



UNIVERSITY of LIMERICK
OLLSCOIL LUIMNIGH

Applying Case-Based Reasoning for Building Autonomic Service-Oriented Systems

by Hervé Weitz
0834629@student.ul.ie

M.Sc Software Engineering Dissertation

Supervised by Muhammad Ali Babar

Department of Computer Science & Information Systems
University of Limerick, Ireland

September 2009

Acknowledgment

With the best thanks to Muhammad Ali Babar from University of Limerick, for supervising this dissertation. Thanks to Philip Johnson and Austen Ito from University of Hawaii for the great support throughout this research on conceptual ideas and implementation techniques. Working with them was a pleasure and a great experience for lifetime. Thanks also to the Google Summer of Code program for a great experience and insight into open source development and very interesting student projects.

Abstract

Service-oriented computing is considered as a successful approach building large-scale software systems, spanning the internet, and globally improving software reuse. Service-oriented architectures are complex and hard to maintain. A service may run on many machines, and single machines may host many services. The concept of distributed composition of services hides a huge amount of complexity in the management of the service-oriented architecture. Users have to deal with complex configuration of services to achieve functional and quality requirements, thus the complexity of the system requires a lot of administrator-interference. Despite the effort of the administrator, the configuration may not be good enough. It is hard for an administrator to monitor individual services and the service-oriented system to determine if the system is running optimal. Therefore a growing trend for autonomic service-oriented systems has emerged. In mid-october 2001, IBM released a manifesto that the main obstacle to further progress in the IT industry is a looming software complexity crisis. The manifesto claimed that the difficulty of managing today's computer systems goes well beyond the administration of individual software environments. Computing system's complexity appears to be approaching the limits of human capability, and there will be no way to make timely, decisive responses to the rapid stream of changing and conflicting. This dissertation discusses autonomic computing in service-oriented computing. We present a framework that builds the foundation for self-healing, self-reconfiguration, self-optimization and self-protecting service-oriented systems. We apply and implement the framework to Hackystat¹, an Open Source Software developed at University of Hawaii. Furthermore we discuss the role of service-oriented computing in autonomic computing, which plays a fundamental role for the relationship between autonomic elements. At the end, we achieved to provide a global overview in the domain of autonomic and service-oriented computing and how to combine them in bidirectional ways. We implemented an open source framework called, Hackystat Service Manager², for achieving an autonomic service-oriented architecture in Hackystat in the scope of Google Summer of Code³, which can be evolved and evaluated or adapted to any other service-oriented system.

¹<http://www.hackystat.org>

²<http://code.google.com/p/hackystat-service-manager/>

³<http://code.google.com/soc/>

Contents

List of Figures	iii
List of Tables	v
Chapter 1. Introduction	1
1.1. Overview	1
1.2. Terminology	4
1.3. Symbols	7
Chapter 2. Background	9
2.1. Software architecture	9
2.2. SOA	10
2.3. Introduction to HackyStat	15
Chapter 3. Autonomic and Service-Oriented Computing	29
3.1. Overview	29
3.2. Related Work	36
3.3. Service Composition and Coordination	39
3.4. Case-Based Reasoning	41
3.5. Monitor, Analysis, Plan and Execute (MAPE)	46
3.6. Autonomic Service Manager	47
Chapter 4. The Hackystat Service Manager Project	51
4.1. Overview	51
4.2. Architecture of Hackystat Service Manager	53
4.3. Agents for Monitoring and Execution Layer	54

4.4. Autonomic Service Manager Layer	56
4.5. Hackystat SOA Layer	67
Chapter 5. Conclusion	69
Bibliography	71

List of Figures

2.1 Hackystat Web-application: ProjectBrowser	16
2.2 HackyStat SOA, Components and Dependencies	20
2.3 Hackystat and Twitter	23
3.1 Structure of an autonomic element (from Kephart and Chess 2003)	31
3.2 Basic schemes of service composition	39
3.3 CBR-cycle	43
3.4 Autonomic MAPE-cycle	46
3.5 Framework for achieving automic service-oriented systems	48
4.1 Hackystat Service Manager (HSM)	51
4.2 The Hackystat Service Manager Architecture	53

List of Tables

1.1 Symbols used in this dissertation	7
4.1 REST API Specification of the Sensorbase Agent	55
4.2 REST API Specification of the TickerTape Agent	55
4.3 Hackystat Sensorbase CBR features	62
4.4 Hackystat DailyProjectData CBR features	63
4.5 Hackystat Telemetry CBR features	63
4.6 Hackystat ProjectBrowser CBR features	64
4.7 Hackystat TickerTape CBR features	64

CHAPTER 1

Introduction

1.1. Overview

Service-oriented Architectures, SOA, is considered as a successful approach to build large-scale software systems, spanning the Internet, and globally improving software reuse (Erl 2005). Service-orientation is the concept of building software components that are platform independent and implement a specific business logic. The business logic is accessible only through a provided interface. The interface defines which service the component is offering. In terms of service-oriented computing and service-oriented architecture, a service provides a specific autonomous functionality (Gorton 2006). The service can be discovered, bound and used by any other software (Gorton 2006). Services can be composed, and they can be hosted by many machines, and one machine can host many services (Li et al. 2005.).

This concept of distributed composition of services hides a huge amount of complexity in the management of these services. Users have to deal with complex configuration of services to achieve functional and quality requirements, thus the complexity of the system requires a lot of human-interference (Papazoglou and Van den Heuvel 2007). Service-oriented computing and SOA increases reusability and enables normal software engineering, by building blocks. The complexity of service-oriented systems is the system itself, since services are generally black boxes and autonomous. Software systems are increasingly constructed by composing multiple applications which leads to the need for self-management of the system (Sadjadi and McKinley 2005).

This dissertation addresses the management of service-oriented architectures and considers taking the autonomy of the services to a higher level: the service-oriented system itself should become autonomic.

Autonomic computing, as defined by IBM (Horn 2001), is the concept of a self-managed computing system with a minimum requirement of human intervention. The goal of autonomic computing is to create smart systems that are able to take best decisions. They are developed in a way, that human-intervention is not needed anymore for tasks that have a limited complexity. Talking about systems without any human-interference would require the system to have consciousness, which is a philosophical aspect and far beyond this dissertation.

A service-oriented architecture is composed of autonomous services which, in general, cannot be influenced because they may belong to different organizations. Sometimes a service has to be manually reconfigured, if accessible. If the service is not accessible in any form, it has to be manually replaced by another service providing the same functionality. These are time and cost consuming tasks, carried out by developers and administrators of the system. Thus, there is a need in software engineering for autonomic self-managing service-oriented systems. It became an essential research domain in software engineering resides mostly in the field of autonomous computing and service-oriented computing.

In this dissertation the core problems, different concepts and proposed solutions for achieving autonomic service-oriented systems are presented. The aim of this research is to create a concrete concept, composed by ideas and other different concepts studied during the research. This concept, resulting in a framework for achieving autonomic service-oriented systems, will be presented and implemented for a service-oriented system called Hackystat (Johnson 2001). Hackystat is an open source project developed by academics at University of Hawaii providing a

perfect example for a case study.

The concrete concept is created in a way, that it can be adopted to any service-oriented architecture by generalizing common issues in service-oriented architectures. The concept also addresses its extendibility in different scenarios. Applying theoretical concepts to a real life project achieves that the previous knowledge of the research domain is summarized and extended. Some conflicts of different concepts by different researchers are solved. Furthermore the implementation of the concept can be evaluated and improved beyond the scope of this dissertation.

At the beginning of chapter two, a general overview of software architecture, service-oriented computing and the better known term SOA is provided. A description of the service-oriented paradigm is presented with just one of its many implementations called Representational State Transfer (REST). Knowledge is extended to base level, in order to understand the issues of service-oriented architectures. In the second part the Hackystat project is presented in detail.

In chapter three, autonomic and service-oriented computing, a combined research paradigm is discussed. Problems and solution proposals from related work, for different scenarios and service compositions are presented. The dissertation focuses on Cased-based Reasoning (CBR) to achieve the autonomy of service-oriented systems. The Case-based Reasoning paradigm seems to be very well suited to the domain of service-oriented architectures (Anglano and Montani 2005).

Chapter four describes implementation of the concept for autonomic service-oriented systems in Hackystat.

In chapter five a conclusion will be drawn.

1.2. Terminology

The terminology for SOA is adopted from Thomas Erl's SOA Glossary (Erl 2009), which it is a very consistent glossary with good definitions.

- **Services and Service Oriented Architectures**

- **Service**

- A service is a unit of solution logic to which service-orientation has been applied to a meaningful extent. It is the application of service-orientation design principles that distinguish a unit of logic as a service compared to units of logic that may exist only as objects or components (Erl 2009).

- **Service-orientation**

- Service-orientation is a design paradigm intended for the creation of solution logic units that are individually shaped, so that they can be collectively and repeatedly utilized in support of the realization of a specific set of strategic goals and benefits, associated with SOA and service-oriented computing.

- Solution logic designed in accordance with service-orientation can be qualified with “service-oriented,” and units of service-oriented solution logic are referred to as services. As a design paradigm for distributed computing, service-orientation can be compared to object-orientation (or object-oriented design). Service-orientation, in fact, has many roots in object-orientation and has also been influenced by other industry developments, including EAI, BPM, and Web services (Erl 2009).

- **Service-oriented computing**

- Service-oriented computing is an umbrella term used to represent a new generation distributed computing platform. As such, it encompasses many things, including its own design paradigm and de-

sign principles, design pattern catalogs, pattern languages, a distinct architectural model, and related concepts, technologies, and frameworks. Service-oriented computing builds upon past distributed computing platforms and adds new design layers, governance considerations, and a vast set of preferred implementation technologies (Erl 2009).

– **Service-oriented architecture**

Historically, the term "service-oriented architecture" (or "SOA") has been used so broadly by the media and within vendor marketing literature that it has almost become synonymous with service-oriented computing itself (Erl 2009).

– **Service oriented solution logic / service-oriented system**

Any body of solution logic to which service-orientation has been applied to a meaningful extent is considered "service-oriented." A service represents the most fundamental unit of service-oriented solution logic.

There has been a common misperception that the use of Web services technology within an application constitutes a service-oriented solution. It is through service-orientation design principles that solution logic is shaped so that it supports the realization of the strategic goals and benefits associated with SOA and service-oriented computing (Erl 2009).

– **QoS**

Quality of Service (QoS) defines the non-functional requirements of a service.

• **Autonomic computing:**

Many authors suggest that autonomic computing should exhibit the self-* properties. There is no general agreement on the definition of the proper-

ties among the literature, which finally should be provided to accomplish autonomic computing. Let us define four categories and one overall term to have a common understanding of the self-* properties.

We analyse the process of a child learning to walk. After falling it has to get up on the legs again, a *self-healing* property is required to find the equilibrium again. The self-healing process may need many little *self-reconfiguration* steps. When the child is walking again, it can *self-optimize* from its previous experience to avoid falling again, and increase performance of walking. In an environment of unpredictable events the child learns to *self-protect* from falling under the occurrence of unpredictable events.

The self-reconfiguring approach is the most important one a system has to adopt. A system has to be able to reconfigure itself, for self-healing, self-optimizing and self-protecting. These abilities cannot be achieved without reconfiguration as we defined it. To fulfill the common understanding, a system must exhibit at least one of the properties to be called self-managing.

- **Self-managing**
 - **Self-reconfiguring**
 - **Self-healing**
 - **Self-optimizing**
 - **Self-protecting**

Other terms like, self-adapting, self-configuring e.t.c. can be seen as synonyms of one of the categories. This point of view is defined in more depth later in this dissertation.

In this dissertation we address two research paradigms: autonomic computing and service-oriented computing. As we will see both domains have a tight relationship in a bidirectional way. We define **autonomic and service-oriented computing**

for analysing this relationship. We call **autonomic service-oriented systems** one of the ways of the relationship, gaining autonomy in service-oriented systems. This was the offspring and the main goal of the dissertation, however we discovered that autonomic computing can also benefit from service-oriented computing, we relate to this carefully as service-oriented autonomic computing.

1.3. Symbols

In this section, the symbols and graphics used during the dissertation are explained in Table 1.1.



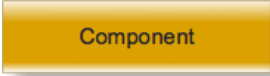
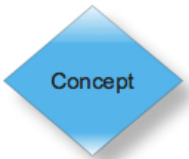

Symbol	Description
	This symbol represents a service in terms of service-oriented computing
	This symbol represents a service in terms of service-oriented computing that belongs to a different organization. From our point of view the service is completely autonomous
	The symbol shows a software component that has no service-oriented attributes
	The symbol indicates that a theoretical concept is implemented in the software components or services it is attached to
	The symbol shows a database. In combination with a service or software component the database symbol indicates that the service or software components dispose an embedded database

TABLE 1.1. Symbols used in this dissertation

For service-oriented systems the OMG has published an UML extension called SoaML, which is not well suited to explain the general concepts used in this dissertation. Therefore this dissertation uses its own symbols, influenced by Thomas Erl's symbols (Erl 2007). The color of symbols is crucial, because they indicate the subtype of a certain type, or differences in the same type.

CHAPTER 2

Background

2.1. Software architecture

Recently the term “software architecture” has gained a lot of attention by researcher and Industry. Technical Architects and Chief Architects are just a few of many job titles emerged in software industry (Gorton 2006). Software architecture is a sub-discipline of software engineering and a hot topic in software engineering research because it reaches from the actual software structure and design, over specific technologies up to the organization structure and business goals (Bass, Clements and Kazman 2003). Software architects have to deal with the requirements specified with the stakeholders up to the possibilities of the recent technology. Thus, software architecture is a broad and complex area with many point of views, and it is hard to share the know-how that software architects gained through their experience. This dissertation adopts the definition by Len Bass, Paul Clements and Rick Kazman from their book “Software Architecture in Practice”:

“The software architecture of a program or computing system is the structure of structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them” (Bass, Clements and Kazman 2003)

This definition could be sufficient requirement to understand service-oriented architectures from a practitioner’s point of view for this dissertation, but it does not reflect the importance and popularity of service-oriented architectures. The definition is not addressing the influence of software architecture on the business

organization structure. It is necessary to understand this influence, and therefore the influence on the Hackystat project and organization structure is analysed. This analysis, accompanied with a deep insight into service-oriented architectures automatically leads to the urge of a certain autonomy in complex software architectures.

2.2. SOA

2.2.1. Introduction. Service oriented architectures and Web services are the latest step in the development of application integration middleware. They provide a basic principle for internet-scaled distributed applications (Gorton 2006). Gorton also states that the technology precipitates the end of technology dependent middlewares because all major vendors are finally agreeing on a single rich set of technology standards and protocols for application integration and distributed computing (Gorton 2006). Service oriented computing and service-oriented architectures are a continuum from older concepts of distributed computing technologies and architectures (Bell 2008) (Gorton 2006) (Erl 2005). Just like J2EE middleware lets Java applications call methods offered by J2EE components, service oriented computing lets platform-independent applications invoke functionality provided by other platform-independent applications through interoperable services.

A service oriented architecture (SOA), is thus, a software architecture where a software component, or a group of software components provide a well defined autonomous interoperable service that can be used by any other software. Thus SOA is a software architecture as defined above, in section 2.1. SOA is a theoretical concept for inter-application communication (Gorton 2006), just like other well-known integration technologies like CORBA (Vinoski 1997), J2EE (Singh 2001) or DCOM (Sessions 1997). The major difference, and reason why SOA can be considered to have a higher level of abstraction, is that SOA is a concept not bound to any language nor to any specific technology, but making use of well defined standards

and communication protocols.

In SOA every single service has to be described by well defined standardized specifications, so that an organization is able to make the service public to other parties (Gorton 2006). In SOA services are autonomous. All that a client has to know about a service is which messages it will accept and return. That is the only dependency between client and service. Implementation of a service can be changed without letting the client know, as long as the messages remain backwards compatible.

2.2.2. Principles of SOA and Implementation. As already mentioned, SOA is a theoretical concept and can be implemented in many ways as long as it follows some standards. Web services implement a service-oriented architecture by making their service available over standard internet protocols independent of platforms or programming languages. Web services can be developed as new standalone application or as extension to legacy systems, in order to make their functionality available to other applications. Gorton defines the four basic principles of service oriented architectures as following (Gorton 2006) :

- **Boundaries are explicit**

Services are independent applications, accessing a service implies crossing over boundaries that separate processes, traversing networks and cross-domain user authentication. Each crossed boundary: process, machine and trust, increases complexity and risk of failure and decreases the performance.

- **Services are autonomous**

SOA separates function into distinct units and services (Bell 2008). Services are deployed onto a network where they can be easily integrated into

any application. They can depend on other autonomous services, but not on the applications that make use of themselves. The implementation is not depending on any language or platform. Services can be written in any language and running on any platform. SOAs create software applications out of the composition of loosely coupled autonomous services. As services are autonomous they are responsible for their own functionality and also security.

- **Share Schemas and Contracts, not Implementations**

This principle is the most important one, even though it is quite simple, it is the core idea behind SOA: services are just applications that receive and send messages. Clients and services share only the definitions of these messages, and certainly don't share code or complex objects. To make this work, SOA relies on services publishing their functionality via interfaces, that other applications and services can use to communicate. This concept is also called the contract between client and service. All that a client needs to know about a service is its contract, the schema of sequence of messages the service will accept and return.

“A service contract is comprised of one or more published documents (called service description documents) that express meta information about a service. The fundamental part of a service contract consists of the service description documents that express its technical interface. These form the technical service contract which essentially establishes an API into the functionality offered by the service.” (Erl 2009)

The most common Web service description documents are the WSDL definition, XML schema definition and the WS-Policy. These contracts are used for Web services which are implemented using SOAP, but services

can also be implemented as components and then the technical service contract can be of any technology-specific API, like for example: REST, RPC, DCOM, CORBA or WCF. The service-oriented architectures are independent of specific technologies (Erl 2005).

- **Service Compatibility is Based on Policy**

A client needs to know more than just the contract, it also needs to know the quality requirements like for example, quality of service, security and encryption. These requirements are assessed in the policies. The policies are integrated into the service contract and are a collection of XML statements that let a service define its requirements for issues like security and reliability.

Another important part of service oriented architectures is to find a suitable service. When looking for a service, one can consult UDDI (Universal Description, Discovery and Integration), an XML-based registry for businesses to publish their services on the Internet.

To successfully build and deploy a distributed SOA, there are four primary aspects that need to be addressed (Papazoglou and Heuvel 2007) :

- (1) Service enablement - Each discrete application needs to be exposed as a service.
- (2) Service orchestration - Distributed services need to be configured and orchestrated in a unified and clearly defined distributed process.
- (3) Deployment Emphasis should be shifted from test to the production environment, addressing security, reliability, and scalability concerns.
- (4) Management Services must be audited, maintained and reconfigured. This requires that corresponding changes in processes must be made without

rewriting the services or the underlying application.

2.2.3. Representational State Transfer (REST). One popular implementation of the SOA paradigm, that can be applied to service-oriented systems, is called Representational State Transfer (REST). It is an architectural style for distributed hypermedia systems. Fielding introduced this architectural hybrid style, derived from several network-based patterns (Fielding 2000).

The most salient feature of a REST architecture is that exchange of messages, between services, should be in a simple form. For example, the HTTP protocol is widely used in a REST architecture and the operations are GET, POST, PUT and DELETE. Another feature is that any resource or service should be identifiable by its URI. All this means that, any client that wishes to connect to a service needs only to know the URI of the service, the protocol and the format of the data returned by the service. REST provides a set of architectural constraints that, when applied as a whole, emphasize scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems (Fielding 2000).

A specific REST style architecture such as the web, consists of client and server. Coupling between client and server on a REST architecture is more loose than with other Remote Procedure Call (RPC) architectures because of this insistence of a simplified, well-known interface. Request and response are built around the transfer of "representations" to or from "resources". A resource can be essentially any coherent and meaningful concept that requests could be addressed to. A representation is typically a document, mostly XML format, that captures the current or intended state of a resource. This makes development of services by different teams easier, especially when a project is open source and team members are geographi-

cally dispersed. Also, REST does not need any resource discovery mechanism.

Fielding defines the desirable properties which the REST style addresses (Fielding 2000):

- Performance
- Scalability
- Simplicity
- Modifiability
- Visibility
- Portability
- Reliability

2.3. Introduction to HackyStat

HackyStat¹ is an open source framework for automated collection and analysis of software engineering process and product data. Hackystat was founded in 2001 by Philip Johnson at the University of Hawaii and has gone through eight major architectural revisions during that time. The Hackystat framework was reimplemented as a service oriented architecture (SOA) and is now released in version 8 (Johnson, Zhang, Senin 2009). The simplified idea of Hackystat is to attach sensors to the integrated development environment (IDE) or operating system (OS) of a developer, collecting raw data, any kind of metrics, and send them to the HackyStat server for storage and evaluation. Once sent to the Hackystat server the data is stored in a database, the so called Sensorbase. The Hackystat framework analyses and visualizes the data of a specific project to a developer. The major user interface is a web application providing a visualization of the collected data for a developer (see figure 2.1). The projects belonging to a developer are listed, and for

¹<http://www.hackystat.org>

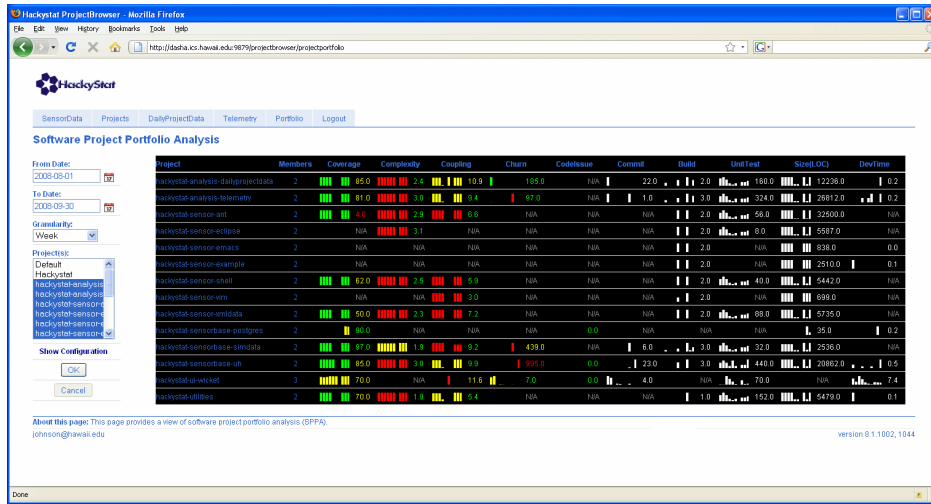


FIGURE 2.1. Hackystat Web-application: ProjectBrowser

each project the data can be shown for a given period.

There are two motivations to focus on projects for the representation of data in Hackystat :

- (1) Usually developers work on more than one project at a time. It is sometimes useful to be able to analyse the work that is done on project A separately from the work done on project B.
- (2) Usually developers work on projects with other developers. In this case, it is sometimes useful to aggregate data collected from one developer's activities with the data collected by other ones. For example, one might be developing software with three other ones and be interested in the total number of unit tests invoked by all members of the project. Or, one developer may have a daily build mechanism that automatically runs tests, computes coverage, and generates size data. It would be nice if that sensor data could be shared by all members of the project, rather than forcing each person to generate it individually.

The project-representation in Hackystat is designed to satisfy these two situations, as well as the combination of the two. Projects also have start and end dates, which allow the analysis of data associated with a single increment of development, for example.

The second user interface is called TickerTape, generating so called Twitter messages on the social networking website Twitter². The aim is to have an automatic way for developers to tell other developers what they are working on. This is interesting if developers are connected via Twitter and are working on the same project. Tickertape has also support for the Nabaztag Rabbit³ service.

Hackystat is in general intended for:

- **Researchers:** Hackystat can be used to support empirical software engineering experimentation, metrics validation, and more long range research initiatives such as collective intelligence.
- **Practitioners:** Hackystat can be used as infrastructure to support professional development, either proprietary or open source, by facilitating the collection and analysis of information useful for quality assurance, project planning, and resource management.
- **Educators:** Hackystat is actively used in software engineering courses at the undergraduate and graduate levels to introduce students to software measurement and empirically guided software project management.

2.3.1. The Vision of Hackystat. Software engineering measurement is a compelling practice in principle. Measurement and observation of developer behaviour and code quality can provide insight into the current state of development, one can make predictions about the future and establish ideas on how to improve

²<http://www.twitter.com>

³<http://www.nabaztag.com>

current development practice and work artifacts (Johnson, Zhang, Senin 2009).

The facts, that not every sensor has to be attached to an IDE, a sensor can be written in and for any developing language (Java, C, C++, Python e.t.c), and that the service oriented architecture enables an extendable independent framework for further functionality, provide an overall vision of a dynamic powerful tool that is open minded and so willing to make advances in software engineering research. The Hackystat framework and its services are employed to use and interpret all kind of data for any kind of analysis needed. For example, the system has been used in Universities to collect and evaluate data of students in order to improve the educational system of software development in computer science (Johnson, Zhang 2009).

Hackystat accomplishes it's REST architectural style through the use of a REST API provided by restlet.org. The client sensors and the sensorbase communicate with one another using this API. One of the main quality attributes that the Hackystat project should deliver is extensibility with regard to new client-side plugin sensors for IDEs and the extensibility of user interfaces. This includes extending existing user interfaces such as the project browser and adding new ones in addition to Twitter, Facebook etc. These extensions are easily delivered to developers. This allows tailoring of Hackystat for their own statistical analyses requirements. The REST API provides low coupling between all of these services by using the standardised HTTP protocol and, hence, makes it easy and available to everyone to extend the system.

The vision of HackyStat is to gather knowledge in the domain of software engineering and software development process and to return this knowledge by supporting practitioners and researchers.

2.3.2. Overview of Hackystat’s Service Oriented Architecture. Hackystat is not a conventional monolithic system but it is a collection of services (see figure 2.2) that are working together to provide developers with data about certain activities or data around their projects. Hackystat has a service-oriented architecture build on REST (see section 2.1.3). Not every service used by the system belongs to the Hackystat project. It also uses external services like Twitter and Google Charts. Twitter is used for communication facilities and Google Charts is used to present the data in visualized forms like charts. Every Service of Hackystat is a standalone project hosted by Google Code. Every service has its own documentation, SVN and mailing groups. The core components of the framework are listed and explained in the following sections.

2.3.3. Hackystat’s Services and Components.

2.3.3.1. *Sensorbase.* Sensorbase is providing the following facilities:

- Receives sensor data transmitted from Hackystat sensors and persists it in a database.
- Receives sensor meta-data (Users, Projects, Sensor Data Types, etc.) transmitted from UI services and persists it in a database.
- Responds to queries from services for information about sensor data and meta-data.

Project: hackystat-sensorbase-uh (Sensorbase)

Description: The Sensorbase service with an underlying database provides a HTTP server, which is responsible for processing GET, PUT, POST, and DELETE requests from other Hackystat web services, an interface to the underlying database persistency layer and a Java Client with high-level methods to interact with the server services via REST.

Interacting: Database

Depending: hackystat-utilities

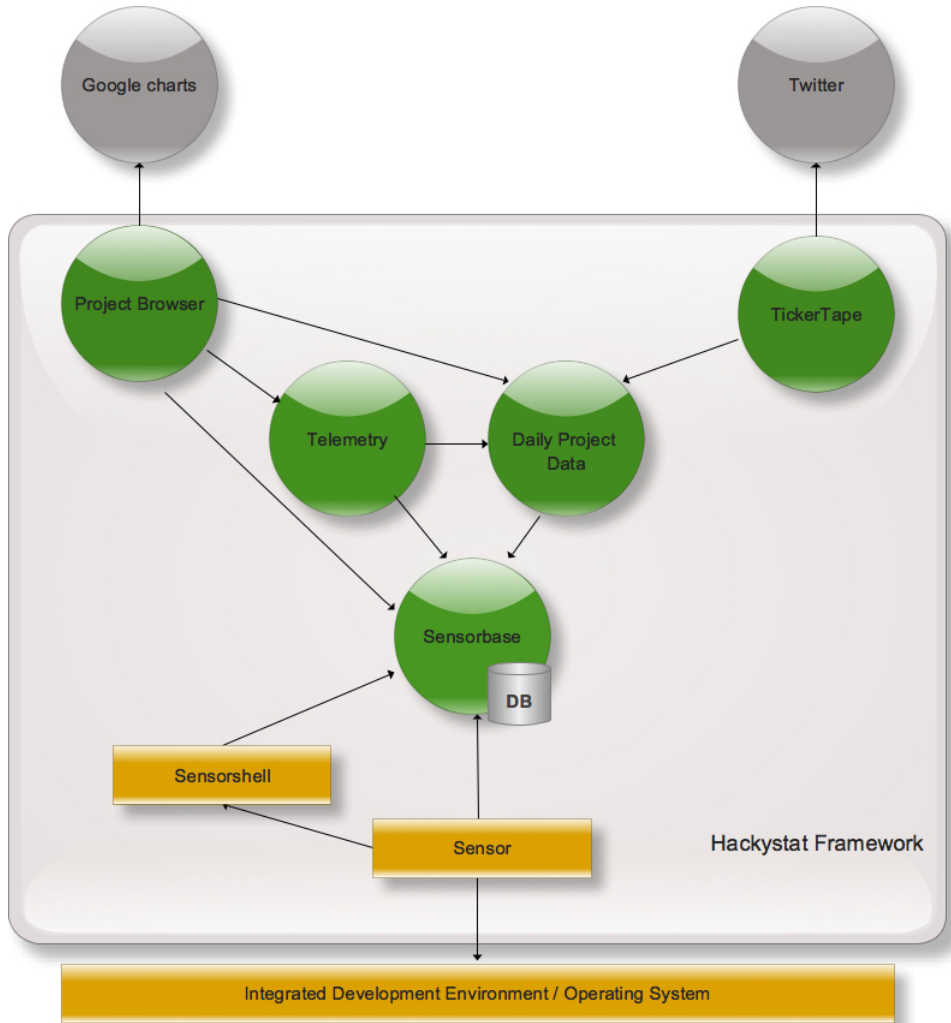


FIGURE 2.2. HackyStat SOA, Components and Dependencies

The Sensorbase server interacts with an Apache Derby database over JDBC. Apache Derby is a Java relational database management system that can be embedded in Java programs and used for online transaction processing. It can be connected over JDBC. Java Database Connectivity (JDBC) is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. The server only interacts with the database over the JDBC driver and a Java client class interacts with the server via the REST API.

2.3.3.2. *Daily Data Project (DPD)*. DPD takes data from the Sensorbase and abstracts it into summaries of project processes and product metrics at the grain size of a day. The DailyProjectData service provides abstractions of the raw sensor data in the Sensorbase repository. Specifically, it creates abstractions of the sensor data associated with a single Project for a single 24 hour period. For example, the "DevTime" abstraction represents the number of minutes that developers were actively interacting with development tools on work products associated with a specific Project. This abstraction is generated by analyzing the DevEvent sensor data and transforming it into a value indicating the number of minutes that the developers were actively working on the project.

Project: `hackystat-dailyprojectdata`

Description: The DailyProjectData project is a service taking data from the sensorbase and abstracts it into summaries of project process and product metrics at the grain size of a day. It provides a server for the services, a front-side cache, interprets metrics and a java client with high-level methods to access the services via REST.

Interacting: Sensorbase

Depending: `hackystat-utilities`, `hackystat-sensorbase-uh`

The service interacts with the Sensorbase service via the REST API and provides its own services.

2.3.3.3. *Telemetry*. Telemetry provides an implementation of Software Project Telemetry suitable for display by Hackystat 8 user interface services. These Telemetry Analysis services utilize data from the Daily Project Data service to produce representations of process and product trends over time.

Project: `hackystat-analysis-telemetry`

Description: This Telemetry Analysis service utilizes data from the DailyProject-Data (DPD) service to produce representations of process and product trends over time. The project provides a server for the services, a prefetch service that speeds up Telemetry Chart display, an analyzer for the DPD data, and a Java client with high-level methods for connecting the services via REST.

Interacting: DailyProjectData Service, Sensorbase service

Depending: hackystat-utilities, hackystat-sensorbase-uh,
hackystat-dailyprojectdata

The service interacts with the DPD Service via REST. It collects DPD data and uses its analyzer classes to interpret them. It also connects the services of Sensorbase over the Java Client SensorbaseClient, which is a high level java class interacting with the Sensorbase services via the REST API. The Telemetry server provides services to retrieve chart data for the Google Chart service.

2.3.3.4. *TickerTape*. TickerTape is a simple user interface that polls Hackystat for changes to a given project, then generates a status report that can be sent to a variety of devices. Similar to a "ticker tape" it can currently connect two devices. The supported devices are the Nabaztag Rabbit (www.nabaztag.com) and Twitter (www.twitter.com) (see figure 2.3). Twitter is a service-oriented web application that allows to publish short messages in its community. The server is configured by an XML file. The XML files also provide the necessary account information for Twitter and/or Nabaztag. The TickerTape is configured to collect data in a given interval about specified projects, identify changes, summarize them and update Twitter or Nabaztag in a given time interval.

Project: `hackystat-ui-tickertape`

Description: Provides a server that polls Hackystat for changes to a given project, then generates a status report that can be sent to a variety of devices.

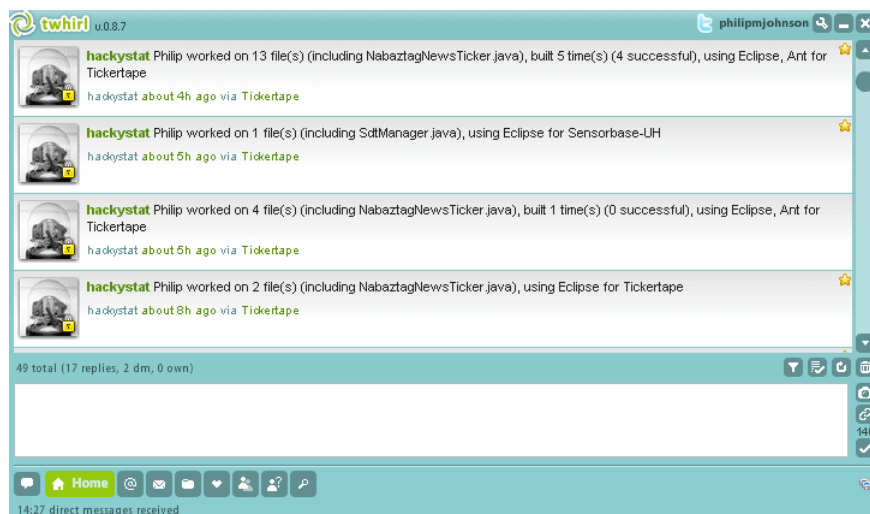


FIGURE 2.3. Hackystat and Twitter

Interacting: Telemetry Service, DailyProjectData Service, Sensorbase service, Google Chart Service

Depending: hackystat-utilities, hackystat-sensorbase-uh, hackystat-dailyprojectdata, hackystat-analysis-telemetry

It is a simple server application, configured by an XML file that is not providing services. It connects the other services, Hackystat and third-parity (example: Twitter) over the REST API. Twitter is also providing a java library with classes to connect the Twitter API. .

2.3.3.5. *Project Browser.* The Project Browser provides a web application interface to Hackystat services. It is the standard user interface for managing projects and providing information on the single projects. One of the goals of the Project-Browser is to simplify the task of creating basic analyses for Hackystat. The initial overhead is learning Apache Wicket, but after that the ProjectBrowser is designed to provide a framework where one can easily implement a new "tab" with a custom analysis.

Project: hackystat-ui-wicket

Description: Provides an web application interface to Hackystat services. It interacts with all the underlying services to present the data to a user. It is built on the Apache Wicket Framework.

Interacting: Telemetry Service, DailyProjectData Service, Sensorbase service, Google Chart Service

Depending: hackystat-utilities, hackystat-sensorbase-uh, hackystat-dailyprojectdata, hackystat-analysis-telemetry

It connects to all the services over the REST API. The chart data from telemetry is sent to the Google Chart service to retrieve chart diagrams, also via the Google's REST API. This project is one of the biggest components of Hackystat and is worth exploring in more detail in the following sections. The Project Browser has a **multi-tier architecture**. It interacts with different services, which interact with other services and the backend is the database.

Wicket is a java web development framework from Apache. There are a copious number of other java web development frameworks in existence and wicket tries to differentiate itself from these on the grounds of being component-based, using generic XHTML/HTML with no added features, and mimics the frameworks of other stateful user interface frameworks such as Sun's Swing, Microsoft's Visual Studio and Borland's Delphi.

An application written for the Wicket framework is a tree of components. In the same way that a conventional desktop framework operates, the components use listeners to react to user events, the events being HTTP requests in this case. The component delegates a listener and associates it with an element on the webpage by using the special HTML attribute 'wicket:id'. This is the only link from the HTML code to the component and the framework handles the rest. Furthermore, due to the fact that the tag attribute 'wicket:id' is a valid HTML attribute and there is

no embedded code, it allows for the HTML to be worked on using ordinary HTML visual editors such as Dreamweaver. Similar to Swing, the Wicket framework considers the 'page' to be the top level container. A page can have components added to it and some of these components can also be considered as containers. Components can be packaged in JAR or ZIP files for reuse. To make an analogy to the MVC architecture, each wicket component has an associated 'model', which is used to hold the data for that component, the 'view' is the HTML and style sheets associated with that component and the controller is the wicket component which contains the business logic and responds to requests from the 'view'.

2.3.3.6. SensorShell and Sensors. Sensors are independent small applications collecting data and metrics and sending them to the Sensorbase service. There are 21 different sensors. Sensors can be depending on the SensorShell project to simplify its implementation.

The SensorShell can be used by Java-based Hackystat clients to simplify collection and transmission of sensor data to a Hackystat Sensorbase service.

The SensorShell provides the following facilities:

- Data transmission. The SensorShell provides an object-oriented interface that makes it easy for Java-based sensors to specify what sensor data should be sent to a Sensorbase server. The SensorShell takes care of the details of emitting the HTTP calls.
- Data buffering. When a SensorData instance is "added" to a SensorShell, it is not immediately sent to the server. Instead, the instance is added to an internal list, and accumulated SensorData instances are sent in a single HTTP request when the SensorShell receives a "send" command.
- Automated sending at user-defined intervals. The SensorShell provides a timer-based subprocess that sends accumulated SensorData at regular intervals (by default, one per minute). Thus, while buffered transmission

is intended to avoid excessive HTTP traffic, automated sending ensures that an excessive amount of data does not accumulate on the client-side.

- **Interactive shell interface.** In addition to a Java API, the SensorShell implements an interactive, string-based command line interface. This enables the SensorShell to be invoked manually, as well as being useful as infrastructure for non-Java-based tools that can communicate with sub-processes.
- **Offline data storage.** If the Sensorbase cannot be contacted, the SensorShell can cache data locally until a connection is established at some future point in time.

Project: hackystat-sensor-shell (SensorShell)

Description: Provides a middleware for accumulating and sending notification of sensor data to Hackystat. SensorShell has two modes of interaction: command line and programmatic.

Interacting: Sensorbase

Depending: hackystat-utilities, hackystat-sensor-base

This project can be used by sensors to interact with the Sensorbase service. It can be used as local service to send data to the Hackystat server over the command line. A sensor can execute these command lines or use the Java interfaces to send data to the server. The SensorShell is configured by an XML file. The project is a middleware because it sits in the middle of the client (sensor) and the application server (Sensorbase), it is a three-tier architecture.

Project: hackystat-eclipse-sensor (EclipseSensor)

Description: Provides a middleware for accumulating and sending notification of sensor data to Hackystat. SensorShell has two modes of interaction: command line and programmatic.

Interacting: Sensorshell/Sensorbase or only Sensorbase

Depending: Eclipse API

This project is part of the Eclipse architecture. It is a stand-alone plugin integrated into Eclipse accessing the Eclipse API to collect data and send the data to the Sensorbase via the Sensorshell. It is also the client in a three-tier architecture. If the sensor is connecting to the Sensorbase directly then we are dealing with a Client-Server architecture.

2.3.3.7. *Utilities, SimData and SystemStatus.*

- The Utilities project provides a repository for generic utilities of potential use to multiple Hackystat services.
- The SimData project supports "simulated" Hackystat sensor data. For example, in order to understand a Hackystat analysis technique such as Software Project Telemetry, it helps to see telemetry generated from sensor data that illustrates various canonical kinds of software project trends. Such simulations form "scenarios" of use.
- The SystemStatus project provides tools to support assessment of Hackystat sensors and services. It is a command line tool that assesses the Sensorbase service status and sends an email to a user with its results.

CHAPTER 3

Autonomic and Service-Oriented Computing

3.1. Overview

3.1.1. Autonomic computing. In mid-october 2001, IBM released a manifesto that the main obstacle to further progress in the IT industry is a looming software complexity crisis (Kephart and Chess 2003). The manifesto claimed that the difficulty of managing today's computer systems goes well beyond the administration of individual software environments. The need to integrate several heterogeneous environments into cooperate-wide computing systems, and to extend that beyond company boundaries into the internet, introduces new levels of complexity (Kephart and Chess 2003).

Computing system's complexity appears to be approaching the limits of human capability, and there will be no way to make timely, decisive responses to the rapid stream of changing and conflicting (Kephart and Chess 2003). The only option remaining is autonomic computing, computer systems that can manage themselves given high-level objectives from administrators (Kephart and Chess 2003). When IBM's senior vice president of research, Paul Horn, introduced the term autonomic computing in 2001 he choose it on purpose with a biological connotation: autonomic nervous systems. The autonomic nervous system controls our heart rate and body temperature, thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, functions. Autonomic computing is a big challenge that reaches far beyond a single organization (Kephart and Chess 2003).

Let's redefine the self-* properties we introduced in chapter 1, more precisely from the point of view of autonomic computer as described by IBM (Horn 2001).

- Self-management

“The essence of autonomic computing is self-management, the intent of which is to free system administrators from the details of system operations and maintenance and to provide users with a machine that runs at peak of performance twenty-four hours a day, seven days a week.” Self-managing system have to adjust by facing changing components, workloads, demands, external conditions, and of course, failure.

IBM cites four aspects of self-management in autonomic computing (Kephart and Chess 2003):

- Self-configuration: *“Automated configuration of components and systems follows high-level policies. Rest of the system adjusts automatically and seamlessly”*
- Self-optimization: *“Components and systems continually seek opportunities to improve their own performance and efficiency”*
- Self-healing: *“System automatically detects, diagnoses, and repairs localized software and hardware problems”*
- Self-protecting: *“System automatically defends against malicious attacks or cascading features. It uses early warning to anticipate and prevent systemwide failures”*.

This definition matches the initial definition provided in chapter 1 of this dissertation.

Autonomic systems are interactive collections of autonomic elements, individual system constituents that contain resources and deliver services to humans and other autonomic elements (Kephart and Chess 2003).

An autonomic element will typically consist of one or more managed elements coupled with a single autonomic manager that controls and represents them, as illus-

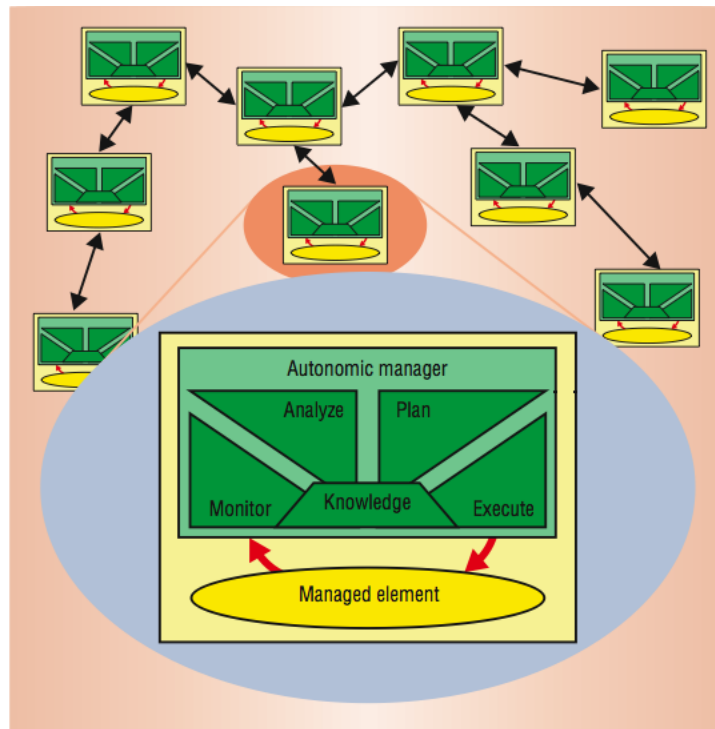


FIGURE 3.1. Structure of an autonomic element (from Kephart and Chess 2003)

trated in figure 3.1. The managed element will essentially be equivalent to what is found in ordinary non-autonomic systems, although it can be adapted to enable the autonomic manager to monitor and control it. The managed element can be a software resource, such as database, a directory service or a web-service but also a hardware resource, such as storage, a CPU or a printer. By monitoring the managed element and its environment, and constructing and executing plans based on analysis of the monitored information, the autonomic manager will relieve humans of the responsibility of directly managing the managed element.

This sounds perfect in theory, however fully autonomic computing is hardly possible to achieve. Fully autonomic computing can be evolved as designers gradually add increasingly sophisticated autonomic managers to existing managed elements (Kephart and Chess 2003).

The distinction between autonomic manager and managed element may become merely conceptual rather than architectural (Kephart and Chess 2003), but for the sake of reusability the distinction should be well defined, for example with interfaces that separate the abstract from its implementation. The boundaries between the system-agnostic and system-specific part within the autonomic manager should be well defined in order to distinct between the autonomic manager and the managed element (Anglano Montani 2005) .

Virtually every aspect of autonomic computing offers significant engineering challenges. The life cycle of an individual autonomic element or of a relationship among autonomic elements reveals several challenges. Some challenges arise in the context of the system as whole, and still more become apparent at the interface between humans and autonomic computing (Kephart and Chess 2003).

The concept for autonomic service-oriented systems, described at the end of this chapter distinguishes well between the autonomic manager and managed elements and the implementation of concept shows the engineering challenges.

Kephart and Chess (Kephart and Chess 2003) discuss the scientific challenges of autonomic computing and state that the success of autonomic computing will hinge on the extent to which theorists can identify universal principals that span the multiple levels at which autonomic systems can exist. This universal principals reach from behavioral abstraction over models to machine learning.

In this dissertation we choose machine learning and its mature domain of Cased-based Reasoning to accomplish our concept for autonomic service-oriented systems, to achieve self-healing, self-reconfiguring, self-optimizing and self-protecting. We combine the autonomic machine learning concept of Case-based Reasoning, presented in section 3.4, with

the **autonomic MAPE-cycle** presented in detail in section 3.5 and add a **service-oriented architecture** to it in order to address the problems and **apply the solutions** presented in the following sections 3.1.2, 3.2 and 3.3.

3.1.2. Service-Oriented Computing . Service-oriented architectures are complex and hard to maintain. In service-oriented computing, a service may run on many machines, and single machines may host many services (Li et al. 2005). The concept of distributed composition of services hides a huge amount of complexity in the management of the services. Users have to deal with complex configuration of services to achieve functional and quality requirements, thus the complexity of the system requires a lot of administrator-interference (Papazoglou and Van den Heuvel 2007). Despite the effort of the administrator, the configuration may not be good enough (Li et al. 2005). It is hard for an administrator to monitor individual services and the service-oriented system to determine if the system is running optimal (Li et. al 2005). Therefore a growing trend for autonomy in service-oriented architectures has emerged. This trend leads, like many other trends of different domains, to a rather specific and isolated research domain called **autonomic computing**. Combining the two domains of **autonomic computing** and **service-oriented computing** results in a domain we can call **autonomic service-oriented computing**. In the research domain of **autonomic service-oriented computing**, we can observe a growing trend of publishing over the last years. Many approaches and solutions, addressing different topics, are proposed. Looking closer at them, they have some common characters with **automic computing**. It is clear that those problems are the same as the problems addressed by **autonomic computing**. Therefore it obvious that we need **autonomic computing** to address these issues.

First, let us remember what we need to successfully build and deploy a distributed SOA. There are four primary aspects that need to be addressed (Papazoglou and Heuvel 2007):

- (1) Service enablement
- (2) Service orchestration
- (3) Deployment
- (4) Management of services

We need to address all four aspects to successfully build service-oriented systems. Some of the aspects are already provided in legacy systems we need to improve. Although these four aspects should not be violated by any proposed solution for autonomic service-oriented systems. Furthermore a proposed concept should improve and automate the last aspect: management of services.

In service-oriented computing, there are several categories of conceptual issues that need to be managed. Some problems, their solution, and the approach to achieve those solutions are presented here in this chapter in a general overview.

The composition of services and the problems it encompasses are conceptually the same as in any other distributed system (Li et al. 2005). Since a service-oriented architecture is a composition of many services, problems of the architecture can depend on only one, or more elements. This can range from a single misconfiguration of a service up to the disfunction of the overall structure of the service-oriented architecture. The problems service composition encompasses, can be adopted from the problems in distributed systems (Li et al. 2005). Christiane Hofmeister (Hofmeister 1998) defines overall areas of possible change in distributed applications. In this research the areas of possible change are adopted to service-oriented systems and called: general types of change in service-oriented computing.

- Service Implementation (Module Implementation): The system's overall structure remains the same. Maybe one or more services are misbehaving and causing problems.
- Service-oriented Architecture (Structure): The system's logical structure (also called either the modular structure or the topology) may change.

The bindings between services, messaging between services, may be altered, new services may be introduced and other services may be removed. Of course, structural change may cause alterations to the implementation of the service-oriented system.

- Service Localization (Geometry): The service-oriented architecture may remain the same, but the mapping of the system's logical structure to the physical service localization - that is the geometry - may change.

These definitions are to be carefully understood. The differences between service implementation, system's logical structure and service localization have to be carefully understood because they can easily fade.

Does bringing up a second instance of a service on the same machine, change the service implementation, the logical structure or a geometrical service location? It does not change the service implementation, neither does it change geometrical location, it changes the service-oriented architecture, the communication and messaging path.

Li et al. (Li et al. 2005) adopted Christiane Hofmeister's concept as well, but in a slightly different way. They argue that for a web-service based system, the first two kinds of problem areas become easy to handle since components, building a service, can be easily and dynamically replaced, because web services separate the interface from its implementation. Thus they focus on geometrical change.

While they are right that service localization addresses the quality of service (QoS), they seem to underestimate the importance of the two first types. Service implementation is also about configuration, optimization and recovery of a single service. Reconfiguring a service may increase performance and diminishes the need for a geometrical change, which can end-up in extra costs. Service-oriented architectures can also be configured and improved. Starting up the same service twice to address load balancing does not need essentially a geometrical change, the binding between

services gets altered and changes the logical structure. Anyway, one major assumption of the first two kinds of configuration change is the accessibility of the services and that they are configurable. First we have to trace back to the initial points of service compositions and the kind of change they may raise, this can be identified in the solution proposed in related work.

3.2. Related Work

In this section, some selected work of different approaches related to this thesis are presented. The papers are classified to point out the capabilities of the concept we will design and which is presented at the end of this chapter. All the listed work influenced our framework in some way. Furthermore we provide a simple general categorization of the problems and solutions.

3.2.1. Service-substitution. BenHalima et. al (BenHalima Jmaiel and Drira 2008) present in their paper a QoS-oriented reconfigurable middleware for self-healing services. The middleware has been achieved in the context of WS-DIAMOND¹ project and covers the whole cycle of adaptation management including monitoring and analysis of QoS values, and substitution-based reconfiguration. The substitution recovery approach is suitable for recovery when a compatible service, with acceptable QoS, exists that can replace a service with QoS misbehavior or QoS degradation.

Hielscher et. al (Hielscher Kazhamiakin, Metzger and Pistore 2008) present in their paper a framework for proactive self-adaptation of service-based applications based on online testing. The service gets monitored online and adaptation requests are sent to the service.

¹<http://wsdiamond.di.unito.it/>

Denaro and Shilling (Denaro and Schilling 2006) head to the direction of service-substitution through service discovery mechanisms to achieve self-adaption in service-oriented architectures.

Grishikashvili et.al (Grishikashvili, Pereira and Taleb-Bendiab 2005) are extending existing works in autonomic computing and service-oriented computing by describing a OSAD (On-demand Service Assembly and Delivery) model which follows the MAPE-cycle to achieve self-healing of distributed services.

Gehler and Heuer (Gehlert and Heuer 2008) propose in their paper called “Towards Goal-driven Self Optimisation of Service Based Applications” a concept for achieving self-optimization by using goal-models. If a service is discovered that suits the goal-model better than the actual one, it gets substituted by the new service. Of course the new service provides the same functionality, but the service fulfills the requirements of the service-oriented system better, e.g has a higher performance than the existing one.

3.2.2. Service-reconfiguration. Tichy and Giese (Tichy and Giese 2004) present in their paper “A self-optimizing run-time architecture for configurable dependability of services” an architecture implemented on top of Jini (FootNOte JINI) that is self-healing and self-optimizing. It manages the instances of services in the service-oriented architecture by restarting if failure and parameter adaptation to improve QoS.

3.2.3. Service-adaption. Di Nitto et. al (Di Nitto, Ghezzi, Metzger, Papazoglou and Phol 2008) provide in their paper “A journey to highly dynamic, self-adaptive service-based applications” a good overview on the topic of autonomic service-oriented systems. They claim that future service-oriented systems will operate in a highly-dynamic world. Technology, regulations, market opportunities and a mixed environment of people, content, and systems will continuously change

and evolve. Service-based applications will thus have to continuously adapt themselves to react to changes in their context and to address changing user requirements. Adaptation must be achieved in an automatic fashion: service-oriented systems should exhibit self-healing, self-optimizing, and self-protecting capabilities. In addition, they should be able to predict problems, such as potential degradation scenarios, future faulty behavior, and deviations from expected behavior, and move towards resolving those issues before they occur. This means that future service-oriented applications will need to become truly proactive (Di Nitto, Ghezzi, Metzger, Papazoglou and Phol 2008).

3.2.4. Service-relocalization. Li et al. (Li, Sun, Qui and Chen 2005) present in their paper “ Self-reconfiguration of service-based systems: A case study for service level agreements and resource optimization” a MAPE-cycle strategy for re-configuring the geometrical location of services within a service-oriented architecture.

3.2.5. Service-oriented and agent-oriented concepts for autonomic computing. Cao et. al (Cao, Wang, Zhang and Li 2004) propose in their paper “A dynamically reconfigurable system based on workflow and service agents” a service-oriented dynamically reconfigurable system framework. They state that service-oriented computing plays a fundamental role in supporting self-management for a software system within an autonomic computing paradigm and that current service technology is far from satisfying. In their framework, service agents can configure their service independent plans into service dependent plans to respond to requests from the environment. A service agent can also optimize its services based on QoS evaluation.

3.2.6. Self-healing through Case-based Reasoning. Anglano and Montani (Anglano Montani 2005) provide the most relevant work for this dissertation in

their paper “Achieving self-healing in autonomic software systems: a case-based reasoning approach”. They address only self-healing, but successfully within service-oriented computing by creating CBR-features and CBR-cases out of the service-oriented system and apply them to Case-based Reasoning. Unlike alternative solutions, service failure gets addressed directly rather than individual component faults, so that unnecessary repair actions are avoided. Moreover, the concept does not require the availability of structure knowledge like, for instance, behavioral models. Thus applying the concept to large-scale, complex systems is more likely feasible.

3.3. Service Composition and Coordination

We analyze the elementary possible service compositions that a service-oriented architecture consists of. There may be just a few scenarios of service composition building the foundation of a service-oriented architecture. They are presented here in a modular perspective, assuming that there are three major basic schemes of service composition. We reflect the general types of changes in service-oriented computing to the schemes from the problem and solution point of view. In a scenario we can identify which possible types of change are sources of problems, and to which type of change a possible solution belongs, presented in the previous section.

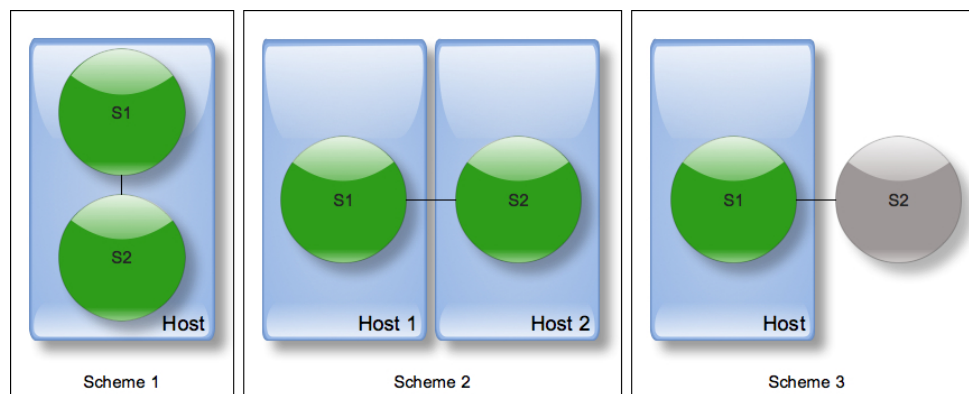


FIGURE 3.2. Basic schemes of service composition

- Scheme 1

Scheme 1 in figure 3.2 shows a simple service composition of two services (S1 and S2) hosted by the same machine. Assuming that S1 is using functionalities of S2, we reflect this scenario on the general types of change in service-oriented computing. Since the service-oriented architecture is not distributed, issues occurring from the geometrical location are not a possible problem source. Problems occurring from the logical structure and service implementation, i.e quality of service issues, are the most probable problem sources. Solutions for problem solving could be of all three types of changes. For example one of the services could be distributed to a different machine, a different geometrical location in order to allocate more resources to both services. QoS-adaption and service-substitution can be a solution too, depending on the problem source.

- Scheme 2

Scheme 2 in figure 3.2 shows a simple scheme of two service (S1 and S2) hosted by two different machines. This is the most common scenario in service-oriented architectures. Problems can occur on all three levels and appropriate solutions can be adopted from all three types of change. Assuming that one host (Host 2) is down, we launch the service on a different machine and change the logical structure. The functionality of the failed service can also be provided, maybe with a better QoS, from another external service located somewhere in the network. QoS-adaption or service recovery can also be possible type of change depending on the problem source.

- Scheme 3

Scheme 3 in figure 3.2 shows a service composition between a service (S1) hosted by a machine and an external service (S2) located somewhere in

the network by a different organization. The difference between scheme 3 and the two previous ones, is that the external service is hosted by a different organization. Reconfiguring or recovering the external service is not possible via our service management framework. The only solution here is a geometrical change to a service providing same functionalities.

A complex service-oriented architecture consists of course of many service compositions that can be of any of these three types combined to a huge web of services. However in order to create a concept for automic service-oriented systems, we have to use the “divide and conquer-principle”to split up the complexity in service-oriented architecture in its smalles units. Here we provided those units and will now present the CBR technology that is able to implement solutions for the different units and their problem source. **Solutions proposed for any problem can be adopted from related work and implemented as CBR case. The goal of using CBR for achieving autonomic service-oriented systems is that CBR can cover a huge problem domain by teaching it the solutions to different problems.**

3.4. Case-Based Reasoning

The first European workshop on Case-Based Reasoning (CBR) took place in November 1993 (Aamodt and Plaza 1994) and is nowadays one of the most successful applied AI technology in recent years (Diaz-Agudo et. al 2007).

CBR is a problem solving paradigm that differs from other major Artificial Intelligence approaches. Instead of relying solely on general knowledge of a problem domain or making associations between problem descriptors and conclusions, CBR is able to utilize the specific knowledge of previously concrete problems called cases (Aamodt and Plaza 1994). CBR is an approach to incremental, sustained learning,

since a new experience is retained, in the so called case-base. Each time a problem has been solved, it is immediately available for future problems. CBR is an automated machine learning approach (Aamodt and Plaza 1994).

CBR transforms unformalized knowledge into formalized knowledge. The cases describe a problem, its solution and the outcome in self-managing autonomic systems. With the help of existing solved cases, new problem cases are solved.

We can compare the CBR approach to human reasoning by remembering solutions to similar problems adopted in the past and by adapting them to current situations (Anglano and Montani 2005). In this comparison, the problem of a fully autonomic systems gets clear. Machines are not human and cannot use qualitative reasoning.

This is why there are two strategies for implementing CBR (Anglano and Montani 2005):

- (1) Quantitative Reasoning, a solution of a similar problem is applied with no adoption. This strategy is called Precedent Case-Based Reasoning
- (2) New or similar cases are solved with an adopted or new solution. A simple form of adaption may be full automated. Although usually some user intervention is needed to perform reuse and adaption. This strategy is called Case-Based Problem Solving and is the most used approach and can be summarized by the following steps called CBR cycle:
 - (a) Retrieve the most similar cases from the case-base
 - (b) Reuse their solutions to solve the new problem
 - (c) Revise the proposed new solution
 - (d) Retain the current case for future problem solving in the case-base.

In CBR, features and cases have to be created for the managed element (Anglano and Montani 2005). For example a system has some attributes that can cause a fault. These attributes are captured with possible values they can hold. A case is a problem scenario where some attributes have one of their possible values and the solution to fix this problem. The core of CBR is an algorithm that compares

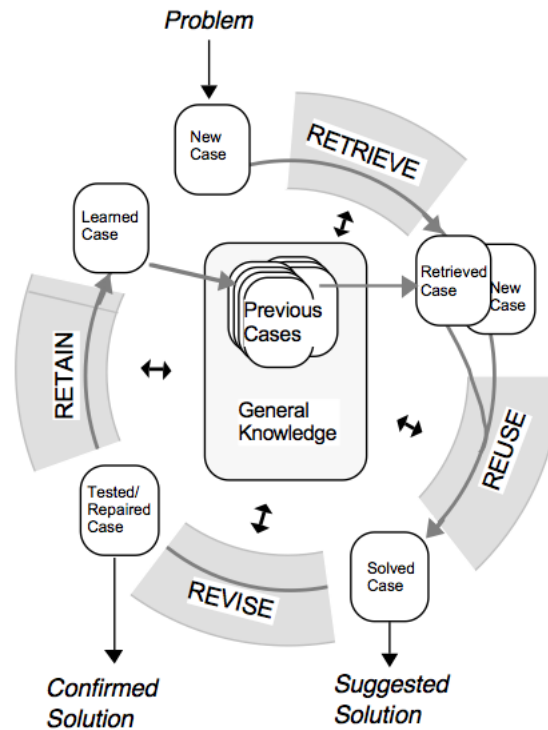


FIGURE 3.3. CBR-cycle

a new case to the existing cases. The solution of the most similar case is used and evaluated. A known set of reconfiguration in the planning module, can adopt the solution. If not successful an intervention of a human can solve the case that is stored for future new problems. The features and cases are compared using similarity functions. Nowadays several CBR tools are available, implementing this functions as well as ontologies for certain domains. Two very good academic CBR tools, we came across in this research are: JCOLIBRi² and myCBR³.

Case-Based Reasoning solves new problems by retrieving previously solved problems and their solutions from a knowledge of cases. CBR uses the k Nearest Neighbor algorithm, with some little differences, to get similar cases from the knowledge

²<http://gaia.fdi.ucm.es/projects/jcolibri/>

³<http://mycbr-project.net/>

and applies a Reuse and Revise stage afterwards (Craw, Wiratunga and Rowe 2006)

A CBR-case normally consist of:

- (1) the problem description – a collection of <feature, value> pairs able to summarize the problem
- (2) the case solution– describing the solution adopted for solving the corresponding problem
- (3) the case outcome – justification of the solution

The problem description is a collection of <feature, value> pairs. In CBR features are attributes describing the managed element. A CBR case describes the state of a managed element by its attributes to a given time. In CBR the Case-base consists of information concerning the failures of the system and their symptoms (problem description), a case solution and the case outcome. First we analyze the given system and identify metrics, i.e CBR features that can cause problems and system failures. The CBR features can take possible values of a certain type. For example for boolean features one can use the so called overlap distance (Wilson and Martinez 1997).

A critical aspect of CBR is the case retrieval, whose computational costs strongly depends on the organization of the case-base (Anglano and Montani 2005). For the different CBR-features similarity functions have to be described for CBR, in order to compare the features and find the overall most similar cases. The most important work in CBR is selecting good CBR features and similarity functions and designing a well defined structure for the case-base. As for AI in general, there are no universal CBR methods suitable for every domain of application. The challenge in CBR as elsewhere is to come up with methods that are suited for problem solving and learning in particular subject domains and for particular application environments (Aamodt and Plaza 1994).In this dissertation we will define CBR features and methods for service-oriented computing and provide a solid body of

knowledge for applying CBR in service-oriented systems.

In research, CBR is still evolving, for example Anglano and Montani state that it is well suited for integration with Rule Based and Model Based Systems (Anglano and Montani 2005). Another issue is automating the acquisition of adaption knowledge, since tasks like design or planning typically require a significant amount of adaption (Craw and Rowe 2001). In adaption knowledge, the adaption of a new solution gets studied through learning algorithms analyzing the problem and solution differences in the case-base. Conversational Case-based Reasoning CCBR is another form of extension of CBR focusing on user-input (Aha, Breslow and Munoz-Avila 2001).

However, CBR has become a mature and established subfield of artificial intelligence (AI), both as means for addressing AI problems and as bases of fielded AI technology. Now that CBR fundamental principals have been established and numerous applications have demonstrated that CBR is a useful technology, many researchers agree on the increasing necessity to formalize this kind of reasoning, define application analysis methodologies, and provide a design and implementation assistance with software engineering tools (Diaz-Agudo et. al 2007).

While the underlying ideas of CBR can be applied consistently across application domains, the specific implementation of the CBR methods, in particular retrieval and similarity functions, is highly customized to the application at hand. Two factors have become critical: the availability of tools to build CBR systems, and the accumulated practical experience of applying CBR techniques to real-world problems (Diaz-Agudo et. al 2007).

In this research, not only identifying CBR-features in service-oriented computing is described, but these features are implemented in a CBR system that is applied to a real-world service-oriented system.

The advantage of CBR is that a solution can be of any type, we revised in the previous sections . With CBR we can accomplish the self-healing, self-reconfiguring, self-optimizing and self-protecting part of a system to achieve its autonomy.

3.5. Monitor, Analysis, Plan and Execute (MAPE)

Most approaches, proposed in the presented literature (section 3.2), make use of the **Monitor Analysis Plan Execute** (MAPE) strategy. We can benefit from it by implementing CBR in the MAPE cycle in order to adopt different solutions and integrate them into our concept. Furthermore it sounds logical to first monitor an object, analyze or diagnose and then create a solution plan which gets executed. This is well illustrated, in a more concrete manner by Cosimo Anglano and Stefania Montani who suggest a case-based reasoning (CBR) approach for achieving self-healing autonomic software systems (Anglano and Montani 2005). In their concept software systems are able to manage themselves in accordance with high-level guidance from humans.

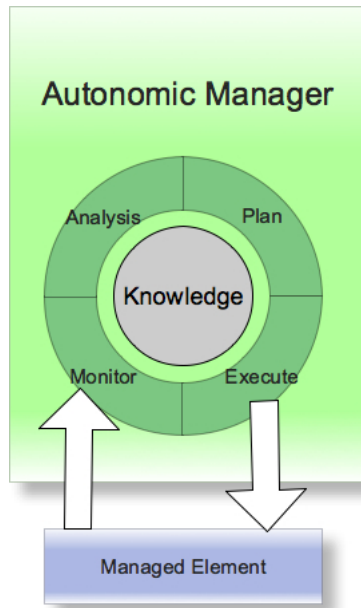


FIGURE 3.4. Autonomic MAPE-cycle

An Autonomic Computing System (ACS) is composed in two entities, the managed element and the manager which is, in the ideal case, full autonomic. The manager is composed of five interacting but self-containing modules. The knowledge element is building the base, learning systems must have a memory. As with our brain, the memory has to be created and changed by different processes. These processes in the ACS Manager can be found in the MAPE strategy, also called autonomic cycle.

3.5.1. The MAPE-cycle in Service-Oriented Computing. We present how this cycle would look like for service-oriented systems.

- Monitor

In the monitoring phase the services are measured for any kind of data.

- Analysis

In the analysis phase the collected metrics to a given time are checked for misbehaviour or inappropriate values. The analysis phases produces a positive result if all of the metrics are correct, and a negative result if any of the metrics is not appropriate.

- Plan

If the analysis phase provides a negative result, a strategic plan has to be created for fixing the misbehaviour, recovering the system and improving the service-oriented architecture

- Execute

In the execution phase the strategic plan gets executed and the misbehaving service or the whole service-oriented architecture gets fixed or improved.

3.6. Autonomic Service Manager

As already mentioned, we combine the concept of Case-based Reasoning, with the autonomic MAPE-cycle and add a service-oriented architecture in order to create a

solid conceptual framework, called Autonomic Service Manager (ASM), for achieving autonomic service-oriented systems, that enables the aspect of self-healing, self-reconfiguring, self-optimizing and self-protecting. In the following steps the MAPE-cycle gets concretized for autonomic service-oriented systems using CBR, a service-oriented and agent-oriented architecture. Figure 3.5 shows the concept of the ASM framework: it has three layers reaching from the SOA, over an *Agent* layer to monitor the services and execute operations on them, to the core Autonomic Manager implementing the MAPE-cycle with Case-based Reasoning. We describe the concept from the MAPE-perspective in the next sections.

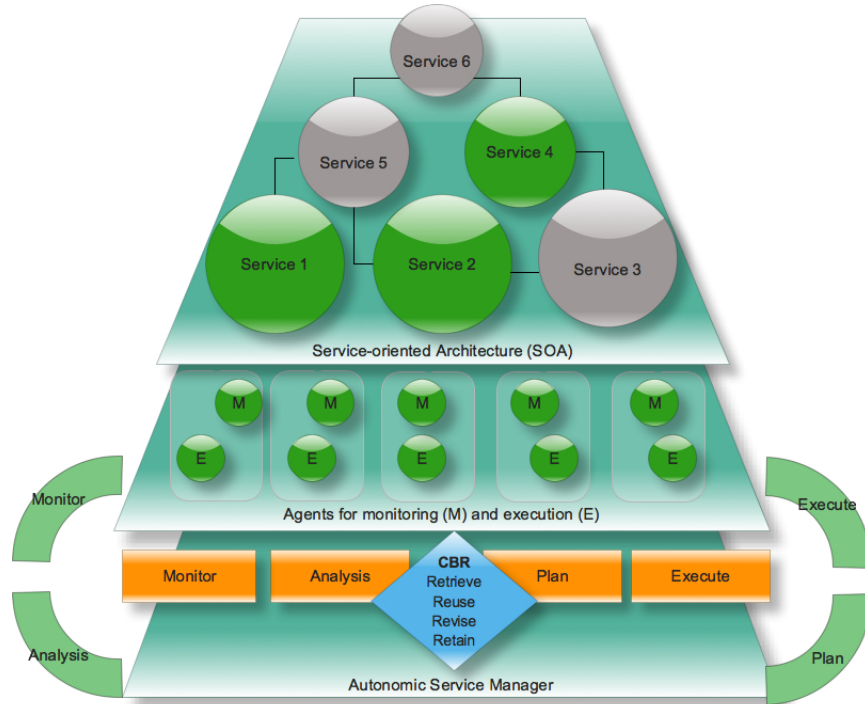


FIGURE 3.5. Framework for achieving automic service-oriented systems

3.6.1. Monitor. The *Monitoring* component makes use of the *Agents for monitoring and execution* layer to monitor the service. This allows distributed services to be monitored locally by its *Agent*. An agent-oriented architecture and service-oriented architecture in autonomic computing is very powerfull and exactly

what we need in this case, in order to also monitor locally the environment, like resources and performance of a distributed service. Combining autonomic computing, agent and service-oriented-computing can be profitable for all, in particular, for the development of autonomic computing systems (Brazier et al. 2009).

In the *Monitoring* component, the services get monitored. The collected metrics are CBR-features that create a case for CBR. The CBR-features have to be carefully chosen and are dependent on the given system, its implementation, and cases we want to create in order to solve different problems. To identify a problem may involve simply noticing its input descriptors, but often, and particularly for knowledge-intensive methods, a more elaborate approach is taken, in which an attempt is made to understand and identify problems within their context (Aamodt and Plaza 1994).

In service-oriented systems we may include the code or name of a service as CBR-feature in order to create system-wide cases. If a problem has been solved for one service it can be reapplied and adopted for any other service. This is an advantage that makes CBR very powerful in service-oriented systems.

3.6.2. Analysis. In the *Analysis* component, the collected CBR-features to a given time are analysed. If the CBR-features show misbehaviour or fall out of range a CBR-case is created out of the metrics describing the problem.

3.6.3. Plan. If a problem occurred, the CBR-case created in the analysis phase, gets used for finding an appropriate solution through Case-based Reasoning:

- **Retrieve**

We retrieve the most similar cases and use the most appropriate one. Usually we just use the best case, without any user-interference. It is possible for example for an administrator to choose the most appropriate

case out of a ranked CBR-case list. Remember, every solved case in the Case-base has a specific solution, which we can extract and re-apply.

- **Reuse**

The extracted solution from the best case is re-applied with or without adoption to fix the current problem case.

- **Revise**

After the solution has been applied, we revise the outcome of applying the solution. One may retain or adopt the solution or one can take a different similar case solution.

- **Retain**

Once a suitable working solution found for the case created in the analysis phase, one should retain the case and its solution. The knowledge gets stored in the Case-base for solving new cases.

3.6.4. Execute. This is actually the same step as Reuse in the CBR-cycle, described in the Plan component above. However the execution component translates the more abstract solution stored in the Case-base to operations for the Agents, in order to execute real operations on the services. An Agent is located on the same machine as the associated service, this allows to control distributed services via an agent-oriented and service-oriented architecture.

CHAPTER 4

The Hackstat Service Manager Project

4.1. Overview

In chapter three, we created the concept of the Autonomic Service Manager. Now we implement this concept for Hackstat.



FIGURE 4.1. Hackstat Service Manager (HSM)

The Hackstat Service Manger (HSM) , is a free and open source software implemented in Java¹. It comes along with a simple GUI providing an overview of the managed services. The current status of each service is provided, and each service can be started or stopped. The innovative idea is that for each service, an autonomic behavior of the service can be enabled, respectively disabled. The autonomic behavior is achieved through a Thread implementing the MAPE-cycle.

¹<http://www.java.com>

The MAPE-cycle and its underlying architecture, functionalities and implementation is the core of the Hackystat Service Manager and can easily be adopted and extended to any other service-oriented software. Furthermore all services can be launched locally at one time, but first we have to set up the appropriate Agents for all services. This is mostly a functionality for demonstration reasons. HSM implements partly the ASM concept and demonstrates its usage and way of how it can be applied in real-life systems. However, due to the time limit of this research, HSM only implements self-healing of the service-oriented system, but provides a flexible extendable framework based on CBR. Introducing new CBR-features and extending the case-base is quite easy and self-optimization, self-reconfiguration and self-protecting can be achieved in the lifetime of the Hackystat-project. The project page² provides a binary distribution of the framework as Download. It also provides a detailed Wiki-directory with Installation Guide, Developer Guide, REST API Specification and of course a SVN repository with the source code, available to everyone. The projects is structured as follows:

- **hsm** - Hackystat Service Manager
- **hsm-agent-sensorbase** - Sensorbase Agent for the Hackystat Sensorbase Service
- **hsm-agent-dpd** - DailyProjectData Agent for the Hackystat DailyProjectData Service
- **hsm-agent-telemetry** - Telemetry Agent for the Hackystat Telemetry Service
- **hsm-agent-projectbrowser** - ProjectBrowser Agent for the Hackystat Web Application
- **hsm.agent-tickertape** - TickerTape Agent for the Hackystat TickerTape component

²<http://code.google.com/p/hackystat-service-manager/>

4.2. Architecture of Hackystat Service Manager

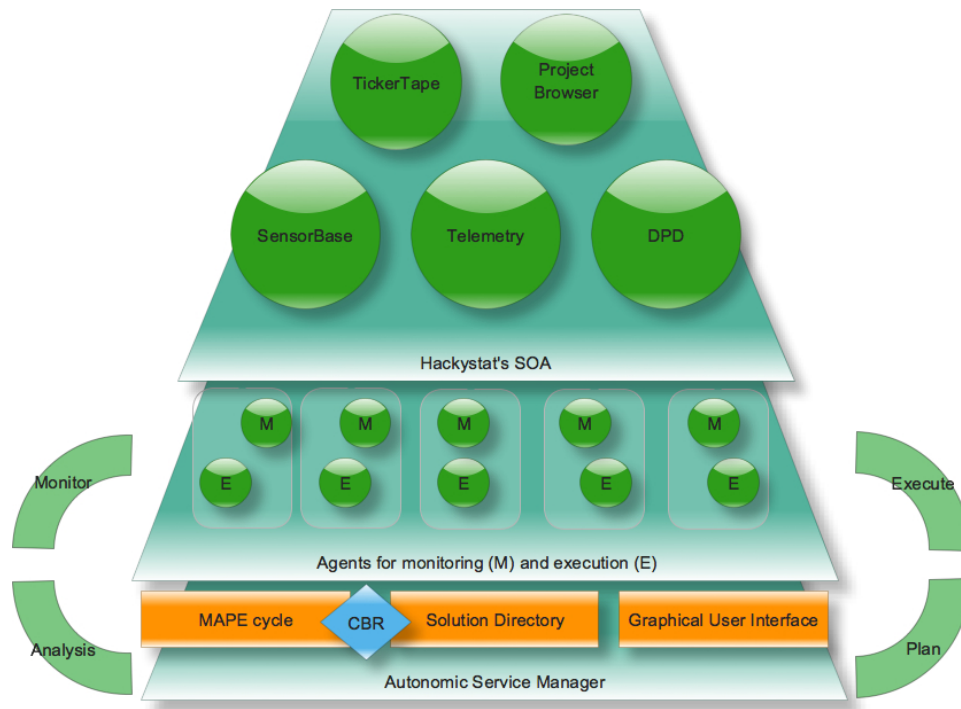


FIGURE 4.2. The Hackystat Service Manager Architecture

The architecture of the Hackystat Service Manager can be divided into three layers. The first one is the Hackystat system itself with its internal services: TickerTape, ProjectBrowser, SensorBase, Telemetry and DailyProjectData (DPD). The second layer is the Agent Layer, each Hackystat service has its own Agent. Agents monitor the Hackystat service and its environment, and execute operations on the service. The third layer is the autonomic service manager itself, implementing the business-logic for achieving the autonomy of the services. A detailed presentation of each layer is provided in the following sections.

4.3. Agents for Monitoring and Execution Layer

We associate for each Hackystat service an Agent. Agents are SOA services again, that are implemented in RESTlet³, an implementation of the REST architectural pattern. Agent functionalities depend on the monitored service, its implementation and its environment. For the conceptual same functionalities, naming conventions are applied. We implemented 5 Agents, an Agent for each service. The implementation of each Agent follows the RESTlet pattern, algorithm 1 shows an implementation for the Sensorbase Agent. We attach the so called Resources implementing the functionalities to the Component Registry which is started under a specific port number.

Algorithm 1 RESTlet Implementation ofr SensorBase Agent

```

package org.hackystat.servicemanager.agents.sensorbase.server;
import org.hackystat.servicemanager.agents.sensorbase.resource.ping.
    PingResource;
import org.hackystat.servicemanager.agents.sensorbase.resource.start.
    StartResource;
import org.hackystat.servicemanager.agents.sensorbase.resource.stop.
    StopResource;
import org.restlet.Application;
import org.restlet.Component;
import org.restlet.Restlet;
import org.restlet.Router;
import org.restlet.data.Protocol;
public class Server extends Application {

    public Restlet createRoot() {
        // Create a Restlet router that routes each call to a
        // new instance of its Resource.
        Router router = new Router(getContext());
        // Defines the routes
        router.attach("/start", StartResource.class);
        router.attach("/stop", StopResource.class);
        router.attach("/ping", PingResource.class);
        return router;}

    public static void main(String[] args) {
        try {
            Component component = new Component();
            component.getServers().add(Protocol.HTTP, ServerConfig.
                SENSORBASE_AGENT_PORT);
            component.getDefaultHost().attach(new Server());
            component.start();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

³<http://www.restlet.org/>

Since Agents are implemented in RESTlet, they provide a REST API for communication over HTTP. We demonstrated the REST API for Sensorbase and TickerTape. For a full list please visit the project homepage.

SensorBase Agent (`hsm-agent-sensorbase`)

METHOD	URI	EFFECT
GET	{host}/start	Starts the Hackystat service
GET	{host}/stop	Stops the Hackystat service
GET	{host}/ping	Pings the Hackystat service
GET	{host}/heapsize	Returns the associated max. Heapsize
PUT	{host}/heapsize	Sets the max. heapsize of a service

TABLE 4.1. REST API Specification of the Sensorbase Agent

Table 4.1 shows an example of the REST API Specification for the Sensorbase Agent. {host} defines URI under which the Agent is running. The column Method advises the type of the HTTP-method.

TickerTape Agent (`hsm-agent-tickertape`)

First of all, we indicate that TickerTape is not really a service but rather service-oriented. It makes us of services, but has no service capabilities itself. Therefore we are not monitoring the software component but execute simple operations on it. We just integrated TickerTape because of completeness reasons.

METHOD	URI	EFFECT
GET	{host}/start	Starts the Hackystat service
GET	{host}/stop	Stops the Hackystat service

TABLE 4.2. REST API Specification of the TickerTape Agent

Table 4.2 shows an example of the REST API Specification for the TickerTape component. The placeholder, {host} defines URI under which the Agent is running.

The column Method advises the type of the HTTP-method.

All agents are following, more or less, the same structure in the REST API, at least for the calls providing the same functionalities. This is because the services should provide the same functionality for the HSM in order to apply monitoring and solutions. This allows a growing knowledge for all services where solution for one service can be applied to any other service.

The REST API methods in detail:

All calls should follow a certain convention in order to let them operate with the HSM independently:

- /start
Returns a String Representation "started" if successful
- /stop
Returns a String Representation "stopped" if successful
- /ping
Returns a String Representation "successful" if successful, otherwise the REST error from the previous call to the Hackystat service or "error" if the call fails at all.
- /heapsize
 - GET - Returns a String Representation with the number of Heap allocated f.ex. "512"
 - PUT - Interpretes a String Representation with the number of Heap allocated f.ex. "128"

4.4. Autonomic Service Manager Layer

The Autonomic Service Manager has three major components: one component implementing the MAPE-cycle for each service in a Java Thread, a solution directory and the Graphical User Interface (GUI).

4.4.1. Autonomic cycle (MAPE-cycle). A superclass provides a template for implementing the MAPE-cycle in a Thread. Since we are working in Java Swing⁴, the Thread is implemented in a SwingWorker class. For each service that should adopt autonomic behavior a class has to be implemented inheriting from the MAPE Java class.

Algorithm 2 Java Class: MAPE.java

```

package org.hackystat.servicemanager.framework.mape;
import javax.swing.SwingWorker;
import org.hackystat.servicemanager.framework.config.HSMConfig;
import jcolibri.cbrcore.CBRQuery;

public class MAPE extends SwingWorker{
    private boolean active= true;

    protected void monitor(){
        // To overwrite }

    protected void analysis(){
        // To overwrite
        //if analysis shows misbhevaieur call plan() }

    protected void plan(CBRQuery query){
        // To Overwrite
        // call execute() when solution has been found }

    protected void execute(int solution) {
        // To overwrite }

    public void stop(){
        active = false; }

    protected Object doInBackground() throws Exception {
        while(active){
            monitor();
            analysis();

            try {
                Thread.sleep(HSMConfig.MONITORING_DELAY);
            }
            catch (InterruptedException e) {}
        }
        return "Done"; }}

```

As we can see the MAPE cycle is derived into little methods, implementing the appropriate MAPE functionality. The MAPE cycle is implemented for each service, and thus data gets collected in the monitoring phase trough the agent associated with a Hackystat service. Since the Agent is a service, the Hackystat service can

⁴<http://www.java.com>

be located everywhere in the network. All the Hackystat Service Manager has to know is the location of the Agent which runs on the same machine as the specific Hackystat Service. When the data is collected for a specific service, its MAPE-class analyses the data and creates a CBR-case of the collected data. The system is designed in a way that collected metrics are CBR-features in the CBR-system, as advised in chapter three.

4.4.1.1. *Planning with Case-based Reasoning.* The Autonomic Manager makes use of the JCOLIBRi⁵ CBR framework for processing CBR functionalities in the planning phase of the MAPE-cycle.

Algorithm 3 Sensorbase Agent MAPE planning phase implementation

```

public class SensorbaseMAPE extends MAPE {

protected void analysis() {
    /*
     * .. Checking CBR-features
     */
    if (misbehaviour) {
        jcolibri.cbrcore.CBRQuery query = new CBRQuery();
        CaseDescription description = new CaseDescription();
        description.setCaseId("Case:" + new Date(System.currentTimeMillis()));
        description.setService(HSMConfig.SENSORBASE_NAME);
        description.setReachable(reachable);
        description.setReqLatency(reqLatency);
        description.setAdeqHeapsize(adeqHeapSize);
        description.setExceHeapsize(execHeapSize);
        description.setDbLatency(dbLatency);
        query.setDescription(description);
        plan(query);
    }

protected void plan(CBRQuery query) {
    SolutionFinder solutionFinder = SolutionFinder.getInstance();
    try {
        solutionFinder.preCycle();
        solutionFinder.cycle(query);
        int solution = solutionFinder.getRetrievedSolution().getSolution();
        solutionFinder.postCycle();
        execute(solution);
    }
    catch (ExecutionException e) {}
}
}

```

JCOLIBRi has been designed as a wide spectrum framework able to support several types of CBR systems from the simple nearest-neighbor approaches based on flat or simple structures to more complex knowledge intensive ones. It also contains

⁵<http://gaia.fdi.ucm.es/projects/jcolibri/>

textual and conversational extensions (Diaz-Agudo et al. 2007). JCOLIBRi is developed by University of Madrid and implemented in Java. It is also very well documented⁶. Algorithm 3 shows an implementation of the analysis and planning phase of the Sensorbase Agent using JCOLIBRi. As we can see in the analysis and planning phase, a `jcolibri.cbrcore.CBRQuery` object gets created and send to the CBR framework.

Algorithm 4 CBR engine - SolutionFinder, Part 1

```

package org.hackystat.servicemanager.framework.mape.cbr;
import jcolibri.casebase.LinealCaseBase;
import jcolibri.cbrapplications.StandardCBRAApplication;
import jcolibri.cbrcore.Attribute;
import jcolibri.cbrcore.CBRCase;
import jcolibri.cbrcore.CBRCaseBase;
import jcolibri.cbrcore.CBRQuery;
import jcolibri.cbrcore.Connector;
import jcolibri.connector.DataBaseConnector;
import jcolibri.method.retrieve.RetrievalResult;
import jcolibri.method.retrieve.NNretrieval.NNConfig;
import jcolibri.method.retrieve.NNretrieval.NNScoringMethod;
import jcolibri.method.retrieve.NNretrieval.similarity.global.Average;
import jcolibri.method.retrieve.NNretrieval.similarity.local.Equal;
import jcolibri.method.retrieve.NNretrieval.similarity.local.Interval;
    import jcolibri.method.retrieve.selection.SelectCases;

public class SolutionFinder implements StandardCBRAApplication {

    private Connector connector;
    private CBRCaseBase caseBase;
    private CaseSolution retrievedSolution;
    private static SolutionFinder instance = null;

    public static SolutionFinder getInstance() {
        if (instance == null)
            instance = new SolutionFinder();
        return instance;
    }

    public void configure() throws ExecutionException {
        // init database
        CaseBaseInizialiser.init();
        // init connectors
        connector = new DataBaseConnector();
        connector.initFromXMLfile(jcolibri.util.FileIO.findFile("config/
            databaseconfig.xml"));
        // Create a Lineal case base for in-memory organization
        caseBase = new LinealCaseBase();
    }

    public CBRCaseBase preCycle() throws ExecutionException {
        caseBase.init(connector);
    }

```

⁶<http://gaia.fdi.ucm.es/projects/jcolibri/jcolibri2/docs.html>

The CBR Framework JCOLIBRi is accessing an embedded Apache Derby⁷ database in the Hackystat Service Manager via the Hibernate⁸ technology. In algorithm 4 and 5 we show the main CBR business logic. It shows four methods:

- `configure()` - Configuration of the CBR Case-base and Database Connector.
- `precycle()` - Initialisation of the Database Connector.
- `cycle()` - Finding similar cases than the one passed in parameters and defining similarity functions for the features.
- `postcycle()` - Closing CBR Case-base and Database Connector.
- `getRetrievedSolution()` - After execution of the first four methods the solution gets saved and is available through this method.

JCOLIBRi is an object oriented CBR framework, i.e that CBR-features in the Case-base are mapped into Java Objects via Hibernate. Therefore we use XML mapping files, a `CaseDescription.class` Java Bean and a `Solution.class` Java Bean. The CBR case consist of the `CaseDescription.class` Bean, for each field in the class we create an `jcolibri.cbrcore.Attribute` object that gets mapped with the similarity function. With the `evaluate Similarity (caseBase.getCases() , query , simConfig)` method we retrieve a collection of the type `Collection<RetrievalResult>` holding the most similar cases with a double value indicating the percentage of similarity. We extract the solution of the best case and store it. This is shown in algorithm 5.

⁷<http://db.apache.org/derby/>

⁸<http://www.hibernate.org/>

Algorithm 5 CBR Engine - SolutionFinder, Part 2

```

public void cycle(CBRQuery query) throws ExecutionException {
    // First configure the NN scoring
    NNConfig simConfig = new NNConfig();
    // Set Global similarity function to average
    simConfig.setDescriptionSimFunction(new Average());
    // Create attributes for Mapping CBR features
    Attribute service = new Attribute("service", CaseDescription.class);
    Attribute reachable = new Attribute("reachable", CaseDescription.class);
    Attribute reqLatency = new Attribute("reqLatency", CaseDescription.class);
    Attribute adeqHeapsize = new Attribute("adeqHeapsize", CaseDescription.class);
    Attribute exceHeapsize = new Attribute("exceHeapsize", CaseDescription.class);
    Attribute dbLatency = new Attribute("dbLatency", CaseDescription.class);
    // Set similarity function for each CBR feature
    simConfig.addMapping(service, new Equal());
    simConfig.setWeight(service, 0.5);
    simConfig.addMapping(reachable, new Equal());
    simConfig.addMapping(reqLatency, new Interval(100));
    simConfig.addMapping(adeqHeapsize, new Interval(4000));
    simConfig.addMapping(exceHeapsize, new Interval(4000));
    simConfig.addMapping(dbLatency, new Interval(100));
    // Execute NN Nearest Neighbor
    Collection<RetrievalResult> eval = NNScoringMethod.evaluateSimilarity(
        caseBase.getCases(), query, simConfig);
    // Select k cases
    eval = SelectCases.selectTopKRR(eval, 3);
    // Print Retrieval
    System.out.println("3_Best_Retrieved_cases, _retain_best:");
    boolean first = true;
    CBRCase bestCase = null;
    for (RetrievalResult nse : eval) {
        System.out.println(nse);
        if (first)
            bestCase = nse.get_case();
        first = false;
    }
    if (bestCase != null) {
        System.out.println("Best_Case:" + bestCase.getSolution());
        retrievedSolution = (CaseSolution) bestCase.getSolution();
        System.out.println("Solution:_ " + retrievedSolution.getSolution());    }}

    public void postCycle() throws ExecutionException {
        // close connectors
        connector.close();
        // close database
        DerbyManager.shutdown();    }

public CaseSolution getRetrievedSolution() {
    return retrievedSolution;    }}

```

4.4.1.2. *CBR-features for Hackystat service and Case-base.* The CBR-features of the Hackystat Services are stored in the Case-base. Therefore we have to analyse the CBR-features we want to integrate into the system and create an appropriate Case-base. We list the tables for each Hackystat service representing the CBR-features of each service. The CBR-features should follow the same structure over the different services. Because of limited time of this dissertation, the features are only implemented partly, but this should provide an idea of where the Case-base should head towards.

Feature	Possible Values	Description	Implementation
reachable	Yes/No	indicates whether the service is reachable over network (Yes) or not (NO)	REST API GET {host}/ping
request latency	None/ Low/ Normal/ High	Depending on caching, this feature indicates if the service responds in a reasonable time	Measuring time of the REST API GET {host}/ping call
adequate heap size	Adequate / Not Adequate	indicates if the JVM heap size of the service exceeds 80%	
excessive heap size	Excessive / Not Excessive	indicates if the services use too much heap size, for example if after N requests heap size is less than 40%	
db latency	None/ Low/ Normal/ High	indicates whether a db operation latency exceeds M milliseconds	

TABLE 4.3. Hackystat Sensorbase CBR features

Feature	Possible Values	Description	Implementation
reachable	Yes/No	indicates whether the service is reachable over network (Yes) or not (NO)	REST API GET {host}/ping
request latency	None/ Low/ Normal/ High	Depending on caching, this feature indicates if the service responds in a reasonable time	Measuring time of the REST API GET {host}/ping call
adequate heap size	Adequate / Not Adequate	indicates if the JVM heap size of the service exceeds 80%	
excessive heap size	Excessive / Not Excessive	indicates if the services use too much heap size, for example if after N requests heap size is less than 40%	

TABLE 4.4. Hackystat DailyProjectData CBR features

Feature	Possible Values	Description	Implementation
reachable	Yes/No	indicates whether the service is reachable over network (Yes) or not (NO)	REST API GET {host}/ping
request latency	None/ Low/ Normal/ High	Depending on caching, this feature indicates if the service responds in a reasonable time	Measuring time of the REST API GET {host}/ping call
adequate heap size	Adequate / Not Adequate	indicates if the JVM heap size of the service exceeds 80%	
excessive heap size	Excessive / Not Excessive	indicates if the services use too much heap size, for example if after N requests heap size is less than 40%	

TABLE 4.5. Hackystat Telemetry CBR features

Feature	Possible Values	Description	Implementation
reachable	Yes/No	indicates whether the service is reachable over network (Yes) or not (NO)	REST API GET {host}/ping
request latency	None/ Low/ Normal/ High	Depending on caching, this feature indicates if the service responds in a reasonable time	Measuring time of the REST API GET {host}/ping call
adequate heap size	Adequate / Not Adequate	indicates if the JVM heap size of the service exceeds 80%	? Some JVM Monitoring tool ?
excessive heap size	Excessive / Not Excessive	indicates if the services use too much heap size, for example if after N requests heap size is less than 40%	? Some JVM Monitoring tool ?

TABLE 4.6. Hackystat ProjectBrowser CBR features

Feature	Possible Values	Description	Implementation
reachable	Yes/No	indicates whether the service is reachable over network (Yes) or not (NO)	REST API GET {host}/ping
request latency	None/ Low/ Normal/ High	Depending on caching, this feature indicates if the service responds in a reasonable time	Measuring time of the REST API GET {host}/ping call
adequate heap size	Adequate / Not Adequate	indicates if the JVM heap size of the service exceeds 80%	
excessive heap size	Excessive / Not Excessive	indicates if the services use too much heap size, for example if after N requests heap size is less than 40%	

TABLE 4.7. Hackystat TickerTape CBR features

Algorithm 6 DerbyManager for creating Case-base

```

package org.hackystat.servicemanager.framework.mape.cbr;

public class DerbyManager {
    public static String embedded_driver = "org.apache.derby.jdbc.
        EmbeddedDriver";
    public static String client_driver = "org.apache.derby.jdbc.ClientDriver";
    private static String dbName = "hsm_case_base";
    public static String embedded_protocol = "jdbc:derby:" + dbName + ";create=
        true";
    public String client_protocol = "jdbc:derby://localhost:1527/" + dbName + "
        ;create=true";

    public static void init() {
        // Load driver
        String driver = embedded_driver;
        String protocol = embedded_protocol;
        Class.forName(driver).newInstance();
        // Connect database
        try {
            Connection conn = DriverManager.getConnection(protocol);
            System.out.println("Connected_to_database_" + dbName);
            try {
                Statement s = conn.createStatement();
                String sql;
                sql = "create_table_case_base_(caseId_vvarchar(15),_service_vvarchar(50),_
                    reachable_integer,_reqLatency_real,_adeqHeapsize_integer,_,_
                    exceHeapSize_integer,_dbLatency_real,_,_solution_integer)";
                s.executeUpdate(sql);
                conn.commit();
                sql = "insert_into_case_base_values('Initial_Case1','+HSMConfig.
                    SENSORBASE_NAME +' ',0,100,512,256,100,1)";
                s.executeUpdate(sql);
                sql = "insert_into_case_base_values('Initial_Case2','+HSMConfig.DPD_NAME
                    +' ',0,100,512,256,0,1)";
                s.executeUpdate(sql);
                sql = "insert_into_case_base_values('Initial_Case3','+HSMConfig.
                    TELEMETRY_NAME +' ',0,100,512,256,0,1)";
                s.executeUpdate(sql);
                sql = "insert_into_case_base_values('Initial_Case4','+HSMConfig.
                    PROJECTBROWSER_NAME +' ',0,0,512,256,0,1)";
                s.executeUpdate(sql);
                sql = "insert_into_case_base_values('Initial_Case5','+HSMConfig.
                    TICKERTAPE_NAME +' ',0,100,512,256,0,1)";
                s.executeUpdate(sql);
                conn.commit();
            }
            catch (SQLException ex) {}

        catch (SQLException e) {
            System.out.println("Database_not_created");
        }

        public static void shutdown() {
            // the shutdown=true attribute shuts down Derby
            DriverManager.getConnection("jdbc:derby:;shutdown=true");
        }
    }
}

```

We now are able to create the Database and its structure with some initial cases in the DerbyManager class, like shown in algorithm 6.

- caseId varchar(15) - Each case must have an id, for the initial cases we name them appropriate, new cases are combined with a timestamp
- service varchar(50) - The name of the service, this is important because through this field the chances that an appropriate case for the same service is found rather than a solution from a different one.
- reachable integer - described in the CBR-feature tables
- reqLatency real - described in the CBR-feature tables
- adeqHeapsize integer - described in the CBR-feature tables
- exceHeapSize integer - described in the CBR-feature tables
- dbLatency real - described in the CBR-feature tables
- solution integer - Solution code for a general re-applicable solution. The solution code is explained in the following section.

Not all services use all of the features, these features are left empty.

4.4.2. Solution Component. The solution for our cases is expressed as code solution number, this number defines a set of actions that are called on the appropriate Agent for executing operation to fix the service. Perfect would be a tool for administrators to select and combine some general scripts for adopting and teaching new solutions. A simple solution for restarting the Sensorbase service is provided in algorithm

Algorithm 7 Solution Directory example for Sensorbase Service

```
protected void execute(int solution) {
System.out.println("execute");
switch (solution) {
case 1:
System.out.println("Exec_Solution_1->Restarting_Service");
CommonSolutions.restartService(HSMConfig.SENSORBASE_ADDRESS, HSMConfig.
SENSORBASE_AGENT_PORT, MainWindow.getInstance().
getLSenorbase_status(), HSMConfig.SENSORBASE_NAME);
break;
default:
System.out.println("Solution_unknown"); } }
```

4.5. Hackystat SOA Layer

As already mentioned Hackystat has a service-oriented architecture. We use the binary distribution of Hackystat for integration with the Hackystat Service Manager. However this binary distribution is not adapted to the HSM. The autonomic manager would benefit from a system that provides more information about itself. This information could be about configuration, and performance. We did not change the Hackytsat system, because we assume that in huge legacy systems, adaptation of the managed system to our needs, would be difficult and related with a huge amount of costs and time. We show that the Autonomic Service Manager can be adapted to any legacy system without changing this system. The ACS is system-agnositic but it would be profitable if the managed system itself would provide as much useful information as possible.

CHAPTER 5

Conclusion

First of all, this dissertation defines a research paradigm called autonomic and service-oriented computing. We define the term autonomic service-oriented systems. None of the reviewed literature uses this term, still this term is essential for describing the application of autonomic computing for achieving autonomy in service-oriented computing. A lot of literature has been published in the domain of autonomic and service-oriented computing, however it is multifaceted and there is no common agreement on terminology. This dissertation tries to provide a clear terminology and overview in the area of autonomic and service-oriented computing.

The research distinguishes consciously between service-oriented computing and service-oriented architectures as suggested by Thomas Erl. It analyses the problems causing the need for autonomy in service-oriented systems. These problems are categorized and mapped to literature addressing the different conceptual problems. A concept has been created, called Autonomic Service Manager (ASM), that is able to address all defined categories of problems. The concept is following a general Monitor Analysis Plan and Execute approach combined with Case-based Reasoning. The Autonomic Service Manager framework is agent-oriented, but also service-oriented. The benefit of both technologies in autonomic computing is discussed. We discuss service-oriented computing in autonomic computing and relate to it as service-oriented autonomic computing. We achieved to provide a global overview in the domain of autonomic and service-oriented computing and how to combine them in bidirectional ways.

The ASM builds the foundation for self-healing, self-reconfiguration, self-optimization and self-protecting service-oriented systems. We apply and implement the framework to Hackystat, an Open Source Software developed at University of Hawaii. We implement ASM as an open source framework for Hackystat called, Hackystat Service Manager, for achieving an autonomic service-oriented architecture in Hackystat. This development was carried out in the scope of Google Summer of Code program allowing a close communication with the founders of the project. We develop a tool, that will be integrated into Hackystat and which can be adapted to any other service-oriented system. Furthermore the integration into a successful open source software like Hackystat allows the system to evolve, and the CBR case-base to grow over time. This can be observed and evaluated.

We mainly extended the work of Anglano and Montani (Anglano and Montani 2005), in a way that solutions from different research can be integrated for achieving self-managing service-oriented systems. The ASM is an essential framework contributing to further research, which should analyse and explore more CBR-features in different SOA technologies for extending the CBR knowledge base. CBR offers the possibility to integrate ontologies, which should be explored for autonomic service-oriented systems.

Bibliography

- [1] Aha, D.W., Breslow, L.A. & Munoz-Avila, H., 2001. Conversational case-based reasoning. *Applied Intelligence*, 14(1), 9–32.
- [2] Aamodt, A. & Plaza, E., 1994. Case-based reasoning. *Proc. MLnet Summer School on Machine Learning and Knowledge Acquisition*, 1–58.
- [3] Anglano, C. & Montani, S., 2005. Achieving self-healing in autonomic software systems: a case-based reasoning approach. In *Proceeding of the 2005 conference on Self-Organization and Autonomic Informatics (I)*. pp. 267–281.
- [4] Bass, L., Clements, P. & Kazman, R., 2003. *Software Architecture in Practice (2nd Edition)* 2nd ed., Addison-Wesley Professional.
- [5] Bell, M., 2008. *Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture*, Wiley.
- [6] Bello-Tomas, J.J., Gonzalez-Calero, P.A. & Diaz-Agudo, B., 2004. jCOLIBRI: An object-oriented framework for building cbr systems. *Lecture notes in computer science*, 32–46.
- [7] BenHalima, R., Jmaiel, M. & Drira, K., A qos-oriented reconfigurable middleware for self-healing web services. In *IEEE International Conference on Web Services (ICWS 2008)*. pp. 104–111.
- [8] Brazier, F.M. et al., 2009. Agents and Service-Oriented Computing for Autonomic Computing: A Research Agenda. *IEEE Internet Computing*, 13(3), 82–87.
- [9] Cao, Wang, Zhang, Li, 2004. A dynamically reconfigurable system based on workflow and service agents. *Engineering Applications of Artificial Intelligence*, 17(7), 771–782.
- [10] Craw, S., Wiratunga, N. & Rowe, R.C., 2006. Learning adaptation knowledge to improve case-based reasoning. *Artificial Intelligence*, 170(16-17), 1175-1192.
- [11] Denaro, G. Schilling, D., 2006. Towards self-adaptive service-oriented architectures. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*. pp. 10–16.
- [12] Di Nitto, E. et al., 2008. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3), 313–341.

- [13] Diaz-Agudo, B. et al., 2007. Building CBR systems with jCOLIBRI. *Science of Computer Programming*.
- [14] Erl, T., 2007. *SOA: Principles of service design*.
- [15] Erl, T., 2005. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*, Prentice Hall PTR.
- [16] Erl, T., 2009. SOA Glossary. Available at: <http://www.soaglossary.com/default.asp> [Accessed July 21, 2009].
- [17] Fielding, R.T., 2000. *Architectural styles and the design of network-based software architectures*. University of California.
- [18] Gehlert, A. & Heuer, A., Towards Goal-Driven Self Optimisation of Service Based Applications. In *1st International Conference of the Future of the Internet of Services (ServiceWave 2008)*. 2008, Springer: Madrid, Spain.
- [19] Gorla, A., 2008. Automatic workarounds as failure recoveries. In *Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium*. pp. 9–12.
- [20] Gorton, I., 2006. *Essential Software Architecture 1st ed.*, Springer.
- [21] Grishikashvili, E., Pereira, R. & Taleb-Bendiab, A., 2005. Performance evaluation for self-healing distributed services. In *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on*.
- [22] Hielscher, Kazhamiakin, R., Metzger, A., Pistore, M., 2008. A framework for proactive self-adaptation of service-based applications based on online testing. In *1st International Conference of the Future of the Internet of Services (ServiceWave 2008)*, Madrid, Spain.
- [23] Hofmeister, C.R., 1998. *Dynamic reconfiguration of distributed applications*.
- [24] Horn, P., 2001. *Autonomic computing: IBM's perspective on the state of information technology*. IBM TJ Watson Labs, NY, 15th October.
- [25] Jarmulak, J., Craw, S. & Rowe, R., 2001. Using case-base data to learn adaptation knowledge for design. In *INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*. pp. 1011–1020.
- [26] Johnson, P.M., 2001. *Project Hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis*. Department of Information and Computer Sciences, University of Hawaii.
- [27] Johnson, P., Zhang, S., Senin, P. (2009), *Experience with Hackystat as a service-oriented architecture*, University of Hawaii, Honolulu
- [28] Johnson, P. Zhang,S. (2009), *We need more coverage, stat! Classroom experience with Software ICU*, Univeristy of Hawaii, Honlulu

- [29] Juan, A., Belén, D. & Pedro, G., 2005. A Distributed CBR Framework through Semantic Web Services. Universidad Complutense de Madrid, Madrid, Spain.
- [30] Kephart, J.O. & Chess, D.M., 2003. The vision of autonomic computing. *Computer*, 41–50.
- [31] Lehman, M.M., 1996. Laws of software evolution revisited. *Lecture notes in computer science*, 1149, 108–124.
- [32] Li, Sun, Qui, Chen, 2005. Self-reconfiguration of service-based systems: A case study for service level agreements and resource optimization. In 2005 IEEE International Conference on Web Services, 2005. ICWS 2005. Proceedings. pp. 266–273.
- [33] Papazoglou, M.P. & van den Heuvel, W.J., 2007. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal The International Journal on Very Large Data Bases*, 16(3), 389–415.
- [34] Sadjadi, S.M. & McKinley, P.K., 2005. Using transparent shaping and web services to support self-management of composite systems.
- [35] Sessions, R., 1997. COM and DCOM: Microsoft's vision for distributed objects, John Wiley & Sons, Inc. New York, NY, USA.
- [36] Singh, I., Johnson, M. & Stearns, B., 2002. Designing enterprise applications with the J2EE platform, Addison-Wesley Professional.
- [37] Tichy, M. & Giese, H., 2004. A self-optimizing run-time architecture for configurable dependability of services. *Lecture notes in computer science*, 25–50.
- [38] Vinoski, S. & Inc, I.T., 1997. CORBA: integrating diverse applications within distributed-heterogeneous environments. *IEEE Communications Magazine*, 35(2), 46–55.
- [39] Wilson, D.R. & Martinez, T.R., 1997. Improved heterogeneous distance functions. Arxiv preprint cs.AI/9701101.