 Daily Build; a Brief Overview 6
3.1. A Brief Overview 6
3.2. Daily Build and Feature Team 8
3.3. The Benefits of Daily Build 9
3.4. Common Risks Using Daily Build 11
4 Daily Build; a Fundamental Description 13
4.1. The Organisation of Responsibility in Design 13
4.2. Build 16
4.3. Testing Strategy 21
4.4. Daily Build in Smaller Projects 24
4.5. System Level for Daily Build 25
4.6. Projects Suitable for Daily Build 26
4.7. Tailoring Daily Build 27
5 Influence of Daily Build on the Software Environment 29
5.1. Influence of Daily Build on the Development Process 30
5.2. Influence of Daily Build on Design Documentation 33
5.3. Effect of Daily Build on Architecture 34
5.4. Requirements Management 36
5.5. Software Project Planning 36
5.6. Software Project Tracking and Oversight 40
5.7. Software Sub-contract Management 40
5.8. Software Configuration Management 41
5.9. Peer Reviews 41
6 Guidelines for Implementation of Daily Build 42
6.1. Self Assessment 42
6.2. Practical Guidelines for Implementation 43
6.3. Measurements for Daily Build 44
6.4. Success Factors 45
7 Experience from the Swedish Software Industry 48
7.1. Ericsson UAB; the OTP Group 49
7.2. Excosoft 53
7.3. IAR Systems 57
7.4. ABB Automation Products 62
7.5. Telelogic 68
7.6. SAAB Military Aerospace 73
7.7. Ericsson: GPRS Development for the BSC 80
7.8. Ericsson: OSS Development in Mölndal 82

7.9. Daily Build in Ericsson 86
8 References 87

1 Introduction

1.1. The Purpose of the Study

In 1995, the book *Microsoft Secrets* [1] was published. This book included, among other things, a description of how Microsoft uses daily build. In 1996, another book called *Rapid Development* [2] was published, also describing daily build. Two years later, the Swedish software industry started to pay attention to daily build, and at the end of 1998, daily build was often mentioned as a method used by Swedish software companies.

At this point, it was shown that many smaller Swedish software companies had been working with daily build for a long time. However, these organizations did not set out to use daily build as such; internal improvements ultimately led to a daily build approach. Around 1998, larger software companies in Sweden also started to use daily build. However, the different companies had applied daily build to their software development in different ways and each company had its definition of daily build. This created problems when persons from different companies wanted to discuss daily build, and organizations did not agree about a standard definition of daily build and its components.

The purposes of this study are as follows:

- provide one common definition of daily build, based on the different definitions that exist in the industry today
- spread the daily build concept in Swedish industry
- meet the demand for literature describing daily build
- investigate how daily build is used in Swedish industry today

1.2. Intended Readership

This report is written for persons in organizations considering implementing daily build, as well as those in the software industry who wish to learn more about daily build. On a more detailed level, the report is targeted to persons responsible for implementing daily build in an organization. The focus in the report is the process used for daily build and how daily build impacts the development process and the organization. Technical aspects of daily build, such as scripts for performing automatic builds or automatic testing, are not covered in this report.

This report is not a 'cookbook' for implementing daily build. It highlights and discusses different ways to tailor daily build and should be used by persons responsible for implementing daily build in their organization to inspire, give ideas and overcome hurdles.

1.3. Report Structure

The report is based on literature, experience of the authors, and interviews with Swedish software companies with experience from daily build.

The report contents and structure were determined together by a steering committee with representatives from various Swedish software companies.

The report is divided into three main parts and eight chapters. The first part, Chapters 1- 2, is an introduction to the report and a summary of recommendations and conclusions. The second part, Chapters 3- 6, is a theoretical description of daily build. The third part, Chapter 7, presents experience from the industry using daily build.

The report is organized as follows:

- Chapter 1. Introduction
- Chapter 2. Compiles conclusions and recommendations.
- Chapter 3. Contains a brief overview of daily build, to introduce the reader to the concept. This chapter also highlights the benefits and risks with daily build.
- Chapter 4. Contains a more detailed description of daily build.
- Chapter 5. Highlights how daily build impacts the software environment, in areas such as:
    - the development process
    - the documentation
    - the architecture
    - Requirements Management
    - Software Project Planning

- Software Project Planning and oversight
- Software Subcontract Management
- Software Configuration Management
- Peer Reviews

- Chapter 6. Contains guidelines for how to implement daily build in an organization.
- Chapter 7. Contains experience from software companies using daily build or planning to implement daily build.
- Chapter 8. References.

2 Conclusions and Recommendations

2.1. Conclusions

This study has proven that daily build is a widespread method used by several different software companies in Sweden. The study has also proven that each company uses daily build in its own way, and that each organization has its own definition. However, the different companies all created their daily build based on the same major corner-stones, namely:

- iterative development (build the system at regular intervals)
- automatic testing
- responsibility based on features

This study has also proven that daily build is a very popular method at the companies using it. The companies that are covered by this study are all very satisfied with the results and the gain they received by introducing daily build.

2.2. Recommendations

Based on the positive response given by software companies using daily build, it is recommended that those software companies experiencing software development problems such as excessive lead times, low quality, and poor project control investigate how daily build can improve their software development.

It is also recommendable that companies currently managing their software development successfully investigate how their projects will look in the long term. A lot of companies

have problems controlling projects that increase in size. One solution to retain the control over larger projects is daily build.

It is very important that the company considering implementing daily build investigates how the organization should apply daily build to attain maximum benefits. As mentioned before each company uses daily build in its own way, and daily build has to be tailored for each company's individual needs. The use of daily build can always be improved and therefore it is important to have a person in the company responsible for continuously improving the application of daily build. It will take some time before daily build works in a satisfactory manner.

An organization that wants to take full advantage of daily build needs to ensure that the following things are in place or can be arranged:

- Organize work so designers can deliver executable code to the system on a daily basis. This can be done either by a detailed construction plan or by organizing design in feature teams [see Chapter 3.2].
- A configuration management system that can keep track of different versions of the code
- An automatic build process
- Automatic test on the product

Note that the first item is more an organizational change, and the remaining three items are technical infrastructure improvements. It is thus advisable to divide the implementation of daily build into two separate parts: one handling the technical infrastructure, the other handling the organizational changes. These two parts can be implemented in parallel, normally with the implementation of the technical infrastructure one step ahead of the organizational changes. This is beneficial since the technical infrastructure is useful independent of how the design is organized; the design organization can assume that the technical infrastructure is in place, and discuss the optimal way to organize design under these conditions [see Chapter 6].

3 Daily Build; a Brief Overview

This chapter provides a short summary of daily build to introduce the concept to readers with no previous experience of daily build, so this audience can fully profit from the rest of the report. This chapter also highlights benefits and potential risks with daily build.

A more in-depth description of daily build is given in Chapter 4.

3.1. A Brief Overview

The principle behind daily build is simple: **build and test the product every day**. This means working in parallel teams/designers that ‘synch up’ and verifies the system at the end of the day. To meet this goal three tasks must be completed.

1. Deliver code to the build every day. Building the system every day is worthwhile only if new code is added to the system every day. It is useless to build the exact same system two days in a row. This does not mean that every designer or every team has to deliver new code to the system every day, but at least one team or designer should deliver so the system receives new code each day; for example, if the project comprises five teams, every team has to deliver code at least once a week.

The code delivered to the system must be synchronized between designers so the functionality in the system increases with the number of deliveries - in other words, designers must break down the functionality into smaller increments that are delivered to the system. Each code delivery must not add any new functionality to the system. Even if the new code does not add any new functionality until other code parts are delivered, it is still important to deliver the code on a daily basis, since this will maintain project momentum and will provide evidence of real progress in the project.

Designers should try to get the code being delivered to the system to be consistent from a system perspective as early as possible. Consistent code means that all impacts for a specific part of an end user functionality are in place. If two modules interact to perform a task, the implementations of this interaction must be checked in at the same time. Take as an example a functionality that needs a new parameter in a function (signal). Both the caller and the called module have to implement the additional parameter at the same time.

Ways to organize the design so code can be delivered on a daily basis are presented in Chapter 4.1.

2. Build the system on a daily basis. When design has delivered the new code, the system has to be built. Most companies have an 'integration group' (also called a 'build group') that is responsible for this task.

The build process contains two major steps:

- Merge the code to the main code line
- Compile and link

The first step is to get in all the new code from the designers to the main system and make sure no code conflicts have arisen between different designers' new code. Then the integration group has to compile the total system (build the system). If compiling is successful the result is the new daily build. Building the whole system must be done in less than 3-4 hours, since the final part - automatic testing - should also be done before the next morning (most builds and tests are run during the night).

The system build has to be automatic. Otherwise, it will require too much effort to build the system every day.

Organizing the building of the system is discussed in Chapter 4.2.

3. Perform automatic testing of the system. Automatic testing is the third part of daily build, as it is of little use to build a system on a daily basis if a certain level of quality cannot be ensured. The idea of the test is to establish that none of the existing functionality in the system is broken; i.e., a regression test is performed. This test has to be automatic as the amount of effort needed to manually do the tests every day is prohibitive.

Most companies using daily build have a test group that handles the automatic testing. When failures are found, the test group will contact the responsible designer.

3.2. Daily Build and Feature Team

Another concept discussed in connection with daily build in this report is *Feature Team,* where the definition of feature is as follows:

A feature is an end user functionality, which might involve several modules in the system.

There are two ways to organize the design teams, module team or feature team.

- *Module team* is the traditional way with one team or individual responsible for one or several modules. The team or individual implements impacts from all features in these modules.
- *Feature team* means that the responsibility is based on end user functionality, where one team or individual is responsible for one feature and implements this feature in all modules that are impacted by this feature.

These two ways of organizing design are not mutually exclusive and can be combined.

A mutual dependency exists between daily build and feature team:

- feature responsibility is a prerequisite (at least for bigger projects) for taking advantage of daily build since the amount of coordination and planning needed between module responsibles to deliver consistent code is unreasonably high. Every module designer has potentially several features impacting his module, where each feature is also impacting many other modules. In a feature team, these coordinations are all handled within the team.
- daily build is a prerequisite as a control mechanism to handle feature teams working in parallel, to synchronize the teams and to ensure that delivered code has the appropriate quality. Additionally, daily build ensures that different feature teams do not work too long in isolation, making the merging of their features at the end more difficult.

Regardless of whether daily build is a prerequisite for feature team or the other way around, these two techniques should be considered together. More information about feature teams is provided in Chapter 4.1.

3.3. The Benefits of Daily Build

This chapter presents the most common benefits experienced by companies using daily build:

- **Better project and quality control**
  - Real progress is very clear - not only document status but real executable code can be seen. If a team has not delivered any code for a long time, this is an indication of a problem and should be investigated.
  - There is always an executable system; even if the current build fails it is possible to go back to the last stable one - each day there is a stable product.

- "Big bang" integration problems at the end of the project are avoided - the system is built together every day.

- **Improved fault handling**
  - Faults are detected early after they are introduced. This does not only pertain to local faults but also global system faults and interference with old functionality. These types of faults are usually discovered later when daily build is not used.
  - It is easy to isolate faults, since in most cases these come from the new code in the current build, and the teams delivering this code should be able to fix the faults.
- **Focusing on end user requirements and code - Code Is King**
  - Facilitates customer involvement - the customer can follow the progress and verify his requirements on the real product.
- **Shorter leadtime**
  - Higher pulse in the project.
  - Less design documentation
  - System test can proceed in parallel with development, thus reducing the leadtime for system test at the end.
- **Increased understanding of the usability of the system**
  - The organisation has to "eat its own dog food", meaning it must load its software product into the hardware system. This gives an increased understanding of the usability of the system.
  - An indirect advantage is the infrastructure, automatic builds and automatic testing, built up by the organisation using daily build. This infrastructure is very useful for fault handling and maintenance of the products.
- **Better working situation for the designers**
  - Increased motivation for the designers when they can see progress every day and get direct feedback on their work
  - Decrease the pressure and stress on the designer. Normally the designer has a very stressful working situation at the end of a project. With daily build the

stress is distributed to a more reasonable level through the whole project [see Figure 1]

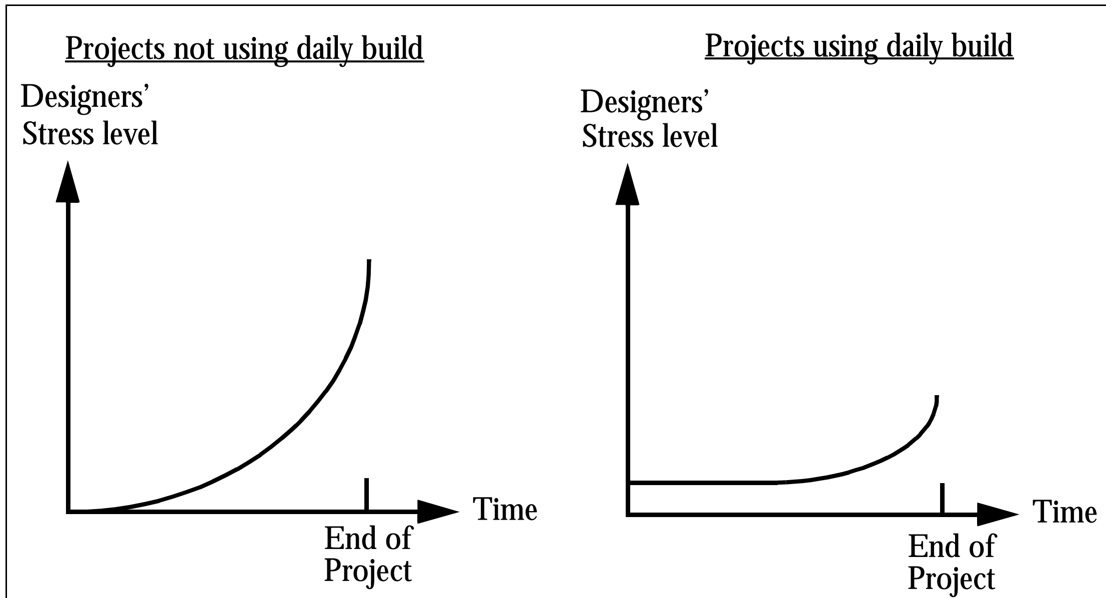


Figure 1. Designers' Stress Level in a Project

Benefits with feature team:

- The feature team has a focus on end user functionality and code. The team has a system view instead of a module view.
- Changes in requirements and plans are usually isolated to one feature and can be handled more easily by the feature team.
- Coordination towards associated projects is usually easier, because new interfaces usually come from new end user features and can be handled by feature teams on both sides.
- Interface updates between different modules in the system will mainly come from new customer features and can take place inside the feature team.

These are just a few of the benefits companies have experienced. More advantages are presented in Chapter 7.

3.4. Common Risks Using Daily Build

During this study, we have identified a number of potential risks. However, we have neither experienced nor met any company that encountered any serious failure because of daily build.

We have identified the following potential risks:

- **Risk:** Too frequent releases. There is a risk that the customer requires more frequent releases since there is always an executable system. More frequent releases will mean that the designers will spend time on documentation and special solutions that are not needed for the final product. Too frequent releases can also lead to releases with too low quality. The lower quality will result in designers spending time answering questions and handling problems from the customer.
  - **Recommended Solution:** We recommend companies using daily build to keep to the release plans - both regarding number of releases and dates for the releases. If the company has to release the product more frequently, include only those features that meet minimum quality requirements.
- **Risk:** Daily build can cause development to be excessively pro-active. It is important to find the right speed in the project and people still need to think before they act.
  - **Recommended Solution:** We recommend using peer reviews [see Chapter 5.9] to handle this risk. Introducing peer reviews will require people to stop, think through the implementation, and adjust the speed in the project.
- **Risk:** the architecture degenerates and the product becomes more complex.
  - **Recommended Solution:** Introduce *architecture police* and *module responsible* [see Chapter 4.1], to keep the focus on the architecture [Also see Chapter 5.3]
- **Risk:** "Quick and dirty" changes instead of a more accurate fix. This is a risk only if the company uses opportunistic handling. Opportunistic handling means that more than one team works with the same module in parallel [see Chapter 4.1]. Opportunistic handling implies a longer time slot between code check in, meaning more work for the project, since more merging problems must be resolved. To avoid this extra work load, a designer might choose a quick and dirty change instead of a more accurate remodelling that takes more time.
  - **Recommended Solution:** This risk can be eliminated if the designers team up to break down fixes in smaller steps. Then they can choose to perform the more accurate remodelling while still checking in the code at regular intervals.
- **Risk:** Too little design documentation. If the organisation has too much design documentation, daily build can help to decrease the documentation [see Chapter 5.2]. On the other hand, if the organisation already has unsatisfactory design documentation there is a risk that daily build will make the situation worse. Daily build leads to faster feedback from the tester to the designers, and this in combination with feature teams will increase the communication between different designers and testers. This means that some problems will be solved verbally in the corridor and that the design documentation will not be updated.
  - **Recommended Solution:** The increased communication between different feature teams, designers and testers is good. This communication will help

feature teams learn more about the whole system and other features. To ensure that documentation remains current and that the documentation is produced, we recommend making a person responsible for the documentation as well as regular peer reviews that include the documentation.

4 Daily Build; a Fundamental Description

This chapter presents a more fundamental description of daily build and the environment required to succeed with it. In Chapters 4.1 and 4.2 we assume that daily build is used in a larger project (more then 20 designers). In Chapter 4.4 we discuss how a simplified application of daily build can be used in a smaller project.

4.1. The Organisation of Responsibility in Design

To manage delivering consistent code from design in bigger projects (more then 20 designers) without unreasonable communication, we recommend that feature teams are used [see Chapter 3.2]. Feature responsibility can also be used in smaller projects but it is not a prerequisite for smaller projects.

This chapter describes the organisation of responsibility in the design phase when using feature team (with design we also include implementation). We recommend that the feature teams are responsible for the feature throughout the project, from requirement specification to testing; but in this chapter we discuss only the design phase. The responsibility of the feature team must extend to the system level on which daily build is used [see Chapter 4.5].

Using feature teams means that the feature team will implement changes in all modules which are necessary to get the end user feature to work, possibly in parallel with other teams implementing impacts from other features in some of the same modules.

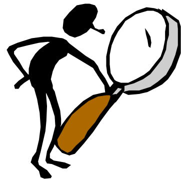
When implementing feature team responsibility there are generally two approaches to handle several feature teams working on the same module: opportunistic and planned. Opportunistic handling means that anyone can check out a module and work on it in parallel with anyone else. The first to check in a change does not have to do any merging, whereas all those checking in later must merge with all who have worked in parallel and checked in before [see Chapter 4.2]. The opportunistic handling is a big incentive for making small changes to the modules and checking them in frequently, thus serving as the engine of daily build.

Planned handling means that each feature team has a time window when it works alone with the module. This time window can be planned centrally or decentralized by the involved feature teams and a module responsible.

Using feature team responsibility does not exclude the idea of module responsibility. This role is becoming increasingly important in a feature team organization. The role is changing from someone who does all the work in a module, to someone who is guiding, checking and approving solutions and implementations proposed by each feature team. Of course the module responsible should participate in the feature team which has the biggest or most complex impact on the module.

Another important responsibility in the design organisation is the ‘architecture police’. The ‘architecture police’ is responsible for keeping a close check on the architecture. This person has to make sure that the designers stick to the architecture and that the architecture does not degenerate.

Figure 2 shows the different roles of responsibility in design.



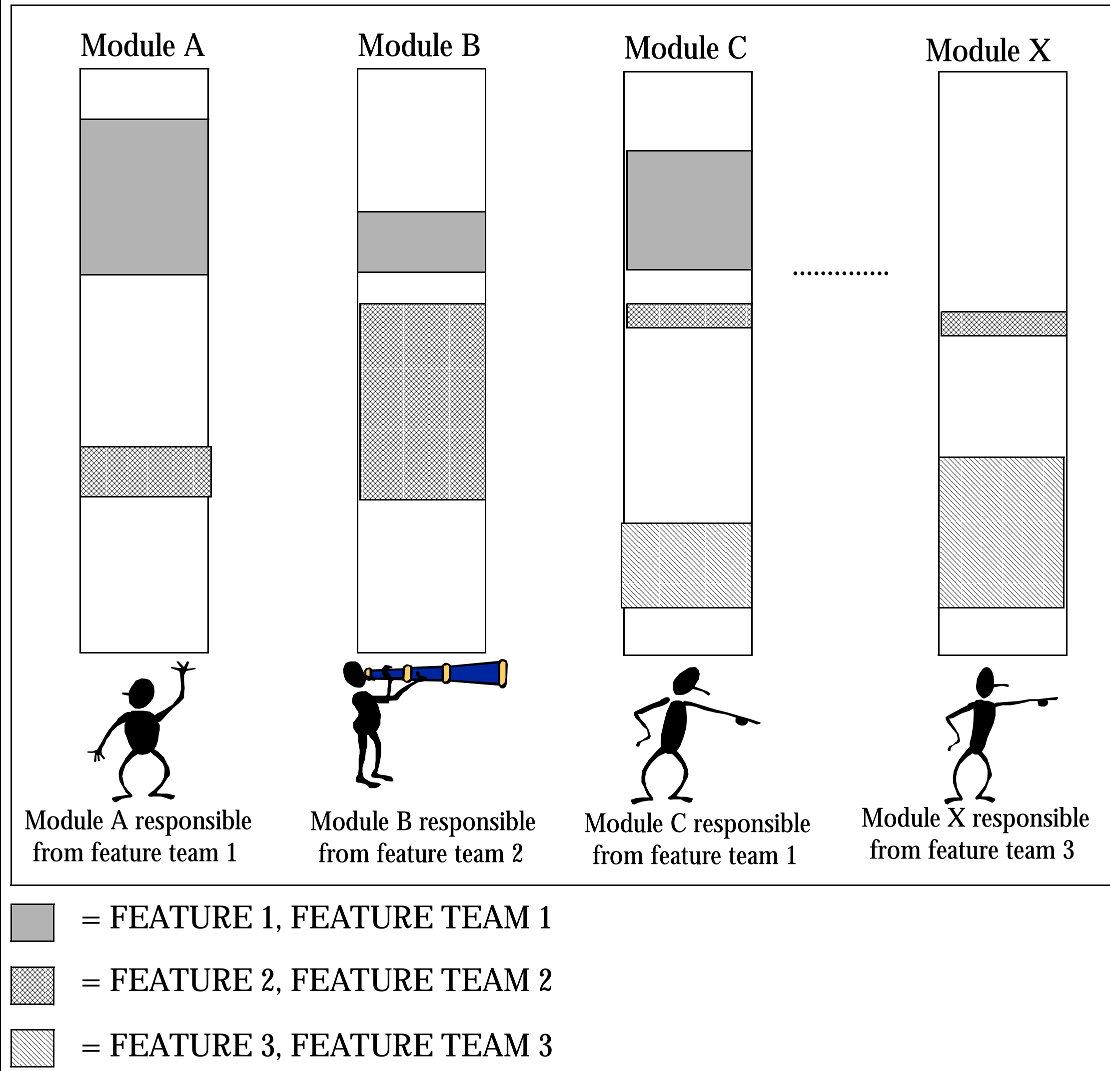


Figure 2. The Organisation of Responsibility in Design

Those parts of the modules that are basic functionality used by many different features are implemented by the feature team in which the module responsible is working.

Two other groups in the daily build organisation are the integration group [see Chapter 4.2] and the test group [see Chapter 4.3].

Using feature teams will imply that designers will modify code written by other designers. This leads to two problems:

1. The code has to be understandable for other designers. It is therefore important that the code is written in such a way that it is easy to understand and that the code is well commented. Introducing a code standard is one way to overcome this problem. It is also important to create a culture in the company so designers try to write understandable code and keep the code on a high level.
2. Designers dislike when other designers modify their code. Designers often think of their code as work of art and do not want anybody else to touch it. This problem is more serious than one would first think. To overcome this problem, the company has to create a culture allowing designers to modify others' code. The designers have to understand that they are a part of a team, all working towards the same goal, and that the common result is the work of art. Also, refocusing the designers' attention from code modules to customer features will help.

Another way to reduce the impact from these two problems is to actively swap assignments. A designer that has been working with feature A, when feature A is ready, can change and start working with feature C in feature team C. In this way, the knowledge about the whole system increases in the different teams. One company that uses this idea is Microsoft, where the designers have to look for new assignments in the organisation when they are finished with their task. Of course, the swapping between teams must be within reasonable bounds.

4.2. Build

The second part of daily build, to build the system on a daily basis, can be divided into two main steps:

- Merge the code to the main code line [see Chapter 4.2.1]
- Compile and link the system [see Chapter 4.2.2]

4.2.1. The Daily Build Process in a Feature Responsibility Organisation

The daily build process describes how the designers merge the code to the main code line. This chapter presents two different daily build processes that can be used in projects working with feature responsibility. The first process is used by an organisation applying opportunistic handling. The reason for the many steps in the process is that many different people can make changes to the same modules in parallel. The second process can be used in an organisation using planned handling. These two processes are just two examples - a lot of different daily build processes exist. The process has to be tailored to suit each organisation.

The opportunistic process (this process is similar to the process used at Microsoft, see [1]:

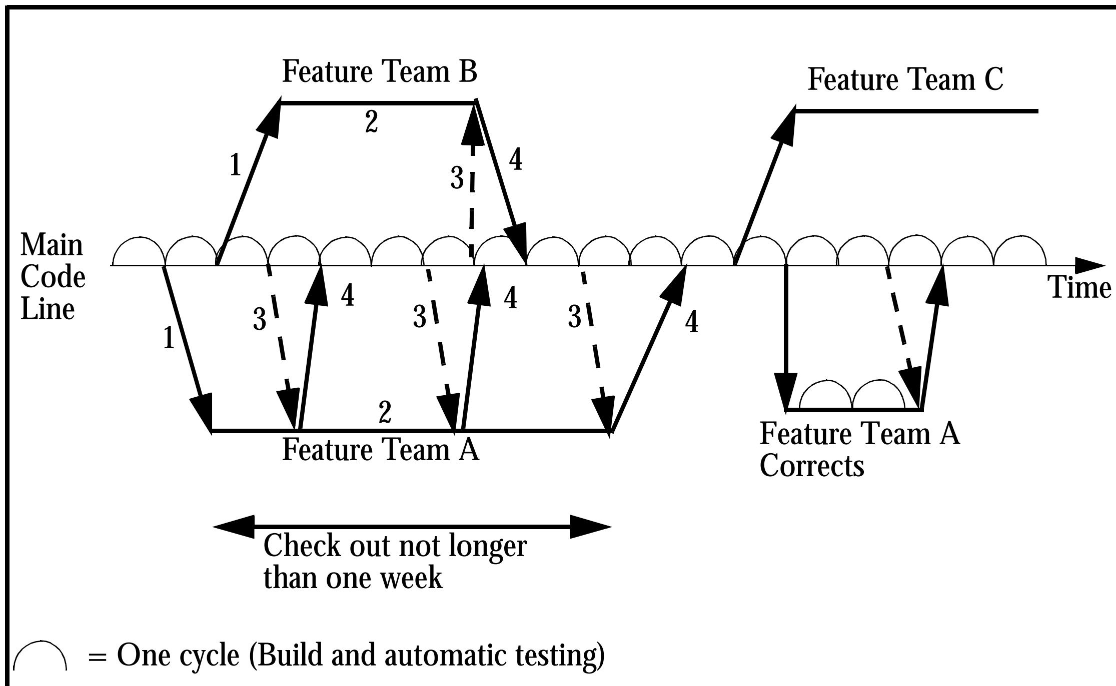


Figure 3. Example of a Daily Build Process 1

1. Check out private copies of the source code from the main code line. The main code line is a centralized master version of the source code, always containing the last successful build. Only the team that checked out these private copies will modify them. Other teams have to check out separate private copies from the main code line. The feature team will only check out one copy of the source files which then are used together in the feature team, i.e. each designer in the feature team does not have his own copy.
2. Implement feature, build and test private release.
   - The feature team implements the feature in the private copy of the source code file.
   - The feature team builds a private version of the system, using the copy of the source code file that includes the new implemented feature.
   - Test the private release of the product to make sure that the newly implemented feature works correctly [also see Chapter 4.3].
3. Check out copy of last successful build of the main code line
   - a: Synchronize code changes. Compare the private copies of the source code that includes the new feature with the current main code line. The current version of the main code line could have changed since the team checked out their private copies in step 1, since other teams implementing other features

could have checked in updated copies of the same source code files that this feature team has checked out. The team should do this comparison in the late afternoon, so they can be sure to use the most recent version of the main code line, since no one is allowed to check in any new code after the daily check in deadline [see Chapter 4.2.2]. The team has to have a tool that automatically determines the differences between the files.

- b: Merge Code changes. Update the private copies of the source code so they include the new feature implemented by this feature team and other changes made by other feature teams. It is recommendable to have a tool that handles this merging automatically and warns the team of any merge conflicts (inconsistencies between the files) that must be solved manually.
- c: Test the private release. Ensure that the team's newly implemented feature still works correctly. If required, the feature team can also perform some sort of 'smoke test' to ensure that basic functionality still works correctly and that the new feature does not interfere with some other features [also see Chapter 4.3.1].

4. Check in private copy before check in deadline. If step 3c was successful the team can check in their private copies to the main code line. The first step when doing this is to synchronize the private copies with the main code line version again (if any new changes are implemented by any other team), see step 3a. Then the merge conflicts must be handled, see step 3b. If there should be any merge conflicts they can not solve, the team has to withdraw those files and resolve the incompatibility in the changes.
5. Now the integration group start to compile and link the whole system [see Chapter 4.2.2]

We recommend that no check out is longer then one week. If implementing the feature takes more then a week, it is recommendable to check in parts of the feature at least on a weekly basis.

If a failure is found later that depends on feature A, feature team A goes through the process again correcting the failure [see Figure 3].

This process is also useful for organisations that are not working in teams, i.e. a single designer is responsible for a feature. The process is applied in the same way.

The Planned process. This process describes a situation where daily build is tailored to be used on the feature branches instead of the main code line. This is not a requirement for projects using planned handling. Using daily build on the feature branch makes it possible for the feature team to follow up results each day, but the total system is not followed up daily, since the main code line is built only on demand (when a new feature is checked in). This means some of the advantages of daily build are lost - the different feature teams will synchronize on a demand basis instead of a daily basis.

It is advisable to use daily build at the main code line and not on the feature branches. The reason this example of daily build on the feature branches is presented (despite that it is not recommended) is to show how differently daily build is applied in the industry. For more examples of how the daily build process is handled in the industry, see Chapter 7.

The process for planned handling does not include as many steps as the opportunistic process since there are no merging problems.

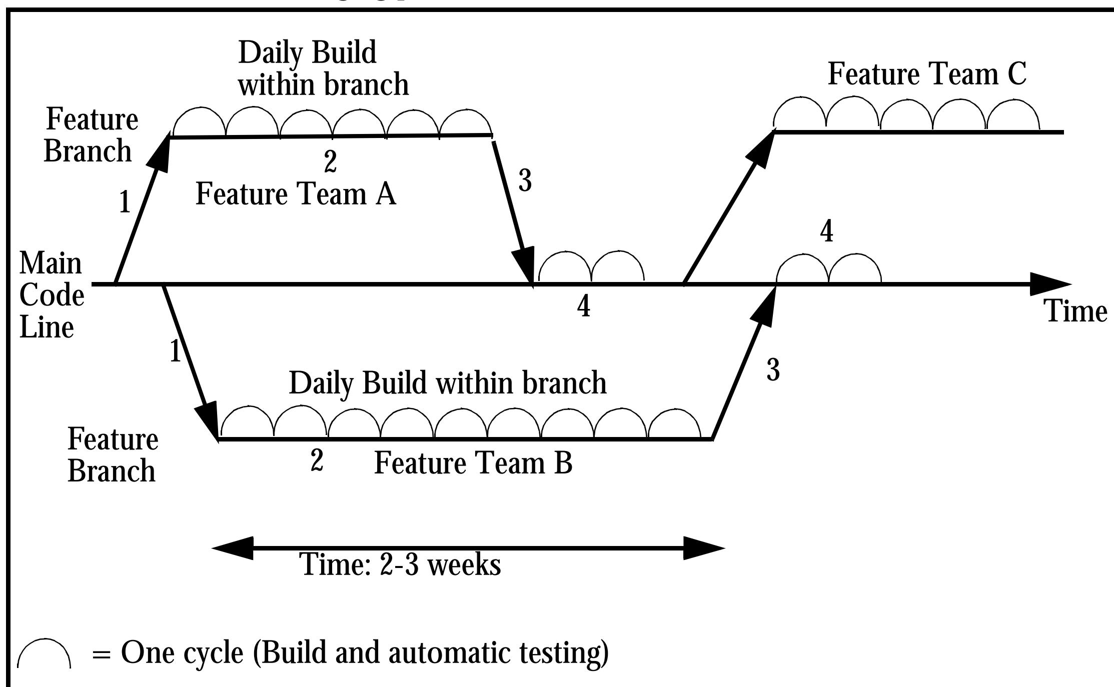


Figure 4. Example of a Daily Build Process 2

1. Check out. The feature team checks out private copy of the main code line.
2. The feature team implements the feature. The feature team builds the system of its own private copies whenever necessary and uses automatic nightly test. The feature team is allowed to make changes only to those files assigned to the team members during this time period. The project must be planned so that other feature teams that need to modify the same files will do it during another time window.
3. Check in the private copies of the files with the new feature implemented. When the team check in they also compare the files with the main code line and handle potential merge problem. There should not be any merge problem since no other team has changed the files in parallel.
4. The integration group starts to compile and link the whole system [see Chapter 4.2.2]. The integration group will not have a major work load, since the main code line are not build more then once or twice every month and the building of the fea-

ture branches are handled by the feature team themselves. The integration group can therefore consist of some designers from some of the feature teams.

In this process, bug fixing is handled in the same way as in the other process.

To use planned handling requires accurate planning of the project, so no feature team is forced to wait on modules. The advantages with planned handling are an easier process and no merge conflicts.

4.2.2. Compile and Link the System

The second step in building the system is to generate a complete build of the system every day. The build is done using the code in the **Main Code Line**. The Main Code Line is the backbone of the system that is being developed, and guarantees that there is always a stable version of the system. If faults are found in the nightly test, the faults are most probably introduced by the code that was checked in the previous day; it is always possible to go back to the Main Code Line of the day before that comprises a stable product.

Most organisations have an integration group responsible for the daily builds of the main code line. The integration group defines a check-in deadline, and all files the teams have checked in before the check-in deadline are included in the build. If the build is successful (compiles), the result is distributed to the test group. If the build group finds a failure that breaks the build, i.e., the system does not compile, the build group must find the responsible team so they can solve the problem as soon as possible. This can be handled in different ways depending on the culture of the company and the country. Some companies in the United States call in their designers around the clock, and these designers must report to work and find the failure - even if the call comes in the middle of the night.

Regardless of whether designers must find fault sources in the middle of the night, the seriousness of breaking the build has to be high. Finding the right point at which to break the build is one of the success factors for daily build. If we set the seriousness of breaking the build too high, we will perform unnecessary work; if we set it too low we will not make enough progress in the build.

Some tricks that have been used by companies to increase the seriousness of breaking the build are:

- Contact the responsible designer in the middle of the night, upon which the designer has to appear and find the failure.
- Create silly hats or door-signs that the person breaking the build must wear until the next person breaks the build. Of course this has to be done with some humor, so the designers do not get upset.
- Let the person who broke the build be responsible for building the system the following nights. This means monitoring the build and calling the next person who

breaks the build.

- Let the integration group contain persons that are recognized as good designers and have authority (informal leaders). To succeed with this, working in the integration group must have high status among the designers.
- High management attention. Every time the build is broken top management in the organisation is informed.

The integration group is also responsible for the configuration management (CM) of the main code line and the different branches that have been checked out. The building of the system has to be done automatically - otherwise it will be too strenuous to do it every day. Most of the modern CM tools can handle this task. The integration group's work is mainly to link the right files into the version that will be built and then follows up the result.

4.3. Testing Strategy

4.3.1. Automatic Testing

Automatic testing is the third part of daily build. It is of little use to build a system on a daily basis if we cannot ensure the quality of the system.

If the daily build was successful automatic tests are executed nightly to ensure that none of the existing functionality in the system is broken because of interference with the newly added code; i.e., a regression test is performed. This is also called a "smoke test". This test has to be automatic as the amount of effort needed to manually do the tests every day is prohibitive. There are two alternatives when it comes to what the test includes:

- Only the functionality level of the last finished increment (if daily build is used in combination with incremental development [see Chapter 5.1.1]); i.e., the functionality in the current increment are not included.
- The functionality level of the last successful build; i.e., the functionality in the current increment is included.

The last alternative makes it possible for teams to include test cases to safeguard their functionality in the test, thus as soon as they have done their own test of the last build to ensure that it has the required functionality, passed test cases can be included in the automatic test. These test cases will ensure that any changes impacting their new functionality will be

detected. This means the number of test cases executed each night will increase during the project [see Figure 5]

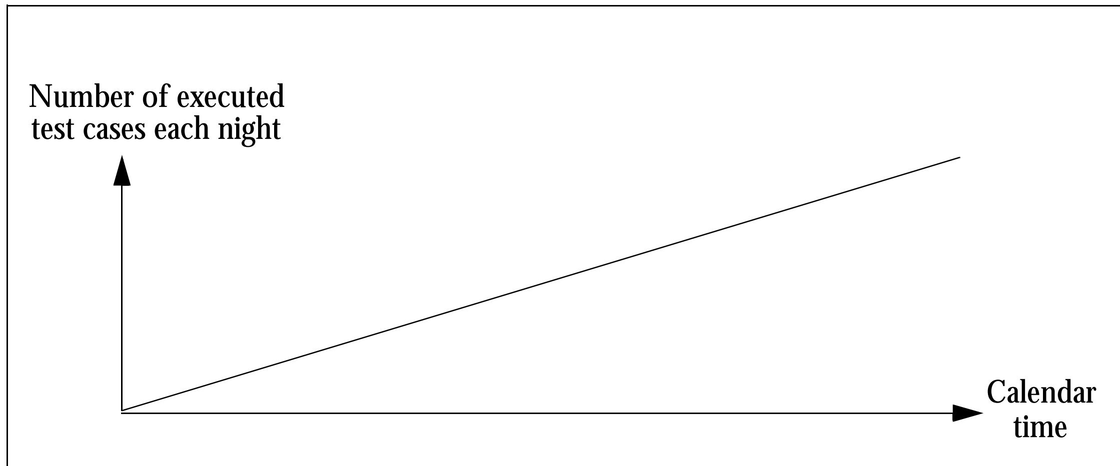


Figure 5. Number of Executed Test Cases

If the amount of test cases becomes too high to be run every day, the test group can devise a plan to alternate test cases so that all test cases are run at least once a week. With several hardware platforms, the test cases can run alternatively on different platforms to obtain full coverage.

If faults are found in the nightly test, the test group has to inform the responsible team. It is usually rather easy to track the faults since the test group knows the faults were introduced by the code checked in the previous day. It is very common that the test group presents the result from the nightly test on an internal web page so the whole company can follow up the progress and quality of the system. Another common way to create understanding and involvement in the development of the product is to involve the whole company in the testing of the product. When the product reaches a stable quality, the company can distribute the successful build to persons in the organisation that use the product in their daily work. Of course this depends on what applications the company develops; for example, companies developing text editors can distribute the product to designers, salespeople, and others, while companies developing aircraft systems will find it more difficult to involve non-development people in the organisation.

Since breaking the build is considered very serious, there must be a possibility for the feature teams to do part of the build and test steps in a private environment before submitting it to the main code line and a system build [see Chapter 4.2.1]. How much of this private activity is cost effective to perform has to be judged for each system, but the important thing is to avoid performing the same tasks twice. We recommend that each team continue with same quality assurance activities that it usually performs in projects not using daily build, such as code review, compilation, and test of new feature functionality, before submitting the code.

We also recommend that all regression tests of old functionality are done centrally by the test group.

The amount of automatic testing possible to execute depends on the product the company develops. Getting the automatic testing to work is probably one of the largest technical challenges to overcome when introducing daily build.

4.3.2. Incremental System Test

After each build the functionality is tested in automatic test. This test covers both new functionality and regression test. However, these tests are smoke tests and the test coverage might not be very high. Therefore a more fundamental function and system test can be executed after each increment that is implemented (if daily build is used in combination with incremental development [see Chapter 5.1.1]). The two alternatives for executing this test are:

- Freeze the code line at the delivery of the increment, letting the function and system test proceed in a stable environment.
- Let function and system test use the same code line as the daily build.

Using a frozen code line has the disadvantage that we must maintain two branches, i.e. serious faults must be corrected both for the frozen code line and the daily build code line. Using the same code line will mean that function and system test will be executed in a less stable environment, and we can never be sure if a fault comes from something delivered in the last increment or last build. Similarly, we can experience that test cases that passed earlier now fail because of faults in code in the latest build that was not detected by the automatic test. However, if we can include all passed test cases in the automatic test run daily we can ensure some quality here. Even if the function and system test is executed in a more unstable environment we have the advantage that faults are detected earlier.

4.3.3. Impact on Traditional Testing

Test are perhaps the part that is most influenced by daily build. Traditionally (not using daily build or incremental development), all the integration and system test was performed at the end of the project [see Chapter 5.1, Figure 6], which often led to “big bang integration” problems. With daily build, the integration and system test is done in parallel with the implementation, avoiding the big bang integration problems [also see Chapter 5.1, Figure 7].

The final test phase (integration and system test) that traditionally has been very long, can now be seen as a final assurance that the system has the required functionality and quality. The lead time of this final test phase can be markedly decreased.

Traditionally, projects have included several different test phases in the development cycle. The practice has been to include as much as possible in the earlier test phases, since it is more expensive to discover a fault in a later test phase, and the feedback loop is longer. The disadvantages with all these test phases are long leadtime and a lot of overlap, i.e. the same functionality is tested several times. With daily build and automatic testing, this assumption is not valid any more, as we can do any sort of tests after each daily build. The feature teams can be more effective in selecting and writing test cases when the test cases make most sense.

4.4. Daily Build in Smaller Projects

In projects with less than 20 designers, it is possible to handle the communication between modules responsible required for delivering consistent code; i.e., it is possible to have module responsibility instead of feature responsibility. Therefore, smaller projects can retain the module responsibility when they introduce daily build. This requires good communication between the designers so they plan together what functionality should be included in the daily builds. For example, designer A implements some functionality in his module that requires that some parts are implemented by designer B in his module. Designer A then has to order that designer B implements those parts. Alternatively, if designers have a good understanding of other designers' modules, designer A can implement the changes himself in designer B's module and tell designer B about the changes, which is close to what a feature team does.

Smaller projects do not exclude feature team responsibility, but feature team is not a prerequisite for smaller projects. One example of a small project using feature responsibility is presented in Chapter 7.8.

A daily build process for a smaller project not using feature teams is simpler than the processes described in Chapter 4.2.1. The following is an example of a daily build process for a smaller project not using feature teams:

1. The designer checks out the file or files to work with.
2. When the designer has made the changes in the file or files the next step is to "check out" a private branch of the current system and do the merge. Then the designer compiles the private branch of the system and, if possible (if implemented in other modules), tests the new functionality.
3. If the merge, compiling and testing were successful, the designer "checks in" the modified file or files.
4. If Step 3 was successful, the modified code is now included in the main code line and will be included in the next build of the main code line.

It is recommendable to have an integration group (or person) responsible for the builds of the main code line, even in smaller projects.

It is much easier to implement daily build in a small project that keeps the module responsibility, since no changes of the responsibility are needed. However, changing to feature responsibility in a small project organization are not too difficult, especially if everyone is in the same location. On the other hand, it is in larger projects and together with feature responsibility that companies achieve the major benefits of daily build.

The testing part is the same as for bigger projects [see Chapter 4.3].

4.5. System Level for Daily Build

It can be difficult to decide at what level in the system daily build will be applied, depending on the product the company is developing. To highlight what we mean we can look at two examples:

Example 1, Ericsson: One of Ericsson's products is a complete GSM system. Ericsson produces all components of a GSM network out to the mobile phone, i.e., the switch, base station controllers and base stations, as well as the Management System. Ideally Ericsson should apply daily build on the whole GSM system, loaded into a GSM network. Today Ericsson does not handle daily build on this level, instead it is aiming for daily build on each node in the GSM network (e.g., a base station controller). These nodes have standardized interfaces, and are usually developed by separate product units. There are usually standardized products for testing the protocols between the nodes. Ericsson could also go into each of the nodes and look at separate subsystems as the system to be built. In this case the hardware has to be a simulator able to simulate the interfaces to the other subsystems. Doing a daily build on a lower level than system means that Ericsson will lose many of the advantages, i.e., consistency check will be local. More information about Ericsson working methods is presented in Chapter 7.

Example 2, Microsoft: Microsoft develops operating systems such as Windows 98 and Windows NT. It also develops applications executed in these operating systems, such as Excel, Word and so on. Daily build could be applied to Windows, Excel, Word and other applications together, since these will operate as one system from the end customer point of view. Microsoft does not apply daily build on this level. Instead it uses daily build on the separate products, i.e., Windows NT, Excel, and so forth are built separately. The reason for this is that this products will also be executed with other programs, i.e., Excel will be executed on other operating systems than Windows, and applications not developed by Microsoft will be executed on Windows. Instead Microsoft ensures the quality of the products together in the nightly test. In the nightly test Microsoft executes the different applications, for example Excel, on different platforms with different operating systems.

No rules exist for choosing a level on which to apply daily build; this has to be determined for each system. The only tip is to apply daily build on the highest level possible that seems

sensible. It will probably require more effort to implement daily build at a higher level, but the organization will also reap greater benefits.

4.6. Projects Suitable for Daily Build

Daily build is suitable for all kinds of projects, but there can be different ways for the organisation to apply daily build - depending on the projects. This chapter presents differences in projects from two points of view:

- Size of the project (both in terms of lines of code and number of designers)
- Type of product (new product or further development of existing product)

Daily build is suitable for projects in all sizes, so there is no upper or lower limit. For example Microsoft has used daily build in projects with more than 30 million lines of code and more than 250 designers. However, there might be differences in applying daily build depending on the size of the project. This is discussed in Chapter 4.2 and 4.4.

The other difference is between projects developing products from scratch (i.e., no earlier releases) and projects developing a new release for a existing product. The main difference between these two types of projects is in their beginning stages. Developing a new product will require more effort to identify features and divide features and basic functionality between the feature teams. Often when a new product is going to be developed, the customers do not know what they want. Daily build can support the customer and the supplier so they can reach a common understanding of what the customer wants, since daily build is a good method for evolutionary developing and prototyping. The deliverer produces some executable code, then discusses functionality with the customer, produces some more executable code - discusses with the customer, and so on. This facilitates the common understanding since they have an executable system to discuss instead of design documents. By using daily build the customer can follow the progress (and understand it) and make sure that they get the desired result.

When developing a new release of an existing product one often knows what new features will be added to the system; then daily build is not used as prototyping.

The conclusion is that the daily build is suitable for all kind of projects, but the type of product the company develops can determine whether daily build is suitable or not. The main restriction is that it must be possible to execute automatic testing on the product.

4.7. Tailoring Daily Build

In the industry today there are different ways that companies use daily build. In this report some different daily build processes are presented, but there are also other possible variations on the theme. The reason for these different adjustments of daily build depends

mainly on the size of the projects in the company, but also on the traditional development process in the company, the people working in the company, and so on. This chapter discusses the most common adjustments of daily build. There are five main areas that change when tailoring daily build, namely:

- The frequency of the builds
- Who checks in code every day
- How to handle the Daily Build process (opportunistic, planned) [see Chapter 4.2.1 and Chapter 4.4]
- At what level in the system Daily Build is used [see Chapter 4.5]
- Integration group or not

The different daily build processes and the level in the system at which daily build is applied have already been discussed. In this chapter we discuss the other three areas:

The frequency of the builds: Daily is not always "daily" in the daily build concept. Some companies build the system only once or twice a week, and still it is considered as "daily build". The frequency of the builds does not have to be constant during the project; in fact, it is very common that the frequency of the builds increases during the project. In the beginning of the project, the system is built once a week and at the end, it is built at least once a day.

Who checks in code every day: We mentioned in Chapter 3.1 that it is not necessary that all the designers and teams check in code every day. The important thing is that new code appears in the system every day and that a check out is not longer than a week. However, some companies want each designer or team to check in new code every day. We do not recommended this since it puts too much pressure on the designer, and it can lead to hacking.

Integration group or not: It is not necessary to have an integration group, but we recommend that the company has a person (or persons) responsible for ensuring the system is built and that it compiles. All companies do not have this - some let the designers build the system themselves.

These are some of the areas where different solutions exist. Other areas can be tailored, and the combination of these areas gives a large number of different adaptations of daily build.

5 Influence of Daily Build on the Software Environment

Daily build started in the PC industry to get control over the development process, while still allowing a focus on the end user requirements and code. Projects in the PC industry are running in an environment without a strong development process and the daily build approach is used to avoid chaos in increasingly larger projects. Lately, companies in other industrial areas, e.g., the telecommunications industry, have started to use daily build in an environment with a traditionally strong development process.

This chapter highlights how daily build influences the architecture, the documentation, and the software process in a company with a strong development process.

The first part contains a brief discussion on the impact on the overall development process (the waterfall model), the architecture and the design documentation needed. Then we present a more fundamental discussion in different areas based on some of the Key Practice Areas (KPA) in the Capability Maturity Model (CMM) levels 2 and 3. CMM is a framework that demonstrates the key elements of an effective software process (see [4]). CMM is often used as an model for software process improvements. The following KPAs will be discussed:

CMM Level 2

- Requirements Management
- Software Project Planning
- Software Project Tracking and Oversight
- Software Subcontract Management

- Software Configuration Management

CMM Level 3

- Peer Reviews

The scenario discussed in this chapter describes how daily build with all three components - build on a daily basis, automatic testing and feature teams - will influence the software process in a larger organisation with a strong development process that is running large projects. However these discussions will also be useful to a smaller company that uses or plans to introduce daily build, since the influence will be the same but on a smaller scale.

5.1. Influence of Daily Build on the Development Process

The waterfall development model [see Figure 6] is used by many traditional software companies. The waterfall development model takes activities such as requirement specification, software design, implementation, and testing, and represents them as separate process phases. After each stage is defined, it is "signed off" and the development goes on to the following stage; none of the phases are executed in parallel.

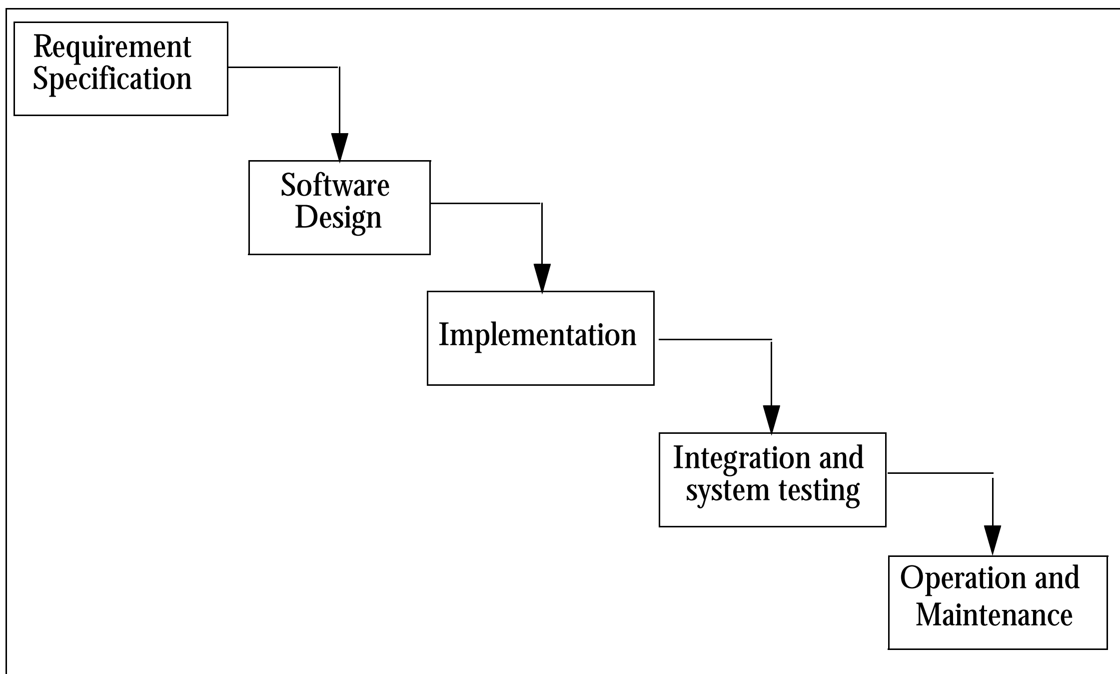


Figure 6. Waterfall Development Model

When using the waterfall development model there is no code to be built until very late in the project, thus the most of the advantages of daily build are lost. To get the full advantage

of daily build, the development has to be split into smaller iterations, where each iteration is taken into executable code [see Figure 7]. Daily build is normally combined with incremental development where the system is divided into a number of increments which are split into smaller iterations, each taken into executable code. This will mean more activities are performed in parallel.

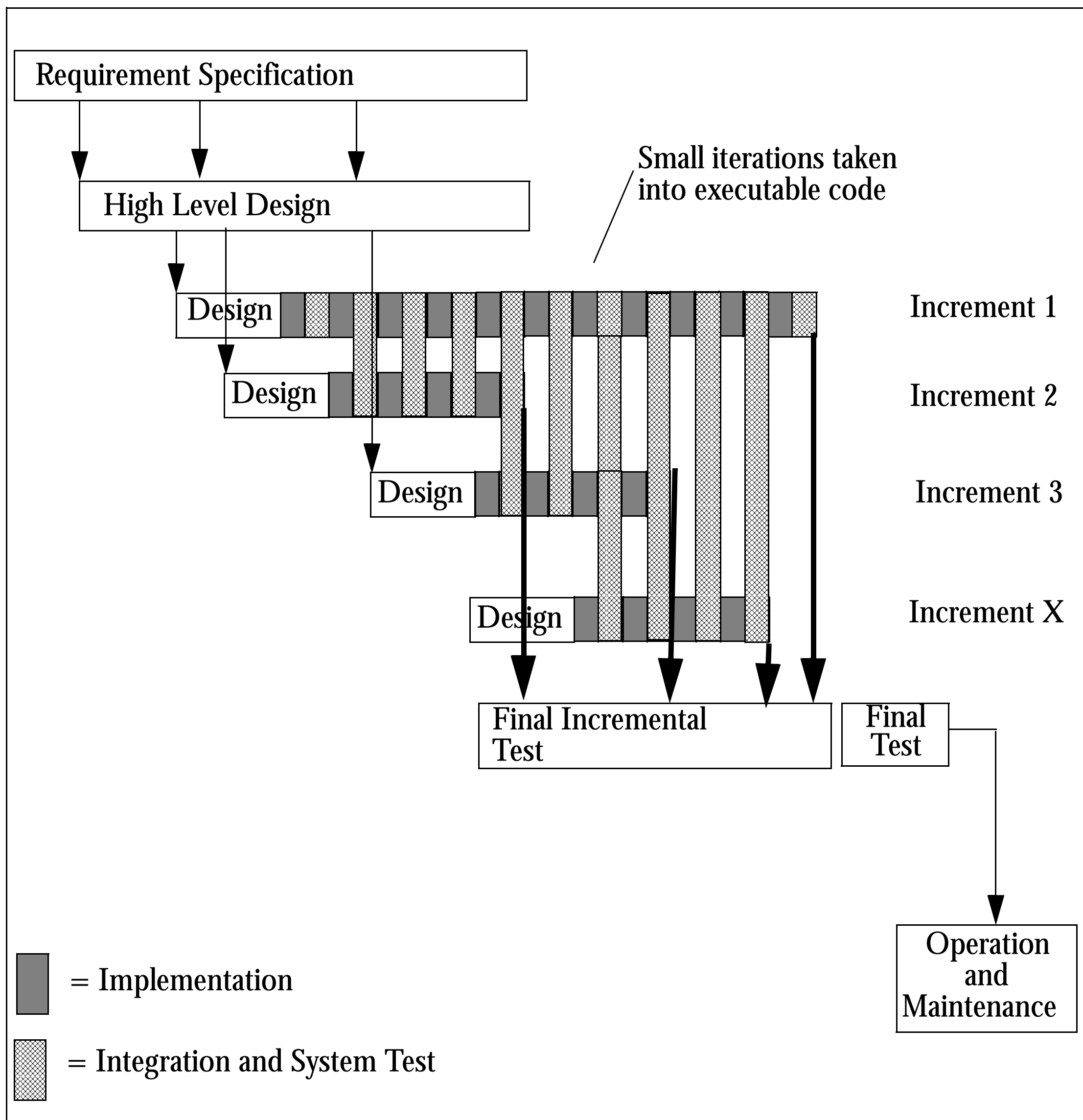


Figure 7. A Development Model Using Daily Build

This is one example of a development model using daily build. Another common approach is that the design of *each increment* is a part of each small iteration instead of being done in the beginning of the increments, i.e., each small iteration includes design, implementation, integration and system test.

5.1.1. Incremental Development and Daily Build

As explained above, daily build is normally combined with incremental development. Incremental development means that you split your system into smaller increments. An increment can have several interpretations within software development, for example:

- In a product line, each release of the product is an increment.
- Within a project there can be several increments, each adding functionality to the earlier increments.
- Each sub-project or team can divide their work within one project into several smaller increments. These could be the smaller iterations that are made on a daily basis.

Several ways exist to decide what functionality each increment should have, i.e., the type of increments. When combining incremental development with daily build and feature teams, it is natural to base the contents of the increments on features, i.e., each increment contains a package of features. The choice of which features to add when can be made based on characteristics such as:

- cost/benefit ratio for customer
- stability of requirements
- basic - advanced (from a user point of view)
- simple - complex (from an implementation point of view)
- critical - non-critical (from a safety point of view)

When using feature increments and daily build, each feature team works in one increment. This allows several increments to go on in parallel [see Figure 7]. The first increment to start can be the last increment to finish, no values have to be set for the duration of the increments.

By using feature increments and feature teams, projects will not be as dependent on the delivery dates for increments as projects running with incremental development and module responsibility. If a module in the later case is delayed, all the features in this module are delayed. A more troublesome situation arises if this module is a ‘spider module’ (impacts a lot of the features); more or less all the features in the system will be delayed. If one of the features is delayed when using feature increments and feature teams, the rest of the features can still be delivered in time.

5.1.2. When Can Designers Start to Write Code?

A frequently asked question concerning daily build is when to start using the smaller iterations, i.e., when to start writing code. There are two major forces that impact the start of writing code:

- What phases of the development process do the iterations cover?
- When can we start to write code in each increment?

Basically the idea is to start coding as early as possible. As soon as the specification of the architecture and interfaces starts designers can start to write code; i.e., the architecture and interfaces are documented in the code. The code will not be sufficient documentation and must be complemented with other types of documentation. The complementary documentation shall be as little as possible; i.e., as much as possible should be documented in the code. The complementary documentation should, if possible, be updated on-line, meaning when a change is made in the code that affects other documents, these are also updated directly.

However there is always a risk that designers start to write code too early, before there is a good understanding of the architecture and interfaces. This might force projects to redo what was done in the earlier iterations because of changes in the design in the later iterations.

When we have a stable system, and the new projects are mainly adding new, rather independent features, we can allow coding to start for a feature rather early. Here however, we have to ensure that we can handle any feature interactions properly.

The desired situation is to start to write code so that no unnecessary design documentation is created. All design decisions should be represented by the code, and not in superfluous documentation. Instead of first writing documentation describing the solution, we directly implement the solution in the system by writing code.

5.2. Influence of Daily Build on Design Documentation

As mentioned before, daily build will split the development into smaller iterations. Therefore, the amount of design documentation has to be adapted to the iterative way of developing software. The design documentation has two different purposes:

- Support maintenance and enhancements of the system
- Serve as input to the next stage of the development process

The first purpose usually requires less documentation than the second. For the maintenance purpose, the design information can be on an overview level and details can be found in the code, as we have all levels available during maintenance. For input to the next design phase

we need more detailed design documentation since this will be used to hand over information from one person to several others. There is a tendency to put additional details in the high level design documentation to help the later design stages where we do not have code. Working more iteratively in feature teams with less handover of intermediate results, with all levels of documentation at the same time, will allow us to put the information directly where it belongs. This requires that we have a design organization where a team is responsible for an end-user functionality. Otherwise, we have a hand over of information about many customer features, to the module responsible. This design information has to be more detailed than the information necessary for maintenance, as the module designer is going to implement the impacts in the modules based only on that information, with limited time to understand each individual feature or communicate with different feature designers.

Changing the documentation in this way will mean that the design documentation will be overview documentation and details are to be found in the code. Therefore, it is important that the code is written in such a way that it is easy to understand and well commented. The feature teams also result in designers modifying code written by other designers. From that aspect, it is also important that the code is easy to understand.

Changes in the design documentation because of failures found during the project will also be easier to handle when using daily build and feature teams. The feature team is responsible both for the code and the documentation and has the responsibility to correct both the code and the documentation. With module responsibility, the person responsible for the failing module has to inform all persons responsible for documentation of the feature that is affected by the failure in the module. However, even if daily build and feature teams are used, the old problem still remains that the documentation is forgotten and is not updated because of time pressure.

5.3. Effect of Daily Build on Architecture

To be able to discuss how daily build impacts the architecture, architecture must be defined. Architecture is a very comprehensive term; to make it less broad, we divide architecture into five different views [3]: design view, implementation view, process view, deployment view and use case view. When discussing architecture in this report, the design view and the implementation view are covered, and the other views are not considered.

The design view encompasses the classes, and interfaces and supports the functional requirements of the system. The implementation view encompasses the components and files used for the physical system. From this point on in the report, the term “architecture” includes only these two views.

One of the most frequently asked questions about daily build is ‘how does daily build influence the system architecture?’. Many people are worried that using daily build in the long term will lead to worse system architecture. The reasons for this concern are:

- daily build increase the pulse in the system which can lead to ‘quick and dirty’ corrections
- too much focus on feature team can lead to less focus on the architecture
- daily build makes it possible to use more prototyping during the development, which can lead to less thought out architecture

Although these concerns seem most likely to be valid, we have not identified any company or project that has encountered problems with the architecture because of daily build; i.e., we have not found any proof that using daily build should lead to problems with the architecture.

Even though we have not found any proof we think it is important to continue the focus on the architecture when using daily build. To avoid potential problem with the architecture it is important to:

- have the right balance between feature responsibility and architecture responsibility. A common solution is to focus more on architecture in the beginning of the project, and when the architecture is stable move over to more strict feature responsibility.
- if using prototyping, go back and structure the architecture when finished with the prototyping
- have one person responsible for the architecture [see *architecture police* in Chapter 4.1]

Using feature teams puts some new requirements on the architecture. All feature teams have to understand the system architecture (all five views) since their feature might be part of many modules. Therefore the architecture has to be easy to understand.

It is also important to minimize critical points in the architecture. Critical points are parts of the system that are affected by many features. These points will be very complex when many feature teams have to modify this code. For those critical points that can not be avoided we suggest that the project has one person responsible for the code, compared with module responsibility in Chapter 4.1.

An advantage in using daily build is that the architecture performance and functionality can be tested in an early stage.

5.4. Requirements Management

Requirements management is influenced differently depending on the organisation of responsibility.

There are several disadvantages for requirements management when using module responsibility; e.g., the software engineering group that must analyze all requirements:

- can be a bottleneck slowing down the project
- has insufficient knowledge about the feature
- has insufficient knowledge about the influenced modules

By using feature teams, the requirements management will be handled more inside each feature team. Normally, a new requirement or a change of a requirement affects only one feature. There are several advantages in letting the feature teams handling all the requirements for their feature, e.g.:

- All requirements and changes to requirements for a feature are handled in one team.
- The feature team has the best understanding of how the requirement will affect their feature and the plans for its feature.
- The feature team is the most suitable to decide if the requirement can be fulfilled with the competence in the feature team.
- The feature team can make decision about all smaller changes not affecting the cost or the lead-time of the project. The project management will only need to be involved if the cost or lead time is affected.

5.5. Software Project Planning

In the waterfall model with module responsibility, there are three major problems with project planning:

- Finding the right scope for the project
- Too many levels in the organisation are involved in the planning
- Planning focuses too much on modules instead of features

The first problem is to decide the scope of the project - what features to include in the project. A common problem at this stage of the planning is that the scope is far too big since the product management is afraid of excluding any features, as there is little chance to get the feature back in again. Therefore, the first task for the project management is to limit the scope as quickly as possible to get the project under control, and to be able to start realistic planning.

This problem can be addressed by using feature increments [see Chapter 5.1.1]. The idea is to have a more flexible set of features to work with throughout the project, based on customer priority, stability of features, and project resources.

For example: when the scope of the project is decided (in terms of end date, total resources and new features) all features are grouped into four categories:

1. Very important and stable features (candidates for first increment)
2. Important and relatively stable features
3. Other important features
4. Other interesting features

Features in group 1 is put on a 'fast track' through prestudy and feasibility for early start of execution in the first increment. Features in group 2 are driven through prestudy in the usual speed. Features in group 3 are given less priority, and only prestudy for complex ones are started. Features in group 4 are in reserve, and no resources are used on them from the prestudy project.

Project management can now start preliminary planning of resources based on this grouping. This does not mean that the plans will not be changed - they indeed will, but at least there is a baseline to change.

The scope will change during the project based on the lead-time and resources and there will be a continuous adjustment of the features. The decisions about the scope will be based on actually facts comparing resources, lead-time and estimated effort for a feature. By grouping the features based on importance, it will be ensured that the most important features are implemented in an early stage. If the project starts to exceed the cost limits and lead time, there is now a possibility to postpone features to the next release or decide not to implement them at all. If all features would be implemented in parallel we would not have this opportunity, and we have to keep on until all the features are implemented.

This will be a rather complex project environment, where the main project steering mechanism is the current list of features, their status, and estimated completion cost. Delegating each feature to a team which will have the complete responsibility to develop this feature, will then make project follow up and replanning much easier. Removing one feature will affect only one team, and it is easier for different teams to work independently.

The other two problems with the planning in the waterfall model are that too many levels in the organisation are involved and that the planning is focused on modules instead of fea-

tures, mainly in the lower levels of the organisation. Figure 8 gives a brief description of one common way to plan larger projects.

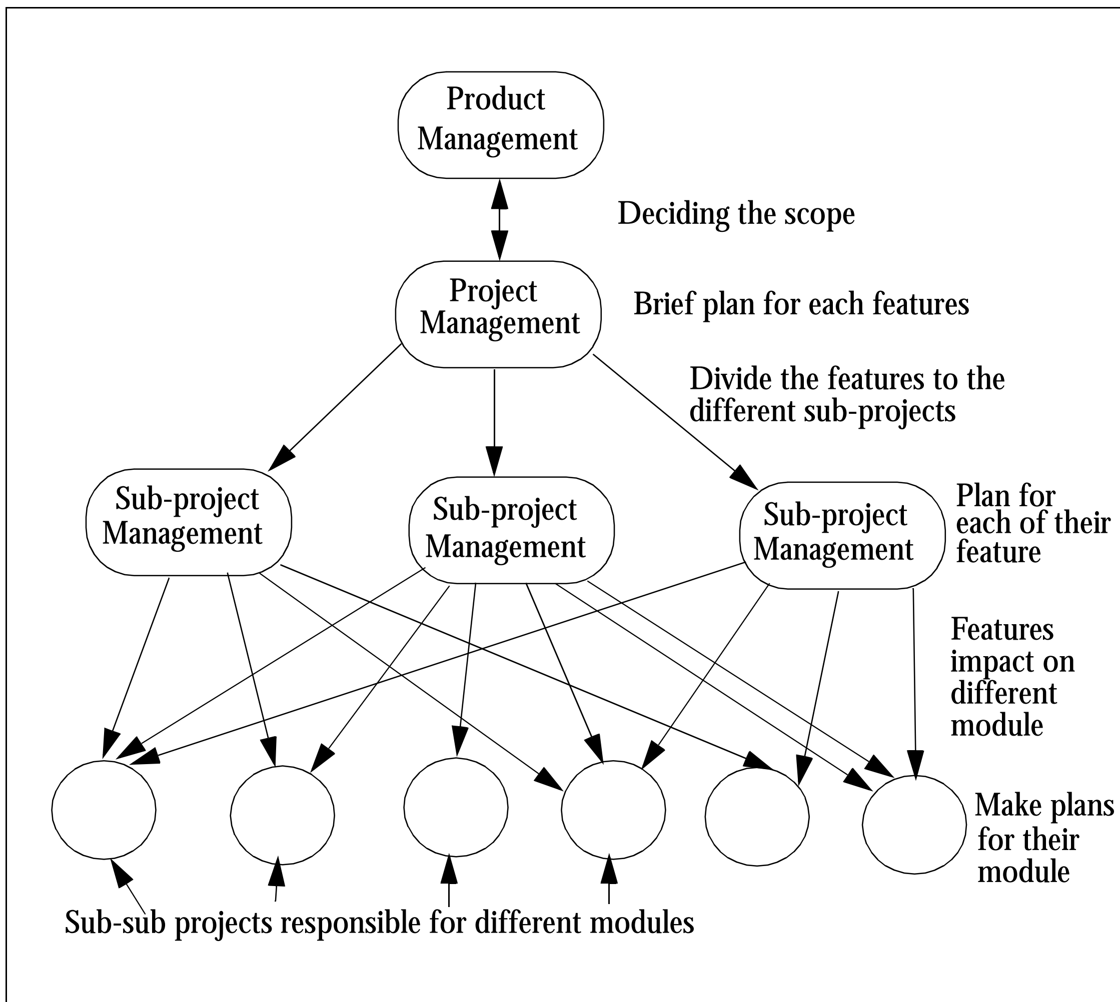


Figure 8. Project Planning at Different Levels in the Organisation

As we can see in the figure, at least four levels in the organisation are involved in the planning procedure. The three top levels plan the time and resources for features while the sub-sub-project plans are made for time and resources for their modules, based on how many features influence the module. It is often not possible to follow up the actual time spent on a feature by the end of the project, since the only thing that has been followed up is time spent on each module. Therefore there is no feedback to the persons making the plans for the features.

When working with feature teams, the team responsible for the feature can do all the planning for the feature. This will also make it easier to follow up the actual time spent on the feature, i.e., providing feedback to the team doing the planning. However, the most signifi-

cant difference is noticed when a change request occurs. Normally a change request affects one feature. When using feature teams, a change is “isolated” and visible, i.e. the change affects only one feature that will be delayed. Since the change can be analyzed and re-planned by the feature team, the change is under control.

If feature teams are not used, the analysis has to be made by the sub-project responsible for the feature. Then each module affected by this feature has to be analyzed and re-planned. Normally this is complex and therefore no real re-planning is made; we just hope that each module responsible will be able to implement the new requirements as well. Thus the current plan for the affected modules will have hidden delays, which usually turn up toward the end of the increment. This will mean all of the features depending on these modules will be delayed.

If the project is distributed on different geographical sites, allocating a feature to a feature team at one site will also allow more flexibility in when we start development of each feature; i.e., when we have resources (with the necessary competence) at one site, we can start a feature there, and we avoid all the coordination with resources on other sites that previously needed to work in a synchronized manner. The feature team will of course need support and guidance from other sites for products (modules) which they are not familiar with, but this is less troublesome than having the other site doing the work. The feature team still has to have the core competence for developing the feature.

Note that our flexibility is not entire, as our resources are not completely interchangeable even if we use feature teams, thus if we want to replace one feature with another, and the team intended for the first feature does not have the competence for developing the other feature, we need to re-allocate people between teams. Thus product management does not have infinite freedom.

The discussion so far has touched upon how the planning procedure is affected by feature team. The other two components of daily build - build on a daily basis and automatic testing - also put new requirements on the planning. Design has to deliver new code to the system on a daily basis. It is not possible to make a plan detailed enough to specify what the feature teams have to deliver each day. Instead, it is recommended to have a plan specifying what each feature team should deliver on a weekly basis. For example a booking system can be used, where each feature team, at the end of each week, books when to deliver new code next week. The feature teams then must coordinate their work so that new code is delivered to the system on a daily basis, i.e., if there are two teams, the first team already checking in one day, then the next team takes the day before or after instead to check in.

There are no measurements proving that the effort spent on planning when using daily build and feature teams decreases or increases. However, most people are convinced that it leads to more effective planning and better plan content.

5.6. Software Project Tracking and Oversight

This is probably the area that will gain the most benefits from daily build and feature team.

Feature teams are an advantage for project and product management, since it can communicate directly with the feature teams, and also see early versions of the features.

With daily build, the project tracking will be on actual code and feature functionality instead of documentation and modules. Daily build gives an opportunity to see 'actual' progress in the code instead of estimated progress. Each day you will see what functionality there is in the system. This can be compared to the feature plan, describing when and what to implement.

As described above [see Chapter 5.5] it will be much easier to handle changes with feature teams. Both changes to external groups that relate directly to one feature and changes to the feature team can be handled internally in the feature team. Each feature team can also follow up effort, cost and changes to the feature internally in the feature team. Even risks associated with the cost, effort, and technical aspects should be handled inside the feature team.

5.7. Software Sub-contract Management

There are two different types of sub-contractors:

- Sub-contractors developing software that many features depend on, e.g., a platform for all the features
- Sub-contractors developing software related directly to one feature

The feature team approach has no impact on the first type of sub-contractor. The second type, the one that relates to just one feature, should be handled internally in the feature team. Thereby the feature team will get closer cooperation and better tracking of the sub-contractor and the sub-contractor will always get in contact with the right people with the right knowledge.

The other two components of daily build - build on a daily basis and automatic testing - also have an impact on how we handle the sub-contractors. The ideal image is that software that the sub-contractor develops is involved in the builds on a daily basis. However, this requires very close cooperation and a good relationship with the sub-contractor. Most sub-contractors are used to a situation where they get a contract defining what to deliver and when. To succeed with this new requirement to deliver new code on a daily basis, the benefits for them must to be clear.

Daily build will also imply problems with financial issues. This new way of working requires the sub-contractors to plan their own projects in a different way than before.

However, it is recommended to involve the software developed by the sub-contractor as frequently as possible in the builds.

5.8. Software Configuration Management

Daily build requires a modern CM tool that:

- Makes parallel development practical
- Identifies common code
- Resolves inconsistence
- Highlights code conflicts
- Handles version control and track changes
- Provides possibility to label or 'tag' a selected version of the code

In the case of feature teams the baselining of artifacts is very important, i.e. that everyone in the project at all times know which files exists, and what the latest status of these files are. It would be rather problematic if I didn't know that someone else had started on the same module that the feature I was working on impacted.

5.9. Peer Reviews

Peer Reviews have an important role when using daily build. Peer reviews are the control mechanism that force the designers to think through the solution before they implement it. Peer reviews works against too much hacking and that the speed in the project is too high, i.e. the project can not be controlled.

Peer reviews also have an important role to spread information between the different feature teams and internal in the feature team.

Peer reviews should be organized so that the module and architecture responsible can control the relevant artifacts. It is also important to organize the peer reviews so that there is a consistency in the reviewing of a certain module or document, as the iterative nature of daily build will mean that many items need to be reviewed several times.

6 Guidelines for Implementation of Daily Build

In this chapter it is discussed how to implement daily build in an organisation. It is not an easy task to implement daily build, since it is a rather fundamental change compared with the traditional way (waterfall method and module responsibility) to develop software. Daily build both requires an infrastructure for automatic build and test as well as a change to the organisation of design, i.e., feature teams. Both the projects and the organisation are affected by the changes.

6.1. Self Assessment

Before starting the implementation of daily build, it is strongly recommended to do a self assessment on the current build and test infrastructure. A gap analysis should then be performed, comparing the current situation with the vision of daily build, and to identify what actions have to be taken.

This self assessment will help the organization to establish a roadmap and a reasonable time scale for implementing daily build.

We also recommend to use the self-assessment regularly during the implementation to understand the current status and the progress of the implementation.

6.2. Practical Guidelines for Implementation

The implementation guideline is divided into two tracks: one focuses on establishing the infrastructure, i.e., build and test. The other track focuses on the changes to the organization of design.

Here we give a rough structure for how to run the improvement effort so it will lead to a persistent change.

6.2.1. Infrastructure Track

This track aims at establishing the infrastructure for daily build:

1. A modern CM tool (for example Clearcase). The organisation need an efficient environment to keep track of the different products, and also be able to know which products are going to be used in the build. Using labels in e.g. Clearcase makes it easy to extract the correct version of a product for build, while work is going on in a newer version. A modern CM tool also supports working on the same products in different branches for later merging.
2. Automatic and fast build. Most organisations have a semi-automatic process to do the build, often the companies have the tools to automate each step, but usually there is a lot of manual work between the steps and for running these tools. This work is often necessary since the quality of the input is not good enough, therefore time is used to chase missing parts or to correct faults. Many of these faults arise because the build process is used so seldom. Assuming that the builds are made on a daily basis, then these problems will go away, and the builds can be automated to a larger extent.
3. Automatic testing. Most organisations have most of the infrastructure for automatic testing in place, but since a test case traditionally is used only once at the end of the project, this has not been used. With daily build this will dramatically change, and we see daily build as one of the major driving forces for automatic testing.

6.2.2. Design Track (Parallel with Infrastructure Track)

The purpose of this track is to change the organization to use feature teams and early coding.

1. Awareness building with line and project management about the organizational impacts, advantages and challenges.
2. Technical analysis of system and feature structure to understand how feature teams could be realized.

3. Organization of the different project phases, e.g. prestudy, feasibility, design/implementation and test in feature teams

To motivate the introduction of feature teams, organizations have successfully used the approach of looking at the advantage for the different stakeholders if we had feature teams in place; i.e, assuming they had overcome all technical, competence and organizational barriers. They have mainly looked at this from the following perspectives with some of the most common advantages found:

- The individual designer will get a more stimulating work environment as he will be more end-to-end responsible for some customer functionality, and will also get a better system overview.
- The customer will have someone in the organization to communicate with continuously regarding different features, and will see early versions of the features as they are developed.
- The local line organization will have a better chance of building and retaining competence, as people can grow in the complete system knowledge in each organization. The line organization can also get an increased business focus, as it can be responsible for a feature area, not only a subsystem.
- The project will be easier to manage as it will consist of self-contained and managed feature teams.
- The product unit will have a higher flexibility in where to allocate work, i.e. a feature can be allocated to the design units where there are resources available, and it is not constrained by the design unit with most work in their subsystem.

6.3. Measurements for Daily Build

To follow up the progress in the implementation, measurements can be used. The measurements will indicate if there is any progress or if alternate plans and actions are required. This chapter presents one example of a measure that can be used to follow up progress.

The measure is "frequency of successful builds".

$$\frac{\text{Number of working days with successful builds}}{\text{Total number of working days between (the first day code is written in the project - last day code is written)}}$$

A successful build must fulfill some criteria, indicating that daily build is used in a correct way:

- one of these criteria is control of the daily code size. We want the size of the code that is checked in to be almost the same size every day. If only small pieces are checked in at the beginning of the project and 80% of the code is checked in the last week of the project, there is no real use with daily build. This is one example of how to follow up this criterion:

$$\text{daily code size} > \frac{0.5 * \text{planned total code size}}{\text{planned number of days of coding}}$$

- a successful daily build verification has to be done; e.g., 65% of the executable test cases are executed.
- a successful distribution of the dump to the ones requiring it has to be done. This means that it must be easy to handle the dump so everyone that requires it can load the software dump into their hardware systems for testing.

Each organisation must define their criteria and how it will follow up these criteria. It is good to have some key indicators to ensure daily build is used as intended.

6.4. Success Factors

These are some of the success factors for implementation of daily build. These success factors are closely related to the implementation steps described in the earlier chapter.

- Get commitment for the changes at the highest level in the organisation, i.e. business unit level, as it will impact both the project and the organisation.
- Have a management seminar discussing the impacts of daily build, and to create a commitment for the changes.
- Be aware that the implementation of daily build is a long process and will require a lot of effort.
- Work in parallel with the technical infrastructure, i.e. build and test, and in preparing the change in the development organization. The implementation of the technical infrastructure should be one step ahead of the organizational changes.
- Involve someone who has experience from implementing daily build in other organizations to guide the implementation.

- Since this change will take some time, a steering group on management level is recommended.

It is also important to have a strategy on how to sell in these changes in the organisation. We recommended the following eight steps from the book *Leading Change* [5]. Some of these steps are similar to the success factors mentioned above:

1. Establishing a sense of urgency:
   - Examining the market and competitive realities
   - Identifying and discussing crises, potential crises, or major opportunities
2. Creating the guiding coalition:
   - Putting together a group with enough power to lead the change
   - Getting the group to work together like a team
3. Developing a vision and strategy:
   - Creating a vision to help direct the change effort
   - Developing strategies for achieving that vision
4. Communicating the change vision:
   - Using every vehicle possible to constantly communicate the new vision and strategies
   - Having the guiding coalition role model the behavior expected of employees
5. Empowering broad-based action:
   - Getting rid of obstacles
   - Changing systems or structures that undermine the change vision
   - Encouraging risk taking and nontraditional ideas, activities, and actions
6. Generating short-term wins:
   - Planning for visible improvements in performance, or “wins”
   - Creating those wins
   - Visibly recognizing and rewarding people who made the wins possible
7. Consolidating gains and producing more change:
   - Using increased credibility to change all systems, structures, and policies that do not fit together and do not fit the transformation vision
   - Hiring, promoting, and developing people who can implement the change vision
   - Reinvigorating the process with new projects, themes, and change agents
8. Anchoring new approaches in the culture:
   - Creating better performance through customer- and productivity- oriented behavior, more and better leadership, and more effective management

- Articulating the connections between new behaviors and organizational success
- Developing means to ensure leadership development and succession

7 Experience from the Swedish Software Industry

This chapter presents experience from the Swedish Software industry using daily build. The following companies has been interviewed:

- Ericsson UAB; the OTP group
- Excosoft
- IAR Systems
- ABB Automation Products
- Telelogic
- SAAB Military Aerospace
- Ericsson: GPRS development for the BSC
- Ericsson: OSS development in Mölndal

The authors also tried to set up an interview with Microsoft development centre in Redmond, USA. Unfortunately there was no possibility to perform this interview with Microsoft. However, we did find out that Microsoft still uses and further develops its daily build method. For more information about how Microsoft uses daily build see [1], [6] or http://www.microsoft.com/windows/server/beta/hall/99Jun30.asp.

7.1. Ericsson UAB; the OTP Group

The OTP group develops Open Telecom Platform (OTP), a development platform for telecommunication systems. There are few people in the OTP group, for the present 17, with high competence within the field and they take full responsibility for the whole process; from specification to the final tests. The OTP group consider this as a fundamental condition for a well working daily build in their organization

The projects involve designers and people from the integration group. The projects develop mainly new releases of the existing product. The latest release is used as input to the next project.

7.1.1. Project Organization and the Daily Build Approach

The OTP group build and test the whole product on a daily basis, i.e. every night the whole product is built and tested containing new or modified code. The daily builds are fully automated and are run by the integration team, which consists of three designers responsible for the configuration and the follow-up of the daily builds. The integration team also defines what branches in the configuration Management (CM) system the designer should use. Each designer decides when to check in new or modified code. The daily builds encourages the designers to check in often and as soon as possible to get the feedback from the test runs. The most valuable feedback from the daily builds is to observe if changes in one application have caused some other application (which depended on the first one) to malfunction.

The OTP group are working with two different types of branches, namely the ‘design branch’ and the ‘release branch’. In the early phase of the projects they only use the design branch. It is more of a prototyping approach to the daily build and a breaking of the build or introducing of failures is not so serious. The designers use daily build to test their new solutions. If the designer has made a larger change or like to do an experiment to test a solution that is not very stable, he can create his own branch, from which the integration team can build and test the code during the night. Builds are also made for different platforms and the number of builds made during a night may be up to seven.

The seriousness of breaking the build increases during the project. Later in a project, when there are about three to four weeks left to the release, the integration team merge over the code to a release branch. However the designers still use the development branch, so new or modified code first can be tested there. The code checked into the release branch must be stable to avoid breaking of the daily build. The merge to the release line is delayed one to two days after the code is tested in the design branch [see Figure 1].

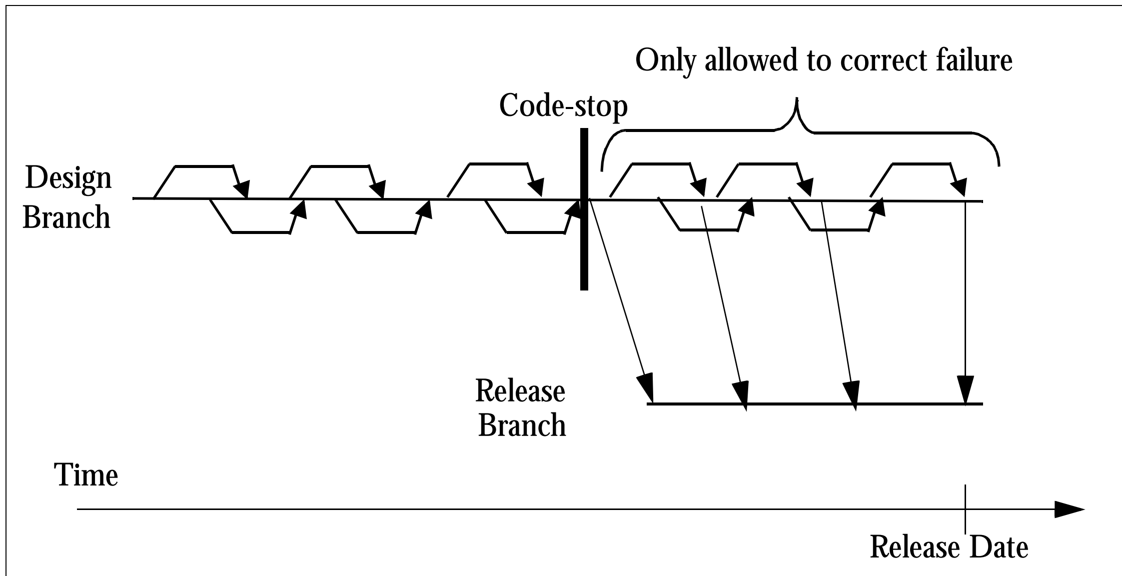


Figure 1: The OTP Group Design and Release Branches

The time-gap between the two branches decreases as they approach the release date. At the end, the two branches are identical.

A code-stop is introduced in the design branch, when the release branch is created. That means, it is not allowed to add any new functionality to the system The only reason to implement new or modified code is to correct failures.

When a failure is found in the release branch, the designer responsible has to correct the failure and test the solution in the design branch before the change is introduced into the release branch. The design branch can be seen as a bug-fixing branch. At this stage, the integration group handle potential merging problems.

The final product consists of a number of OTP applications. A change in one application can force a change in another application, and this is solved by good communication between the members in the OTP group.

7.1.2. Automatic Testing

The OTP group has progressed well with automatic testing. The group developed its own test server, i.e., a framework for writing automated tests. The test cases are written by the designers, and when adding new functionality into the system, the designer also writes test cases to verify it. Additionally, the integration team writes test cases to ensure the product has the right functionality. All the test cases are executed each night, both to test the new

functionality and as a regressions test. The number of test cases executed increases by each day. At the time of this interview, the OTP group executes about 1700 test cases each night.

The results from the automatic tests are presented as a set of linked HTML pages where one can see summaries, lists of test case groups, log files of fault reports, and links directly into the source code where a test case failed. The integration team is obligated to report and follow up all failures.

When running a daily build the failures can be caused by improvements in the daily build script itself, the script configuration, disks that are full, bugs in the test server program, and other problems. The designers are not responsible for these things, so sending mails to them automatically (a practice discussed in the beginning of this report) can create a risk that the fault reports are ignored since they rarely handle real bugs. An exception from this is the build of documentation; the person responsible gets e-mails daily from the build.

The product that the OTP group develops supports many different operating systems and operating system versions. The designers mainly do the coding and testing on Solaris Sparc. However, they must ensure the functionality on other operating systems as well. Thanks to the daily build process, designers can test their product on different operating systems in an early phase. This saves them a lot of time.

The OTP group also builds all of the documentation on a daily basis. The sources are SGML, GIF and Postscript files. From these sources, they generate the documentation as HTML pages, Unix man pages and PDF. The nightly tests also check that all the Unix main pages can be viewed, as well as for broken HTML links and any files that are never referenced. The results from these tests are sent automatically to the person responsible for the documentation.

Besides automatic testing at OTP, some customers do Beta-testing on the product.

7.1.3. History

When this report was written, the OTP group was the company that had developed the most automatic process to build and test on a daily basis. Considering that they only have been using the daily build process for four months at the time for this interview, OTP has achieved a very fast introduction of the concept. The designers had a good starting point, however, since they have been using automatic testing for some time.

The OTP group also thinks it is important to realize that the daily build process will not be perfect. Therefore, one person will be pointed out to be responsible for continuous improvement of the build process and to take care of feedback from designers.

7.1.4. Benefits and Hindrances

The main benefits that daily build has given the OTP group are:

- Possibility to perform regression test
- Possibility to test on different operation systems
- Stability in the product
- Aid to a designer
- The building and checking of the documentation give more time for focusing the content
- Smoother integration and shorter integration time
- Intermittent failures or failures on specific platforms can be found and solved early

The OTP group still has some minor tasks to overcome before their daily build is totally perfect:

- The amount of test set-ups increases dramatically with the number of supported operating systems, different versions of the same operating system, different hardware, different virtual machines used in the product, product versions, and patched product versions. To fully cover all combinations much more hardware is required than what is available now.
- The daily follow-up on failed test cases, build problems, and other issues is time consuming. It means that the integration team must use some time to check the results; nevertheless, the overall saving of time is greater.

7.2. Excosoft

Excosoft is a software manufacturing company that develops a literate programming editor. Literate programming is a developmental approach that combines code and documentation in source files. Excosoft's internal projects are largely dedicated to the development of new releases of this software. The most recent release of the software is input to the next project. There are approximately four external customer releases per year. Between these external releases there are a number of internal releases. Each project involves about 14 designers and lead time is 2-4 months.

7.2.1. History

Excosoft has used the daily build approach since the company was founded, the company has thus more than ten years of experience using this method and cannot conceive of using any other. The reason that Excosoft pioneered the daily build approach is to be found in the very nature of the products they develop. Daily build is a natural solution when developing applications with a pronounced customer focus. During the ten years Excosoft has utilized the daily build process one major change has been implemented: the company now works with files in parallel using a proprietary CM tool. In the early days development was carried out sequentially and designers were compelled to wait on each other.

There is one aspect of the daily build process at Excosoft that is problematic: it is difficult to apply automatic testing routines. Excosoft is evaluating solutions that will remedy this deficiency.

7.2.2. The Build Process at Excosoft

Each designer at Excosoft normally initiates a whole or partial system build more than once every day. Since building the system only takes about 5-60 minutes (depending on the platform) there is no need for any dedicated integration group and the designers themselves are responsible for the individual builds.

The build process can be broken down into four separate steps:

1. The designer creates a private branch line from a frozen version of the current system on the main development line.
2. He then edits the file/files designated for change in a working version of the private branch. The private branch is then compiled and tested.
3. If the compilation was successful the designer then attempts to merge the branch version back into the main development line.
4. If the merge is successful the modified code will be included in a new frozen version on the main development line and will be included in the next internal release.

Excosoft utilizes opportunistic locking when designers work with the same file in parallel on the same development line. There are two notorious problems associated with working in parallel on the same code line or in temporary branch versions. Excosoft has solved both these problems.

The first problem is that designers dislike other designers meddling with their code. The second problem is that it can be difficult to understand code written by others, usually due to lack of documentation. Excosoft has fostered a company culture that promotes collaborative working and emphasizes the importance of documenting code. Documenting the code is more important than performance and quality since these latter can be improved if the code is easy to understand.

This culture fits well with the product that Excosoft develops, namely an editor that supports literate programming. This tool, working within the in-house company culture ensures that problems encountered when designers modify code written by others are minimized.

Another interesting feature of the Excosoft daily build process is the "do not merge" flag. This flag is set by a designer carrying out a substantial change in a branch version, e.g., a total redesign of part of the system. The flag is set on individual files in the main development line and will prevent other designers from merging in other branches containing changed versions of these files.

Software modules written by sub-contractors are also incorporated in the Excosoft daily builds since the daily builds reproduce the whole system.

7.2.3. Test Strategies

Because of the very nature of the software product developed by Excosoft it is difficult to implement automatic testing routines. It is hard to write test cases that address the functionality of a literate programming editor. For example, when the editor produces Postscript files it is impossible to verify the result automatically as the files may vary through time and still be correct.

This type of problem is solved by involving the whole company in the testing process. There is a high frequency of internal program releases which are distributed throughout the company. All staff members use the latest internal release in their daily work (with the exception of early, instable releases that are tested by the designers themselves). As projects start off from the previous release designer testing is completed early on, and as the whole company progressively becomes involved in daily testing it becomes a sport to find faults. A salesperson, for example, will be excited if he or she finds a fault that the designers have missed. The collaborative aspects of the company culture are reinforced: everybody is involved and everybody is responsible for the quality of the product.

Excosoft partners also participate in this testing.

When a fault is discovered a fault report is written. This report contains a detailed description of what went wrong and the editing context in which it occurred. It is essential that it be possible to reproduce the failure. Faults are collected in lists. If the person who detected the fault knows who is responsible he may contact the designer responsible directly. Designers monitor the fault lists and correct faults that fall within their particular areas of responsibility. The fault lists are also used to collect improvement proposals. Corrected faults and introduced improvements are moved to another list which also contains links to the modified code.

Excosoft products run on several platforms. Excosoft personnel use these different platforms in their daily work to ensure that cross-platform testing is effectively carried out. However, as it is not possible to include all supported platforms in these daily test routines a period of two weeks is reserved prior to the external release. During this test period the main focus is on non-typical platforms. Development at this stage is limited to fault correction and code reviews of all modifications.

Designers perform a basic functionality test prior to merging new code into the main development line. This testing is ad hoc, and its extent depends on the magnitude of the change introduced.

7.2.4. Project environments

Excosoft is a small company engaged in small-scale projects and does not require a formalized development process. The only documents produced in-house are customer manuals, project specifications and project follow-up reports. Code documentation is intertwined with the code itself following the precepts of literate programming. Thus the code documentation itself is rebuilt on a daily basis.

Project responsibilities are distributed among designers according to the technical functionality of the system., i.e., a particular designer may be given overall responsibility for the function that generates Postscript files. The problems of interfacing functions owned by different designers are solved by direct communication. This is straightforward as projects are small-scale. The problem of delivering consistent code to the builds is also solved via direct communication.

7.2.5. Advantages and disadvantages

The advantages of the daily build approach at Excosoft are:

- Increased project control
- Quality is built into the product over a long period
- Designers are highly motivated

Excosoft has not experienced any disadvantages using the daily build approach

7.3. IAR Systems

IAR Systems develops software development tools, i.e., C compilers, for a wide variety of embedded micro-controllers. The development at IAR Systems is divided into two departments:

- Core Technology Division
- Target Projects Division

All IAR Systems products are built on a number of common components that are reused in all projects. The Core Technology division is responsible for maintaining and modifying these common components and the Target Projects division is responsible for developing new products based on the common components. The divisions use different types of development processes.

This part of the report focuses only on the Target Projects division of IAR System since this is the group working with daily build.

The projects running at the Target Projects division are rather small, with two to three designers each. The lead times of projects are between 12 and 15 months. About 85% of the code in the products consists of code from the common components that are reused from earlier products.

IAR Systems did not fully use daily build at the time of this interview. However, they been using automatic testing for a long time. Since the company is considering implementing more from the daily build concept this report about IAR Systems captures the organization's experience from automatic testing and how they plan to tailor daily build to their development.

7.3.1. IAR Systems Development Process

The IAR Systems development process is similar to the waterfall process. It is divided into five main steps [see Figure 9].

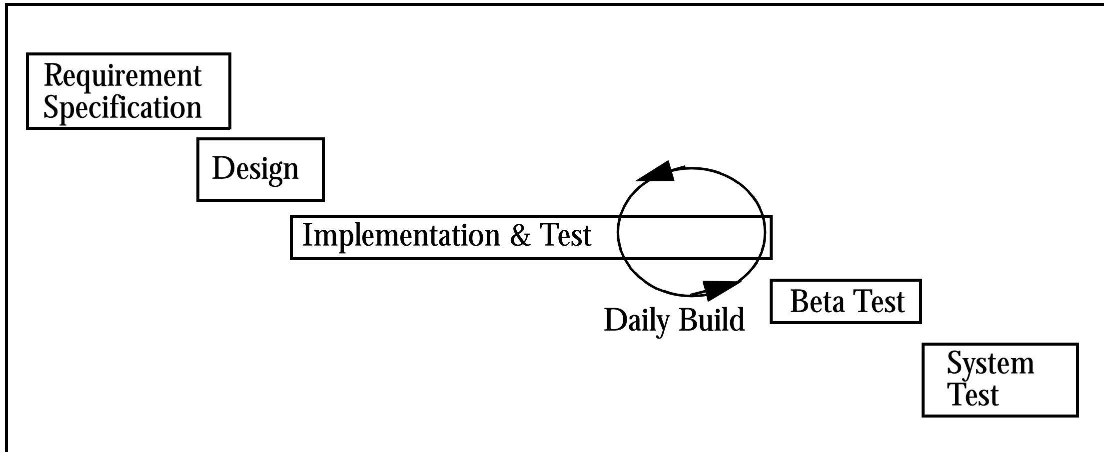


Figure 9. IAR Systems Development Process

In the later part of the implementation-test phase, IAR Systems starts to use daily build and automatic testing [see Figure 9]. In this phase, they mainly correct failures; i.e., little new functionality is added.

Since the projects at IAR Systems involve only two to three designers each, and every designer owns his own code, a designer does not modify code written by another designer. Furthermore, it is not stated anywhere in the process that designers must use daily build. It is up to the designers to decide when to do a build and test. Usually, they do it a couple of times per week in the later stage of the implementation-test phase. The designers do the build and test themselves, making it easy for them to handle the daily build loop - they do not need a process for handling it.

The reason the daily build loop starts so late in the project depends on two things:

- The reused code is not consistent in the earlier stages
- The new implemented functionality is not implemented in any defined order, therefore it is not consistent and no test-cases exist

The principal result of these two problems is that IAR Systems does not get any consistent code to test until late in the implementation-test phase. The code has to be consistent to do any testing of the functionality, but at IAR Systems, where about 85% of the code in each project is reused from earlier projects, this code is not consistent. There are no dummies to simulate those parts of the code that are not yet implemented.

The problem with new implemented code is that there is no defined order for the implementation. All the new products that are developed at IAR Systems are similar; i.e., the functionality that is to be implemented is often the same, but developed for different chips. Each project starts to implement this new code in a different order, meaning the project does not follow any defined order for what sub-functionality to implement first. Sub-functionality can be implemented in various sequences [see project A and B in Figure 10]. Since the new code is not implemented in any defined order, each project has to write specific test cases and these test cases cannot be reused by other projects. It costs too much to write test cases when they cannot be reused in other projects, so no test cases are written for the sub-functionality and thereby no automatic testing is used on the sub-functionality.

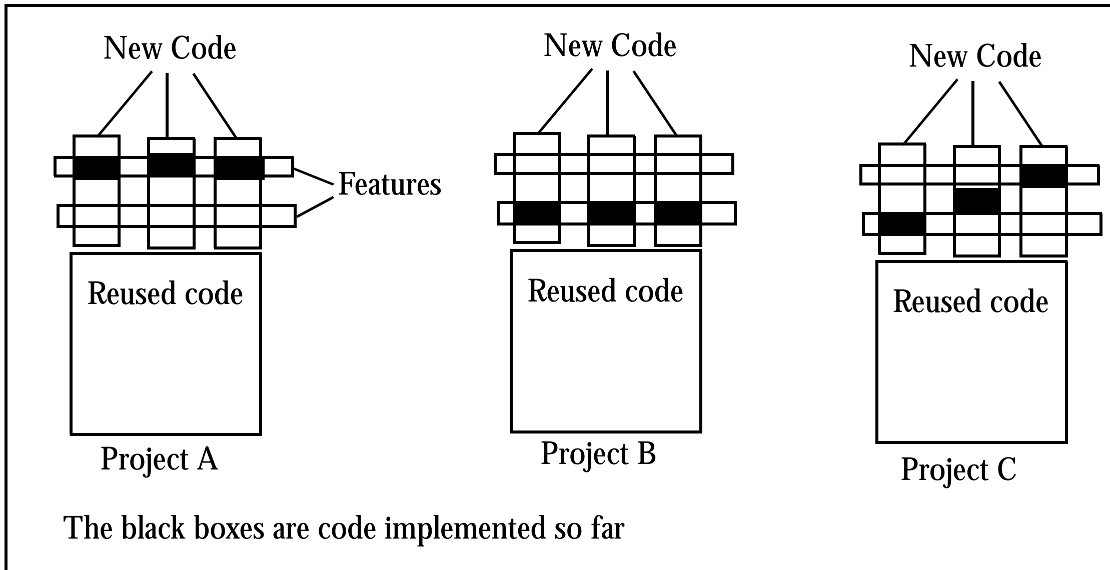


Figure 10. Examples of how to Implement the new code

If they implement the new code divided over many features, as in project C in Figure 10, they will also have the problem that the new implemented code will not be consistent.

7.3.2. IAR Systems Automatic Testing

IAR Systems has used automatic testing for about 7 years - a considerable amount of time, but this is because of the type of product they develops. All C compilers must fulfill standards and in order to confirm that they do, test packages are available. Purchasing these test packages means one also receives a simple test engine that executes the test package automatically.

IAR Systems started automatic testing with such a test package and test engine. Today the company has built its own test engine and develops new test cases when implementing new

functionality. Now IAR Systems has a large number of automatic test cases that all take about a week to execute. All of the test cases are executed just before the project enters the Beta-test phase.

When IAR Systems uses automatic testing in the daily build loop, not all test cases are executed. It is up to each design team to decide what test cases to include in the automatic test. Usually designers execute the most common use cases, as well as the use cases where they optimize the compiler as much as possible (i.e., the most complex use cases for the compiler). In the daily build loop, the designer also adds new test-cases after adding new functionality in the system. If these new test cases are common for all compilers that IAR Systems develops, they are stored together with the rest of the test cases and executed in the larger test prior to Beta-test.

When a designer wants to execute the automatic test on his product he does this on his own computer. Upon completion of the test, the test engine generates a text file with the feedback from the test.

IAR Systems has one person responsible for the test engine and the common test cases.

7.3.3. Gaining Benefits from Daily Build in IAR Systems' Development Process

IAR Systems has the ambition to shorten its lead time. Therefore the company wants to start using the daily build loop at an earlier stage. To be able to do this, IAR Systems must produce consistent code earlier in the project.

As described above, IAR Systems has encountered problems with consistent code - both for the parts that are reused and for new code. For reused portions, the company intends to build dummies to make this code act in a consistent manner.

To solve the problem with new implemented code, the organization wants to develop a model describing the order in which the sub-functionality will be developed. By doing this designers can also write common test cases that can be used in many projects to test the sub-functionality. As a result, IAR Systems will be able to do automatic testing on the sub-functionality.

If IAR Systems succeeds in combining these two solutions, designers should be able to develop consistent code much earlier, thereby enabling earlier use of the daily build loop.

The major benefits that IAR Systems expects to gain by using daily build earlier are:

- Shorter lead time
- Better tracking of progress in the project

- Higher quality

7.4. ABB Automation Products

ABB Automation Products (in this chapter referred to as ABB APR) develops automation products and systems used for protection and control in industrial plants.

7.4.1. Product Structure

The structure and architecture of the ABB APR products must be known in order to understand how ABB APR applies the daily build process. This chapter provides a brief description of the systems that ABB APR develops and the architecture of these systems.

ABB APR develops several products, and applies daily build on the product level; therefore, this part of the report will focus on the development process for products.

The products contain a number of functional sub-systems (FSS) which in turn contain a package of features. The products are developed by combining these FSSs. In this way, the FSSs can be viewed as components.

In general the ABB system consists of clients, servers, controllers and field equipment connected by networks as shown in Figure 11. Functional subsystems can consist of functionality in all of these nodes. The figure is simplified and describes a general view of an ABB APR product and the functional sub-systems.

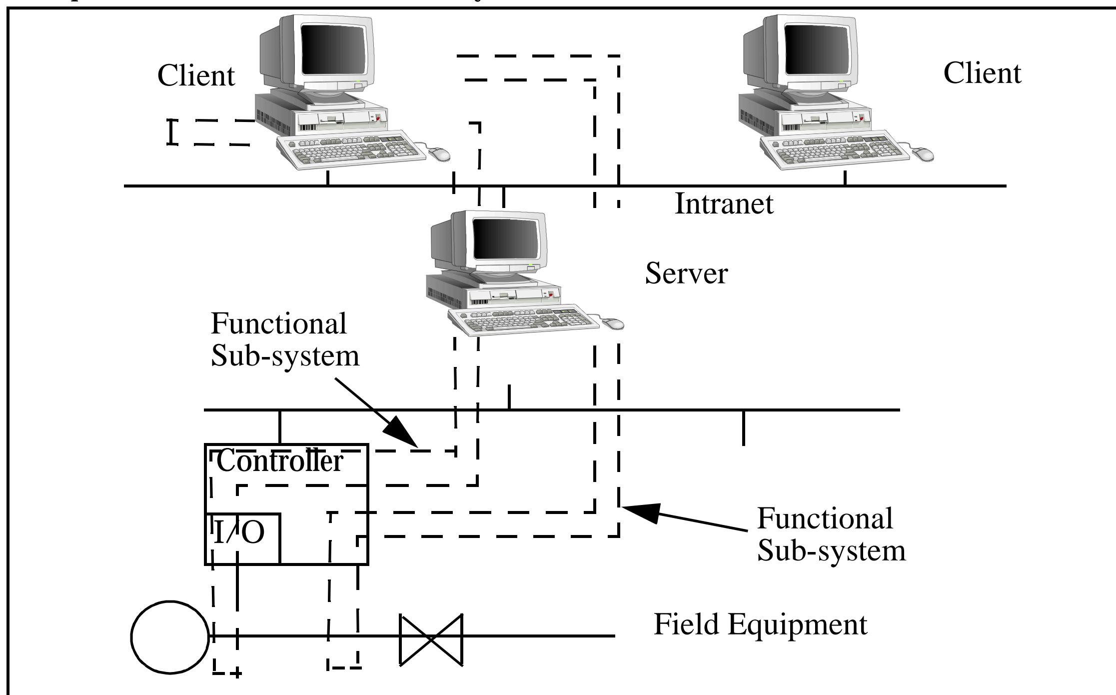


Figure 11. A General View of a ABB APR Product and the Functional Subsystems

7.4.2. Project Organization and the Daily Build Approach

When a new product is to be developed, a Product Project is initiated. The Product Project is responsible for building the product (by combining the FSSs), documentation and marketing, as well as other projects tasks. The relationship between the Product Project and the FSS-projects can be seen as a prime/sub-contractor relationship.

Together with the market division, the Product Project writes a requirement specification for the new product. This results in one requirement specification for each of the FSS-projects that will deliver to the Product Project. Customer requirements on FSS level are also identified by the FSS teams themselves. Based on the requirements collected, a plan for the project is made that states when and what the FSS teams are to deliver to the product project.

Daily build is used by the Product Project, which builds what FSS-projects are delivering on a daily basis. The Product Project is divided into several of sub-projects. One of these is the integration project responsible for the integration and the daily builds. The Product Project does not develop any software itself; however it builds the system from the software developed by the FSS-projects. Each FSS team reports to the Integration sub-project responsible regarding what files to include in the daily builds [see Figure 12].

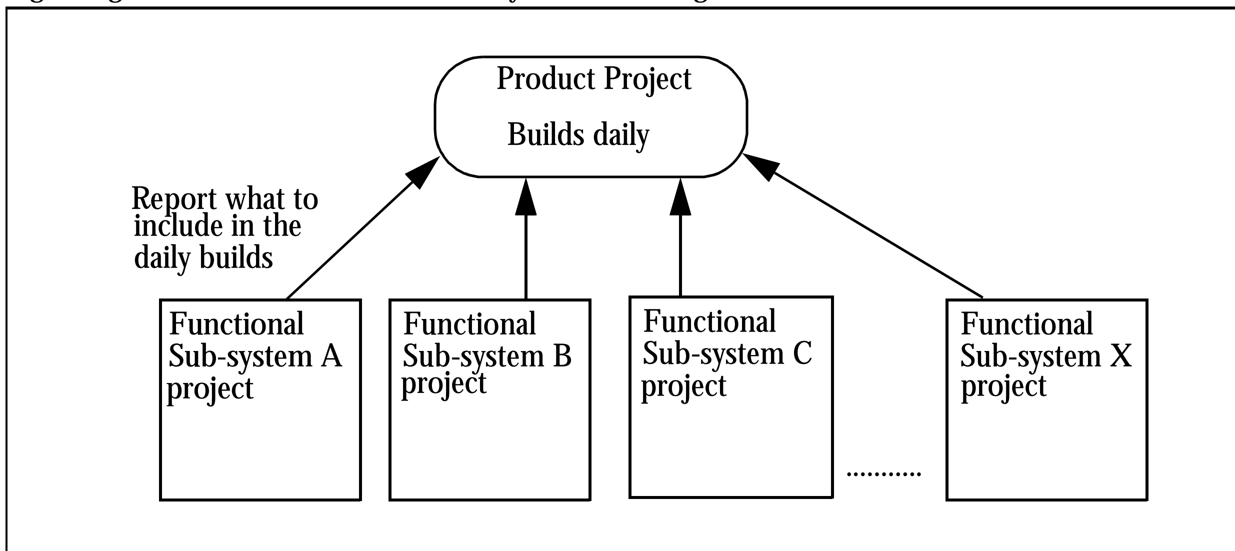


Figure 12. Relationship Between Product Project and the FSSs

This approach is similar to that of a smaller project using an integration group.

At the beginning of the Product Project, the integration group builds twice a week. The build frequency increases so at the end of the project builds are made more than once a day.

To make this approach work, the FSS-projects must deliver consistent code. ABB APR has solved this problem by combining feature responsibility with proper planning.

One team is responsible for each of the FSSs. The leader of the FSS team has the final responsibility for all the features in the FSS. The team leader delegates this responsibility so one person is in charge of each feature in the FSS; this means one person is responsible for each end feature in the FSS. This person must coordinate to ensure that consistent code is delivered.

Generally, all the code for a end feature is developed by one FSS team. In this case, it is not a major task to coordinate and deliver consistent code, but if the feature goes cross-functional over multiple FSSs, the situation becomes more complicated. It is the person responsible for the feature that must coordinate with the other FSS projects to ensure delivery of consistent code. The person responsible for the end feature usually works in the functional sub-system team that affects the feature most.

The work of the feature responsible is facilitated by a Feature Plan formed in the beginning of the project; the plan specifies when different features should be delivered, as well as short term plans made each day that determine what features to include in the builds two days ahead.

ABB APR tries to handle external sub-contractors in the same way as the FSS projects, but it is difficult to persuade an external sub-contractor to deliver code on a daily basis. Nevertheless, ABB APR tries to involve the code from sub-contractors as often as possible.

7.4.3. The Merging Process

The FSS teams owns the Main Code Line for its FSS which means that the Product Project does not have its own Main Code Line, but the project is informed of what code to include in the builds. The project is also responsible for saving the old builds so they can be re-created.

Each FSS contains a number of modules and files. These modules and files are owned by the FSS team and no other team modifies that code. Each file in the FSS has a person responsible for the file, and normally this is the only designer who modifies this code. If a designer has to modify something in a file owned by another designer, this is coordinated in order to ensure that not more than one designer modifies the same code at the same time. This coordination significantly reduces merging problems, eliminating the need for a well-defined merging process.

7.4.4. Test Strategy

When the FSS team members inform the integration group what to include in the daily build, they also tell the group about new functionality implemented in the product and what test cases the test group should use to test the new build. The FSS teams execute some test-

ing themselves before allowing the code to be used in the daily builds. They do this to ensure that their new code will not break the build and has the right functionality. The FSS teams use old builds when executing this test.

When a build is successful the integration group brings together people from the Product Project that can do the testing. The testing team executes the new test cases and some old basic test cases as well as a small amount of regression tests. The test cases executed to control old functionality are few and chosen in an ad hoc manner. ABB APR does not use more regression testing since it does not apply any automatic testing that would require extensive effort.

When the Product Project has achieved a stable quality in the builds, ABB APR enters a more elaborate test phase called Product Test, where the test group executes all the test cases for the product. In Product Test, daily build is still used, but no new functionality may be added. It is only allowed to modify the code to correct failures.

After the Product Test ABB APR performs System Test. At this point the functionality in the entire system, together with other systems, is tested.

When a failure is found during testing the test group writes a fault report, identifies the FSS team responsible for the failure, and reports the failure to the team.

7.4.5. History

At ABB APR it was never clearly stated that the organization should use daily build. Instead, the demand came from the projects that felt that something must be improved.

Daily build was first used in a project that had been working with specifications for over a year. The persons working in the project felt nothing was happening, and that they needed to do something about the lack of progress. They started to write code instead of writing specifications. The 10-15 persons in the project worked closely and started to discuss and implement use cases. By this time, a decision was made to use FSSs as the basis for the system architecture. Group members began to see progress in the project and they started to focus on the right problems instead of assumed problems. After working with this approach for some time, the group had implemented prototyped functionality of the system. However, the structure of the software in the prototype was not sufficient, and the group was forced to rewrite the code with a better structure. Even with this setback, project members are convinced it was very helpful for them to start code in an early phase.

The next step was to implement the code in the different FSSs in parallel. This method brought no progress so the designers decided to integrate the whole system - and discovered that they had major integration problems. They also discovered that they had been solving the same problems at different places and that communication between teams was insufficient. After this the teams decided to build on a more regular basis.

The next step was to define "missions" for the project. This was a plan presenting what functionality to include the next time the system was integrated. At this stage designers built the system once every four weeks and soon they realized that this was good for the project and that it kept up the pace. As a result, they started to integrate the system more frequently and introduced the integration team that was building the system and gave feedback to the designers.

However ABB APR has not fully implemented all components of daily build since they still use a lot of manual testing.

7.4.6. Benefits and Hindrances

The major benefits ABB APR has gained from daily build are:

- Provides a high pace in the project
- Better tracking of the progress in the project
- Better communication and cooperation between the FSSs
- Focus on the right problems
- Failures are found earlier

Daily build has changed the status of the code. When using the daily build, the code is the most important part while the design documentation is not that important any more. This means the design documentation must not possess the previous level of detail any more.

The main hindrance they have had when implementing daily build was to create a good model for how to do the integration automatically.

7.5. Telelogic

Telelogic develops software tools that provide a graphical development environment for communicating real-time applications, such as telecom applications.

The projects that Telelogic runs involve about 35 designers, 10 subcontractors, and 10 testers. The projects mainly develop new releases of the existing product. The latest release of the product is used as input to the next project, and the lead time for the projects is about nine months.

7.5.1. Product Architecture

Telelogic develops a package of software tools called Tau that contains the three products: ORCA (analysis), SDT (design and implementation), and ITEX (testing). These cover the major parts of a development process. Each of these products contains commercial off-the-shelf software tools developed by Telelogic, e.g., SDL Editor, SDL Analyzer, and SDL Code Generator, and these are integrated into Telelogic Tau.

Figure 13 illustrates the physical view of the architecture of Telelogic Tau. The central part of Tau is a postmaster working as a client-server, handling all the communication between the different tools in Telelogic Tau.

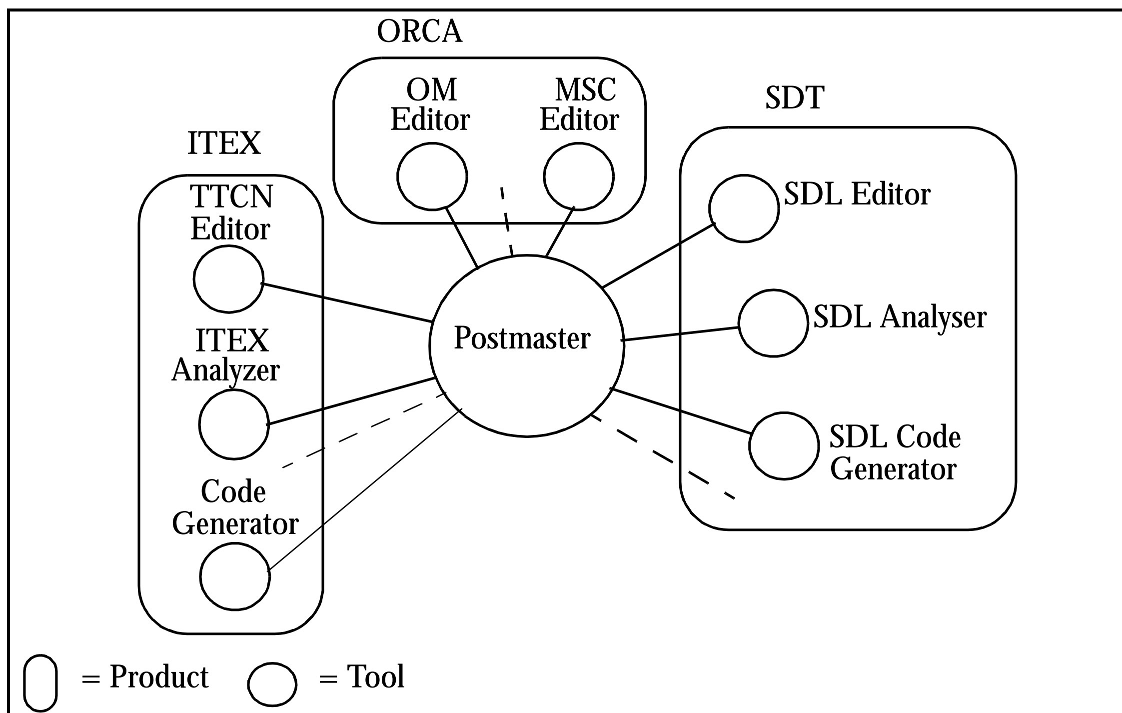


Figure 13. General View of the Telelogic Tau Architecture

7.5.2. Project Organization and the Daily Build Approach

Telelogic combines incremental development, weekly build and daily build in its development process, i.e., they build regularly with three different frequencies [see Figure 14]

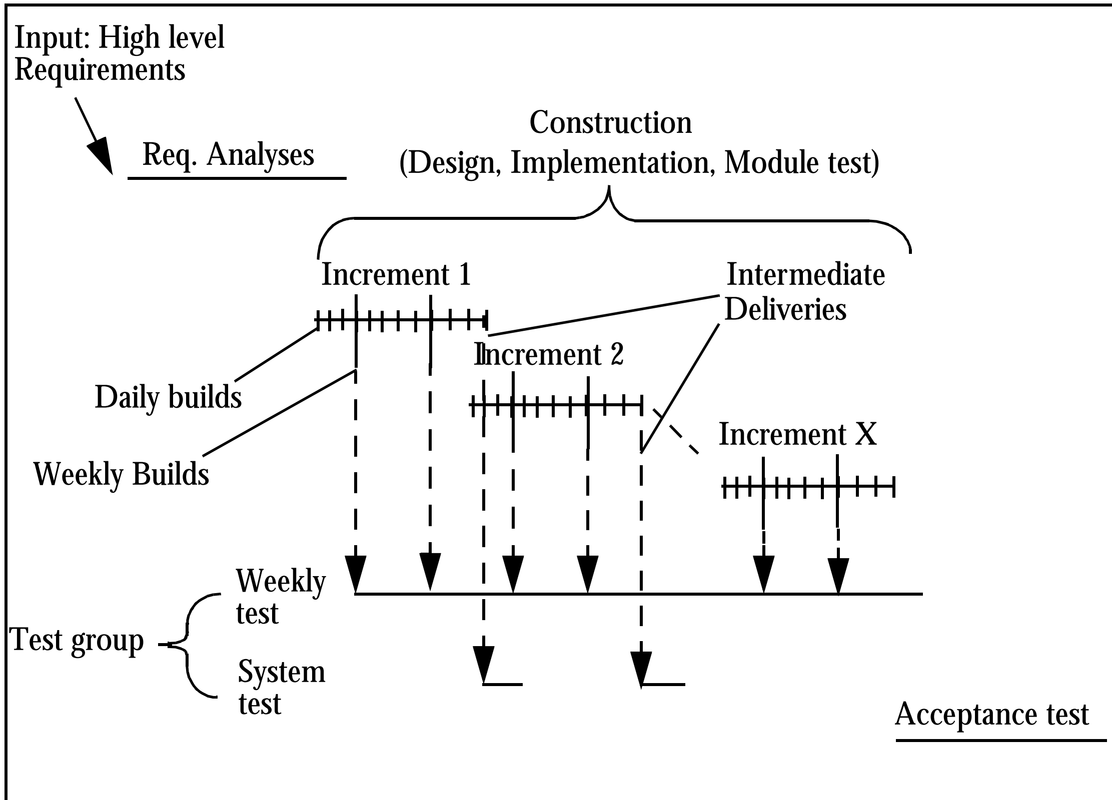


Figure 14. A Simplified Description of Telelogic's Development Process

It is the products that are built both on a daily and a weekly basis [see Figure 13]. However the daily and weekly tests are performed on the tool level. The only difference between the daily and the weekly approach is the scope of the testing (more information is available about this in the next chapter).

The builds are made by the Configuration Management Group (i.e., integration group) that includes two persons. The group uses ClearCase to generate the entire software product from the code in the main code line. It is up to each designer to check in new code to include in the build. The designer checks in code once it compiles without any problems and the designer thinks the code is free from bugs.

The Configuration Management Group works with a configuration management plan; however, this plan is not so detailed that it guarantees new functionality or code is included in the builds every day. Theoretically designers can build the exactly same system more than

one day in a row. But in practice several check ins are done every day, and only some code parts are normally built with no changes since last build.

The code has to be checked in before 16.00 on Thursdays to be included in the weekly build. Then the integration group does the weekly build. If any problems arise in building the system - i.e., it could not be compiled - the person responsible must fix this the following Friday morning. Then the build is made again and distributed to the test group.

Telelogic applies a hierarchical responsibility: one person or team is responsible for each tool, one person or team is responsible for each module in the tool, and one person is responsible for specific functionality in a module. For example, one person is responsible for all editors, and one is responsible for each of the editors (such as the SDL editor or the MSC editor).

Telelogic also assigns persons responsibility for interfaces between tools. No one, except for the interface responsible, is allowed to make changes in the interface.

The company recently named a person responsible for the architecture in the system.

Telelogic is not required to deliver consistent code to each daily and weekly build. Since the company performs only module test in the daily and weekly test, it is first after the increments that the features in the system are tested. Ensuring that consistent code is delivered to each increment is done by planning what functionality to include in each increment.

7.5.3. Automatic Testing

The builds made on a daily basis are tested by the designers themselves.

The amount of test cases automatically executed depends on the tool that is to be tested. Some tools, such as SDT analyzer and SDT Code generator, have a large amount of automatic testing while others, e.g., the editors, have a very limited amount of automatic testing. The reason is that it is difficult to perform automatic testing for editors possessing a high degree of editing, since it is not easy to compare the actual outcome with the expected outcome. Implementing automatic testing on the editors is the major hindrance that Telelogic must overcome to apply daily build on all its products.

It is up to the designers to decide what test cases to execute on the daily build. Usually, the designer chooses some test cases to test the new functionality in the tool and some for regressions test. It is also up to the designer to write new test cases when adding new functionality. Faults found in the daily tests are printed in a logfile.

The builds made on a weekly basis are tested by the test group. The tests performed on the weekly build are much more extensive than in daily testing. In the weekly test, all faults found by the test group in the weekly tests are reported directly to the designer responsible.

After a successful weekly build and test, the build is delivered to the whole company. It is then used by the designers to test the functionality in the new implemented code. The build is also used by persons writing manuals to verify that the product executes as described in the manuals.

The system tests after each increment are performed by the test group. The test is made as black box test and the features in the system are tested, i.e., functionality that uses many tools are tested. The test is very accurate and the idea is that after this test, the increment can be delivered as a product. This test is carried out on code that is frozen when the increment is delivered. This means if a failure is found during this test it has to be corrected both in this code line and the code line that is used for the next increment.

The whole product family, i.e., the interfaces between the products, is also tested in the system test after each increment.

7.5.4. History

Telelogic started to use daily build one year ago. The first ideas came from a project working with the SDL Analyzer. The company already had some automatic testing and had realized the importance of early test. The idea spread in the company and soon it was introduced in all the projects. At the same time, ClearCase was introduced as the CM-tool at Telelogic. This made it possible to handle the CM of the files in the structured way required by daily build.

7.5.5. Benefits and Hinders

The major benefits that Telelogic has gained from daily build are:

- Failures are found earlier - fast feedback
- Better tracking of the progress
- Higher flexibility

Telelogic has one potential problem with its adoption of daily build. Since it is the weekly builds that are used by the designers for testing and prototyping their new implemented functionality, designers do not use the latest versions from the other teams. The reason they do not want to use the latest versions is that they cannot be sure that these are stable.

For example, the following scenario involves two teams: Team 1 is working on Tool 1, and Team 2 uses Tool 1 when prototyping the functionality in Tool 2.

Week 1: Team 1 is working on Version 2 of tool 1. Team 2 uses Version 1. Team 1 does not include Tool 1 in the weekly build.

Week 2: On Monday, Team 1 is finished with Version 2 and starts developing Version 3. Team 2 is still using the build from last week, i.e., Version 1.

This can cause problems when Team 2 tests its functionality in the newer versions of Tool 1. However, it is not a major problem, and it can be solved with good planning and effective communication between teams.

7.6. SAAB Military Aerospace

Interviews were conducted with two different departments in SAAB Military Aerospace. Neither department uses daily build. However, both departments make regular builds of their products. This report about SAAB will capture how the departments make their regular builds and what focus they must have to implement the daily build concept completely.

In some areas this part of the report is divide for the two departments; in some areas the two departments are described together. The two departments are:

- Gripen systems.

  Projects at this department involve around 50 designers. Gripen Systems works with incremental development and the lead-time for a increment is approximately 12 months. Gripen systems delivers to the customers approximately once every second year.
- General Military Programs.

  Projects at this department contain about 10-15 designers and the lead-time between internal releases is approximately 3 months. The lead-time between deliveries to customer is approximately 1-2 years.

Both departments develop software for military aircraft, but for different types of aircraft.

7.6.1. Physical View of the Product Architecture

The physical view of the product architecture is similar for both systems. It contains a number of Computer Systems (CS) that communicate [see Figure 15]. Each CS handles different

tasks, for example there is one CS handling the flight control system, one handling the display system, one for the radar system and so on.

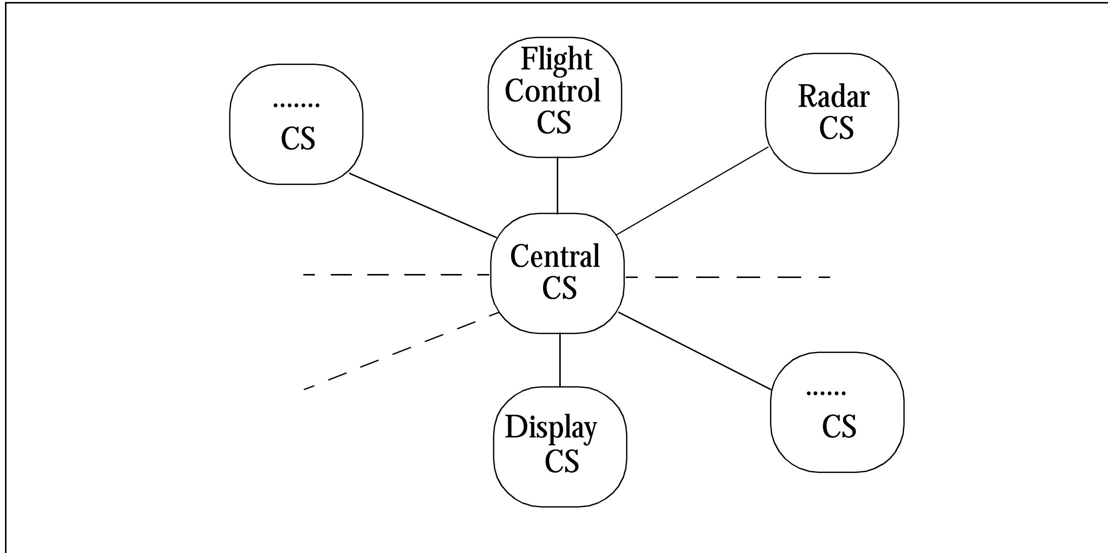


Figure 15. The Physical View of the Architecture

SAAB develops the central CS and some others CSs. The rest of the CSs are developed by sub-contractors or partners.

7.6.2. Gripen System's Development Process

The projects at Gripen system are mainly developing new releases of the existing product. The latest release of the product is used as input to the next project. Each new release of the product is divided into a number of increments that are sometimes executed in parallel. Each of these major increments are handled as a sub-project. Later increments handles new features and correct failures from the earlier increments [see Figure 2].

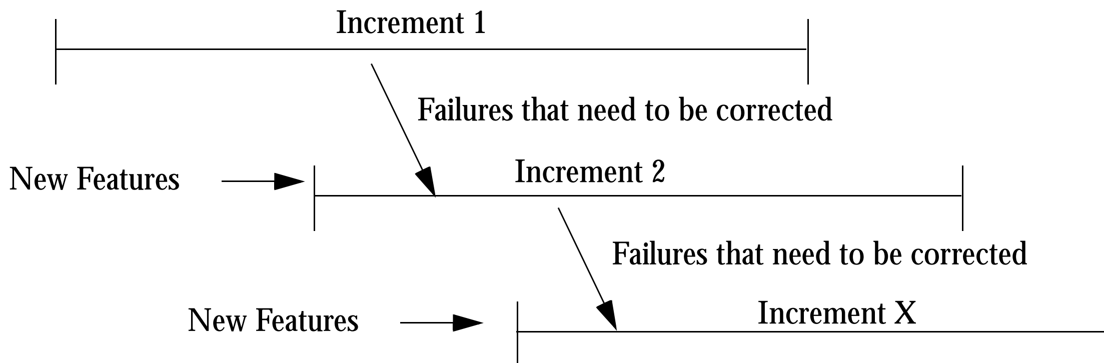


Figure 16. The Projects are Divided into Increments

Each increment follows the development process [see Figure 3]. Each sub-project divides their major increments into smaller increments (increments of the increment) [see Figure 3].

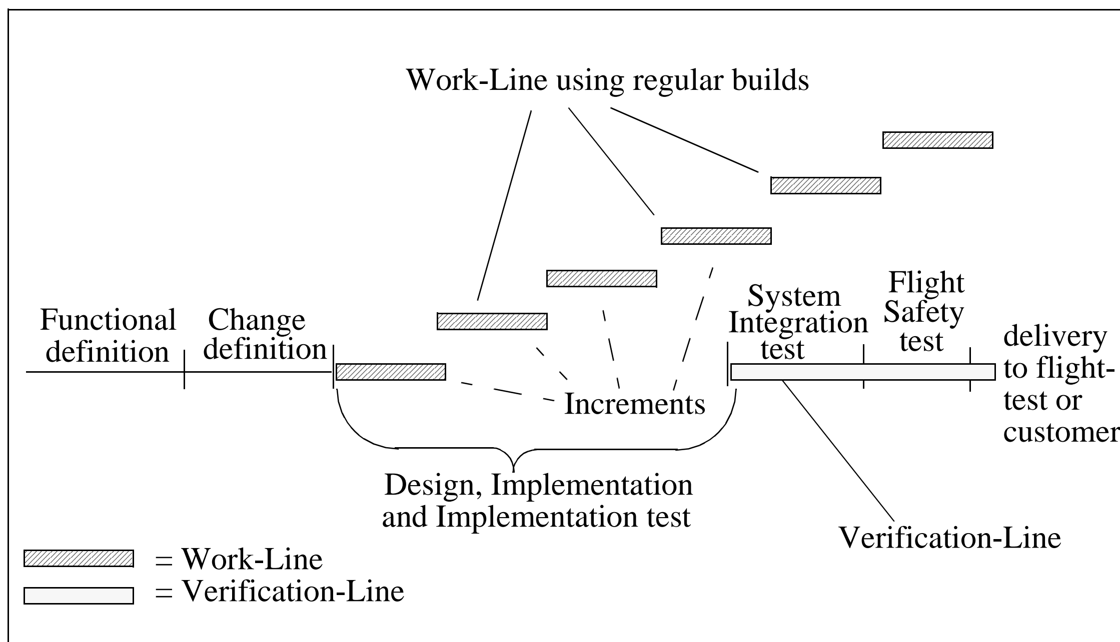


Figure 17. Gripen System's Development Process used for each Increment

During the functional definition phase the project decides what functionality should be implemented in this increment (requirement specification). In the change definition phase, change requests from earlier increments are analyzed and it is decided if they should be handled in the current increment. Gripen System is working with two different lines of code, the working line, where all new code is implemented, and the verification line where all the functionality is verified. As shown in the picture the work line is executed in parallel with the verification line during the later phases. Needed corrections because of faults or requirements to support the flight test are first implemented in the work line and then merged over to the verification line.

When the increment is finished it is delivered to flight-test, where it undergoes additional testing. This might be an iterative process where flight-test finds failures and report them back to the project, the project then corrects the failures and deliver the updated result back to flight-test. Once the last increment, increment X in figure 16, is finished and tested in the flight-test, the product is delivered to the customer.

Eighty percent of the lead-time for each increment is spent in the design-implementation phase and the implementation test phase. These two phases can be seen as one phase since design, implementation and implementation test are performed iteratively. During this

phase, the system is built regular. In the beginning, the system is built on a weekly basis (once per week). The frequency of the builds increases during the project, and at the end of the project the system is built on a daily basis. The different increments are well defined, regarding functionality and end dates. But no rules or policy is established for when to build the system or how often to build the system during an increment or what functionality to include in the different builds during an increment. The individual designers decide when their new code should be included in a build. Designers who want a build containing their new code check in their code, coordinate their delivery with other designers (if necessary), and contact the group building the system. Before the code is checked in, the designer has to verify that his new code compiles and pass own module test. The build group consist of two persons.

The builds at this phase initially only includes the CSs developed by Gripen systems. During the implementation test pre-deliveries from sub-contractors and partners are often included, but not in a formal way. The CSs that are developed by sub-contractors and partners are formally involved during the System Integration Test. To be able to cooperate with the sub-contractors and partners CSs in the builds, the interface and the communication between the CSs has to be well specified in an early state. Changes to the interfaces has to be synchronized in both ends.

7.6.3. General Military Program's Development Process

The projects at General Military Programs principally develop new releases of the existing product. The latest release of the product is used as input to the next project. They have the same phases in the General Military Program's process as in the Gripen system's process. However, small differences can be seen in how the two groups perform their regular builds. The General Military Programs department also works with two different code lines, the work line and the verification line [see Figure 18]

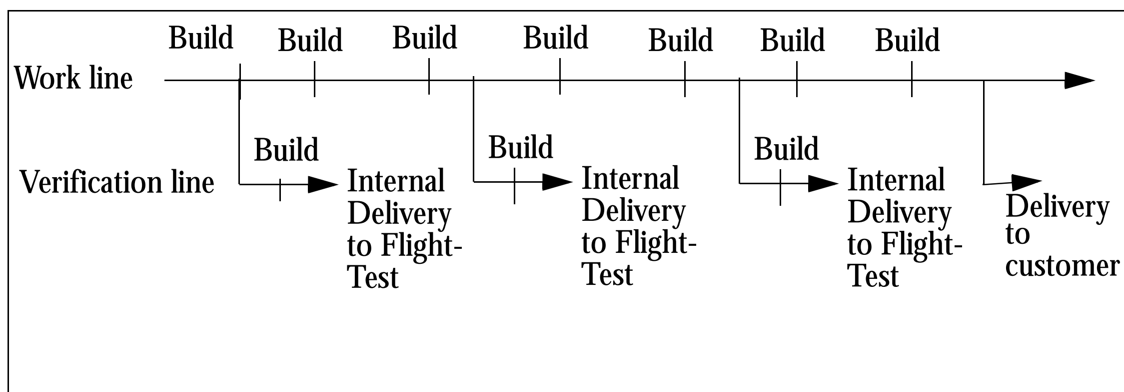


Figure 18. General Military Program's Work line and Test line

All new functionality is added in the work line. The verification line is created about a month before a delivery. No new functionality is added to the system in the verification line.

In the verification line, they perform more comprehensive testing and correct failures. When the verification line is created, work on the work line continues in parallel.

Regular but very informal builds are used at the work line. It is up to each designer to decide when to do a build and what to include. Designer does the build themselves, i.e. there is no build group. The builds are personal, so there is no common build of the system that many designers use. Each designer that needs a build must make it personally.

7.6.4. Testing Strategy for Gripen Systems and General Military Programs

This chapter describes the testing strategy for both Gripen systems and General Military Programs.

At the time of this report, testing of the regular builds during the implementation phase is done in an ad hoc manner. After each build, each designer that checked in new code for that build receives a copy of the build (or designers make the build themselves, in the case of General Military Programs). Then the designer has to load this build into the simulator and perform the testing. The testing in the simulator is not automatic - the designer has to execute test cases manually. Not all builds are tested, some of the builds are done just to check that the system links and compile without failure. As mentioned before, CSs developed by sub-contractors are not formally included in the builds during the implementation test phase. However, the builds are tested in the simulator against CSs developed by sub-contractors. The CSs in the simulator are often earlier versions of the CSs and not the versions that the sub-contractors are developing at the moment. The testing against older versions of the CSs make it possible to ensure some of the functionality of the whole system.

In the system integration test phase the test group starts to handle the testing in a more structured way. No automatic testing is used in this phase either, but some regression testing is performed at this point.

The main reason that SAAB do not have any automatic testing is the type of application the company develops. The majority of the testing is done in a simulator and therefore must be executed manually.

7.6.5. Applying Daily Build in SAAB Military Aerospace's Development Process

To apply daily build effectively, SAAB Military Aerospace has to change two major things in the way it produce software:

- Find a way to deliver consistent code to the builds, including how to involve the sub-contractors

- Tailoring the build process in an other way.
- Start with automatic testing

The first problem is to structure the build process, and two aspects are important here. How to deliver consistent code internally to the builds and how to involve the sub-contractors. First Saab must create some policy regarding when to build, so the system will be built regularly. This task also includes exploring how to deliver consistent code to the builds. Today Saab works with module responsibility and it is rare that designers change or add code in other designers' modules. Two different methods are available to handle the code consistency problem: start to use feature teams that coordinate what is delivered to the builds, or keep on working as today, letting the persons responsible for the modules, plan and coordinate when and what to deliver to the builds. The way they works today requires regular meetings and a lot of coordination. If the projects grows, in number of designers and functionality, it might be hard to handle the required coordination, without implementing feature teams.

The second aspect involves the sub-contractors. Since major parts of the system are developed by sub-contractors, SAAB has to involve the software developed by the sub-contractors in the builds as regularly as possible. Otherwise the company will lose some of the advantages with daily build.

The second problem is the automatic testing. If SAAB does not have any automatic testing it will cost to much to follow up and test the builds. SAAB has started to look into how they can start using automatic testing. Saab plans to introduce the automatic testing in a stepwise manner. Today they use the automatic testing on modules, i.e. black-box testing. Next step will be to integrate modules into functional sub-systems and introduce automatic testing on those sub-systems. The last step will be to introduce automatic testing on system level involving the flight simulator.

The major benefits that SAAB expect to gain if they start to use daily build in a more structured manner are:

- Find failures earlier
- Find integration problems earlier
- Shorter lead-time

7.7. Ericsson: GPRS Development for the BSC

The General Packet Radio Service (a faster access to the internet from a mobile phone) requires some changes in the BSC (Base station controller) node in the GSM network. The project implementing these changes have used daily build.

7.7.1. Organization

As with many other Ericsson projects, this project is also distributed over several sites. The project is led from Linköping, but design is also done in centers in Karlskrona, Hässleholm, Ireland, and the UK. The project is developing a limited part of the BSC system which is rather self-contained, and it can be tested in isolation. The part of the system developed is new, thus it has no design base. The overall GSM project has planned three larger deliveries for integration and testing on the complete GSM network level.

Each design center is responsible for a set of files; each center performs all work with its respective file set.

7.7.2. Construction Planning

The GPRS project relies on a rather detailed construction plan developed by an experienced system expert. The construction plan specifies deliveries from each of the design teams (each delivery takes about two weeks to implement), and the deliveries are sequenced so that they can be integrated and tested as they are completed by design. The deliveries to each integration are whole or parts of whole functions that are testable from an external point of view. Note that several design teams can contribute to a delivery.

A skeleton of the total system was made before the real development of the functionality started, so that all the modules in the final system were defined and the basic mechanism of the system was working.

7.7.3. Build and Test

Build occurs automatically with a Unix daemon that takes the code from a Clearcase repository. The code in Clearcase is marked with three labels:

- Baseline: This is the latest stable version of the code
- Temp_Baseline: This is the latest checked in version which needs to be verified. When a build is successful, the Temp_baseline label is changed to Baseline
- Use_This: This label is used for shared files where the changes need to be propagated to the complete project as soon as they have happened

The separate PUT (Process and Unit Test) team is responsible for testing the delivered functionality in each of the builds. This is because the functionality can consist of parts from different design teams, and some independent testing is needed. The design teams are responsible for testing their files in isolation before delivery. Note that the PUT team is testing the new functionality, as the old functionality is tested by the automatic regression test (part of the daily build). As the functionality is tested successfully new test cases are added to the automatic regression test.

The delivery to integration is followed by a delivery report where each design team specifies in more detail what is delivered. The documentation is also updated for each delivery.

Trouble reports are sent back to the responsible design team. If it is necessary to correct a fault, a fault branch is made in Clearcase where all faults on this build are corrected.

The result of a build is distributed via SMS to mobile phones of interested parties. The detailed test results are also stored in log files for later analysis.

7.7.4. Evaluation

The following are advantages of the daily build process as implemented in this part of Ericsson:

- When using daily build, the design activities can be done more in parallel, and lead time can be cut.
- With a shorter delay between design, integration, and test it is simpler to isolate and correct faults, thereby decreasing the test time.
- Tests are run by an independent team, which is a good testing practice.
- There is always an executable version of the code available.

7.8. Ericsson: OSS Development in Mölndal

Part of the Operation and maintenance system (OSS) for the GSM network system is created in Mölndal at ETX/A/PO. This group has used feature teams in connection with daily build. The project successfully made the deliveries to system test on time with contents and quality according to the agreed requirements. Feature teams and daily build were key to this achievement.

The scope was to introduce new features in an existing product. The design activities were performed by nine designers organized in three design teams.

7.8.1. Feature Teams

The design teams worked with feature responsibility. Each team got the goals to plan and introduce a new feature throughout the phases design, implementation and basic test. A feature consisted of a set of requirement that formed functionality of value for customer, e.g., table tabs. The impact on the products for a feature varied from one to quite a number of software units. The development was made in increments where the design teams delivered two or three increments each. The increments were gathered into five deliveries to function test. The time between the deliveries varied from one to seven weeks.

A management team, consisting of the project manager and the line manager, acted as an "orderer" to the design teams. The main responsibilities for the management team were:

- defining the goals for the teams
- approving the team plans from the teams
- following up the results from the teams
- resolving any conflicts in project resources and external resources concerning the teams

The responsibilities for a design team included:

- division of the feature into suitable increments for delivery to function test
- detailed planning of all activities required to fulfill the team goals. The activities were documented in a team plan which formed a "contract" between the team and the management team
- performing all activities according to the team plan, e.g., design, implementation, basic test and planning and review activities
- follow-up of the activities in the team plan and reporting to the management team at check-point meetings
- re-planning if necessary and decided at a check-point meeting

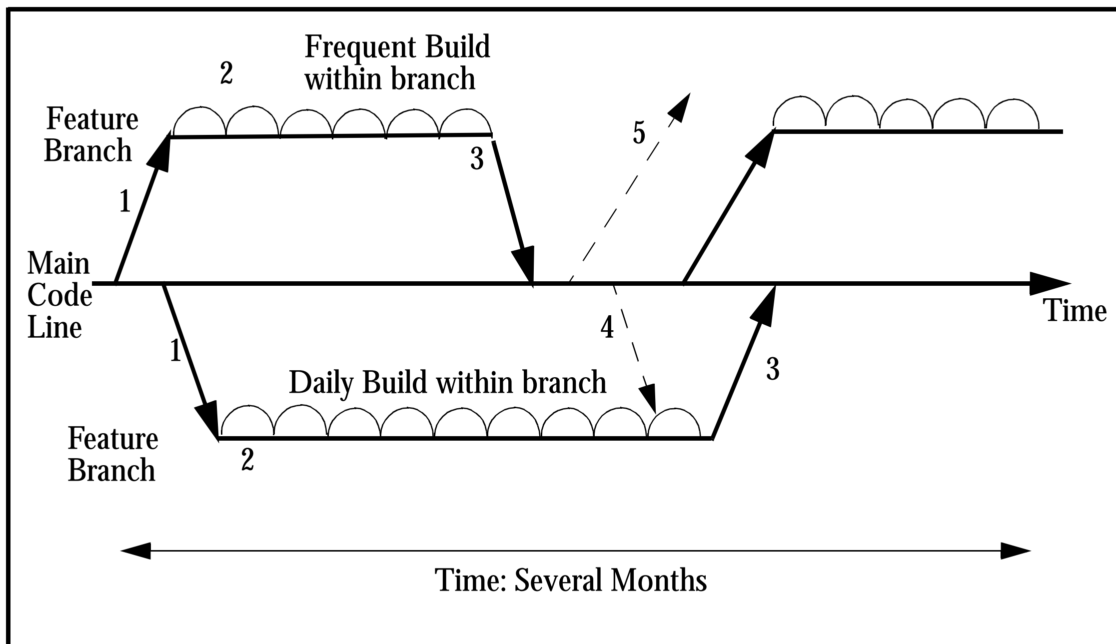
In addition, different support roles facilitated the development work:

- The basic test coordinator defined the overall requirements for the basic test and followed up the basic tests performed
- A Configuration Management group was responsible for the Clearcase environment, producing load modules and also baseline documentation for deliveries to function test

A plan for the deliveries to function test was produced by the basic test coordinator in cooperation with the project manager and the function test leader. It was based on the planned increments from the teams.

7.8.2. Incremental Development and Frequent Build

All source code was under configuration management control by means of Clearcase. Clearcase views, multiple branches and labeling mechanisms were used. Each design team (feature team) was assigned a new branch for each increment. Automated basic test programs existed in the design base for certain parts of the products. Instructions were defined for managing the source code with respect to the Clearcase environment. The main goal was to have source code with good and known quality at every moment in the project.



1. The feature team got a new feature branch from the main code line (the project main branch). Each designer got a private view connected to the new branch. The new branch was always taken out from a defined Clearcase label on the main code line, either the label for the original design base or a label set after a feature merge.

2. Using check in and check out of the code and frequent build within the branch, the feature team developed the feature. The build could be started in such a way that also the automated test programs were run. The team performed all the phases of implementation, code review and basic test of the feature within the feature branch. Designers frequently built and tested the whole product including the new code. The basic test was partly performed in the development environment, where the automated basic tests programs and some of the GUI tests were run. Furthermore, other tests were run in a dedicated run-time basic test machine, where the products were installed for basic test of run-time platform dependent parts.
3. The team merged the feature into the main code line. After the merge the team built the product and performed appropriate regression tests on the main code line to ensure that the new feature was still working and no side effects were found. If no changes had been made in the main code line since the feature branch was created no regression tests were needed. The automated tests were very useful for the regression tests; also, a selection of the manual tests was re-run. When the test had been completed a Clearcase label was set by the configuration management group on the main code line, and the feature branch was then deleted. A new feature branch was then created where the team worked with the next increment of the feature.
4. A "Pull-merge" method was used to resolve merging problems. This was very important to do - it was the way to synchronize the work of one feature team with the work of the other teams, as soon as possible. When a label was set after a feature merge the other feature teams were notified (e.g., by e-mail). If the changes in main code line would affect a feature developed by another team, the team made a merge from the main code line to the feature branch, to avoid merging problems later on at the feature merge.
5. The stabilization phase was entered. When a label had been set the quality of the main code line was sufficient to generate a load module for function test and system test. Prior to function test a pre-function test was run, where the product was installed in the target machine and some tests were done to check the installation and the major functions of the product.

In addition to the work performed by the team, the following actions were taken:

- Teams tried to avoid situations where the same source code file was modified by several feature teams in parallel. This aspect was considered in project planning when defining the features and composing the feature teams. This was not always possible to achieve. However, when parallel modifications were avoided, merges became considerably easier.
- A new branch was used for fault correction packages, in the same way as for each feature increment. The same applied for fault corrections merged from other projects. However, in the later part of the project faults were corrected directly on the main code line.

- The basic test coordinator made a plan of how the feature increments from the teams were gathered for pre-function tests.
- The code was developed very much in parallel with the design as well as the documentation of the design. Parts where coding could start early were identified. However the design documents as well as code and basic test programs and specifications were reviewed for each feature increment.

7.8.3. Important Issues

It is very crucial that the teams make good detail plans, covering all aspects of the work. The plans must be followed up frequently. It is essential to meet planned delivery dates.

If deviations to the team plans are arise, e.g., in delivery time, it is important to re-plan with respect to resources, functional contents, and other aspects. The team plan should then be updated.

Establish instructions for how to use Clearcase branches, views and labels and what working methods to use for merges. CM should ensure that the rules and methods are followed and should have a good cooperative relationship with the feature teams. CM should also be responsible for making sure that the branches and views are created from the appropriate label.

Establish instructions of how and when to perform code review and basic test with respect to the branches, merges and increments.

It is great advantage to have as much as possible of the basic tests automated; it simplifies the regression tests and makes it possible to re-run a larger amount of tests, which gives more control of the quality of the product.

7.9. Daily Build in Ericsson

Generally daily build and feature teams are attracting increased interest within Ericsson to increase their focus on code and customer features. Traditionally Ericsson's development process has been rather document-focused, and coding was seen as a rather mundane task at the end of the intellectual chain. This created long lead times in projects, and designers wanted to get out of coding and do more “interesting” documents. Introducing feature oriented design where teams are responsible for a complete customer feature from start to completion is one way of changing this situation.

Examples of large Ericsson improvement programs including daily build are:

- The Ericsson System Software Initiative (ESSI) where one of the good practices/vital few actions for 1999 is daily build and incremental development. This means that Ericsson's ESSI team is collecting good practices from within the company, and actively spreading these within the company.
- The World Class Provisioning (WCP) program, which aims to drastically improve quality, leadtime and productivity within Ericsson's GSM network organization, has daily build and incremental development as one of their 10 guiding principles. In particular the Base Station Controller organization (managed from Linköping) has been a leader in implementing feature teams and daily build - both on a small scale as in the GPRS project, and on a large scale for the complete AXE 10 BSC system.
- The CMS8800 (Ericsson's system for the mobile standard for the American market) has implemented feature teams and build on demand as one of its key strategies to meet the tougher competition on the American market.

Note that all these examples involve rather large systems, where for instance the problem of building the system in one day has not been trivial to solve.

8 References

1 Michael A. Cusumano, Richard W. Selby, *Microsoft Secrets,* 1995

2 Steve McConnell, *Rapid Development,* 1996

3 Grady Booch, James Rumbaugh, Ivar Jacobson, *The Unified Modeling Language User Guide*, 1998

4 Software Engineering Institute, *The Capability Maturity Model*, 1994

5 John P. Kotter, *Leading Change*, 1996

6 G. Pascal Zachary, *Showstopper,* 1994