ICT PROJECT 258109
Monitoring Control for Remote Software Maintenance

Project number: 258109
Project Acronym: FastFix
Project Title: Monitoring Control for Remote Software Maintenance
Call identifier: FP7-ICT-2009-5
Instrument: STREP
Thematic Priority: Internet of Services, Software, Virtualization
Start date of the project: June 1, 2010
Duration: 30 months

D2.1: State-of-the-art in monitoring control for remote maintenance

Lead contractor: TUM
Author(s): Dennis Pagano, Nitesh Narayan, Joao Coelho Garcia, Benoit Gaudin, Walid Maalej, Tobias Röhm
Submission date: September 2010
Dissemination level: Public

Project co-founded by the European Commission under the Seventh Framework Programme
© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: This report summarizes the current state of research and practice in the area of software maintenance with a particular emphasis on remote software maintenance. We summarize models and processes used in classical software maintenance and remote software maintenance. Then, we describe approaches and tools in software-fault monitoring using two different taxonomies. Concluding, we identify research islands within remote software maintenance and propose how these islands should be merged beneficially.

This document has been produced in the context of the FastFix Project. The FastFix project is part of the European Community’s Seventh Framework Program for research and development and is as such funded by the European Commission. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors view.
1 Introduction

2 Foundations

2.1 Software Maintenance ............................................. 9
2.2 Terminology ....................................................... 10
2.3 Software Maintenance Categories ............................... 13
2.4 Software Maintenance Models ................................. 15
2.5 Error Types ....................................................... 18

3 Classical Software Maintenance ................................. 20

3.2 ISO/IEC 12207: Software Life Cycle Processes ................. 23
3.4 Open Source Software Maintenance Process ................ 26
3.5 Summary ........................................................... 26

4 Remote Software Maintenance .................................. 29

4.1 Remote Software Maintenance Processes .................... 29
  4.1.1 Feedback Driven Development ............................... 31
  4.1.2 The Gamma System ........................................... 34
  4.1.3 Summary ........................................................ 35
4.2 Software-Fault Monitoring ..................................... 37
  4.2.1 Monitoring Systems .......................................... 37
  4.2.2 Monitoring Taxonomies ...................................... 39
  4.2.3 Existing Systems ............................................. 41
  4.2.4 Summary and Shortcomings ................................ 49
4.3 Fault Replication .................................................. 50
4.4 Self-Maintainable Systems ..................................... 50

5 Existing Systems .................................................... 53

5.1 Microsoft Fix It .................................................... 53
5.2 Rainbow Framework ........................................... 54
5.3 Gamma System ................................................... 55
5.4 Summary .......................................................... 56

6 Conclusion ......................................................... 58
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The waterfall model.</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Modification request according to ISO/IEC14764-2006 [33].</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Correspondences between ISO and IEEE maintenance categories.</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>The quick-fix process model according to Basili [2].</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>The iterative-enhancement model according to Basili [2].</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>The full-reuse model according to Basili [2].</td>
<td>18</td>
</tr>
<tr>
<td>2.7</td>
<td>Distribution of maintenance effort [46].</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>An example of maintenance process [80].</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>The IEEE maintenance process.</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>The ISO/IEC 12207 life cycle processes.</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>The ISO/IEC 12207 maintenance process.</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>The ISO/IEC 14764 Maintenance Process [33].</td>
<td>26</td>
</tr>
<tr>
<td>3.6</td>
<td>Classification of maintenance issues.</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>General software maintenance process.</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Feedback driven development according to Maalej et al. [50].</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Classification of user input [50].</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Maintenance and evolution process supported by the Gamma system [58].</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>General remote software maintenance process.</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>High-level view of a runtime monitor according to [14].</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>Software runtime monitoring model according to [26].</td>
<td>39</td>
</tr>
<tr>
<td>4.7</td>
<td>Runtime monitoring taxonomy according to [14].</td>
<td>40</td>
</tr>
<tr>
<td>4.8</td>
<td>Features of software runtime monitoring according to [26].</td>
<td>42</td>
</tr>
<tr>
<td>4.9</td>
<td>HP Insight Remote Support Advanced Architecture.</td>
<td>47</td>
</tr>
<tr>
<td>4.10</td>
<td>IBM Tivoli Monitoring Components.</td>
<td>47</td>
</tr>
<tr>
<td>4.11</td>
<td>Quantitative information about the relation between features and runtime</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>software-fault monitors according to [26].</td>
<td></td>
</tr>
<tr>
<td>4.12</td>
<td>The autonomic feedback loop.</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>The Rainbow framework</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Software Tomography according to [57].</td>
<td>55</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>IEEE categories of software maintenance.</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison of standard process activities [30].</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Activities of the ISO/IEC 12207 maintenance process and the open source software maintenance process.</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Monitors classified according to the specification language branch [14].</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Monitors classified according to the monitoring branch [14].</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Monitors classified according to the event-handler and operational issues branches [14].</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Monitors classified according to the taxonomy of Guo et al. [26].</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of existing remote software maintenance systems.</td>
<td>57</td>
</tr>
</tbody>
</table>
1 Introduction

Today software is everywhere: it drives economic processes, it controls manufacturing processes, it runs on mobile phones, and much more. Many sectors of society depend and rely on the correct operation of software. But what “correct operation” means may change over time. Software is used in the real world to solve problems. It is due to the fact that this real world changes that also the software often needs to be changed. In fact one of the main reasons why functionality is realized in software is its flexibility. If no flexibility was needed, the functionality could also be implemented as a hardware component.

Developing reliable software is difficult because of the inherent complexity of current systems and the limited availability of resources. Analysis and testing techniques have been proposed for improving the quality of software during development. However, because of the limitations of these techniques, time-to-market pressures, and limited development resources, software products are still being released with missing functionality or errors. Because of the growth of the Internet and the emergence of ubiquitous computing, the situation has worsened in two respects. First, the widespread use of computer systems has caused a dramatic increase in the demand for software. This increased demand has forced companies to shorten their software development time, and to release software without performing required analysis and testing. Second, many of today’s software products are run in complex and variable environments: a software product may need to interact through a network with software products running on other computers; a software product may also need to be able to run on computers that are each configured differently. It may be impractical to analyze and test these software products before release under all possible runtime environments and configurations. Software has become more distributed using the Internet. Hence, there is less interaction between clients and developers or maintainers, making remote maintenance increasingly necessary.

Any means capable of reducing maintenance costs would improve success opportunities for a software vendor, providing a reduction on costs and a more competitive edge in the market. Vendors need a system to remotely provide a high quality support service to software customers, to improve user experience and ease maintenance. Remote Software Maintenance deals with the remote analysis and improvement of software products after deployment.

Though it seems self-evident that monitoring the maintained software system is a key
enabler for effective remote software maintenance, there is no commonly accepted remote software maintenance process comprising this activity and no common monitoring standard. This report surveys the state of the art in remote software maintenance and in particular in monitoring software systems to support remote software maintenance.

The contribution of this report is fourfold. First, it summarizes models and processes used in software maintenance. Second, it defines remote software maintenance, and proposes an appropriate process. Third, the report summarizes the current state of the art approaches and tools in software-fault monitoring, using two different taxonomies. Fourth, the report concludes with the identification of research islands within remote software maintenance and a proposal how these islands could be merged beneficially.

Section 2 deals with important definitions and concepts of software maintenance. Section 3 summarizes classical software maintenance processes and the most important standards. Section 4 introduces the notion of remote software maintenance, and discusses remote software maintenance processes. Furthermore it surveys software-fault monitoring systems and introduces the related fields fault replication and self-maintainable systems. Section 5 summarizes existing remote software maintenance systems and their features. Finally, section 6 concludes the report, summarizing findings and shortcomings of current approaches and systems and states challenges for future remote software maintenance systems.
2 Foundations

In this section we provide the most important definitions for software maintenance. The objective is to build a common vocabulary and to delineate the focus of the document. We also describe the main activities in software maintenance, introduce common categories of software maintenance, in order to be able to classify activities and determine important types of errors.

2.1 Software Maintenance

Although software does not deteriorate while being in use and as time goes by, it is necessary that software is modifiable after it has been delivered. According to Lehman’s laws of evolution [44], successful software systems are condemned to change over time. A predominant proportion of changes is to meet ever-changing user needs which are essential to keep a software system useful. This fact is expressed by the first law of Lehman: “A program that is used in a real world environment necessarily must change or become progressively less useful in that environment”. Significant changes are also due to the fact that software needs to be adapted to interact with external entities, such as users, security concerns, organizations, and other (software) systems. Since software is a very flexible concept, it is often considered the easiest part to change in a system [5].

Software maintenance and support services are key factors to customer’s perception of software product quality. Besides, maintenance and support services constitute a growing percentage of the software business market. Business models are changing for software vendors: from license centred to services centred. This trend can be particularly observed in the case of Open Source software, where licenses are very often free of charge and services represent the major source of income. For example, approximately 65% of the Eclipse foundation [18] members rely on services of the Eclipse open source platform – most of them are European Small and Medium Enterprises (SME’s). Eclipse is one of the largest open source communities in Europe and the world comprising 11 Million users and more than 10,000 developers.

Any means capable of reducing maintenance costs would improve success opportunities for a software vendor, providing a cost reduction and a more competitive edge in the market. Vendors need a system to remotely provide a high quality support service to software
customers, to improve user experience and facilitate perfective, corrective, adaptive and preventive maintenance. This has to be applicable to both new and existing software products. No highly specialized knowledge should be required.

On the other hand, maintaining software is a tedious and time-consuming task. Fjeldstad and Hamlen found out in a frequently cited empirical study that software engineers spend more than 50% of their time working on software maintenance tasks [18]. Similarly, Boehm claims that, assuming a typical software life cycle, between 50% and 75% of the costs are due to maintenance [3]. This workload slows down the growth of software vendors and may postpone new development opportunities. It also causes a reduction in overall software quality given the possibility of introducing latent errors in the maintained software.

Many of nowadays’ software applications are running on customers’ hardware and system software environment, and have been developed either by large software companies or by SME’s. Others run on the cloud, in a usually more controlled environment and are offered as SaaS (Software as a Service). In both cases, applications interact with users and run in environments that are not exactly the same ones that were used at development and testing time. Besides, some characteristics of the environment are not under control, such as changes and updates in system software, network components, and new security threats. Also, there are changes in the way users interact with the applications and the goals they aim to achieve with their help.

2.2 Terminology

Software Maintenance The term “Software Maintenance” is used for a very broad range of activities, often defined as all changes made on a software system after it has become operational [53]. This description covers the correction of errors, the enhancement, removal and addition of functionality, the adaptation of the system to changes in data and operation requirements and also the improvement of the system regarding non-functional requirements like performance or usability. The IEEE definition [31] is as follows:

“Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.”

The definition by IEEE reflects the common view that software maintenance is an activity that takes place after the delivery of the system: it begins with the release of the system to the customer or user and comprises all activities ensuring that the system remains operational and suits the needs of its users. The classical waterfall models of the software
life cycle are well aligned with this view, since they encompass in general a final phase of operation and maintenance, as shown in Figure 2.1.

There are several authors though, who disagree with this view and state that software maintenance has to start already before a system becomes operational. According to Schneidewind [72] one reason for the complexity of maintenance is this restricted view that maintenance is only a post-delivery activity. Osborne and Chikofsky [79] state that adopting a life cycle approach to the management and change of a software system is essential. They claim an approach that considers maintenance within all aspects of the development process. Pigoski [62] redefined software maintenance, taking into account that it should start at the beginning of development:

“Software maintenance is the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage. Pre-delivery activities include planning for post-delivery operations, supportability, and logistics determination. Post-delivery activities include software modification, training, and operating a help desk.”

The ISO standards on software life cycle processes [32] follow an approach which is consistent with this definition by Pigoski. The image that software maintenance comprises only fixing bugs or mistakes is definitely dispelled in these standards.
D2.1: State-of-the-art in monitoring control for remote maintenance

Maintainability The IEEE definition of maintainability reflects the definition of maintenance: “The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment” [31]. The ISO/IEC 9126 standard [35] assumes maintainability as one of the six primary characteristics of its definition of software quality. In this standard maintainability is defined as “a set of attributes that bear on the effort needed to make specified modifications”. According to ISO maintainability depends on five sub-characteristics: analyzability, changeability, stability, testability and maintainability compliance. ISO/IEC 9126 is superseded since 2005 by ISO/IEC 25000: Software engineering: Software product Quality Requirements and Evaluation (SQuaRE): Guide to SQuaRE. Although the ISO/IEC SQuaRE quality model is well-known in industry and covers also maintenance issues, it is too general and too abstract to be directly applied; moreover some of the metrics that the model suggests are very difficult and expensive to apply. Bombardieri and Fontana [4] specialized the ISO/IEC SQuaRE quality model in order to make it easier to use for a company.

Modification Request (MR) According to ISO/IEC14764 [33], a modification request is “a generic term used to identify proposed modifications to a software product that is being maintained. [...] The MR may later be classified as a correction or enhancement and identified as corrective, preventive, adaptive, or perfective maintenance. MRs are also referred to as change requests.”. See Figure 2.2 for an overview of modification requests.

Problem Report (PR) According to ISO/IEC14764 [33], a problem report is “a term used to identify and describe problems detected in a software product. [...] PRs are either submitted directly to denote faults or established after impact analysis is performed on Modification Requests and faults are found.”.

2.2.0.0.1
2.3 Software Maintenance Categories

From 1970 to 1990 the phenomenon of software maintenance was broadly researched with the goal of identifying the reasons for the changes made to software systems, to investigate their relative frequencies and estimate their costs. The result of this research comprises classifications of maintenance activities which help to understand the significance of maintenance and its relation to the costs and quality of software systems in use. This categorization of maintenance efforts proved that software maintenance goes beyond the correction of defects.

Lientz and Swanson [46] separate maintenance into three categories: corrective, adaptive, and perfective maintenance. Corrective maintenance comprises all changes made to a system in order to remove defects in the software. Adaptive maintenance addresses the change in the environment in which a software system must operate. This encompasses the adaption of a system to a new hardware platform, operating system, database system, virtual machine or network. In contrast, perfective maintenance regards changes triggered by actual requests of the users of a software system. This includes adding, changing or removing functions, documentation changes, addressing performance or usability issues.

A further refinement of this definition by Pigoski [62] is the suggestion to join adaptive and perfective maintenance creating a category named “enhancements”, making it explicit that these categories are rather improvements than corrections. In fact sometimes the term “software maintenance” is used to refer only to the implementation of small changes. In contrast, all other modifications are subsumed under “software development”.

As maintenance operations change a system, they may also degrade the reliability and structure of the system and affect its maintainability. This of course makes changes progressively more difficult and costly to implement. According to Parnas [60], this is often the case in real-world maintenance, which induces the phenomenon of software aging. This is also expressed by the second law of Lehman [44]: “As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving the semantics and simplifying the structure”. Accordingly, a fourth category of maintenance, preventive maintenance, is considered in literature [63]. Preventive maintenance includes all the modifications made to a piece of software after delivery in order to detect and correct latent faults in the software product before they become effective faults.

The ISO/IEC 9000-3 [34] introduces three categories of software maintenance:

- **Problem resolution**, comprising the detection, analysis, and correction of software nonconformities causing operational problems,

- **interface modification**, required when additions or changes are made to the system controlled by the software,
D2.1: State-of-the-art in monitoring control for remote maintenance

- **functional expansion** or **performance improvement**, which may be required by the customer in the maintenance stage.

It is recommended that all maintenance changes be made in accordance with the same procedures used for software development. However, during problem resolution, temporary fixes may be used to minimize system downtime and implement persistent changes after this step.

The IEEE Std 1219-1998 standard [30] redefines the maintenance categories claimed by Lientz and Swanson [46] of corrective, adaptive, and perfective maintenance, and adds emergency maintenance as a fourth category. The IEEE definitions are as follows:

- **Corrective maintenance**: reactive modification of a software product performed after delivery to correct discovered faults.

- **Adaptive maintenance**: modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.

- **Perfective maintenance**: modification of a software product performed after delivery to improve performance or maintainability.

- **Emergency maintenance**: unscheduled corrective maintenance performed to keep a system operational.

These definitions introduce the idea that software maintenance can be either scheduled or unscheduled and reactive or pro-active, as shown in Table 2.1. Figure 2.3 depicts the correspondences that exist between ISO and IEEE categories.

The ISO/IEC 14764 standard [33] refines or redefines these definitions. Following are the according definitions from the ISO/IEC 14764 standard:

- **Corrective maintenance**: the reactive modification of a software product performed after delivery to correct discovered problems.

- **Adaptive maintenance**: the modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment.

- **Perfective Maintenance**: the modification of a software product after delivery to detect and correct latent faults in the software product before they are manifested as failures.

---

1 In Annex A: Maintenance guidelines of the IEEE standard, also preventive maintenance is defined: “Maintenance performed for the purpose of preventing problems before they occur”. Preventive maintenance thus makes the fifth category of maintenance defined in IEEE Std 1219-1998.
D2.1: State-of-the-art in monitoring control for remote maintenance

<table>
<thead>
<tr>
<th>ISO categories</th>
<th>IEEE categories</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>problem resolution</strong></td>
<td>emergency maintenance</td>
</tr>
<tr>
<td><strong>interface modification</strong></td>
<td>corrective maintenance</td>
</tr>
<tr>
<td><strong>functional expansion or</strong></td>
<td>adaptive maintenance</td>
</tr>
<tr>
<td><strong>performance improvement</strong></td>
<td>perfective maintenance</td>
</tr>
</tbody>
</table>

Figure 2.3: Correspondences between ISO and IEEE maintenance categories.

<table>
<thead>
<tr>
<th></th>
<th>Unscheduled</th>
<th>Scheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reactive</td>
<td>Emergency</td>
<td>Corrective, Adaptive</td>
</tr>
<tr>
<td>Pro-active</td>
<td></td>
<td>Perfective</td>
</tr>
</tbody>
</table>

Table 2.1: IEEE categories of software maintenance.

- **Preventive maintenance**: the modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults.
- **Emergency maintenance**: an unscheduled modification performed to temporarily keep a system operational pending corrective maintenance.

### 2.4 Software Maintenance Models

This section introduces general models of maintenance processes. These general models describe the approaches generally possible to perform a maintenance task. Of course some are better suited than others depending on the environment, current situation and organization using them. The next section deals with the processes that are considered to be standard today.

A typical approach to software maintenance is to work on code first, and then make the necessary changes to the accompanying documentation, if any. This approach is captured by the quick-fix model, shown in Figure 2.4, which demonstrates the flow of changes from the old to the new version of the system [2]. Ideally, after the code has been changed the requirement, design, testing and any other form of available documents impacted by the
modification should be updated. However, due to its perceived malleability, users expect software to be modified quickly and cost-effectively. Changes are often made on the fly, without proper planning, design, impact analysis, and regression testing. Documents may or may not be updated as the code is modified; time and budget pressure often entails that changes made to a program are not documented and this quickly degrades documentation. In addition, repeated changes may demolish the original design, thus making future modifications progressively more expensive to carry out.

Evolutionary life cycle models suggest an alternative approach to software maintenance. These models share the idea that the requirements of a system cannot be gathered and fully understood initially. Accordingly, systems are to be developed in builds each of which completes, corrects, and refines the requirements of the previous builds based on the feedback of users [23]. An example is iterative enhancement [2], which suggests structuring a problem to ease the design and implementation of successively larger/refined solutions. Iterative enhancement explains maintenance too, as shown in Figure 2.5. The construction of a new build (that is, maintenance) begins with the analysis of the existing system’s requirements, design, code and test documentation and continues with the modification of the highest-level document affected by changes, propagating the changes down to the full set of documents. In short, at each step of the evolutionary process, the system is redesigned based on an analysis of the existing system.

A key advantage of the iterative-enhancement model is that documentation is kept updated as the code changes. Visaggio [77] reports data from replicated controlled-experiments conducted to compare the quick-fix and the iterative-enhancement models and shows that the maintainability of a system degrades faster with the quick-fix model. The experiments also indicate that organizations adopting the iterative-enhancement model make maintenance changes faster than those applying the quick-fix model; the latter finding is counter-intuitive, as the most common reason for adopting the quick-fix model is time pressure.
Basili [2] suggests a model, the full-reuse model shown in Figure 2.6, that views maintenance as a particular case of reuse-oriented software development. Full-reuse begins with the requirement analysis and design of a new system and reuses appropriate requirement, design, code, and test artifacts from existing systems during these activities. While both iterative-enhancement model and full-reuse model encourage reuse, the iterative-enhancement model focuses on the enhancement of an existing system by starting with the analysis of the current system and developing and improving it further. In contrast, the full-reuse model focuses on building a new system reusing existing artifacts of an old version of a system and other related systems. Consequently, the iterative-enhancement model is well suited for systems that have a long life and evolve over time because of its support of system evolution that eases future modifications. The full-reuse model is more suited for the development of lines of related products.

Central to the full-reuse model is the idea of a repository of documents and components defining earlier versions of the current system and other systems in the same application domain. This makes reuse explicit and documented. It also promotes the development of more reusable components. Implementation of the full-reuse model tends to be more costly on the short run, whereas the advantages may be sensible in the long run. Organizations that apply the full-reuse model accumulate reusable components of all kinds and at different levels of abstractions. This makes future development more cost effective but requires mechanisms to manage artifacts of different types at different levels of abstraction and their relationships.
2.5 Error Types

Several technical and managerial problems contribute to the costs of software maintenance. Among the most challenging problems of software maintenance are: program comprehension, impact analysis, and regression testing. Though maintenance is more than just correcting errors, the concept of an error is central in software maintenance. Lientz and Swanson [46] showed that about 25% of software maintenance efforts are spent on corrective and preventive tasks (see Figure 2.7), both of which are concerned with errors.

There is a commonly used terminology for error-related concepts. The most important ones are the following, which are defined e.g. in [6]:

- *Error, erroneous state.* Error means that the system is in a state such that further processing by the system will lead to a failure, which then causes the system to
deviate from its intended behavior.

- **Fault, defect, bug.** A fault, also called "defect" or "bug", is the mechanical or algorithmic cause of an erroneous state.

- **Failure.** A failure is any deviation of the observed behavior from the specified behavior.

While there exist several taxonomies of faults in special areas of computer science (e.g. SOA [8] or Web Applications [52]), we didn’t find a common general taxonomy describing high-level categories of errors, faults and failures.

The following list describes common error and failure categories we found:

- **Errors**
  - *Configuration errors.* Instance of an application has wrong configuration parameters or hardware configuration e.g. screen resolution is set up wrongly.
  - *System crashes.* A system that the software depends on crashes (virtual machine, operating system, etc.).
  - *Application crashes.* Unhandled exceptions that lead to a crash.
  - *Usability errors.* Difficulties of users in using particular features as well as user interface problems. There exists a special taxonomy for usability errors [24].
  - *Resource related errors.* All errors related with resource management, having as root cause the availability, concurrence of use, or integrity of the resources.
  - *Invalid user input.* Errors are caused by unforeseen user input.

- **Failures**
  - *Wrong functionality.* E.g. security issues or data or information loss due to wrong behavior.
  - *Missing features.* Some specified features are missing in the system.
  - *Incompatibility with environment.* Interference with other applications or runtime environment incompatibilities causes a failure of the system.
  - *Incompatibility of versions.* Features or artifacts that are associated with different versions of the software are not compatible.
3 Classical Software Maintenance

Several authors have proposed process models for software maintenance. These models organize maintenance into a sequence of related activities, or phases, and define the order in which these phases are to be executed. Sometimes, they also suggest the deliverables that each phase must provide to the following phases. An example of such a process is shown in Figure 3.1. Although different authors identify different phases, they agree that there is a core set of activities that are indispensable for successful maintenance, namely the comprehension of the existing system, the assessment of the impact of a proposed change, and the regression testing.

The generic process model by Harjani [27] is an example for a process model that defines activities within a software maintenance process:

1. **Trigger.** The trigger of a software maintenance process is usually the receipt of a “software problem report” by the maintenance organisation, stating a problem in the use of the software or indicating a need for change, as well as a priority level.

2. **Problem understanding.** The problem understanding phase is the first step of the maintenance process. It is performed as a preliminary filter to decide whether the problem results from wrong usage of the software, if the problem is new or if it has already been reported.

3. **Localization.** The localization is an investigation process aimed at determining precisely what is requested, and which parts of the software have to be modified.

4. **Solution analysis.** Solution analysis is the phase during which the various possible changes are devised, in the light of the results of the localization step.

5. **Impact analysis.** The impact analysis is the activity aimed at evaluating, for each change foreseen in the solution analysis, all the consequences of its application.

6. **Decision of implementation.** This phase aims at deciding which action should be undertaken concerning the problem in hand, with regard to the different solutions foreseen (and their impact).
7. **Implementation.** Implementation is the phase during which a modification process, similar to a mini-development cycle, is carried out, from the specification of the solution to its validation.

8. **Regression testing.** The phase of testing that the new or modified components interact correctly with unmodified parts of the software and checking that behaviour that has not been modified intentionally is unchanged.

9. **Acceptance.** Acceptance aims at checking that the solution implemented was the right answer to the reported problem. Activities performed within this phase are very dependent on the problem to be solved and may or may not include additional testing, depending on the problem and on the type of maintenance.

10. **Closure of the intervention.** This phase takes place at the end of the intervention, before the new release of the software is delivered. Activities within this phase are checking that test plan and quality assurance procedures have been satisfactorily carried out, configuring the new or modified components, recording all changes performed, officially approving the change.

11. **Re-insertion.** Re-insertion is the phase during which the modified software is installed in its operational environment, to replace the old version.

IEEE and ISO have both addressed software maintenance, the first with a specific standard [30] and the latter as a part of its standard on life cycle processes [32, 33]. The next three sections describe the maintenance processes defined by these documents.

The IEEE standard 1219-1998 [30] organizes the maintenance process in seven phases, as demonstrated in Figure 3.2. In addition to identifying the phases and their order of execution, for each phase the standard indicates input and output deliverables, the activities grouped, related and supporting processes, the control, and a set of metrics.

**Problem/modification identification, classification, and prioritization** This is the phase in which the request for change (MR – modification request) issued by a user, a customer, a programmer, or a manager is assigned a maintenance category, a priority and a unique identifier. The phase also includes activities to determine whether to accept or reject the request and to assign it to a batch of modifications scheduled for implementation.

**Analysis** This phase devises a preliminary plan for design, implementation, test, and delivery. Analysis is conducted at two levels: feasibility analysis and detailed analysis. Feasibility analysis identifies alternative solutions and assess their impacts and costs, whereas detailed analysis defines the requirements for the modification, devises a test strategy, and develop an implementation plan.

**Design** The modification to the system is actually designed in this phase. This entails using all current system and project documentation, existing software and databases, and the output of the analysis phase. Activities include the identification of affected software modules, the modification of software module documentation, the creation of test cases for the new design, and the identification of regression tests.
Implementation  This phase includes the activities of coding and unit testing, integration of the modified code, integration and regression testing, risk analysis, and review. The phase also includes a test-readiness review to assess preparedness for system and regression testing.

Regression/system testing  This is the phase in which the entire system is tested to ensure compliance to the original requirements plus the modifications. In addition to functional and interface testing, the phase includes regression testing to validate that no new faults have been added. Finally, this phase is responsible for verifying preparedness for acceptance testing.

Acceptance testing  This level of testing is concerned with the fully integrated system and involves users, customers, or a third party designated by the customer. Acceptance testing comprises functional tests, interoperability tests, and regression tests.

Delivery  This is the phase in which the modified systems is released for installation and operation. It includes the activity of notifying the user community, performing installation and training, and preparing and archival version for backup.

3.2 ISO/IEC 12207: Software Life Cycle Processes

While the standard IEEE Std 1219-1998 [30] is specifically concerned with software maintenance, the standard ISO/IEC 12207 [32] deals with the totality of the processes comprised in the software life cycle. The standard identifies seventeen processes grouped into three broad classes: primary, supporting, and organizational processes. Processes are divided into constituent activities each of which is further organized in tasks. Figure 3.3 shows the processes and their distribution into classes. Maintenance is one of the five primary processes, i.e. one of the processes that provide for conducting major functions during the life cycle and initiate and exploit support and organizational processes. Figure 3.4 shows the activities of the maintenance processes; the positions do not indicate any-time dependent relationships.

Process implementation  This pre-delivery activity includes the tasks for developing plans and procedures for software maintenance, creating procedures for receiving, recording, and tracking maintenance requests, and establishing an organizational interface with the configuration management process. Process implementation begins early in the system life cycle; Pigoski [62] affirms that maintenance plans should be prepared in parallel with the development plans. The activity entails the definition of the scope of maintenance
and the identification and analysis of alternatives, including offloading to a third party; it also comprises organizing and staffing the maintenance team and assigning responsibilities and resources.

**Problem and modification analysis** The first task of this activity is concerned with the analysis of the maintenance request, either a problem report or a modification request, to classify it, to determine its scope in terms of size, costs, and time required, and to assess its criticality. It is recommended that the maintenance organization replicates the problem or verifies the request. The other tasks regard the development and the documentation of alternatives for change implementation and the approval of the selected option as specified in the contract.

**Modification implementation** This activity entails the identification of the items that need to be modified and the invocation of the development process to actually implement the changes. Additional requirements of the development process are concerned with testing procedures to ensure that the new/modified requirements are completely and correctly implemented and the original unmodified requirements are not affected.

**Maintenance review/acceptance** The tasks of this activity are devoted to assessing the integrity of the modified system and end when the maintenance organization obtains the approval for the satisfactory completion of the maintenance request. Several support-
D2.1: State-of-the-art in monitoring control for remote maintenance

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Process implementation</td>
<td>(Maintenance planning)</td>
</tr>
<tr>
<td>Problem and modification analysis</td>
<td>(a) Problem/modification identification, classification, and prioritization (b) Analysis</td>
</tr>
<tr>
<td>Modification implementation</td>
<td>(a) Design (b) Implementation</td>
</tr>
<tr>
<td>Maintenance review / acceptance</td>
<td>(a) System Testing (b) Acceptance Testing</td>
</tr>
<tr>
<td>Migration</td>
<td></td>
</tr>
<tr>
<td>Retirement</td>
<td>(Software replacement policy)</td>
</tr>
<tr>
<td>-</td>
<td>Delivery</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of standard process activities [30].

ing processes may be invoked, including the quality assurance process, the verification process, the validation process, and the joint review process.

**Migration** This activity happens when software systems are moved from one environment to another. It is required that migration plans be developed and the users/customers of the system be given visibility of them, the reasons why the old environment is no longer supported, and a description of the new environment and its date of availability. Other tasks are concerned with the parallel operations of the new and old environment and the post-operation review to assess the impact of moving to the new environment.

**Software retirement** The last maintenance activity consists of retiring a software system and requires the development of a retirement plan and its notification to users.


ISO/IEC 14764[33] describes in greater detail management of the maintenance process described in ISO/IEC 12207, including amendments. It also establishes definitions for the various types of maintenance. ISO/IEC 14764 provides guidance that applies to planning, execution and control, review and evaluation, and closure of the maintenance process. The scope of ISO/IEC 14764 includes maintenance for multiple software products with the same maintenance resources. The current edition is the result of merging the original edition with IEEE Std 1219-1998. From the perspective of the different activities within the maintenance process, ISO/IEC 14764 is not different from ISO/IEC 12207 resp. IEEE Std 1219-1998. Figure 3.5 provides an overview over the ISO/IEC 14764 maintenance process.
D2.1: State-of-the-art in monitoring control for remote maintenance

3.4 Open Source Software Maintenance Process

Koponen and Hotti [42] studied the maintenance processes of two large open source software projects\(^1\). They identified several tasks and grouped them into activities according to the ISO/IEC 12207 standard. The five identified activities are process implementation, problem and modification analysis, implementation, modification review and acceptance and release management. While the first four activities are similar to the according ISO/IEC activities, the last one, release management, replaces the migration activity of the ISO/IEC process. This is due to the fact that new versions are only released, not migrated. The ISO/IEC activity retirement was not found in the open source process. Table 3.2 shows the comparison of the open source software maintenance process and the ISO/IEC 12207 process.

3.5 Summary

The current standard processes deal with problem reports and modification requests as entities that trigger maintenance activities. We define a common term for referring to both of these entities, namely *Maintenance Issue* (see Figure 3.6).

The activities of the IEEE Std 1219-1998 process can be grouped by the ISO/IEC 12207 resp. ISO/IEC 14764 process categories (see Table 3.1). The activity delivery

---

\(^1\)The case study surveyed the Apache and Mozilla open source projects.
D2.1: State-of-the-art in monitoring control for remote maintenance

<table>
<thead>
<tr>
<th>ISO/IEC maintenance process</th>
<th>Open source software maintenance process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process implementation</td>
<td>Process implementation</td>
</tr>
<tr>
<td>Problem and modification analysis</td>
<td>Problem and modification analysis</td>
</tr>
<tr>
<td>Modification implementation</td>
<td>Implementation</td>
</tr>
<tr>
<td>Maintenance review / acceptance</td>
<td>Modification review and acceptance</td>
</tr>
<tr>
<td>Migration</td>
<td>Release management</td>
</tr>
<tr>
<td>Retirement</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.2: Activities of the ISO/IEC 12207 maintenance process and the open source software maintenance process.

![Classification of maintenance issues](image)

Figure 3.6: Classification of maintenance issues.

of the IEEE Std 1219-1998 process doesn’t have an according correspondent in the ISO processes, according to the comparison in [30]. In our opinion it can be matched most closely to the migration activity, though. But we propose to use the term Solution delivery which corresponds closely to the Release Management activity of the current open source maintenance processes.

Based on this alignment and the definition of a maintenance issue we try to identify a common general process for software maintenance. It should not be a new process definition, but rather an approach to define a set of high-level activities which are common to all current definitions (see Figure 3.7 for an overview):

1. Maintenance issue analysis. This activity corresponds to the Problem and modification analysis activity of the ISO/IEC 12207 process. It comprises the following activities from the IEEE Std 1219-1998 process:
   a) Maintenance issue identification, classification, and prioritization
   b) Analysis

2. Solution design and implementation. This activity corresponds to the Modification implementation activity of the ISO/IEC 12207 process. It comprises the following activities from the IEEE Std 1219-1998 process:
   a) Design
b) Implementation

3. Testing. This activity corresponds to the Maintenance review / acceptance activity of the ISO/IEC 12207 process. It comprises the following activities from the IEEE Std 1219-1998 process:

   a) System Testing
   b) Acceptance Testing

4. Solution delivery. This activity corresponds to the Release Management activity of the open source maintenance process. It comprises the following activity from the IEEE Std 1219-1998 process:

   a) Delivery
4 Remote Software Maintenance

In contrast to classic software maintenance, the term remote maintenance denotes that certain activities of the software maintenance process are performed remotely by the maintainers. Although the term remote software maintenance seems quite ordinary, there is no common definition to our knowledge though there are individual papers using it [47]. In areas other than software, remote maintenance denotes that an expert supports the user of a system remotely, i.e. from a distant place (e.g. remote maintenance of aircraft by distant experts).

Sneed [74] claims that software maintenance should be offered as an offshore service by specialized companies. He regards the remote software maintenance infrastructure to be distributed, with high speed Internet connections between maintainer and customers running the maintained system. However, he states that representatives of the service provider would have to be on site at the user organization. His understanding of remote software maintenance is essentially a redefinition of what is software maintenance in general today, comprising bug reports, feature requests and release management.

In this chapter we survey how processes may change when maintaining software remotely and what are special needs to support the remote software maintenance scenario, like the monitoring of the maintained system. Furthermore, we introduce shortly the advanced related fields of fault replication and self-maintainable systems.

4.1 Remote Software Maintenance Processes

Remote Software Maintenance processes need to deal with the distance between the maintainer and the running maintained system. This distribution has an impact on (a) how status information about the maintained system is received, (b) how errors in maintained systems are recognized, (c) how problems can be understood, and (d) how a solution can be delivered.

In the literature no standard processes for remote software maintenance could be found. The classical software maintenance processes (see section 3) provide a general structure for software maintenance according to the activities software maintenance has to deal with. The distribution of maintainer and maintained system is not considered so far, and no alignment of software maintenance processes with this distribution has been done.
D2.1: State-of-the-art in monitoring control for remote maintenance

One step towards a remote software maintenance process was done by the identification of an open source software maintenance process (see section 3.4). Koponen and Hotti [42] identified release management as an activity in the open source software maintenance process, replacing the migration activity of the classical ISO/IEC 12207 process.

Sneed [74] points out that an essential success factor for software maintenance as an offshore service is a well defined and well structured maintenance process. He proposes to consider the prominent standards IEEE Std 1219-1998 [30] and ISO/IEC 12207 [32] (see section 3), but to adapt the defined processes to the special needs of the maintenance organizations and the particular environment.

To maintain software, different types of information about the maintained system are necessary. In practice common information collection mechanisms are feature requests, issue and bug reports, enhancement requests as well as the collection of usage data [50]. The classical software maintenance processes do not deal with the collection of information about the maintained system, even though it is an important activity in current software maintenance approaches. The IEEE Std 1219-1998 software maintenance process starts with the \textit{Problem/modification identification, classification, and prioritization} activity, assuming that the required information is already gathered. According to the ISO/IEC 14764 software maintenance process, the inputs for the \textit{Process Implementation} activity should include “a Modification Request (MR) or Problem Report (PR), if applicable.” [33]. The standard also deals theoretically with the establishment of a reporting and processing infrastructure for modification requests and problem reports:

\begin{quote}
“The maintainer shall (ISO/IEC 12207 sub-clause 5.5.1.2) establish procedures for receiving, recording, and tracking problem reports and modification requests from the users and providing feedback to the users. Whenever problems are encountered, they shall (ISO/IEC 12207 sub-clause 6.8) be recorded and entered into the Problem Resolution Process.”
\end{quote}

Although the standard introduces the tasks required to establish a working maintenance infrastructure, the entities it deals with are problem reports and modification requests. There is no task or activity that deals with the monitoring of the maintained system or the interpretation of its behavior regarding its specification, so no activity covers the identification of a problem or a modification need. Further, no tasks or procedures describe the monitoring of the users of the maintained software, though today’s practice in software maintenance includes gathering feedback from users in several ways, e.g. face to face interviews, workshops for users and stakeholders, collection of usage data, surveys, etc. [50].

The following section deals with a new process which makes user feedback a first order concern within requirements engineering. Since requirements engineering and software
maintenance are highly related, the process may be easily adopted to the area of software maintenance.

4.1.1 Feedback Driven Development

Maalej et al. [50] propose a continuous and context-aware approach for communicating user input to engineering teams and other users during requirements engineering. Even though requirements engineering and software maintenance are considered two different fields in computer science, there exist similarities and various commonalities within these fields. Issues, requirements and change requests in requirements engineering can be related to problem or bug reports, feature requests and modification reports in software maintenance. A boundary between the two fields is not clear, in particular in today’s iterative software life-cycle models. Thus, benefits from processes within one of these fields should be considered potentially beneficial in the other as well.

Continuous feedback model  Figure 4.1 depicts the visionary process of “feedback driven development”, which reduces several gaps in development and maintenance processes by continuous, integrated interaction streams. User input is formulated within a concrete software environment, which allows to seamlessly and more precisely communicate feedback into the development process. The engineering team can then analyze, consolidate and consider the input, make clarification requests and implement actual modifications. These are again propagated into the users’ runtime system, providing feedback on the improved version, starting the cycle again. Complementary to this, users can collaboratively engage in discussing requests but also in sharing know how. Following are specific functions in this process.

- Prospective Observation. The context in which user requirements have "emerged"
is captured and included communication threats. Users can continuously, efficiently
and intuitively communicate issues, requirements and change requests from the
working environment. The communication and understanding of the requirements
become a prospective, asynchronous process. Thereby the main assumption is that
users always have changing needs and new requirements, and that applications it-
theratively grow in functionality.

- **Assisted Feedback.** The application automatically recognizes the problem situations
by observing repetitive patterns in the users’ interactions (e.g. iteration in the
navigation or long times to execute a certain activity). Users are proactively asked
to communicate their input and feedback to the application engineers. Similar to
the detection of typical problem situations, feedback data might be leveraged to
analyze for common patterns in application workflows.

- **Community Sharing.** Users themselves can form communities on application us-
age or even about usage of certain features. In these communities users exchange
experiences and evaluate or comment features. Users get proactive suggestions in
problem situations, what they should do and what other users have done in similar
context. The application can also suggest other users with similar usage behavior,
problems or particular pains.

- **Back- Feedback (Integrated Intelligent Help-desk).** By observing and comparing
user interactions, the application learns how certain problems are solved by which
features. New users are supported by pro-active “How-To” suggestions in problem
situations. The workflows which can lead to solution of the problem are suggested.
Also, users’ interaction data can be analyzed to identify most valuable features
which would require the most extensive documentation.

**Advantages and required technical building blocks** According to Maalej et al. a
prospective and context-aware requirements engineering approach would offer advantages
to users as well as to application vendors. Users can send feedback to the application
manufacturers at any time without much additional effort. This feedback may consist
of new requirements, semantic mistakes or enhancements requests. A context switch
to write an email or to participate on a “requirements elicitation meeting” is no longer
necessary with the same frequency as in current processes. Application vendors and
engineering teams have concrete and formal instantiations of the problems that should
be addressed by the new features. Distributed, asynchronous requirement specification
becomes more efficient and less error-prone. Risks resulting from human factors are
minimized. Applications grow faster and more purposeful.
Maalej et al. identify Context awareness as one of the major building blocks for the feedback driven development process. Context awareness comprises the aggregation of the domain instrumentation and the interpretation of automatically collected context information. In order to collect context information from users at runtime, their domain and work environment need to be instrumented. If users perform most of their activities on computer systems (most knowledge workers such as engineers, online traders or people working in a back-office belong to this group) this is feasible. Context elicitation can be realized by using application frameworks such as the Eclipse Rich Client Platform or the Apple Cocoa Framework. These frameworks offer common interfaces for user interface events, independently from the application. It is possible to collect user interaction and sense particular context properties such as how long the user has been reading or editing a particular document or which information she was searching for. Another alternative is to use program tracing tools (such as dtrace\(^1\) or systemTap\(^2\)) to sense context such as particular actions or relations from the system. Apart from the efficiency of the instrumentation, one major challenge is to aggregate, filter and interpret the context and render it in both human but also machine readable form (for the requirements engineers and for systematic analysis and comparison with other contexts). Maalej et al. propose statistical means and semantic technology to solve this issue.

Communication and feedback Context information plays an important role in the process of feedback driven development. As stated before, to collect this context information (feedback) at runtime from the users, their environment needs to be monitored.

However there are several types of communication between the users of a system and the according developers or maintainers. Maalej et al. use two dimensions to impose a structure on these types of communication. They distinguish between pull communication, if the feedback is pulled from the user and push communication, if it is pushed by the user. Furthermore, user input might be explicit, if the user has the intent to provide the input, or implicit, if the user unintentionally provides input information. Figure 4.2 shows these different kinds of input.

If seen in the context of software maintenance, implicit feedback via pull communication is mostly realized using some kind of software monitoring system. Those kind of systems, which can be seen as Usage Data collectors, are described in further detail in section 4.2. The perpetual beta paradigm [55] can be seen as a software maintenance technique as well as a requirements engineering technique. Here, engineers continuously modify the product and roll out new versions, including incremental feature updates. Typically, software providers such as Google roll out new features for a limited set of users and

\(^{1}\)http://www.sun.com/bigadmin/content/dtrace/
\(^{2}\)http://sourceware.org/systemtap/
D2.1: State-of-the-art in monitoring control for remote maintenance

Figure 4.2: Classification of user input [50].

decide upon actual usage data if to include it in the public “main” version. While this model originally emerged with web-based applications, it is also increasingly adopted by desktop applications. Software systems such as the web browser Mozilla Firefox or the development environment Eclipse come with a modular plug-in architecture, which triggers automatic updates several times a week.

A common form of push communication and explicit feedback in today’s software maintenance practice are issue and bug reports and enhancement requests (problem and modification reports in the terminology of ISO/IEC 14764) and feature requests.

4.1.2 The Gamma System

With the Gamma system [57, 58, 56] Orso et al. describe an approach and a realization for the continuous improvement of software systems after their deployment. The Gamma system facilitates remote monitoring of deployed software using an approach that exploits the opportunities presented by a software product being used by many users connected through a network. Gamma splits monitoring tasks across different instances of the software, so that partial information can be collected from different users by means of lightweight instrumentation, and integrated to gather the overall monitoring information. This system enables software engineers to perform continuous, minimally intrusive analyses of their software’s behavior, and to use the information thus gathered to improve and evolve their software. According to the authors, the Gamma system can be used to gather information to detect, investigate, and solve problems that occur when the software is in use. The problems include errors, incompatibility with the running environment, security holes, poor performance, poor usability, or failure of the system to satisfy the users’ needs.

Orso et al. identify two main cycles within the maintenance and evolution process supported by the Gamma system (see Figure 4.3):

- **Incremental monitoring.** Software engineers can interact with the software instances to collect information incrementally. Such interaction provides an efficient way of
D2.1: State-of-the-art in monitoring control for remote maintenance

Figure 4.3: Maintenance and evolution process supported by the Gamma system [58].

locating problems in the software. Incremental monitoring lets software engineers investigate problems directly in the field, without having to replicate the user environment in-house, which is often difficult and expensive, and sometimes impossible.

- **Feedback-based evolution.** In the process, software is maintained and evolved based on the results of the continuous monitoring. Because the information gathered through monitoring reflects, and is highly dependent on, the way users use the software, it is more likely for the software evolution to fit users’ needs. For example, features reported as most commonly used will be fixed or improved before features reported as rarely used, thus maximizing users’ satisfaction.

### 4.1.3 Summary

The feedback-based development process defined by Maalej et al. [50] makes continuous user input and feedback a first order concern in requirements engineering processes. Requirements engineering and software maintenance have common principles and overlaps, in particular regarding the interaction between users and software engineers. Software maintenance is more than just reacting to critical system errors (see section 2.2), like requirements engineering it addresses also new features and enhancements. User input and feedback are needed to get a measure for the software quality observed by the users, but also context information is necessary to react on errors and deviating behavior. Remote Software Maintenance makes particular demands on the collection of this information; information collection approaches like monitoring, as well as the establishment and maintenance of information collection mechanisms become first order concerns in this area.

Orso et al. [57, 58, 56] describe a maintenance and evolution process which is supported
D2.1: State-of-the-art in monitoring control for remote maintenance

by a software monitoring system. This process focuses on incremental monitoring and feedback-based evolution and particularly addresses distribution of software maintainer and users. Though in literature no definition of a remote maintenance process could be found, the work by Orso et al. could be regarded as a first approach to structure remote software maintenance activities, even though it focuses on a specific technology. However, user and program input as well as feedback are key concepts in this approach as well.

Summing up, Remote Software Maintenance processes (in addition to classical software maintenance processes) have to deal with the distribution of software systems and their users on the one hand, and software maintainers on the other hand. Information about the running maintained software as well as its users and the according context is crucial to interpret software and user behavior remotely. Pro-actively and iteratively gathering this information can be beneficial, supporting failure prevention as well as reduced information collection overhead. Monitoring is a key enabler for a remote maintenance process, dealing with the collection of this type of context information.

According to these findings, a remote software maintenance process needs to consider the collection of context information as an important activity. Starting with the generalized and simplified software maintenance process defined in section 3.5 we can thus define a general remote software maintenance process by adding the corresponding activity. Using the classification of communication provided by Maalej et al., it is also possible to distinguish different information collection approaches within this process. The following is the general remote software maintenance process we defined based on these findings (see Figure 4.4 for an overview):

1. Collection of context information
   a) Pull communication and implicit feedback (e.g. Monitoring of user context and program interaction)
   b) Push communication and implicit feedback
   c) Push communication and explicit feedback (e.g. Change requests, bug reports)
   d) Pull communication and explicit feedback: (e.g. Clarification requests)

2. Maintenance issue analysis

3. Solution generation

4. Testing

5. Solution delivery
4.2 Software-Fault Monitoring

A goal of runtime software-fault monitoring is to observe software behavior to determine whether it complies with its intended behavior. Monitoring allows one to analyze and recover from detected faults, providing additional defense against catastrophic failure. Although runtime monitoring has been in use for over 35 years, there is renewed interest in its application to fault detection and recovery, largely because of the increasing complexity and ubiquitous nature of software systems.

Delgado et al. [14] provide an excellent summary of influential monitors and offer a common language with which to describe and categorize them. Their work provides a taxonomy resulting in a classification of based on the application and implementation of monitors that are used for software-fault detection, diagnosis, and recovery.

4.2.1 Monitoring Systems

According to Peters “a monitor is a system that observes the behavior of a system and determines if it is consistent with a given specification” [61]. Thus a monitor surveys if given properties from the specification of a software system hold for the current execution. Monitoring can take place either during the runtime of a software system or in a post-facto fashion using a recording of the program behavior during execution.

Delgado et al. [14] propose a generic monitoring model (see Figure 4.5) consisting of
D2.1: State-of-the-art in monitoring control for remote maintenance

two major components: the *monitor* and the *event-handler*. The monitor itself consists of an *observer* which is used to analyze the current state of the monitored software and an *analyzer* that evaluates if asserted properties of the software systems hold in the current state. The event-handler on the other hand is a component that records monitoring results and performs additional actions like forwarding the results to a logging system or notifying the user. Also responding to a violation of the asserted properties in a given state can be done by an event-handler. The behavior of the system is specified in the system’s *requirements*, which are an input artifact for the monitoring component.

According to Guo et al. [26], the generic monitoring model of Delgado et al. is applicable for most runtime monitoring tools and techniques. However, they argue that with the increasing appearance of distributed computing systems, software systems more and more rely on application platforms. Today’s systems often base on certain operating systems and their APIs, they use middleware to exchange messages among different nodes, they run in virtual machines and use large databases to store information. Guo et al. state that monitoring models should consider platform dependencies and response mechanism between monitor and monitored software. They propose another generic monitoring model, considering the components of previous monitoring models, platform dependencies and a response mechanism to control the monitored software (Figure 4.6).
4.2.2 Monitoring Taxonomies

Delgado et al. introduce a taxonomy of runtime software-fault monitoring systems based on the common elements of monitoring systems, namely specification language, monitor and event-handler. Additionally, operational issues such as the type of programs that are targeted by the monitoring system, platform dependencies and level of maturity of the tool are considered. Figure 4.7 shows the runtime monitoring taxonomy. Following is a summary of the main branches of the taxonomy, for detailed information the reader is referred to [14].

- The specification language branch classifies the language that is used to define the monitored properties, the level of abstraction of the specification and the expressiveness of the language.

- The monitor branch distinguishes systems by characteristics of the monitor, including the monitor placement, i.e. where the monitoring code executes, as well as implementation details.

- The branch event-handler comprises the configurability of the event response given by a system, as well as the type of response effect, i.e. to what extent the monitor’s response to a violation can affect program behavior.

- Further, the branch operational issues comprises categories regarding the external environment of the monitor, such as source program type and dependencies.
Figure 4.7: Runtime monitoring taxonomy according to [14].
Guo et al. [26] provide a software runtime monitoring taxonomy based on their model of software runtime monitoring (Figure 4.6). They introduce categories according to the components of this model: Monitored Object, Runtime Monitor, Monitor Access Method, Execution Relationships, and Platform Dependencies. In contrast to Delgado et al. also the monitored system is considered within the taxonomy. An overview about the taxonomy can be found in Figure 4.8.

- The category Monitored Object Features addresses properties of the monitored system, such as performance, programming language, architecture, platform dependencies, and distribution properties.

- Monitoring Access Methods Features is canonically composed of two subcategories, regarding monitoring the system (Monitoring Code Instrument Methods) and performing reactions on the system (Response Mechanisms), i.e. influencing the states and behavior of the monitored system, especially when a violation or exception happens.

- Execution Relationships Features categorize the type of communication between the monitor and the monitored system, including shared variables, inter-process communication, and middleware.

- The category Runtime Monitor Features determine how the monitoring mechanism is implemented, including implementations like algebra, automata, logic, policy rules, and statistics.

- In contrast to the platform dependencies of the monitored object, Platform Dependencies Features classify the dependencies of the monitor, such as operating system, virtual machine, database, and middleware.

### 4.2.3 Existing Systems

There has been research done on how to categorize runtime software-fault monitoring systems (see previous section) and several systems have been surveyed and categorized according to these taxonomies. This section summarizes the findings about these existing systems for runtime software-fault monitoring.

Studies by Delgado et al. and Guo et al. Delgado et al. [14] and Guo et al. [26] surveyed several runtime software-fault monitoring systems and categorized them using the taxonomies they had defined beforehand (see section 4.2.2). This section provides a brief overview of their findings, in particular tabular overviews of the categorization of the monitoring systems as well as quantitative results of their studies.
The study of Delgado et al. comprised the following software-fault monitoring systems: Alamo [36, 37], Annalyzer [49, 48], Anna Consistency Checking System [51], Annotation PreProcessor (APP) [69], BEE++ [7], DB-Rover [15, 17], Dynamic Monitoring with Integrity Constraints [22], Falcon [19], Java with Assertions [1], Java PathExplorer [28], Java Runtime Timing-Constraint Monitor [54], Monitoring and Checking [40, 41], Monitoring-Oriented Programming [9], Noninterference Monitoring Architecture [75, 76], Program Monitoring and Measuring System [45], Runtime Assertion Checker for the Java Modeling Language [12], ReqMon [67, 68], Sentry System [13], and Temporal Rover [17]. Tables 4.1, 4.2 and 4.3 show an overview of the categorization of these systems as presented in [14].

Guo et al. [26] surveyed 40 existing monitoring techniques and tools and analyzed their features according to the taxonomy they propose (see section 4.2.2). The analyzed monitoring systems comprise systems that were already analyzed and categorized by Delgado et al., in other words there exists a common subset of monitoring systems that were surveyed by both researcher groups. The full list of software-fault monitoring systems analyzed by Guo et al. can be found in [26]. Table 4.4 shows the subset of monitoring techniques and tools, they presented in their work, namely Argus[20], ComPol[81], GAMMA [57], MaC[40, 41], and JPaX [28].

**Other monitoring systems** Additionally to the software monitoring systems studied above there are monitoring platforms that allow monitoring of heterogeneous software and hardware components using a single platform and collecting the necessary input data.
### Table 4.1: Monitors classified according to the specification language branch [14].

<table>
<thead>
<tr>
<th>TOOL</th>
<th>LANGUAGE TYPE</th>
<th>ABS. LEVEL</th>
<th>PROPERTY TYPE</th>
<th>MONITORING DIRECTIVES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ALGEBRA</td>
<td>AUTOMATA</td>
<td>DOMAIN</td>
<td>IMPL</td>
</tr>
<tr>
<td>ALAMO</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>ANNALYZER</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>ANNA CCS</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>APP</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>BEE++</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>DB ROVER</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>DynaMICs</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>FALCON</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>JASS</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>JPAX</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>JRTM</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>MAC</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>MoP</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>NON-INTER</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>PMMS</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>RAC</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>RegMon</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SENTRY</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>T ROVER</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
### Table 4.2: Monitors classified according to the monitoring branch [14].

<table>
<thead>
<tr>
<th>TOOL</th>
<th>MONITORING POINTS</th>
<th>PLACEMENT</th>
<th>IMPLEMENTATION</th>
<th>PLATFORM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MANUAL</td>
<td>AUTOMATIC</td>
<td>IN-LINE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AUTOMATIC, DYNAMIC</td>
<td></td>
<td>OFF-LINE SYNCHRONOUS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OFF-LINE ASYMMETRIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SINGLE PROCESSOR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MULTIPLE PROGRAMM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MULTIPLE PROCESS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SOFTWARE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HARDWARE</td>
<td></td>
</tr>
<tr>
<td>ALAMO</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ANALYZER</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ANNA CCS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>APP</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>BEE++</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DB ROVER</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DYNAMICS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>FALCON</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>JASS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>JPAIX</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>JRTM</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>MAC</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>MOP</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>NON-INTER</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>PMMS</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>RAC</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>REGMON</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>SENTRY</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>T ROVER</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Table 4.3: Monitors classified according to the event-handler and operational issues branches [14].
for activities such as observation of components or detection of bottlenecks and failures. Examples for such systems are IBM Tivoli Monitoring and HP Insight Remote Support Advanced.

**Hewlett Packard Insight Remote Support Advanced** Hewlett Packard Insight Remote Support Advanced (Insight RSA) (see e.g. [59]) is a product developed by HP which aims to enable monitoring and remote support of hardware resources such as servers and storage devices through the Internet. Insight RSA allows local monitoring and secure transmission of the collected data to HP’s backend support system HP Support, analysis of collected data at HP backend (system health checks, availability reports, ...) and secure remote access and management of managed systems and resources. Insight RSA contains three components:

1. The Remote Hardware Event Manager, which monitors the status of the hardware components and generate notifications when predefined situations are detected. These notifications are then sent to HP servers for review and support action.

2. The Remote Data Collection and Proactive Services, which collects system information that will be sent and assessed by HP Support. Assessments include health-checks, current patching levels, system audits, and system availability reports. No business data is collected but information such as IP addresses, system details, and system administrator contact information can be collected.

3. The Remote Device Access, which allows remote access and management of the system or single resources.

Figure 4.9 details the Insight RSA architecture. It shows that several types of devices can be monitored and the collected information is sent to various Insight RSA components using different protocols. Finally, some components such as the Remote Support Software Manager and the Remote Support Client will send information to HP Backend Services for possible support.

**IBM Tivoli Monitoring** IBM Tivoli Monitoring (see e.g. [25]) is part of the system management platform IBM Tivoli Management Framework (TMF) from IBM and allows to monitor the IT infrastructure and collect data about current status and performance of single components. Components of several categories like platforms (Windows, Unix, ...), applications (SAP, .NET, ...), business integration (web services, ...), databases (DB2, Oracle, ...) and messaging and collaboration (Lotus Domino, Exchange, ...) can be monitored.

Figure 4.10 illustrates different IBM Tivoli Monitoring components. The Tivoli Enterprise Monitoring server is the key component on which every other component relies on
D2.1: State-of-the-art in monitoring control for remote maintenance

Figure 4.9: HP Insight Remote Support Advanced Architecture.

Figure 4.10: IBM Tivoli Monitoring Components.
and collects information gathered by monitoring agents and tracks predefined situations and policies. Tivoli Enterprise Monitoring Agents are in charge of gathering data and provide them to the monitoring server. There are different types of agents depending on the component they are monitoring, e.g. operating system or application, and IBM Tivoli Monitoring provides ways to build own monitoring agents and customize them to specific own components. The Tivoli Enterprise Portal server and the Tivoli Enterprise Client that connects to it are responsible for displaying, visualizing and analyzing of the monitored data.

**Summary of existing systems** There is a strong interest in the field of dynamic software-fault monitoring today, which results mainly from software systems getting bigger, increasingly complex and thus harder to verify. Even though there has been work done on developing a general model of runtime software-fault monitoring, the various existing systems diverge in several ways. There are taxonomies for runtime software-fault monitors, providing categories that allow a classification of the different systems. Multiple surveys on existing systems allow also for quantitative statements about the properties of current runtime software-fault monitoring systems.

According to Delgado et al. most of the systems use logic or a high-level to very-high-level language to specify software properties. All approaches they investigated are able to specify and monitor implementation-level properties, while fewer can capture domain-level and design-level properties. In particular monitoring of user behavior is not part of today’s software-fault monitoring systems. Although runtime software-fault monitoring by definition covers only errors of the monitored systems, it should be taken into account, how the systems are used by their respective users. Usability errors and problems that result from users misunderstanding the systems cannot be identified in today’s software-fault monitoring systems. The capability of error classification is hence limited to errors resulting from bugs in the software or the direct system context.

Guo et al. found that only 40% of the systems are capable of monitoring non-distributed software systems. Further, they discovered that only 46% of runtime software-fault monitoring systems have built-in response mechanisms, and are thus bidirectionally associated with the monitored system. More than half of today’s existing runtime software-fault systems are monitoring systems only, but in other words this means that nearly half of the monitoring systems have also response capabilities to influence the monitored software. About 20% of current runtime software-fault monitoring systems depend on a virtual machine, while none of the systems surveyed in [26] according to Guo et al. relies on the operating system. The latter statement seems quite surprising, and in fact two of the systems, namely Alamo and BEE++ depend on the operating system according to Delgado et al. But still this means that only about 10% of the systems depend on the
Figure 4.11: Quantitative information about the relation between features and runtime software-fault monitors according to [26].

operating system. Figure 4.11 summarizes the quantitative findings by Guo et al.

### 4.2.4 Summary and Shortcomings

Software-fault monitors are systems that deal with recognizing abnormal behavior in a system. There exist several categories of software-fault monitors and we found two different taxonomies for the classification of the different systems and approaches. However, the systems described deal only with the notion of errors in terms of system behavior that does not conform to a specified constraint. In the surveyed systems there is no monitoring of the user and no collection of further contextual information that could lead to additional helpful insights about the cause of errors. Also the specification of the constraints is based on local information and not on a broader context. Furthermore, the specification of constraints and system behavior using logic or high-level languages provides an approach which is not easily or automatically adaptable to a changing software or user context.

Process models like [50] propose continuous user monitoring and feedback to proactively provide information that may also be used to predict and prevent errors before they occur. In particular these approaches claim the collection of contextual information in order to achieve context awareness.
4.3 Fault Replication

Fault replication means repeating a software error, a crash or a bug, in order to be able to correct the program involved. When debugging a program, maintainers are frequently doing fault replication locally: they run the same faulty program repeatedly to identify what the problem is. Once the program is shipped to users, who use it remotely and in circumstances one can’t control, debugging becomes more complex. In that situation, one requires fault replication. Repeating remote bugs has two basic requirements: being able to gather all the information that describes the error and transmitting it to the developers. In order to gather the relevant information, maintainers have to modify the execution environment so that all the choices made during a particular program execution are recorded. The relevant information is then transmitted to the software developers via the Internet. At the software development site, the execution is the replayed using a special debugger. If the program is corrected, the process could even be extended to the next step which would be correcting the program being executed on users’ devices.

4.4 Self-Maintainable Systems

As mentioned in Section 2.2, maintenance processes aim to ensure costs-effective software support. As most of this cost is induced by the time spent by operators/engineers to perform maintenance tasks, automating this process could reduce costs considerably. The automated process may be viewed as being part of the system to be maintained itself, hence leading to the notion of self-adaptive [71] or autonomic systems [38, 29]. Self-adaptive systems possess the ability to adapt to changes or situations. In particular, they can modify their configurations or behavior in order to optimize or repair themselves. Although the term is not defined in literature, we call such systems self-maintainable systems due to the fact that they can perform important maintenance tasks autonomously.

Autonomic systems can be seen as another term referring to self-adaptive systems. Historically, autonomic systems referred to biologically inspired systems, e.g. the human body immune system. Software systems are increasingly becoming complex and are dominated by the need to understand the varying environments and frequently changing user needs, and thus leading to increased overhead of maintaining and supporting them. These problems motivate the need of autonomic software paradigm with so called self-* properties such as self-healing, self-configuring, self-managing, self-optimizing and so on. Among several important aspects of the self-* systems, self-adaptivity is considered traversal and can be explained as:

“Self-adaptive systems can configure and reconfigure themselves, augment their functionality, continually optimize themselves, protect themselves, and
To achieve the so-called self-* capability, the system needs to continuously monitor its execution environment, input parameters, produced output, and detect requirements violations and to aid the system to switch to (predefined) variants of its behaviours that allows restoring requirements satisfaction. These different steps are usually presented as the autonomic feedback loop. Figure 4.12 recalls the one presented in [16].

Monitoring software execution can be a very complex and consuming task. Therefore it can lead to a state where the resource overhead shadows the self-* capabilities. To overcome the limitation of monitoring the environment in its entirety Jon et al. [78] suggest to use human commentary, to build up a more clear picture of the operating context. This approach propose intuitive way of taking user input in consideration but does not seems very feasible except in few scenarios. Although it puts forward the importance of user feedback for localization of the problem scope. Some recent work such as [64] introduce techniques which aim to ensure effective and robust data network self-monitoring. Where as a large community of researchers are focusing on providing assistance on engineering adaptive requirements [66] which encompasses the notion of variability in it while elaborating either a functional or a quality attribute of the software system. These

Figure 4.12: The autonomic feedback loop.

*recover themselves, while keeping most of their complexity hidden from the user and administrator.*” [10]
requirements can include monitoring specification that takes into account the variability in the operational context, evaluation criteria and alternative software behavior to be adopted at runtime to fulfill the intended user’s goal. To achieve the mentioned objective they utilize domain ontologies along with goal oriented domain model linked together to capture essential aspects of requirement for adaptive systems.

In autonomic computing, observations are typically based on log files that capture information about failures, performance statistics or system health metrics. In many cases self-adaptive systems monitor purely quantitative data and trigger adaptations policies when a specific threshold is reached. Although by the nature of self-* systems they should be capable of dealing with uncertainty in their model of environment; considering it as a core requirement. But a fixed environment model, even with evolving capabilities is limited to sets of parameters that were decided at design-time and thus could not capture key aspects of the environment in order to realize when adaptation is necessary.

Finally some biology inspired concepts have been studied and shown that some systems can be designed in order to increase fault-tolerance and self-maintainability. In [43], systemic computing concepts are considered as well as system components duplication in order to restore proper system behaviors in situation where the system could crash and in order to facilitate system self-maintenance.

One important part of self-maintainable systems is the ability to self-heal, i.e. the capability of a software system to deal with bugs [70]. General approaches on self-healing are described in e.g. [70, 39]. These approaches follow the general autonomic feedback loop (see Figure 4.12) and rely on the ability for the system to monitor itself, analyse the collected data and decide on what actions to be performed in order to overcome any detected issue. Work such as [65, 73] propose more specific approaches. In [65], the author consider changes in the system architecture where changes and validation are performed using architectural models. In [73] the authors consider rescue points, which correspond to locations in the application code for handling failures. This rescue points make it possible to rollback whenever some issue is detected and some possible fix has been identified. Once rolled back, the system is executed with previously selected error handling facilities, characterized as being relevant to the observed issue.
5 Existing Systems

This section describes current existing systems for remote software maintenance and compares their features according to different categories like the supported remote maintenance process activities.

5.1 Microsoft Fix It

The Microsoft Fix it\(^1\) solution is inspired by the fact that when most of the users face any problem they tend to search for a suitable solution on the web in the hope that someone else might have had the same problem and would have tried to resolve it. Solutions are mostly available in discussion or support forums, described in a series of documents stating the exact steps to fix the problem. Some of these articles are quite handy and easy to understand even for a novice. But most often it's the case that the user just wants a quick solution without having any idea on how to follow the mentioned steps in the forum. Often, the solutions for any given problem are quite difficult to be applied manually, and requires expert knowledge of core functionality of the system.

The “Fix it” initiative by Microsoft proposes to create an easy solution for any given problem and allows the user to overcome the problem simply by clicking on the “Fix it” button provided in the discussion forum rather then following the steps manually to fix the given problem. This solution tends to focus on fixing the problem discussed at the web-page as the “Fix it” button is given. Each "Fix it" is a separate, small download of a Windows installer package that when run, automates the manual steps spelled out in the appropriate KB document. To extend “Fix it” solution Microsoft propose to go through its existing problem solving tips repository and create the scripts for the problem which are encountered frequently by the users. Further they propose to provide a desktop version of “Fix it” solution center which will automatically diagnose the complete operating system and then try to either “Find and Fix” a problem or “Find and Notify” it to user or experts. This application will analyze the current contextual information extracted by diagnosing the system with the help of various troubleshooters and then provide preventive measures for the problem which might occur in future.

\(^1\)support.microsoft.com/fixit
“Fix it” propose to provide a system specific solution for any given problem by analyzing and collecting all the software and hardware information for a given system. This contextual information is automatically attached in a support request being forwarded to experts for examination, which in turn helps in getting the basic information about the operational context.

5.2 Rainbow Framework

Rainbow [11] is a framework for engineering systems with run-time self-adaptive capabilities to monitor, detect, decide and act on opportunities for system improvement. It utilizes an external control mechanism which localizes the concern of problem detection and resolution in separable modules, providing the possibility to analyze, modify, extend and reuse across different systems [21]. The framework provides the capability to handle a wide variety of systems along with framework to monitor a target system and its execution environment. Since the systems have different architectural styles, properties of interest, and dynamic modification mechanisms, it provides the possibility to tailor architectural control model and modification strategies to be system specific.

Rainbow uses an abstract architectural model to reason about the system’s dynamic behaviour, evaluate a model for constraint violations, and in case of any anomaly trigger adaptations on the running system. This abstract architectural model provides a global perspective of the system and exposes important system-level properties and integrity constraints. Probes and gauges (see Figure 5.1) monitor the target systems and update the abstract architectural model managed by Model Manager. The constraint evaluator evaluates the updated model to identify any constraint violations, in case of a violation, the Adaptation Engine is triggered to handle it, which in turn identifies a suitable strategy to solve the problem at hand. The selected strategy is executed by the Adaptation
Executor with the help of system level effectors on the running system.

## 5.3 Gamma System

As we described in 4.1.2, the Gamma system[^2] is a remote software maintenance system proposed and prototypical implemented by Orso et al. [57, 56, 58].

The Gamma system facilitates remote monitoring of a deployed software using *Software tomography* (see Figure 5.2), an approach that exploits the opportunities presented by a software product being executed by users connected through a network, like many of today’s systems. The system is able to split monitoring tasks across different instances of the software, so that partial information can be collected from different users resulting in a light-weight instrumentation of the monitored software. This partial information is then integrated to gather the overall monitoring information. This system enables software engineers to perform continuous, minimally intrusive analyses of their software’s behavior, and to use the information thus gathered to improve and evolve their software.

The latter is realized by *Onsite code modification/update* which enables maintainers not only to deploy updates to the software instance being maintained, but also to determine and alter the level of monitoring by changing the software instrumentation.

According to the authors, the Gamma system can be used to gather information to detect, investigate, and solve problems that occur when the software is in use. The problems include errors, incompatibility with the running environment, security holes, poor performance, poor usability, or failure of the system to satisfy the users’ needs.

[^2]: According to the paper, a patent is pending for the Gamma system. We could not establish whether the patent is still pending, rejected or approved.
5.4 Summary

The existing systems for remote software maintenance differ in various ways. They recognize different types of errors, faults are specified and stored in different ways. Further, the systems support different activities of the remote software maintenance process.

Table 5.1 shows an overview of the commonalities and differences of the surveyed systems. Notably none of the surveyed systems comprises fault replication, while all of them collect a certain amount of context information to support the maintenance issue analysis. Though frameworks like Rainbow and Gamma would allow for the support of fault replication by providing the required information to maintainers, it is not clear if the collected information would suffice to replicate the users’ environment. The Rainbow framework allows the development of self-maintainable systems using effectors to alternate the monitored software. On the other hand, Microsoft’s Fix It is able to change application and operating system configurations and provide patches for faulty behaving software. The monitoring techniques of existing systems vary from operating system monitors to instrumented software.
<table>
<thead>
<tr>
<th>Recognizable Error Types</th>
<th>Microsoft Fix It</th>
<th>Rainbow</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Configuration</td>
<td>(a) General errors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) Incompatibilities</td>
<td>(b) Usability errors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) Security</td>
<td>(c) Incompatibilities</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d) Performance</td>
<td>(d) Security</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e) System crashes</td>
<td>(e) Performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(f) Resource related errors</td>
<td>(f) Wrong functionality</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(g) Missing features</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error Specification</th>
<th>Knowledge base</th>
<th>Constraint Specification</th>
<th>Constraint Specification</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Supported Maintenance Activities</th>
<th>Collection of context information, Maintenance issue analysis, Solution Deployment</th>
<th>Collection of context information, Maintenance issue analysis, Solution generation</th>
<th>Collection of context information, Maintenance issue analysis, Solution Deployment</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Monitoring Approach</th>
<th>Operating System Monitors</th>
<th>Solution Deployment, Runtime Environment Monitors</th>
<th>Software Tomography</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Fault Replication</th>
<th>Application Configuration Monitor</th>
<th>Unknown</th>
<th>Not included</th>
</tr>
</thead>
</table>

| On site code modification | Not included | Not included | Yes |

Table 5.1: Comparison of existing remote software maintenance systems.
6 Conclusion

This report summarizes the most important definitions, models and processes of software maintenance. It surveys, in particular, remote software maintenance concepts and systems. Surprisingly there are no common standards for remote software maintenance processes, but some approaches that propose in particular the continuous monitoring of the maintained software system. Consequently this report introduces and summarizes runtime software-fault monitoring systems and taxonomies, fault replication and self-maintainable systems.

The result of the survey of remote software maintenance is the identification of several research islands: Runtime software-fault monitoring considers only faults that are specified and tested as constraints; no further contextual information is collected and evaluated. Fault replication does not make use of further contextual information as well, though this may lead to easier replication scenarios. Also self-maintainable systems are modeled as closed components that do not make use of further context information. All these theories could benefit from incorporating context awareness and from an interdisciplinary research line.

The term self-maintainable system is not defined yet, but can best be established via autonomous systems. Autonomous systems, however, are described independently from software maintenance concepts and processes. Further there are only single research intersections between autonomous systems and software monitoring. Approaches like the Gamma System [57] and the Rainbow framework [21] lie in this intersection. However, there is no common theory on how to best establish a self-maintainable system from those points of view: software maintenance, autonomous systems, and software monitoring.

Since there is no standard for remote software maintenance processes and models, the existing systems are not based on common knowledge and research but are rather isolated individual solutions. However there exist some relevant proposals for processes strongly related to remote maintenance processes like the feedback-based development process [50] in requirements engineering.

Remote software maintenance approaches will have to deal with the following challenges in order to be successful:

- **Privacy and Security.** Privacy concerns arise during information collection, where sensitive and confidential information may be sent from the user site to the mainte-
nance site. Security concerns involve both maintainers and users, and are related to the possibility of an attacker tampering with the communication between the two parties.

- **Performance.** In particular the monitoring of a software system may have an impact on the performance. Thus, depending on the platform type monitoring frequency and amount have to be considered carefully. Approaches like [57] already deal with this issue.

- **Filtering.** Monitoring and continuously collecting user feedback produces high amounts of data which has to be filtered in order to be usable. Filtering, however, should happen already during monitoring to prevent information overload.

- **Intrusiveness.** The primary role of users is to use a system and not to develop and maintain it. Thus, a major challenge is not being intrusive to users. It is crucial that the applications which integrate a remote maintenance approach “assess” if the situation is appropriate to recommend users to share their input or to perform time and performance consuming monitoring tasks.

- **Comprehensiveness.** Providing developers with a clear picture of the maintenance issues that arise. Information from one or several users should be integrated and related to the corresponding problems, so that developers have a more encompassing understanding of the status of the execution platform that led to the issue at hand.
Bibliography


D2.1: State-of-the-art in monitoring control for remote maintenance


[18] Eclipse. All Eclipse Foundation Members.


D2.1: State-of-the-art in monitoring control for remote maintenance


D2.1: State-of-the-art in monitoring control for remote maintenance


D2.1: State-of-the-art in monitoring control for remote maintenance


D2.1: State-of-the-art in monitoring control for remote maintenance

[75] Tsai, J., Fang, K.-Y., Chen, H.-Y., and Bi, Y.-D. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering* 16, 8 (1990), 897–916.


